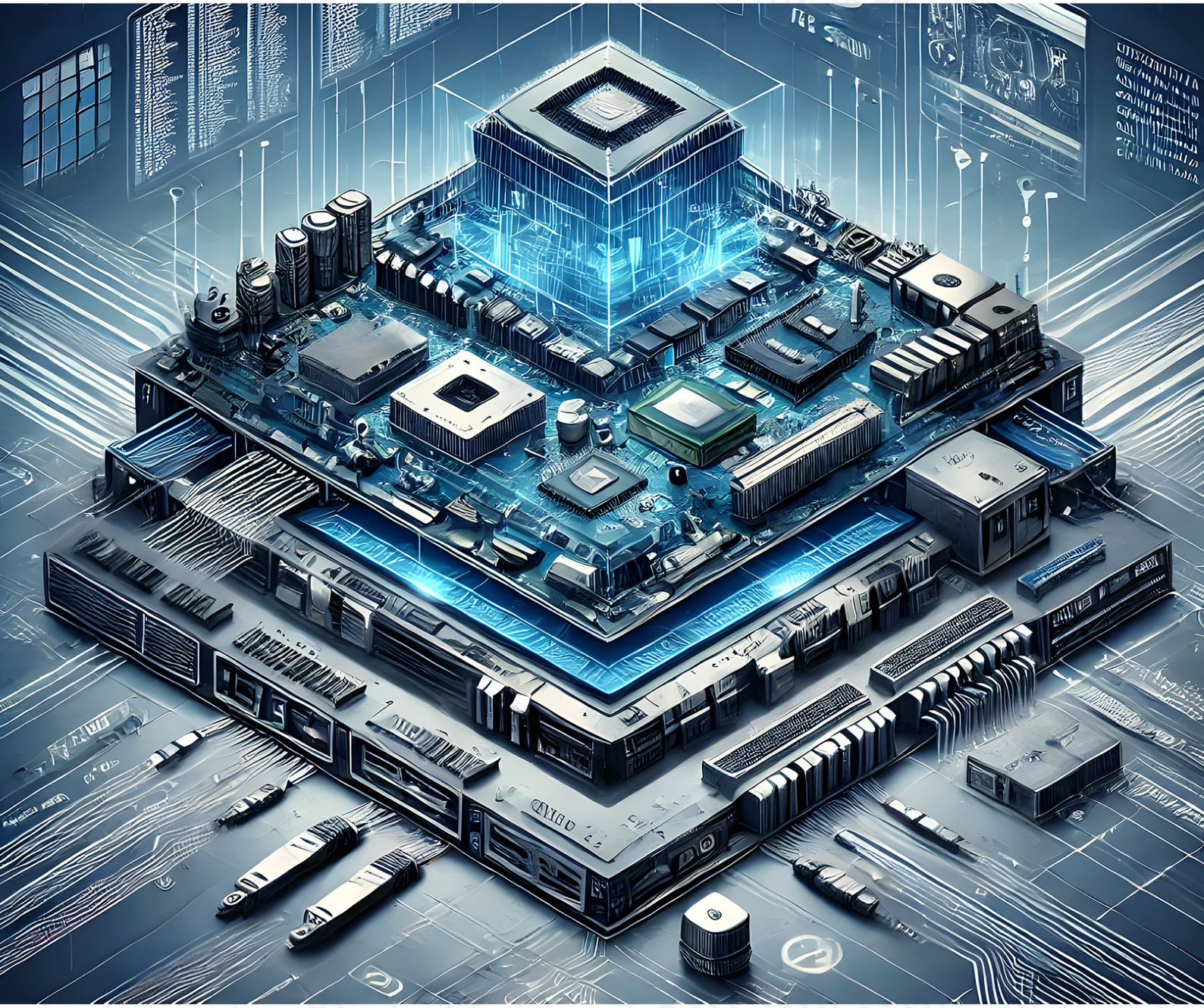
COMPUTER ARCHITECTURE AND OPERATING SYSTEMS

COURSE CODE: B21CA03DC

Bachelor of Computer Application

Discipline Core Course



SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Computer Architecture and Operating Systems

Course Code: B21CA03DC

Semester - II

Discipline Core Course
Undergraduate Programme
Bachelor of Computer Applications
Self Learning Material
(With Model Question Paper Sets)

SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

Computer Architecture and Operating Systems
Course Code: B21CA03DC
Semester - II
Discipline Core Course
BCA

SREENARAYANAGURU OPEN UNIVERSITY

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from Sreenarayanaguru Open University.
Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.
www.sgou.ac.in

ISBN 978-81-978764-5-5

9 788197 876455

DOCUMENTATION

Academic Committee

Dr. Aji S.
P. M. Ameera Mol
Shamly K.
Dr. Jeeva Jose
Dr. Priya R.
Dr. Anil Kumar
Sreekanth M. S.
Dr.Vishnukumar S.
Joseph Deril K. S.
Dr. Bindu N.
Dr. Ajitha R. S.
N. Jayaraj

Development of the Content

Dr. Jennath H.S., Abhayadev M., Thafseela Koya P., Sheena C.V., Fazia M. Aliyar, Unikrishnan S. Kumar

Review

Content : Rajagopal A.
Format : Dr. I. G. Shibi
Linguistics : Sujith Mohan

Edit

Rajagopal A.

Scrutiny

Dr. Jennath H.S., Shamin S, Suramya Swamidas P.C., Lekshmi A.C., Greeshma P.P., Sreerekha V.K.

Co-ordination

Dr. I.G. Shibi and Team SLM

Design Control

Azeem Babu T.A.

Cover Design

Jobin J.

Production

September 2024

Copyright

© Sreenarayanaguru Open University 2024

YouTube

MESSAGE FROM VICE CHANCELLOR

Dear

With immense joy and excitement, I extend my heartfelt greetings to all of you and warmly welcome you to Sreenarayanaguru Open University.

Established in September 2020 as a state-driven initiative, Sreenarayanaguru Open University is dedicated to advancing higher education through open and distance learning. Our vision is guided by the principle of "access and quality define equity," laying the foundation for a celebration of excellence in education. I am delighted to share that we are steadfast in our commitment to uphold the highest standards and refrain from compromising on the quality of education we offer. The university draws its inspiration from the legacy of Sreenarayana Guru, a revered figure in the Indian renaissance movement. His name serves as a constant reminder for us to prioritize quality in all our academic endeavors.

Sreenarayanaguru Open University operates within the practical framework of the widely recognized "blended format." Acknowledging the constraints faced by distance learners in accessing traditional classroom settings, we have curated a pedagogical approach centered on three main components: Self Learning Material, Classroom Counselling, and Virtual Modes. This comprehensive blend is poised to deliver dynamic learning and teaching experiences, maximizing engagement and effectiveness. Our unwavering commitment to quality ensures excellence across all aspects of our educational initiatives.

The university aims to offer you an engaging and stimulating educational environment that fosters active learning. The SLM is designed to offer a comprehensive and cohesive learning experience, fostering a deep interest in the study of technological advancements in IT. Careful consideration has been given to ensure a logical progression of topics, facilitating a clear understanding of the discipline's evolution. The curriculum is thoughtfully crafted to provide ample opportunities for students to navigate through the current trends in information technology. Furthermore, this course is designed to provide essential insights into computer hardware, software classification, and foundational HTML concepts crucial for web development.

We assure you that the university student support services will closely stay with you for the redressal of your grievances during your studentship. Feel free to write to us about anything that seems relevant regarding the academic programme.

Wish you the best.

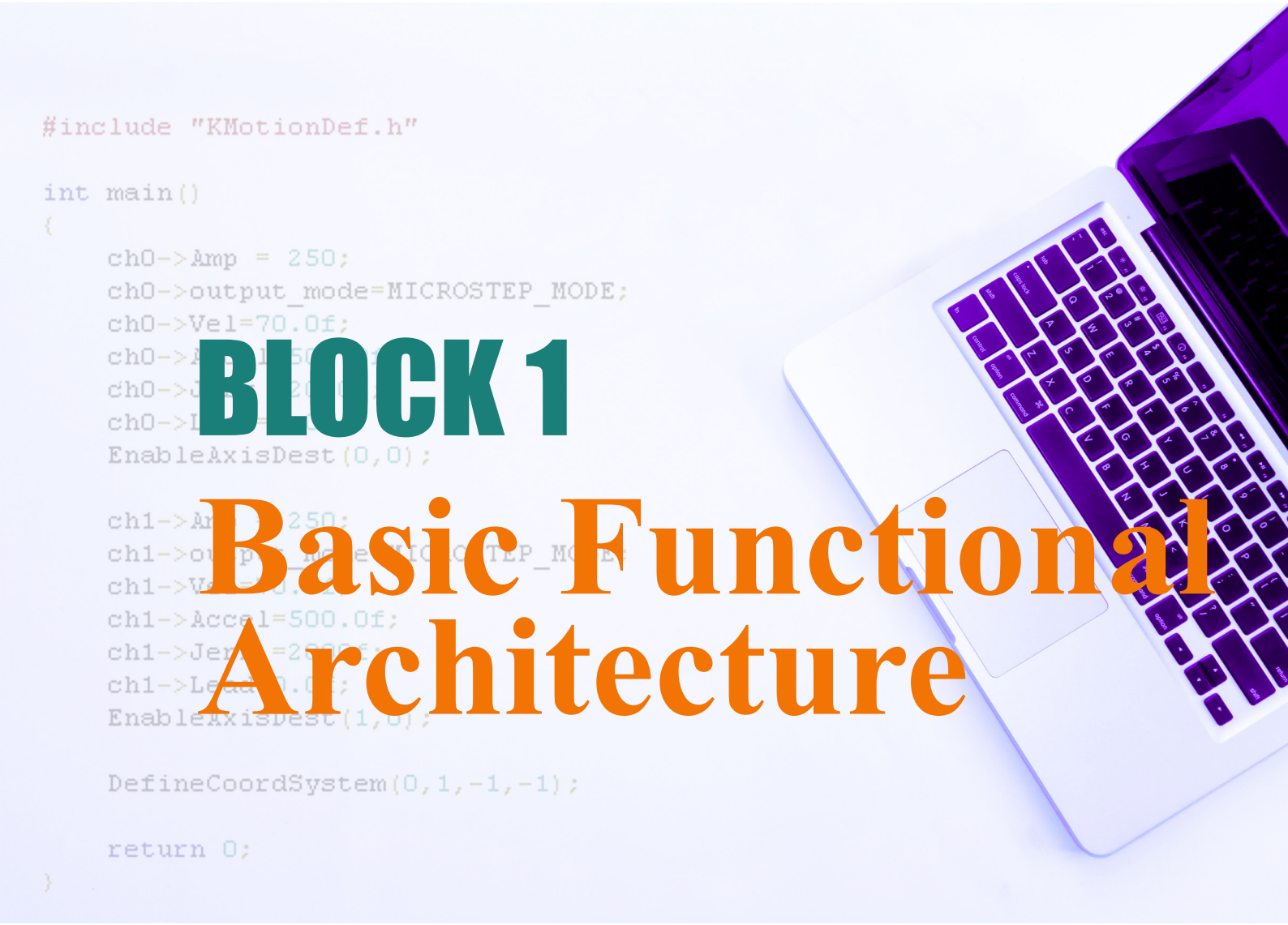


Regards,
Dr. Jagathy Raj V. P.

24-04-2024

Contents

Block 01	**Basic Functional Architecture**	**1**
Unit 1	Functional Units and Bus	2
Unit 2	Timing and Control	14
Unit 3	Addressing Modes	27
Unit 4	Program Control	38
Block 02	**I/O and DMA**	**48**
Unit 1	Register Transfer Languages	49
Unit 2	Input-Output Organization	63
Unit 3	Priority Interrupts	76
Unit 4	Direct Memory Access	87
Block 03	**Parallel Computer Structures**	**95**
Unit 1	Introduction to Parallel Processing	96
Unit 2	Architectural Classification	115
Unit 3	Pipelining - Introduction and Classification	132
Unit 4	Vector Processing and Array Processors	144
Block 04	**Basic Concepts of Operating Systems**	**156**
Unit 1	Operating System Components and Design	157
Unit 2	Operating System Services	178
Unit 3	Process Scheduling	191
Unit 4	Multiple Processor Scheduling	219
Block 05	**Process Synchronization**	**228**
Unit 1	Interprocess Communication	229
Unit 2	Introduction to Process Synchronization	253
Unit 3	Semaphores and Monitors	265
Unit 4	Deadlock in Operating System	279
Block 06	**Memory Management and File Systems**	**293**
Unit 1	Memory Management Strategies	294
Unit 2	Paging and Segmentation	311
Unit 3	Virtual Memory Management	321
Unit 4	File allocation and Management	335
	Model Question Paper Sets	**348**



BLOCK 1

Basic Functional Architecture



Functional Units and Bus

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- explain the basic concept of functional units of a computer
- familiarise the architecture of a computer
- narrate the internal operational concept of a computer
- introduce the concept of bus structure

Prerequisites

We can introduce a computer as an electronic computing machine that accepts input and processes it based on internally stored instructions, which we call programs. The result will be sent to some output device. However, when describing a computer, it is important to introduce it in terms of architecture and computer organisation.

Computer architecture describes various attributes like the instruction set of a computer, the number of bits used to represent various data types (e.g., the number of bits required to represent an integer, floating-point number or a character, etc.), input-output mechanisms, and addressing memory techniques, etc., that have a direct impact on the logical execution of a program. Computer organisation refers to the operational units and their interconnections that realise the architectural specifications.

Computers differ in their shape, size and performance. However, the basic organisation remains the same. The basic model is proposed by renowned Mathematician and Computer Scientist John Von Neumann.

Keywords

Arithmetic and Logic Units, Memory Units, Control Units, System Bus, Data Bus, Address Bus, Control Bus

Discussion

1.1.1 Generations of Computers

In this topic, we will discuss about various generations of computers. The generation term refers to how the hardware and software have evolved during the past years. The generation of computer terminology may change the world of information processing systems.

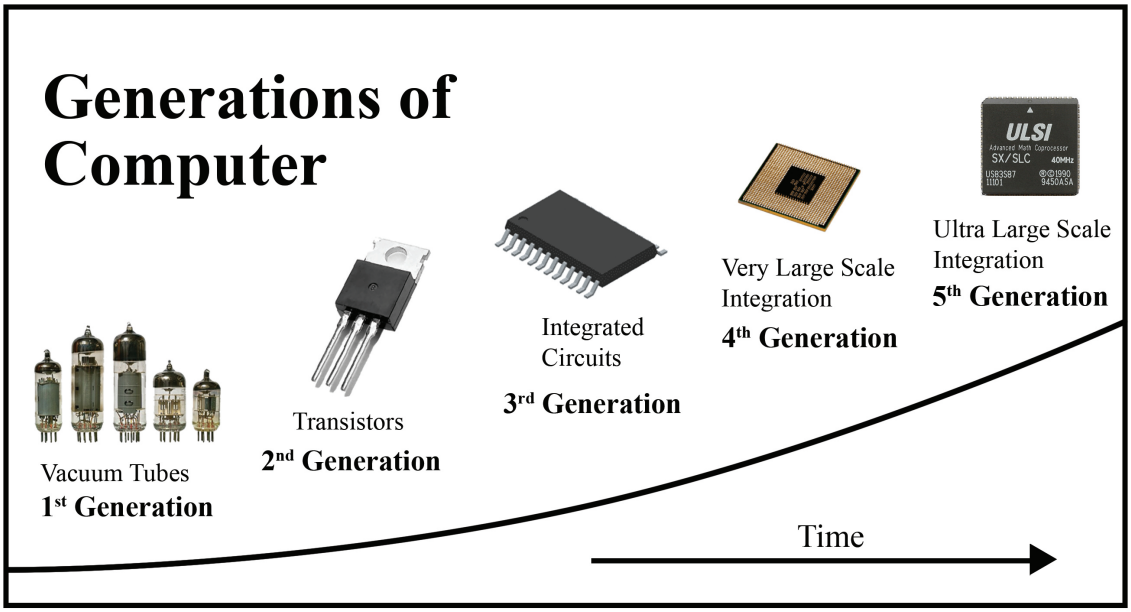


Fig 1.1.1 The evolution of computers

Fig. 1.1.1 explains the computer generations based on time and technology.

First Generation Computer (1940-1956): The first generation computers were extremely large. These early machines employed magnetic drums as storage and vacuum tubes as switches and amplifiers. These computers were too huge, expensive and consumed a lot of electricity, but they could perform complex calculations faster than previous mechanical devices. Examples: UNIVAC, ENIAC, EDVAC

Second Generation Computer (1956 to 1963): Used transistors instead of vacuum tubes, making computers smaller and more efficient. These computers included high-level programming languages like COBOL and FORTRAN. These computers were significantly faster than their predecessors. It had low power consumption and generated less heat. Examples of 2nd generation computers are the IBM1400 series, IBM 7090 and 7094, UNIVAC 1107, CDC 3600 etc.

Third generation computers (1964 - 1971): Third-generation computers employed using integrated circuits instead of transistors. The integrated circuit (IC) is a semiconductor material that contains thousands of transistors miniaturised in it. With the help of IC, the computer becomes more reliable and faster, requires less maintenance, is small in size, generates less heat, and is less expensive. Examples of third generation

computer are IBM 370, PDP-11, IBM System/360, UNIVAC 1108, Honeywell-6000 etc.

Fourth generation computers (1971 to 2010): The fourth generation computers saw the creation of even smaller computers, including laptops and hand-held devices. VLSI (Very Large Scale Integrated) Circuit, or microprocessors, is utilised in this generation. A microprocessor chip is made from hundreds of Integrated Circuits constructed on a single silicon chip. The use of personal computers (PCs) elevated in this generation, and the first personal computer (PC) advanced through IBM. Examples of 4th generation computers are Apple, CRAY-1.

Fifth Generation Computer (Present and Future): It is primarily based on artificial intelligence (AI). Artificial intelligence describes the medium and manner of creating computer systems like how human thinks, the manner people act, etc. Examples of 5th generation computer includes PARAM 10000, IBM notebooks, etc.

1.1.2 Functional Units of a Computer

The functional units of a computer refer to the distinct parts of a computer system that work together to process data and execute instructions. These units perform specialised roles, coordinating with each other to ensure the computer operates effectively. A computer, in its basic form, according to Neumann's model, comprises five main functional units, namely:

- Input unit
- Output unit
- Memory unit
- Arithmetic and Logic unit
- Control unit

Electrical cables or wires connect these units to permit inter-unit communication. These cables or wires are called buses. For the computer to function and solve problems, it should have access to both data and program statements (code). The computer gets the code and data from the input devices, and they are stored in the memory. The Central Processing Unit (CPU) is the brain of a computer. The CPU is responsible for processing the data from the input devices. After processing the input, the intermediate or final result from the CPU is sent to one or more output devices. A work area is required to carry out mathematical operations or other logical computations for data manipulation. This work area is termed the Arithmetic and Logic Unit (ALU). The overall coordination of all activities in a processor needs a central control unit. The CPU is the central controller of the system. CPU comprises a control unit, an arithmetic-logic unit, and an internal storage unit termed registers. Each internal unit within the CPU has a specific function to perform for the execution of intended tasks in a computer.

Fig. 1.1.2. shows the functional units of a computer system. Let us briefly discuss each functional unit:

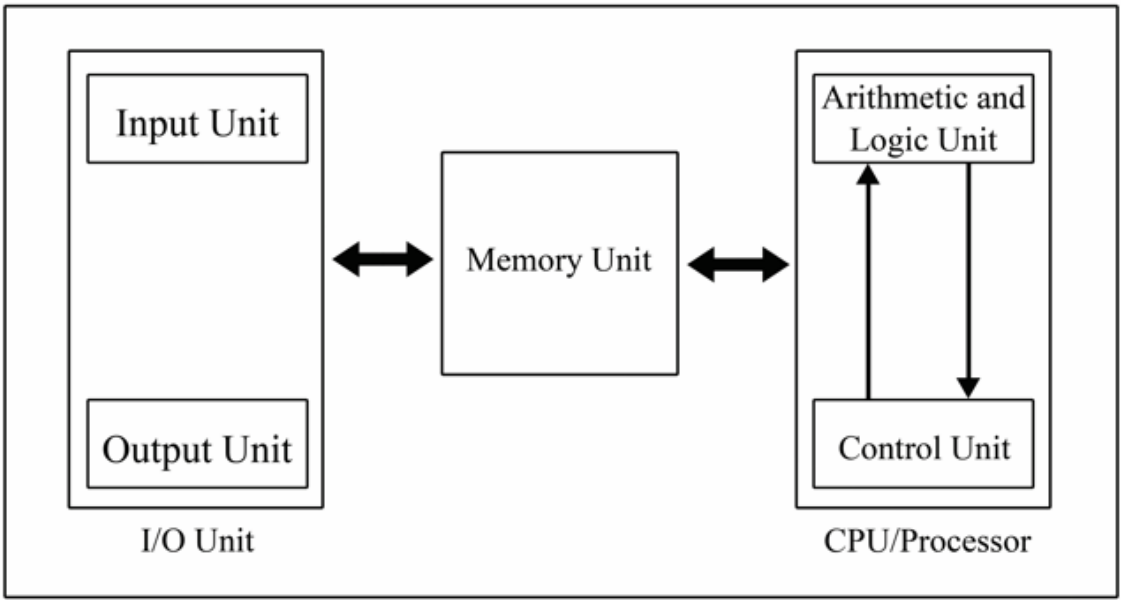


Fig. 1.1.2 Functional Units of a Computer

1.1.2.1 Input Unit

The input unit converts the input data into binary impulses before transferring it into the CPU. A few examples of input units/input devices are listed below:

- Keyboard: A standard input device. The keyboard controller sends the code to the CPU/Memory corresponding to every key-press.
- The mouse is another input device that permits you to select options from on-screen menus. The mouse's navigation could be done across a flat surface by pressing its buttons.
- The trackball and touchpad belong to the mouse species that allows us to draw or paint on the screen. The joystick is an input device used for playing video games. It consists of a swivelling lever mounted on a stationary base.
- Scanners and cameras are other examples of input units

The input device provides access to inputs or provides data and programs to a computer. Input data for the computer could come from various input sources such as magnetic disks, optical disks, or from keyboard or mouse. Encoded information or code from the input unit is sent to the processor.

1.1.2.2 Memory Unit

The memory unit is the storage unit of a computer. The code or the program instructions to be executed, the data to be processed, the output of computations, etc., are stored in the Memory Unit.

The memory unit comprises two classes of storage, namely:

1. Primary Memory

2. Secondary Memory.

Primary memory is a semiconductor memory that provides high-speed access. Programs are anticipated to reside in the main memory during execution. For the fast execution of programs at electronic speed, run-time program instructions and operands are stored in the main memory. Main memory is categorised into two types, namely:

Read-Only Memory (ROM): ROM stores system programs and firmware routines that are crucial for managing the hardware of a computer, such as BIOS, POST, I/O Drivers, etc. Volatility refers to the property of losing the contents of the memory after a power shutdown. ROM is a non-volatile memory.

Random Access Memory (RAM): RAM is termed Read/Write memory or user memory that holds run-time program instructions and data. RAM storage is expensive and volatile.

Secondary memory is a non-volatile memory and is less expensive compared to primary memory. Additional storage requirements of a computer could be compensated using this cheaper option. Examples of secondary memory include magnetic-based disks (Hard disks), optical disks, or semiconductor-based storage options. The access time of the secondary memory is greater than that of the primary memory.(access time refers to the time required for fetching a data from memory)

1.1.2.3 Arithmetic and Logic Unit

The arithmetic unit performs mathematical operations such as addition, multiplication, subtraction, and division of two numbers. ALU holds necessary arithmetic and logic circuits like adder, comparator, etc., for data processing. The logic unit performs logical operations involving comparisons. The comparison consists of checking against two data sets to see whether one is equal to, less than, or greater than the other. ALU is an indispensable building block of the central processing unit of a computer. In short, ALU is a digital circuitry inside the CPU that is responsible for performing two types of operations, namely arithmetic and logical.

1.1.2.4 Control Unit

The control unit is considered the nerve centre of a computer. It is responsible for the overall coordination of activities across all units in a computer. It tells the input unit, output unit, ALU, and memory units how to respond to program instructions. Control signals are issued by the control unit to achieve the system governance. Control signals govern the data transfers and provide appropriate timing for various operations. The control unit interprets or decides the operation/action to be performed at every instance.

1.1.2.5 Output Unit

The standard output device is the computer monitor, which comes in different types, such as CRT, LCD/TFT, and LED monitors. After successful computations and other operations, the computer returns the results, error messages, etc., via one or more output units. Other output devices are printers, projectors, plotters, speakers, etc.

1.1.3 Example of Basic Operational Concepts of a Computer

Here, we explore the operational concept of a computer with an example. Generally, an Instruction consists of two parts:

- **Operation code or Opcode:** It is a portion of a machine language instruction that tells the computer's CPU which operation to execute, such as addition, subtraction, load, or store. For example, in the instruction ADD A, 5; here ADD is the opcode, which tells the processor to perform an addition operation.
- **Operand/s:** The operand is the data or the location of the data that the instruction will process. It can be a constant value (immediate data), a memory address, or a register that holds the data. In an instruction, the operand is the operation applied to.

For example, in the instruction ADD A, 5, the operands are A (a register) and 5 (a constant value).

Consider the instructions for an additional operation given below. The location address of operand D and the content of register R0 are shown. Let us see how the computer processes this operation.

ADD (Address of D), R0

The following are the steps to execute the instruction:

1. Step 1: Instruction from main memory is fetched into the processor
2. Step 2: Operand at address location of D is fetched from main memory into the processor
3. Step 3: Perform addition operation with the memory operand to the contents of register R0
4. Step 4: Store the sum or result of the operation in R0.

The operations of a computer can be summarised as follows:

1. The program to be executed is regarded as a set of instructions, which must reside in the computer's main memory.
2. The instructions are fetched sequentially by the CPU, one after the other, from the main memory, which decodes them and executes the desired operations in ALU.
3. Processed data and results will be sent to one or more output units.
4. The control unit monitors and governs all activities related to processing and data movement inside the computer machinery.

1.1.4 Bus Structures

A bus is a group of wires/lines or a shared transmission medium that acts as a connecting path to establish a communication channel to interconnect multiple devices. For example, a 16-bit bus can transfer 16 bits at a time. Bus width also determines the max-

imum memory addressing capacity of the system. For example, the 8080 processor has a 16-bit address bus, giving 64k memory addressing capacity (216 address locations).

A basic computer has registers, a memory unit, a control unit and an arithmetic unit. Communication paths must be provided for the exchange of information between devices. The communication channel may be connecting one register to another, between memory locations and registers, or between CPU and other peripheral devices. If we have to provide separate wires for each of these communications, there will be excessive wires. A more efficient scheme for the exchange of information in a system with many registers or devices is to use a common bus.

Individual devices or units must communicate over a communication channel for the exchange of data, address and control information. This communication channel is termed a bus, where each line is capable of sending binary signals, either a one or zero. Multiple lines of a bus enable data to be transmitted in parallel. There are mainly two types of buses.

1. Internal bus (System bus)
2. External bus.

- External bus: It is mainly used to interconnect external devices to the CPU. Peripheral devices, such as a keyboard, mouse, scanner, etc., are connected to the CPU using the external bus.
- Internal bus: This bus is used to connect the CPU with the internal circuitry.

System bus is a bus that connects processor, memory and I/O units. Fig. 1.1.3. shows the architecture of a system bus.

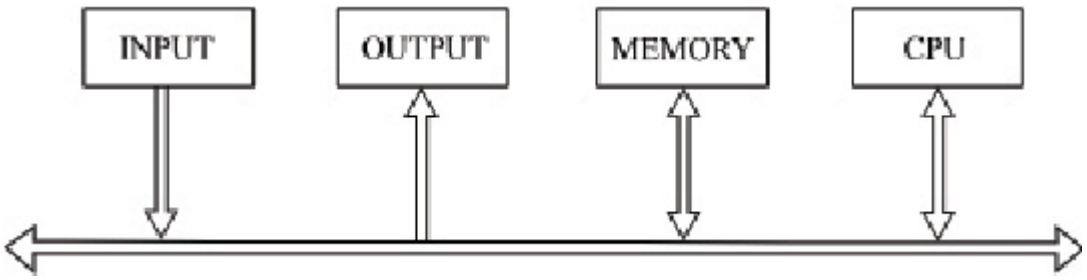


Fig. 1.1.3 System Bus

System Bus comprises three functional groups, namely:

- Data bus
- Address bus
- Control bus

1. Data bus: Data bus lines are bi-directional, and they provide the path for exchanging data between various system units. The data bus is a set of parallel signal lines of the order of 8, 16, 32 or more. Thus, the CPU will be able to read data on these lines from a port or a memory location and send data out on these lines to some memory location. These lines are collectively

termed data bus.

2. Address bus: It is a unidirectional bus. The address bus refers to a set of lines used to transmit address information, indicating the source and destination of the data that will be transferred via the data bus. The CPU sends out the address of the memory location it wants to read on the address line. Address bus generally consists of 16, 20, 24 or more parallel signal lines.
3. Control bus lines: The control bus is a collection of wires or signals used by the CPU to communicate control signals to various components within the computer system, such as memory, input/output devices, and other peripherals. These control signals manage and coordinate the different operations that occur during the execution of instructions.

The operation of the bus can be summarised as follows: For any module to send data to another module, it should:

- **Obtain Bus Control**:
  - The sending module must first gain control of the bus, often through a bus arbitration process if multiple devices are requesting access simultaneously.
- **Transmit Address**:
  - Once the bus is available, the sending module places the address of the destination module (or memory location) on the address bus to specify where the data is to be sent.
- **Transmit Data**:
  - The sending module places the data to be transferred onto the data bus, which will carry it to the destination module.
- **Send Control Signals**:
  - The sending module uses the control bus to transmit the necessary control signals (e.g., read/write signals) that indicate the type of operation (e.g., read, write, interrupt) to be performed on the data.
- **Wait for Acknowledgment**:
  - The sending module waits for an acknowledgement from the receiving module, which indicates that the data has been successfully received or that the operation has been completed.

This process ensures the orderly and successful transmission of data between modules in a computer system.

If a module needs to request data from another module, it should:

1. **Obtain Bus Control**:

The requesting module must first gain access to the bus, often through a bus arbitration process if multiple devices are trying to access the bus simultaneously.

2. **Transmit Address**:

The requesting module places the address of the data it needs on the address bus to specify the location in memory or the address of the other module from which the data is to be retrieved.

3. **Send Read Signal**:

The requesting module sends a read control signal via the control bus to indicate that it wants to read data from the specified address. This signal instructs the data source to prepare the requested data for transfer.

4. **Wait for Data**:

The requesting module waits for the data to be placed on the data bus by the module that holds the data. The data source module responds to the read request by placing the data on the data bus.

5. **Receive Data**:

The requesting module reads the data from the data bus once it is available. It then processes or stores the received data as needed.

This sequence of operations ensures that data requests and transfers between modules are managed efficiently and correctly.

A Single bus structure is summarised below:-

A single-bus structure is a computer architecture design where a single bus is used to connect multiple functional units within the system, such as the CPU, memory, and input/output (I/O) devices. This bus system facilitates communication and data transfer between these components using a unified set of pathways.

1.1.5 Bus Types

Bus lines are of two types: dedicated and multiplexed.

1. A dedicated bus line is permanently assigned to a function or a physical unit of a computer component. An example is the use of separate dedicated addresses and data lines.
2. Multiplexed bus lines. In this type, bus lines will not be dedicated to any single function.

An example is a multiplexed address and data lines in microprocessors. Address and data may be transmitted over the same set of lines using an Address Valid Control line. The Address Valid line is activated when the address is placed on the bus. For the data transfer operation, the Address Valid Control line is set low for read or write data transfer. This method of using the same lines for multiple purposes is known as time multiplexing.

Recap

- Five functional units of a computer: input and output unit, Arithmetic (ALU), control unit, and memory unit.
- The input unit provides - data and program instructions for the computer.
- Examples of input devices are a keyboard, computer mouse, joysticks, camera, microphone, scanner, tracker ball, light pen, etc.
- The memory unit stores the data and code.
- Two types of Memory -primary and secondary memory.
- Primary memory -fast, volatile (temporary memory) and expensive.
- Secondary memory - cheap, permanent and slow.
- Examples of secondary memory include semiconductor, magnetic and optical disks.
- ALU comprises various logical units such as comparators, adders, multipliers, etc.
- CPU comprises ALU, Control Unit and primary memory.
- The control unit generates control signals for the centralised control of the various units involving data transfer and memory access.
- Examples of output units include monitors, plotters, speakers, etc.
- Bus refers to a group of lines or a shared transmission medium to establish a communication channel for multiple devices.
- An external bus is used to connect external devices to the CPU
- An internal bus connect the CPU with internal devices or circuitry.
- System Bus has three functional groups - data bus, address bus and control bus.

Objective Type Questions

1. To which functional unit do a scanner and camera belong to?
2. Where are the operands and opcodes stored on a computer?
3. Which type of memory does RAM belong to?

4. Which one is a volatile memory, RAM or a ROM?
5. Which type of memory holds firmware-related programs?
6. Which sub-unit performs the mathematical or arithmetical operations of computations?
7. Which unit is considered the nerve centre of a computer?
8. Which sub-unit contains comparators for logical operations?
9. Which functional unit is responsible for overall control of the data transfer in a computer?
10. Which bus is used to control the data bus and address bus?

Answers to Objective Type Questions

1. Input Unit
2. RAM
3. Primary Memory
4. RAM
5. ROM
6. Arithmetic unit
7. Control Unit
8. Logical unit
9. Control Unit
10. Control bus.

Assignments

1. Explain functional units of a computer system with diagrams.
2. What is the need of input and output units in a computer system?
3. Explain different input devices.
4. What is memory? Explain the need for memory in computers. Explain different types of memory.
5. What is a bus on a computer? Explain different bus systems.
6. What are the functions of control units?
7. What is an Arithmetic logic unit in a computer?

Suggested Reading

1. Carl Hamacher -Computer Organization, Fifth Edition, Tata McGraw-Hill.
2. John P Hayes -Computer Architecture Organization–Mc GrawHill
3. William Stallings-Computer Organization and Architecture, Seventh Edition, Pearson Education



Timing and Control

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- narrate the functions of control signals
- familiarize the types of control signals
- explain the working of a control unit
- make them understand the need for timing signals

Prerequisites

We have a basic understanding of the fundamental elements of a processor, like ALU, registers (cache memory), and control units. Computer programs instruct the processor to perform certain mathematical and logical operations to get the desired result. Each processor has a defined instruction set. Instructions primarily contain two parts, namely opcode and operands. Opcode defines the operation to be performed, and the operand could be a direct value of a register or an address of memory location of the operand. Program instructions can be considered a set of instructions loaded into the main memory, which are later fed into the processor for execution.

The execution of an instruction is ideally the execution of a sequence of sub-steps, termed cycles. For instance, an execution may consist of fetch, decode, execute, and store cycles. Each cycle consists of a sequence of fundamental operations called micro-operations. A micro-operation could be one of the operations explained below:

- Data Transfer from one register to another.
- Data Transfer from a register to an external interface.
- Data Transfer from an external interface to a register.
- Perform an arithmetic or logic operation using register accumulators for input and output.

The operands located in the memory need to be fetched using a system bus for further processing. Mathematical operations like addition or logical operations like AND gate,

OR gate etc., are performed by ALU of the processor. The processor keeps track of the next instruction to be executed and is indexed using the program counter. All these operations are managed by the control unit. The two primary functions of the control unit are to perform sequencing and execution functions.

1. Sequencing: The control unit causes the processor to step through a series of micro-operations in the proper sequence based on the program being executed.
2. Execution: The control unit causes each micro-operation to be performed. Primarily the control unit controls the operations of the processor using its control signals.

Keywords

Control bus, Control path, Control signal, Control unit, Hardwired implementation and micro-operations

Discussion

1.2.1 Discussion

The timing and control unit is an important section of the CPU. It is employed to generate timing and control signals necessary for the instruction execution. It is used to control data flow between CPU and peripherals such as input-output units, memory, etc. The organisation of a computer is defined in terms of its internal registers, the instruction set, bus structure and timing and control structure. One major responsibility of the control unit is to send control signals to various system components. The control unit is responsible for managing the ALU operations, bus operations, memory operations, register operations, etc. The control signals generated in the control unit provide control inputs for various components of the processor, like

1. multiplexers in the common bus,
2. various processor registers,
3. micro-operations for the accumulator and
4. controlling the external system bus.

Before we discuss timing and control, it's important first to become familiar with some common terms in computer architecture

1.2.1.1 Flip-flops

Flip-flops are fundamental building blocks in digital electronics, it is used to store and manipulate binary data. They are bistable devices, meaning they have two stable states and can hold a bit of information (0 or 1). Flip-flops are essential for creating registers,

memory units, and other sequential circuits in digital systems. Examples of flip-flops are SR Flip-Flop (Set-Reset Flip-Flop), SR Flip-Flop (Set-Reset Flip-Flop), D Flip-Flop (Data Flip-Flop or Delay Flip-Flop), JK Flip-Flop (jack- kilby FF), T Flip-Flop (Toggle Flip-Flop).

Basic Characteristics of Flip-Flops

1. Bistable: A flip-flop has two stable states, representing binary values 0 and 1.

2. Edge-Triggered: Flip-flops can change states on specific edges (rising or falling edge) of a clock signal, making them synchronous devices.

3. Set and Reset: Flip-flops can be set to a high state (1) or reset to a low state (0) based on input signals.

Applications of Flip-Flops

1. Registers: Flip-flops are used to build registers, which store multiple bits of data. For example, an 8-bit register can be constructed using eight flip-flops to store 8 bits of data.

2. Counters: Flip-flops are used in binary counters, which count pulses and can be configured for different counting sequences (e.g., up counters, down counters).

3. Memory Units: Flip-flops are the building blocks of memory cells in static RAM (SRAM) and other types of memory.

4. Clock Division: T flip-flops are used in frequency dividers to generate lower frequency signals from a higher frequency clock.

5. State Machines: Flip-flops are used in sequential logic circuits and finite state machines (FSMs) to store and manage state information.

1.2.1.2 Register

Registers are small, high-speed storage locations within the CPU that hold data, instructions, addresses, or control information. They are used for various purposes, including holding intermediate results of computations, storing addresses for memory operations, and controlling the flow of instructions.

Registers are crucial for the CPU's operation because they provide fast access to data and instructions needed for processing. They are faster compared to other forms of memory, like RAM, because they are located directly within the CPU.

Types of Registers

Accumulator (ACC): Holds the intermediate results of arithmetic and logic operations. Many CPU instructions use the accumulator to perform calculations and then store the results.

Program Counter (PC): Purpose: Holds the address of the next instruction to be executed. After fetching an instruction, the Program Counter is updated to point to the subsequent instruction.

Instruction Register (IR): Stores the current instruction being decoded and executed. It holds the instruction fetched from memory before it is decoded.

Data Register (DR): Temporarily holds data that is being transferred to or from memory or I/O devices.

Memory Address Register (MAR): Holds the address of the memory location to be accessed. It is used to specify where data should be read from or written to.

Stack Pointer (SP): Points to the current top of the stack in memory. The stack is used for the temporary storage of data, function parameters, return addresses, and local variables.

Memory Buffer Register (MBR): Holds the base address for indexed addressing. It is used in some architectures to simplify the calculation of memory addresses.

Index Register (IRX): Used to hold an index value for indexed addressing. It is often used in array processing to modify the address in memory.

Status Register (SR) / Flags Register: Holds flags or status bits that indicate the result of operations or the status of the CPU. Flags may include zero, carry, overflow, and sign bits.

Control Register (CR): Stores control information that influences the operation of the CPU or the memory subsystem. It is used to configure or control certain CPU functions.

1.2.1.3 Decoder

A **decoder** is an essential component in computer architecture that translates encoded instructions or addresses into control signals or actionable information. It plays a crucial role in the CPU and memory management, ensuring that operations are carried out correctly. The functions of a Decoder are Instruction Decoding, Address Decoding, and Control Signal Generation:

Types of Decoders:

1. **Binary Decoder**: Converts binary input values into a specific output line. For example, a 2-to-4 binary decoder converts a 2-bit input into one of four output lines, with only one output line being activated at a time.
2. **BCD (Binary-Coded Decimal) Decoder** - Converts a 4-bit binary input into a corresponding decimal output. It is often used in applications where decimal representation is required.
3. **Address Decoder**: Used in memory and I/O systems to decode address lines and select the appropriate memory or I/O device. For example, in a system with multiple memory modules, an address decoder ensures that the correct memory module is accessed based on the address provided.
4. **Instruction Decoder**: Part of the CPU's control unit, it decodes the instruction

opcode into control signals that direct the various components of the CPU to perform the required operation.

Example of Decoder Operation:

Consider a simple 2-to-4 binary decoder:

- **Inputs**: 2-bit binary value (e.g., 00, 01, 10, 11).
- **Outputs**: 4 lines, where only one line is activated based on the binary input.
  - For input 00, output line 0 is activated.
  - For input 01, output line 1 is activated.
  - For input 10, output line 2 is activated.
  - For input 11, output line 3 is activated.

1.2.1.4 Latches

Latches are fundamental components in digital circuits and are essential building blocks in computer architecture, particularly in memory elements and timing circuits. They are used for storing a single bit of data and are a critical part of flip-flops, registers, and other storage elements within processors and digital systems. A latch is a type of bistable circuit, meaning it has two stable states: 0 and 1. It can store a bit of data and remain in its state until it is changed by an input signal. Latches are level-sensitive, meaning their output depends on the input as long as the enabling signal (clock or enable) is active at a certain level (high or low). Different types of latches are SR Latch (Set-Reset Latch), D Latch (Data Latch), JK Latch, and T Latch (Toggle Latch).

1.2.1.5 Clock

Clocks in computer architecture play a fundamental role in synchronising the operations of a computer's components. They provide the timing signals that regulate the sequencing of instructions and data flow across various parts of the system.

A clock is a circuit that emits a series of pulses with a precise pulse width and precise interval between consecutive pulses. The time interval between the corresponding edges of two consecutive pulses is known as the clock cycle time. Pulse frequencies are commonly between 100 MHz and 4 GHz, corresponding to clock cycles of 10 nsec to 250 psec. To achieve high accuracy, the clock frequency is usually controlled by a crystal oscillator. In a computer, many events may happen during a single clock cycle. If these events occur in a specific order, the clock cycle must be divided into subcycles.

A common way of providing finer resolution than the basic clock is to tap the primary clock line and insert a circuit with a known delay in it, thus generating a secondary clock signal that is phase-shifted from the primary, as shown in Fig. 1.2.1 (a). The timing diagram of Fig. 1.2.1(b) provides four-time references for discrete events:

1. Rising edge of C1.
2. Falling edge of C1.

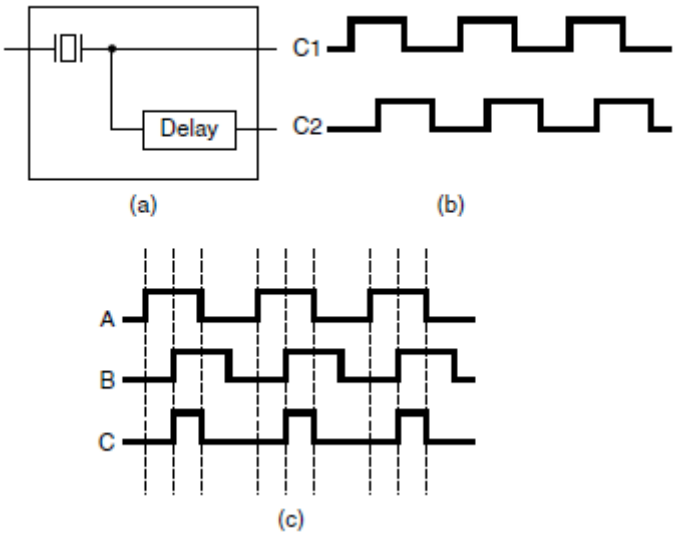


Figure 1.2.1 a) Clock b) Timing diagram for the clock c) Generation of the asymmetric clock

3. Rising edge of C2.

4. Falling edge of C2.

By tying different events to the various edges, the required sequencing can be achieved. If more than four-time references are needed within a clock cycle, more secondary lines can be tapped from the primary, with one with a different delay if necessary.

In some circuits, one is interested in time intervals rather than discrete instants of time. For example, some events may be allowed to happen whenever C1 is high rather than precisely at the rising edge. Another event may happen only when C2 is high. If more than two different intervals are needed, more clock lines can be provided, or the high states of the two clocks can overlap partially in time. In the latter case, four distinct intervals can be distinguished: (C1 AND C2, C1 AND C2, C1 AND C2, and C1 AND C2). As an aside, clocks are symmetric, with time spent in the high state equal to the time spent in the low state, as shown in Fig. 1.2.1.(b). To generate an asymmetric pulse train, the basic clock is shifted using a delay circuit and ANDed with the original signal, as shown in Fig. 1.2.1 (c) as C.

1.2.2 Timing and Control

A master clock generator is used to synchronise the timing for all registers in a basic computer. A master clock generator is a crucial component in digital systems and electronic devices, responsible for generating a precise timing signal (clock signal) that synchronises the operation of various parts of the system. The clock signal acts as a timing reference, coordinating activities across components like the CPU, memory, I/O devices, and other circuits to ensure that data is transferred, processed, and executed in a synchronised manner. The control unit (CU) of a processor converts machine instructions to corresponding control signals for managing the micro-operations. The clock pulses are applied to all registers in the system. The clock pulses would only

change the state of a register if the control signal is high/enabled for the register.

There are two types of control organisations:

1. Hardwired control and
2. Micro-programmed control.

1.2.2.1 Hardwired control

In Hardwired control, the control logic is realised using logic gates, flip-flops, decoders, and some other digital circuits. This approach offers a faster mode of operation due to the optimised performance of the digital circuits. However, the limitation of the approach is that it is required to change the wiring and the associated components for any modification or change in the hardwired control design.

The block diagram of the hardwired control organisation unit is shown in Fig. 1.2.2. It consists of

- Instruction register

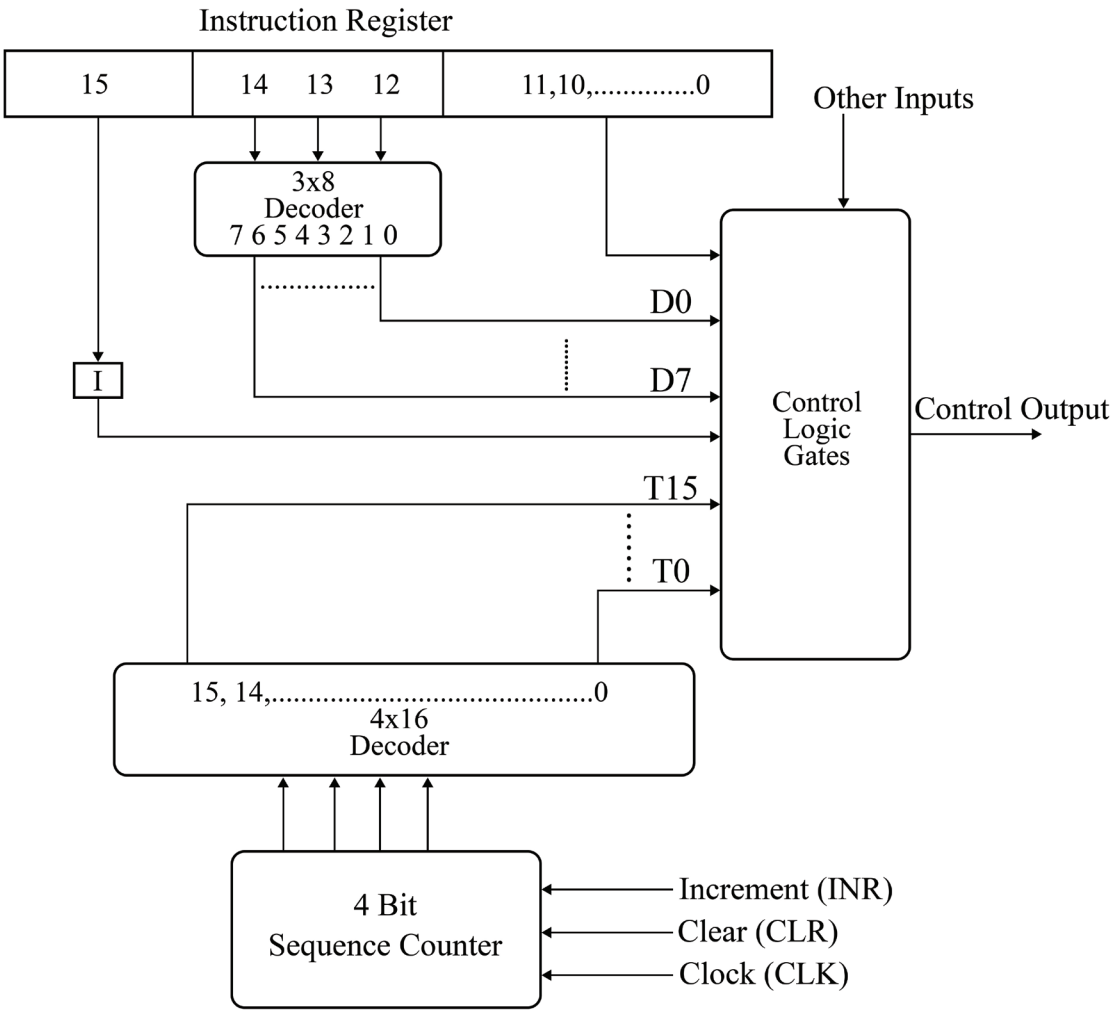


Fig 1.2.2 Hardwired Control Unit of a Basic Computer

- A sequence counter, and
- A number of control logic gates.

The instruction register (IR) is the part of a CPU's control unit that holds the instruction currently being executed or decoded. An instruction read from memory is placed in the instruction register (IR). The instruction register is divided into three segments, they are:

- The I bit section holds bit number 15 of IR,
- The three-bit operation code (bits 14,13 and 12) forms the second section
- Bits 0 to 11 form the third section.

It is evident from Figure 1.2.2 that hardwired control consists of a

- A 3x8 decoder has three inputs and produces eight outputs. The opcode part in IR (bits 14,13 and 12) is decoded using a 3 into 8 decoders.
- The output of the decoder is designated with labels D0 through D7. The decimal number in the subscript refers to the binary value of the analogous operation code.
- Bit 15 of the IR is given as an input to a flip flop labelled I. The output of this flip-flop is another input to the control logic gate.
- Bits 0 through 11 of the IR are given as input to the control logic gate.
- A 4 bit sequence counter can count from 0 to 24 −1, i.e. from 0 through 15. The output of the sequence counter is decoded using a 4 into 16 decoders, as shown in the figure. The decoded output is a set of 16 timing signals designated with labels from T0 through T15. These timing signals are the inputs to the control logic gate.

1.2.2.2 Micro-programmed Control

It refers to the design and implementation of control units in digital computers using micro- programming techniques. In micro-programmed organisation control, the control information is loaded into a control memory. The control memory is programmed to initiate the required sequence of micro-operations.

1. Microprogramming: A technique used to implement control units using software like instructions
2. Control Unit: Responsible for managing data transfer, arithemetic operations, and instruction execution
3. Micro instructions: Basic building blocks of microprograms, specifying control signals and operations

The main advantage of the micro-programmed control is that, for any change in

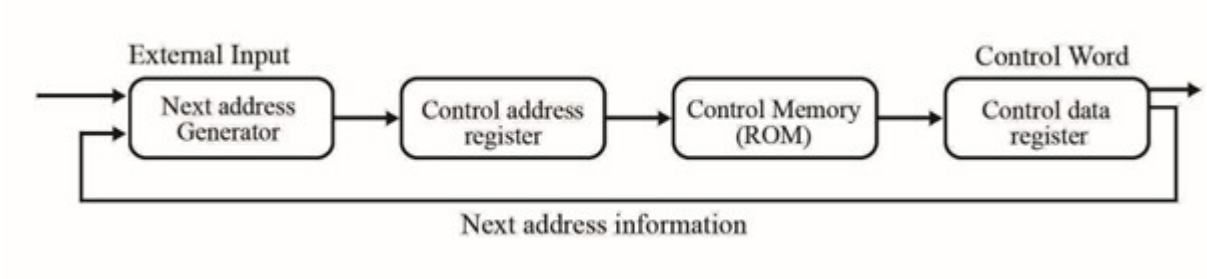


Fig. 1.2.3 Block diagram of a Micro-programmed Control organisation

design, the modifications correspond to the mere updating of the micro-program in the control memory. A microprogram, consisting of microinstructions, is stored in the control memory of the control unit. Execution of micro instruction is responsible for the generation of a set of control signals.

Fig. 1.2.3 is the block diagram of a Micro-programmed Control organisation.

- The address of the micro-instruction is specified in the Control memory address register.
- The Control memory is where the control information is permanently stored. It is usually a Read Only Memory (ROM).
- The micro-instruction fetched from the memory is held in the control register.
- Each micro-instruction contains a control word. Each control word refers to one or more micro-operations for the data processor.

Table 1.2.1 Comparison between Hardwired and Microprogrammed control units

Hardwired Control	**Microprogrammed Control**
The hardwired control unit induces the control signals required for the processor.	The microprogrammed control unit induces the control signals through microinstructions.
Hardwired control unit is quicker than a microprogrammed control unit.	Microprogrammed control unit is slower than a hardwired control unit.
It is hard to modify.	It is easy to modify.
It is more expensive as compared to the microprogrammed control unit.	It is affordable as compared to the hardwired control unit.
It faces difficulty in managing the complex instructions because the design of the circuit is also complex.	It can easily manage complex instructions.
It can use limited instructions.	It can generate control signals for many instructions.

- During the execution of the micro-operations, the next address generator circuit estimates the next address. It is transferred into the control address register to read the next microinstruction.

Table 1.2.1 shows a comparison between Hardwired and Microprogrammed control units.

1.2.3 Timing signal

Consider the example of a Control Timing Signal as shown in Fig. 1.2.4. In this example, the system is designed to generate 5 timing signals: T0, T1, T2, T3, T4. The sequence counter is incremented from 0 to 4 to generate the timing signals T0, T1, T2, T3, T4. At T4, the sequence counter is required to reset, and it is cleared to zero using the CLR input. For each clock, the sequence counter could be either incremented or reset-ed. Normally, the counter is permitted to increment till it reaches the desired count. In this case, after reaching the count T4, the counter reset to 0 at the next clock. The CLR input of the decoder output resets the counter. This causes the timing signal T0 to be active in the next time step. This is represented symbolically as

D3T4: SC ← 0

From Fig. 1.2.4. It is evident that the sequence counter (SC) is a positive edge-triggered

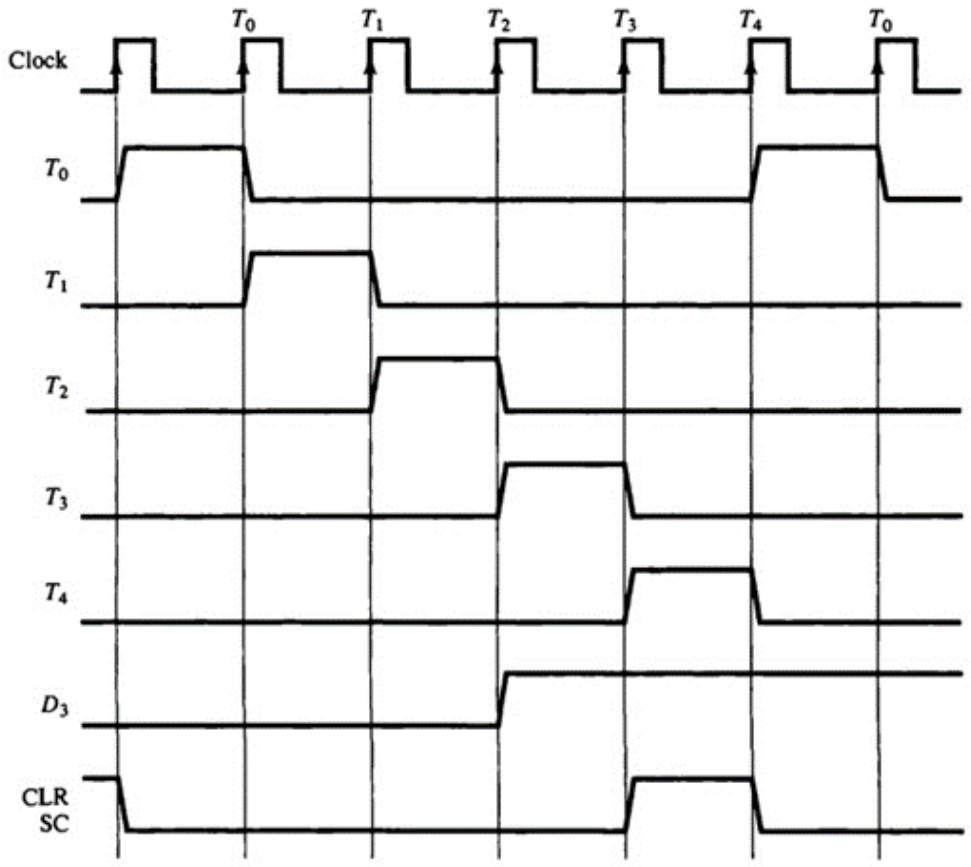


Fig 1.2.4 Example of Control Timing Signal (Source: M. Morris Mano, Computer System Architecture)

device. This refers to the fact that SC responds to the positive transition of the clock. In the beginning, the CLR input of SC was set high (active). Hence, the first positive transition of the clock clears the sequence counter to 0. This would activate the timing signal T0 at the output of the decoder. T0 will be active for the duration of one clock cycle. In the subsequent cycles, T2 will go high. The 4 bit sequence counter can generate up to 15 counters.

SC increments with every positive edge of the clock transition except for the condition when CLR input is active or high. In the example being considered, CLR is high at T3, giving a reset. This causes the decoder to produce a sequence of timing signals T0, T1, T2, T3 and T4, as depicted in Figure 5 and then again back to T0. If SC is not cleared, the timing signals will continue with T5, T6 up to T15 and back to T0.

Recap

- Introduction to register, latches, flip-flop, decoder
- Instruction register -keeps currently executing instruction
- The control unit provides control inputs for various components of the processor, such as controlling

  (a) Internal bus operation,

  (b) Registers,

  (c) Micro-operations for the accumulator and

  (d) External system bus.
- Two types of control organisation: Hardwired control and micro-programmed control.
- In Hardwired Control Unit: The control logic is realised using digital circuits
- In micro-programmed control, the control information is loaded into a control memory. The control memory initiates the sequence of micro-operations.

  (a) Two decoders,

  (b) A sequence counter, and

  (c) A number of control logic gates.
- The advantage of micro-programmed control is that for any change in design it is required to make the modifications in the programming part. This correspond to the mere updating of the micro-program in the control memory.
- Control Signals: Timing signal and Command signal.

Objective Type Questions

1. What is the type of control unit designed using digital circuitry called?
2. Where is the control information stored in micro-programmed control?
3. What is used for synchronisation of digital devices?
4. To change the content of registers, what is required in addition to the clock?
5. How many timing signals could be generated by a 4 bit sequence counter?
6. Which device is used to convert the output sequence counter to timing signals?
7. Which type of control unit uses Programmable control for control signal generation
8. Where is the current instruction being fetched from the memory stored?

Answers to Objective Type Questions

1. Hardwired Control
2. ROM
3. Clock
4. Control signals
5. 16
6. Decoder
7. Micro-programmed control
8. Instruction register

Assignments

1. Explain control signals in a computer.
2. Explain hardwired control signals and microprogrammed control signals.
3. Explain timing signals and command signals.
4. How do you coordinate all operations of a computer by control signals?
5. Explain registers.
6. What are flip-flops? What are the applications of registers?
7. Write a comparison between Hardwired control and microprogrammed control.

Suggested Reading

1. M.Morris Mano-Computer Systems Architecture, Third Edition, Pearson Education
2. Kai Hwang and F A Briggs-Computer Architecture and parallel processing, McGraw Hills,1990
3. Carl Hamacher -Computer Organization, Fifth Edition, Tata McGraw Hill.
4. John P Hayes -Computer Architecture Organization -Mc Graw Hill
5. William Stallings-Computer Organization and Architecture, Seventh Edition, Pearson



Addressing Modes

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- recall the concept of addressing modes
- list various addressing modes
- familiarise examples of various addressing modes
- understand the need for addressing modes

Prerequisites

An instruction set is a collection of commands or instructions defined for a processor in machine language. It covers all possible instructions supported by the CPU. Some instructions are simple read-write commands, and some involve moving commands that direct data to different hardware or periphery. There are arithmetical commands or logical commands that perform various operations over the data.

Consider an example of a move instruction command given below: MVI A,03H

Every instruction has two parts. The first part of the instruction, here in this example, “MVI”, is termed the opcode part. The second part of the instruction, “A, 03H”, is the operand part.

- Opcode (or operation code)- defines the task or operation to be performed by the CPU.
- Operand - This refers to the data to be operated on.

The operands of the instructions can be given directly, or they can be located either in the main memory or in the CPU registers.

Keywords

Immediate Mode, Register Mode, Register Indirect Mode, Auto Increment/ Decrement Mode, Direct Addressing Mode, Indirect Addressing Mode, Indexed Addressing Mode

Discussion

1.3.1 Addressing Modes

The term addressing modes refers to a mode of resolving the operand or data to be used in the execution of its instructions by the CPU. If the operand is placed in the main memory, then the instruction provides the location address in the operand field. Many methods are followed to specify the operand address. The various formats for specifying operands or data in an instruction are called the ADDRESSING MODES. Different computers support different types of addressing modes. The selection of addressing modes depends on the design and architecture of the computer.

Various common types of Addressing Modes are

1. Implied Mode
2. Immediate Mode
3. Register Mode
4. Register Indirect Mode
5. Auto Increment/Decrement Mode
6. Direct Addressing Mode
7. Indirect Addressing Mode
8. Indexed Addressing Mode

1.3.1.1 Implied mode

Implied addressing mode is also known as implicit addressing mode; the operands are specified implicitly in the definition of the instruction. All instructions that use an accumulator are implied mode instructions.

Given below are few examples of implied mode:

1. CLC (used to reset the Carry flag to 0)
2. INCA (“increment the contents of the accumulator register by one”)
3. DECA (“decrement the contents of the accumulator register by one”)
4. NOP (Perform No operation, used to add wait cycles)
5. RRC (rotate accumulator A right by one bit)
6. RLC (rotate accumulator A left by one bit)

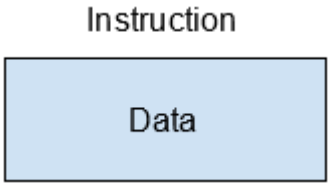


Figure 1.3.1 Implied Addressing Mode

Also zero address instruction works in implied addressing mode. Implied addressing refers to instructions that contain only an opcode and no operand, as shown in Figure 1.3.1. Any data required by the instruction and the destination of any result from the instruction are implied by the instruction itself. Zero address instructions in a stack-organized computer also belong to implied mode instructions. Hence, the implied addressing mode is also termed the stack addressing mode.

1.3.1.2 Immediate Addressing Mode

In Immediate addressing mode, the operand is specified in the instruction itself. In this, the operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.

Consider the example: ADD 6;

This instruction says to Add the value of 6 to the contents of the accumulator and save the result in the accumulator. ADD is the opcode, and 6 is the operand here. A few other examples are

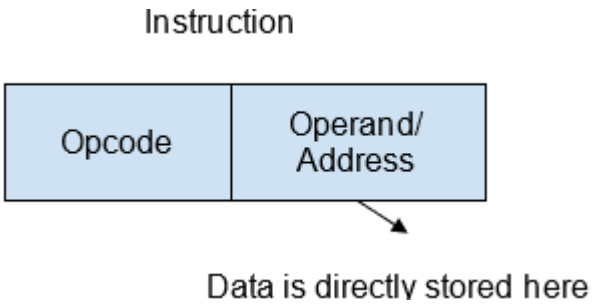


Fig. 1.3.2 Immediate Mode

1. ADD 07; Add the operand (here 07) with the contents of the accumulator
2. MOV AX, 40H ; Immediate constant 40H is transferred to accumulator (AX)
3. MVI B 47 (move the data 47H immediately to register B)
4. MOV AL, 30H (move the data 30H into the AL register.)
5. JMP 30021 (jump to the operand address immediately)

In this mode of addressing, the operand is given immediately or directly after the opcode. Immediate addressing mode is not a real memory addressing mode. It simply gives an instruction format that directly includes the data or operand as part of the instruction, as shown in Fig. 1.3.2. An immediate mode instruction has a direct operand field and not the address field in the definition of the instruction. Also, the operand

in the accumulator register is implied in the instruction definition. Immediate-mode instructions offer convenience in initialising registers to some constant value.

1.3.1.3 Direct Addressing Mode

In direct addressing mode, the address field contains the address of the operand.

Consider the following example instructions:

Code Comment

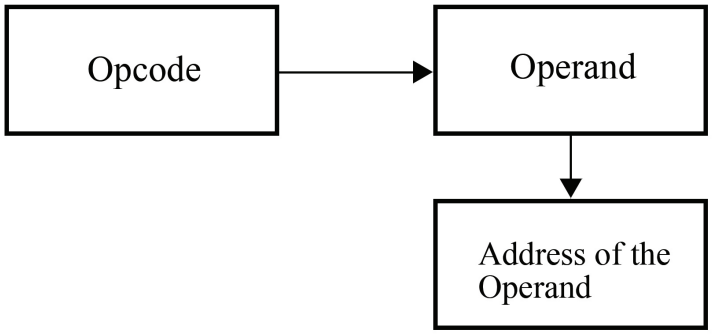


Fig 1.3.3 Direct Addressing Mode

1. ADD AL,[0301] //add the contents of offset address 0301 to AL
2. LDA 2050 //(load the contents of memory location 2050 into accumulator A)
3. LHLD 3010 //(load contents of 16-bit memory location into H-L register pair)
4. IN 45 //(read the data from port whose address is 45)

In this addressing mode, the operand is an address. This operand refers to the address of the data in the memory. In other words, the data to be operated is available inside a memory location. That memory location is directly specified as an operand, as shown in Fig. 1.3.3. The operand is directly given in the instruction. In a branch-type instruction, the address field specifies the actual branch address. In short, in this mode, the effective address is equal to the address part of the instruction. The operand resides in memory, and its address is given directly by the instruction field.

1.3.1.4 Register Mode

In register addressing mode, the operand is placed in one of 8/16-bit general purpose registers. The data is in the register that is specified by the instruction.

Consider the following instructions

1. MOV DX, TAX RATE; Register in the first operand
2. MOV COUNT, CX; Register in the second operand
3. MOV EAX, EBX; Both the operands are in registers; Instructs to move the contents of the register EBX to EAX

4. ADD R1, R2 ; Add the contents of the register R1 with R2 and save the result in R1

We can see that in the above examples, the instructions involve the internal registers as operands. In this addressing mode, the operands include registers that are part of the CPU. The specific register is selected from a register field in the instruction. Register

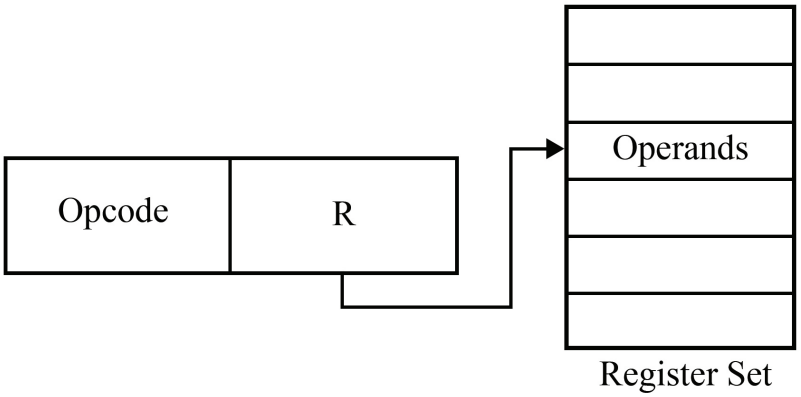


Fig 1.3.4 Register Addressing Mode

could be both the operands or either one of them, as shown in the example above. The pictorial representation of the Register

Addressing Mode is shown in Fig. 1.3.4.

1.3.1.5 Register Indirect Mode

In Register indirect mode, the operand's offset is placed in any one of the registers BX, SP, SI, DI as specified in the instruction.

Consider the following examples :

1. MOV A, M (move the contents of the memory location pointed by m to the H-L pair to the accumulator)
2. LDAX B (move contents of B-C register to the accumulator)

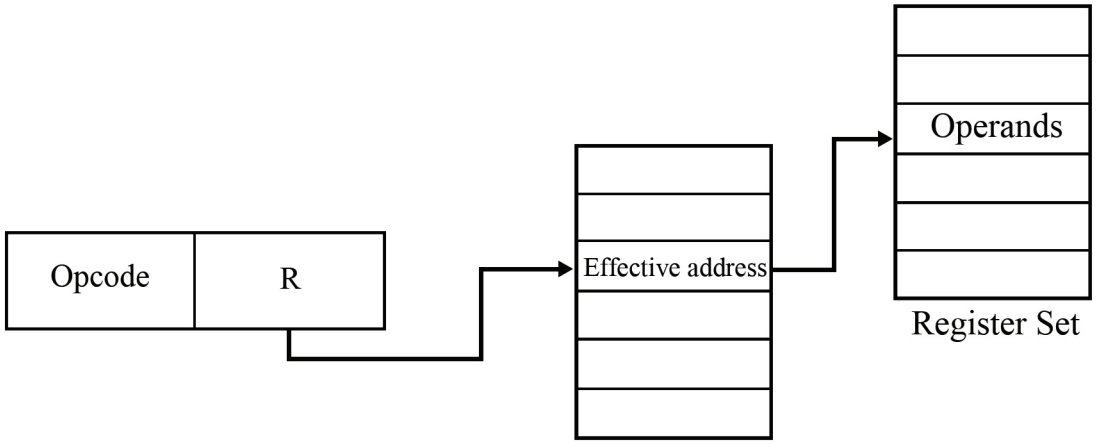


Fig 1.3.5 Register Indirect Addressing Mode

3. LXIH 9570 (load immediately the H-L pair with the address of the memory location 9670)
4. ADD R (will increment the value stored in the accumulator by the content of memory location specified in register R. This is equivalent to Accumulator ← Accumulator + [Contents of the memory location stored in [R]]

In this addressing mode, the address field of the instruction refers to a CPU register that contains the effective address of the operand, as shown in Fig. 1.3.5. It requires only one single reference to memory to fetch the operand. In register indirect addressing mode, the data to be operated is available inside a memory location, and that memory location is indirectly specified by a register pair.

The advantage of this addressing mode is that it requires fewer bits to select a register than to explicitly specifying a memory address in the instruction

1.3.1.6 Auto-increment or Auto-decrement Mode

Auto increment or auto decrement mode is similar to register indirect mode, but this register is incremented or decremented after or before its value is accessed from the memory.

Consider the following example instructions:

- Add R1, (R2)→ This instruction is explained below.
- R1 = R1 +M[R2] Contents of R1 is the sum of content of R1 + content at the memory location specified by R2
- R2 = R2 + d In this step, R2 is incremented by a distance d to a new register.

The major difference in this mode is that the register is incremented or decremented after or before its value is accessed from the memory. When the address stored in the register refers to a table of memory locations, it is necessary to increment or decrement the register after every access. It is possible to perform this by incrementing and decrementing the register address using this increment or decrement instruction. In short, Effective Address of the Operand = Content of Register

1.3.1.7 Indirect Addressing Mode

In indirect addressing mode, the address field of instruction contains the address of the effective address.

Consider the example of an add instruction ADD @200H will increment the value stored in the accumulator by the value stored at the memory address location 200H.

AC ← AC + [[200H]]

LOAD R1, @1005

This example loads the operand to register R1. For that, it fetches the address stored at the memory location 1005 and fetches the operand from that address location to load the register R1. In this mode, (**the address field of the instruction gives the address**

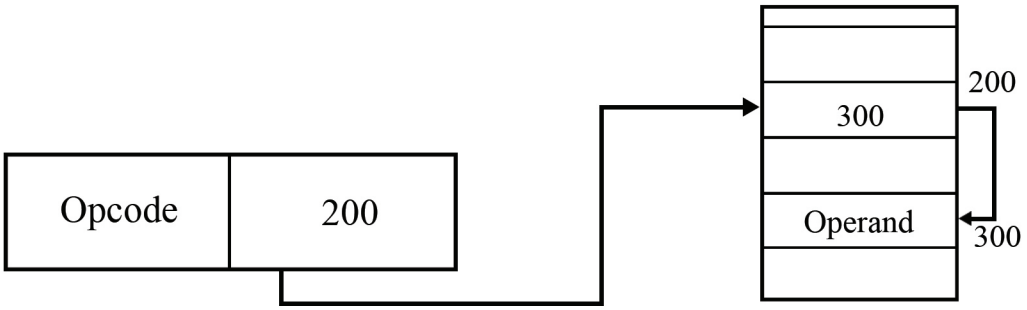


Fig 1.3.6 Indirect Addressing Mode

where the effective address is stored in memory), as shown in Fig 1.3.6. During execution, the processor fetches the instruction from memory and uses it as the address part to access memory again to read the operand.

This is the mode of addressing the instruction that contains the address of the location where the target address is stored. In short, here, the instruction is indirectly storing the address of the target location in another memory location, which is called the Indirect Addressing mode.

1.3.1.8 Indexed Addressing Mode

In indexed addressing mode, the offset of the operand is the sum of the content of an index register SI or DI and an 8-bit or 16-bit displacement.

See the following register transfer example

Load R2, A

Load R3, (R2)

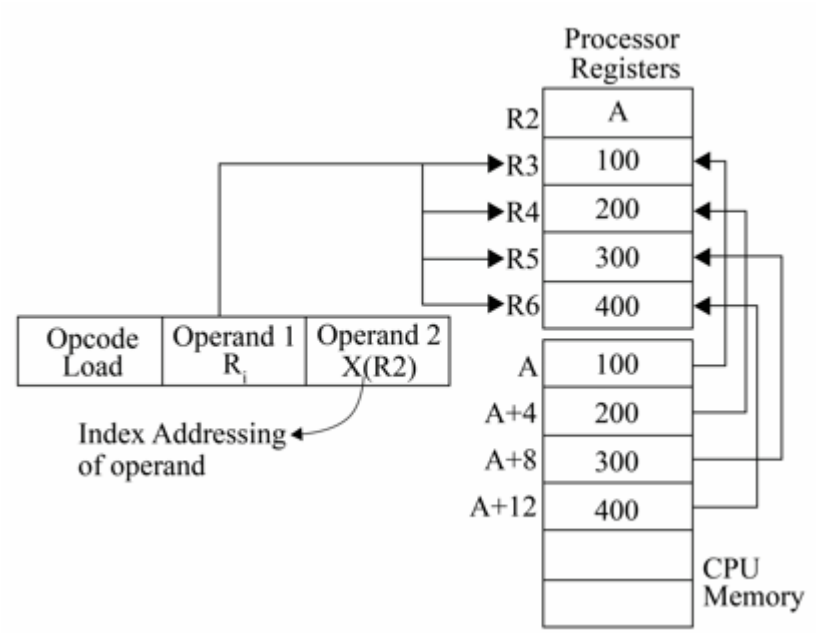


Fig. 1.3.7 Indexed Addressing Mode

Load R4, 4(R2)

Load R5, 8(R2)

Load R6, 12(R2)

The instructions above will load the Registers R3, R4, R5 and R6 along with the contents that are present at succesive memory addresses corresponding from the address location of A.

Consider the example given below where Base = 2800h. Details are given in the fig. 1.3.7. Symbolic representation of indexed addressing mode can be denoted as X(R) where Effective address, EA = X+(R). Where X is the index register.

In this mode, the content of an index register is added to the address part of the instruction to obtain the effective address. The address part of the instruction holds the beginning/ base address and is called the base. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. The base remains the same, but the index changes.

For the example given above, 2800H is the base address, and the operand is 9F. We can see that for the given index, 01H, the effective address becomes 2801H (operand F3). As shown in the fig. 1.3.7. for the index 04H, the effective address is 2804 with an operand value of 59.

eg: MOV AX, [SI + 05], or ADD AX, [BX + SI]

1.3.2 Advantages of Addressing Modes

To give programmers to facilities such as

- including pointers to the memory addresses
- counters for loop controls
- indexing of data
- program relocation.
- to reduce the number of bits in the addressing field of the Instruction
- furthermore, it offers assembly language programmers the opportunity to write programs that are more efficient in terms of the number of instructions and execution time, thereby providing increased flexibility.

Recap

- The value of the operand is directly given in: Immediate addressing mode.
- The value of the address of the operand is specified in : Direct addressing mode
- Given a base address based on the indexing address, the operand is accessed in the Indexed addressing mode
- Register is used to specify any or both of the operand addresses in register addressing mode.
- Indirect addressing mode: the data section in the instruction is the address of the operand in the memory.
- Addressing modes
- Advantages of using addressing modes
- Different types of addressing modes

  a. Implicit addressing modes

  b. Immediate addressing modes

  c. Register addressing modes

  d. Register indirect addressing modes

  e. Indirect addressing modes

  f. Auto-increment and Auto-decrement addressing modes

Objective Type Questions

1. Which addressing mode does the instruction INCA belong to?
2. Which addressing mode holds the NOP instruction?
3. What are the supported operands in implicit addressing mode?
4. MVI belongs to which addressing mode
5. MOV R1, 03H belongs to which addressing mode
6. MOV R1, [3000] belongs to which addressing mode

7. MOV CX, BX: What operation does this instruction do?
8. MOV CX,BX; Which addressing mode does this instruction belongs to?
9. ADD R belongs to which addressing mode
10. ADD instruction belongs to which addressing mode
11. ADD R1,(R2)+ belongs to which addressing mode
12. SUB R1,(R2)- belongs to which addressing mode
13. Which addressing mode computes the effective address of the Operand given the base address and the index value?
14. Which addressing mode does this instruction MOV AX, [BX] belongs to?

Answers to Objective Type Questions

1. Implicit
2. Implicit
3. No operand
4. Immediate
5. Immediate
6. Direct
7. Move the contents of the BX register into CX
8. Register direct mode
9. Register Indirect mode
10. Implicit mode
11. Auto-increment mode
12. Auto-decrement mode
13. Index addressing mode
14. Register indirect mode

Assignments

1. What is the need for addressing mode in computer architecture?
2. Explain different addressing modes with examples
3. What is the difference between implicit addressing mode and Immediate addressing mode?
4. What is the difference between register addressing mode and register indirect addressing mode?
5. Explain Auto-increment and Auto-decrement addressing mode.
6. Explain the indirect addressing mode.
7. Assume you are accessing an array stored in memory using indexed addressing mode. Write an instruction (in a hypothetical assembly language) that retrieves the value from the 3rd index of an array starting at memory location 1000h. Explain how the indexed addressing mode works in this scenario.
8. Explain the concept of relative addressing mode and its use in branching instructions (e.g., JMP, CALL). Consider an instruction JMP 2000h. If the program counter (PC) is currently at address 1500h, calculate the new value of the PC if relative addressing mode is used. Why is this addressing mode particularly useful in modern processors?

Suggested Reading

1. M.Morris Mano-Computer Systems Architecture, Third Edition, Pearson Education
2. Kai Hwang and F A Briggs-Computer Architecture and parallel processing, McGraw Hills,1990
3. Carl Hamacher -Computer Organization, Fifth Edition, Tata McGraw Hill.
4. John P Hayes -Computer Architecture Organization -Mc Graw Hill
5. William Stallings-Computer Organization and Architecture, Seventh Edition, Pearson



UNIT 4 Program Control

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- discuss the need for program control instructions
- familiarise the process of instruction execution
- recall the concept and types of program control
- narrate the details of the available program control instructions

Prerequisites

Computer programs are essential components of any computer system. However, the computer can only understand machine instructions, which are in a binary format with only zeros and ones. However, as it is difficult for humans to write programs in binary format, high-level languages like Java, C++, etc., are generally used and written in human-readable format. The programs written in these high-level languages are then converted into machine instructions. The processor is expected to execute this set of instructions to perform the desired operations. It is the responsibility of the CPU to execute these instructions. The basic operation of the CPU is termed the instruction cycle, which is a four-step process involving fetch-decode-execute and store. The instruction cycle could also be understood as the time required for the CPU to execute one instruction involving those four steps. These program instructions are usually stored in the main memory, which is termed Random Access Memory or RAM. This memory could be considered as a matrix or array of cells arranged continuously. Each cell has a unique address attached to it. The address of the cells will be in an incremental fashion. Hence, the program instructions loaded into the memory will also have the address details attached to them. The processor initiates the program execution by fetching the program instructions one after the other from the main memory. CPU executes these instructions by repetitively involving the four steps in the instruction cycle. Each step in the instruction cycle needs a certain number of machine cycles. The current instruction being executed is held in the instruction register. The program counter keeps track of the execution of the program. It contains the memory address of the next instruction to be fetched and executed.

The programs are stored in the memory in continuous location with incremental addresses in sequential order. For the sequential program execution for ordinary instructions, the program counter has to increment the location address it holds to fetch the next instruction. However, there will be various situations like an interruption or some particular conditions where the execution has to take a non-sequential mode. It involves certain operations that are to be skipped or terminated based on the requirement. It marks the need for the processor to transfer control of instruction execution.

Keywords

Program control instruction, Conditional and unconditional instructions, Halting Instructions, Interrupt instruction

Discussion

1.4.1 Program Control Instructions

In computers, program instructions are always stored in consecutive memory locations. These instructions are fetched from successive memory locations for the program execution. Instructions can be primarily classified into three types: data transfer instructions, data manipulation instructions, and program control instructions. Data transfer and data manipulation instructions deal with criteria regarding data processing operations.

The program counter keeps track of the next instructions to be executed. After every successful fetch from the memory, the program counter is incremented by 1. This incremented value ensures that the program counter points to the address of the next consecutive instruction in the memory. After completing the execution of the data transfer and data manipulation instruction, the program control and the program counter will return to the fetch cycle.

7	6	5	4	3	2	1	0	Bit Number
S	Z	X	AC	X	P	X	CS	Status Flag

X	Undefined Bits
CS	Carry Status
P	Parity Status
AC	Auxilary Carry Status
Z	Zero Status
S	Sign Status

Fig 1.4.1 Program Status Word

The program control instructions specify the conditions or details to be satisfied to change the content of the program counter and, hence, the execution of the program instructions. The program control instructions control the flow of program execution

and are capable of branching to different program segments. An interrupt/break in the instruction execution happens with the change in the content of the program counter.

Conditional Branching uses various flags in the program status word. Program Status Word (PSW) is a register that encapsulates information about the basic execution state of a program at any instant. It is a register that performs the function of a status register. It holds the program status flags that the processor can access. It holds the status of condition flags for mathematical operations like zero status, carry status, parity status, etc. An example of a PSW is shown in Fig. 1.4.1. The processor uses them to determine whether or not to execute conditional instructions. An example structure of a PSW (processor intel 8081) is shown in Fig. 1.4.1. The actual structure of a PSW is a design consideration of the processor.

Program Control Instructions are the machine code used to change the sequential flow of execution in a processor as instructed by a user. A program control instruction changes the address value in the PC and alters the normal flow of execution. There are different types of Program Control Instructions:

1. ADD R1, R2 ; Add the contents of the registers R1 and R2.
2. SZA; Skip the next instruction if Auxiliary Carry is zero.
3. SKP ; Unconditional Skip instruction-skips next instruction unconditionally Branch (BR) and Jump (JMP) instructions are commonly used in PCI.

Table 1.4.1 Examples of the program control instructions

Program Control Instructions	
Name	Mnemonics
Branch	BR
Jump	JMP
Skip	SKP
Call	Call
Return	RET
Compare (by Subtraction)	CMP
Test (by ANDing)	TST

Branch, skip, and jump types have conditional and unconditional branching instructions. Skip instructions are used to skip the next instruction execution. It can be used in both conditional and unconditional PCI. It does not have an address field.

1.4.2 Unconditional Branch Instruction

It causes an unconditional change of execution sequence to a new location.

Consider the following example instructions:

1. JUMP L2; L2 is the label to which control of the program moves.
2. MOV R3, R1 go to L2 ; After moving the contents of Register1 to register 2, change the control of program execution to Label L2
3. L2: ADD R1, 23H

In the above example, the first instruction is an unconditional jump to label L2. The second instruction to be executed will be the add instruction to add 23H to the contents of R1. The instructions MOV R3, R1 will not be executed. Consider the example

1. BR ADR
2. ....
3. ....
4. ....
5. ADR:

In the above example, the instruction BR ADR is executed, and the control is unconditionally transferred to the ADR.

Consider another example

1. MVI 10H ; move 10H into the accumulator
2. ADD 5H ; Add 5H with the contents of the accumulator
3. SKP ; Skip Next instruction execution.
4. ADD 15H ; Add 5H with the contents of the accumulator
5. SUBB 2H; subtract 2H from the accumulator with borrow, if any.

The workings of the above example are as follows: First instruction moves the hexadecimal value (10H) into the accumulator AX. In the above example, the instruction ADD 15H will be skipped. This is an immediate addressing mode instruction, and the accumulator details are implied. The second instruction adds hexadecimal 5 to the contents of the accumulator. Hence, the accumulator has 15H. The next instruction is skip instruction. It skips the next instruction from execution. In this example, the ADD instruction is used to add 15H to the accumulator. The processor continues to subtract 2H from the accumulator with borrow, if any.

JMP, BR, SKP, etc., are examples of unconditional branch instructions.

The branch and jump are one-address instructions. It is represented as BR ADR and JUMP ADR, as shown in the example. ADR is a mnemonic for an address. The branch instruction transfers the value of ADR into the program counter. The branch and jump instructions are interchangeably used in similar contexts to change the control of the execution. However, they mostly denote different addressing modes.

1.4.3 Compare Instruction

Consider the example:

1. CMP R1, R2 ; Compare the contents of the register R1 and R2
2. JE address1; jump to label address1, if the comparison result in zero flag is 0
3. .....
4. .....
5. address1:
6. ADD 5H;

The above example is an example of a conditional jump. The above example compares the contents of the registers R1 and R2. The compare instruction performs an arithmetic subtraction of two operands. However, the result of the operation is not saved; instead, the status bit or the flags in PSW are set in accordance with the result. Comparison instruction sets the status flags of PSW. This type of instruction is usually followed by a branch instruction to conditionally check the status bit and perform a branching.

1.4.4 Conditional Branch Instruction

A conditional branch instruction checks for the value of bits in PSW (processor status word) to decide on the branch decision. If the test turns positive, the execution proceeds with the transfer of control of execution to the given address. Status bits for checking the condition for branching will be different for each conditional branch instruction. If the conditional test turns positive, the control of execution is changed by transferring the effective address to the program counter. PC← Effective Address. If the conditional test turns negative, the normal sequential execution of the program continues.

PC ← PC+1. Below is a list of Conditional Branch instructions; the letter 'N' stands for NOT. Table 1.4.2 shows various available conditional branching instructions.

Table 1.4.2 Conditional Branch Program Control Instructions

Program Control Instructions		
Name	Mnemonics	Test Condition
Branch if zero	BZ	Z=1
Branch if not zero	BNZ	Z=0
Branch if carry	BC	C=1
Branch if no carry	BNC	C=0
Branch if minus	BN	N=1
Branch if plus	BNN	N=0
Branch if overflow	BV	V=1
Branch if no overflow	BNV	V=1

An example of the conditional branch is given below.

1. SUBB 05H; Subtract 05H from the accumulator or AX register.

2. BC address1; jump to label address1, if the subtraction result in carry flag=1, otherwise
3. continue the execution of moving 04H to R1.
4. MOV R1, 04H; move 04h to R1 register, R1=R1+04H
5. .....
6. .....
7. address1:
8. ADD 5H; add 05 to accumulator A=A+5H

A few examples of skip conditional program control instructions are given below.

1. SKI: Skip next instruction on Input Flag
2. SKO : Skip next instruction on Input Flag
3. SPA : Skip next instruction if AC is positive
4. SNA : Skip next instruction if AC is negative
5. SZA : Skip next instruction if AC is zero
6. SZE : Skip next instruction if E is zero

1.4.5 Subroutines

A subroutine is a program fragment which is used to perform a well-defined task. It is invoked by another user program and returns control to the calling program when finished. When we CALL a subroutine, it transfers control to a specific position. At the end of the subroutine, the RETURN instruction is executed, and this causes the execution to return to the next instruction from where it was called.

Example: CALL and RET

1.4.6 Halting Instructions

Consider the following examples:

1. ADD R1, 02H ; Add 02H to the contents of register R1
2. NOP: Do no operation
3. MOV R1 ; Move the contents of R1 to accumulator
4. HALT: The execution comes to a halt and continues in an idle state
5. ADD R2, 02: This statement will not be executed.

The first instruction of the above example performs the addition operation. 02H is added to the contents of R1, and the result is in R1. The second instruction is an NOP, or a no-operation instruction, where the processor does nothing in that instruction cycle. NOP works in implied addressing mode. There is no operand in this instruction. The third instruction moves the contents of register R1 to the accumulator. This is an immediate addressing mode, where we assume the accumulator as default. The fourth

instruction is HALT. The processor's operation halts with this instruction. HALT also involves no operand, which is an implied addressing mode.

NOP Instruction – NOP is no operation. It causes no change in the processor state other than an advancement of the program counter. It can be used to synchronise timing or introduce waits.

HALT – It brings the processor to a halt condition such that it would remain in an idle state until a valid interrupt, trace, reset, or external action occurs. In case of an interrupt, it executes the corresponding interrupt service subroutine corresponding to the interrupt number. The processor then continues with the execution of instruction after the HLT instruction. So, it will not effectively enter a halt state in that situation. However, most HLT instruction is used to terminate the program.

1.4.7 Interrupt Instructions:

Examples of Interrupt instructions are RESET, TRAP, INTR, etc.

An interrupt is a mechanism by which an I/O or an instruction can suspend the normal execution of a processor and get itself serviced. Details of the above Interrupt instruction are given below.

1. RESET – It reset the processor. This may include any or all setting registers to an initial value or setting the program counter to the standard starting location.
2. TRAP – It is a non-maskable edge and level-triggered interrupt. TRAP has the highest priority and vectored interrupt.
3. INTR – It is a level-triggered and maskable interrupt. It has the lowest priority. It can be disabled by resetting the processor.

Recap

- Types of program control instructions

  (a) Conditional program control instructions

  (b) Unconditional program control instructions

  (c) Subroutines

  (d) Halting Instructions

  (e) Interrupt Instructions
- Examples of conditional program control instructions- JUMP, BR, SKP
- Examples of unconditional program control instructions- JE, BN, BZ, SKI
- Examples of Subroutines program control instructions- CALL, RET

- Examples of Halting program control instructions- NOP, HALT
- Examples of Interrupt program control instructions- INTR, TRAP, RESET
- Example of conditional skip instructions: SKI, SZA, SZE, SPA, etc.

Objective Type Questions

1. Which type of program control instructions does JMP belong to?
2. BR belongs to which type of program control instructions?
3. JE belongs to which type of program control instructions?
4. BNZ belongs to which type of program control instructions?
5. Which type of program control instructions does NOP belong to?
6. Which type of program control instructions does HALT belong to?
7. CALL belongs to which type of program control instructions?
8. RET belongs to which type of program control instructions?
9. RESET belongs to which type of program control instructions?
10. TRAP belongs to which type of program control instructions?
11. Which type of program control instructions does INTR belong to?
12. Which instruction is the commonly used instruction to set the flags for conditional checking?
13. Which instruction is used both in conditional and unconditional program control instruction for skipping an instruction?

Answers to Objective Type Questions

1. Unconditional
2. Unconditional
3. Conditional

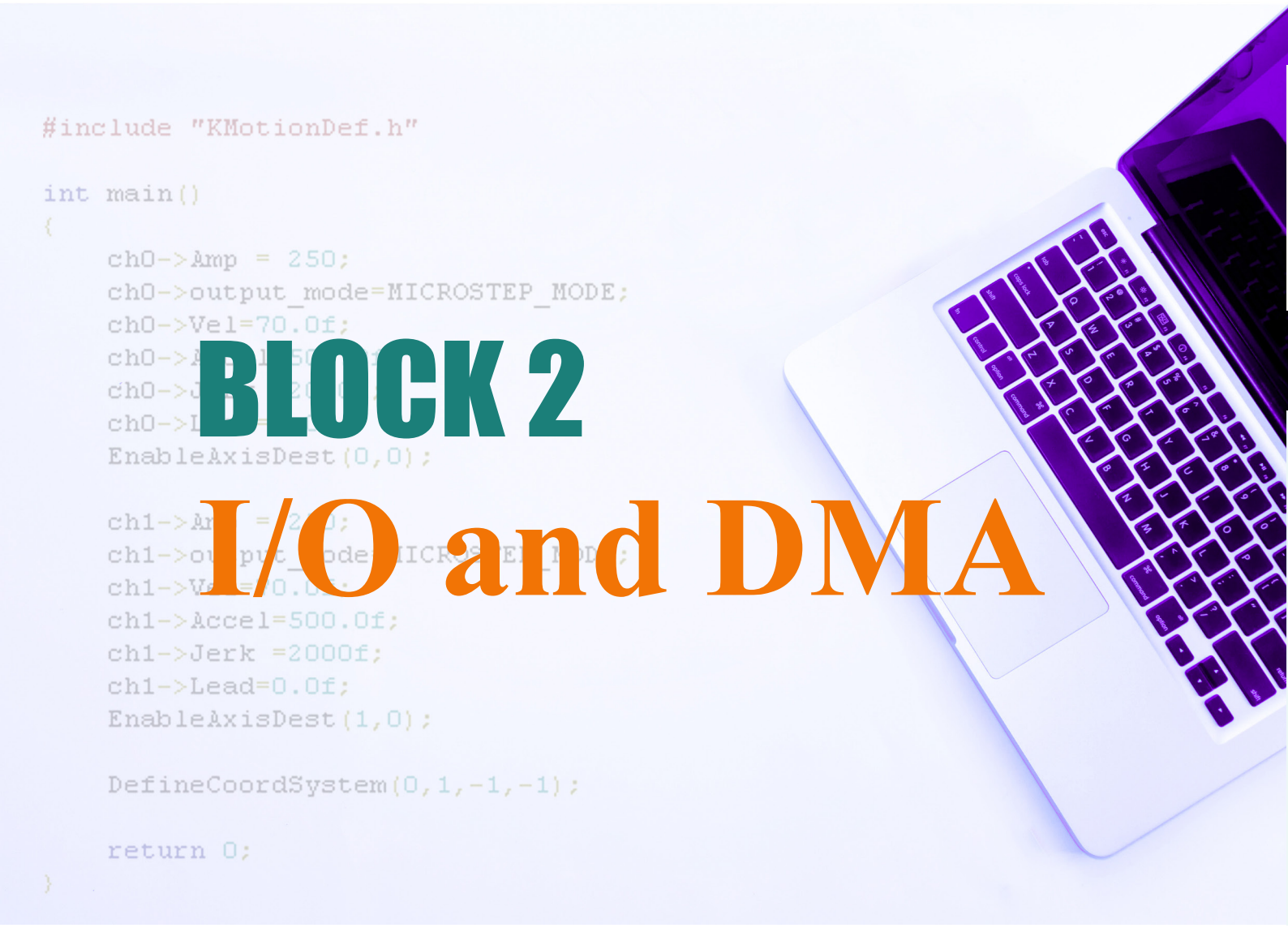
4. Conditional
5. Halting
6. Halting
7. Subroutines
8. Subroutines
9. Interrupt
10. Interrupt
11. Interrupt
12. CMP
13. SKIP

Assignments

1. Write a note on Program Control Statements.
2. Write different Program Control Statements.
3. Explain Interrupt handling mechanism.
4. Explain unconditional branch instructions.
5. Write notes on comparing instructions.
6. What are subroutines? Write different subroutine instructions.
7. What are branching instructions?

Suggested Reading

1. M.Morris Mano-Computer Systems Architecture, Third Edition, Pearson Education
2. Kai Hwang and F A Briggs-Computer Architecture and parallel processing, McGraw Hills,1990
3. Carl Hamacher -Computer Organization, Fifth Edition, Tata McGraw Hill.
4. John P Hayes -Computer Architecture Organization -Mc Graw Hill
5. William Stallings-Computer Organization and Architecture, Seventh Edition, Pearson



BLOCK 2

I/O and DMA



Register Transfer Languages

Learning Outcomes

By the completion of this unit, the learner will be able to:

- understand the importance of RTL in describing the operations of a digital system at a low level
- describe how data is transferred between registers using RTL notation
- aware of the symbols of register transfers

Prerequisites

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity, from a few integrated circuits to a complex set of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components like registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called microoperations. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. We need a suitable symbology to describe the transfer of data between registers. Register transfer languages are used for this.

Keyterms

Registers, Register transfer languages, register transfer notation, flip-flops

Discussion

2.1.1 Introduction to Register Transfer Language

Register Transfer Language (RTL) is a low level language used to describe the functioning of digital circuits. It is used in digital circuits and computer architecture to describe the operations, data flow, and transfer of data between registers within a computer system. RTL is used to express the behaviour of digital circuits, particularly during the design and implementation phases of hardware. It is a kind of intermediate representation (IR) that is very close to assembly language.

The internal hardware organisation of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of microoperations performed on the binary information stored in the registers.
3. The control that initiates the sequence of microoperations.

You can describe the sequence of microoperations in a computer by explaining each step, but this method tends to be long and detailed. It is easier to use symbols to represent the transfer of data between registers and the arithmetic and logic operations linked to these transfers. Using symbols instead of words offers a clear and concise way to outline the sequences of microoperations and the control functions that trigger them.

The symbolic notation used to describe the microoperation transfer among registers is called a register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Similarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing the microoperation sequences among the registers of a digital module in symbolic form. It is a convenient tool for describing the internal organisation of digital computers in a concise and precise manner. It can also be used to facilitate the design process of digital systems.

2.1.2 Key Concepts in Register Transfer Language

2.1.2.1 Registers

A register is a small, fast storage location within the CPU or other digital logic that holds data temporarily. Registers hold data that is to be processed or transferred. They are central to RTL as they are the primary storage locations that the language manipulates.

Example: R1, R2, R3 could be registered in a computer system.

2.1.2.2 Transfer

Transfer refers to the movement or copying of data from one register to another or to

a different part of the system. Transfers are the primary operations described in RTL, representing the flow of data between different storage elements.

Example: A transfer operation can be represented as R1 ← R2, meaning the content of register R2 is copied to register R1.

2.1.2.3 Operations

Arithmetic or logical operations are performed on data within registers. RTL allows for the representation of various operations, such as addition, subtraction, AND, OR, etc., which are fundamental to processing data.

Example: R3 ← R1 + R2 implies that the contents of registers R1 and R2 are added, and the result is stored in register R3.

2.1.2.4 Control Signals

Signals are used to control the timing and execution of operations and data transfers. Control signals determine when a particular transfer or operation should occur, often based on the state of the system or the fulfilment of certain conditions.

Example: if (Enable) then R1 ← R2.

When the "Enable" signal is active, it allows data transfer or processing to occur; when it is inactive, it prevents the operation.

2.1.2.5 Conditional Transfers

Transfers that occur only if certain conditions are met. Conditional transfers enable more complex and conditional operations within a system, allowing for decision-making processes to be represented.

Example: if (C = 1) then R1 ← R2, meaning that the transfer from register R2 to register R1 only happens if condition C is true.

2.1.2.6 Sequential and Parallel Operations

Sequential Operations: Operations that occur one after the other, in a specific order , with the output of one task serving as the input for the next. e.g., R1 ← R2; R3 ← R1.

Parallel Operations: Operations that occur simultaneously, e.g., R1 ← R2 and R3 ← R4 in parallel.

RTL can represent both sequential and parallel data flows, providing flexibility in describing how a system processes data.

Add content on:

I. Understanding Register transfer examples

a. Accumulator- This is the most commonly used register for storing data read from memory

b. General purpose registers: used to store data on intermediate outcomes during a program execution

c. Special purpose registers: users do not have access to this register. These belong to computer system registers

1. MAR - memory address registers that store memory units address

2. MBR - stores the instruction and data sent and received from the memory

3. PC - program counter holds the address of the next command to be executed

4. IR - stores the current instruction to be executed

2.1.2.7 Example of RTL Statement

Consider a simple example where two registers, R1 and R2, hold values, and we want to add these values and store the result in another register R3. The RTL description might look like this:

R3 ← R1 + R2

This statement means that the contents of R1 and R2 are added together, and the result is stored in R3.

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name MAR. Other designations for registers are PC (for program counter), IR (for instruction register, and R1 (for processor register). A register is a storage device made up of flip-flops that hold binary data (0 or 1); each bit requires one flip-flop for storage. The individual flip-flops in an n-bit register are numbered in sequence from 0 through n - 1, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 2.1.1 shows the representation of registers in block diagram form.

The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 2.1.1 (a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c).

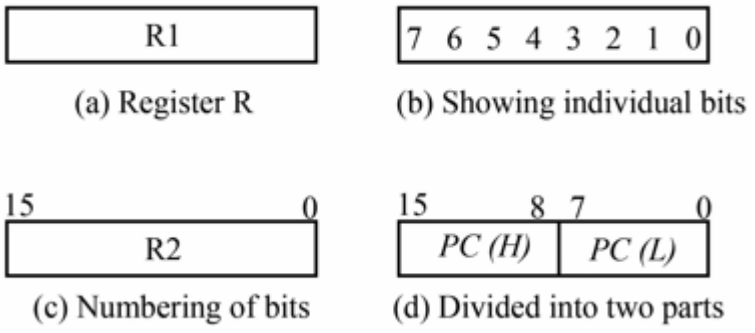


Fig 2.1.1 Block diagram of registers

The 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte), and bits 8 through 15 are assigned the symbol H (for high byte). For example, the figure shows the PC as a 16-bit register. The symbol PC(0-7) or PC(L) refers to the low-order byte, and PC(8-15) or PC (H) to the high-order byte.

Information transfer (MOVE) from one register to another is designated in symbolic form by means of a replacement operator. The statement

$$R2 \leftarrow R1$$

denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R 1 . By definition, the content of the source register R1 does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Parallel load allows multiple bits of data to be loaded simultaneously rather than sequentially. Normally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

$$If\ (P = 1)\ then\ (R2 \leftarrow R1)$$

where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a

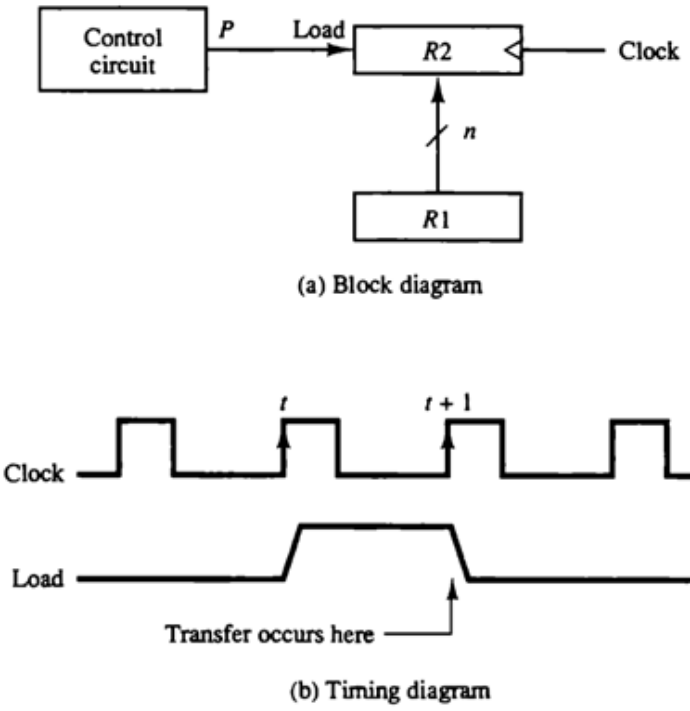


Fig 2.1.2 Transfer from R1 to R2 when P =1

control function. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

$$P: \quad R2 \leftarrow R1$$

The control condition is terminated with a colon. It symbolises the requirement that the transfer operation is executed by the hardware only if P = 1. Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 2.1.2 shows the block diagram that depicts the transfer from R1 to R2.

The n outputs of register R1 are connected to the n inputs of register R2. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register R2 has a load input that is activated by the control variable P. It is assumed that the control variable is synchronised with the same clock as the one applied to the register.

As shown in the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t. The next positive transition of the clock at time t + 1 finds the load input active, and the data inputs of R2 are then loaded into the register in parallel. P may go back to 0 at time t + 1; otherwise, the transfer will occur with every clock pulse transition while P remains active.

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition, such as P, becomes active just after time t, the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time t + 1.

The basic symbols of the register transfer notation are listed in Fig. 2.1.3. Registers are denoted by capital letters, and numerals may follow the letters. MAR holds the address of the memory unit. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

$$T: \quad R2 \leftarrow R1, \quad R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

2.1.4 Symbols used in Register Transfer Language

Table 2.1.1 Basic symbols for register transfer

Symbol	Description	Example
Letters or combination of letters and numbers	Denotes registers	R1, R2, A, B

Arrow ←	Denotes data transfer	R1 ← R2, The contents of register R2 are transferred to register R1.
Parentheses ()	Denotes a part of a register	R2(0-7), R(l)
Comma ,	Separates two micro operations	R2 ← R2, R1 ← R2
Arithmetic operators +, -, *, /	Perform arithmetic operations on registers	R3 ← R1 + R2, Add the contents of R1 and R2, and store the result in R3.
Logical Operations ^ or AND, v or OR, ⇁ or NOT, ⊕ or XOR	Perform logical operations on registers	R3 ← R1 ∨ R2 , Perform a bitwise OR operation between R1 and R2, storing the result in R3.
Shift operator <<, >>	Perform left shift and right shift operations on registers	R1 ← R1 << 1, Shift the contents of R1 left by one bit.
Conditional operator, if() <,>,<=,>=, ==, !=	Perform left shift and right shift operations on registers	if (C = 1) then R1 ← R2 Meaning: If the condition C = 1, is true, then the contents of R2 are transferred to R1.
Control signals Enable, Load, Clear.	The transfer operation is controlled by the signal Enable. If Enable is active (true), the transfer occurs.	if (Enable) then R1 ← R2, The transfer operation is controlled by the signal Enable. If Enable is active (true), the transfer occurs.
Bus Operations BUS	The contents of the bus are transferred	R1 ← BUS , The contents of the bus are transferred into R1.
Memory access M[Address]	Perform memory operations	R1 ← M[100], Load the contents of memory location 100 into register R1

2.1.3 Applications of RTL

2.1.3.1 Hardware Design:

RTL is widely used in the design and simulation of digital circuits, such as microprocessors, memory units, and other hardware components.

2.1.3.2 Logic Synthesis:

RTL descriptions are often converted into actual hardware through a process called logic synthesis, where the RTL code is transformed into a gate-level representation.

2.1.3.3 Microarchitecture Specification:

RTL is used to specify the detailed design of the internal structure of CPUs and other processing units, including how data flows through the system.

2.1.4 Bus and Memory Transfers

An efficient method for transferring data between registers in a multi-register setup is the Common Bus System. This system includes a set of shared lines, with each line corresponding to a bit in a register. Control signals dictate which register is selected by the bus during each specific transfer. A Common Bus System can be constructed using different approaches, such as with multiplexers or tri-state buffers.

A common bus system using multiplexers works by selecting the source register, placing its binary data onto the bus. Figure 2.1.3 illustrates the construction of a bus system designed for four registers.

The bus system is composed of four 4-to-1 multiplexers, each with four data inputs labelled 0 through 3 and two selection inputs, S1 and S0. For instance, the first output of register A is linked to input 0 of MUX 1 because this input is labelled A1. As illustrated in Figure 2.1.3, the corresponding bits from each register are connected to the

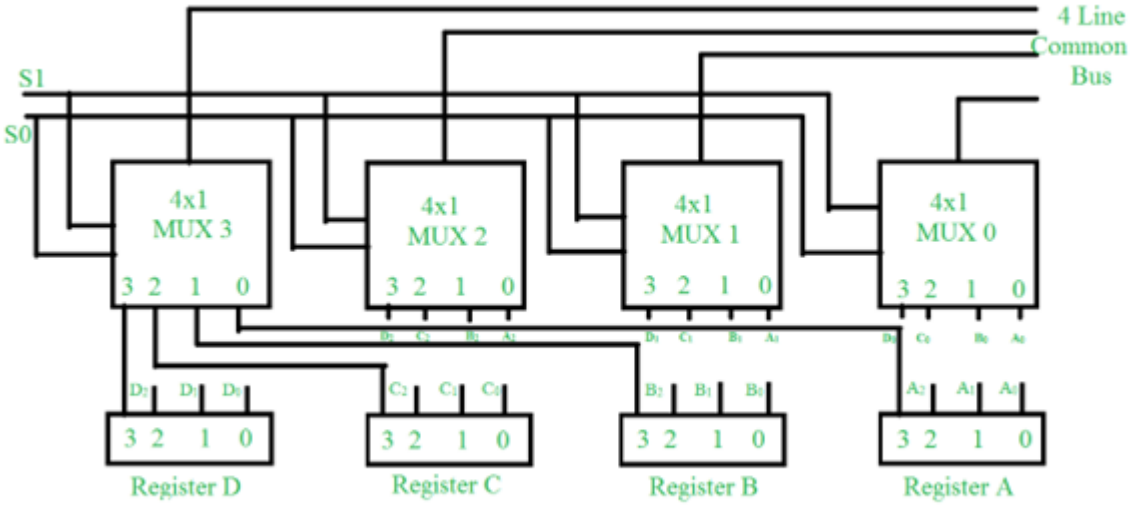


Fig 2.1.3 Computer Architecture and Organization

S0	S1	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

data inputs of a single multiplexer, forming one line of the bus. Specifically, MUX 0 handles the four 0 bits from the registers, MUX 1 manages the four 1 bits, and the same applies to the remaining bits. The selection lines S1 and S0 are connected to the selection inputs of all four multiplexers. These lines determine which register's four bits are selected and transmitted through the four-line common bus. When S1S0 = 00, the 0 data inputs of all multiplexers are selected and sent to the outputs, which make up the bus. As a result, the bus lines carry the content of register A since its outputs are connected to the 0 data inputs of the multiplexers. Similarly, if S1S0 = 01, register B is selected, and so forth.

Table shows the register that is selected by the bus for each of the four possible binary values of the selection lines.

In general a bus system has multiplex "k" Registers each register of "n" bits to produce "n-line bus"

no. of multiplexers required = n

size of each multiplexer = k x 1

When the bus is included in the statement, the register transfer is symbolized as follows: BUS← C, R1← BUS

The content of register C is placed on the bus, and the content of the bus is loaded into register R1 by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer. R1← C

2.1.5 Micro Operations

Micro-operations, or micro-ops, are the fundamental operations that occur within a CPU to execute machine-level instructions. They represent the smallest unit of processing that a CPU can perform and are the building blocks of more complex operations at the instruction level. Understanding micro-operations is crucial for grasping how a CPU processes instructions at a granular level.

2.1.5.1 Execution Cycle of Micro-operations:

The execution cycle of micro-operations refers to the sequence of steps the CPU follows to execute each machine-level instruction. This cycle consists of several stages, during

which various micro-operations are performed to process the instruction from start to finish. The execution cycle typically includes the following stages:

1. Fetch Cycle: Retrieve the instruction from memory.

MAR ← PC: The memory address register (MAR) is loaded with the contents of the program counter (PC), which holds the address of the next instruction.

MBR ← Memory[MAR]: The memory buffer register (MBR) is loaded with the instruction located at the address in MAR.

IR ← MBR: The instruction register (IR) is loaded with the instruction from the MBR.

PC ← PC + 1: The program counter (PC) is incremented to point to the next instruction.

2. Decode Cycle: Interpret the fetched instruction to determine the required actions.

Decode IR: The control unit decodes the instruction in the instruction register (IR) to understand what operations need to be performed, including identifying the opcode and operands.

3. Fetch Operands Cycle (if necessary): Retrieve the operands (data) needed to execute the instruction.

MAR ← Effective Address (EA): The effective address (EA) of the operand is calculated and stored in the MAR.

MBR ← Memory[MAR]: The operand is read from memory into the MBR.

R1 ← Register: The operand is directly loaded from a register into another register or an internal CPU bus.

4. Execution Cycle: Perform the operation specified by the instruction.

For an addition operation:

R1 ← R2 + R3: The contents of registers R2 and R3 are added, and the result is stored in register R1.

For a logical operation:

R1 ← R2 AND R3: The contents of registers R2 and R3 are logically ANDed, and the result is stored in R1.

For a branch operation:

PC ← EA: The program counter (PC) is updated to the effective address (EA) for a conditional or unconditional branch.

5. Memory Access Cycle (if necessary): Access memory to read or write data (not always needed).

For a read operation:

MAR ← EA: The effective address is placed in the MAR.

MBR ← Memory[MAR]: Data from the memory address in MAR is read into the MBR.

For a write operation:

MAR ← EA: The effective address is placed in the MAR.

Memory[MAR] ← MBR: Data from the MBR is written to the memory location specified by the MAR.

6. Write-back Cycle: Store the result of the execution back into a register or memory.

R1 ← Result: The result of the operation is stored in a specified register (e.g., R1).

Memory[MAR] ← MBR: If the result needs to be stored in memory, it is written from the MBR to the memory location specified by the MAR.

7. Interrupt Handling Cycle (if an interrupt occurs): Handle any interrupts that may have occurred during execution.

Save the current state: The current state of the CPU (PC, flags, etc.) is saved to allow resuming later.

Transfer control: The CPU jumps to the interrupt service routine (ISR) to handle the interrupt.

Restore state: After handling the interrupt, the CPU restores the saved state and resumes normal operation.

2.1.5.2 Types of Micro-operations:

a. Data Transfer Operations: Moving data from one register to another, such as R1 ← R2, where the content of register R2 is transferred to register R1.

b. Arithmetic Operations: Performing basic arithmetic, such as addition or subtraction, e.g., R1 ← R2 + R3, where the sum of R2 and R3 is stored in R1.

c. Logical Operations: Executing logical functions like AND, OR, XOR, etc., e.g., R1 ← R2 AND R3.

d. Shift Operations: Shifting the bits in a register left or right, e.g., R1 ← SHR(R2), where the contents of R2 are shifted right by one bit and stored in R1.

Recap

- Introduction to Register Transfer Language
- Key Concepts in Register Transfer Language
  - Register
  - Transfer
  - Operations
  - Control signals
  - Conditional Transfers
  - Sequential and Parallel Operations
- Symbols used in Register Transfer Language
- Applications of RTL
  - Hardware design
  - Logic synthesis
  - Micro architecture specification
- Bus and Memory Transfers
- Micro Operations
  - Execution Cycle of micro-operations
  - Types of Micro operations

Objective Type Questions

1. What does RTL stand for?
2. Which language is the symbolic notation used to describe the microoperation transfers among registers?
3. Which micro-operations carry information from one register to another?
4. In which transfer is the computer register indicated in capital letters to depict its function?
5. Which register holds an address for the memory unit?

6. Which register is designated as MAR in RTL?
7. Which register is designated as PC in RTL?
8. Write the micro-operation syntax for performing the sum of R2 and R3 is stored in R1.
9. Different applications of RTL
10. Write the names of different types of micro-operations

Answers to Objective Type Questions

1. Register Transfer Language
2. Register Transfer Language
3. Register transfer
4. Register transfer
5. Memory Address Register
6. Memory Address Register
7. Program Counter
8. R1 ← R2 + R3
9. Hardware design, Logic synthesis, Microarchitecture Specification
10. Data transfer operations, Arithmetic Operations, Logical operations, Shift operations.

Assignments

1. Explain Register Transfer Language
2. What are the key concepts in Register Transfer Language
3. Explain different symbols used in Register Transfer Language
4. What are the Applications of RTL

5. Explain RTL in Bus and Memory Transfers
6. What are Micro Operations? Explain different types of Micro operations
7. Explain execution cycle of micro operations

Suggested Reading

1. M.Morris Mano-Computer Systems Architecture, Third Edition, Pearson Education

References

1. Kai Hwang and F A Briggs-Computer Architecture and parallel processing, McGraw Hills,1990
2. Abraham Silberschatz, Peter Galvin and Greg Gagne - Operating System Principles, Seventh Edition, John Wiley
3. William Stallings - Operating Systems, Sixth Edition, Prentice Hall of India, Pearson
4. Carl Hamacher -Computer Organization, Fifth Edition, Tata McGraw Hill.
5. John P Hayes -Computer Architecture & Organization–McGraw Hill
6. William Stallings-Computer Organization and Architecture, Seventh Edition, Pearson Education
7. Milan Kovic - Operating Systems, Second Edition, TMH



Input-Output Organization

Learning Outcomes

By the completion of this unit, the learner will be able to:

- know about peripheral devices.
- familiarise the concepts of I/O Bus and Interfacing Modules.
- provide a basic introduction to the memory-mapped I/O.

Prerequisites

The computer is designed to perform operations based on the given input and produce a well-structured and defined output. Fundamentally, a well-organized input-output system is essential. The input can be any value, whether meaningful or meaningless, structured or unstructured, and the computer must be able to process it effectively. Therefore, a well-defined input system is necessary. Additionally, to ensure that the output is meaningful and meets the desired criteria, a well-defined output system is also required.

The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing, and results obtained from computations must be recorded or displayed for the user. A computer serves no useful purpose without the ability to receive information from an outside source and to translate results in a meaningful form. The most familiar means of entering connotation into a computer is through a typewriter-like keyboard that allows a person to enter alphanumeric information directly. Every time a key is depressed, the terminal sends a binary-coded character to the computer. The fastest possible speed for entering information this way depends on the person's typing speed.

Keywords

Peripheral Devices, ASCII Characters, Interface Unit, I/O Command, Control Command

Discussion

2.2.1 Peripheral Devices

The central processing unit is an extremely fast device capable of performing operations at very high speed. When input information is transferred to the processor via a slow keyboard, the processor will be idle most of the time while waiting for the information to arrive. To use a computer efficiently, a large amount of programs and data must be prepared in advance and transmitted into a storage medium such as magnetic tapes or disks. The information in the disk is then transferred into computer memory at a rapid rate. Results of programs are also transferred into high-speed storage, such as disks, from which they can be transferred later into a printer to provide a printed output of results. Devices that are under the direct control of the computer are said to be connected on-line. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be part of the total computer system. Input or output devices attached to the computer are also called peripherals. Among the most common peripherals are keyboards, display units, and printers. Peripherals that provide auxiliary storage for the system are magnetic disks and tapes. Peripherals are electromechanical and electromagnetic devices of some complexity. Only a very brief discussion of their function will be given here without going into detail about their internal construction.

The input-output (I/O) organisation of a computer system refers to the mechanisms and structures through which a computer interacts with the outside world, including both receiving input data and sending output data. The difference between a small and a large system is mostly dependent on the amount of hardware the computer has available for communicating with peripheral units and the number of peripherals connected to the system. Since each peripheral behaves differently from any other, it would be prohibitive to dwell on the detailed interconnections needed between the computer and each peripheral.

2.2.2 ASCII Alphanumeric Characters

Input and output devices that communicate with people and the computer are usually involved in the transfer of alphanumeric information to and from the device and the computer. The standard binary code for the alphanumeric characters is ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters; the seven bits of the code are designated by b1 through b7, with b7 being the most significant bit. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 characters that can be printed and 34 nonprinting characters used for various control functions. The printing characters consist of the 26 uppercase letters A through Z, the 26 lowercase letters a-z, the 10 numerals 0 through 9, and 32 printable special characters such as %, *, $, etc

2.2.3 Input-Output Interface

The Input-Output (I/O) Interface is a crucial component in computer systems, and it

is responsible for managing the communication between the CPU (Central Processing Unit) and peripheral devices like keyboards, displays, storage devices, and network interfaces. The I/O interface ensures that data is correctly transferred between the computer and external devices, facilitating smooth operation and interaction. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central processor and each peripheral.

The major differences are:

1. Peripherals are electromechanical and electromagnetic devices, and their manner of operation is different from that of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronisation mechanism may be needed.

3. Data codes and formats in peripherals differ from the word format in the CPU and memory.

4. The operating modes of peripherals are different from each other, and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronise all input and output transfers. These components are called **interface units** because they interface between the processor bus and the peripheral device. Each device may have its own controller that supervises the operations of the particular mechanism in the peripheral.

2.2.3.1 Functions of I/O Interface:

The I/O interface acts as a mediator between the CPU and peripheral devices, allowing them to communicate effectively despite differences in speed, data formats, and operational requirements. Different functions of I/O interfaces are

- **Data Transfer:** Facilitates the transfer of data between the CPU and I/O devices.
- **Control Signal Management:** Manages control signals that coordinate the operation of I/O devices.
- **Data Conversion:** Converts data formats between the CPU and the I/O devices, ensuring compatibility.
- **Synchronization:** Ensures that data transfer is synchronised between the CPU and I/O devices, preventing data loss or corruption.

2.2.3.2 Types of I/O Interfaces:

The types of I/O (Input/Output) interfaces describe how peripheral devices connect

and communicate with a computer's CPU and memory. Each type of I/O interface has specific characteristics that make it suitable for different applications. Below are the main types of I/O interfaces

- **Memory-Mapped I/O:** In memory-mapped I/O, peripheral devices are assigned specific memory addresses in the system's address space. The CPU uses standard memory instructions to communicate with I/O devices, treating them as if they were part of the main memory. It simplifies the CPU design by using the same instructions for both memory and I/O operations, thereby allowing larger and more flexible addressing of I/O devices. It may reduce available address space for actual memory. It can lead to more complex decoding logic. It is often used in systems where I/O devices need to be accessed quickly and efficiently, such as in embedded systems.

- **Isolated I/O:** Isolated I/O, also known as port-mapped I/O, uses a separate address space specifically for I/O devices, distinct from the memory address space. Special instructions (like IN and OUT in x86 assembly) are used to access these I/O ports. Keeps I/O and memory operations separate, avoiding potential address space conflicts. It is easier to implement in systems where memory and I/O need to be distinctly managed. It requires special instructions for I/O operations, which can increase instruction set complexity. It typically has a smaller address space compared to memory-mapped I/O.

- **Programmed I/O (Polling):** In programmed I/O, the CPU actively checks (or polls) the status of an I/O device to determine if it is ready for data transfer. The CPU directly controls the data transfer by executing instructions to read from or write to the I/O device. It is simple to implement and control, and there is no need for complex hardware mechanisms. However, it is inefficient CPU usage, as the CPU must wait (poll) for the I/O device to be ready, leading to wasted processing power. Also, it is not suitable for high-performance applications. It is commonly used in simple, low-speed, or real-time systems where the CPU can afford to dedicated time for managing I/O devices directly.

- **Interrupt-Driven I/O:** In interrupt-driven I/O, the CPU is interrupted by an I/O device when it is ready to perform a data transfer. The CPU can perform other tasks while waiting for the I/O device and only processes the I/O when the interrupt occurs. It has more efficient CPU utilisation compared to programmed I/O, as the CPU is free to execute other tasks until the I/O device is ready. It reduces CPU idle time and improves overall system performance. However, it requires a more complex control mechanism to handle interrupts, and interrupt handling can add overhead, especially in systems with many I/O devices. It is widely used in multitasking environments, operating systems, and scenarios where multiple I/O devices need to be managed efficiently.

- **Direct Memory Access (DMA):** Direct Memory Access allows I/O devices to transfer data directly to or from memory without involving the CPU for each byte of data. A DMA controller manages the data transfer, freeing the CPU for other tasks. It is extremely efficient for large data transfers, as the CPU is not involved in the actual data transfer process. It reduces CPU overhead and improves system performance. However, it requires additional

hardware (DMA controller) and a more complex system design. Potential for data corruption if the CPU and DMA controller attempt to access the same memory simultaneously without proper synchronisation. It is used in high-speed data transfer applications, such as disk drives, network interfaces, and multimedia devices.

- **Synchronous I/O**: In synchronous I/O, the CPU waits for an I/O operation to complete before continuing with the next instruction. The CPU and the I/O device are synchronised. Typically used in situations where data consistency and timing are critical, such as in real-time systems.

- **Asynchronous I/O**: In asynchronous I/O, the CPU continues executing other instructions while the I/O operation is in progress. The CPU checks the status of the I/O operation periodically or is notified by an interrupt when the operation is complete. Common in modern operating systems to handle I/O operations without stalling the CPU, improving overall efficiency.

- **Serial I/O**: Data is transmitted one bit at a time over a single communication line. Serial communication is slower but requires fewer wires. Commonly used in long-distance communication and devices like serial ports (RS-232) and USB.

- **Parallel I/O**: Multiple bits are transmitted simultaneously over multiple communication lines. Parallel communication is faster but requires more wires. Used in scenarios where speed is critical and distance is short, such as in internal data buses and older printer connections.

2.2.3.3 I/O Interface Components:

The I/O (Input/Output) interface components are essential for facilitating communication between the CPU and peripheral devices. These components ensure that data can be transferred, controlled, and synchronised effectively across the different parts of a computer system. Here are the key components of an I/O interface:

- Data Bus: Transfers data between the CPU, memory, and I/O devices.

- Address Bus: Carries the address of the I/O device or memory location where data is to be read or written.

- Control Bus: Carries control signals that manage the operations of the I/O devices (e.g., read/write signals).

- I/O Ports: Physical interfaces through which the CPU communicates with I/O devices, often represented as I/O registers.

- I/O Controllers: I/O controllers act as intermediaries between the CPU and peripheral devices, managing data flow, buffering, and converting signals as needed.

- Buffers and Latches: Buffers and latches are temporary storage elements that hold data during transfer processes.

- Interrupt Lines: Interrupt lines are used by I/O devices to signal the CPU that

they require attention, such as when they are ready for data transfer or when an error occurs.

- Clock and Timing Circuits: Clock and timing circuits synchronise the operations of the CPU, memory, and I/O devices, ensuring that data transfer occurs at the correct times.
- Control Logic: The control logic within an I/O interface is responsible for interpreting control signals from the CPU and managing the corresponding operations on the I/O device.
- Power Management Circuits: Power management circuits ensure that I/O devices receive the appropriate power levels and can manage power states (e.g., sleep or active) to conserve energy.

2.2.4 IO Bus and Interface Modules

A typical communication link between the processor and several peripherals is shown in Fig. 2.2.1.

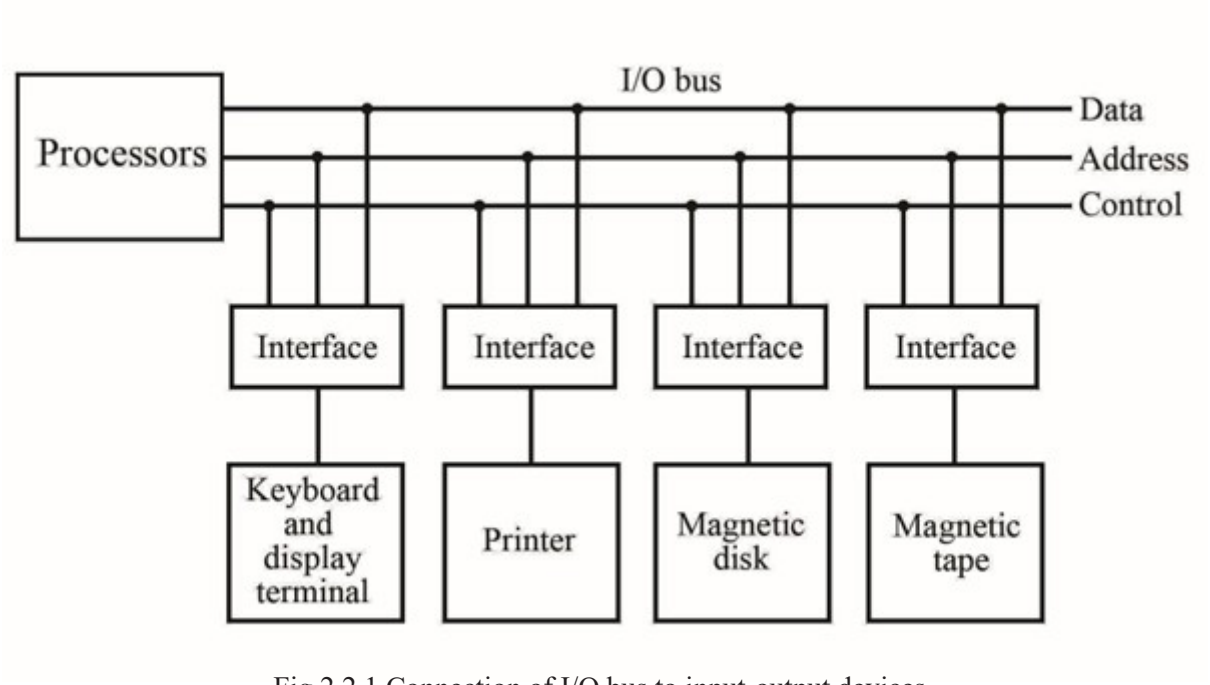


Fig 2.2.1 Connection of I/O bus to input-output devices

The I/0 bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device is associated with an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronises the data flow and supervises the transfer between the peripheral and the processor. Each peripheral has a controller that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters. A controller may be housed separately or may be physically integrated with the peripheral. The I/O bus from the processor is attached to a peripheral interface. To communicate with

a particular device, the processor places a device address on the address lines. Each interface connected to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interface. At the same time that the address is made available in the address lines, the processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command which is an instruction that is executed in the interface and its attached peripheral unit. The interpretation of the command depends on the peripheral that the processor is addressing.

There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.

- A **control command** is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction. The particular control command issued depends on the peripheral, and each peripheral receives its distinguished sequence of control commands depending on its mode of operation.

- A **status command** is used to test various status conditions in the interface and the peripheral. For example, the processor may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur, which are detected by the interface. These errors are designated by setting bits in a status register that the processor can read at certain intervals.

- A **data output command** causes the interface to respond by transferring data from the bus into one of its registers. Consider an example with a tape unit. The computer starts the tape moving by issuing a control command. The processor then monitors the status of the tape by means of a status command. When the tape is in the correct position, the processor issues a data output command. The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register. The interface then communicates with the tape controller and sends the data to be stored on tape.

- **The data input command** is the opposite of the data output. In this case, the interface receives an item of data from the peripheral and places it in its buffer register. The processor checks if data is available by means of a status command and then issues a data input command. The interface places the data on the data lines, where the processor accepts them.

2.2.5 I/O versus Memory Bus

In addition to communicating with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address, and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.

2. Use one common bus for both memory and I/O but have separate control lines for each.

3. Use one common bus for memory and I/O with common control lines.

In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for IO. This is done by I/O processor in computers, that is providing a separate I/O processor (lOP) in addition to the central processing unit (CPU). The memory communicates with both the CPU and the IOP through a memory bus. The IOP also communicates with the input and output devices through a separate I/O bus with its own address, data, and control lines. The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory. The I/O processor is sometimes called a data channel.

2.2.5.1 Isolated versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and an I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or an interface register by enabling one of two possible read or write lines. The I/O read and 1/0 write control lines are enabled during an I/O transfer. The memory read and memory writes control lines are enabled during a memory transfer. This configuration isolates all I/O interface addresses from the addresses assigned to memory and is referred to as the isolated I/O method for assigning addresses in a common bus.

In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O to read (for input) or write (for output) the control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory to read or write the control line. This informs the external components that the address is for a memory word and not for an I/O interface. The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read-and-write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory-mapped I/O. The computer treats an interface register as being part of the memory system. The assigned addresses for interface registers cannot be used for memory words, which reduces the memory address range available.

In a memory-mapped I/O organisation, there are no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organised as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers. Still, in general, they can be located at any address as long as there is not also a memory word that responds to the same address. Computers with memory-mapped I/O can use memory-type instructions to access I/O data. It allows the computer to use the same instructions for either input-output transfers or memory transfers. The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers. In a typical computer, there are more memory-reference instructions than I/O instructions. With memory-mapped I/O, all instructions that refer to memory are also available for I/O.

2.2.5.2 Example of IO Interface

An example of an I/O interface unit is shown in block diagram form in Fig. 2.2.2. It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits.

The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and I/O write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface. The I/O data to and from the device can be transferred into either port A or port B. The interface may operate with an output device or with an input device that requires both input and

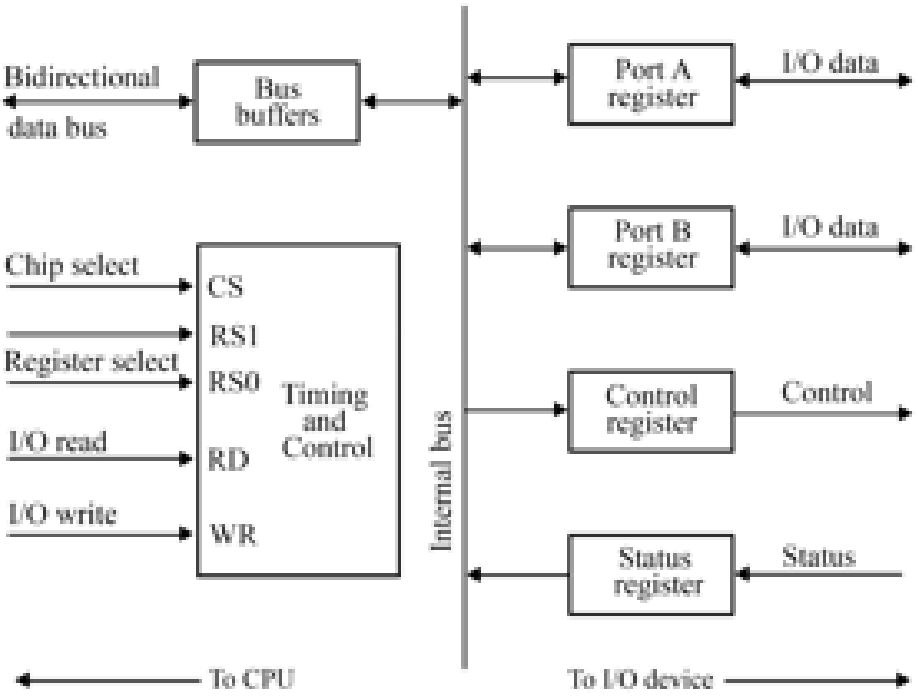


Fig 2.2.2 I/O Interface Unit

CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

Fig. 2.2.3 Example of I/O Interface unit

output. If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data.

A magnetic disk unit transfers data in both directions but not at the same time so that the interface can use bidirectional lines. A command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports A and B registers. Thus, the transfer of data, control, and status information is always done via the common data bus. The distinction between data, control, or status information is determined by the particular interface register with which the CPU communicates. The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes. The bits in the status register are used for status conditions and for recording errors that may occur during the data transfer.

For example, a status bit may indicate that port A has received a new data item from the I/O device. Another bit in the status register may indicate that a parity error has occurred during the transfer. The interface registers communicate with the CPU through the bidirectional data bus. The address bus selects the interface unit through the chip select, and the two registers select inputs. A circuit (usually a decoder) must be provided externally to detect the address assigned to the interface registers. This circuit enables the chip to select (CS) input when the interface is selected by the address bus. The two register select inputs, RSl and RS0, are usually connected to the two least significant lines of the address bus. These two inputs select one of the four registers in the interface as specified in the table accompanying the diagram. The content of the selected register is transferred into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

Recap

- Discussion on Peripheral Devices
- Understanding of ASCII Alphanumeric Characters
- Input-Output Interface details
  - Functions of I/O interfaces
  - Types of I/O interfaces
  - Components of I/O interfaces
- I/O Bus and Interface
- I/O versus Memory Bus
- Isolated versus Memory-Mapped I/O
- Example of I/O Interface

Objective Type Questions

1. How many types of modes of I/O Data Transfer?
2. Input or output devices that are connected to computers are categorised into which type of devices?
3. How do the I/O devices use the address space in memory-mapped I/O?
4. What is the usual BUS structure used to connect the I/O devices?
5. What is the advantage of I/O-mapped devices to memory-mapped devices?
6. How is the system notified of a read or write operation through the bus?
7. How can the lag in the operating speeds of the I/O device and the processor be overcome?
8. What is the method of accessing the I/O devices by repeatedly checking the status flags?
9. What is the name of the process that the processor constantly checks the status flags?

Answers to Objective Type Questions

1. 3
2. Peripheral Devices
3. Shared
4. Single BUS structure
5. Deal with fewer address lines
6. Sending a special signal
7. Status Flags
8. Program-controlled I/O
9. Polling

Assignments

1. Discussion on different peripheral devices used in Computer
2. Explain different ASCII Alphanumeric Characters used in I/O operations
3. Explain I/O Interface in detail
4. Explain types of I/O interfaces
5. What are the components of I/O interfaces
6. Explain different buses used in data transfer
7. I/O versus Memory Bus

Suggested Reading

1. M.Morris Mano-Computer Systems Architecture, Third Edition, Pearson Education.
2. Kai Hwang and F A Briggs-Computer Architecture and parallel processing, McGraw Hills,1990.
3. Carl Hamacher -Computer Organization, Fifth Edition, Tata McGraw Hill.
4. John P Hayes -Computer Architecture & Organization–McGraw Hill.



Priority Interrupts

Learning Outcomes

By the completion of this unit, the learner will be able to:

- introduce the concept of types of priority interrupts.
- identify the methods of establishing Priority in simultaneous interrupts.
- make them understand the concept of priority encoder.

Prerequisites

How do we handle rare events with unpredictable arrival times and must be acted upon quickly? E.g., keystroke, in-bound network, disk I/O

- Option 1: write every program with periodic calls to a service routine (i.e., polling); polling frequency affects worst-case response time– expensive for rare events needing fast response. What if a programmer does it wrong or forgets?
- Option 2: normal programs blissfully unaware– event triggers an interrupt on-demand – forcefully and transparently transfer control to the service routine and back

In a typical application, there could be a lot of I/O devices attached to a computer, with each device being able to originate an interrupt request, so to provide services to the device which initiates an interrupt request, the task of the interrupt system is to identify the source(device) of interrupt and then provide services to them.

However, in most cases, there is a possibility that several sources will request service simultaneously. So, in this case, the interrupt system must also decide which device to service first. Moreover, this simple interrupt system is not capable of that, so another system known as a Priority interrupt system is required.

Key Concepts

Maskable Interrupt, Exception, Polling, Daisy Chaining, Priority Encoder

Discussion

2.3.1 Priority Interrupts

Priority Interrupt is a system that establishes priority over the various sources (interrupt devices) to determine which condition is to be serviced first when two or more requests arrive simultaneously. This system may also determine which conditions permit the computer to interrupt while another interrupt is being serviced. Figure 2.3.1 shows interrupt servicing.

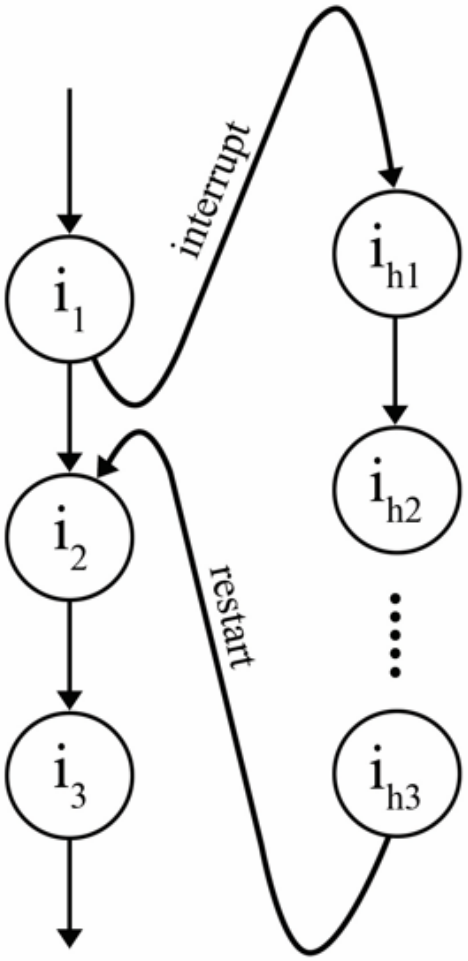


Fig 2.3.1 Servicing Interrupts

Usually, in Priority Systems, higher-priority interrupt levels are served first, as if they are delayed or interrupted, could have serious consequences. The devices with high-speed transfer, such as magnetic disks, are given high priority, and slow devices, such as keyboards, receive low priority. When two or more devices interrupt the computer

simultaneously, the computer services the device with a higher priority first.

2.3.2 Establishing Priority of Simultaneous Interrupt

The priority of simultaneous interrupts can be established either by software method or hardware.

2.3.2.1 Hardware Interrupts

When the signal for the processor is from an external device or hardware, this interrupt is known as a hardware interrupt. Let us consider an example: when we press any key on our keyboard to do some action, then this pressing of the key will generate an interrupt signal for the processor to perform certain actions. Such an interrupt can be of two types:

- **Maskable Interrupt**

Hardware interrupts can be delayed when a much higher priority interrupt occurs at the same time.

- **Non Maskable Interrupt**

The hardware interrupts cannot be delayed and should be processed by the processor immediately.

2.3.2.2 Software Interrupts

The interrupt that is caused by any internal system of the computer system is known as a software interrupt. It can also be of two types:

- **Normal Software Interrupt**

The interrupts that are caused by software instructions are called normal software interrupts.

- **Exception**

Unplanned interrupts which are produced during the execution of some programs are called exceptions, such as division by zero.

2.3.3 Polling (Software Method)

Polling involves continuously checking the status of a device to determine if it needs attention. In this method, when the processor detects an interrupt, it branches to an interrupt service routine whose job is to pull each I/O module to determine which module caused the interrupt.

The poll could be in the form of a separate command line(e.g., Test I/O). In this case, the processor raises the Test I/O and places the address of a particular I/O module on the address line. If it has an interrupt, then define it. It is the order in which they are tested, i.e., the order in which they appear on the address line (Service Routine) determines

the priority of each interrupt. While testing, the highest-priority sources (devices) are tested first, then lower-priority devices. This is a very simple method of establishing priority on simultaneous interrupts. However, the disadvantage of polling is that it is very time-consuming.

2.3.4 Daisy-Chaining Priority (Hardware Method)

This way of deciding the interrupt priority consists of the serial connection of all the devices, which generates an interrupt signal. The device with the highest priority is placed in the first position, followed by lower-priority devices, and the device with the lowest priority among all is placed at the last in the chain.

In a daisy chaining system, all the devices are connected in a serial form. The interrupt line request is common to all devices. If any device has an interrupt signal in a low-level state, then the interrupt line goes to a low-level state and enables the interrupt input in the CPU. When there is no interrupt, the interrupt line stays in a high-level state. The CPU responds to the interrupt by enabling the interrupt acknowledge line. This signal is received by device 1 at its Priority Interrupt input. The acknowledged signal passes to the next device through PO output only if device 1 is not requesting an interrupt. The

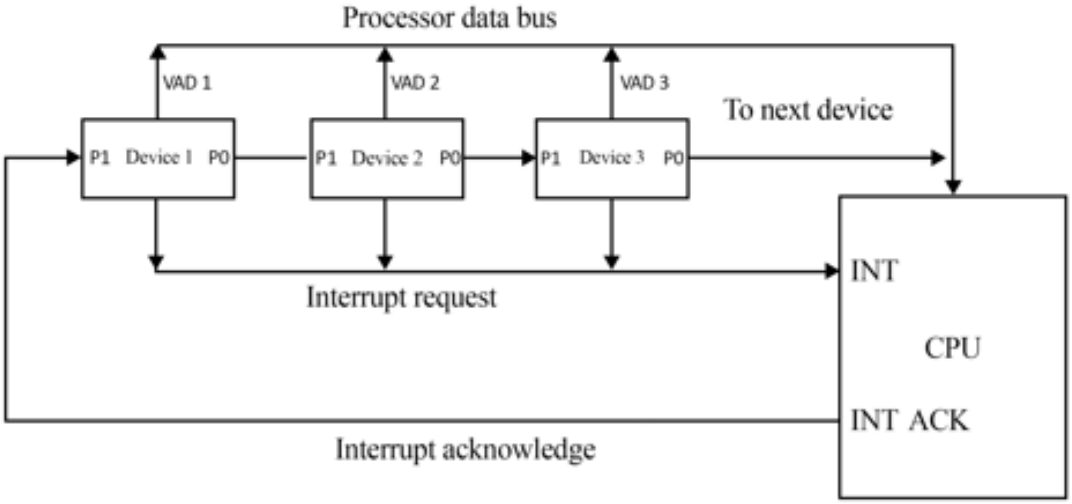


Fig 2.3.2 Daisy-Chain Priority interrupt

following figure shows the block diagram for the daisy chaining priority system. This method of connection between devices and the CPU is shown in Fig. 2.3.2.

If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the P0 output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle. A device with a 0 in its P1 input generates a 0 in its P0 output to inform the next-lower-priority device that the acknowledged signal has been blocked. A device that is requesting an interrupt and has a 1 in its P1 input will intercept the acknowledged signal by placing a 0 in its P0 output. If the device does not have pending interrupts, it transmits the acknowledged signal to the next device by placing a 1 in its P0 output. Thus, the device with P1 = 1 and P0 = 0 is the one with the highest priority that is requesting an interrupt,

and this device places its VAD on the data bus. The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The following fig. 2.3.3 shows the internal logic that must be included within each device when connected in the daisy-chaining scheme.

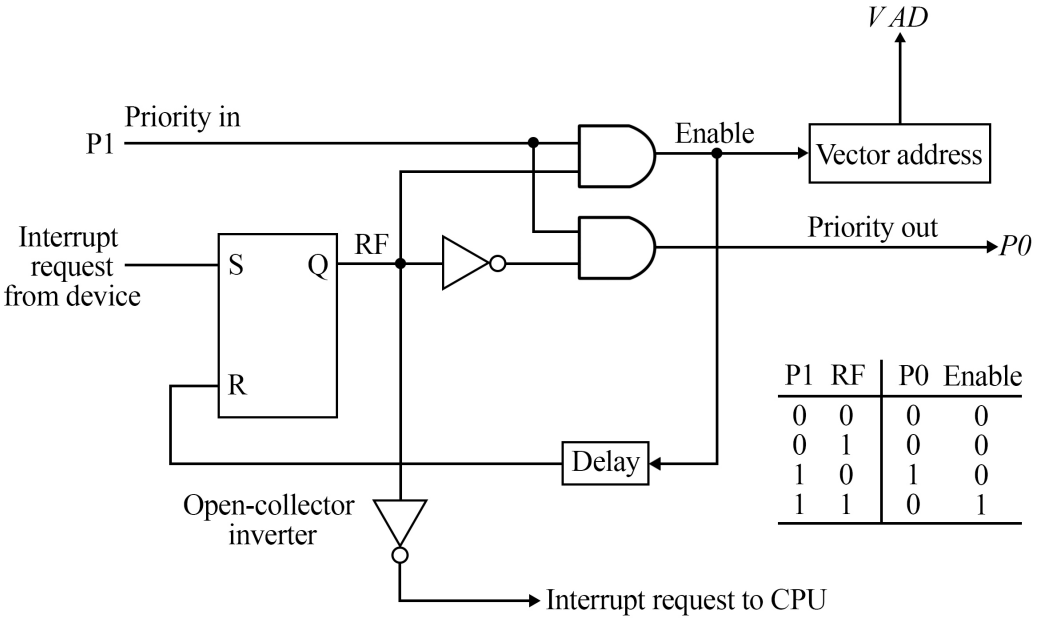


Fig. 2.3.3 One stage of the daisy-chain priority arrangement

- The device sets its RF flip-flop when it wants to interrupt the CPU.
- If P1 = 0, both P0 and the enable line to VAD are equal to 0, irrespective of the value of RF.
- If P1 = 1 and RF = 0, then P0 = 1 and the vector address is disabled. This condition passes the acknowledge signal to the next device through P0.
- The device is active when P1 = 1 and RF = 1. This condition places a 0 in P0 and enables the vector address for the data bus.
- The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

2.3.5 Parallel Priority Interrupt

The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register, the circuit may include a mask register whose purpose is to control the status of each interrupt request. The priority logic for a system of four interrupt sources is shown in Fig. 2.3.4. It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions.

The magnetic disk, being a high-speed device, is given the highest priority. The printer

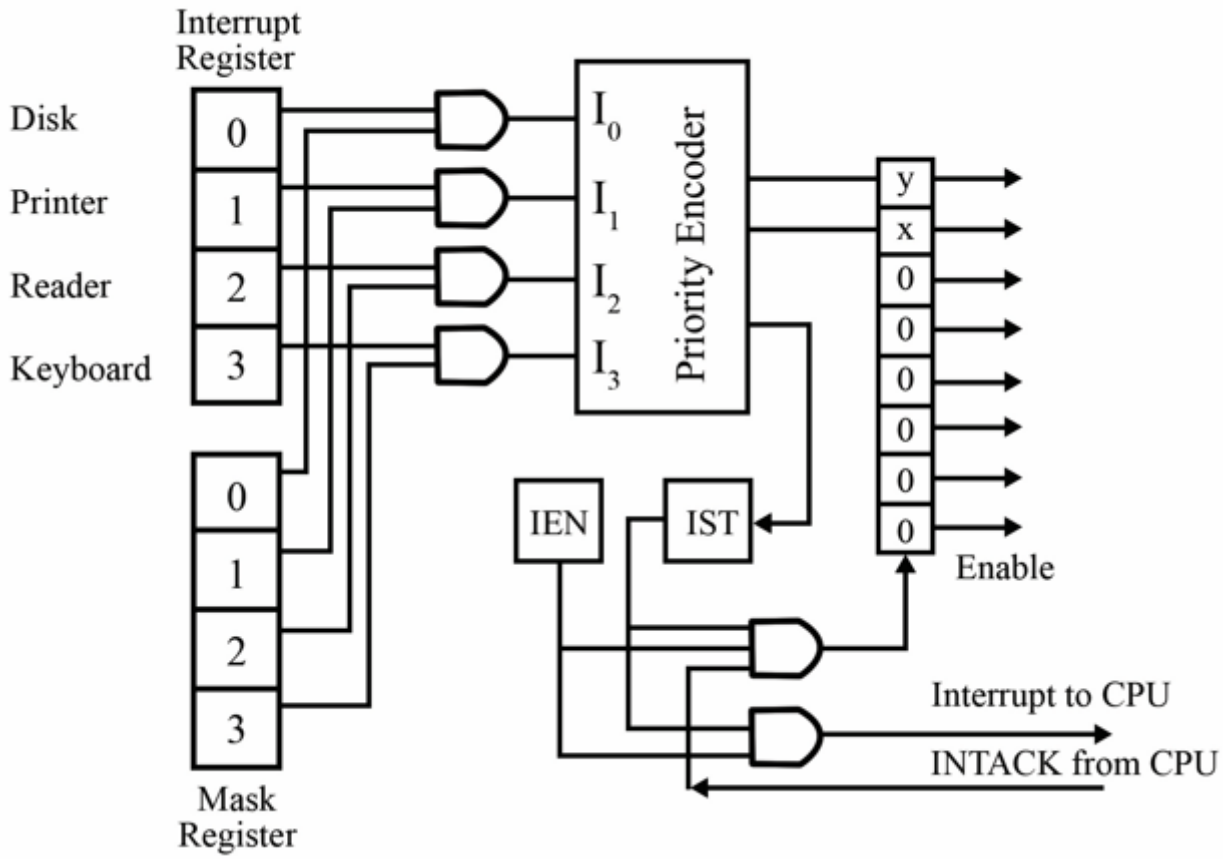


Fig. 2.3.4 Priority interrupt hardware

has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register.

By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way, an interrupt is recognised only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs. The interrupt-enabled flip-flop IEN can be set or cleared by the program to provide overall control over the interrupt system. The outputs of IST ANDed with IEN provide a common interrupt signal for the CPU. The interrupt acknowledges the INTACK signal from the CPU, enables the bus buffers in the output register, and a vector address VAD is placed into the data bus.

2.3.6 Priority Encoder

The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that if two or more inputs arrive at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority

encoder is given in Table 2.3.1.

Table 2.3.1 Priority Encoder Truth Table

Inputs				Outputs			Boolean Functions
I0	I1	I2	I3	x	y	IST	
1	x	x	x	0	0	1	$x = I'_0 I'_1$
0	1	x	x	0	1	1	$y = I'_0 I_1 + I'_0 + I'_2$
0	0	1	x	1	0	1	$IST = I_0 + I_1 + I_2 + I_3$
0	0	0	1	1	1	1	
0	0	0	0	x	x	0	

The output of the priority encoder is used to form part of the vector address for each interrupt source. Interrupt Cycle The interrupt enables flip-flop IEN, as shown in Fig. 2.3.4, which can be set or cleared by program instructions. When IEN is cleared, the CPU neglects the interrupt request coming from IST. The program-controlled IEN bit allows the programmer to choose whether to use the interrupt facility. If an instruction to clear IEN has been inserted in the program, it means that the user does not want his program to be interrupted. Instructions on setting IEN indicate that the interrupt facility will be used while the current program is running. At the end of each instruction cycle the CPU checks IEN and the interrupt signal from IST. If either is equal to 0, control continues with the next instruction.

If both IEN and IST are equal to 1, the CPU goes to an interrupt cycle.

During the interrupt cycle, the CPU performs the following sequence of microoperations.

←SP - 1: Decrement stack pointer

M[SP] ←PC: Push PC into the stack

INTACK←1: Enable interrupt to acknowledge

PC←VAD: Transfer vector address to PC

IEN←0: Disable further interrupts

Go to fetch the next instruction

2.3.7 Software Routines

A priority interrupt system is a combination of hardware and software techniques. Fig. 2.3.5. shows the programs that must reside in memory to handle the interrupt system. Each device has its own service program that can be reached through a jump (JMP) instruction stored at the assigned vector address. The symbolic name of each routine

represents the starting address of the service program. The stack shown in the diagram is used for storing the return address after each interrupt.

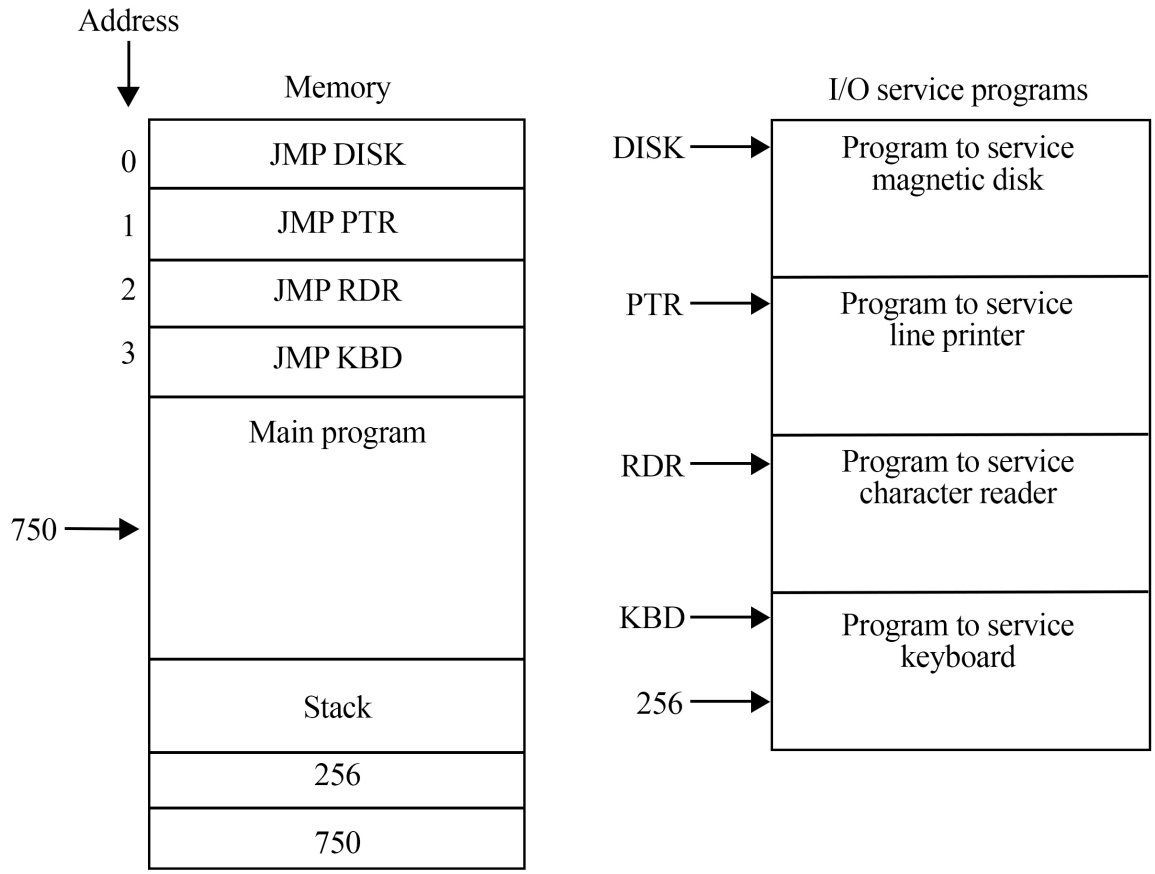


Fig. 2.3.5 Programs stored in memory for servicing interrupt

2.3.7.1 Initial(prologue) and Final(epilogue) Operations

Each interrupt service routine must have an initial and final set of operations for controlling the registers in the hardware interrupt system.

The initial sequence of each interrupt service routine (prologue) must have instructions to control the interrupt hardware in the following manner:

1. Clear lower-level mask register bits.
2. Clear interrupt status bit IST.
3. Save contents of processor registers.
4. Set interrupt enable bit IEN.
5. Proceed with service routine

The final sequence in each interrupt service routine (epilogue) must have instructions to control the interrupt hardware in the following manner:

1. Gear interrupt enable bit IEN.

2. Restore the contents of processor registers.
3. Clear the bit in the interrupt register belonging to the source that has been serviced.
4. Set lower-level priority bits in the mask register.
5. Restore the return address to the PC and set IEN.

Recap

- A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously.
- Devices with high-speed transfers, such as magnetic disks, are given high priority, and slow devices, such as keyboards, receive low priority.
- When two devices interrupt the computer at the same time, the computer services the device with the higher priority first.
- Establishing the priority of simultaneous interrupts can be done by software or hardware.
- A polling procedure is used to identify the highest-priority source by software means. In this method, there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence.
- The order in which they are tested determines the priority of each interrupt. The highest-priority source is tested first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next lower-priority source will be tested, and so on.
- The disadvantage of the software method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. In this situation, a hardware priority-interrupt unit can be used to speed up the operation.
- A hardware priority-interrupt unit accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination
- To speed up the operation, each interrupt source has its own interrupt vector to access its own service routine directly. Thus, no polling is required because all the decisions are established by the hardware priority-interrupt unit.
- The hardware priority function can be established in two ways.
    - Serial connection (daisy chaining)
    - Parallel connection

Objective Type Questions

1. Which program-controllable flipflop is in-built into a microprocessor specifically used to set or clear the program instructions?
2. Which exception is also called software interrupt?
3. Which technique is used to identify the highest priority resource by means of software?
4. Which type of interrupt establishes a priority over the various sources to determine which request should be entertained first:
5. Which method is used to establish priority by serially connecting all devices that request an interrupt:
6. In daisy chaining device 0, will pass signal only if it has?
7. Which interrupt method uses a register whose bits are set separately by interrupt signal for each device:

Answers to Objective Type Questions

1. IEN flip-flop
2. System calls
3. Polling
4. Priority interrupt
5. Daisy chaining
6. No interrupt request
7. Parallel priority interrupt

Assignments

1. Explain priority interrupt
2. Compare and contrast different interrupt handling mechanisms, focusing on priority interrupts. Discuss their advantages and disadvantages in various systems.

3. What is polling?

4. Differentiate hardware interrupt and software interrupt

5. Advantages and disadvantages of priority interrupt

6. Analyze a case study where priority interrupts played a crucial role in a system's performance. Discuss the outcomes and how priority management affected the system's efficiency.

7. Identify a real-world application (e.g., automotive systems, medical devices) that heavily relies on priority interrupts. Explain how priority management is implemented and its impact on the system.

Suggested Reading

1. M.Morris Mano-Computer Systems Architecture, Third Edition, Pearson Education

2. Kai Hwang and F A Briggs-Computer Architecture and parallel processing, McGraw Hills,1990

3. Abraham Silberschatz, Peter Galvin and Greg Gagne - Operating System Principles, Seventh Edition, John Wiley

4. William Stallings - Operating Systems, Sixth Edition, Prentice Hall of India, Pearson

5. Carl Hamacher -Computer Organization, Fifth Edition, Tata McGraw Hill.

6. John P Hayes -Computer Architecture & Organization–Mc Graw Hill

7. William Stallings-Computer Organization and Architecture, Seventh Edition, Pearson Education

8. Milan Kovic - Operating Systems, Second Edition, TMH

9. S TANENBAUM, A. N. D. R. E. W., and HERBERT BOS. "Operating Systems." (2015).



Direct Memory Access

Learning Outcomes

After completion of this unit, the learner will be able to:

- to explain the need for DMA
- to introduce DMA controller architecture
- to make the student aware of the different modes of data transfer in DMA

Prerequisites

The execution of a computer program requires the synchronous working of more than one component of a computer. For example, Processors provide necessary control information, addresses, etc, and buses to transfer information and data to and from memory to I/O devices. The interesting factor of the system is the way it handles the transfer of information among processor, memory, and I/O devices. Usually, processors control all the processes of transferring data, right from initiating the transfer to the storage of data at the destination. This adds load to the processor, and most of the time, it stays in the ideal state, thus decreasing the efficiency of the system. To speed up the transfer of data between I/O devices and memory, the DMA controller acts as a station master. The DMA controller transfers data with minimal intervention from the processor.

Keyterms

Direct Memory Access, DMA controller, Input Output Processing

Discussion

2.4.1 Direct Memory Access

Direct Memory Access (DMA) is a feature in computer systems that allows peripheral devices to transfer data directly to or from the system's main memory, bypassing the CPU. This process frees the CPU from being involved in the data transfer, allowing it to perform other tasks. This can significantly improve system performance, especially in

systems where large amounts of data need to be transferred quickly. DMA is designed to handle high-speed data transfers between memory and peripheral devices, such as disk drives, network cards, or sound cards, without burdening the CPU with the task of moving data byte by byte.

2.4.2 Components of DMA:

2.4.2.1 DMA Controller (DMAC):

The term DMA stands for direct memory access. The hardware device used for direct memory access is called the DMA controller. A DMA controller is a control unit, part of an I/O device's interface circuit, that can transfer blocks of data between I/O devices and the main memory with minimal intervention from the processor. Fig. 2.4.1. shows the architecture of the DMA controller.

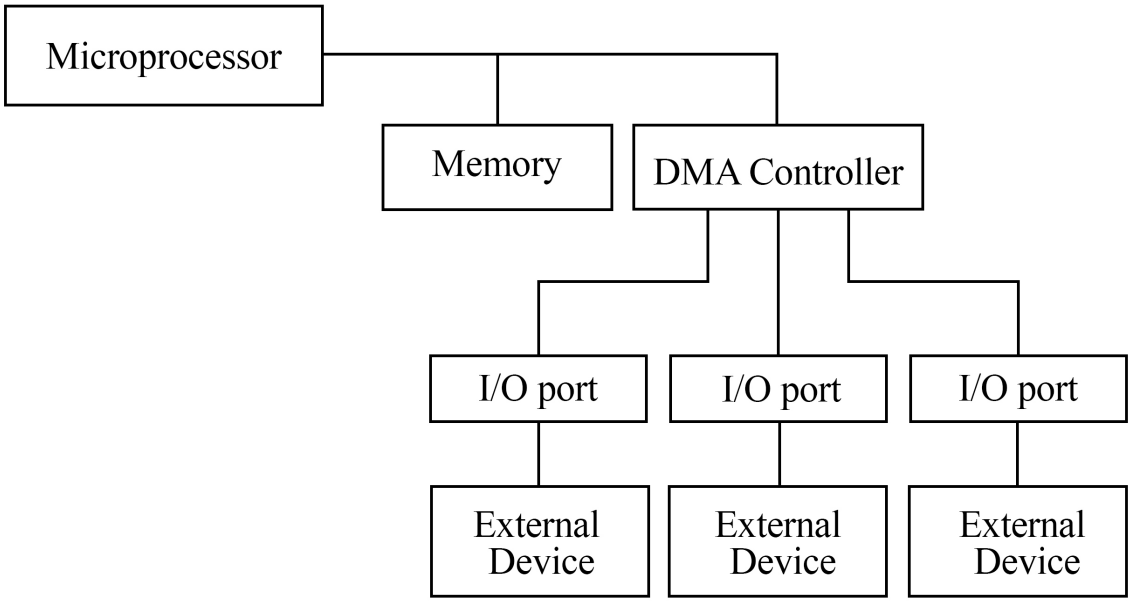


Fig 2.4.1 DMA Controller Architecture

DMA controller contains an address unit for generating addresses and selecting I/O devices for data transfer. It also contains a control unit, and a data count to keep track of the number of blocks transferred and to indicate the direction of data transfer. When the transfer is completed, the DMA informs the processor by raising an interrupt. The typical block diagram of the DMA controller is shown in Fig. 2.4.2.

Working of DMA Controller

The DMA controller has to share the bus with the processor to transfer the data. The device that holds the bus at a given time is called the bus master. When a transfer from the I/O device to the memory or vice versa has to be made, the processor stops the execution of the current program, increments the program counter, moves data over the stack, and then sends a DMA select signal to the DMA controller over the address bus.

If the DMA controller is free, it requests the control of the bus from the processor by

raising the bus request signal. The processor grants the bus to the controller by raising the bus grant signal, and now, the DMA controller is the bus master. The processor initiates the DMA controller by sending the memory addresses, number of blocks of data to be transferred and direction of data transfer. After assigning the data transfer task to

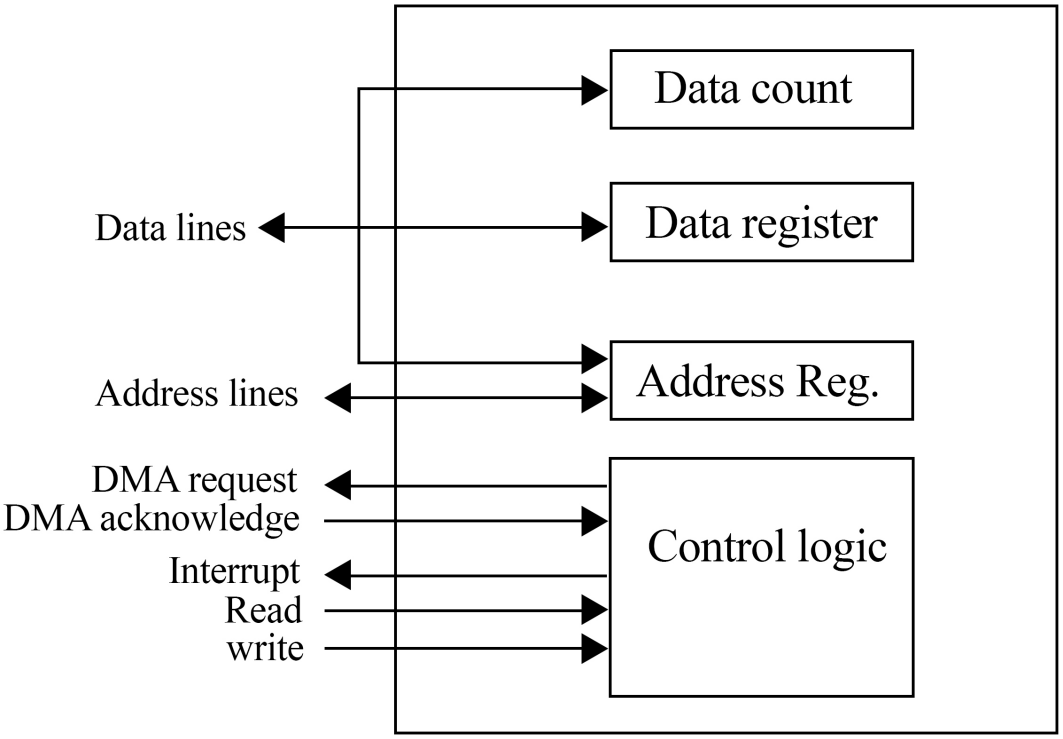


Fig 2.4.2 Block diagram of DMA Controller

the DMA controller, instead of waiting ideally till completion of data transfer, the processor resumes the execution of the program after retrieving instructions from the stack.

The DMA controller now has full control of the buses and can interact directly with memory and I/O devices independently of the CPU. It performs the data transfer based on the control instructions received from the processor. After completion of data transfer, it disables the bus request signal, and the CPU disables the bus grant signal, thereby moving control of buses to the CPU.

When an I/O device wants to initiate a transfer, it sends a DMA request signal to the DMA controller, for which the controller acknowledges if it is free. Then, the controller requests the processor for the bus, raising the bus request signal. After receiving the bus grant signal, it transfers the data from the device. For n channelled DMA controller, n number of external devices can be connected. Fig. 2.4.3 shows the transfer of data in the computer by DMA Controller.

The DMA transfers the data in three modes, which include the following.

- Burst Mode: In this mode, DMA handover the buses to the CPU only after the completion of the whole data transfer. Meanwhile, if the CPU requires the bus, it has to stay ideal and wait for data transfer.

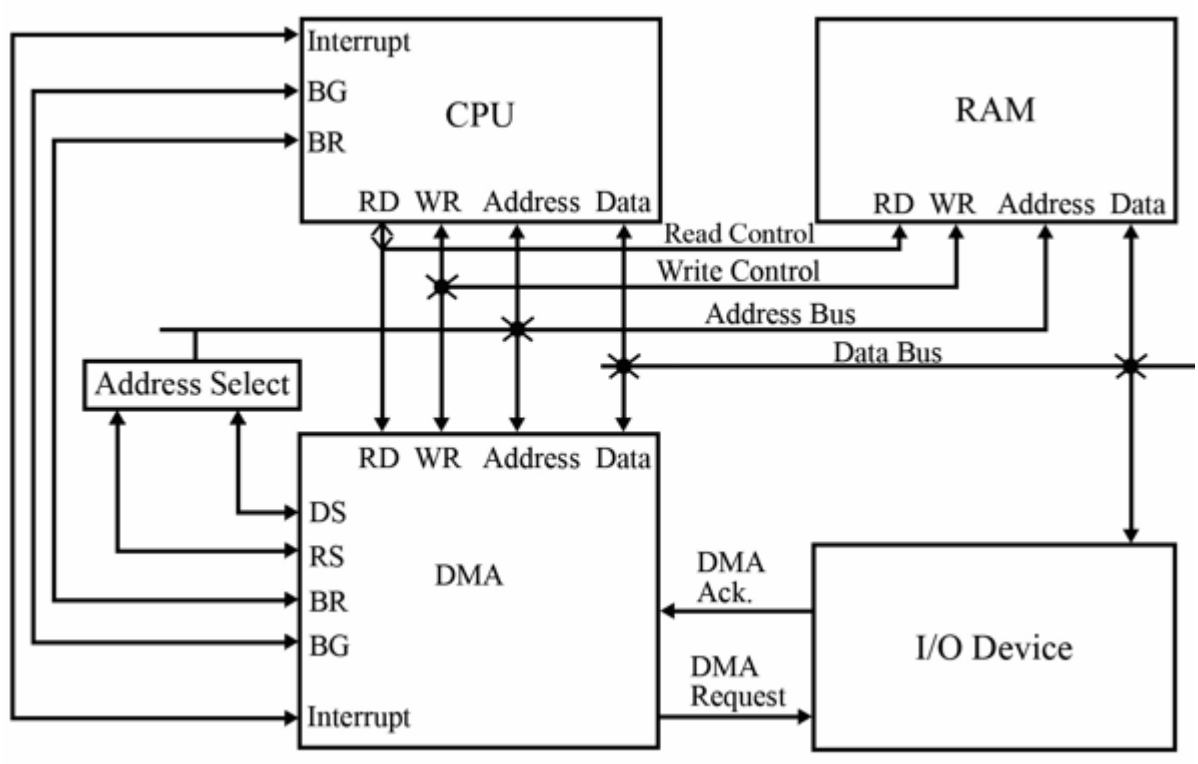


Fig 2.4.3 Data transfer by DMA

- Cycle Stealing Mode: In this mode, DMA gives control of buses to the CPU after the transfer of every byte. It continuously issues a request for bus control, makes the transfer of one byte and returns the bus. This way, the CPU doesn't have to wait for a long time if it needs a bus for a higher-priority task.
- Transparent Mode: Here, DMA transfers data only when the CPU is executing the instruction, which does not require the use of buses.

2.4.2.2 System Memory

The location in the main memory where data is transferred to or from.

2.4.2.3 I/O Device

The peripheral device either sends data to or receives data from the system memory.

2.4.3 DMA Process

a. Request: When an I/O device needs to read from or write to memory, it sends a DMA request to the DMA controller.

b. Grant: The DMA controller then requests control of the system bus from the CPU. If granted, the CPU temporarily halts its own memory access operations.

c. Transfer: The DMA controller manages the data transfer directly between the I/O

device and memory. The CPU is free to execute other instructions during this time.

d. Completion: After the data transfer is complete, the DMA controller releases control of the system bus back to the CPU and sends an interrupt signal to notify the CPU that the transfer is done.

2.4.4 Advantages of DMA

a. Efficient Data Transfer: By allowing data transfers directly between I/O devices and memory, DMA speeds up the process significantly compared to CPU-controlled data transfers.

b. CPU Offloading: DMA reduces the load on the CPU, allowing it to perform other tasks while the data transfer occurs, leading to better multitasking and overall system performance.

c. Higher Throughput: DMA can handle larger volumes of data in less time, making it ideal for high-performance systems like servers, gaming systems, and multimedia applications.

2.4.5 Disadvantages of DMA

a. Bus Contention: Since the DMA controller takes control of the system bus, there can be conflicts with the CPU or other bus masters, leading to potential performance bottlenecks.

b. Complexity: Implementing DMA requires additional hardware and a more complex system design, particularly in managing the synchronisation between DMA transfers and CPU operations.

c. Data Coherency: Ensuring that the data in memory remains consistent and preventing the CPU from accessing partially updated data during a DMA transfer can be challenging.

2.4.6 Applications of DMA

- **Disk Drives:** DMA is commonly used in hard drives and SSDs to transfer large blocks of data directly to memory.
- **Network Cards:** Network interface cards (NICs) use DMA to transfer incoming and outgoing data packets quickly.
- **Sound Cards:** DMA allows sound cards to stream audio data directly to and from memory, reducing latency in audio processing.
- **Graphics Cards:** DMA is used to transfer data between the CPU and GPU, particularly in rendering graphics or processing video.

Recap

- Direct Memory Address
- Components of DMA
- The DMA controller transfers data with minimal intervention from the processor.
- DMA controller speeds up the transfer of data between I/O devices and memory
- DMA controller architecture
- DMA transfers data in three modes: burst, cycle stealing, transparent
- Burst Mode: In this mode, DMA handover the buses to the CPU only after the completion of the whole data transfer.
- Cycle Stealing Mode: In this mode, DMA gives control of buses to the CPU after the transfer of every byte.
- Transparent Mode: Here, DMA transfers data only when the CPU is executing the instruction, which does not require the use of buses.
- System Memory
- I/O Devices
- DMA Process
- Advantages of DMA
- Disadvantages of DMA
- Applications of DMA

Objective Type Questions

1. Who performs the functions of the processor in DMA?
2. In DMA transfers, who gives the required signals and addresses?
3. How many registers does a DMA Controller have?
4. Which unit of DMA controller generates addresses and selects I/O devices for transfer?

5. Which unit in the DMA controller keeps the counts of the number of blocks transferred and indicates the direction of transfer of data?

6. Which bus is the DMA controller connected to?

7. Can a single DMA controller perform operations on two different disks simultaneously?

8. When does the process request for a DMA transfer?

9. Who initiates DMA transfer?

Answers to Objective Type Questions

1. DMA Controller

2. DMA Controller

3. 3

4. Address unit

5. Control unit

6. System bus

7. True

8. Process is temporarily suspended & Another process gets executed

9. I/O devices

Assignments

1. Explain the concept of Direct Memory Access (DMA). How does it differ from traditional CPU-driven I/O operations? Discuss the advantages of using DMA in a computer system.

2. Describe the different types of DMA transfer modes (e.g., burst mode, cycle stealing, block transfer).

3. Design a simple DMA controller for a microcontroller system. Describe the key components and the role each component plays in facilitating DMA operations.

4. Compare and contrast DMA with interrupt-driven I/O. In what situations would one be preferred over the other, and why?

5. What are the common challenges or issues faced when implementing DMA in a computer system? Discuss potential solutions or workarounds for these challenges.

6. Explore the security implications of DMA. How can DMA be exploited in a system, and what measures can be taken to mitigate these risks?

7. Conduct a performance analysis of a system using DMA versus one using traditional CPU-driven I/O. What metrics would you use to evaluate performance, and what results would you expect?

Suggested Reading

1. M.Morris Mano-Computer Systems Architecture, Third Edition, Pearson Education

References

1. Kai Hwang and F A Briggs-Computer Architecture and parallel processing, McGraw Hills,1990

2. Abraham Silberschatz, Peter Galvin and Greg Gagne - Operating System Principles, Seventh Edition, John Wiley

3. William Stallings - Operating Systems, Sixth Edition, Prentice Hall of India, Pearson

4. Carl Hamacher -Computer Organization, Fifth Edition, Tata McGraw Hill.

5. John P Hayes -Computer Architecture & Organization–Mc Graw Hill

6. William Stallings-Computer Organization and Architecture, Seventh Edition, Pearson Education

7. Milan Kovic - Operating Systems, Second Edition, TMH

BLOCK 3

Parallel Computer Structures



UNIT 1

Introduction to Parallel Processing

Learning Outcomes

By the completion of this unit, the learner will be able to:

- narrate the basics of parallel processing
- familiarise the evolution and various generations of computer
- explain the processes and types of data processing
- understand the concept of pipeline computers and multiprocessor systems

Prerequisites

In the past, computers were typically perceived as operating in a sequential manner. Programming languages typically necessitate that programmers define algorithms as sequences of instructions. Processors execute programs by following machine instructions in a sequential manner. Each instruction is processed through a series of steps: fetching the instruction, retrieving operands, performing the operation, and storing the results. However in a parallel organization, several processing units work together to run applications. With the advancement of computer technology and the reduction in hardware costs, designers have increasingly pursued opportunities for parallelism. This approach aims to boost performance and improve availability.

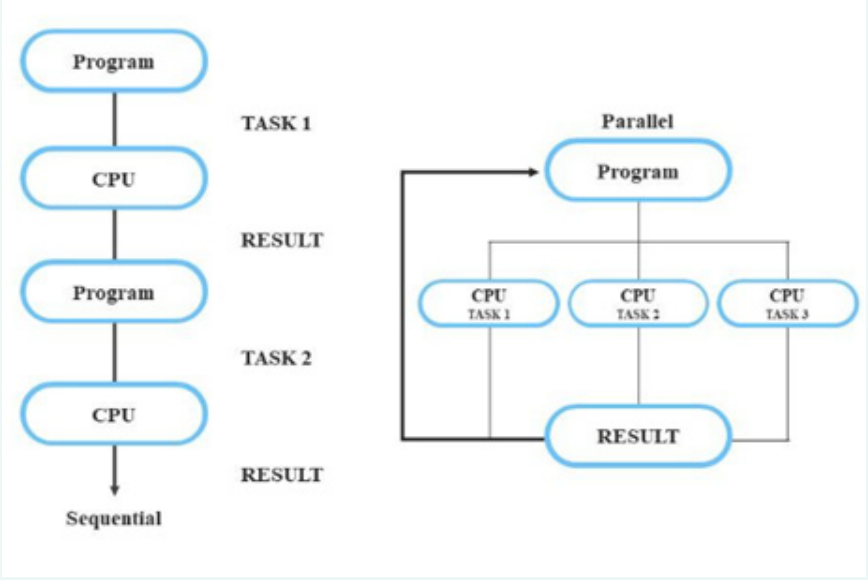


Figure 3.1.1 Serial Vs Parallel Processing

Understanding the basics of parallel processing in computer architecture requires familiarity with fundamental computing concepts. This includes an understanding of what a processor is, how it executes instructions, and the concept of a central processing unit (CPU), which we have studied in the previous blocks. Learners should also grasp the concepts of computational tasks and how they can be broken down into smaller, manageable parts. Familiarity with basic programming principles and data structures, such as arrays and loops, is essential. Following an overview, this chapter explores some of the most notable methods for parallel organization.

Keywords

Serial and Parallel processing, Data Processing, Pipeline Computers, Multiprocessor System

Discussion

3.1.1 Parallel Processing Concepts

In this section, we are going to discuss the concepts of parallel processing. A computer is a device that can take data as input from the user and process it under the control of a group of instructions. After processing the input data, the system generates output. The generated output is saved into the system memory.

Computers like personal desktop computers, laptops, and tablets have become crucial parts of everyday human life in resolving day-to-day life problems. In 1944, the first computer was developed, and that computer was called the "Mark I"; this computer was used for calculation purposes.

All computer instructions can be executed in sequential and parallel ways. If we execute instructions sequentially, it will take more time to execute that instruction.

Let us see an example of program execution using a sequential approach.

1. Suppose there are three instructions stored in the computer. The Central Processing Unit (CPU) fetches the first instruction from the main memory and executes the first process.

2. The CPU brings the second instruction to the main memory and completes the second instruction.

3. The CPU fetches the third instruction from the main memory and executes the third instruction. Likewise, the CPU completes instructions in a one-by-one manner.

The sequential approach to instruction execution slows the system's performance. The system takes more time to execute these sequential instructions. If you want to enhance the performance of the computer, then you have to use parallel processing. The main purpose of parallel processing is to enhance the processing capability and increase the data processing throughput of computers.

3.1.2 Data Processing

In Block 1, we have learned about the basic functional architecture of the computer system. Let us discuss the data processing life cycle.



Fig 3.1.2 Data Processing Cycle

Figure 3.1.2 is the representation of the data processing cycle. The data processing cycle first gathers the raw data. The collected raw data are processed manually or automatically and stored in the system memory for further use. The whole data processing cycle method is shown in Figure 3.1.4. The data processing cycle goes through a sequence of steps to process the raw data and convert it into a usable form. There are 6 stages in the data processing cycle. Figure 3.1.3 illustrates the data transfer from the I/O device to the CPU.

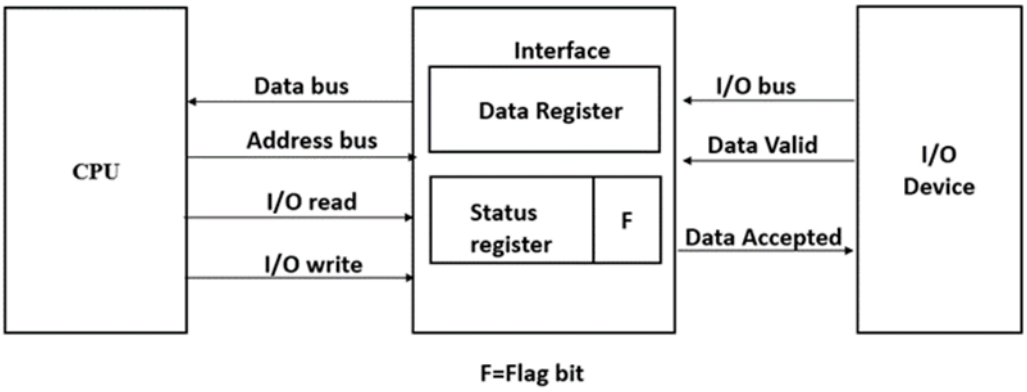


Fig 3.1.3 Data transfer from I/O device to CPU

3.1.3 Data Processing Cycle Stages

Figure 3.1.4 shows different stages of data processing. The first stage in the data processing

cycle is data collection. It is the process of collecting raw data. Data preparation is the second activity. It is the process of manipulating the data meaningfully and converting it into the desired form. The third is data entry. It is the process where verified data is inserted into the system with the help of any input devices. Data processing is the fourth stage, where collected data are processed using different methodologies. The fifth stage is data interpretation. It is the process of providing meaningful information from data. Data storage is the final stage of data in the data processing cycle, where instructions and information related to data are stored in the system for future use.

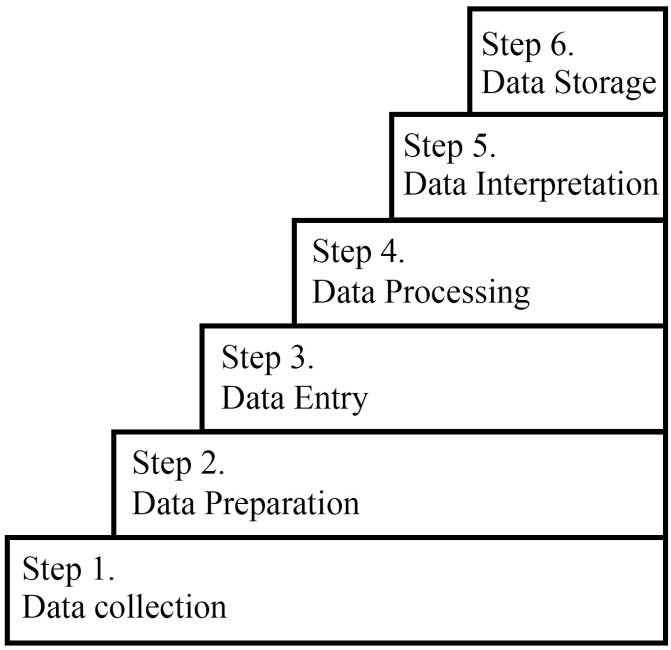


Fig 3.1.4 Steps in Data Processing

3.1.4 Serial and Parallel Data Processing

There are two methods of data processing in computer systems: serial data processing and parallel data processing. In parallel data processing, each processor executes the tasks assigned to it simultaneously. Serial processing is a type of processing in which one task is completed at a time, and the processor executes all the tasks in a sequence.

Parallel processing is a type of processing in which different processors complete multiple tasks at a time. This is the fundamental difference between serial and parallel processing.

3.1.5 Serial Processing

In serial processing, the processor executes the instructions sequentially, i.e. the processor will execute the instructions one at a time in a sequential manner.

Figure 3.1.5 represents a real-time example of serial data processing. Assume that you are in a shopping mall complex having only one billing counter (Counter1) and three Queues (Queue1, Queue2, Queue3). During peak hours, there may be a lot of customers

availing the service in the queues. You have to wait in the queue, and your time may be lost waiting in the line at the counter. You can pay and come out of the mall only when it's your turn, and this is the principle behind serial processing.

In the serial processing system, only one customer is taken care of at a time at the billing counter. However, Parallel data processing can handle multiple customers at the billing counter, and the overall processing time required to manage the customers can be improved.

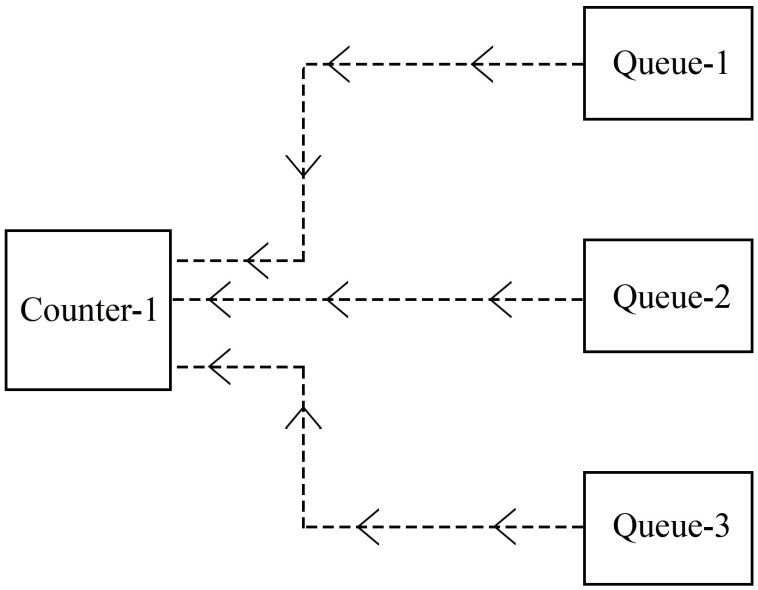


Fig 3.1.5 Example of a Serial processing

3.1.6 Parallel Processing

Parallelism refers to doing multiple things at a time. In parallel processing, a programmer can process multiple instructions and data simultaneously. Data processing systems can perform numerous tasks at a time with the help of parallel processing techniques. The parallel processing system aims to speed up the computer processing capability and increase the throughput.

Parallel Processing Definition: "Parallel processing is an efficient form of information processing that emphasizes the exploitation of concurrent events in the computing process. Parallel events may occur at the same instant, and it demands concurrent execution of many programs in the computer" taken from the book *Computer Architecture and Parallel Processing* by Hwang, Kai, and A. Faye. (1984).

Now, let us try to learn parallel processing from a real-life example.

Let us take the example of a shopping complex billing counter and queue. In Figure 3.1.6, assume that you are in a shopping mall with three billing counters (Counter 1, Counter 2 and Counter 3). Then, you don't have to wait for long at the counter. Because multiple counters are operating in parallel, the time you spend on the purchase will be significantly less than in the previous shopping mall. This is one of the best examples

of parallel processing.

Parallel processing facilitates the fast processing of a large number of system computations by dividing the amount of work between more than one processor, where all the calculations are done in a simultaneous fashion. Parallel computing aims to increase the available computation power for faster application processing of parallel processing, which is listed below. Parallel processing has several advantages and disadvantages.

Advantages	Disadvantages
1. The parallel process improves the performance of the execution of data and instruction. 2. Parallel computing is better suited for many computational systems. 3. Parallel processing is cost-effective 4. It can process a massive set of data and manage big data computational work at a faster pace.	1. The data programming for parallel processing is complex 2. The parallel processing solutions are harder to implement.

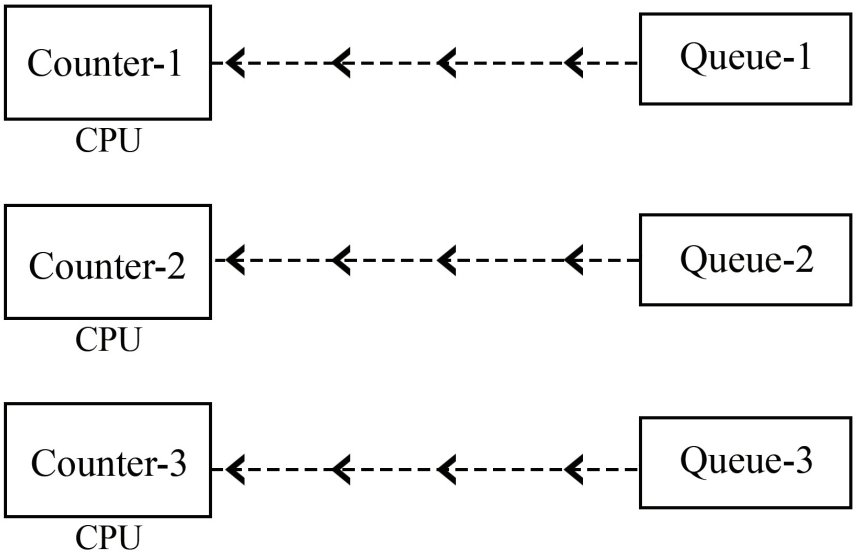


Fig 3.1.6 Example of a Parallel processing

Parallel architecture refers to a set of parallel processors where a collection of processing elements cooperate to solve large problems quickly. Parallel processing architecture is mainly divided into three configurations.

1. Pipeline Computers
2. Array processors
3. Multiprocessor system

3.1.7 Pipeline Computer

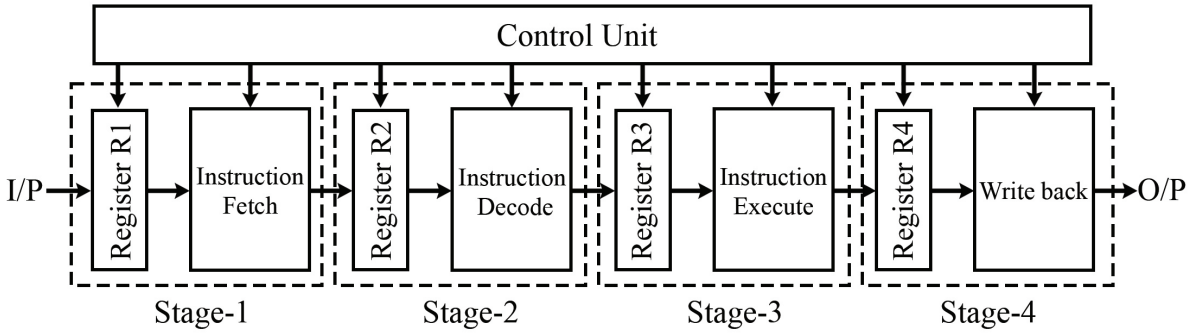
In this section, we discuss pipeline computers. Parallel processing is a type of data processing computational task where many types of data calculations and process executions are performed simultaneously.

Pipeline Computers are the systems that can arrange the computer hardware elements of the Central Processing Unit so that the system's overall performance is increased. In pipeline process hardware architecture, execution of more than one instruction takes place in a pipeline fashion.

Figure 3.1.7 shows the hardware arrangements in a pipeline computer architecture. Consider a pipelined computer with instructions (I1, I2, ...In) and registers (R1,R2.... Rm). In the Figure 3.1.7, there are four segments (stages). Each segment contains instructions and registers. The input data is entered into the first stage, and the program instructions are executed. The interface register can store the executed result data of each stage. The output from the first stage is transferred into the second stage. A common clock controls all the stages of the pipeline. In the pipeline process, the output of one stage is the input of the next stage. In this way, a pipelined computer executes multiple instructions simultaneously.

The simplest way to understand the pipeline structure is to envision each segment as having an input register followed by a combinational circuit. The register stores the data, while the combinational circuit performs a sub-operation for that segment. The output from the combinational circuit is then sent to the input register of the next segment. A clock signal is applied to all registers after sufficient time has passed for each segment's activity to complete. This allows information to move through the pipeline one step at a time.

An example demonstrating the pipeline organisation is shown below. Suppose we want



Four Stage Instruction Pipeline

Fig 3.1.7 Hardware Arrangements in a Pipeline Computer

to perform the combined multiply and add operations with a stream of numbers.

Ai*Bi + Ci for i =1, 2, 3 ….7

Each sub-operation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit, as shown in Figure 3.1.8.

Registers R1 through R5 receive new data with each clock pulse. The multiplier and adder are combinational circuits. The sub-operations performed in each segment of the pipeline are as follows:

R1<- Ai	R2<-Bi	Input Ai and Bi
R3<-R1*R2	R4<-Ci	multiply and input Ci
R5<-R3+R4		add Ci to product

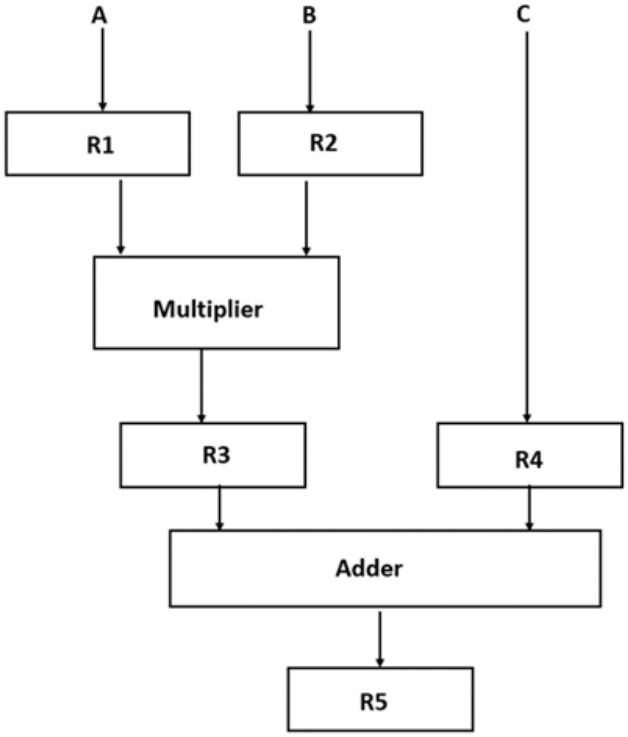


Fig 3.1.8 Pipeline implementation of Ai*Bi + Ci; for i=1, 2, 3 ….7

The five registers are loaded with new data every clock pulse.

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1	-	-	-

2	A2	B2	A1* B1	C1	-
3	A3	B3	A2* B2	C2	A1* B1+C1
4	A4	B4	A3* B3	C3	A2* B2+C2
5	A5	B5	A4* B4	C4	A3* B3+C3
6	A6	B6	A5* B5	C5	A4* B4+C4
7	A7	B7	A6* B6	C6	A5* B5+C5
8	-	-	A7* B7	C7	A6* B6+C6
9	-	-	-	-	A7* B7+C7

The first clock pulse transfers A1 and B1 into registers R1 and R2. The second clock pulse transfers the product of R1 and R2 into R3 and moves C1 into R4 while simultaneously transferring A2 and B2 into R1 and R2. During the third clock pulse, all three segments operate simultaneously: A3 and B3 are placed into R1 and R2, the product of R1 and R2 is moved into R3, C2 is transferred into R4, and the sum of R3 and R4 is placed into R5. It takes three clock pulses to fill the pipeline and produce the first output from R5. After that, each clock pulse generates a new output and advances the data one step down the pipeline, provided new input data continues to flow into the system.

Four Segment Pipeline

Figure 3.1.9 shows the basic structure of a four-segment pipeline. The operands move through all four segments in a set sequence. Each segment has a combinational circuit, Si, that performs a sub-operation on the data stream. Registers, Ri, separate the segments and hold intermediate results. A common clock applied to all the registers at the same time controls the flow of information between the stages, as shown in Figure 3.1.9.

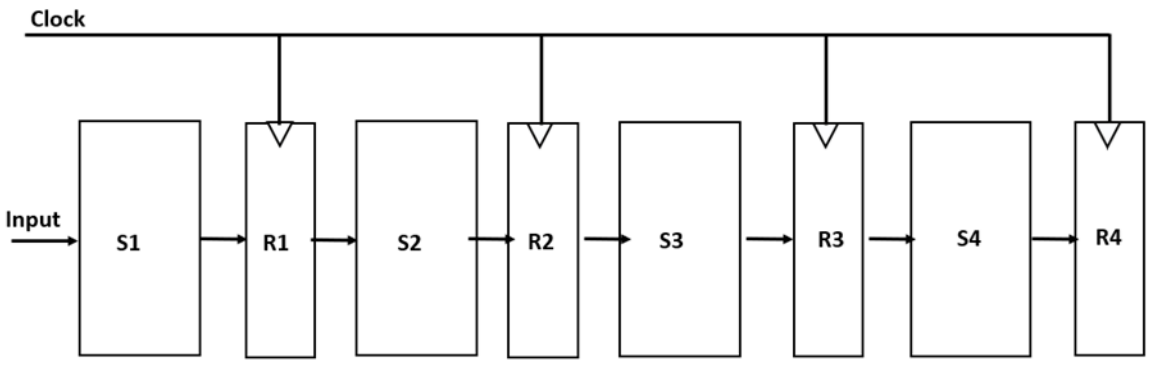


Fig 3.1.9 Illustration of four-segment pipeline

Space-Time Diagram

The behaviour of a pipeline can be demonstrated with a space-time diagram, which shows segment utilization over time. In Figure 3.1.10, the horizontal axis represents time in clock cycles, while the vertical axis shows the segment numbers. The diagram illustrates six tasks, T1 through T6, being executed across four segments. Initially, task T1 is processed by segment 1. After the first clock cycle, segment 2 processes T1, while segment 1 starts on T2. This pattern continues, and the first task T1 is completed after the fourth clock cycle. From that point on, the pipeline completes one task per clock cycle.

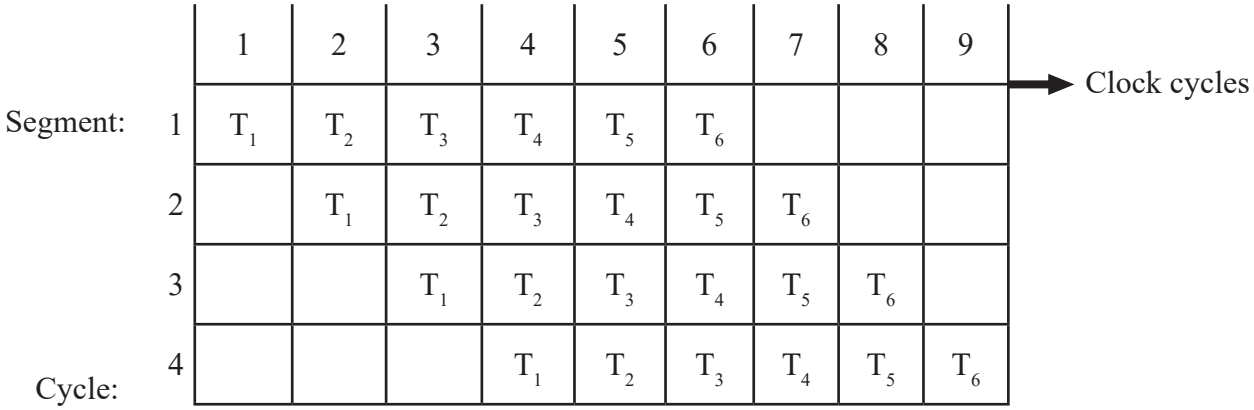


Fig 3.1.10 Space-Time Diagram

Consider a k-segment pipeline with a clock cycle time of tp used to execute n tasks. The first task, T1, takes a time equal to ktp to complete because there are k segments in the pipeline. The remaining n-1 tasks emerge from the pipeline at a rate of one task per clock cycle, completing after (n-1)tp. Therefore, completing n tasks with a k-segment pipeline requires a total of k + (n-1) clock cycles.

3.1.8 Array Processor System

The array processor falls under the category of a multiprocessor system where a single instruction controls the simultaneous execution of a number of processing elements. The key to the design of a supercomputer or array processor is to recognize that the main task is to perform arithmetic operations on arrays or vectors of floating-point numbers. In a general-purpose computer, this will require iteration through each element of the array. Figure 3.1.11 shows an example of vector addition.

The array processor usually refers to an auxiliary processor attached to a general-purpose processor and used to perform vector computation. An array processor may use either the pipelined or parallel ALU approach. Array processing is sometimes used to refer to a parallel ALU, although any of the three organizations would be optimized for the processing of arrays. To make matters worse, an array processor usually refers to an auxiliary processor attached to a general-purpose processor and used to perform vector computation. An array processor may use either the pipelined or parallel ALU approach.

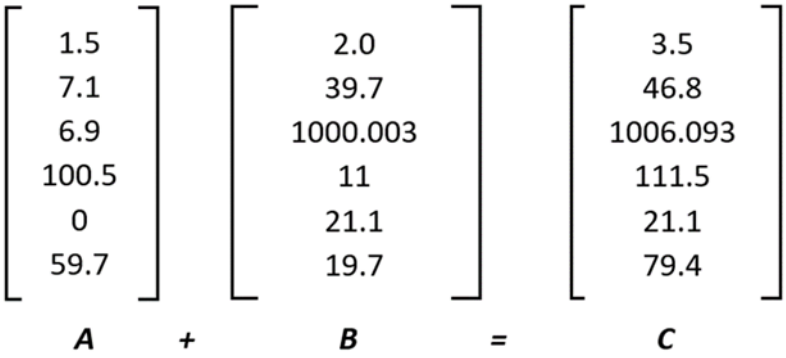


Fig 3.1.11 Vector Addition

3.1.9 Multiprocessing System

It is a computer with more than one Central Processing Unit, each sharing the main memory and peripherals in order to simultaneously process programs. These computer systems are also called tightly coupled systems and can process large volumes of data within a short period.

A multiprocessing system has multiple central processing units. The system contains a number of standard processors put together in an innovative way to improve the performance and speed of computer hardware. Based on the usage of processors in the multiprocessing system, they are divided into two types.

1. The processors that share a physical address space are called a **shared memory multiprocessor system**.

2. The processors that have private, non-shared address spaces are called a **private memory multiprocessor system**.

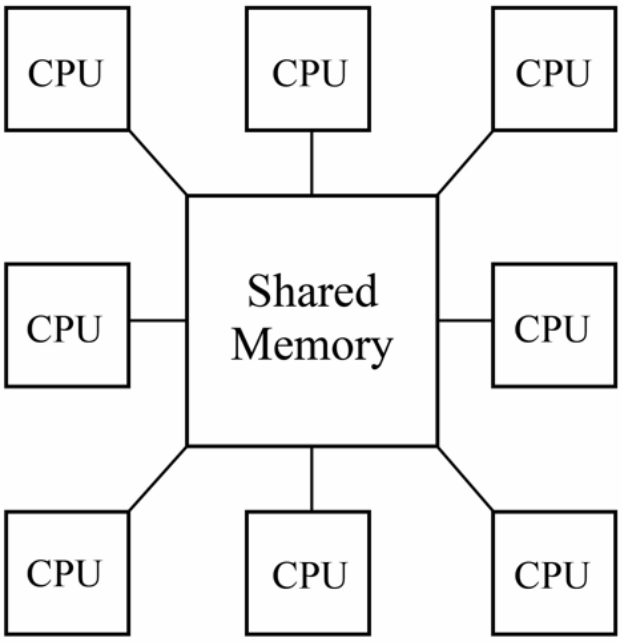


Fig 3.1.12 Multiprocessing Computer System

Figure 3.1.12 shows a multiprocessor system. The system contains multiple CPUs and shares the main system memory. The multiprocessor system is based on the symmetric multiprocessing model, in which each processor runs an identical copy of the operating system, and these copies communicate with each other.

3.1.9.1 Interconnection Structures

A multiprocessor system consists of CPUs, I/O processors (IOPs) connected to input-output devices, and a memory unit that may be divided into multiple modules. The way these components are interconnected varies depending on the number of transfer paths available between the processors and memory in a shared memory setup or among processing elements in a loosely coupled system. There are various physical configurations for creating an interconnection network. Some of these configurations are discussed in this section. They are:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

A. Time Shared Common Bus

In a time- shared common bus organization, multiple processing units or devices share a single communication bus to transfer data. Since the bus is a shared resource, only one unit can transmit data at any given time, preventing simultaneous communication by different devices. Fig. 3.1.13(a) only one processor can communicate with the memory or another processor at any given time. When one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus.

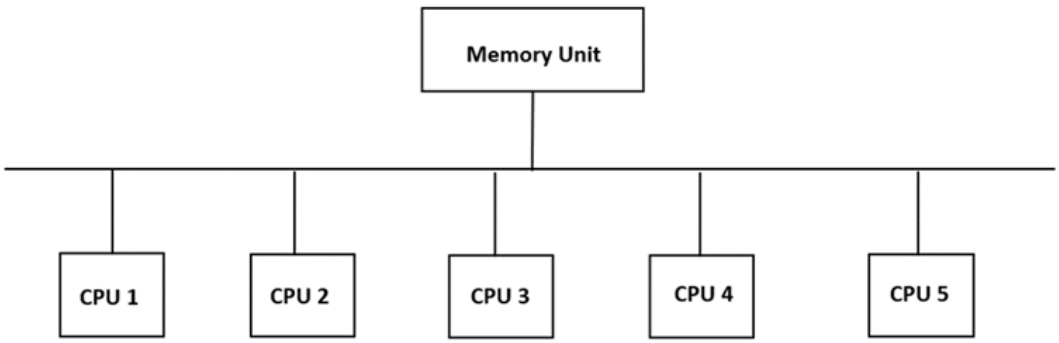


Fig 3.1.13(a) Time Shared Common Bus Organization

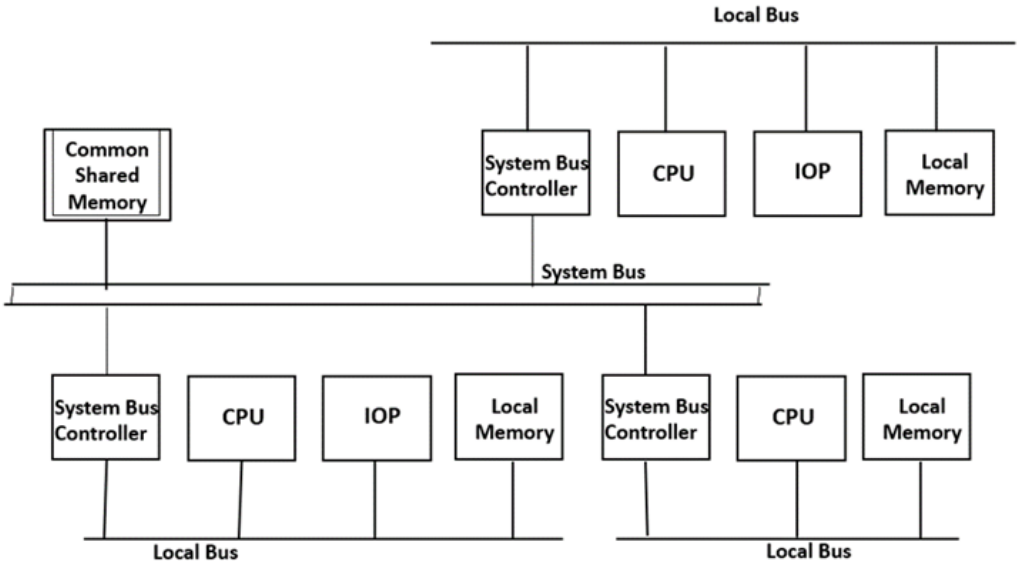


Fig 3.1.13 (b) System Bus Structure for Multiprocessor

B. Multiport Memory

A multiport memory system uses individual buses to connect each memory module to each CPU. Figure 3.1.14 demonstrates this setup with four CPUs and four memory modules (MMs). In this arrangement, each processor bus is linked to every memory module, comprising the address, data, and control lines necessary for memory communication.

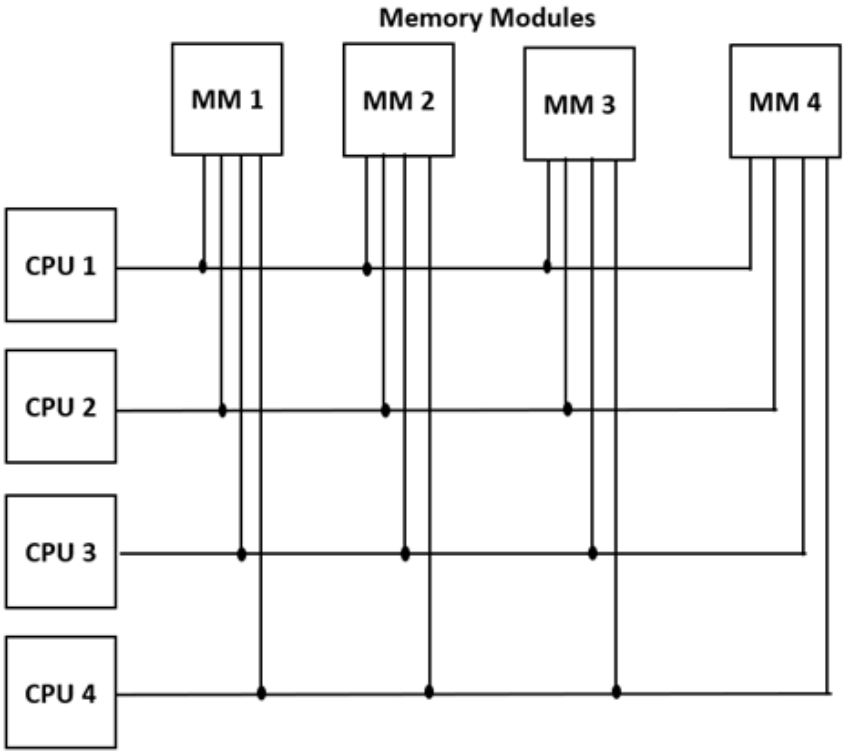


Fig 3.1.14 Multi-port Memory Organization

C. Crossbar Switch

The crossbar switch system is an example of a Multiprocessing Computer System. The system contains a number of crosspoints that are kept at intersections among memory module and processor bus paths. Figure 3.1.15a and b represent a model of a cross-bar switch system. The architecture contains many processors, shared memory and input-output devices. In the crossbar switch system, large bandwidth data buses switch between resources like processors, I/O, or memory. The crossbar switches connect a transmitting resource to a receiving resource. They are good for moving large blocks of data very quickly.

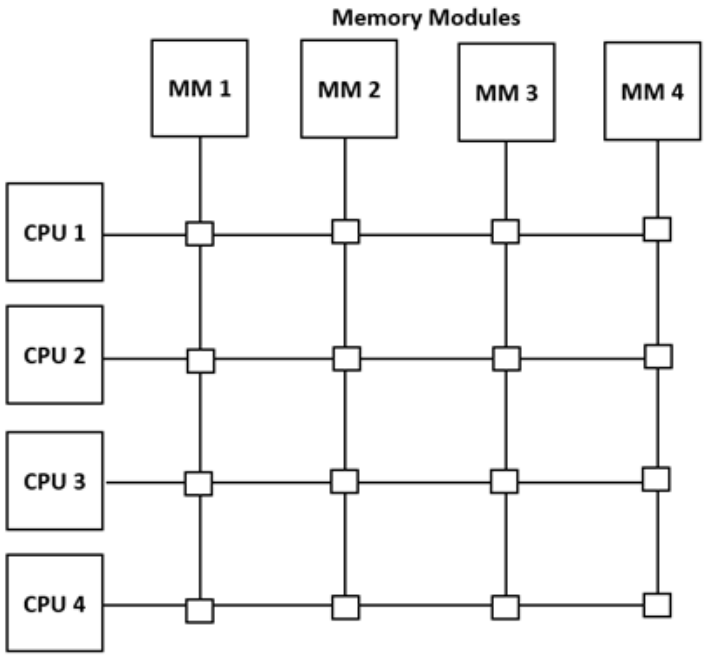


Fig 3.1.15 (a) Crossbar switch architecture

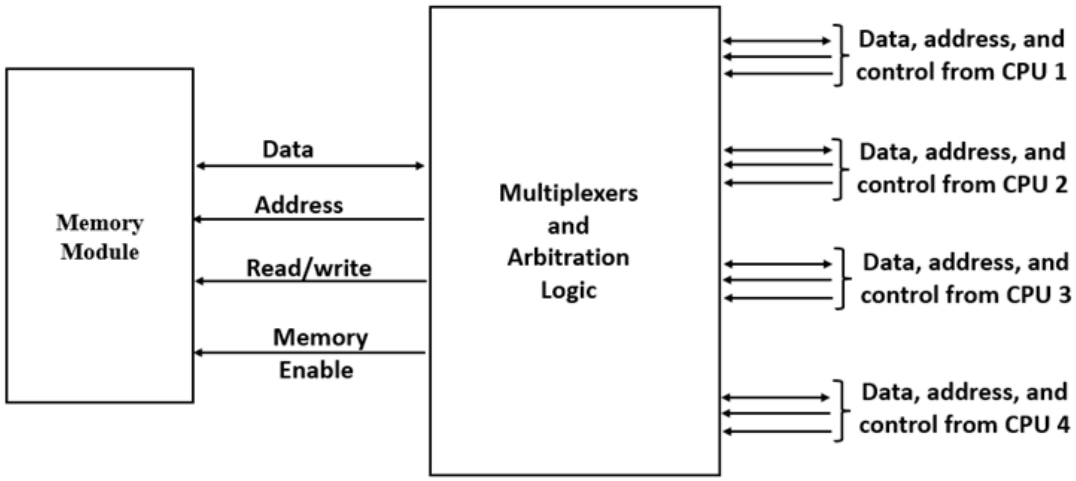


Fig 3.1.15 (b) Block diagram of crossbar switch

D. Multistage Switching Network

The basic component of a multistage network is a two-input, two-output interchange switch. As shown in Figure 3.1.16(a), the 2 x 2 switch has two inputs, labelled A and B, and two outputs, labelled 0 and 1. There is control sigma (not shown) associated with the switch that establishes the interconnection between the input and output terminals. The switch can connect input A to either of the outputs. Terminal B of the switch behaves similarly. The switch can also arbitrate between conflicting requests. If inputs A and B request the same output terminal, only one of them will be connected, and the other will be blocked.

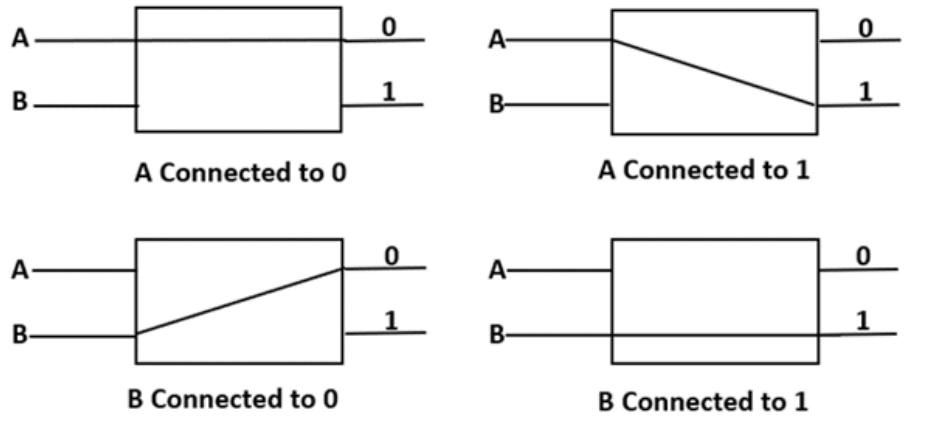


Fig 3.1.16 (a)Operation of a 2 X 2 interchange switch

Numerous topologies have been suggested for multistage switching networks to manage processor-memory communication in tightly coupled multiprocessor systems and handle communication between processing elements in loosely coupled systems. An example of such a topology is the omega-switching network, illustrated in Figure 3.1.17

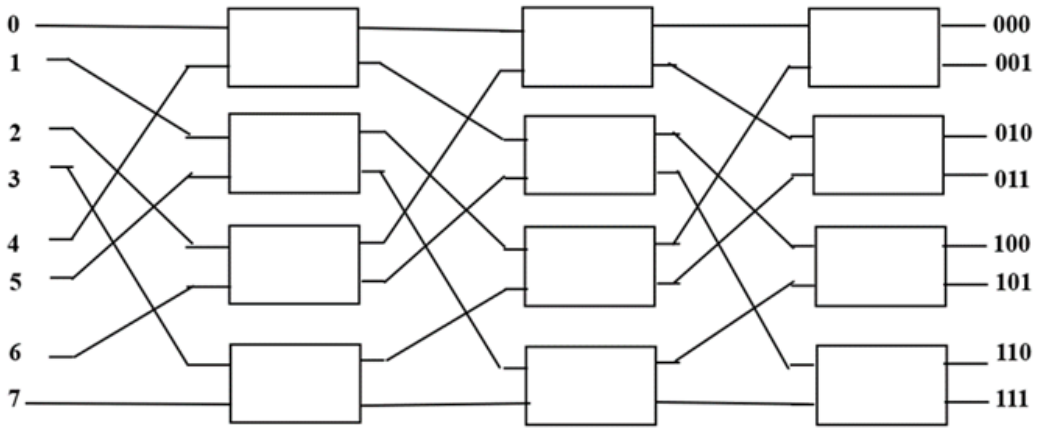


Fig 3.1.17 8 × 8 Omega switching network

E. Hypercube Interconnection

The hypercube, or binary n-cube multiprocessor structure, is a loosely coupled system

consisting of N = 2^n processors interconnected in an n-dimensional binary cube. Each processor acts as a node within the cube. Typically, each node is considered to have a processor, but it actually includes a CPU, local memory, and an I/O interface. Each processor is directly connected to n neighbouring processors through communication paths, which correspond to the edges of the cube. The processors are assigned 2^n distinct n-bit binary addresses, with each processor's address differing from each of its n neighbours by exactly one bit.

Figure 3.1.18 depicts the hypercube structure for n = 1, 2, and 3. A one-cube structure with n = 1 and 2^n= 2 includes two processors connected by a single path. A two-cube structure with n = 2 and 2^n= 4 consists of four nodes interconnected in a square. A three-cube structure comprises eight nodes interconnected as a cube. An n-cube structure features 2^n nodes, each with a processor. Each node is assigned a binary address such that the addresses of two neighbouring nodes differ by exactly one bit. For instance, the three neighbours of the node with address 100 in a three-cube structure are 000, 110, and 101, as each of these binary addresses differs from 100 by one bit.

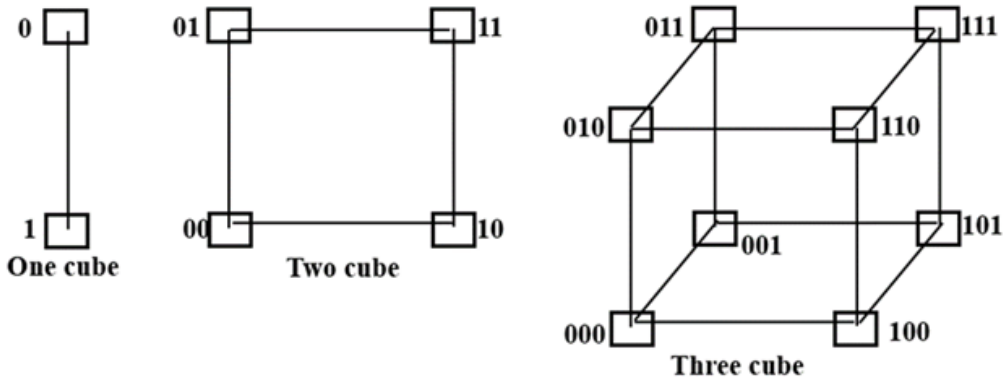


Fig 3.1.18 Hypercube structures for n=1, 2, 3

3.1.9.2 Characteristics of Multiprocessing System

- The system contains two or more similar processors.
- All the processors share the same memory and input and output facilities inside the system.
- All the processors are connected by a bus or other internal connection scheme, such that memory access time is the same for each processor.
- All processors can perform the same function
- The system is controlled by the operating system and the Multiprocessor system, providing interaction between the processors

Recap

- Computer instructions can be executed in a sequential and parallel fashion.
- Computer instructions executing sequentially will take more time to implement.
- Data Collection, Data Preparation, Data Entry, Data Processing, Data Interpretation, and Data Storage are the steps in data processing.
- Serial processing solutions are simple to implement, whereas parallel processing is harder to implement.
- Parallel processing is suitable for organizing massive sets of data and managing big data computational works.
- The parallel processing architecture is divided into three configurations such as pipeline computers, array processors, and multiprocessor systems.
- Pipeline Computers are the systems that can arrange the computer hardware elements of the Central Processing Unit such that the system's overall performance is increased
- Multiprocessing systems are computers with more than one Central Processing Unit.
- Pipelining is achieved by having multiple independent functional units in hardware. These independent functional units are called stages.
- A crossbar switch architecture cross-connects each port to all the other ports via the backplane. This means that the switch requires N x N connections for N ports.

Objective Type Questions

1. State true or false. Serial processing has a single execution flow.
2. Which processing leads to concurrency?
3. In which processing are several instructions executed simultaneously?
4. What are uniprocessing computing devices called?
5. What type of processing takes more time to implement?
6. Which processing method is harder to implement?

7. What is suitable for managing big data computational works?
8. What increases overall system performance in Pipeline Computers?
9. What connects each port to all other ports in a crossbar switch?
10. CPU does not perform logic and arithmetic operations. Is the given statement true?
11. Which technology did fourth-generation computers were based on?
12. What are collections of lines that connect several devices called?
13. List one feature of the multi-microprocessor architecture.

Answers to Objective Type Questions

1. True
2. Parallel Processing
3. Parallel Processing
4. single-processor systems
5. Sequential
6. Parallel
7. Parallel
8. Pipelining
9. Backplane
10. False
11. Integrated circuits
12. buses
13. Parallel processing

Assignments

1. Find out five interesting topics/ideas about parallel programming.
2. Discuss how instructions are executed in a pipelined processor.

Suggested Reading

1. Hwang, Kai, and A. Faye. "Computer architecture and parallel processing." (1984).
2. Mano, M. Morris. Computer system architecture. Prentice-Hall, Inc., 1993..
3. Carl Hamacher -Computer Organization, Fifth Edition, Tata McGraw Hill.
4. Stojcev, Mile. "John. P. Hayes, Computer Architecture and Organization, McGraw-Hill Book Company, Inc., Boston, 1988, Softcover, pp. 604, plus XIV, ISBN 0-07-115997-5." Microelectronics Reliability 46.1 (2006): 196-197.
5. Stallings, William. “Computer organization and architecture: designing for performance.” Pearson Education India, 2003.

UNIT 2

Architectural Classification

Learning Outcomes

Upon completion of this Unit, the learner will be able to:

- to introduce the basic working principle of Computers
- to narrate the basics of architectural classification
- to familiarise with Flynn's organisational classification
- to explain the details of Feng's Classification
- to make them understand miscellaneous concepts of architectural classification

Prerequisites

In the previous blocks, we have studied how Computer Systems works and its basic principles, functional blocks of computers, parallel processors and pipelining techniques. Computers do the work primarily in the machine, and a control centre called a central processing unit (CPU) converts the information data input to output.

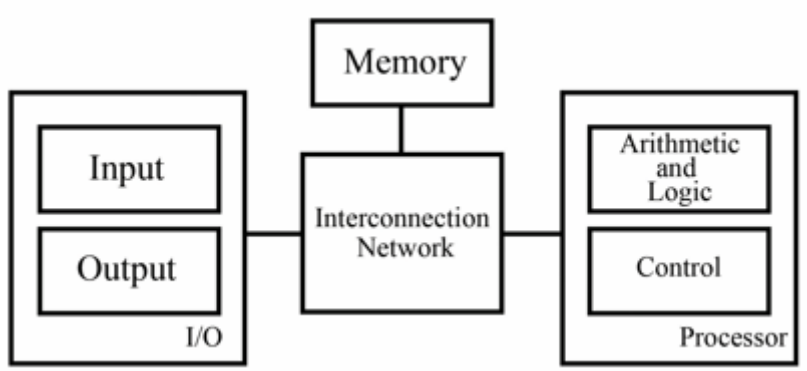


Fig 3.2.1(a) Basic functional units of a computer

Primarily, a computer consists of five functionally independent blocks. It includes input, memory, output, arithmetical and logical unit, and control units. All the blocks are listed below in Figure 3.2.1(a). A computer system primarily functions by processing the

inputs using the control unit termed central processing unit (CPU), making use of the storage space termed memory unit to produce the relevant outputs.

Parallel processing refers to the method of processing or performing the computation in parallel to manage separate parts of an overall task by running two or more processors (CPUs). Parallel processing is mainly used for large volumes of data processing in computer systems. The amount of time required for executing a program will be reduced by splitting up different parts of a task among multiple processors. The Parallel processing task speeds up the computer processing capability and increases the system throughput. The pipelining is the arrangement of hardware elements in the system architecture; the pipelining process can improve the overall performance of the system. In this unit, we will discuss the architectural classification of computer organisations and their schemes and types. Figure 3.2.1(b) shows the taxonomy of parallel processor Architecture.

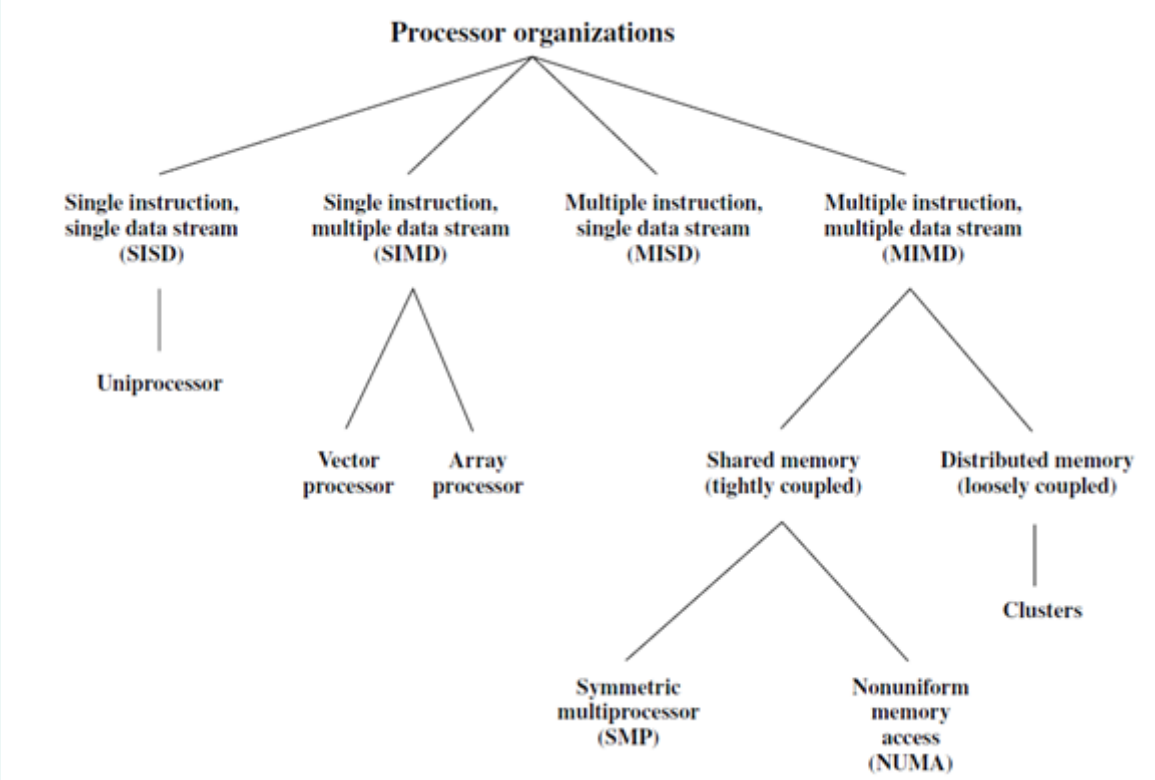


Fig 3.2.1(b) Taxonomy of Parallel processor architecture

Keywords

Flynn's classification, Feng's classification, Handler's classification, instruction stream, data stream, Uniform Memory Access (UMA) Architecture, and Non-Uniform Memory Access (NUMA) architecture.

Discussion

3.2.1 Architectural Classification Scheme

The Parallel computing architecture scheme works with several processors running simultaneously and executing multiple calculations. The computing scheme breaks large and complex calculations into smaller portions and performs operations.

Parallel computing computers can perform concurrent tasks like multiprogramming, multiprocessing, or multi-computing. Parallelism concepts like multitasking, multiprogramming, multithreading, and distributed computing are performed at different processing levels in the classification scheme.

3.2.2 Types of Classification

Parallel computers are essentially based on the same principles as sequential or "Von Neumann" architecture, except that multiple instructions can be executed at any given time. However, some parallel computers, sometimes referred to as "Non-Von Neumann," are completely different; they have no central point of control. Control is totally distributed, with the availability of operands triggering the activation of instructions. Three scientists introduced three types of classification schemes for parallel processing: Flynn in 1966, Feng in 1972, and Handler in 1977. These three types of architectural classification schemes in parallel processing are based on different data processing principles.

3.2.2.1 Flynn's Classification

In 1966, M. J. Flynn proposed a four-way classification of multiprocessor computer architectures. Two types of information flow into a processor, the instructions and the data. Flynn's classification is based on the notions of a multiplicity of instruction streams and data streams. The instruction stream defines the sequence of instructions performed by the processing unit. The data stream defines the data traffic exchanged between the memory and the processing unit. Flynn's classification has become standard and is widely used.

According to Flynn's classification, either of the two dimensions, namely instruction or data streams, can have only one of two possible states, single or multiple. Based on Flynn's classification, computer architecture can be classified into the following four distinct categories namely:

1. Single-instruction single-data streams (SISD)
2. Single-instruction multiple-data streams (SIMD)
3. Multiple-instruction single-data streams (MISD)
4. Multiple-instruction multiple data streams (MIMD)

Flynn's classification will be discussed in detail in the next section.

3.2.2.2 Feng's Classification

Feng, in 1972, proposed a scheme based on the degree of parallelism to classify computer architectures. Based on Feng's principle, four different bit and word-level

classification schemes are available. It is based on the way contents are stored in the memory and processed. The contents can be either data or instructions.

1. Word-Serial Processing
2. Word-Parallel Processing
3. Bit-Serial Processing
4. Bit-Parallel Processing

In the following section, a more detailed explanation of this classification is provided.

3.2.2.3 Handler's Classification

In 1977, Wolfgang Handler proposed an elaborate notation for expressing the pipelining and parallelism of computers. Handler's classification addresses the computer at three distinct levels:

1. Processor control unit (PCU) - each PCU corresponds to one processor or one CPU.
2. Arithmetic logic unit (ALU) - ALU is equivalent to the processing element (PE).
3. Bit-level circuit (BLC) - BLC corresponds to combinational logic circuits request for 1 bit operations in ALU.

Handler's Classification uses three pairs of integers containing to independent entities that describe a computer system.

Computer = <K * K, D+D, W*W> where

K= number of processors

K = number of PSU's that can be pipelined

D = number of ALU's (PEs) under the control of PSU

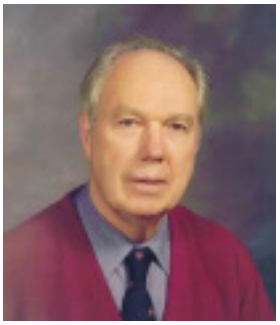
D = number of ALU's (PEs) that can be pipelined

W= word length of a PE.

W = number of pipeline stages in all PE's

Other types of Architectural Classification

1. Classification based on Coupling between processing elements
2. Classification based on the mode of accessing memory

Michael J. Flynn (1934_ Age 86 years), an American professor emeritus at Stanford University, introduced a Classification scheme called Flynn's Classification in 1966 based on the diversity of an instruction stream data stream in a computer system. He proposed Flynn's taxonomy, classifying parallel digital computers, in 1966, and he received the Harry H. Goode Memorial Award.	

Tse-Yun Feng (1928-2015) was a professor at Universities like Wayne, Wright, Ohio, and Pennsylvania University. Tse-Yun Feng's major experience was associative processing, parallel processors, and computer architecture. Feng served as a Distinguished Visitor of the IEEE Computer Society.	

Wolfgang Händler (1920-1998) was a German mathematician, pioneering computer scientist, and professor at Leibniz University Hannover and University of Erlangen–Nuremberg works on parallel computing. He proposed the Handler's Classification in 1977. Handler's Classification is calculated based on the degree of parallelism and pipelining at system levels.	

3.2.3 Flynn Classification

Flynn Classification: Multiplicity of Instruction Stream and Data Streams. The scientist Michael J. Flynn proposed four Important classifications. All these four methods are based upon the number of concurrent instruction (or control) and data streams available in the architecture.

Flynn introduced the concept of instruction and data streams for categorising computers. All the computers classified by Flynn are not parallel. Flynn's Classification is based on instruction and data streams,

- The instruction stream is defined as the sequence of instructions performed by the processing unit.
- The data stream is defined as the data traffic exchanged between the memory and the processing unit.

a) The instruction stream

The instruction cycle consists of a series of phases needed to execute an instruction in a program. A typical instruction in a program is composed of two parts called Operand and Opcode.

The Operand part specifies the data on which the specified operation is to be done. The Operand part is divided into two parts:

1. Addressing mode.
2. Operand.

The first part, addressing mode, specifies the method of determining the actual data

addresses on which the operation is to be performed. The second part, the operand, is used as an argument to decide the exact address.

b) The data stream

The word 'stream' refers to a continuous sequence or flow of instructions or data operated by the computer. In a system instruction execution, a flow of data instructions is transferred from the main memory to the CPU. This type of data flow of instructions is called an instruction stream. In parallel processing, the flow of data between the processor and memory is bidirectional. This flow of operands is called a data stream.

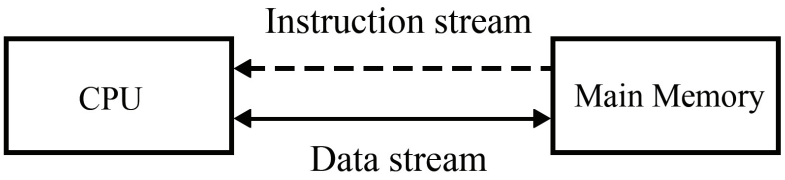


Fig 3.2.2 Instruction and data stream

The above figure (Figure 3.2.2) shows two types of data flow. The instruction stream and the data stream. The sequence of instructions executed by the CPU forms the Instruction streams and the sequence of data (operands) required for execution of instructions forms the Data streams. Figure 3.2.3 shows the taxonomy of Flynn's classification.

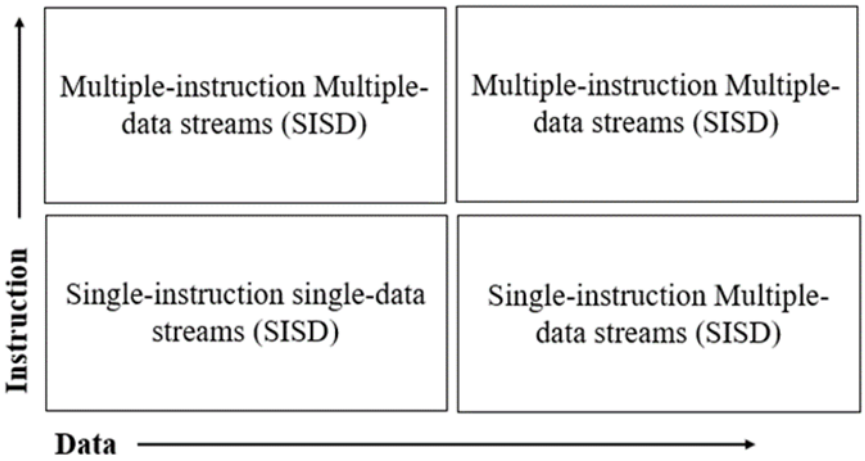


Fig 3.2.3 Taxonomy of Flynn's Classification

3.2.3.1 Single Instruction Single Data Stream (SISD)

The scientist Flynn introduced four classical taxonomies. The conventional single processor Von Neumann computers are classified as SISD system category. The SISD machines are traditional serial computers that process only one stream of instructions and one stream of data. Only one instruction stream is being executed by the central processing unit during one clock cycle, and only one data stream is used as input during one clock cycle.

A Single Instruction Single Data stream computing system is a uniprocessor machine

capable of executing a single instruction (refer to Figure 3.2.4), which operates on a single data stream. All the data instructions and processed data are to be stored in primary memory.

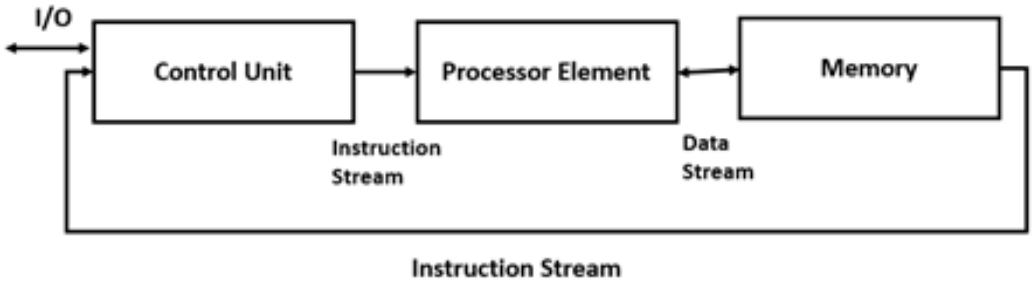


Fig 3.2.4 SISD Architecture

Advantages of Single Instruction Single Data Stream

- SISD architecture can be implemented in all serial computers
- One single instruction stream can be activated by the CPU during one cock cycle.
- One data stream is used as an input during one clock cycle

3.2.3.2 Single Instruction Multiple Data Stream (SIMD)

The Single Instruction Multiple Data Stream (SIMD) model of parallel computing consists of two parts: a front-end computer of the usual Von Neumann style and a processor array (refer to Figure 3.2.5). The processor array is a set of identical synchronised processing elements capable of simultaneously performing the same operation on different data. The multiple processing elements work under the control of a single control unit. It has one instruction and multiple data streams.

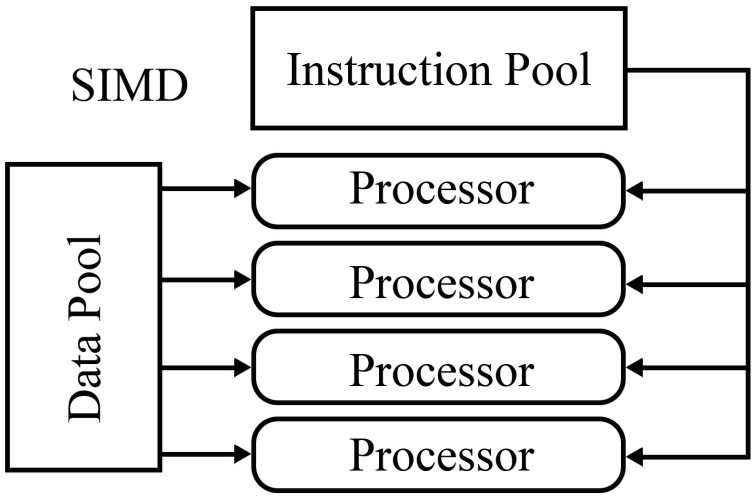


Fig 3.2.5 SIMD Architecture

It has a single control unit that is used to evoke several isolated processing units. The shared memory unit is divided into modules so that it can interact with all the processors simultaneously.

Advantages of Single Instruction Multiple Data Stream (SIMD)

- This architecture is suitable for all types of parallel computers
- All the processing units execute the same instruction at any given clock cycle
- Each processing unit can operate on a multiple-data element
- Synchronous and deterministic execution

3.2.3.3 Multiple-Instruction Single-Data Streams (MISD)

In the Multiple-Instruction Single-Data Streams (MISD) type model, the same stream of data flows through a linear array of processors executing different instruction streams. This computing is a multiprocessor method capable of executing different instructions on processing elements but operating on the same dataset. Each processor operates under the control of an instruction stream issued by the system control unit.

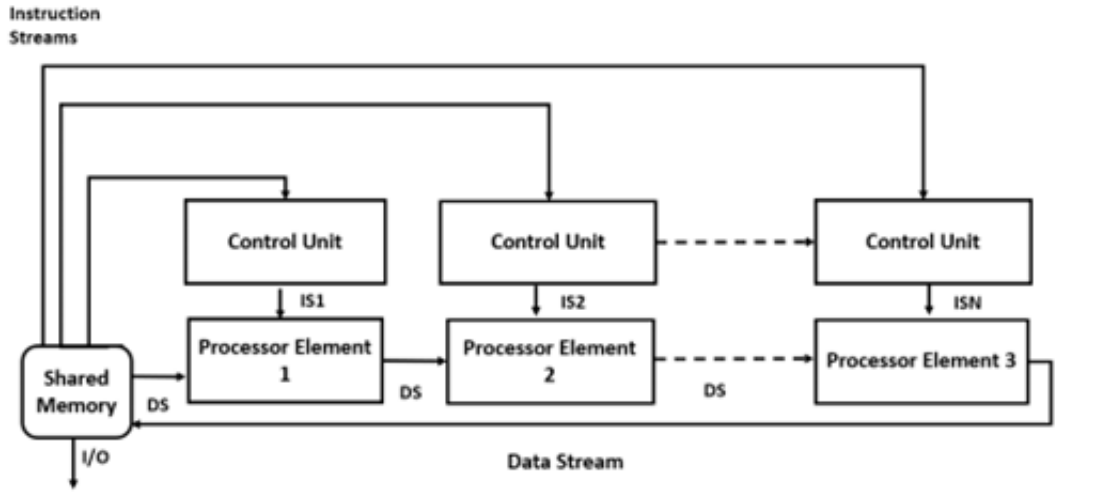


Fig 3.2.6 MISD Architecture

The MISD processor can execute different programs on different data while solving different sub-problems of a single problem.

Advantages of Multiple-Instruction Single-Data Streams (MISD)

- MISD architecture is suitable for all the parallel computer
- Multiple instructions can be implemented in the MISD architecture. Each processing unit operates on the data independently via separate instruction streams.
- In the classification method, a single data stream is fed into multiple processing units.

3.2.3.4 Multiple-Instruction Multiple-Data Streams (MIMD)

Multiple-instruction multiple-data streams (MIMD) parallel architectures are made of

multiple processors and multiple memory modules connected via some interconnection network. It is a multiprocessor machine that is capable of executing multiple instructions. Multiple data streams exist to handle multiple instruction streams, control units, and processing elements. Multiple processing elements are running numerous data streams from the system's main memory.

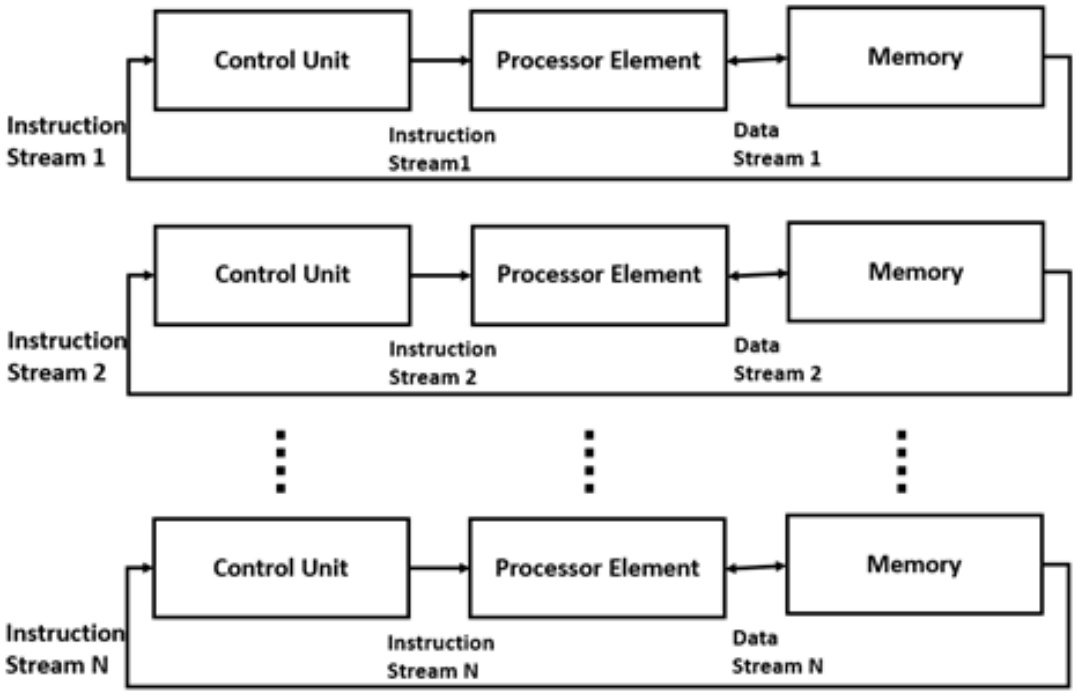


Fig 3.2.7 MIMD Architecture

Advantages of Multiple Instruction Multiple Data Streams (MIMD)

- The MIMD architecture can be implemented on parallel computing.
- Multiple instructions can be executed in every processor using different instruction stream
- Every processor is working with multiple data with a different data stream
- The program execution can be synchronous or asynchronous, deterministic or non-deterministic.

3.2.4 Feng's Classification

3.2.4.1 Feng's Classification: Serial Processing Versus Parallel Processing

Tse-Yun Feng's Classification is mainly based on serial and parallel processing in computer system architecture. According to Feng's Classification, computer architecture can be classified into four. The classification is based on the way contents are stored in memory for data processing. The contents can be either data or instructions.

This method explains that the maximum number of binary digits that can be processed within a computer unit is called the full parallelism degree. This parallelism is exhibited by word length and bit-slice length. Based on the word and the bit-slice length, Feng

suggested four types of processing methods.

1. Word Serial and Bit Serial (WSBS)
2. Word Parallel and Bit Serial (WPBS)
3. Word Serial and Bit Parallel (WSBP)
4. Word Parallel and Bit Parallel (WPBP)

Word Serial and Bit Serial (WSBS): It has been called bit-parallel processing because one bit is processed at a time. In this method, both words and bits are processed serially. This means that data is processed one bit at a time, and each word is processed sequentially.

Word Parallel and Bit Serial (WPBS): It has been called bit-slice processing because the m-bit slice is processed at a time. Here, while bits are processed serially, words are processed in parallel. This allows for multiple words to be processed simultaneously, but each word is handled one bit at a time.

Word Serial and Bit Parallel (WSBP): It is found in most existing computers and has been called Word Slice processing because one word of an 'n' bit is processed at a time. In this approach, words are processed serially, but each word is processed in parallel at the bit level. This enables all bits of a single word to be processed simultaneously while each word is handled sequentially.

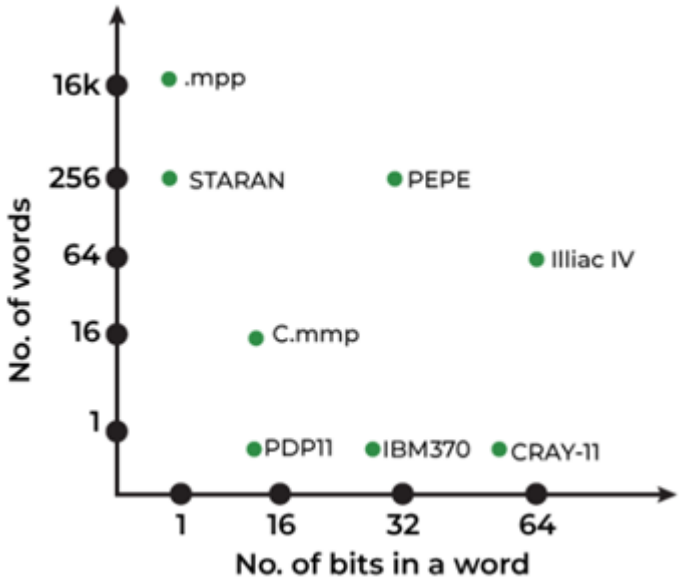


Fig 3.2.8 Processor classified according to Feng's Classification

Word Parallel and Bit Parallel (WPBP): This is known as fully parallel processing, in which an array of n x m bits is processed at one time. This method achieves full parallelism by processing both words and bits in parallel. Multiple words are processed simultaneously, and each word is processed entirely in parallel at the bit level, resulting in the highest degree of parallel processing.

Thus, according to Feng's Classification, processors can be categorised and represented graphically by plotting the number of bits against the number of words processed in

parallel. Figure 3.2.8 below illustrates this graphical representation.

3.2.5 Wolfgang Handler Classification

Wolfgang Handler's classification scheme, introduced in 1977, categorises computer architectures based on their control and processing units.

3.2.5.1 Wolfgang Handler Classification: Parallelism Versus Pipelining

Handler has proposed the Parallelism Versus Pipelining classification scheme to identify the parallelism degree and the pipeline degree of a computer's hardware structure. He divided the computer into three subsystem levels.

1. Processor Control Unit (PCU): Each PCU corresponds to one processor or one CPU.
2. Arithmetic Logic Unit (ALU): ALU is equivalent to the processing element (PE).
3. Bit Level Circuit (BLC): BLC corresponds to the combinational logic circuit required for 1-bit operations in ALU.

Each Processor Control Unit corresponds to one processor or one central processing unit. The Arithmetic Logic Unit is equivalent to Processor Element (PE). The Bit Level Circuit unit corresponds to a combinational logic circuit that is needed to perform 1-bit operations in the Arithmetic Logic Unit. Apart from the three types of scientists who introduced the Classification of the parallel model, two types of models are also available.

Flynn's classification is straightforward to apply, whereas Handler's classification is more complex. The use of numerical nomenclature in Handler's classification makes it abstract and, therefore, challenging to grasp directly. Handler's classification focuses heavily on describing pipelines and chains. While it effectively describes parallelism within a single processor, it does not adequately address the diverse parallelism found in multiprocessor systems.

3.2.6 Structural Classification: Coupling Between Processing Elements

Processors classified according to Feng's Classification can be represented graphically by plotting the number of bits against the number of words processed in parallel. The figure below illustrates this graphical representation. The Coupling refers to how Processor Elements incorporate this model of two types,

1. Loosely coupled: The degree of Coupling between the Processor Element is less. It consists of multiple processors with their own local memory, connected through a communication network. Each processor operates independently, with minimal data exchange, making the system flexible and scalable. This type of architecture is ideal for distributed computing environments where tasks can be performed in parallel without frequent communication. Figure 3.2.9 shows a generic block diagram of a loosely coupled multiprocessor.

2. Tightly coupled: Coupling between the Processor Elements is high, and they share a standard memory. This organisation has processors that share a common memory and are connected through a high-speed bus or other interconnects. These systems allow for close cooperation and rapid data sharing between processors, making them suitable for tasks requiring high inter-processor communication and synchronisation. Tightly coupled architectures are often used in applications demanding high performance and low latency, such as scientific computing and real-time processing. Figure 3.2.10 shows a generic block diagram of a tightly coupled multiprocessor.

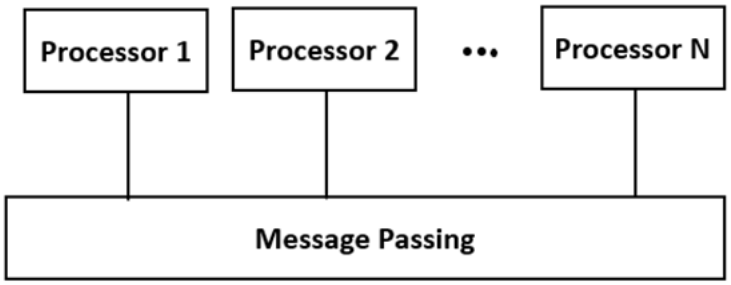


Fig 3.2.9 Generic Block Diagram of a Loosely Coupled System

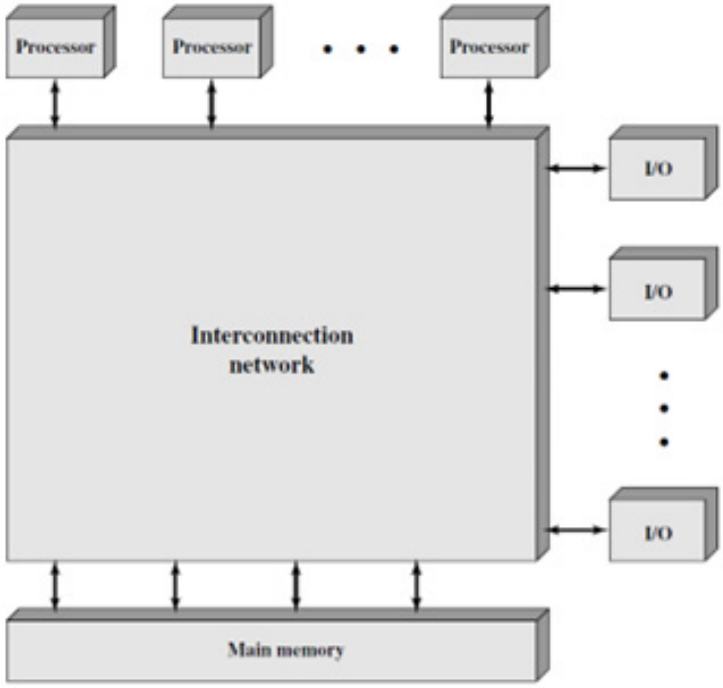


Fig 3.2.10 Generic Block Diagram of a Tightly Coupled Multiprocessor

3.2.7 Classification based on the mode of accessing memory

Classification based on the mode of accessing memory involves categorizing different memory access methods used in computer systems. Understanding these modes helps in designing efficient memory systems and optimising overall computer performance.

1. **Uniform memory access parallel computers (UMC):** The shared memory computer system processors share a common global address space. In UMA systems, all processors share the same physical memory and have equal access times to any memory location. This uniformity simplifies programming and memory management since any processor can access any memory location with the same latency. UMA systems are often found in symmetric multiprocessors (SMP), where each processor is connected to a shared memory via a common bus. While UMA is easier to implement and manage, it can suffer from memory contention issues as the number of processors increases, leading to potential bottlenecks.

Thus, in UMA, all processors have access to all parts of the main memory using loads and stores. The memory access time of a processor to all regions of memory is the same. The access times experienced by different processors are the same.

2. **Non-uniform memory access parallel computers (NUMA):** Each processor may own local memory and may not share a standard memory. The objective of NUMA is to maintain a transparent system-wide memory while permitting multiple multiprocessor nodes, each with its own bus or other internal interconnect system. NUMA systems address the scalability limitations of UMA by allowing processors to access local memory faster than remote memory. In NUMA architectures, the overall memory is divided into distinct regions, each associated with a particular processor or a group of processors. Accessing local memory is quicker while accessing memory from another region incurs a higher latency. This approach reduces the memory contention problem and enhances performance for applications with significant local memory access patterns. However, NUMA requires more complex programming techniques to optimize memory access and minimize the performance penalties associated with remote memory access.

The UMA architecture provides uniform latency for processors accessing memory, making it less advantageous for local memory access since the delay remains consistent. Conversely, in NUMA architecture, each processor has its own dedicated memory, reducing delay when accessing local memory. The latency varies depending on the distance between the CPU and memory. Nonetheless, NUMA offers better performance compared to the UMA design.

Recap

- Parallel computing involves the simultaneous use of multiple processors or computers to perform complex computations more efficiently and quickly by dividing tasks into smaller, concurrent sub-tasks.
- Computer architecture can be classified based on control flow and data flow.
- Flynn's Classification is a system used to categorize computer architectures based on their instruction and data streams. They are:
  - Single-Instruction Single-Data Streams (SISD)
  - Single-Instruction Multiple-Data Streams (SIMD)
  - Multiple-Instruction Single-Data Streams (MISD)
  - Multiple-Instruction Multiple-Data Streams (MIMD)
- Feng's Classification categorizes computer architectures based on their parallelism and data handling capabilities. They are:
  - Word Serial and Bit Serial (WSBS)
  - Word Parallel and Bit Serial (WPBS)
  - Word Serial and Bit Parallel(WSBP)
  - Word Parallel and Bit Parallel (WPBP)
- Handler's Classification categorizes computer architectures based on their functional characteristics and operational capabilities.
- An instruction stream refers to a sequence of machine instructions that dictate the operations and tasks performed by a computer processor
- A data stream refers to a continuous flow or sequence of data elements.
- UMA stands for Uniform Memory Access. It refers to a computer architecture where multiple processors share a single, common memory space.
- NUMA stands for Non-Uniform Memory Access. It refers to a computer architecture where processors in a multiprocessor system have local access to their own memory.

Objective Type Questions

1. What is the term used to describe the efficiency of an algorithm that performs the task in the smallest number of operations
2. Give the name of the technique used for dividing one task into multiple subtasks and executing the subtasks in parallel with multiple hardware units.
3. How many types of classification are there in the Flynn computing architecture classification scheme?
4. Who introduced bit and word-level architectural classification schemes.
5. What are the design criteria for Handler's Classification, given that the system incorporates pipelining at system levels?
6. Name the important classification algorithms proposed by scientist Michael J. Flynn.
7. What are the criteria for Flynn Classification on Multiplicity?
8. Name one of the axes in M.J. Flynn's taxonomy of parallel computers.
9. Can the two streams of Flynn's taxonomy be either serial or parallel? True/ False
10. What is the basis of the classification of M.J. Flynn's parallel processing?
11. What kind of an organization does SIMD represent?
12. What kind of architecture is SISD_architecture?
13. What kind of classification do the Single-Instruction Single-Data Streams belong to?
14. What kind of processor does the SISD category present?
15. Which category was predicted by Flynn's taxonomy but not yet built?
16. What does SIMD stand for?
17. What type of architecture is not suitable for realising SIMD
18. What is the term that refers to the maximum number of binary digits that can be processed within a unit of time by a computer system?
19. How many subsystem levels are in Wolfgang Handler Classification?

Answers to Objective Type Questions

1. Time complexity
2. Parallel computing
3. Four
4. Feng's Classification
5. Degree of parallelism
6. Four
7. Instruction stream and Datastream
8. Instruction stream
9. False
10. Multiple Instructions and data
11. Single instruction is applied to multiple pieces of data simultaneously.
12. Von Neumann
13. Flynn's Taxonomy.
14. Traditional uniprocessor
15. MISD
16. Single Instruction Multiple Data
17. Von Neumann
18. Throughput
19. four

Assignments

1. Explain Flynn’s computer system classification with examples.
2. Discuss how parallelism can be achieved using Feng’s and Handler’s classification.

Suggested Reading

1. Hwang, Kai, and A. Faye. "Computer architecture and parallel processing." (1984).
2. Mano, M. Morris. “Computer system architecture.” Prentice-Hall of India, 2003.



UNIT 3

Pipelining - Introduction and Classification

Learning Outcomes

By the completion of the unit, the learners will be able:

- understand the basic principles of pipelining and vector processing
- identify the stages involved in a pipeline.
- classify different types of pipelines based on the level of processing - instruction level and data level pipelines.
- classify pipelines based on the type of instruction and data they process - scalar, vector and superscalar pipelines.

Prerequisites

Pipelining is similar to the concept of a water pipe, where water continues to flow without waiting for the previous water to exit the pipe, as illustrated in Figure 3.4.1. This model will help to increase clock speed or reduce the power consumption at the same speed in a parallel processing system.

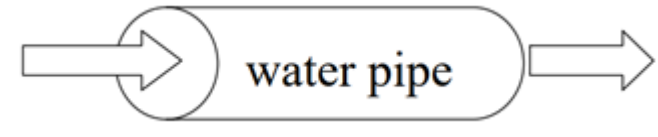


Fig 3.3.1 Water flows through a water pipe

This pipelined execution of the instruction is one of the important methods of increasing the speed of Processing Elements. Parallel processing is a method in computing in which two or more processors handle separate parts of an overall task. In the parallel processing architecture, a pipeline consists of a chain of processing elements arranged so that the output of each element is the input of the next. Parallel processing and pipelining can also be combined to increase instruction throughput and the execution speed of the system processors. Thus, both parallel and pipelined methods use the concurrency present in the task but in different ways.

Keywords

Pipeline computer, Pipeline architecture, Instruction pipeline design, Arithmetic pipeline design

Discussion

3.3.1 Pipelining and Vector Processing

The pipelining process is a data execution and implementation technique in parallel data processing, where multiple instructions overlap in execution. Vector processing is one of the parallel data processing methods in computer architecture.

In parallel computing, vector processing is a method that can implement instructions using one-dimensional arrays. The one-dimensional arrays of data are called vectors. The vector processing is done in the CPU with the help of a vector processor.

3.3.2 Introduction to Pipelining

Pipelining is a particularly effective way of organising concurrent activity in a computer system. Pipelining for instruction execution is similar to the construction of a factory assembly line for product manufacturing.

The basic idea is to arrange the instruction execution process into a collection of smaller functions that can be independently performed by subsystems in the processor implementation. For pipelining, we will organize these subsystems (which are called pipeline stages), implementing the instruction interpretation process into concurrently executing systems.

In five-stage pipelining processor organisations, five clock cycles complete the execution of each instruction. The five stages are Fetch, Decode, Compute, Memory, and Write.

Instruction is fetched in the first cycle and moves through the remaining stages in the following cycles. In the second cycle, instruction is fetched while instruction is in the Decode stage, where its operands are also read from the register file. In the third cycle, instruction is fetched while instruction is in the Decode stage, and instruction is in the Compute stage, where an arithmetic or logic operation is performed on its operands. This overlapping pattern of execution would be possible for all instructions. Each instruction takes five cycles to complete its execution, and instructions are completed at the rate of one per cycle.

A Pipeline data processing method is a chain of stages where certain operations are done in each stage. The process is not marked finished until all the sub-operations are passed through all stages. The pipeline process has some advantages and disadvantages. A situation called data hazard occurs when operand data is not available.

Advantages of Pipelining

- Instruction throughput increases.

- An increase in the number of pipeline stages increases the number of instructions executed simultaneously.
- Faster ALU can be designed when pipelining is used.
- Pipelined CPUs work at higher clock frequencies than the RAM.
- Pipelining increases the overall performance of the CPU.

Disadvantages of Pipelining

- The design of the pipelined processor is complex.
- Instruction latency increases in pipelined processors.
- The throughput of a pipelined processor is difficult to predict.
- The longer the pipeline, the worse the problem of hazard for branch instructions.

Three types of following factors affect the construction of pipeline processor systems in the parallel processing architecture, they are:

1. Level of Processing
2. Pipeline configuration
3. Type of Instruction and data

3.3.3 Classification according to the level of processing

Based on the classification levels, computer pipeline data processing operations are classified as

1. Instruction Execution Pipeline
2. Arithmetic Operation Pipeline

3.3.3.1 Instruction Execution Pipeline Process

Instruction pipelining is similar to an assembly line in a manufacturing plant. An assembly line leverages the fact that a product undergoes multiple production stages. By organising the production process into an assembly line, different stages can be worked on simultaneously. This method is called pipelining because, similar to a pipeline, new inputs are introduced at one end while previously accepted inputs are processed and emerge as outputs at the other end.

To apply this concept to instruction execution, we must acknowledge that an instruction goes through several stages. For instance, Figure 3.3.2 divides the instruction cycle into 10 sequential tasks. This division clearly presents an opportunity for pipelining.

Consider a simple approach that subdivides instruction processing into two stages: fetching and executing an instruction. During the execution of an instruction, there are periods when the main memory is not accessed. This idle time can be utilized to fetch

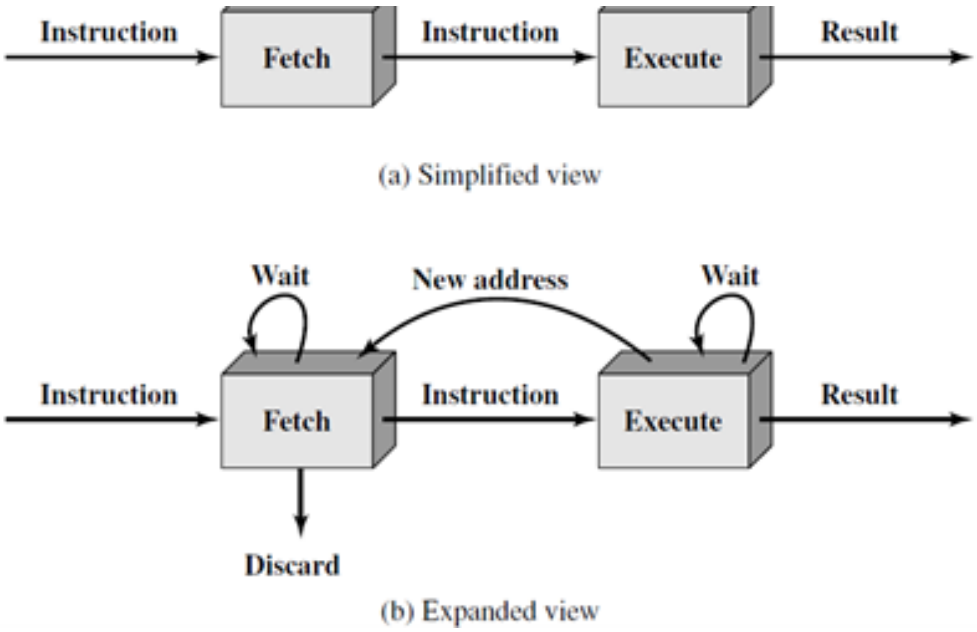


Fig 3.3.2 Two-stage Instruction Pipeline

the next instruction concurrently with the execution of the current one. Figure 3.3.2a illustrates this method. The pipeline consists of two independent stages. The first stage fetches and buffers an instruction. When the second stage is available, the first stage passes the buffered instruction to it. While the second stage executes the instruction, the first stage uses any idle memory cycles to fetch and buffer the next instruction. This technique is known as instruction prefetch or fetch overlap. Note that this method involves instruction buffering and requires additional registers. Generally, pipelining necessitates registers to store data between stages.

It is evident that this process will accelerate instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be cut in half. However, a closer examination of this pipeline (Figure 3.3.2b) reveals that achieving a doubling of the execution rate is unlikely for two reasons:

1. The execution time is typically longer than the fetch time. Execution involves reading and storing operands and performing some operations. As a result, the fetch stage may have to wait before it can empty its buffer.

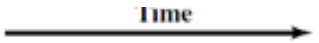
2. A conditional branch instruction makes the address of the next instruction to be fetched unknown. Consequently, the fetch stage must wait until it receives the next instruction address from the execute stage. This may cause the execute stage to wait while the next instruction is fetched.

While these factors reduce the potential effectiveness of the two-stage pipeline, some speedup is still achieved. To further enhance speedup, the pipeline must include more stages. Consider the following breakdown of instruction processing:

- **Fetch instruction (FI)**: Read the next expected instruction into a buffer.
- **Decode instruction (DI)**: Determine the opcode and operand specifiers.

- **Calculate operands (CO)**: Calculate the effective address of each source operand, which may involve displacement, register indirect, indirect, or other forms of address calculation.
- **Fetch operands (FO)**: Fetch each operand from memory, although operands in registers do not need to be fetched.
- **Execute instruction (EI)**: Perform the indicated operation and store the result, if any, in the specified destination operand location.
- **Write operand (WO)**: Store the result in memory.

Using this approach, Figure 3.3.3 demonstrates that a six-stage pipeline can significantly reduce the execution time for 9 instructions from 54 time units to 14 time units.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Fig 3.3.3 Timing Diagram for Instruction Pipeline Operation

John Von Neumann Execution Architecture

In the First Stage, the control unit of a processor fetches the instruction from memory. In the Second Stage, the control unit decodes the instructions to determine the type of operation to be performed.

In the Third Stage, the operation requires operands. The control unit also determines the address of each operand and fetches them from memory. In the Fourth Stage, the operation is performed on the operands, and finally, the result is stored in the specified location.

Figure 3.3.4 illustrates the effects of a conditional branch using the same program as

Figure 3.3.3. Assume that instruction 3 is a conditional branch to instruction 15. Until the branch instruction is executed, it is unknown which instruction will follow next. In this example, the pipeline loads the next instruction in sequence (instruction 4) and continues. In Figure 3.3.3, the branch is not taken, allowing the pipeline to achieve the full performance benefit. However, in Figure 3.3.4, the branch is taken, which is only determined at the end of time unit 7. At this point, the pipeline must be cleared of any instructions that are not useful. During time unit 8, instruction 15 enters the pipeline. Consequently, no instructions are completed during time units 9 through 12, representing the performance penalty incurred due to the inability to anticipate the branch. Figure 3.3.5 shows the logic required for pipelining to manage branches and interrupts.

Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments.

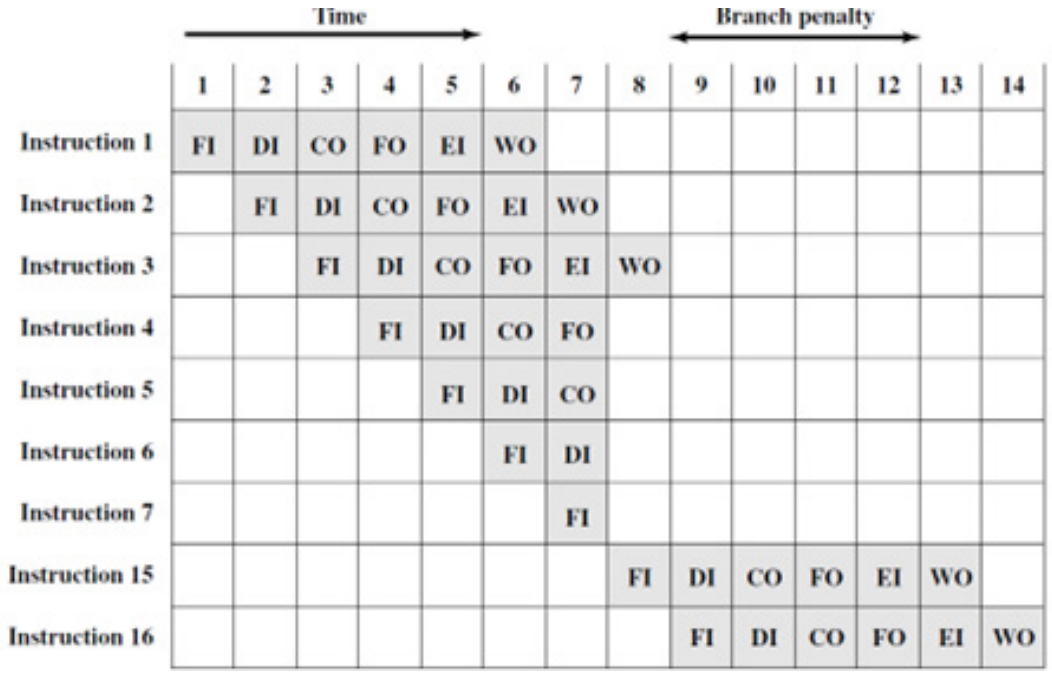


Fig 3.3.4 The Effect of a Conditional Branch on Instruction Pipeline Operation

The instruction fetch segments can be implemented in a first-in-fist-out (FIFO) manner. These instructions are inserted into the FIFO buffer and executed on a first-in-first-out basis. Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. For all the computer systems to perform pipeline instructions, the following six sequences of steps are used. Figure 3.3.5 indicates the logic needed for pipelining to account for branches and interrupts.

Six different stages in a Pipeline instruction are

1. Instruction fetch (IF): Retrieval of instructions from main memory.
2. Instruction decoding (ID): Identification of the operation to be performed.
3. Address Generator (AG): generates the address.

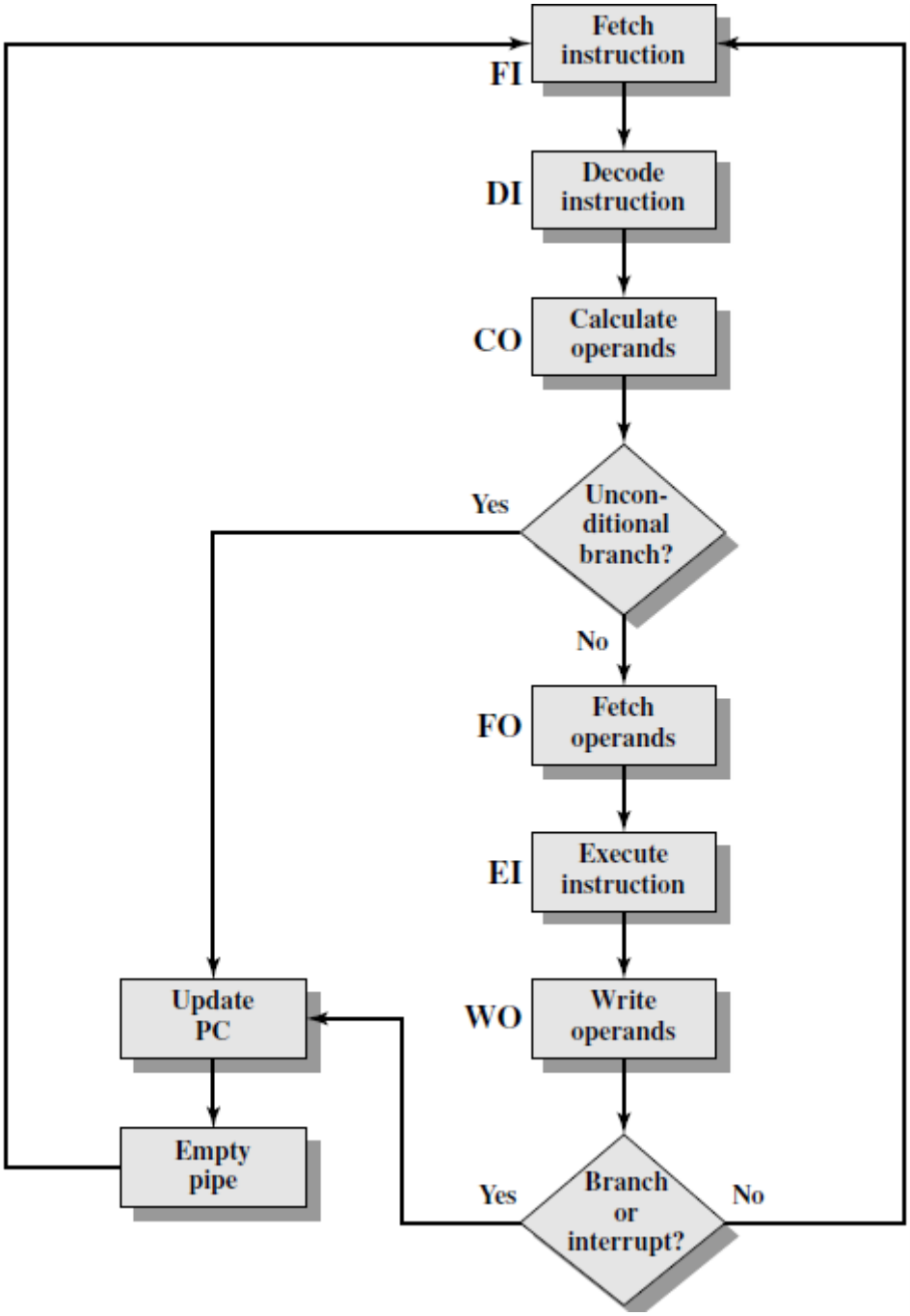


Fig 3.3.5 Six-Stage CPU Instruction Pipeline

4. Data Fetch (DF): Data Fetch fetches the operands into the data register.
5. Execution (EX): Operating on the operands.
6. Write-back (WB): Updating the destination operands and writing the result to the register.

All these sets of instructions mentioned above are not required for completing the pipeline process. These six stages differ based on the system hardware components.

All of the six phases are different in their performance, and each step is considered

independent. The additional property of these phases is overlapping layers. All these separate phases are overlapped with each other so that multiple operations can be performed simultaneously. Instruction pipelining is an effective method for boosting performance, but it necessitates meticulous design to achieve the best results while keeping complexity manageable.

3.3.3.2 Arithmetic Operation Pipeline Process

These processes are used to implement floating-point operations, multiplication of fixed-point numbers, and similar calculations. The example process of arithmetic operations on a pipeline unit for floating-point addition and subtraction. Thus, the pipelines used for arithmetic operations are known as arithmetic pipelines.

The floating-point addition and subtraction were carried out in four segments.

1. **Compare the exponents**: First, the exponents of the floating-point numbers involved in the operation are compared to determine the alignment needed for further processing.

2. **Align the mantissa**: After determining the necessary alignment based on the exponent comparison, the mantissas (significant digits) of the numbers are aligned accordingly.

3. **Add or subtract mantissa**: Once aligned, the actual addition or subtraction of the mantissas takes place. This step involves performing the arithmetic operation on the aligned mantissas according to the specified operation (addition or subtraction).

4. **Produce the result**: Finally, the result of the arithmetic operation-whether it's the sum or difference of the mantissas-is produced as the final output of the pipeline unit.

Figure 3.3.6 shows the pipeline for floating point addition and subtraction.

3.3.4 Classification According to Pipeline Configuration

According to the configuration of a pipeline, two important pipeline classifications are there.

1. Unifunction Pipelines: In a unifunctional pipeline, a fixed and dedicated function is performed throughout the entire pipeline process. This means that the pipeline is designed to execute a specific task repeatedly and consistently. For example, in a processing unit dedicated to arithmetic operations, each stage of the pipeline might handle a specific operation, such as addition or multiplication, exclusively.

2. Multifunction Pipelines: Conversely, multifunction pipelines are designed to perform different functions at different times within the same pipeline structure. This flexibility allows the pipeline to handle diverse tasks depending on the input and processing requirements. For instance, a multifunction

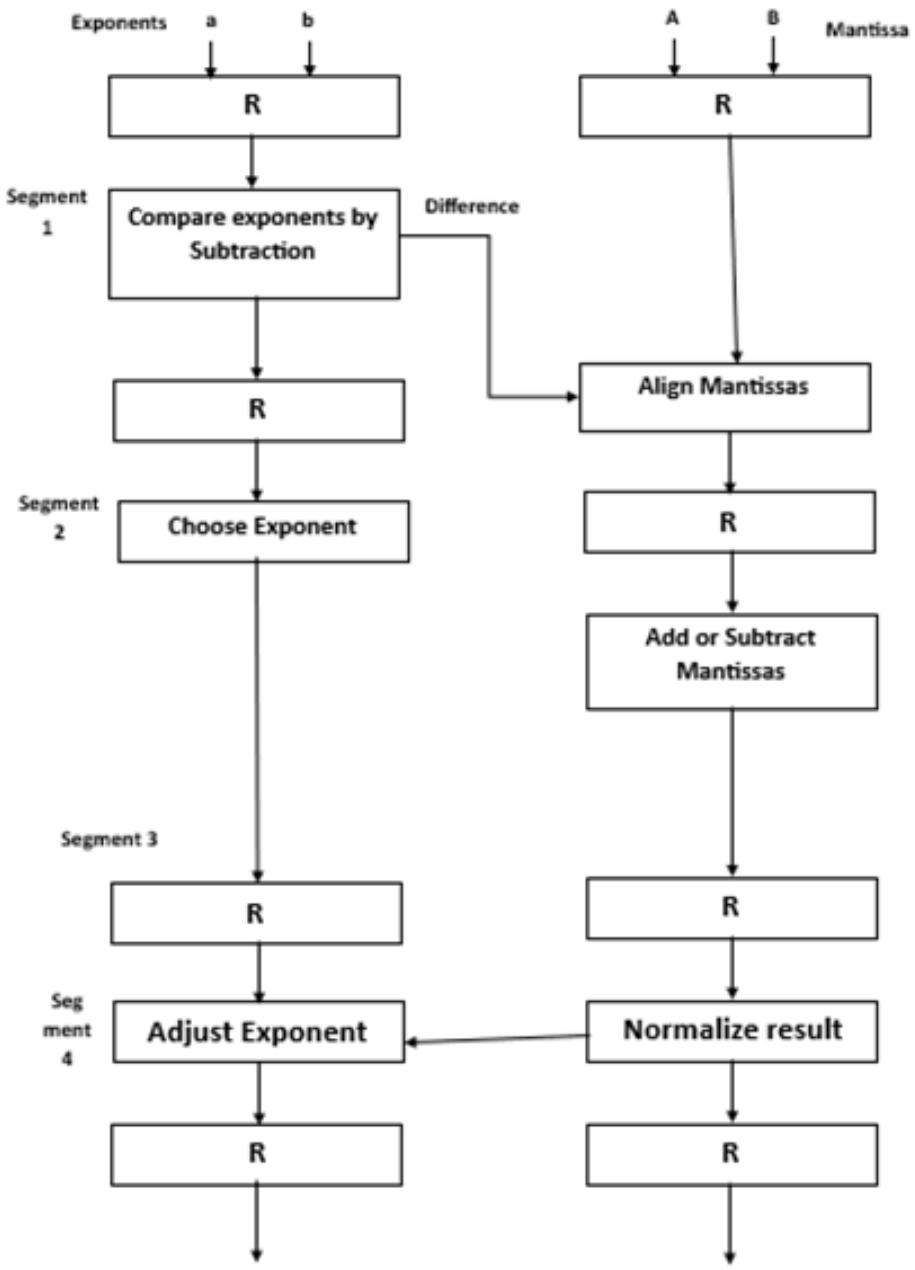


Fig 3.3.6 Pipeline for Floating Point Addition and Subtraction

pipeline in a general-purpose computing environment might handle tasks such as arithmetic operations, data transfers, and logic operations within the same pipeline structure, adapting dynamically based on the instructions being processed.

3.3.5 Classification According to the Type of Instruction and Data

According to the types of instruction and data, the following types of parallel pipelines are available.

- Scalar Pipelines: Scalar pipelines are designed to process scalar operands

using repeated scalar instructions. In this type of pipeline, each stage operates on individual data elements (scalars) independently. Scalar pipelines are well-suited for tasks where operations are performed sequentially on single data items, such as arithmetic calculations or simple logic operations.

- Vector Pipelines: On the other hand, vector pipelines are optimised for handling vector instructions that operate over vector operands. These pipelines are capable of simultaneously processing multiple data elements (vectors) in parallel across their stages. Vector pipelines are particularly effective for tasks that involve parallel computation on arrays or matrices, such as matrix multiplication, image processing, or scientific simulations where operations are applied uniformly across large sets of data.

Recap

- Vector processing is one of the parallel data processing methods in computer architecture
- Pipelined process execution involves breaking down tasks or operations into sequential stages, with each stage managed by specialized processing units.
- It increases throughput by allowing multiple instructions to overlap in execution stages
- An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments.
- Arithmetic Operation Pipelines are used to implement floating-point operations, multiplication of fixed-point numbers, and similar calculations
- In unifunction pipelines, a fixed and dedicated function is performed through a pipeline.
- In multifunction pipelines, different functions at different times are performed through the pipeline, and those types of pipelines are known as multifunction pipelines.
- Scalar pipelines process scalar operands of repeated scalar instructions.
- Vector pipelines process vector instructions over vector operands.

Objective Type Questions

1. How many types of Pipelining exist?
2. Which type of pipeline is used for floating point operations?
3. "Cycle time of the processor is reduced" is one of the disadvantages of Pipelining. True/False
4. Instruction throughput increases is an advantage of pipelining. True/False
5. In the Arithmetic Pipeline, how many parts are used for floating-point addition and subtraction?
6. Which strategy is the instruction prefetch used?
7. What is the name of a situation where the data of operands are not available?
8. What is the name of the technique where multiple instructions are overlapped during execution?
9. What are the two categories of a pipeline?
10. What is the core element of parallel processing?
11. What is the name of the implementation technique whereby multiple instructions are overlapped in execution?
12. Pipeline implement for calculating operand and decoding instruction. Yes/No
13. What is the situation that occurs in the pipeline due to hazards in pipelines?

Answers to Objective Type Questions

1. 2
2. Arithmetic pipeline
3. False
4. True
5. 4

6. Pipeline strategy
7. Data hazard
8. Pipelining
9. Instruction pipeline and arithmetic pipeline
10. CPU
11. pipelining
12. Yes
13. pipeline stall

Assignments

1. What are the disadvantages of increasing the number of stages in pipelined processing?
2. Find out five major implementation limitations of the pipelining technique.

Suggested Reading

1. Hwang, Kai, and A. Faye. "Computer architecture and parallel processing." (1984).
2. Mano, M. Morris. Computer system architecture. Prentice-Hall of India, 2003



Vector Processing and Array Processors

Learning Outcomes

By the completion of the unit, the learner will be able to:

- familiarise with the concept of pipeline computers
- narrate the basics of vector processing
- understand the Array Processor
- explain the basics of vector instructions.

Prerequisites

Pipelining is an implementation technique where multiple instructions are overlapped in execution. The computer pipeline is divided into stages. Know the parallel pipeline process and understand the concepts of pipeline process computer architecture. Parallel computing is a form of computation in which many calculations are carried out simultaneously. Figure 3.4.1 illustrates a potential method of dividing the execution unit into eight functional units that operate simultaneously.

Operating on multiple data in just one instruction is also called Single Instruction Multiple Data also termed as Vector (Array) instructions. These instructions are stored in the vector register. The vector instruction performs the same operation on the different data sets repeatedly.

Keywords

Pipeline computer, Vector Processing, Vector Instruction Array Processors

Discussion

3.4.1 Vector Processing

In this section, we explain the basic concepts of vector processing concepts. Multiple data items processing with the conventional computer (including the scalar operator) takes days or even weeks to complete the task. Those complex instructions can be solved with the help of vector processors.

We distinguish vector processing from scalar processing based on the characteristics of vector instruction. In computing, a vector is either a pointer or an array with only one dimension. The vector processor is the central processing unit (CPU) that carries out an instruction set containing instructions that operate one-dimensional arrays. Computers with vector processing capabilities are in demand in many specialized applications. Some of the application areas of vector processing is given below.

- Long-range weather forecasting
- Petroleum explorations
- Medical diagnosis
- Aerodynamics simulations
- Artificial intelligence and expert systems and image processing

The central processing unit is part of a computer that allows the laptop to perform different function-oriented operations. These operations are at least on a comprehensive level. The CPU processes the instructions of various programs. This unit breaks instruction into individual parts, executes and generates results. All the processors were developed with the help of scalar and vector processing.

3.4.1.1 Vector Operations

Many scientific problems involve performing arithmetic operations on large arrays of numbers, often represented as vectors and matrices of floating-point numbers. A vector is an ordered collection of data items arranged in a one-dimensional array. For instance, a vector V of length n is depicted as a row vector V=[V1 V2 V3 ... Vn]. Alternatively, it can be represented as a column vector if the data items are listed vertically. In a conventional sequential computer, operations are typically carried out on operands one at a time. Therefore, computations involving vectors necessitate breaking down operations into individual calculations using subscripted variables. The element Vi of vector V is denoted as V(i), where i denotes a memory address or register location where the corresponding number is stored.

Matrix multiplication is one of the most computationally intensive operations performed in computers with vector processors. The multiplication of two n x n matrices consists of n^2 inner products or n^3 multiply-add operations. An n x m matrix of numbers has n rows and m columns and may be considered as constituting a set of n row

vectors or a set of m column vectors. Consider, for example, the multiplication of two 3 x 3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The product matrix C is a 3 x 3 matrix whose elements are related to the elements of A and B by the inner product:

$$C_{ij} = \sum_{k=1}^{3} a_{ik} * b_{kj}$$

For example, the number in the first row and first column of matrix C is calculated by letting i = 1, j = 1, to obtain

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

Figure 3.4.1 shows the instruction format for the vector processor.

Operation code	Base address source 1	Base address source 2	Base address destination	Vector Length

Fig 3.4.1 Instruction format for Vector Processor

This involves performing three multiplications and (starting with C11 set to 0) three additions. The total number of multiplications or additions needed to calculate the matrix product is 9 x 3 = 27. If we treat the combined multiply-add operation c + a x b as a single cumulative operation, computing the product of two n x n matrices requires n^3 multiply-add operations. This computation comprises n^2 inner products, each of which necessitates n multiply-add operations, assuming that c is initialized to zero before computing each element in the product matrix.

In general, the inner product consists of the sum of k product terms of the form C = $A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4 +$.....+ A_kB_k; In a typical application, k may be equal to 100 or even 1000. The inner product calculation on a pipeline vector processor is shown in Figure 3.4.2.

The values of A and B are stored either in memory or in processor registers. The floating-point multiplier pipeline and the floating-point adder pipeline each consist of four segments. Initially, all segment registers in both the multiplier and adder pipelines are set to 0. Consequently, the output of the adder remains 0 for the first eight cycles until both pipelines are fully populated.

Pairs of Ai and Bi are brought in and multiplied at a rate of one pair per cycle. After

the first four cycles, the products start to be added to the output of the adder. Over the subsequent four cycles, zeros are added to the products entering the adder pipeline. By the end of the eighth cycle, the first four products (A1 B1 through A4 B4) are in the four segments of the adder, and the next four products (A5 B5 through A8 B8) are in the segments of the multiplier.

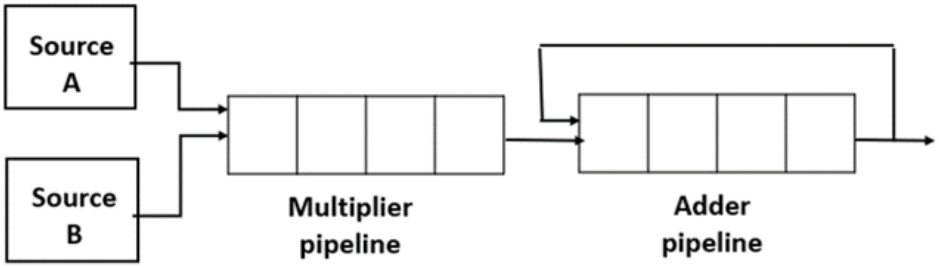


Fig 3.4.2 Pipeline for calculating an inner product

At the beginning of the ninth cycle, the output of the adder holds A1 B1, and the output of the multiplier holds A5 B5. Therefore, the ninth cycle initiates the addition of A1 B1 + A5 B5 in the adder pipeline. Similarly, the tenth cycle commences with the addition of A2 B2 + A6 B6, and so forth. This sequencing divides the summation process into four distinct sections as follows:

$$C = A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \ldots\ldots$$
$$+ A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14}\ldots\ldots$$
$$+ A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15}\ldots\ldots$$
$$+ A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16}\ldots\ldots$$

Once all product terms have been processed, the system fills the multiplier pipeline with four zeros. Subsequently, the adder pipeline contains one partial product in each of its four segments, corresponding to the sums listed in the four rows of the equation above. These four partial sums are then combined to produce the final sum.

Scalar Processors: These are the most fundamental types of processors. They process one item at a time. The scalar processor works on integers and floating-point numbers. All the instructions in the scalar processing method are handled sequentially so that this method can take up some time.

Vector Processor: A vector processor aggregates multiple data points, processing each in turn. It takes complicated tasks and is broken down into smaller jobs that will respond to a similar instruction. Vector processors are efficient when it comes to getting things done, but this efficiency can cause other parts of the computer system to go slowly.

3.4.1.2 Memory Interleaving

Pipeline and vector processors often require simultaneous memory access from multiple sources. For example, an instruction pipeline may need to fetch an instruction and an operand simultaneously from different segments. Similarly, an arithmetic pipeline

typically requires multiple operands to enter the pipeline simultaneously. Instead of using two separate memory buses for this purpose, memory can be divided into multiple modules connected to shared memory addresses and data buses. Each memory module consists of a memory array along with its own address register (AR) and data register (DR). Figure 3.4.3 illustrates a memory unit with four such modules, where each memory array has dedicated AR and DR. The address registers receive inputs from a common address bus, and the data registers communicate bidirectionally with a shared data bus. Using the two least significant bits of the address, it distinguishes between the four modules. This modular system allows one module to initiate memory access while others are in the process of reading or writing data. Each module can respond to memory requests independently, regardless of the state of the other modules, thereby enhancing overall memory access efficiency and flexibility.

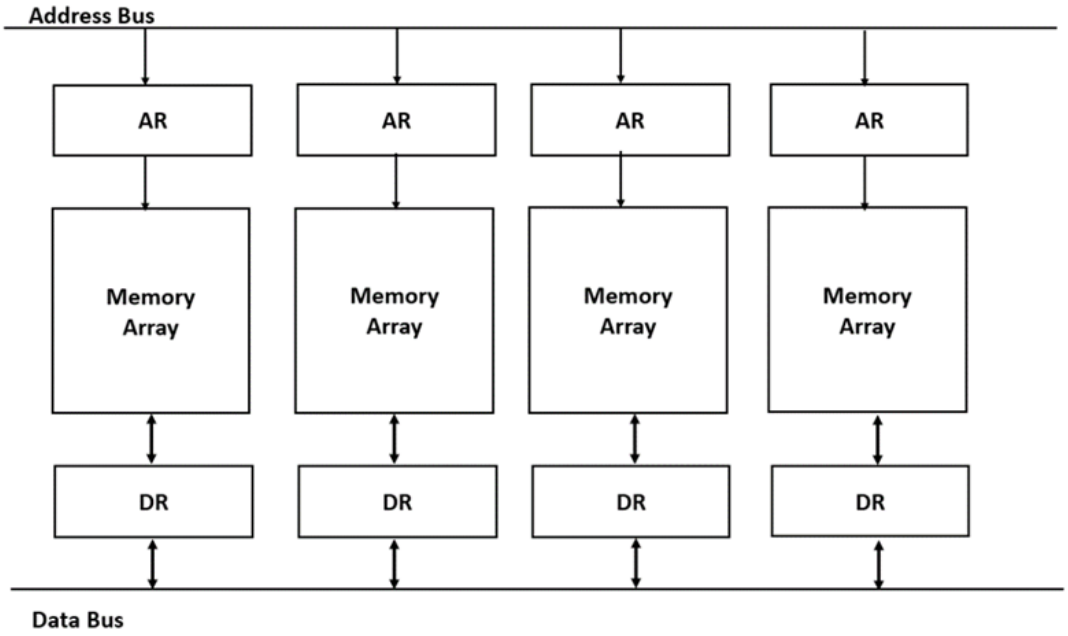


Fig 3.4.3 Multiple Module Memory Organization

Modular memory offers the advantage of enabling interleaving, a technique where different sets of addresses are assigned to distinct memory modules. For instance, in a two-module memory system, one module may handle even addresses while the other manages odd addresses. When the number of modules is a power of 2, the least significant bits of the address determines which memory module is selected, while the remaining bits specify the exact location to access within that chosen module.

Modular memory proves beneficial in systems employing pipeline and vector processing. For instance, a vector processor utilising an n-way interleaved memory can concurrently fetch n operands from n distinct modules. This interleaving of memory access helps reduce the effective memory cycle time by a factor close to the number of modules. Similarly, a CPU equipped with an instruction pipeline can optimise performance by leveraging multiple memory modules. This allows each segment within the pipeline to access memory independently without being hindered by memory accesses from other segments.

3.4.4 Performance Measures of Vector Processing

The vector processing mechanism has two necessary time-related overhead properties: setup time and flushing time. The time required to route the vector to the functional unit is called the setup time, and flushing time is the time duration that a vector instruction from the initial system processing level to the first result is out of the pipeline.

For the performance of vector processing, the following four points are to be followed.

1. Improving the vector instruction: Improve the vector instruction by reducing memory access and maximising resource utilisation.

2. Integrate the scalar instruction: scalar instruction of the same type must be integrated as a batch as it will reduce the overhead of reconfiguring the pipeline again and again.

3. Algorithm: Choose the algorithm that would work faster for vector pipelined processing

4. Vectorizing Compiler: A vectorising compiler must regenerate the parallelism by using a higher-level programming language. In advanced programming, the four stages are identified in the development of parallelism. Those are

    1. Parallel Algorithm(A)
    2. High-level Language(L)
    3. Efficient object code(O)
    4. Target machine code (M)

3.4.5 Supercomputers

A commercial computer with vector instructions and pipelined floating-point arithmetic operations is commonly known as a supercomputer. Supercomputers are exceptionally powerful machines designed primarily for scientific computations, requiring high performance. To enhance their speed, components are densely packed to minimise the distance electronic signals need to travel. Additionally, supercomputers employ specialised techniques to dissipate heat from circuits efficiently, preventing overheating issues due to their proximity and high operational intensity.

The instruction set of supercomputers contains the standard data transfer, data manipulation, and program control instructions of conventional computers. This IS augmented by instructions that process vectors and combinations of scalars and vectors. A supercomputer is a computer system best known for its high computational speed, fast and large memory systems, and extensive parallel processing. It is equipped with multiple functional units with its pipeline configuration. Although the supercomputer is capable of general-purpose applications found in all other computers, it is specifically optimised for numerical calculations involving vectors and matrices of floating-point numbers.

Supercomputers are not suitable for the normal processing of a typical computer installation. They are limited in their use to several scientific applications, such as numerical weather forecasting, seismic wave analysis, and space research. They have limited use and limited market because of their high price.

A metric used to assess computers based on their capability to execute a specified number of floating-point operations per second is known as flops (floating-point operations per second). Megaflops denoted a million flops, while gigaflops denoted a billion flops. In typical supercomputers, which have a basic cycle time ranging from 4 to 20 nanoseconds (ns), the processor can perform one floating-point operation per cycle through a pipeline. This capability results in a performance range of 50 to 250 megaflops. This rate is sustained once operations begin producing results and excludes the initial setup time required for the pipeline.

The first supercomputer developed in 1976 was the Cray-1 supercomputer. It utilized vector processing with 12 separate functional units operating in parallel. Each functional unit was segmented to process incoming data through a pipeline. All functional units could work concurrently, utilizing over 150 registers in the CPU to store operands. During each clock cycle of 12.5 nanoseconds, the Cray-1 could perform a floating-point operation on two sets of 64-bit operands, achieving a rate of 80 megaflops while data was processed through the pipeline.

The Cray-1 had a memory capacity of 4 million 64-bit words, divided into 16 banks with a 50-nanosecond access time per bank. This configuration allowed for a memory transfer rate of 320 million words per second when all 16 banks were accessed simultaneously.

Subsequently, Cray Research expanded its supercomputer lineup with multiprocessor configurations like the Cray X-MP and Cray Y-MP. The Cray-2 supercomputer, introduced later, boasted a vector processing power twelve times greater than the Cray-1.

3.4.6 Array Processors

An array processor may be a processor that performs computations on large arrays of knowledge. Array processors also are referred to as multiprocessors or vector processors. The parallel processor system architecture has two sorts of array processors available, and they've attached array processors and Single Instruction multiple data organisation (SIMD) array processors.

All the overall purpose computers are adopting attached array processors for human computational work. Therefore, the SIMD array processor operates vector instructions through multiple functional units that respond to standard instructions.

An array processor has a number of Arithmetic Logic Units (ALU) that allow all the elements of an array to be processed at the same time. In an array processor, a single instruction is issued by a control unit, and that instruction is applied to a number of data sets at the same time.

Array processors perform a single instruction in multiple execution units in the same

clock cycle. The different execution units have the same instruction using the same set of vectors in the array. The array processor is capable of processing array elements and is suitable for scientific computations involving two-dimensional matrices.

Why use the Array Processor?

- Array processors increase the overall instruction processing speed.
- As most of the Array processors operate asynchronously from the host Central Processing Unit, it improves the system's overall capacity.
- Array Processors have their local memory, hence providing extra memory for systems with low memory.

3.4.7 Types of Array Processors

There are two types of array processors:

1. Attached Array Processors
2. Single Instruction Multiple Data Organization (SIMD) Array Processors

3.4.7.1 Attached Array Processors

An attached array processor may be a processor attached to a general-purpose computer. Its purpose is to reinforce and improve the performance of that computer in numerical computational tasks. It achieves high performance through multiprocessing with multiple functional units.

An attached array processor is designed as a peripheral for a conventional host computer. Its purpose is to enhance the computer's performance by providing vector processing for complex scientific applications.

It achieves high performance through parallel processing with multiple functional units. It includes an arithmetic unit containing one or more pipelined floating-point addresses and multipliers. The user can program the array processor to accommodate a variety of complex arithmetic problems.

The attached array processor has two interfaces:

1. An input-output interface to a standard processor
2. An Interface with local memory.

The input-output interface is connected with the array processor, and the array processor system has a local memory. In this array processor architecture, another computer called attached array processor systems is also included. It can complete a sizable computational task. Whenever the complicated task moves from the general computer to local memory, the attached processor will execute the complex task and transfer it into the main memory. In this way, the attached array processor can work. Figure 3.4.4 illustrates the diagram of the attached array processor with the host computer.

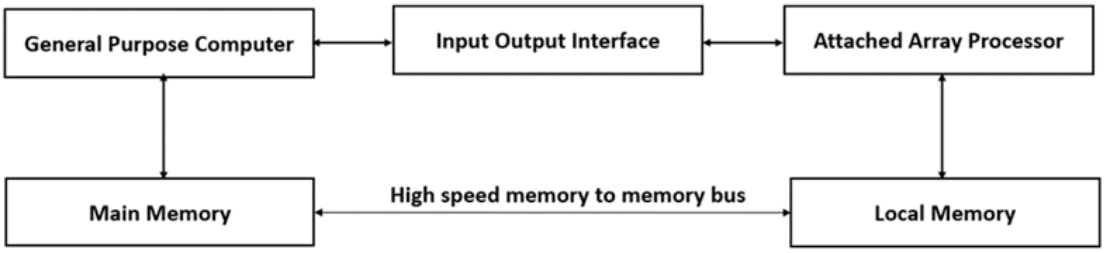


Fig 3.4.4 Attached array processor with the host computer

3.4.7.2 Single Instruction Multiple Data Organization (SIMD) Array Processors

Single Instruction Multiple Data Organization (SIMD) is the organization of one computer containing multiple processors operating in parallel. The processing units are made to work under the control of a standard control unit. Thus, the system provides one instruction stream and multiple data streams.

Each processing element has a separate memory. The main memory is used to store the program. Data comes from memory to the processing element (PE) as data stream-flow. The instructions also come from the memory and are sent to the control unit. In the SIMD architecture, there is a single control unit and many processing elements. The control unit decodes the instructions and determines how the instruction is to be executed. The single instructions are transmitted from the control unit to the processing element.

The processing elements receive all the instructions and broadcast that instruction from the control unit. An array processor is an asynchronous parallel computer with multiple ALUs that can operate parallelly in a lockstep fashion.

It comprises several identical Processing Elements under the control of a single control unit and several memory modules. Array processors also frequently use parallel computation called pipelining, where an operation is divided into smaller steps, and the steps are performed simultaneously. It can significantly improve performance on specific workloads, mainly in numerical simulation. Figure 3.4.5 illustrates the diagram of the SIMD array processor organisation.

The most well-known SIMD (Single Instruction, Multiple Data) array processor is the ILLIAC IV computer, which was developed at the University of Illinois and manufactured by the Burroughs Corporation. Although no longer operational, the ILLIAC IV was a highly specialized computer designed primarily for numerical problems that could be formulated in vector or matrix formats. However, SIMD processors like the ILLIAC IV are not as effective for other types of computations or conventional data processing programs.

3.4.8 Limitations of Array Processing

All the array processing architecture relies on the fact that the data sets are all acting on a single instruction. However, if these data sets somehow rely on each other, then you cannot apply parallel processing. For example, if data A has to be processed before data

B, then you cannot do A and B simultaneously. This dependency is what makes parallel processing difficult to implement. This is the major limitation of array processing.

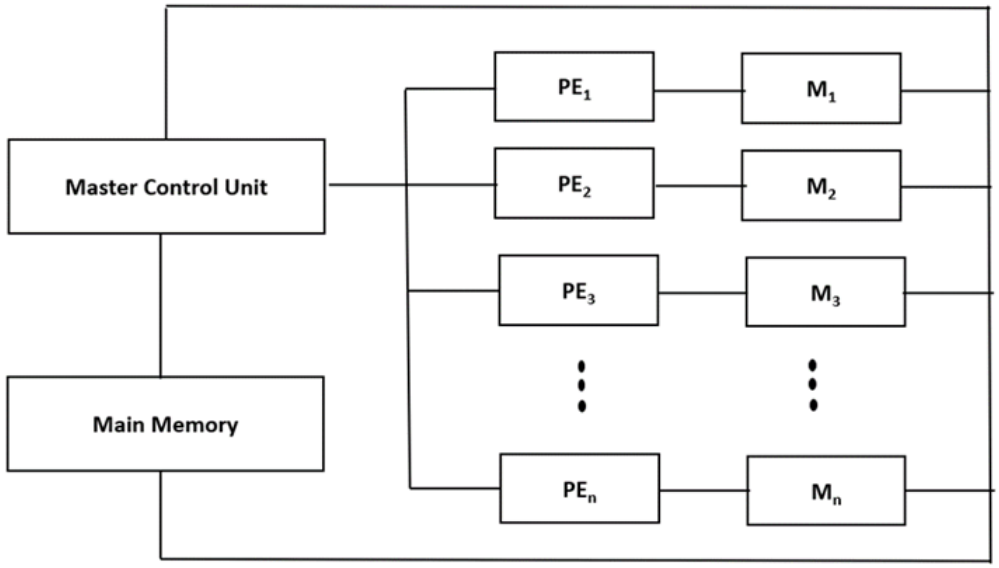


Fig 3.4.5 SIMD Array Processor Organization

Recap

- Scalar Processors are the most fundamental types of processors; they process one item at a time.
- Vector Processor aggregates multiple data points, processing each in turn
- The SIMD is also called Vector instructions, and these vector instructions are stored in a vector register.
- The time required to route the vector to the functional unit is called the setup time
- Flushing time is a time duration that a vector instruction from the initial system processing level to the first result is out of the pipeline.
- An array processor may be a processor that performs computations on large arrays of knowledge.
- Array processors are also referred to as multiprocessors or vector processors.
- There are two types of array processors. They are
  1. Attached Array Processors
  2. Single Instruction Multiple Data Organization (SIMD) Array Processors

Objective Type Questions

1. Which operation is performed by vector processing on a large array of integers or floating-point numbers?
2. A vector processor is a CPU design wherein the instruction set includes operations that can perform mathematical operations on multiple data elements simultaneously. Is the given statement true or false?
3. Who is the father of Vector Processing and Supercomputing?
4. To which queue does data read into vector registers?
5. What is a vector processor known as?
6. How many types of array processors are there in parallel processing?
7. An attached array processor is a processor which is attached to which part?
8. What does the SIMD Array Processors element include?
9. Array Processors have their own local memory, hence providing extra memory for systems with low memory. Is the given statement true?

10. What does an array processor perform in multiple execution units in the same clock cycle?

11. Which component has an input-output interface to a common processor and another interface with local memory?

Answers to Objective Type Questions

1. Arithmetic
2. True
3. Seymour Cray
4. Vector pipeline
5. Array processor
6. Two
7. General purpose computer
8. Registers, ALU, Memory Interface and Interconnection Network
9. False
10. Single instruction
11. Attached array processor

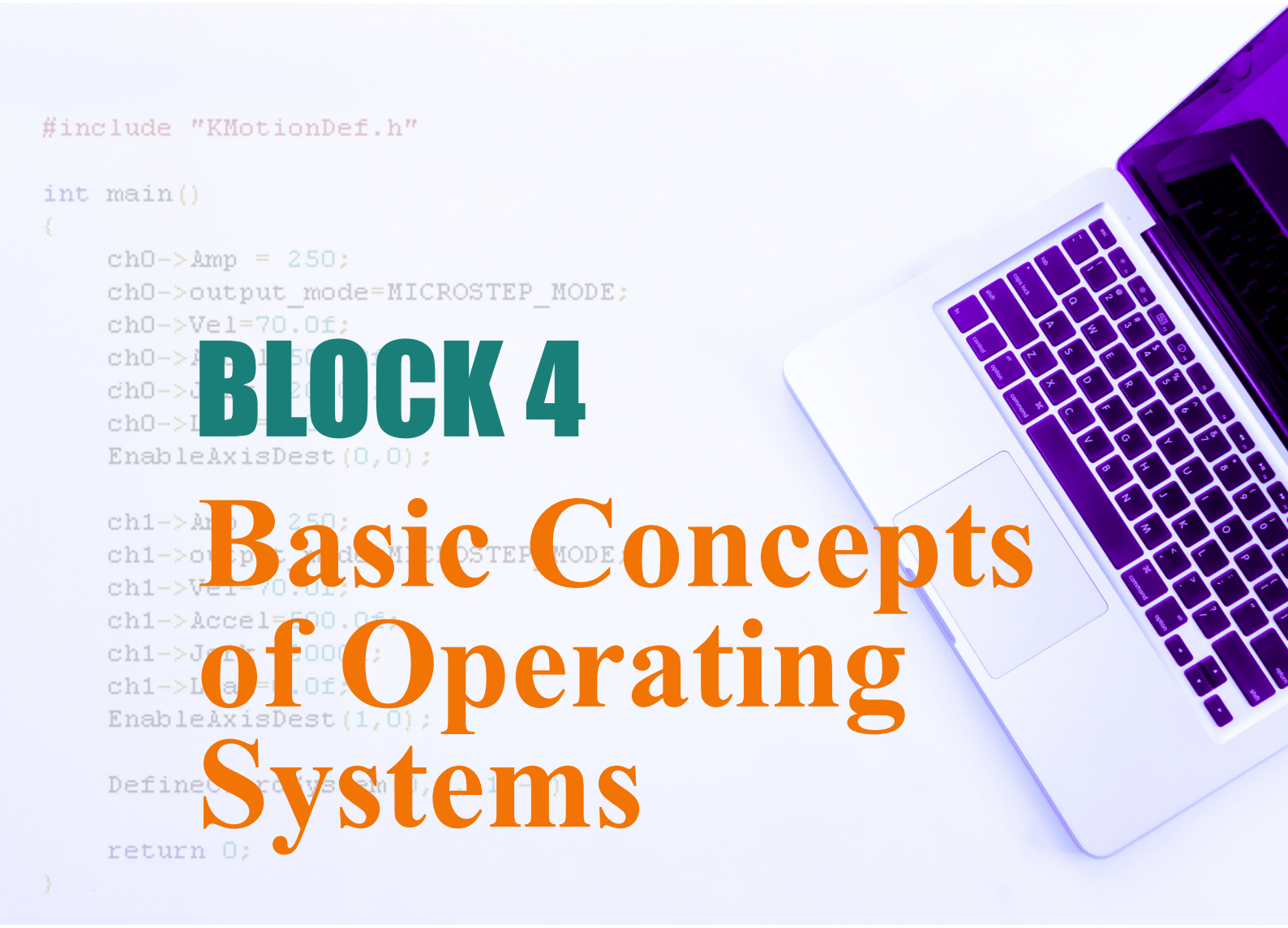
Assignments

1. Draw a computer processor architecture to overcome the dependency in the Array processor system.
2. Why Systolic Architecture? List the advantages of this processor.

Suggested Reading

1. Hwang, Kai, and A. Faye. "Computer architecture and parallel processing." (1984).

2. Mano, M. Morris. Computer system architecture. Prentice-Hall of India, 2003.



BLOCK 4

Basic Concepts of Operating Systems



UNIT 1 Operating System Components and Design

Learning Outcomes

By the completion of the unit, the learner will be able to:

- introduce concepts of process management.
- familiarise the concept of memory management.
- explain to file, disk, and I/O management.
- understand the concept of booting
- identify different structures of the operating system

Prerequisites

System Software

System software maintains the computer resources and also provides a path for application software to run. System software is written in low-level languages. A computer cannot run without system software. It does not depend on application software. It starts when we turn on the system and stops when the system is turned off. System Software programming is complex compared to application software. System software examples include **operating systems** (Microsoft Windows, macOS, and Linux), Compiler, and Assembler.

Application Software

It is a specific-purpose software. It enables users to perform specific tasks. Application software is written in high-level languages, such as Java and C++. A computer can run without application software. Application software cannot run without system software. It starts according to the user's request. Its programming is simpler compared to system software. Application software examples include Google Chrome, VLC media player, Microsoft Word, and Photoshop.

In this unit, we will discuss about the most important system software- the operating system

Keywords

Process Management, Memory Management, Storage Management, File System Management

Discussion

4.1.1 Operating System

Do you know what an Operating System is? The operating system is a system program whose job is to control all the resources of the computer and also to provide a platform over which application programs can be executed.

> An operating system is a program that controls the execution of application programs and acts as an interface between applications and computer hardware.

The Operating System is also known as the manager of the computer. Before we discuss the operating system in detail, let us understand what happens when you switch on a computer.

When we switch on a computer, the initial task of the computer is to start up a special program called an operating system (OS). An example of the operating system is Windows. One important job of the operating system is to help other computer programs work by handling the chaotic details of controlling the computer's hardware.

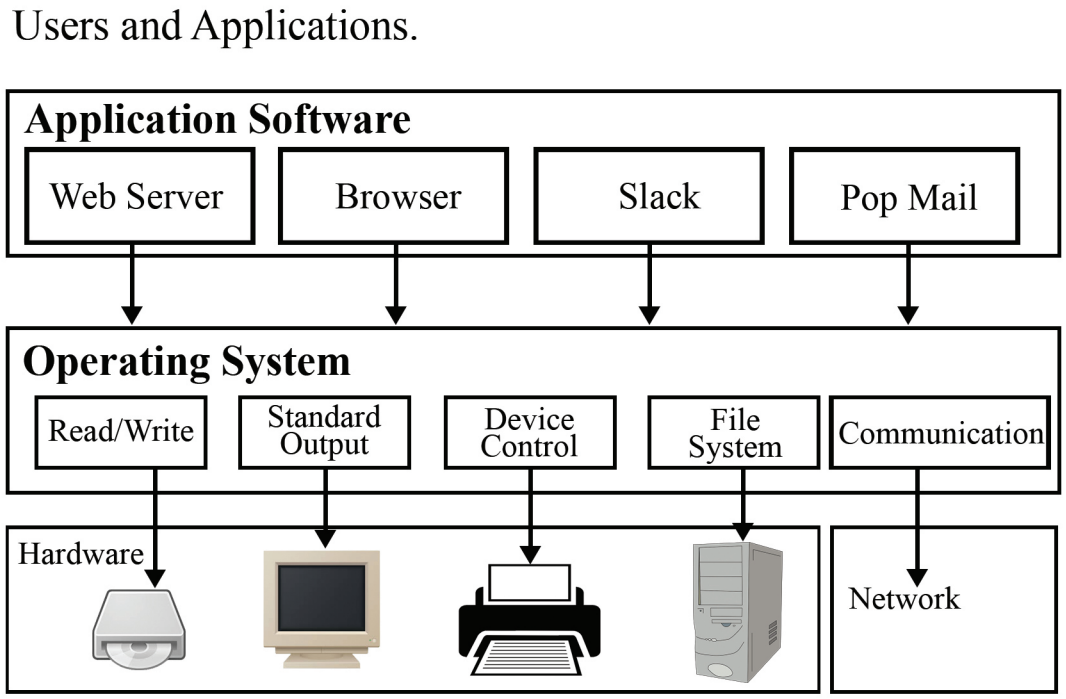


Fig. 4.1.1 Operating System Structure

How is the computer system getting ready itself? How is the OS getting loaded to the system as when we power it on? The computer system gets ready by itself through the process known as booting. Let us understand the term booting in order to answer the above question.

An operating system is a construct that allows the user application programs to interact with the system hardware. The Operating System does not provide any function by itself, but it provides an atmosphere in which different applications and programs can do useful work. A sample structure of an operating system with connections is shown in figure 4.1.1.

4.1.1.1 Design Requirements in Operating System

The design requirements are quite hard to specify in an operating system. They are basically divided into two parts: User design requirements and System design requirements. Details about these are given as follows:

- User Design Requirements

The operating system should be convenient, easy to use, reliable, safe and fast according to the users.

- System Design Requirements

The operating system should be easy to design, implement and maintain. These are specifications required by those who create, maintain and operate the operating system. However, there is no specific method to achieve these goals.

4.1.1.2 Design Goals in Operating Systems

There are many design goals that modern operating systems have to fulfil to be considered successful.

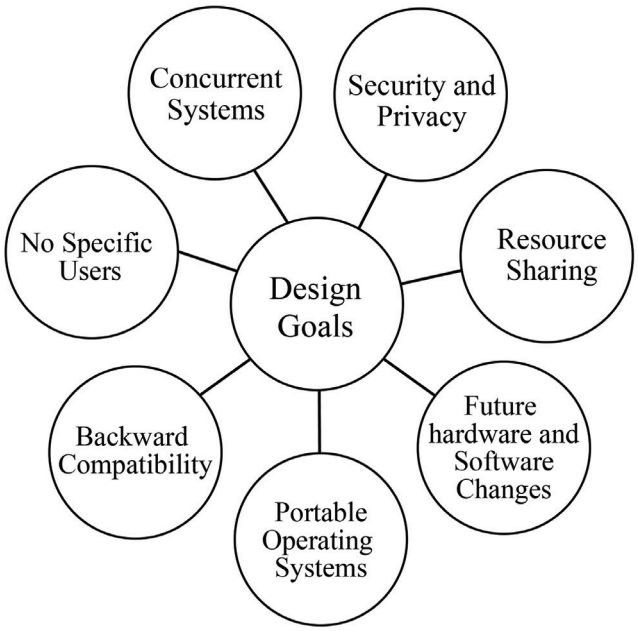


Fig. 4.1.2 Design Goals in Operating Systems

Concurrent Systems

Modern operating systems should be able to handle multiple users as well as multiple devices at the same time. This is necessary for modern multi-core architectures. Because of these specifications, the operating system design can be quite complex and difficult to create.

Security and Privacy

Operating systems should be able to provide security and privacy for a system. This is very important as many malicious users may want to hack into the computer system and steal user programs.

Resource Sharing

The operating system should make sure that resources are shared correctly between multiple user processes. This can get quite complex when multiple users share the same device as well.

Future Hardware and Software Changes

A key design consideration is ensuring that the operating system remains adaptable to future hardware and software changes, preventing obsolescence. This is necessary as changing the operating system again and again is quite a costly process.

Portable Operating Systems

The operating systems should be portable, i.e. they should work with different hardware and machines. There may be some specialized operating systems that only work on one kind of machine. However, most of them are portable.

Backward Compatibility

The new operating systems created should be compatible with the previous models, i.e., they should have backward compatibility.

No Specific Type of Users

Operating systems should be developed for the general user base so that many users can use them. Even specially developed operating systems that target a specific user base contain some generality.

4.1.2 Operating System Components

The components of an operating system play a key role in making a variety of computer system parts work together. The components of an operating system include the following:

1. Process Management
2. Memory Management
3. File System Management

4. Mass-Storage Management
5. I/O System Management
6. Protection and Security

Operating system components help to ensure accurate computing by detecting CPU and memory hardware errors.

4.1.2.1 Process Management

A program remains inactive until its instructions are executed by a CPU. When a program is actively running, it is referred to as a process. For example, a time-shared user program such as a compiler or a word-processing application operated by an individual on a PC qualifies as a process. Additionally, system tasks like sending output to a printer may also be considered processes or components thereof. Initially, a process can be understood as a job or a time-shared program; however, the concept encompasses a broader range of definitions. It is feasible to implement system calls that enable processes to create subprocesses for concurrent execution. Each process requires various resources such as CPU time, memory, files, and I/O devices to accomplish its tasks. These resources may be allocated at the time of process creation or assigned during its execution. Besides the physical and logical resources acquired at creation, processes might also receive initialization data (input). For instance, a process designed to display the status of a file on a terminal will provide the file name as input and utilize the appropriate instructions and system calls to retrieve and present the information on the terminal. Upon completion of the process, the operating system will reclaim any reusable resources.

A program is a static entity, similar to the contents of a file stored on a disk, whereas a process is a dynamic entity. In a single-threaded process, there is a single program counter that indicates the next instruction to be executed. The execution of this process must occur sequentially, with the CPU executing one instruction at a time until the process is completed. Consequently, at any given moment, only one instruction is executed for the process. Therefore, while two processes may be associated with the same program, they represent distinct execution sequences. In contrast, a multithreaded process features multiple program counters, each directing the execution of instructions for a specific thread.

A process serves as the fundamental unit of work within a system. A system comprises a set of processes, which include operating-system processes (responsible for executing system code) and user processes (executing user code). These processes have the potential to execute concurrently by sharing a single CPU through multiplexing. The operating system is responsible for the following activities in connection with process management:

- Scheduling processes and threads on the CPUs
- Creating and deleting both user and system processes
- Suspending and resuming processes

- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

4.1.2.2 Memory Management

In modern computer systems, main memory serves as a critical component for operational functionality. This memory consists of a large, addressable byte array, typically ranging in size from hundreds of thousands to billions of bytes. It acts as a central repository for data that can be rapidly accessed by both the central processing unit (CPU) and input/output (I/O) devices. Within the von Neumann architecture, the CPU fetches instructions during the instruction-fetch cycle, and both reads and writes data during the data-fetch cycle, all from main memory. Notably, main memory is the sole large-scale storage device directly addressable and accessible by the CPU. For instance, processing data stored on a disk requires the data to be transferred to main memory first via CPU-generated I/O calls. Similarly, instructions must reside in the main memory for the CPU to execute them.

To execute a program, absolute addresses must be allocated and loaded into memory. During execution, the program accesses instructions and data from memory by generating these absolute addresses. Once the program finishes running, its allocated memory is released, making way for the loading and execution of subsequent programs. To enhance both CPU utilization and the speed of the computer's response to users, general-purpose computers must maintain multiple programs in memory simultaneously, which necessitates effective memory management. Various schemes for memory management are employed, each offering different strategies, and the effectiveness of any given scheme is contingent on the specific context. When choosing a memory management approach for a particular system, several factors, especially the system's hardware design, must be carefully considered. The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and who is using them.
- Deciding which processes (or parts of processes) and data to move into and out of memory.
- Allocating and deallocating memory space as needed.

4.1.2.3 File-System Management

File management is one of the most prominent components of an operating system. Computers store information on various types of physical media, with magnetic disks, optical disks, and magnetic tapes being the most common. Each type of media has distinct characteristics and physical organization, and each is managed by a specific device, such as a disk drive or tape drive, each possessing unique attributes. These attributes include access speed, capacity, data transfer rate, and access method (sequential or random). A file is a collection of related information defined by its creator and can represent programs (in both source and object forms) as well as data. Data files

may be numeric, alphabetic, alphanumeric, or binary and can be either free-form (e.g., text files) or rigidly formatted (e.g., fixed fields).

The operating system implements the abstract concept of a file by managing mass-storage media, such as tapes and disks, along with the devices that control them. Files are typically organized into directories to enhance usability. When multiple users have access to files, it is often necessary to regulate user access and permissions, specifying how each user may interact with a file, such as reading, writing, or appending. The operating system is responsible for the following activities in connection with file management:

- Creation and deletion of files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

4.1.2.4 Mass-Storage Management

Main memory is insufficient in size to accommodate all data and programs, and its contents are lost when power is disrupted. Therefore, computer systems must provide secondary storage to support main memory. In most modern computer systems, disks serve as the primary online storage medium for both programs and data. Programs, including compilers, assemblers, word processors, editors, and formatters, are typically stored on a disk until they are loaded into memory. These programs use the disk as both the source and destination for their processing tasks. Consequently, the proper management of disk storage is crucial to the overall functionality of a computer system. The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

The overall speed of a computer's operation can significantly depend on the performance of the disk subsystem and the efficiency of the algorithms managing it. There are various storage options available that are slower and less expensive than secondary storage, often offering higher capacity. Examples include backups of disk data, storage for infrequently accessed data, and long-term archival storage. Tertiary storage devices, such as magnetic tape drives with their tapes and CD and DVD drives with their platters, serve these purposes. Tertiary storage media, such as tapes and optical platters, can come in different formats, including WORM (write-once, read-many-times) and RW (read-write). While tertiary storage is not critical for system performance, it still requires management. Some operating systems handle this task, while others delegate

the management of tertiary storage to application programs. The functions provided by operating system include mounting and unmounting media in devices, allocating and releasing devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Cache Memory

Caching is a fundamental concept in computer systems. Information is typically stored in a primary storage system, such as main memory. For quicker access, this information is temporarily copied into a faster storage system known as a cache. When a specific piece of information is needed, the system first checks if it is available in the cache. If the information is found in the cache, it is used directly from there. If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon. In addition, internal programmable registers, such as index registers, provide a high-speed cache for main memory.

The programmer or compiler uses algorithms to decide which information should be kept in registers and which should be kept in main memory. Other types of caches are completely implemented in hardware. For example, most systems have an instruction cache that holds the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles to fetch an instruction from the main memory. Similarly, most systems have one or more high-speed data caches in the memory hierarchy for faster data access. Since caches have limited sizes, managing them effectively is a significant design challenge. Carefully choosing the cache size and replacement policy can greatly improve performance. Various replacement algorithms are used for software-controlled caches to optimize their efficiency.

The file-system data, which resides permanently on secondary storage, may appear on several levels in the storage hierarchy. At the highest level, the operating system may maintain a cache of file-system data in the main memory. In addition, solid-state disks may be used for high-speed storage that is accessed through the file-system interface.

Table 4.1.1 Performances of various levels of storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	<1 O TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80- 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

The bulk of secondary storage is on magnetic disks. The magnetic-disk storage, in turn, is often backed up onto magnetic tapes or removable disks to protect against data loss in case of a hard disk failure. Some systems automatically archive old file data from secondary storage to tertiary storage.

4.1.2.5 I/O Systems

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O subsystem. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

4.1.2.6 Protection and Security

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated by only those processes that have gained proper authorization from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control.

Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected. Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot be defended against use (or misuse) by an unauthorized or incompetent user. A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information is stolen. The user's data could be copied or deleted, even though file and memory protection are working. It is the job of security to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is considered as an operating-system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents, operating-system security

features represent a fast-growing area of research and implementation.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated user identifiers (user IDs). In Windows parlance, this is a security ID (SID). These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list. In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and group identifiers. A user can be in one or more groups, depending on operating-system design decisions.

The user's group IDs are also included in every associated process and thread. In the course of normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to escalate privileges to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the set-uid attribute on a program causes that program to run with the user ID of the owner of the file rather than the current user's ID. The process runs with this effective UID until it turns off the extra privileges or terminates.

4.1.3 Booting

We often hear the word boot the computer. What does booting actually mean? Let us learn what the term booting means.

Booting is basically the process of starting the computer. When the CPU is first switched ON, there would be nothing inside the memory. So, to start the computer, we load the operating system into the main memory. The loading of the OS to the main memory to start the computer is known as booting.

The Boot process begins with each PC containing a motherboard. On this motherboard is a program called the system BIOS (Basic Input/Output System). The BIOS contains fundamental low-level I/O software that manages basic input and output operations. Mainly, it includes routines that handle tasks such as processing data from the keyboard, displaying information on the screen, and managing data read and write operations to storage disks. Currently, the BIOS is stored in flash RAM, which is nonvolatile but can be updated by the operating system when bugs are found in the BIOS.

When the computer is booted, the BIOS initializes and begins by checking the amount of installed RAM and verifying that the keyboard and other fundamental devices are correctly installed and operational. It starts by scanning the PCIe and PCI buses to detect all the devices attached to them. If the devices present are different from when the system was last booted, the new devices are configured.

The BIOS then identifies the boot device by sequentially attempting to boot from a list of devices stored in the CMOS memory. Users can modify this list by accessing the BIOS configuration program shortly after booting. Typically, an attempt is made to boot from a CD-ROM (or sometimes USB) drive if one is present. If that fails, the system boots from the hard disk. The first sector from the boot device is read into memory and executed. This sector contains a program that normally examines the partition table at the end of the boot sector to determine which partition is active. Then, a secondary boot loader is read from that partition. This loader reads in the operating system from the active partition and starts it.

The operating system then queries the BIOS to get the configuration information. For each device, it checks to see if it has the device driver. If not, it asks the user to insert a CD-ROM containing the driver (supplied by the device's manufacturer) or to download it from the Internet. Once it has all the device drivers, the operating system loads them into the kernel. Then, it initializes its tables, creates whatever background processes are needed, and starts up a login program or GUI.

4.1.4 More about operating system

The early computers did not have an OS! In those days a programmer was also a hardware expert who knew how to write programs in binary language. (S)he was also responsible for "mounting" the input, executing the program and getting the output. Only a single program was being executed. The programmers interacted with the computer hardware directly. There was no operating system.

Thus, everyone was an expert in hardware and software. Later, a separate OS was released. As a result, common people do not need to know about hardware interactions and functioning. As a result, the task of common people became much easier. This is the advantage of having an operating system. Hence, a special group of people with knowledge of system programming were required to create system software.

> Systems programmers are technical experts on some or all of the computer's system software (operating systems, networks, DBMSs, etc.). They are responsible for the efficient performance of the computer systems.

However, we do not find common people as system programmers. As a result, system programmers are in high demand. System programmers develop software like an operating system. Now, let us learn the objectives of the OS. When we are asked what the main objective of an OS is, our answer should be convenience. Another objective of the OS is to act as an interface for user security as well as data security.

4.1.5 Different types of Operating systems

Have you ever thought about the different types of OS?

Different types of OS are

1. Serial Processing

2. Simple Batch System
3. Multi programmed Batch System
4. Time-Sharing System

Let us learn the different types of operating systems so that we can understand the requirements and significance of OS.

4.1.5.1 Serial Processing

In order to understand serial processing, let us take a real-world example. Suppose A's mother is a housewife. She is planning to prepare rice and curry. Suppose that there is only one burner on the stove. Suppose she is planning to cook the rice first, and she keeps the rice pot with rice and water on the stove. In order for her to cook the curry, she has to wait till the rice is cooked. i.e., here, the cooking takes place in a series.

Have you ever wondered how the earliest computers worked?

In the earliest computers, the programmer interacted directly with the hardware. Since the users are accessing the computer in series, one after another, this mode of operation is also termed serial processing.

The series of cooking that is practiced by A's mother is an illustration of what happened in serial processing. The figure 4.1.3 illustrates a simple serial processing.

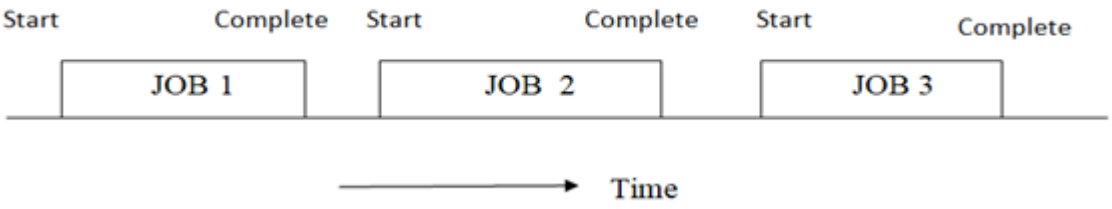


Fig: 4.1.3 Serial Processing

4.1.5.2 Simple Batch System

In order to understand a simple batch system, let us take the earlier real-world example with a slight modification. Suppose there is a party at A's house, and she decides to make the vegetable samosa, vegetable cutlet, spring roll, and smashed potato. So, A's mother decided to make more snacks in an organised fashion. So, she made a list of items that she was planning to prepare and then sorted out their common ingredients. Suppose the common ingredients included beans, carrots, potato, cauliflower, and beetroot. So, she chopped enough beans, carrot, potato, cauliflower, and beetroot for all the snacks together and then cooked the beans, carrot, potato, cauliflower, and beetroot separately. Then she mixed enough beans, carrots, potatoes, cauliflower, and beetroot to make the vegetable samosa, vegetable cutlet, spring roll, and smashed potato.

Now, let us discuss a simple batch system and then discuss the above example to make the concept clear.

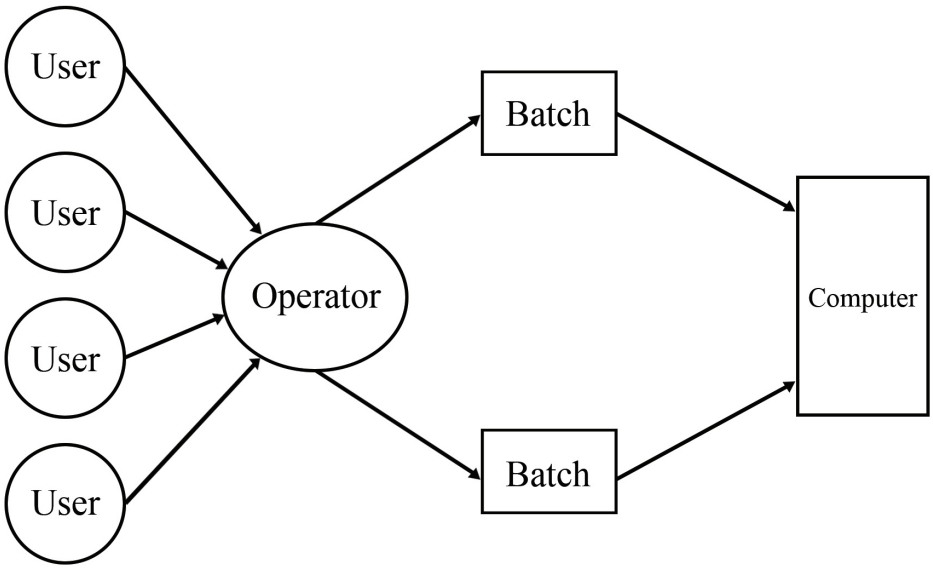


Fig 4.1.4 Simple batch system

Serial processing had many disadvantages. Due to this, a simple batch system was introduced. Following figure 4.1.5 illustrates a simple batch system.

The above figure 4.1.4 illustrates a simple batch system. As you may see, users submit the jobs to the operator. The operator then groups the similar jobs, and they are treated as a batch. Then, they are stored on the punched card. Do you know what punched cards are?

Punched cards are stiff paper where the digital data is stored. The digital data in the punched card are represented using some specific sequence of holes.

The punched cards are later submitted to the system for processing.

Here, similar jobs are grouped and treated as a batch by an operator. Then, they are stored on the punched card, which will be submitted to the system for processing. The system will then perform all the required operations in a sequence.

So, what happens after the jobs are grouped into batches and are in punched cards? Once the jobs are grouped into batches and are in punched cards, the computer operator places the cards on an input device.

In the next stage, a special program called monitor manages the execution of each program in the batch. Now, you might be having a question. What is a monitor?

The monitor is system software that is responsible for carrying out the instructions in the batch jobs. When the monitor starts a job, it hands over control of the entire computer to the job, which then controls the computer until it is finished. The monitor is the first operating system.

Coming back to our real-world example, A's mother is the operator in this case. The cut vegetables are the grouping of jobs, and the punch card is the separate cooked vegetables. Now, A's mother's job is only to make them into batches for vegetable

samosa, vegetable cutlet, spring roll, and smashed potato. Then, once the patties, rolls, etc. are made, she can cook them in batches.

There were many disadvantages to batch processing. The main problem was that the computer operators had to have a strong knowledge of batch systems. Also, batch systems were hard to debug.

4.1.5.3 Multi programmed Batch System

To resolve the problems with a simple batch system, multiprogrammed batch systems were introduced.

Consider a real-world example to understand multiprogrammed batch systems. Suppose A's sister has also started going to school. Now, A's mother has to prepare food and get A's sister ready. So, what she does is start preparing food and, in between, goes to check on her daughter. She multitasks by switching between cooking and checking on her daughter. The same is true for the concept of a multiprogrammed batch system.

The input/output (I/O) devices, such as the keyboard, mouse, and printer, are slow compared to the processor. Hence, when an input /output activity is going on (such as entering data through a keyboard or printing pages), the processor will be idle and waiting for the completion of the IO activity. So, a solution is to switch the processor between jobs which are not likely waiting for I/O. This approach is known as multiprogramming or multitasking. It is the central theme of the modern operating system. Multiprogramming allows the processor to handle multiple batch jobs and multiple interactive jobs at a time.

Let us take a simple example to understand this scenario. When the printer is printing, we can use a Word document to type. So, the processor is able to multitask by switching between these jobs.

4.1.5.4 Time-Sharing System

An example of a time-sharing system is the Airline ticket system. Here, users or, more specifically, the customers and the airline ticket booking agents work from multiple

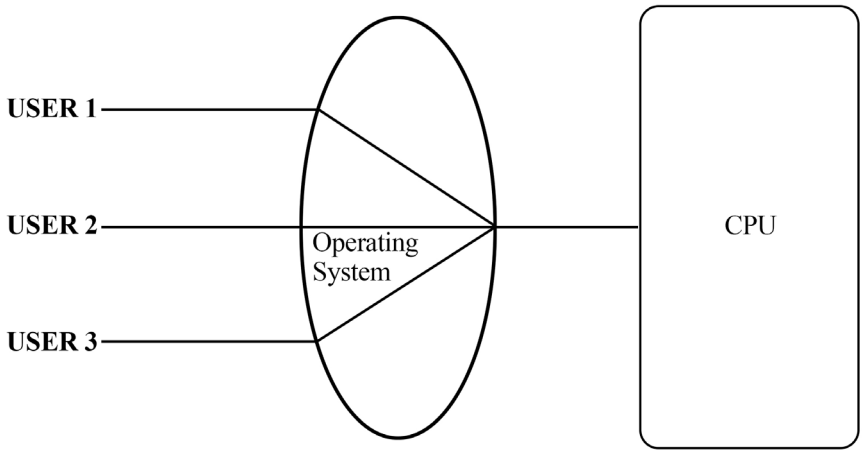


Fig 4.1.5 Time-sharing system

terminals and time is given for each terminal. Users have the sole impression that they are the only people using the system at that time.

Today, we have an interactive computing facility met by the use of a dedicated personal computer or workstation. In the 1960s, there was no interactive computing facility as we have today. Instead, time-sharing was developed. The concept of the time-sharing system is shown in figure 4.1.5.

As shown in Figure 4.1.5, user 1, user 2, and user 3 are accessing the operating system. Here, the processor time is shared among multiple users. This is made possible by dividing the CPU time among the users on a scheduled basis. The OS allocates a set of times to each user. When this time expires, it passes control to the next user on the system. The time allowed is extremely small, and the users are given the impression that they each have their own CPU and are the sole owners of the CPU. This short period during which a user gets the attention of the CPU is known as a time slice or a quantum. Disadvantages of time-sharing systems include reliability problems, the security and integrity of user programs, data as well as data communication problems.

4.1.6 Operating System Structure

Have you ever wondered how the inside of an operating system or, rather, the operating system structure is designed? In this section, we will discuss the operating system structure.

Let us take an example to understand the concepts better. Suppose you are planning to prepare a cake. There is the possibility of making a simple round cake, or you can prepare a layered cake with smaller layers above the bottom layer. But a simple round cake has a disadvantage. When the size of the cake becomes bigger, it becomes more difficult to manage.

With a layered cake, each layer has to be baked separately and then stacked one above the other. Now, let us discuss the operating system structure. OS is a very large program. Hence, it must be designed carefully for proper functioning and easy modification. The common approach is to divide the task into small components or modules rather than having a single huge system (similar to our layered cake example). However, each of the modules should have carefully defined inputs, outputs, and functions so that the module as a whole is well-defined.

Various ways to structure OS are

1. Simple structure – MS-DOS
2. Layered
3. Microkernel –Mach

4.1.6.1 Simple Structure

In the case of a simple OS, a well-defined structure is absent. In most situations, the simple operating system started as a small, simple, and limited system and then grew beyond its original scope. An example of a simple structured OS is MS-DOS. MS-DOS

was designed and implemented by a few people who had no idea that it would become very popular. Since it was designed in such a manner that most of the functionalities are provided in the least space, the modules were not divided carefully. Thus, Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated. As a result, the possibility of system crashes or getting stuck when user programs fail happens.

Can you think of another example of a limited OS structure? The UNIX operating system is another example of a limited structure. It consists of two separable parts:

- Kernel
- System programs

Kernel – The kernel is the important program of the OS,. i.e. the heart of the operating system is the kernel. It manages the system resources. We shall learn more about the kernel in a short while.

System programs - System programs provide a convenient environment for program development and execution.

4.1.6.2 Layered Approach

The layered approach operating system can be compared to our layered cake example. In the layered approach, the operating system is divided into a number of levels, often called layers. Figure 4.1.6 shows a layered operating system. As you can see in Figure 4.1.6, each layer is built on top of the lower layers. The bottom layer (layer 0) is the hardware, and the highest layer (layer N) is the user interface.

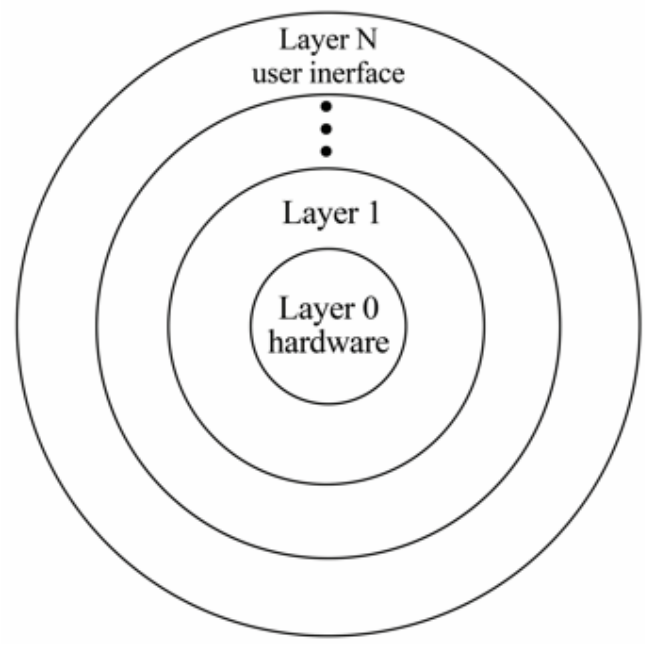


Fig. 4.1.6 A layered operating system

4.1.6.3 Microkernel

In order to understand Microkernel in a good way, we need to understand what a kernel

is and what the types of kernel are.

As we mentioned earlier, the heart of the operating system is the kernel. The process of loading the OS to the main memory to start the computer is known as booting. During booting, it is actually the first program of the OS (kernel) that is loaded into the main memory. The kernel remains in the main memory till the computer is shut down. The main function of the kernel is to translate the commands entered by the user in a way that the computer understands what the user has requested. The kernel acts as an interface between the computer's hardware and software. The kernel is responsible for the most important functions of the OS, such as inter-process communication, memory management, and CPU scheduling.

Following are the tasks managed by the kernel.

- Memory management
- Process management
- Task management
- Disk management.

In the case of memory management, the kernel allocates and checks for memory space in order to execute the application program properly. Kernel also takes care of the creation and destruction of memory, which helps in the execution of the software.

We shall be discussing process management in detail in another unit.

A kernel is classified into the following two types:

- Monolithic Kernel
- Microkernel

In the case of a monolithic kernel operating system, the entire operating system is working in the kernel space. However, the disadvantage here is that, for example, a fault in a device driver could bring down the whole system.

Microkernel operating systems do not provide any operating system services; they only provide the mechanisms that are required for the implementation of such services as low-level address space management, thread management, and interprocess communication. We shall discuss interprocess communication in detail in the next section.

In the case of a microkernel operating system, user services and kernel services are kept in separate address spaces. The user programs invoke the microkernel through system calls. The advantage here is that, for example, a fault in a device driver will not bring down the whole system.

Mach is an example of a Microkernel structure. Here, communication between the user modules takes place using message passing. The main advantages of microkernel structure is many.

Few are:

It is easier to implement a microkernel.

- Microkernel is more reliable as less code is running in the kernel mode.
- It is more secure, too.

Recap

- Operating systems act as an interface between applications and the computer hardware.
- The components of an operating system include:
  - Process Management
  - Memory Management
  - File System Management
  - Mass-Storage Management
  - I/O System Management
  - Protection and Security
- An actively running program is known as a process.
- Functions of operating system in process management include:
  - Scheduling processes and threads on the CPUs
  - Creating and deleting both user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication
- Functions of the operating system in memory management include:
  - Keeping track of which parts of memory are currently being used and who is using them.
  - Deciding which processes (or parts of processes) and data to move into and out of memory.
  - Allocating and deallocating memory space as needed.
- Functions of the operating system in memory management include:
  - Creation and deletion of files

- Creating and deleting directories to organise files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

- The operating system is responsible for the following activities in connection with disk management:
  - Free-space management
  - Storage allocation
  - Disk scheduling
- The process of loading the OS to the main memory to start the computer is known as booting.
- Different types of operating systems are:
  - Serial Processing
  - Simple Batch System
  - Multi programmed Batch System
  - Time-Sharing System
- A serial processing operating system is a type of operating system that handles tasks sequentially, one after another.
- In the batch operating system, based on similar functions, the jobs are grouped to batches. These batches of jobs are executed simultaneously
- In Multiprogramming it allows the processor to handle multiple batch jobs and multiple interactive jobs at a time.
- The Time-Sharing operating system provides computer resources to numerous programs simultaneously in a time-dependent manner.
- Various ways to structure OS are:
  - Simple structure – MS-DOS
  - Layered
  - Microkernel –Mach

Objective Type Questions

1. What is the name of a program in execution?
2. What are the two types of OS design requirements?
3. Who is responsible for ensuring resources are shared correctly between multiple user processes?
4. How many program counters does a single-threaded process have?
5. Which is the collection of related information defined by its creator.
6. Allocation of free space to data and programs comes under which component of the OS?
7. What type of processing does a simple operating system use?
8. What is the key feature of a time-sharing operating system?
9. What is the process of starting a computer called?
10. What is the first program that runs when a computer starts?
11. Where is the BIOS typically stored?

Answers to Objective Type Questions

1. Process
2. User and System Design requirements
3. Operating System
4. One
5. File
6. Memory Management
7. Sequential
8. Multiprogramming
9. Booting
10. BIOS
11. ROM

Assignments

1. Identify the OS in your computer.
2. Identify five application programs in your computer.
3. Explain the components of the operating system in detail.
4. Explain the booting process.
5. Explain multiprogramming and batch operating system in detail.
6. Explain the time-sharing operating system.
7. Explain the layered structure of the operating system.
8. Explain the simple structure of the operating system.
9. Explain the microkernel operating system.

References

1. Silberschatz, Abraham, and Peter Baer Galvin. "Operating System Concepts." (2013).
2. Andrew, S. Tanenbaum. "Modern operating systems Pearson." (2014).
3. S TANENBAUM, A. N. D. R. E. W., and HERBERT BOS. "Operating Systems." (2015).

Suggested Reading

1. GALVIN, PETER BAER, and GREG GAGNE. "ABRAHAM SILBERSCHATZ." (2005).
2. Crowley, Charles. Operating systems: a design-oriented approach. McGraw-Hill Professional, 1996.
3. Harris, J. Archer Archer, and John Cordani. Schaum's outline of operating systems. McGraw-Hill, Inc., 2001.



Operating System Services

Learning Outcomes

By the completion of the unit, the learner will be able to:

- familiarise themselves with the most common services of the operating system
- understand system calls in the operating system
- identify system calls used in process control
- explain system calls used in device management

Prerequisites

A computer is an electronic device that manipulates information or data. There are different types of computers. Examples include Personal Computer (PC), Laptops, Mainframe Computers, Supercomputers, etc.

All computers have two things in common: hardware and software.

Do you know what hardware and software mean?

Hardware is any part of the computer that we can touch. Example: Keyboard, Mouse, Monitor, System Unit, etc. It also includes the computer's internal parts.

Software is any set of instructions that tells the hardware what to do and how to do it. Examples of software include Operating System (OS), web browsers, games, and word processors.

Software is classified as system software and application software.

The software that deals with the user inputs and helps the user to complete the task is known as Application software. It is also called an end-user program or only an app.

Examples of Application software include Word, Excel, PowerPoint, sending email, and Graphics.

System software is a type of computer program that is designed to run a computer's

hardware and application programs. An operating system is an example of system software.

Can you tell me what makes a computer different from another device, like, say, an electric fan? When the fan is connected well and when the power is on, it will work. But the computer system is not like this. A set of hardware items that different companies manufacture is assembled together. There has to be a binding force that can manage and control these resources properly, and the entire computer system has to do its work. This is where the OS comes into the picture.

An operating system is a system program whose job is to control all the resources of the computer and also to provide a platform over which application programs can be executed. In the earlier days, computers were not very sophisticated. In the earliest computers, the programmer interacted directly with the hardware. Since the users were accessing the computer in series, one after another, the operation was known as serial processing. In the later stage, a simple batch system was introduced where common jobs were sorted together and then processed in batches. Later, a multi-programmed batch system was introduced that worked on multitasking, where the highlight is the switch of the processor between jobs that are not likely waiting for I/O. Multiprogramming improves processor utilization as it organizes many jobs so that the processor always has one to execute.

Later, a time-sharing system was introduced that made it possible to divide CPU time among users on a scheduled basis. A time-shared operating system allows multiple users to share computers simultaneously.

Now that we have given you an idea of how the different types of operating systems have evolved, it is time to introduce you to how the operating system has been designed. This is referred to as operating system structure.

The next thing we need to know is what actually enables the execution of programs in a convenient manner. It is Operating system services that enable the execution of the programs in a convenient manner. These operating-system services are also provided for the convenience of the programmer to make the programming task easier.

Have you ever thought about how a process can request services from the OS? It is with the help of system calls. System calls provide methods for which a computer program requests a service from the kernel of the OS. System calls are the only entry points for the kernel system.

Keywords

Operating system, Booting, Operating system structure, Operating system services, System Calls

Discussion

4.2.1 Operating System Services

Operating systems offer a lot of services both for programmers and programs. Examples include reading a file from a disk and plugging a USB. The service enables the execution of the programs in a convenient manner. These operating-system services are also provided for the convenience of the programmer to make the programming task easier.

Following are a few common services provided by an operating system:

1. User Interface
2. Program execution
3. Resource Allocation
4. I/O operations
5. File System manipulation
6. Communication
7. Error Detection
8. Accounting
9. Protection and security

4.2.1.1 User Interface

User Interface (UI) is the means by which the user and a computer system interact. Almost all the operating systems have a user interface.

There are several types of interfaces commonly used by the operating system. The prominent among them are Command Line Interface (CLI), Graphical User Interface (GUI) and Audio Interface (AUI). *Which among these are you familiar with?

Command Line Interface (CLI) or Character User Interface (CUI) uses text commands and methods for entering them. The MS-DOS operating system is an example of command-line interfaces. When we use CUI, we have to know the exact commands that are to be used for communicating with the machine. However, CUI is not user-friendly.

A command-line interface or command interpreter allows users to directly enter commands that are to be performed by the operating system.

A graphical user interface is a visual way of interacting with a computer using items such as windows, icons, and menus, which are used by most modern operating systems.

In the case of Audio Interface, the users speak and the system is able to recognize a user based on sound recognition. To be more precise, we can say that the system is trained with the user's voice.

Layered View of the Computer System

Have you heard about the Layered view of a computer? Figure 4.2.1 depicts the layered or hierarchical approach taken by the hardware and software in order to provide applications to the user. As you can see from the figure, only the operating system designer has to interact with the computer hardware. At the same time, the programmer has to interact with the utilities and the operating system. The end-user needs to interact only with the application programs.

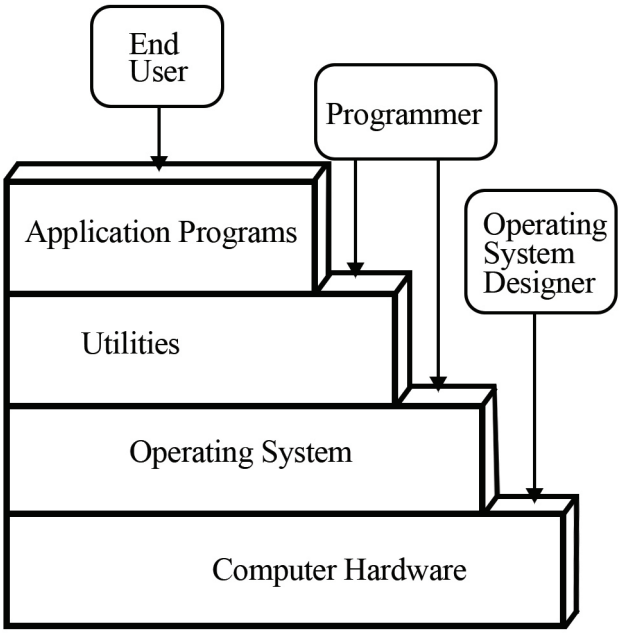


Fig. 4.2.1 Layers and Views of a Computer System

As mentioned earlier, the user is not interested in the details of computer hardware. The end user's views of the computer are as a set of application software. As you may see, the end-users only have to interact with the application programs.

Do you still remember what application programs are? Any guesses? Examples include Microsoft Word, Excel, PowerPoint and Browser. In the case of the programmer, interaction is with the OS and the Utilities. Do you know what utility programs are? What does the term utility mean?

There are certain programs that help your computer perform better. These programs are known as utilities. Examples include antivirus, backup, disk repair, file management, security, and networking programs. Applications such as screensavers, font and icon tools, and desktop enhancements are also utilities. Some utility programs help keep your computer free from unwanted software such as viruses or spyware, while others add functionality that allows you to customise your desktop and user interface.

Programs that help make your computer better are considered utilities

As shown in Figure 4.2.1, only the operating system designer needs to have deep computer hardware knowledge. Thus, the layered view of the computer is highly advantageous to the end-users as it provides applications to the users without the user having to know the chores of the computer's internal working.

4.2.1.2 Program Execution

Last semester, you learned C programming. Have you ever thought about how these programs were actually executed and how this execution was facilitated? This is one of the important services offered by an OS.

When we write a program in C or use any High Level Language (HLL), that file is translated into machine language. Later this machine language file is used by the OS for running the program. The translated file is stored in the secondary memory. When we run this program, the OS will load it to the primary memory (RAM). At that time, the OS will create a process for this program (A detailed explanation of the process is given in the next unit).

To run the process, the OS has to allocate the resources required. In order to execute a program, a number of steps need to be performed. Instructions and data need to be loaded into main memory, I/O devices and files need to be initialised, and other resources must be prepared. The OS relieves the user by handling these chores.

4.2.1.3 Resource Allocation

The hardware and software required by a program to complete its execution are known as resources. Allocation and management of resources is the main job of the operating system. For example, it is the responsibility of the operating system to provide the resources to the programs when multiple programs are running on the computer. The OS will allocate it to one process at a time, take it back once the process is executed completely, and allocate it to other processes as and when needed. Allocation and deallocation will continue till all the needs of resources are over.

4.2.1.4 I/O Operation

Running programs may require Input/Output (I/O), which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a CRT screen). For efficiency and protection, users usually cannot directly control I/O devices. Therefore, the operating system must provide a means to do I/O.

4.2.1.5 File System Manipulation

A file represents a collection of related information. Computers may store files on disks (secondary storage) for long-term storage purposes. Some examples are magnetic tape, magnetic disks, and optical disk drives like CDs and DVDs. They have their own properties like speed, capacity, data transfer rate and data access methods.

Do you know what the difference between a file and a directory is?

A file system is generally organized into directories for easy navigation and usage.

These directories can contain files and other directories. A file and a directory can be visualized as a rack shelf with books. Each row in the rack is a directory, while each book is a file.

Some of the major activities of an operating system with respect to file management are −

- The program requires you to read a file or write a file.
- The Operating System gives permission to the program to operate on a file.
- This permission varies from read-only, read-write, denied and so on.
- The Operating System provides an interface for the user to create/delete files and directories.
- The Operating System provides an interface to create and back up a file system.

4.2.1.6 Communications

There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communication between processes is called Inter-Process Communication (IPC). There are various mechanisms to implement IPCs, such as shared memory and message passing, which are two examples of IPC mechanisms commonly used in OS.

4.2.1.7 Error Detection

The OS continuously monitors the system to find or detect errors and prevent the system from the error. For example, when there is a problem with the keyboard connection, proper messages are given to the programmers as well as the users.

4.2.1.8 Accounting

The Operating System tracks an account of all the functionalities taking place in the computer system at a time. Examples include accounting for the errors that occur while managing the computer's resources, such as the central processing unit, memory, disk drives, and printers. All the details, such as the types of errors that occurred, are recorded by the Operating System. Another example includes monitoring the performance parameters, such as response time. We shall discuss the response time in more detail soon.

4.2.1.9 Protection and Security

Protection and security require that computer resources, such as CPU, software, memory, etc. are protected. This extends to the operating system as well as the data in the system. This can be done by ensuring integrity, confidentiality and availability in the operating system. The system must be protected against unauthorized access, viruses, worms, etc.

4.2.2 System Calls

Let us take a real-world example to understand what system calls are. Suppose your college is organizing a college tour, and your father is not giving you permission. In that case, your final resort is buttering your mother to convince your father to permit you. Similar to the way your mother is acting as an interface between your father and yourself, system calls provide an interface between a process and an operating system to allow user-level processes to request services from the operating system.

Have you ever thought about how a process can request services from the OS? It is with the help of system calls. System calls provide methods for which a computer program requests a service from the kernel of the OS. System calls are the only entry points for the kernel system.

Now, you might be wondering how the system offers the operating system services to the user programs. They are through the Application Programming Interface (API). API is a software intermediary that allows two applications to talk to each other. For example, every time you send a message on WhatsApp or Facebook, you are using an API.

System calls are mainly classified into five categories:

1. Process Control
2. File Management
3. Device Management
4. Information Maintenance
5. Communication

4.2.2.1 Process Control

Process control deals with processes such as process creation and process termination. Examples of process control system calls include fork(), exit(), wait().

- fork() system call makes an exact copy of the original process(parent). The process that is being created is known as a child.
- exit() system call terminates the execution of any process.
- wait() system call terminates the execution of a parent process until one of the child processes completes the execution.
- abort() system call in process control is used to terminate a process abnormally.

4.2.2.2 File Management

File management deals with file manipulation, such as creating a file, reading a file, and writing into a file. Examples of file management system calls include open (), close (), etc.

- open() system call is used to open a file for reading, writing or both.
- close() system call is used to close a file.
- read(): system call is used to read data from a file.
- write(): system call is used to write data to a file. This is used whenever the user makes any kind of modification to the file.

4.2.2.3 Device Management

Device management deals with device manipulation, such as reading from device buffers and writing into device buffers. Examples of Device Management System calls are listed below.

- request device ():

The operating system manages various resources, which can be considered as devices. Some of these devices are physical devices (for example, tapes), while others can be thought of as abstract or virtual devices (for example, files). When multiple users are on the system, you may need to request the device first to ensure exclusive access. So, request device () is used to request a particular device for the user.

- release device ():

After finishing the use of the requested device, then the system calls the release device(), which is used to make it available for others to use.

- read(): System call used to obtain data from a device.
- write(): System call allows a process to send data to a device, such as a disk, terminal, or network device.
- get device attributes(): System call is used to obtain various attributes and status information on devices.
- set device attributes(): System call used to set or modify device attributes, depending on the device and operating system.

4.2.2.4 Information Maintenance

Information Maintenance deals with information and its transfer between the operating system and the user program. Examples of information maintenance system calls are listed below.

- get system data(): This system call allows a program to get important system related information such as hardware configurations, operating system versions, available memory, and processor details.
- set time or date() : The system call allows a program to update the system's current time or date.
- get time or date(): System call used to get the current system time or current system date.

- set system data(): This system call helps to change system level data or configuration settings.

4.2.2.5 Communication

Creating and deleting a communication connection is also dealt with by the communication system call. Some of the important system calls used in communication are listed below:

- create(): This system call refers to the process of creating resources like files, processes that are necessary for communication between different process or systems
- delete(): It is used to remove or release resources that were previously created for interprocess or network communication
- send message(): This system call is used to send a message from one process to another
- receive message(): This system call is used to receive messages that have been sent by another process
- transfer status information(): It is used to retrieve and communicate the current state or status of a process, resource or communication channel between system or processes
- attach or detach remote devices(): This system call is used to manage connection and disconnection of remote devices such as storage devices, network devices or peripherals, within a system

Recap

- Operating system offers a lot of services both for programmers and programs
- Operating-system services are also provided for the convenience of the programmer
- A few common services provided by an operating system are:
  - User Interface
  - Program execution
  - Resource Allocation
  - I/O operations
  - File System manipulation
  - Communication
  - Error Detection

- Accounting
- Protection and security

- User interface:
  - A user interface (UI) is the medium through which a user interacts with a computer system.
  - Example: Command Line Interface (CLI), Graphical User Interface (GUI and Audio Interface (AUI).
  - A command-line interface (CLI), or command interpreter, enables users to input commands directly to be executed by the operating system.
  - A graphical user interface (GUI) is a visual method of interacting with a computer.
- Program execution: The program execution service of an operating system is responsible for loading programs into memory, preparing them for execution, and managing their execution.
- Resource allocation: Allocation and management of hardware and software resources is the main job of the operating system.
- I/O operation: The I/O operation service of an operating system manages input and output operations.
  - It facilitates communication between the computer system and external devices like keyboards, mouses, printers, and storage devices.
- File system manipulation: It provides mechanisms for creating, deleting, reading, writing, and managing files and directories on storage devices.
- Communication: It facilitates the exchange of data between processes, either within the same computer or across different networked computers.
- Error detection: The OS continuously monitors the system to detect and prevent errors, ensuring its stability and reliability.
- Accounting: The operating system keeps a record of all the activities occurring in the computer system at any given time.
- Protection and security: It involves securing computer resources, including the CPU, software, and memory.
- System calls provide an interface between a process and the operating system.
- System calls are mainly classified into five categories:
  - Process Control

- File Management
- Device Management
- Information Maintenance
- Communication

- System calls used in process control are:
  - fork(), exit(), wait(), abort()
- System calls used in file management are:
  - open(), close(), read (), write()
- System calls used in device management are:
  - request device(), release device(), read() , write(), get device attributes(),set device attributes()
- System calls used in information maintenance are:
  - get system data(), set time or date(), get time or date(), set system data()
- System calls used in communication are:
  - create(), delete(), send message(), receive message() , transfer status information(), attach or detach remote devices()

Objective Type Questions

1. What is the means by which the user and a computer system interact?
2. What allows users to directly enter commands that are to be performed by the operating system?
3. What is a visual way of interacting with a computer?
4. What are programs that help make your computer better called?
5. What deals with processes such as process creation and process termination?
6. Which component deals with file manipulation, such as creating a file, reading a file, or writing into a file?

7. Which component deals with device manipulation, such as reading from device buffers and writing into device buffers?

8. Which component deals with information and its transfer between the operating system and the user program?

Answers to Objective Type Questions

1. User Interface
2. Command line interface or command interpreter
3. Graphical User Interface
4. Utilities
5. Process control
6. File management
7. Device management
8. Information maintenance

Assignments

1. Explain different services provided by the operating system.
2. Explain system calls used in process control.
3. Explain the User interface and Error detection services of the operating system.
4. Explain system calls used in file manipulations and device management.
5. What is resource allocation?

References

1. Silberschatz, Abraham, and Peter Baer Galvin. "Operating System Concepts." (2013).

2. Andrew, S. Tanenbaum. "Modern operating systems Pearson." (2014).

3. S TANENBAUM, A. N. D. R. E. W., and HERBERT BOS. "Operating Systems." (2015).

Suggested Reading

1. GALVIN, PETER BAER, and GREG GAGNE. "ABRAHAM SILBERSCHATZ." (2005).

2. Crowley, Charles. Operating systems: a design-oriented approach. McGraw-Hill Professional, 1996.

3. Harris, J. Archer Archer, and John Cordani. Schaum's outline of operating systems. McGraw-Hill, Inc., 2001.



UNIT 3 Process Scheduling

Learning Outcomes

By the completion of the unit, the learner will be able to:

- Introduce the functions of the long-term scheduler, medium-term scheduler, and short-term scheduler
- Make them aware of the different criteria for scheduling
- Familiarize preemptive and non-preemptive scheduling algorithms
- Identify different states of the process

Prerequisites

In Unit II, we discussed that a program is stored in secondary memory and, hence, is a passive entity. A process exists in time and is created when an executable file is open and, hence, is an active entity. We also learnt that the process state represents the current status of the process. It may be new, ready, running or waiting.

Imagine you are baking something, and your mother is cutting vegetables. If she accidently cuts her finger, you would need to switch between baking and attending to her to provide first aid. Here attending of your mother in between the baking process was due to an interruption (burnt finger).

You make a note of where you were in the recipe (process current state is saved), then take a first aid book and follow the instructions. Here, we saw that the processor (You) is switching between processes (first aid, baking) based on priority. (First aid is a higher priority than baking). When the first aid has been provided, you go back to the process of cooking.

Our inferences from this example were that a process is an activity of some kind. A process has a program, input, output and state. Another inference was that a single processor may be shared among multiple processes with the help of some priority schemes. This is where scheduling comes into the picture. We can understand scheduling algorithms that help to determine when to stop work on one process and service a different one.

The OS also needs to switch between jobs based on interrupts and priority. The process of removing the current running process and selecting another process on the basis of a particular strategy by the process manager of the OS is known as process scheduling.

There are different scheduling algorithms. We shall discuss them in this unit.

Keywords

Process Scheduling, Process Scheduling Queues, Scheduler, Scheduling Algorithms, Non Preemptive Algorithms, Preemptive Algorithms, First Come First Serve, Shortest Job First, Shortest Remaining Time First, Round Robin.

Discussion

4.3.1 Introduction to Process

A process is defined as a program in execution, which includes the code of the program, the current activity of the program, and its allocated resources. For example, when you open Microsoft Word in your computer to write a document, then the operating system will create a process for Microsoft Word. This involves loading the Word program code from the hard drive into the computer's memory. While running the Microsoft Word process, you can insert data, edit text, save data, change fonts, and insert pictures in the document. Also, while executing a process, it will go through various states, such as running state, while you actively type text or perform any other activities. The next stage is the waiting state when you pause writing text or suppose you are loading data from a file. It will also be terminated when you close the Microsoft Word application. We will cover the states of a process in a different section.

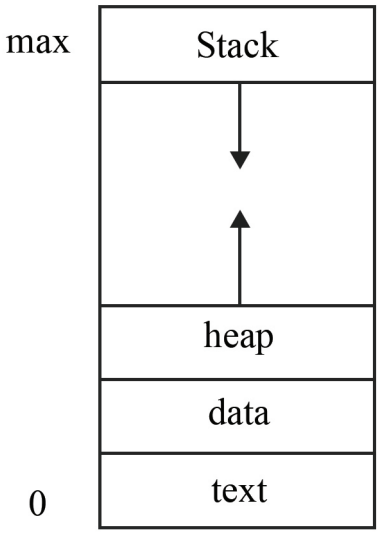


Fig. 4.3.1 Process in memory

A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a heap, which is a memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure 4.3.1

4.3.2 Process State

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- New state: The process is being created.
- Running state: Instructions are being executed.
- Waiting state: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- Ready state: The process is waiting to be assigned to a processor.
- Terminated state: The process has finished execution.

4.3.3 Process Control Block

Each process is represented in the operating system by a process control block (PCB), also known as a task control block. A PCB is shown in Figure 4.3.2. It contains many pieces of information associated with a specific process, including these:

- **Process state:** The state may be new, ready, running, waiting, halted, and so on.
- **Process ID(PID):** It is a unique identifier assigned to each process by the operating system to differentiate it from other processes.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs to allow the process to be continued correctly afterwards.
- **Cpu-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameter
- **Memory-management information**: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

- **Accounting information:** This information includes the amount of CPU and real-time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

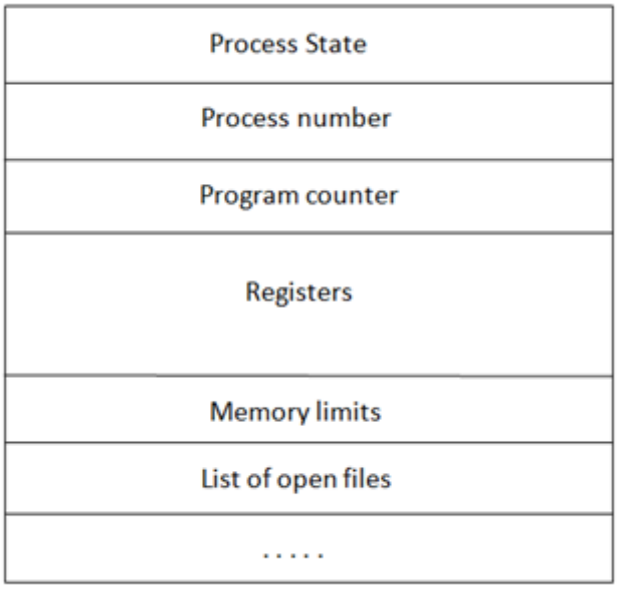


Fig. 4.3.2 Process Control Block (PCB)

When there is more than one process ready to execute with the processor, a selection decision needs to be made to pick a process for execution from among the ready processes; this activity is called Process scheduling.

Let us take a real-world example to understand this situation.

Suppose there is an Audio-Visual Room (AVR) at your University, and at a time, only one department can conduct its seminars in the AVR. Suppose many departments compete to book the AVR. In that case, someone needs to be an AVR scheduler who is responsible for allocating the AVR according to the time requested by one of the department representatives.

The CPU scheduler is like an AVR scheduler: A request is made to the CPU scheduler whenever a process wants to use the CPU. When the CPU is available, it is allocated to the process making the request.

Now, coming back to process scheduling, let us understand where process scheduling is utilized.

Process scheduling is an integral part of multiprogramming OS.

4.3.4 Process Scheduling Queues

Now that we have learnt that process scheduling takes place in multiprogramming OS, let us learn how process scheduling is achieved by the OS. The OS maintains a separate queue for each of the process states. They are:

- Job queue
- Ready queue
- Device queue

Job queue consists of all the processes in the system. When a process enters a system and is in a ready state, they are placed in the ready queue. Those processes that are waiting for a device to be available are placed in the device queue. The device queue is unique for each I/O device. The process is placed in the I/O queue if the process requests an I/O request.

Now, you might have a question: How does the OS differentiate between different queues?

Different queues are identified by taking into account the state of the PCB. All the processes in the same state are classified into the same queue. Figure 4.3.3 shows the queuing diagram representation of process scheduling.

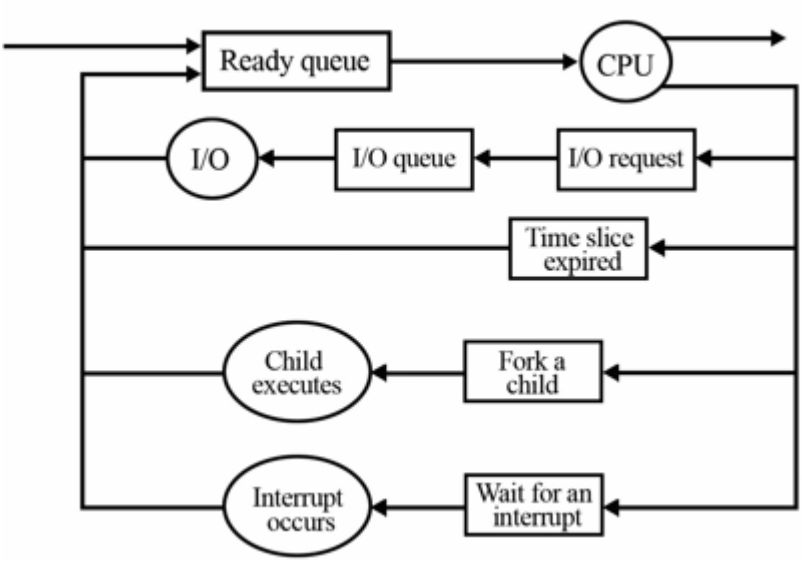


Fig. 4.3.3 Queuing diagram representation of process scheduling

4.3.5 Process Scheduling

Do you know what process scheduling means?

As you can see from Fig 4.3.3, a new process is put in the ready queue initially. The process waits in the ready queue till selected or dispatched for execution with the CPU. Following are the events that could be encountered by a process while executing

- Request a I/O request and be placed in I/O queue
- It would create a new subprocess (child) and wait for it to terminate.
- It could be removed forcibly due to an interrupt and be put back into the ready queue.

As we see from the above diagram, a process migrates between the various scheduling queues throughout its lifetime.

4.3.6 Scheduler

Now that we have learnt how the OS achieves process scheduling, let us learn who is responsible for the scheduling process in the OS. Schedulers in the OS are the process that decides which task and process should be accessed and run at what time by the system resources. Let us again take our AVR example to understand the scenario. In AVR scheduling, a chart is maintained to show the times when the AVR is reserved or free. While the university executives have a priority for certain days, other departments will be prevented from using the AVR.

In an OS, the scheduler is responsible for selecting a process for scheduling/operating with the resource queues serving the resource.

4.3.7 Types of Schedulers

Now, let us discuss the different types of schedulers.

There are three types of schedulers. They are:

- Long term scheduler
- Short term scheduler
- Medium-term scheduler

4.3.7.1 Long term scheduler

A long-term scheduler is also known as a job scheduler. The long-term scheduler runs less frequently. The long-term scheduler decides which process should get into the job queue. A process is selected from the disk and is loaded into the main memory for execution. The main aim of the long-term scheduler is to maintain a good degree of multiprogramming.

4.3.7.2 Short term scheduler

A short-term scheduler is also known as a CPU scheduler. Short-term scheduler runs more frequently. It selects a process from among the processes in the ready queue for allocation of CPU. Short term scheduler is invoked every time the CPU requires a new process for execution.

4.3.7.3 Medium term scheduler

Medium-term scheduler is a process swapping scheduler. Let us take a scenario where

the process is running and has made an I/O request. In that case, the process becomes suspended. Since a suspended process cannot make any further progress in completion, it needs to be removed from the memory in order to make space for another process. The suspended process is moved to the secondary storage. This process is termed as swapping.

Medium term scheduler helps in swapping. The speed of the medium-term scheduler is between the short-term scheduler and long-term scheduler. Medium term scheduler reduces the degree of multiprogramming due to swapping. The process in the secondary storage can be reintroduced into the memory for execution of process activity and their continuation. Before we proceed further, let us get introduced to the term context switch.

4.3.8 Context Switch

A context switch is the task of switching the CPU from one process to another, saving the state of the old process and loading the saved state of the new process. Context Switch Time (CST) is the time involved in this task. CST reduces the efficiency of the working of the computer as no useful work is done at this time. CST varies from machine to machine. The moments in which context-switching calls are made are known as Preemption points.

4.3.9 Scheduling Algorithms

Now, you might be wondering how the scheduler actually does the process of scheduling. The answer is with the help of different scheduling algorithms. By using different scheduling algorithms, the scheduler schedules different processes to be assigned to the CPU. We can compare our AVR booking scheduling schemes to the scheduling algorithms. The scheduling algorithms are classified into two types. They are:

1. Non-Preemptive Algorithms
2. Preemptive Algorithms

The non-preemptive algorithm can be compared to the booking of AVR by the people according to the order in which they have requested the room, and no priorities are assigned.

Non-preemptive Algorithms are those algorithms where once the process enters a running stage, it cannot be blocked until it completes its allocated time. Examples of Non-preemptive Algorithms include

- First Come First Serve (FCFS)
- Shortest Job First (SJF)

Let us take our AVR example again to understand preemptive algorithms. In AVR scheduling, a chart is maintained to show the times when the AVR is reserved or free. When the executives of the university have decided to hold an urgent meeting, the bookings made by the other departments will be interrupted. This scenario can be compared to preemptive algorithms.

Preemptive algorithms are those algorithms based on priority where a scheduler may interrupt a low-priority running process when a high-priority process enters into a ready state.

Examples of preemptive algorithms include

- Shortest Remaining Time First (SRTF)
- Round Robin (RR)

We shall discuss these scheduling algorithms in detail in a while. However, before we proceed, we have to understand the scheduling criteria.

4.3.10 Scheduling Criteria

Scheduling criteria are the parameters that are used to compare different scheduling algorithms. They are:

1. CPU utilization
2. Throughput
3. Turnaround time
4. Waiting time
5. Response time

4.3.10.1 CPU utilization

We can compare CPU utilization to our AVR room booking, where we measure how much time the room is engaged. CPU utilization is a measure of CPU usage in terms of how busy the processor is or how much is the load on the processor.

4.3.10.2 Throughput

Throughput can be compared to AVR room examples, which measure how many room allocations have been completed. Throughput measures the work being done. It is the number of processes that are completed per unit time.

4.3.10.3 Turnaround time (TAT)

Turnaround time (TAT) can be compared to an AVR room booking example, which measures the time when the request for the room was made and the room was allocated.

Turnaround time is the interval time between the submission time of the process and its completion time. TAT includes the sum of time spent waiting to get into memory, waiting in the ready queue, executing in CPU, and waiting for I/O. TAT is limited by the speed of the output device.

Now, you might be wondering how we can calculate the TAT. The formula for TAT is as follows:

Let ST be the submission time (the time when the process is dispatched from the ready queue), CT be the completion time (the time when the process completes), AT be the arrival time (the time when the process enters the system), respectively of a process P.

Then:

$$TAT = CT - AT \quad (1)$$

4.3.10.4 Waiting Time (WT)

In the AVR room example, Waiting Time (WT) measures the time the department had to wait to be assigned the AVR room for holding the meeting. Waiting Time (WT) measures the time a process waits in the ready queue. Waiting time is directly affected by the type of CPU scheduling algorithm.

Now, let us see how we can calculate the waiting time.

The formula for WT:

Let TAT be the turnaround time, and BT be the CPU burst time (How much time CPU is required). Then

$$WT = TAT - BT \quad (2)$$

4.3.10.5 Response Time (RT)

In the case of the AVR room example, response time can be compared to the time taken to decide which department will be given the AVR room first for the meeting.

Response Time (RT) is a measure of time taken to produce the first response for a process (before it is output to the user). It is a metric for interactive/time-sharing systems.

In conclusion, the best scheduling algorithms are desirable to have maximized CPU utilization and throughput, but with minimized turnaround time, waiting time and response time. Average turnaround time and waiting time can also be used.

Now, we have learnt all the criteria that are used to measure the efficiency of different scheduling algorithms. Now, let us look into the algorithms in detail.

4.3.10.6 Scheduling Algorithms and Criteria

As discussed earlier, the scheduling algorithms are classified into two types. They are:

1. Non-Preemptive Algorithms
2. Preemptive Algorithms

4.3.11 Non-Preemptive Algorithms

As mentioned before, a non-preemptive algorithm can be compared to booking an AVR by the people according to the order in which they have requested the room, and no priorities are assigned.

Non-preemptive Algorithms are those algorithms where once the process enters a running stage, it cannot be blocked until it completes its allocated time. Examples of Non-preemptive Algorithms include

1. First Come First Serve (FCFS)
2. Shortest Job First (SJF)

4.3.11.1 First Come First Serve (FCFS)

In this algorithm, processes are assigned to the CPU in the order they request. Here, there is only one ready queue. As soon as the first job enters the system, it is started immediately and allowed to run as long as it wants to. When the other processes enter, they are put onto the end of the queue.

Now, you might be wondering what would happen when a running process is blocked. When a running process is blocked, the very first process in the queue is run next.

In the same manner, when a blocked process becomes ready, it is put at the end of the queue like a newly arrived job.

Let us take a real-world scenario to understand this situation. It is similar to a person willing to go to the theatre to wait and collect a movie ticket for the first show at 2.00 A.M., i.e. the person who arrives first gets the ticket early.

So, to summarise, we can say that the criteria for FCFS are arrival time (AT) and the mode of working, which is non-preemptive. Non-preemptive means non-interrupted.

Let us do a simple example to help you understand how the criteria act as a measure of the performance of the scheduling algorithms.

Suppose we are given the process name (P) and Arrival Time (AT) as well as Burst Time (BT), as shown in the following table 4.3.1.

In order to calculate the Completion Time (CT), Turnaround Time (TAT) and Wait Time (WT), we use a Gantt chart. Gantt chart helps represent the scheduling process graphically. In the Gantt chart, the x-axis represents the timeline. The timeline is based on the added-up BT. The processes are indicated in the rectangular box.

Table 4.3.1 Example 1 FCFS

Process(P)	AT	BT
P1	0	4
P2	2	5
P3	3	3
P4	4	2
P5	5	1
P6	6	6

Following figure 4.3.4 shows the Gantt chart for Table 4.3.1.

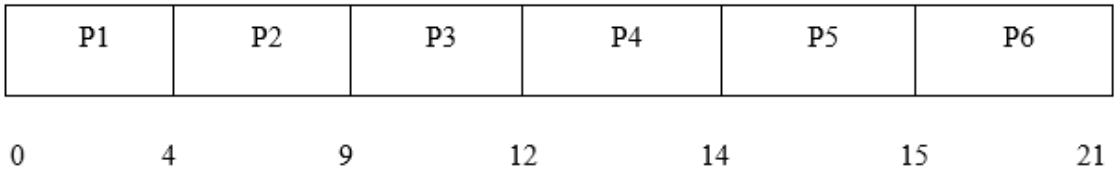


Fig. 4.3.4 Gantt chart for table 4.3.1

As you can see, the BT for P1 is 4. This is represented at the end of P1 in the Gantt Chart. Now, moving to P2, BT is 5. The Completion Time (CT) of P2 will be an added sum of the CT of P1 and BT of P2, i.e., P1 CT + P2 BT.

i.e. 4 + 5 = 9

Similarly, for P3 the timeline value will be P2 CT + P3 BT

9 + 3 = 12

In the case of P4, the CT will be

12 + 2 = 14

In the case of P5, the CT will be

14 + 1 = 15

In the case of P6, the CT will be

15 + 6 = 21

Let us tabulate CT so that it will be easier to understand

The following table 4.3.2 shows table 4.3.1. along with CT

Table 4.3.2 Example 1 FCFS with CT

Process(P)	AT	BT	CT
P1	0	4	4
P2	2	5	9
P3	3	3	12
P4	4	2	14
P5	5	1	15
P6	6	6	21

As we discussed earlier, TAT = CT – AT

For example, in the case of P1, TAT = 4 – 0 = 4

in the case of P2, TAT = 9 – 2 = 7

in the case of P3, TAT = 12 – 3 = 9

in the case of P4, TAT = 14 – 4 = 10

in the case of P5, TAT = 15 – 5 = 10

in the case of P6, TAT = 21 – 6 = 15

Let us tabulate TAT so that it will be easier to understand

Table 4.3.3 Example 1 FCFS with CT and TAT

Process(P)	AT	BT	CT	TAT
P1	0	4	4	4
P2	2	5	9	7
P3	3	3	12	9
P4	4	2	14	10
P5	5	1	15	10
P6	6	6	21	15

As we discussed earlier, WT = TAT – BT

For example, in the case of P1, WT = 4 – 4 = 0

in the case of P2, WT = 7 – 5 = 2

in the case of P3, WT = 9 – 3 = 6

in the case of P4, WT = 10 – 2 = 8

in the case of P5, WT = 10 – 1 = 9

in the case of P6, WT = 15 – 6 = 9

Let us tabulate WT so that it will be easier to understand

Table 4.3.4 Example 1 FCFS with CT, TAT, and WT

Process(P)	AT	BT	CT	TAT	WT
P1	0	4	4	4	0

P2	2	5	9	7	2
P3	3	3	12	9	6
P4	4	2	14	10	8
P5	5	1	15	10	9
P6	6	6	21	15	9

TAT Average = (4+7+9+10+10+15)/6 =55/6 = 9.1667

WT average = (0+2+6+8+9+9)/6 = 34/6 = 5.667

4.3.11.2 Shortest Job First (SJF)

Let us take our earlier AVR example to understand this. SJF can be compared to the AVR example, where the booking criteria are based on the amount of time for the meeting. The department that requires the least amount of time for the meeting is given priority, followed by other departments with the next shortest time requirement.

In the case of SJF, when several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the shortest job first. i.e. the allocation of CPU to the process is done according to the smallest next CPU burst (BT).

Let us consider the following example in Table 4.3.5.

Table 4.3.5 Example 2 SJF

Process(P)	AT	BT
P1	1	7
P2	2	5
P3	3	1
P4	4	2
P5	5	8

Figure 4.3.5 shows the Gantt chart for table 4.3.5.



Fig 4.3.5 Gantt chart for table 4.3.5

Now, you must have noticed that at 0, there are no processes in the Gantt chart. This is because at AT 0, there are no processes. At AT 1, you will see that P1 has arrived. So P1 is in the Gantt chart at AT 1. The criteria of SJF is BT. From now on, the scheduler will look into the process with the least BT.

As you can see, the BT for P1 is 7. This is represented at the end of P1 in the Gantt chart. Now, the AT of P1 is added to BT of P1 to get the CT of P1, which is equal to 8.

The least BT is for P3, and BT is 1. So, the next process P3 is added to the Gantt chart.
CT of P3 = 8 + BT P3

= 8 + 1

= 9

The process with the next least BT is P4, and the BT of P4 is 2. So, the next process to be added to the Gantt chart is P4.

CT of P4 = 9 + BT P4

= 9 + 2

= 11

The process with the next least BT is P2, and BT of P2 is 5. So, the next process to be added to the Gantt chart is P2.

CT of P2 = 11 + BT P2

= 11 + 5

= 16

The process with the next least BT is P5, and BT of P5 is 8. So, the next process to be added into the Gantt chart is P5.

CT of P5 = 16 + BT P5

= 16 + 8

= 24

CT of each process is taken from the first encounter from right to left of a process.

In the next stage, TAT and WT are calculated as discussed in the case of FCFS. The formula for the calculation of TAT and WT is as follows

TAT = CT – AT (1)

WT = TAT – BT (2)

Table 4.3.6 Example 2 SJF with CT, TAT, WT

Process(P)	AT	BT	CT	TAT CT - AT	WT TAT - BT
P1	1	7	8	8 – 1 = 7	7 – 7 = 0
P2	2	5	16	16 – 2 = 14	14 – 5 = 9
P3	3	1	9	9 – 3 = 6	6 – 1 = 5
P4	4	2	11	11 – 4 = 7	7 – 2 = 5
P5	5	8	24	24 – 5 = 19	19 – 8 = 11

Average TAT = (7 + 14 + 6 + 7 + 19)/5 = 10.6

Average WT = (0 + 9 + 5 + 5 + 11)/5 = 6

4.3.12 Preemptive Algorithms

As we mentioned earlier, In AVR scheduling, a chart is maintained to show the times when the AVR is reserved or free. When the executives of the university have decided to hold an urgent meeting, the bookings made by the other departments will be interrupted. This scenario was compared to preemptive algorithms.

Preemptive algorithms are those algorithms based on priority where a scheduler may interrupt a low-priority running process when a high-priority process enters into a ready state.

Examples of preemptive algorithms include

1. Shortest Remaining Time First (SRTF)
2. Round Robin (RR)

4.3.12.1 Shortest Remaining Time First (SRTF)

Shortest Remaining Time First (SRTF) can be compared to the AVR example, where the booking criteria are based on the amount of time for the meeting and the current meeting's remaining time. The department that requires the least amount of time for the meeting is given priority, followed by other departments with the next shortest time requirement. If the time requirement of another department's meeting is less than the current meeting's time requirement, the current meeting is suspended, and a new meeting is started.

Shortest Remaining Time First (SRTF) is the preemptive version of SJF. In this scheme, the scheduler selects the job whose remaining time is the shortest. When a new job arrives, its total time is compared to the current process's remaining time. If the time

requirement of the new job is less than the current process's time requirement, the current process is suspended, and the new job is started.

Though in the real world, this looks impractical, as the CPU works very fast, SRTF works well.

Let us consider the following example in Table 4.3.7.

Table 4.3.7 Example 3 SRTF

Process(P)	AT	BT
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

Table 4.3.8 Example 3 SRTF with BT

Process(P)	AT	BT	CT	TAT CT - AT	WT TAT - BT
P1	0	7 – 6 - 0			
P2	1	5 – 4 - 0			
P3	2	3 - 2 - 1 - 0			
P4	3	1 - 0			
P5	4	2 - 0			
P6	5	1 - 0			

Figure 4.3.6 shows the Gantt chart for Table 4.3.8

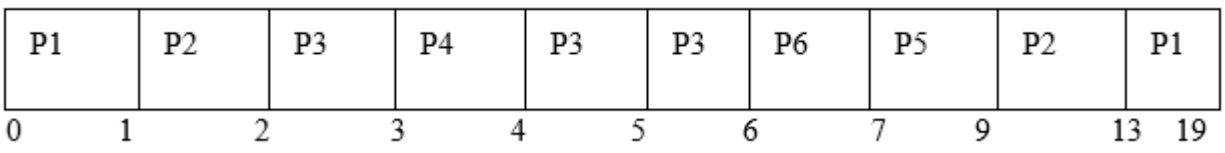


Fig. 4.3.6 Gantt chart for table VII

The criteria of SJF is BT.

- At the 0th unit of the CPU, there is only process P1, so P1 is executed for 1 unit time.
- Process P2 arrives at the 1st unit of time. Now, P1 needs 6 more units to be executed, and P2 needs only 5 units. So, P2 is executed first by preempting P1. (Preempting means interrupting)
- At the 2nd unit of time, the process P3 arrives, and the burst time of P3 is 3 units, which is less than the completion time of P2, which is 4 units, so P3 executed by preempting P1.
- At the 3rd unit of time, the process P4 arrives, and the burst time of P4 is 1 unit, which is less than the completion time of P3, which is 2 units, so P4 is executed by preempting P3.
- At the 4th unit of time, the process P5 arrives, and the burst time of P5 is 2 units, which is the same as the completion time of P3, which is 2 units, so P3 executed as the process arriving earlier has higher preference.
- At the 5th unit of time, the process P6 arrives, and the burst time of P6 is 1 unit, which is the same as the completion time of P3, which is 1 unit, so P3 executed as the process arriving earlier has higher preference.
- In the next stage, process P6 with completion time 1 is executed and completed
- P5 gets completed with completion time 2 in the next unit.
- While comparing P1 and P2, the completion time of P2 is 4 while that of P1 is 6. Hence, P2 is completed first, followed by P1.

The Completion Time CT is calculated from Figure 4.3.6 by taking the CT from Right to Left.

For example, the first encounter of P1 is at unit 19 while moving from Right to Left. Similarly, CT for P2 is 13. This is represented in the following table: 4.3.9

Table 4.3.9 Example 3 SRTF with CT

Process(P)	AT	BT	CT	TAT CT - AT	WT TAT - BT
P1	0	7 – 6 - 0	19		
P2	1	5 – 4 - 0	13		
P3	2	3 - 2 - 1 - 0	6		
P4	3	1 - 0	4		
P5	4	2 - 0	9		
P6	5	1 - 0	7		

TAT is calculated as CT – AT.

WT is calculated as TAT – BT.

The following Table X shows the TAT and WT calculated.

Table 4.3.10 Example 3 SRTF with CT, TAT, WT

Process(P)	AT	BT	CT	TAT CT - AT	WT TAT - BT
P1	0	7 – 6 - 0	19	19	12
P2	1	5 – 4 - 0	13	12	7
P3	2	3 - 2 - 1 - 0	6	4	1
P4	3	1 - 0	4	1	0
P5	4	2 - 0	9	5	3
P6	5	1 - 0	7	2	1

The Average TAT is calculated as dividing the total TAT by no. of process

Average TAT = (19 + 12 + 4 + 1+ 5 + 2) / 6 = 7.17

The Average WT is calculated as dividing the total WT by no. of process

Average WT = (12 + 7 + 1 + 0 + 3 + 1) / 6 = 4

4.3.12.2 Round Robin (RR)

The algorithm gets its name from the round-robin principle, which states that each person gets an equal share of something in turn. Round robin works on the assumption that all processes are equally important. In round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. Here, each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

An example of a high-priority process is a process displaying a video film on the screen in real time. However, compared to this process, a daemon process sending an email has lower priority. The concept here is that the CPU time (in units of time slice/time quantum) is shared among the requesting processes in a round-robin sequence. Here, the criteria is Time Quantum (TQ).

The Round Robin algorithm can be compared to the AVR example, where the booking criteria is based on the amount of time for the meeting and a maximum duration of, say, 2 hours per meeting (TQ in RR). If the time required is more than 2 hours, the department can do the meeting for 2 hours, then leave the room and wait for the next AVR room allocation to do the rest of the meeting. Here, the department's time requirement is

compared with the TQ. If the meeting agenda is complete, then the AVR booking for that department is completed, and the next department is assigned the AVR room in the same fashion.

Let us consider the following example in Table 4.3.11.

In the following example, there are six processes named P1, P2, P3, P4, P5, P6 and their arrival time and burst time are shown in table 4.3.11. The time quantum of the system is 4 units.

Table 4.3.11 Example 4 RR

Process(P)	Arrival Time(AT)	Burst time (BT)
P1	0	5
P2	1	6
P3	2	3
P4	3	1
P5	4	5
P6	6	4

According to the algorithm, we need to update both the ready queue and the Gantt chart after each scheduling event.

1. Initially, at time 0, only process P1 arrives, which will be scheduled for the time slice 4 units. So, in the ready queue, there will be only one process P1, starting with CPU burst time 5 units.

Table 4.3.12 Ready queue at time 0

P1
5

At time 0, process P1 is in the ready queue and it will execute 4 units first because the time quantum of the system is 4 units. After the execution of the P1 Gantt chart is shown in Figure 4.3.7

Gantt chart

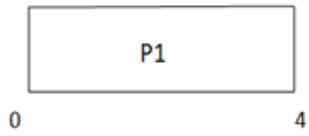


Fig. 4.3.7 Gantt chart after execution of P1

2. At time 4, five processes P1, P2, P3, P4, P5 are in the ready queue. Process P1 has not yet completed its execution and requires an additional 1 unit of time. Therefore, it will be reinserted into the ready queue.

Ready queue

Table 4.3.13 Ready queue after execution of P1

P2	P3	P4	P5	P1
6	3	1	5	1

Process P1 and process P2 will be executed for 4 units of time, as illustrated in the Gantt chart shown in Figure 4.3.8

Gantt chart

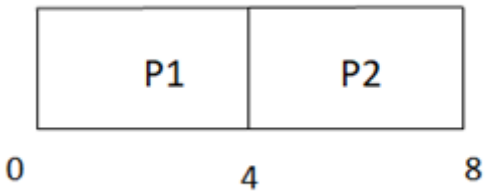


Fig.4.3.8 Gantt chart after execution of P2

3. During the execution of P2, another process, P6, arrives in the ready queue. Since P2 has not yet completed its execution, it will be reinserted into the ready queue with its remaining burst time of 2 units, as shown in Table 4.3.14

Ready queue

Table 4.3.14 ready queue after execution of P2

P3	P4	P5	P1	P6	P2
3	1	5	1	4	2

After P1 and P2, process P3 will be executed for 3 units of time, as its CPU burst time is only 3 seconds. Gantt chart after execution of P3 is shown in Figure 4.3.9

Gantt chart

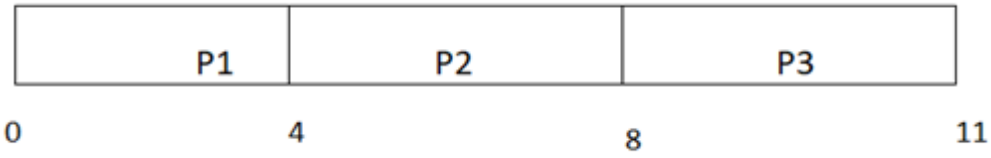


Fig. 4.3.9 Gantt chart after execution of P3

4. Since P3 has completed its execution, it will be terminated and not added back to the ready queue. The next process to be executed will be P4.

Ready queue

Table 4.3.15 Ready queue after execution of P3

P4	P5	P1	P6	P2
1	5	1	4	2

Next, process P4 will be executed. Since its burst time is only 1 unit, which is less than the time quantum, it will be completed as shown in Figure 4.3.10

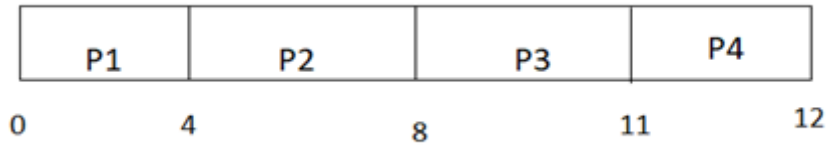


Fig. 4.3.10 Gantt chart after execution of P4

5. The next process in the ready queue is P5, which has a burst time of 5 units. Since P4 has completed its execution, it will not be added back to the queue.

Ready queue

Table 4.3.16 Ready queue after execution of P4

P5	P1	P6	P2
5	1	4	2

Next, P5 will be executed for the entire time quantum, that is 4 units of time. This is because it requires 5 units of burst time, which exceeds the duration of the time slice. The Gantt chart after the execution of Process P5 is shown in Figure 4.3.11.

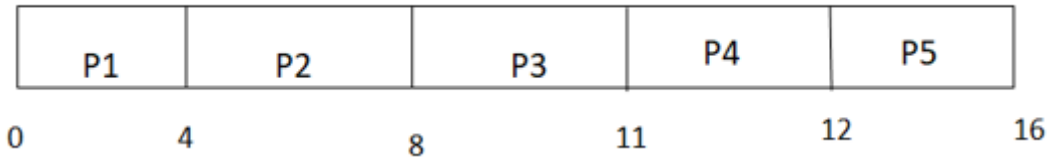


Fig 4.3.11 Gantt chart after execution of P5

6. P5 has not yet completed its execution and will be reinserted into the queue. It will be added with the remaining burst time of 1 unit.

Ready queue

Table 4.3.17 Ready queue after execution of P5

P1	P6	P2	P5
1	4	2	1

Process P1 will be allocated the next turn to finalise its execution. It requires only 1 unit of burst time. Consequently, it will be completed during this turn.

Gantt chart

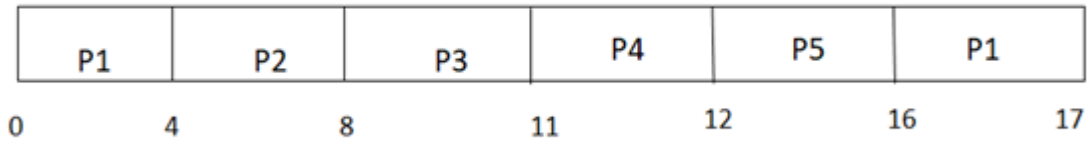


Fig. 4.3.12 Gantt chart after execution of P1

7. Since P1 is completed, it will not be reinserted into the ready queue. The next process, P6, requires only 4 units of burst time and will be executed next, as shown in table 4.3.18.

Ready queue

Table 4.3.18 Ready queue after execution of P1

P6	P2	P5
4	2	1

P6 will be executed for 4 units of time. It will continue running until it is completed. Gantt chart after execution of P6 is shown in Figure 4.3.13

Gantt chart

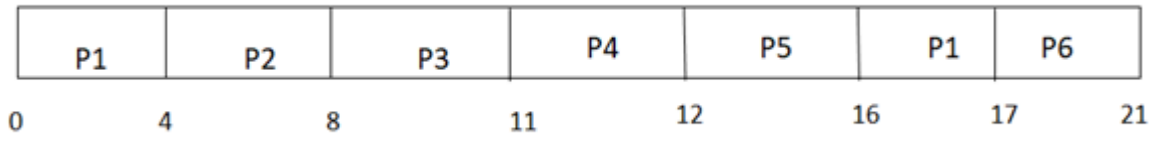


Fig 4.3.13 Gantt chart after execution of P6

8. Since P6 has been completed, it will not be reinserted into the queue. Only two processes are remaining in the ready queue. The next process, P2, requires only 2 units of time. Now, the status of the ready queue is shown in Table 4.3.19.

Ready queue

Table 4.3.19 ready queue after execution of P6

P2	P5
2	1

P2 will be executed again, as it requires only 2 units of time. Consequently, it will be completed during this turn. Gantt chart after execution of P2 is shown in figure 4.3.14

Gantt chart

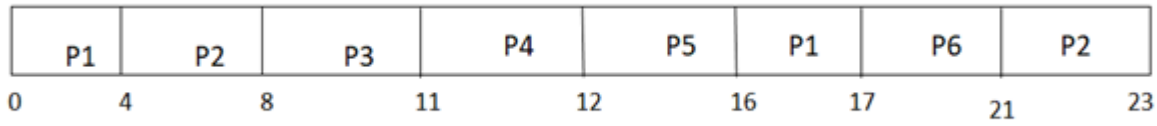


Fig. 4.3.14 Gantt chart after execution of P2

9. Currently, the only process remaining in the queue is P5, which requires 1 unit of burst time. Given that the time slice is 4 units, P5 will be completed in the next burst.

Ready queue

Table 4.3.20 ready queue after execution of P2

P5
1

P5 will get executed till completion.

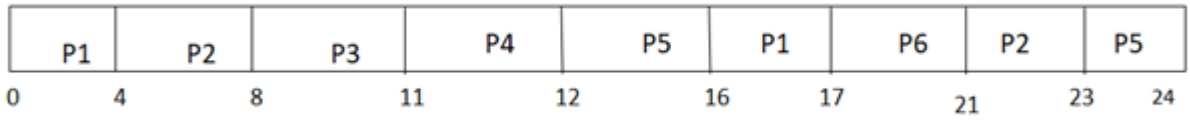


Fig. 4.3.15 Final Gantt chart after completion of P5

The Completion Time CT is calculated from Figure 4.3.15. For example, the completion time of P1 is 4, and P2 is 8 etc., as shown in Table 4.3.21 below.

Table 4.3.21 Example 4 RR with CT

Process(P)	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)
P1	0	5	17		

P2	1	6	23		
P3	2	3	11		
P4	3	1	12		
P5	4	5	24		
P6	6	4	21		

TAT is calculated as CT – AT.

WT is calculated as TAT – BT.

The following table shows the TAT and WT calculated.

Table 4.3.22 Example 4 RR with CT, TAT, WT

Process(P)	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)
P1	0	5	17	17	12
P2	1	6	23	22	16
P3	2	3	11	9	6
P4	3	1	12	9	8
P5	4	5	24	20	15
P6	6	4	21	15	11

The Average TAT is calculated as dividing the total TAT by no of process

Average TAT = (17 + 22 + 9 + 9 + 20 + 15) / 6

= 15.3

The Average WT is calculated as dividing the total WT by no of process

Average WT = (12 + 16 + 6 + 8 + 15 + 11) / 6

= 11.3

Recap

- Process scheduling - removal of the current running process and selection of another process
- In an OS, the scheduler is responsible for selecting a process for scheduling Types of Scheduler
    1. Long term scheduler
    2. Short term scheduler
    3. Medium term scheduler
- Long term scheduler - maintain a good degree of multiprogramming.
- Short term scheduler is invoked every time the CPU requires a new process for execution.
- Medium term scheduler helps in swapping.
- Context switching is the task of switching the CPU from one process to another.
- Non-preemptive Algorithms are those algorithms where once the process enters a running stage, it cannot be blocked until it completes its allocated time.
- Examples:
    1. First Come First Serve (FCFS)
    2. Shortest Job First (SJF)
- Preemptive algorithms are based on priority, where a scheduler may interrupt a low-priority running process when a high-priority process enters into a ready state.
- Examples:
    1. Shortest Remaining Time First (SRTF)
    2. Round Robin (RR)

Objective Type Questions

1. What is the process of removing the current running process and selecting another process on the basis of a particular strategy by the process manager of the OS?
2. What consists of all the processes in the system?

3. When a process enters a system and is in a ready state, they are placed in which queue?

4. Those processes that are waiting for a device to be available are placed in which queue?

5. What is the interval from the time of submission of a process to the time of completion?

6. Which module controls the CPU for the process selected by the short-term scheduler?

7. Which scheduler is invoked every time the CPU requires a new process for execution?

8. Which scheduler helps in swapping?

9. What is the task of switching the CPU from one process to another, saving the state of the old process and loading the saved state of the new process?

10. In the priority scheduling algorithm, when a process arrives at the ready queue, its priority is compared with the priority of which process?

Answers to Objective Type Questions

1. Process scheduling

2. Process Control Block(PCB)

3. Ready queue

4. Device queue or I/O queue

5. turnaround time

6. Dispatcher

7. Short term scheduler

8. Medium term scheduler

9. Context switching

10. Currently running process

Assignments

1. Differentiate between long-term, midterm, and short-term scheduler

2. Compare priority vs non-priority scheduling

3. Consider the set of 5 processes whose arrival time and burst time are given below-

Process id	Arrival time(AT)	Burst time(BT)
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

If the CPU scheduling policy is FCFS, you need to calculate the average waiting time. Additionally, determine the average turnaround time.

4. Consider the set of 5 processes whose arrival time and burst time are given below-

Process id	Arrival time(AT)	Burst time (BT)
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

If the CPU scheduling policy is SJF non-preemptive, calculate the average waiting time and average turnaround time.

5. Consider the set of 5 processes whose arrival time and burst time are given below-

Process id	Arrival time(AT)	Burst time (BT)
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

If the CPU scheduling policy is Round Robin with a time quantum of 2 units, you need to determine the average waiting time. Additionally, determine the average turnaround time.

6. Consider the following set of 6 processes with their respective arrival times and burst times provided below.

Process id	Arrival time (AT)	Burst time (BT)
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

If the CPU scheduling policy is Shortest Remaining Time First (SRTF), determine the average waiting time and the average turnaround time.

References

1. Silberschatz, Abraham, and Peter Baer Galvin. "Operating System Concepts." (2013).

2. Andrew, S. Tanenbaum. "Modern operating systems Pearson." (2014).

3. S TANENBAUM, A. N. D. R. E. W., and HERBERT BOS. "Operating Systems." (2015).

Suggested Reading

1. GALVIN, PETER BAER, and GREG GAGNE. "ABRAHAM SILBERSCHATZ." (2005).

2. Crowley, Charles. Operating systems: a design-oriented approach. McGraw-Hill Professional, 1996.

3. Harris, J. Archer Archer, and John Cordani. Schaum's outline of operating systems. McGraw-Hill, Inc., 2001.



Multiple Processor Scheduling

Learning Outcomes

By the completion of the unit, the learner will be able to:

- explain the concept of multi-processor scheduling
- identify the various approaches to multiprocessor scheduling
- understand Symmetric Multiprocessor and its components
- familiarise with the concept of Virtualization and Threading

Prerequisites

A multiprocessor is a system with several processors. In the presence of multiple processors, it becomes complex to design a scheduling algorithm. The multiprocessor system can be categorised into:

Loosely Coupled or Distributed Multiprocessor: Multiple processors in the system are independent of each other. Each processor in the system has its own memory and I/O channels.

Functionally Specialized Processor: In this system, among the collection of multiple processors, there is a master processor, which is a general-purpose processor. This master processor controls the other specialized processors in the system and provides services to them.

Tightly Coupled Multiprocessors: In this system, the processors are under the integrated control of the operating system. All the processors in this system share common memory. These processors are sometimes also termed as homogeneous as they are identical in terms of their functionality.

The multiple CPUs in the system are in close communication which shares a common bus, memory, and other peripheral devices. So, we can say that the system is tightly coupled. These systems are used when we want to process a bulk amount of data, and these systems are mainly used in satellite, weather forecasting, etc. There are cases when the processors are identical, i.e., homogenous, in terms of their functionality in multiple-processor scheduling. Multiple processor scheduling for multiprocessor scheduling focuses on designing the scheduling function for the system, which consists of ‘more than one processor’.

Keyterms

Processor Affinity, Load Balancing, Symmetric Multithreading, Master-Slave Multiprocessor, Virtualization

Discussion

4.4.1 Multiple-Processor Scheduling in Operating System

Multiple processor scheduling for multiprocessor scheduling focuses on designing the system's scheduling function, which consists of more than one processor. Multiple CPUs share the load (load sharing) in multiprocessor scheduling so that various processes run simultaneously. In general, multiprocessor scheduling is more complex than single-processor scheduling. In multiprocessor scheduling, there are many processors, and they are identical, and we can run any process at any time.

Multiprocessor systems may be heterogeneous (different kinds of CPUs) or homogenous (the same CPU). There may be special scheduling constraints, such as devices connected via a private bus to only one CPU. There is no policy or rule which can be declared as the best scheduling solution for a system with a single processor. Similarly, there is no best scheduling solution for a system with multiple processors as well.

4.4.2 Approaches to Multiple-Processor Scheduling

One approach is when all the scheduling decisions and I/O processing are handled by a single processor, which is called the Master Server, and the other processors execute only the user code. This is simple and reduces the need for data sharing. This entire scenario is called **Asymmetric Multiprocessing**.

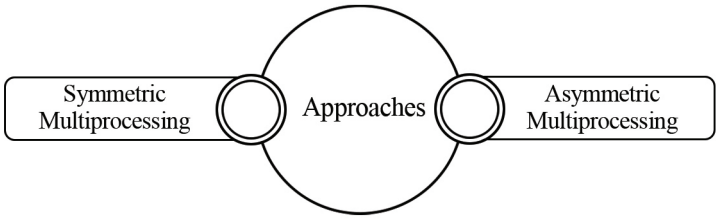


Fig. 4.4.1 Approaches to Multiple-Processor Scheduling

A second approach uses **Symmetric Multiprocessing** where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

4.4.2.1 Processor Affinity

Processor Affinity means a process has an **affinity** for the processor on which it is currently running. When a process runs on a specific processor, certain effects are on the

cache memory. The data most recently accessed by the process populates the cache for the processor, and as a result, successive memory accesses by the process are often satisfied in the cache memory. Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most **SMP** (symmetric multiprocessing) systems try to avoid the migration of processes from one processor to another and try to keep a process running on the same processor. This is known as **processor affinity.**

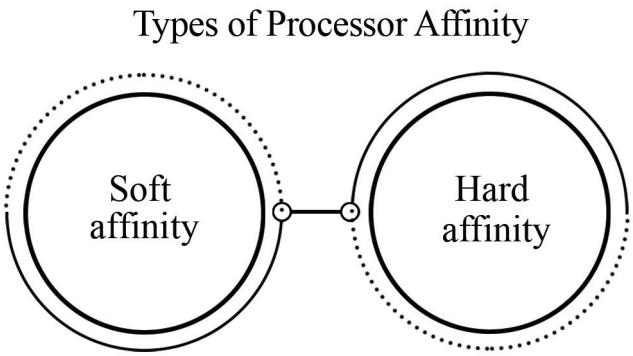


Fig 4.4.2 Types of Processor Affinity

There are two types of processor affinity:

1. Soft Affinity – When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.

2. Hard Affinity – Hard Affinity allows a process to specify a subset of processors on which it may run. Some systems, such as Linux, implement soft affinity but also provide some system calls like **sched_setaffinity()** that support hard affinity.

4.4.2.2 Load Balancing

Load Balancing is the phenomenon which keeps the workload evenly distributed across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of processes which are eligible to execute. Load balancing is unnecessary because once a processor becomes idle, it immediately extracts a runnable process from the common run queue. On SMP (symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else. One or more processors will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.

There are two general approaches to load balancing, as given in Figure 4.4.3

1. Push Migration – In push migration, a task routinely checks the load on each processor, and if it finds an imbalance, then it evenly distributes the load on each processor by moving the processes from overloaded to idle or less busy processors.

2. Pull Migration – Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

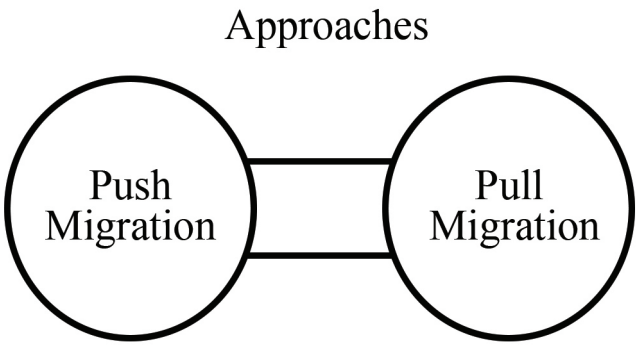


Fig 4.4.3 Approaches

4.4.2.3 Multicore Processors

In multicore processors, multiple processor cores are placed on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.

However, multicore processors may complicate the scheduling process. When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation is called a memory stall. It occurs for various reasons, such as cache miss, which is accessing the data that is not in the cache memory. In such cases, the processor can spend up to fifty percent of its time waiting for data to become available from memory. To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore, if one thread stalls while waiting for the memory, the core can switch to another thread.

There are two ways to multithread a processor:

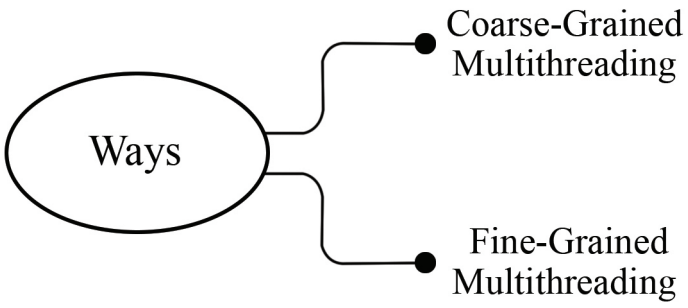


Fig 4.4.4 Ways of multithreading a processor

1. **Coarse-Grained Multithreading** – In coarse-grained multithreading, a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor

must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution, it will fill the pipeline with its instructions.

2. **Fine-Grained Multithreading** - This multithreading switches between threads at a much finer level, mainly at the boundary of an instruction cycle. The architectural design of fine-grained systems includes logic for thread switching, and as a result, the cost of switching between threads is small.

4.4.2.4 Symmetric Multiprocessor

Symmetric Multiprocessors (SMP) is the third model. There is one copy of the OS in memory in this model, but any central processing unit can run it. Now, when a system call is made, the central processing unit on which the system call was made traps the kernel and processes that system call. This model balances processes and memory dynamically. This approach uses Symmetric Multiprocessing, where each processor is self-scheduling.

The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute. In this system, it is possible that all the processes may be in a common ready queue, or each processor may have its own private queue for the ready process. There are three main sources of contention in a multiprocessor operating system.

- **Locking system**: As we know, the resources are shared in the multiprocessor system, and there is a need to protect these resources for safe access among the multiple processors. The main purpose of the locking scheme is to serialize access to the resources by the multiple processors.
- **Shared data:** When multiple processors access the same data at the same time, then there may be a chance of inconsistency of data, so to protect this, we have to use some protocols or locking schemes.
- **Cache coherence**: It is the shared resource data that is stored in multiple local caches. Suppose two clients have a cached copy of memory, and one client changes the memory block. The other client could be left with an invalid cache without notification of the change, so this conflict can be resolved by maintaining a coherent view of the data.

4.4.2.5 Master-Slave Multiprocessor

In this multiprocessor model, there is a single data structure that keeps track of the ready processes. In this model, one central processing unit works as a master and another as a slave. All the processors are handled by a single processor, which is called the master server (refer to Fig. 4.4.5)

The master server runs the operating system process, and the slave server runs the user processes. The memory and input-output devices are shared among all the processors, and all the processors are connected to a common bus. This system is simple and reduces data sharing, so it is called asymmetric multiprocessing.

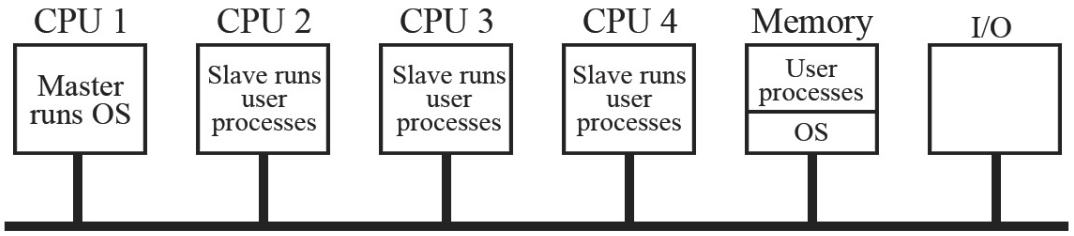


Fig. 4.4.5 Master-Slave Multiprocessor

4.4.2.6 Virtualization and Threading

In this type of multiple-processor scheduling, even a single CPU system acts as a multiple-processor system. In a system with Virtualization, the virtualization presents one or more virtual CPUs to each of the virtual machines running on the system. Then, it schedules the use of the physical CPU among the virtual machines.

- Most virtualised environments have one host operating system, and many guest operating systems, and the host operating system creates and manages the virtual machines.
- Each virtual machine has a guest operating system installed, and applications run within that guest.
- Each guest operating system may be assigned for specific use cases, applications, or users, including time-sharing or real-time operation.
- Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization.
- A time-sharing operating system tries to allot 100 milliseconds to each time slice to give users a reasonable response time. A given 100-millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more, which results in a very poor response time for users logged into that virtual machine.
- The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all cycles and scheduling all of those cycles. The time-of-day clocks in virtual machines are often incorrect because timers take no longer to trigger than they would on dedicated CPUs.

Virtualisations can thus undo the good scheduling algorithm efforts of the operating systems within virtual machines.

Recap

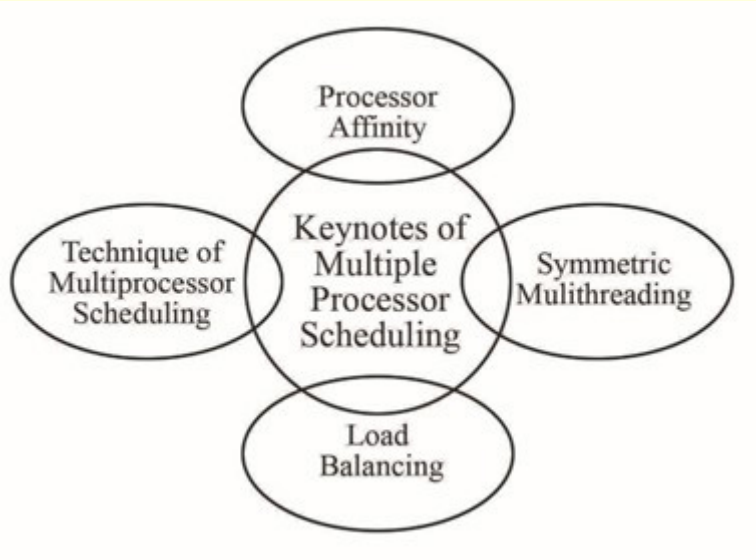


Fig 4.4.6 Different Types of Multiprocessors

- Asymmetric multiprocessing: all the scheduling decisions and I/O processing are handled by a single processor, which is called the Master Server and the other processors execute only the user code.
- Symmetric Multiprocessing: Each processor is self-scheduling.
- Soft affinity: It refers to a policy where an operating system attempts to keep a process running on the same processor but does not guarantee that this will always be the case.
- Hard Affinity: It allows a process to specify a subset of processors on which it may run.
- Load balancing: It involves distributing tasks evenly across all available processors to ensure that no single processor is overwhelmed while others are underutilized.
- Push migration:
  - Here, a task routinely checks the load on each processor.
  - If it finds an imbalance, then it evenly distributes the load on each processor by moving the processes from overloaded to idle or less busy processors.
- Multicore processors: multiple processor cores are placed on the same physical chip.

Objective Type Questions

1. Which approach for multiprocessor thread scheduling and processor assignment processes is not assigned to a particular processor?
2. Which approach does the scheduling routine of the operating system run of that processor to select the next thread?
3. Which multiprocessor system contains a master-slave relationship?
4. What is a common synchronization mechanism used in multiprocessor operating systems known as?
5. What are the two types of multiprocessor scheduling?
6. When a processor accesses memory, then it spends a significant amount of time waiting for the data to become available. What is this phenomenon termed?
7. What scheduling approach involves distributing tasks evenly among all processors?
8. Which scheduling method assigns tasks based on previous processor usage patterns?

Answers to Objective Type Questions

1. Load Sharing
2. Load Sharing
3. Asymmetric Multiprocessors
4. Locks
5. Centralized scheduling (asymmetric multiprocessing) or distributed scheduling (symmetric multiprocessing)
6. Memory stall
7. Load balancing
8. Affinity

Assignments

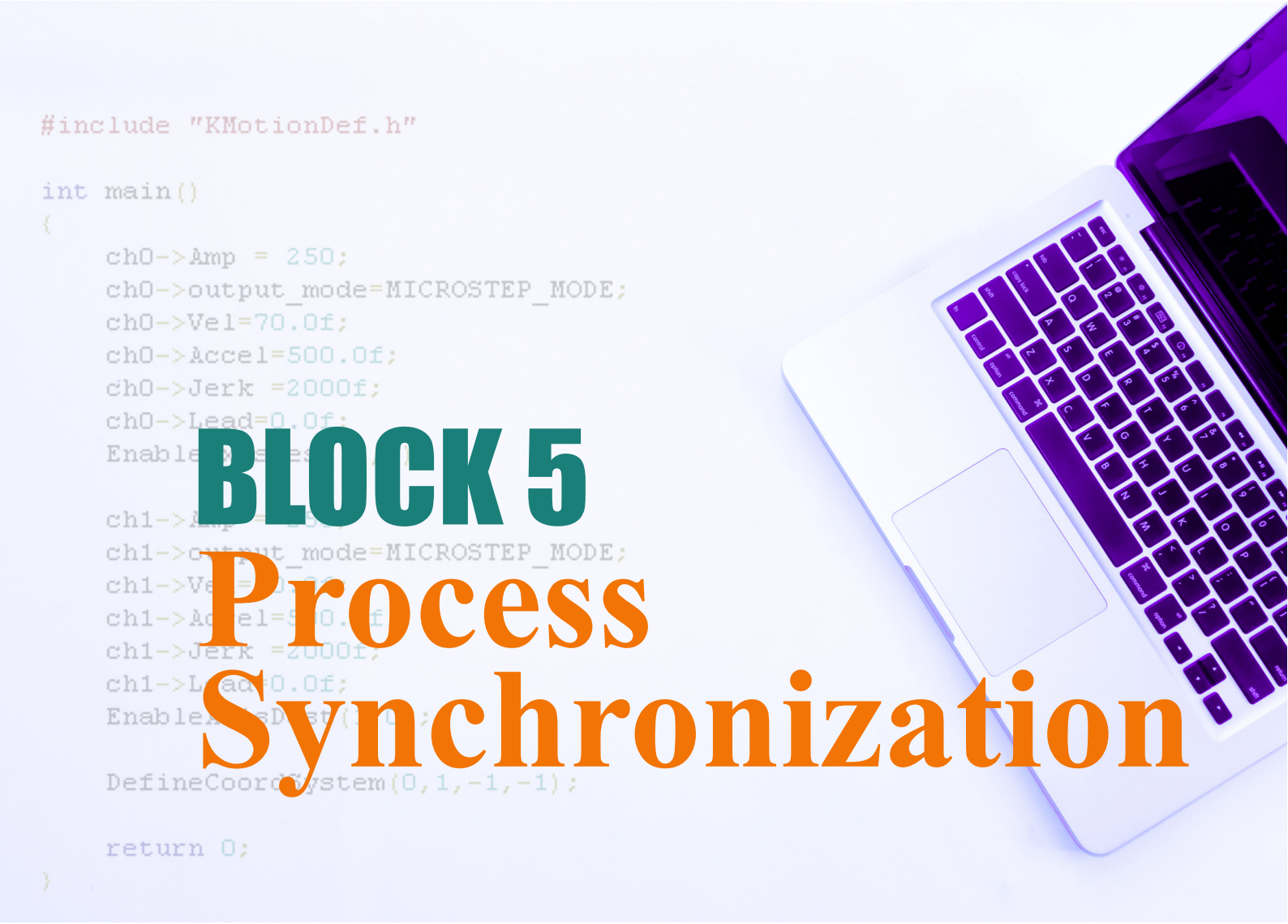
1. Explain processor affinity in detail.
2. Explain concepts of load balancing in symmetric multiprocessing.
3. Explain multi-core processors in detail.
4. Explain master slave multiprocessor.
5. Explain virtualization and threading concepts.

References

1. Silberschatz, Abraham, and Peter Baer Galvin. "Operating System Concepts." (2013).
2. Andrew, S. Tanenbaum. "Modern operating systems Pearson." (2014).
3. S TANENBAUM, A. N. D. R. E. W., and HERBERT BOS. "Operating Systems." (2015).

Suggested Reading

1. GALVIN, PETER BAER, and GREG GAGNE. "ABRAHAM SILBERSCHATZ." (2005).
2. Crowley, Charles. Operating systems: a design-oriented approach. McGraw-Hill Professional, 1996.
3. Harris, J. Archer Archer, and John Cordani. Schaum's outline of operating systems. McGraw-Hill, Inc., 2001.
4. Andrew, S. Tanenbaum. "Modern operating systems Pearson." (2014).



BLOCK 5
Process Synchronization



UNIT 1 Interprocess Communication

Learning Outcomes

By the completion of this unit, the learner will be able to:

- be aware of the process and operations of the process.
- identify different methods for interprocess communication.
- narrate the issues in interprocess communication.

Prerequisites

In this unit, we discuss the concept of process in detail. Before we go into the details of a process, let us understand what a process is.

We do several tasks simultaneously. A mother, while engaging herself in the kitchen busy with the preparation of breakfast, will process her brain with multiple tasks – it could be about waking up the children and getting them ready, answering the call of a newspaper boy for clearing the monthly newspaper bill or reminding her husband about an important function to be attended on that day! Whatever the task, when she switches from one task to another, she knows exactly from which point she temporarily stopped the current task, and when she returns to the current task, she will resume her task from that point.

Here, we can see that every task that is done by the mother is specific – it has its own start time and end time, and it requires its own resources. This will help her to manage all the multiple tasks to be done by her – she can track the progress of each of these tasks, and when she is engaged in a task, she temporarily "suspends" the other tasks. Here each task is a separate activity which can be easily tracked by the human brain. This technique is mimicked in modern operating systems when a user wants many tasks to be done at a time.

With present day modern operating systems such as Microsoft Windows or Linux, a user may be able to run several programs simultaneously: a word processor, a web browser, and an e-mail package (Remember that these tasks are identical to the tasks that were handled by a mother in the previous example). Of course, the user does not need to take up all these tasks at the exact same time. He may prepare his document in Word. During the preparation, he may browse the web for information that is to be

included in the document. At last, he may email his document. It may also happen that he plays background music while preparing his document.

How can the operating system track all these activities, similar to the mother, who did a lot of tasks at a time? OS has to allocate resources for each of these programs, track the progress of each of these programs, and make sure that when the user moves from one activity to another, each activity is resumed and not restarted from its previous stop!

The OS makes use of a concept called a process to manage all these tasks. OS will allocate main memory and other resources to each of the programs (email, word processor and browser in the previous example) by creating separate working spaces in memory for each of these programs. In this example, there are 3 programs, and we say that 3 processes will be created by the OS as soon as these 3 programs are chosen for execution by the user. The concept of a process is fundamental in the design of modern operating systems. For example, the mother is able to isolate each task that she may do simultaneously or in serial order, attend to each of them, and finish them. The concept of a process helps the OS handle multiple tasks easily.

We always hear the terms process and program. Is there any difference between them? Let us discuss them.

When we create a program to compile it, it is permanently stored in secondary memory. We have learned that when the program is to be executed, the executable code is to be transferred from secondary memory to the primary memory. The birth of a process occurs at that point. A process is a program in execution.

Keywords

Process, Process States, Process Creation, Process Termination, Interprocess Communi-Cation, Interprocess Communication Issues, Semaphores

Discussion

5.1.1 Concept of Process

A process is a program in execution

In order to understand the scenario better, let us take a real-time example. Suppose you are planning to bake a cake. You have a kitchen well stuffed with all the ingredients and a good recipe. Your ingredients, like flour, egg, baking powder, milk, and so on, are your input, while the recipe is your program. You are the processor. The process includes reading the recipe, taking the ingredients, and baking the cake.

Now, let us imagine a situation where your mother comes running with a burnt finger. You will make a note of where you were in the recipe (process current state is saved),

then take a first aid book and follow the instructions. Here, we see that the processor (You) is switching between processes (first aid, baking) based on priority. (First aid is a higher priority than baking). When the first aid has been provided, you go back to the process of cooking.

So, what do we infer from this example?

The first inference is that a process is an activity of some kind. A process has a program, input, output and state. Another inference is that a single processor may be shared among multiple processes with the help of some priority scheduling schemes.

Before we discuss a process in detail, let us take a real-time example. Suppose we are in an airport and we have taken a boarding pass to travel. The boarding pass contains our name, the destination we are travelling to, our current location, and our seat in the aeroplane.

A process includes:

- Process stack segment
- Data section segment
- Heap segment
- Text segment

If we compare the boarding pass to the structure of a process, the process stack is similar to the section in the boarding pass that specifies the destination we are travelling to as well as our current location. The data section of a process is similar to the section of the boarding pass with our name. The heap section of the process is similar to the seat that we are allocated in the aeroplane. The text section of the process is similar to the time of boarding the flight.

Now, let us discuss the structure of a process in memory. The structure of a process in memory is shown in Figure 5.1.1.

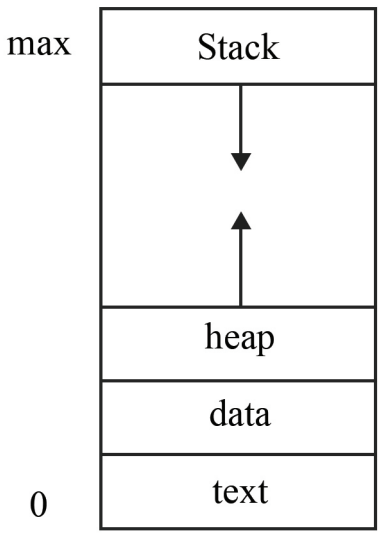


Fig. 5.1.1 A process in memory

As you may see from Figure 5.1.1, a process includes:

- Process stack segment
- Data section segment
- Heap segment
- Text segment

- Text – includes the current activity represented by the program counter and the contents of the processor's registers.
- Data section - contains global variables and static variables.
- Heap - memory that is dynamically allocated during process run time.
- Process stack - contains temporary data (such as function parameters, return addresses, and local variables).

You might be wondering why a process needs all these segments. When process A is switching to process B, all the important information, such as temporary data, global variables, static variables, dynamically allocated memory information, current activity represented by the program counter, and the contents of the processor's registers, needs to be saved. This is the purpose of stack segment, heap segment, data segment, and text segment of a process.

An example of a program is a list of instructions stored on a disk (often called an executable file), whereas a process is an active entity with a program counter specifying the next instruction to execute and a set of associated resources.

Usually, the program is not present in the memory. It will be on the hard disk. A program becomes a process when an executable file is loaded into the memory (i.e. RAM – Random Access Memory.

When two processes are associated with the same program, they are not considered as two separate execution sequences. For example, the same user may invoke many copies of the web browser program. Each of these is a separate process, and although the text sections are equivalent, the data, heap, and stack sections vary. It is also typical for a process to create multiple subprocesses during its execution.

Table 5.1.1 shows the difference between a process and a program.

Table 5.1.1 Difference between a process and a program

Sl. No	Program	Process
1	It is a set of instructions that have been designed to complete certain tasks.	It is an instance of a program that is currently executing.

2	A program is stored in secondary memory and, hence, is a passive entity	A process exists in time and is created when an executable file is open and, hence, is an active entity
3	Program is static	Process is dynamic, i.e. it changes
4	Program remains in secondary storage until it is deleted	Process exists for a limited period and is terminated upon the completion of the task
5	Program does not have a control block	Process has a control block
6	Program does not have any resource requirement	Process has a high resource requirement. It needs resources like CPU, memory address, I/O, etc.

Now, let us switch back to our baking example. As we have seen, you were switching between tasks when your mother came with a burnt finger. You had to note down where you were in the baking process, attend to your mother and look into first aid instructions. i.e., you were switching between stages of baking known as states to first aid and then back to baking. Similar to states in the baking process, processes also have states.

5.1.2 Process State Diagram (PSD)

Process states are the various states that pass through a process from its creation to completion; that is, each state represents the current activity of that process.

Let us take a real-world example to understand states. Suppose you are travelling from Calicut to Kollam. You boarded the bus, and then after the bus driver started the engine, the bus started travelling. Suppose one passenger had to get down in between; the conductor rings the bell, and the bus stops for the passenger to get down. Suppose at another stop; few passengers want to get into the bus. Then, the bus stops and waits for the passengers to board the bus. Finally, when the bus reaches Kollam, the journey halts. Here, we can see that for the travel management team, the process of travel was "created" when the driver started the journey from Calicut. At this stage, a route name is allocated for it. Later, the vehicle, driver and other required resources will be allocated to it – it is in a "ready" state now. When it is time for the trip to start, the driver picks up the vehicle – "run" state. During the journey, the driver stops the vehicle at various locations, waiting for commuters in a "wait" state. Finally, when the bus arrives at Kollam, the state changes to "terminated". When we give different states for a trip like this, vehicle management can easily track the entire vehicle's activity during a trip.

Similarly, as a process executes, it changes its state. A process has a minimum of five states.

The states of a process are:

- New
- Ready

- Running
- Waiting
- Terminated

If we compare the states of a process with the stages through which the bus has passed while travelling from Calicut to Kollam, the new state of the process is similar to the stage where you entered the bus. The ready state is similar to the engine starting stage of the bus. The running state of a process is similar to the bus moving stage. The waiting state is similar to the bus waiting for passengers to board on stage. The termination state is similar to the bus halting stage.

The Process State Diagram, as shown in Fig.5.1.2, shows the different states and transitions between states a process makes during its execution.

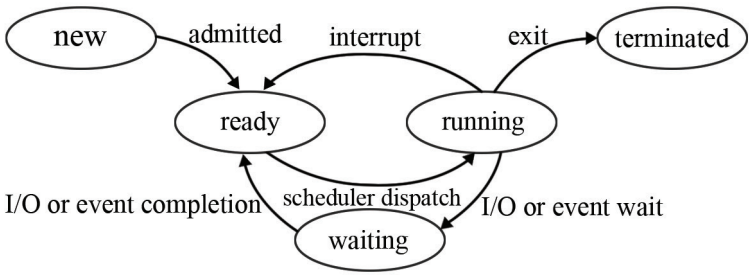


Fig. 5.1.2 Process State Diagram

As you can see from Figure 5.1.2, the process state includes:

- **New (Create)** - The state in which a process is created. No resources will be allocated for the process. Just the PCB for the process will be created. It is still in secondary memory.

- **Ready** - In this state, the process is ready to execute. It is loaded into the main memory and will be allocated all the required resources, except the CPU.

- **Running** - One of the processes in the ready state will be allocated CPU. Then, its state changes to a running state. In this state, the process execution begins.

- **Waiting(Blocked)** - During the execution of a process, some data may be gathered from the user using a keyboard, or some data may be printed. Since input-output activities are slower compared to the CPU, the CPU will have to wait till the input-output activity is over. During this stage, the state of the process will be waiting. After the input/output operation the process goes into the ready state.

- **Terminated (Completed)** - Indicates that the process has finished execution. If the process is currently executing, it is running. The state refers to any of the 5 states described above (new/ ready / running/waiting/terminated). Two additional states can also inferred from the process states :

- **Suspended ready** - when a process with high priority enters, and there is

no space in the main memory. The process with low priority is suspended from the ready state to the suspended ready state and moved to secondary memory. The process is brought to its ready state when the main memory is accessible.

- **Suspended wait** - when a process with high priority enters, and there is no space in the main memory, then the process with low priority is suspended from the wait state to the suspended wait state and moved to secondary memory. The process is brought to its ready state when the main memory is accessible.

These names of the states vary across operating systems. The states that they represent are found on all systems. Only one process can be running on any processor at any instant. Many processes may be ready and waiting, however.

Transition states: Simply stated, a transition is just a name given for the change from one state to another. For example, the change of state of the process from new to ready is named as "Admitted".

A transition is the immediate transfer from one state to another.

Following are the transitions between states

- Admitted: From new to ready
- Scheduler Dispatch: From ready to running
- I/O wait: From running to waiting
- I/O completion: From waiting to ready
- Interrupt: From running to ready
- Exit: From running to terminate

Note:

When your mother's finger is burnt, you stop the baking process temporarily and came to attend to your mother with first aid. Similar is the function of the transition named as interrupt. During interrupt, normal execution will be suspended temporarily, and the interrupt service will be handled by the processor.

Taking the example of a bus trip from Calicut to Kollam, we understand that when we give different states for the trip (as done there earlier), the vehicle management can easily track the entire activity of the vehicle during a trip. Now, you may note that there has to be a facility to record the states of each trip so that the progress of each trip can be tracked. Hence, the vehicle management takes note of information such as the state of the vehicle trip, its bus route locations, the name of the driver and the conductor, the expenses incurred during the travel, and the income generated by bus fares given by passengers. Now, it will be easy for the management team to keep track of different

buses, their bus routes, their income generation and the expenses incurred.

Similarly, OS makes use of a data structure called a Process Control Block (PCB) to store all the details about each process.

Let us now discuss the Process Control Block (PCB).

5.1.3 Process Control Block (PCB)

When a process is created by the operating system, it creates a data structure to store information of that process called Process Control Block (PCB). It is also called a task control block. Figure 5.1.3 shows the sketch of a simplified Process Control Block (PCB).

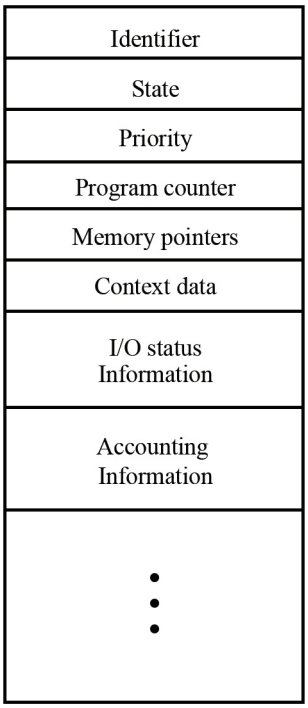


Fig. 5.1.3 Simplified Process Control Block

The exact number of data items captured in a PCB varies from OS to OS. In Figure 5.1.3, a few of the most common fields of a PCB are outlined. They are listed below:

- Identifier
- State
- Priority
- Program counter
- Memory pointers
- Context data
- I/O status information

- Accounting information

Identifier (Process ID): Identifier is the unique ID assigned to each process when it is created; that unique ID distinguishes each process from many running processes in the memory.

State: This field is used to store the state of a process, i.e., New, Ready, Running, Waiting or Terminated.

Priority: This field is used to store the priority of a process. Priority refers to the urgency of doing a task over the other. We can relate to the baking example where you had to prioritize first aid for your mother's burnt finger over cooking. Similarly, there will be a priority for each process. Some processes will have to be executed urgently, even by blocking other processes.

Priority is usually represented by an integer ranging from 0 (lowest priority) to 127 (highest priority). We are not going further deep into that level. In brief, priority depicts the priority level of a process relative to other processes.

Program counter: A program counter field contains the address (location) of the instruction being executed at the current time. As each instruction is fetched, the program counter increases its stored value by 1. Thus, in brief, the address of the next instruction in the program to be executed is stored in the program counter.

Memory pointers: Pointers are used to link one PCB to another PCB. PCB includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes. This helps the operating system hierarchically maintain an easy control flow between parent processes and child processes.

Context data: Earlier, when we considered our baking example, you had to pause from the baking process and attend to your mother with first aid for her burnt finger. This can be considered as a context switching where you are noting down where you were in the baking process, attending to your mother with first aid, and then resuming from the point where you paused in baking. This is similar to the context switch that takes place between processes.

The context data field contains data that are present in registers in the processor while the process is executing.

I/O status information: The I/O status information field includes outstanding I/O requests, I/O devices (e.g., disk drivers) assigned to this process, and a list of files used by the process.

Accounting information: We can relate accounting information to our bus journey from Calicut to Kollam, where the vehicle management takes note of information such as the amount of petrol consumed, the distance travelled, the expense incurred during the travel, and the income generated by bus fares given by passengers.

The Accounting information field includes the amount of processor time and clock time used, time limits, and account numbers.

5.1.4 Operations on Processes

Before we proceed further, let us understand the practical applications of basic operations in the process.

Whenever we are double-clicking an icon in the computer, a program is executed. As we discussed earlier, a program becomes a process when an executable file is loaded into memory. When this happens, the process of creation takes place. When the file is closed, the process is terminated.

The basic operations of a process are as follows:

- Process creation
- Process termination

5.1.4.1 Process Creation

A process is created in the following circumstances:

1. System initialisation - (booting)
2. When a process is to be created within a Process (this often by a programmer)
3. Users request to create a new process (e.g., when we double click on the Chrome browser icon on our desktop, a new process that runs the Chrome Browser is created by the OS).
4. Initiation of a batch job

When the OS is booted, several processes are created. Some of the created processes are foreground processes, i.e. processes that interact with users (humans) and perform work for them.

> Processes that stay in the background to perform some activity, such as web pages, printing, and so on, are known as background processes or daemons.

A process can create another process. As we mentioned earlier, the programmer will include the code for process creation in the program. A new process can be created using the fork( ) system call. A process may create several new processes. The processes that are being created are called as children of that process, while the original creating process is called the parent process. When these processes in turn create other processes, a tree of processes is formed. As discussed earlier, a process is identified by a process identifier (pid). The parent and children share all resources allocated to the parent process.

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

When the fork() system call is invoked, an exact copy of the parent process is created in the Random Access memory (RAM). This is illustrated in the following Figure 5.1.4.

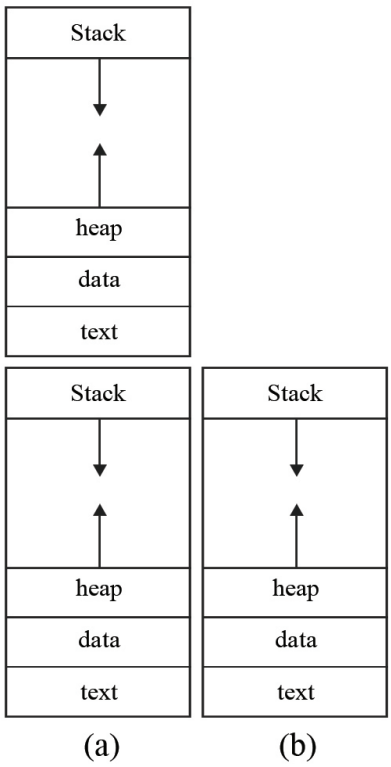


Fig. 5.1.4 (a) single process and (b) process replication during fork () system call

As you may see in Figure 5.1.4, all the sections of a process are duplicated when the fork() system call is invoked. The cloned process will be loaded with a new program.

5.1.4.2 Process Termination

You know that nothing lasts forever, even in the process. Usually, the process terminates due to one of the following conditions:

1. **Normal exit** - The usual scenario in which a process terminates is when their work is completed. For example, when a work is completed by a compiler, it executes a system call to the OS to tell it that it has completed its task.

2. **Error exit** - Here, the reason for process termination is a program bug or error.

3. **Fatal error** - When a fatal error is discovered, say divide by zero is encountered while programming, the process is suspended.

4. **Killed by another process** - When one process executes a system call telling the OS to kill some other process. The exit() system call is used by most operating systems for process termination

5. **Inter Process Communication (IPC)**

6. Unavailability of the required memory

7. Exceed in the execution time limit

8. **Request from parent process -** Parent process has the right to terminate any of its child processes at any time during their execution.

9. **Parent process termination -** When a parent process terminates, it causes all its child processes to stop their execution.

We learnt how a process is created as well as how a process is terminated. Now, we have to understand how the processes communicate with each other. Inter-Process communication will let you understand this in more detail.

When it comes to an individual, he can be an independent person or a person who cooperates with everyone. When it comes to being an independent person, his interactions with others are limited. However, when it comes to a person who interacts with others, he might use various modes of communication.

We all communicate with each other. We might be communicating with our parents, friends, and relatives. Suppose we are living as a joint family, then our communication is easy. We just have to talk to each other. Suppose we want to talk to a relative or a friend who is living far away, then our communication could be either through voice call, video call, text message or email.

Before we go into the details of IPC, let us understand the different types of processes. A process can be of two types:

- Independent process
- Co-operating process

We can compare the independent process with the independent person and the cooperating person with the co-operating process, respectively.

In the case of an independent process, the execution of other processes does not affect another process, but with cooperating processes, it can be affected by other executing processes. However, with cooperating processes, there will be better computational speed and convenience. In summary, if a process's execution affects or is affected by the execution of another process, then the processes are cooperative; otherwise, they are independent processes.

IPC is a mechanism that allows processes to communicate with each other and synchronize their actions.

The purposes of IPC are

- Data transfer
- Sharing data

- Event notification
- Resource sharing
- Synchronization
- Process control

Now, you might be wondering how the processes are communicating with each other. Processes can communicate with each other through the following approaches.

1. Shared memory
2. Message passing

Shared Memory

In the shared memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

Message passing

In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.

In the Message passing method, read and write operations among multiple processes take place in the message queue without being connected to each other. These messages are saved and stored inside the queue until the recipient retrieves them. Most of the OS uses message passing, which is quite useful for IPC.

An example of a shared memory method is the producer-consumer problem. We shall discuss the producer-consumer problem shortly.

5.1.5 Interprocess Communication Methods

As mentioned before, interprocess communication (IPC) is a set of interfaces that are usually programmed in order for the programs to communicate between a series of processes. The advantage is that this method allows the concurrent running of programs in an operating system.

Following are the methods in IPC.

1. Pipes (Same Process)
2. Named Pipes (Different Processes)
3. Message Queuing
4. Semaphores
5. Shared memory
6. Sockets

Pipe is a communication medium between two or more related or interrelated processes. Communication is achieved by one process of writing into the pipe and another by reading from the pipe.

5.1.5.1 Pipes (Same Process)

Pipe mechanism can be viewed with a real-time scenario such as filling the water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe, and the reading process is nothing more than retrieving from the pipe.

Pipes allow flow of data in one direction only Eg: Keyboard.

Figure 5.1. 5 shows the pipe within one process.

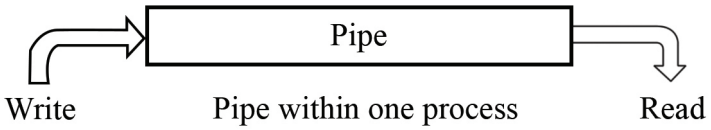


Fig. 5.1.5 Pipe within one process

5.1.5.2 Named Pipes (Different Processes)

Named pipe is used for two-way communication (For example, communication between the server and the client, plus the client and the server at the same time). Named Pipe supports bi-directional communication. Another name for the pipe is FIFO (First-In-First-Out).

Unlike a regular pipe, a named pipe can be used by processes that do not have to share a common process origin, and the message sent to the named pipe can be read by any authorised process that knows the name of the named pipe.

5.1.5.3 Message Queuing

This allows messages to be passed between processes using either a single queue or several message queues.

A real time example of messages could be mailboxes where the sender's mail and receiver's mail are stored. The sender and the receiver do not need to interact with each other at the same time.

In the message queue, the communication pattern is such that the sending and receiving party do not need to interact with the message queue at the same time. Messages placed in the queue are stored until the recipient retrieves them.

5.1.5.4 Shared memory

This allows the interchange of data through a defined area of memory. Semaphore values have to be obtained before data can get access to shared memory.

5.1.5.5 Sockets

This method is mostly used to communicate over a network between a client and a server. It allows for a standard connection that is computer- and OS-independent.

5.1.6 Interprocess Communication Issues

Processes need to communicate with each other, and this communication must occur in a structured way. We already discussed interprocess communication in the earlier section.

In the following section, we will discuss some of the issues related to interprocess communication. There are basically three issues:

- How can one process pass information to another?
- Making sure that the processes don't get into each other's way during critical section activity.
- Checking for proper sequencing when dependencies are present.

Before we discuss these issues, let us introduce the following terms:

1. Race Conditions
2. Critical sections
3. Mutual exclusion

5.1.6.1 Race Condition

A race condition in an operating system (OS) occurs when multiple processes or threads access shared resources simultaneously, resulting in unpredictable and undesirable outcomes. This can happen because the timing and order of execution of the threads are not deterministic. A race condition occurs when the system's behaviour depends on the sequence or timing of uncontrollable events, such as task scheduling. Here's a real-world example involving a banking system:

Two users, Alice and Bob, have access to a joint bank account with a balance of ₹1000. They both initiate a withdrawal of ₹500 at almost the same time from different ATMs.

Steps:

1. Initial Balance: The account balance is ₹1000.

2. Alice's Withdrawal:

- Alice's transaction is processed, and the system reads the balance as ₹1000.
- Alice requests to withdraw ₹500.
- The system calculates the new balance as ₹1000 - ₹500 = 500.

3. Bob's Withdrawal:

- Before Alice's transaction completes, Bob's transaction is processed.
- The system reads the balance as ₹1000 (since Alice's transaction hasn't been completed yet).
- Bob requests to withdraw ₹500.
- The system calculates the new balance as ₹1000 - ₹500 = ₹500.

4. Updating Balance:

- Alice's transaction was completed, and updates the balance to ₹500.
- Bob's transaction also completes and updates the balance to ₹500.

Result:

Both withdrawals are successful, and the final balance is incorrectly updated to ₹500 instead of ₹0.

The race condition occurs because both transactions read the initial balance of ₹1000 before either transaction can update the balance. Due to the lack of synchronization, both transactions result in an incorrect final balance. Proper synchronization mechanisms, such as locks, should be used to ensure that the account balance is accurately updated in such concurrent operations.

The above example is a good illustration of race conditions.

Situations where two or more processes are reading or writing some shared data, and the final result depends on who runs precisely when are called race conditions.

5.1.6.2 Critical section problem

Now, you might be doubting your mind. How can we avoid race conditions? The main method to avoid race conditions is by prohibiting more than one process from reading or writing in the shared memory at the same time. We are thus coming into the concept of mutual exclusion.

Mutual exclusion is some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

In some situations, a process might be accessing a shared memory or file.

That part of the program where the shared memory is accessed is called the critical section or critical region.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion
2. Progress
3. Bounded waiting

Mutual exclusion

The mutual exclusion principle is that if process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

Progress

Suppose no process is executing in its critical section, and some processes wish to enter their critical sections. In that case, only those processes that are not executing in their remaining sections can participate in the decision, which will enter its critical section next. This selection cannot be postponed indefinitely.

Bounded waiting

Bounded waiting is a condition in the critical section problem in operating systems that limits the number of times other processes can enter and leave their critical sections after a process has requested to enter. This ensures that each process will only have to let another process go first once before it's their turn again.

Recap

- A process is a program in execution.
- A process includes process stack, heap, data, text, and heap segment. Process state diagram shows the different states of transition
- Program Control Block (PCB) is used to represent a process in the operating system.
- Four scenarios in which a process is created are:
  - System initialization - (booting)
  - Execution of process creation system call
  - User request to create a new process
  - Initiation of a batch job
- Processes that stay in the background to perform some activity, such as web pages, printing, and so on, are known as daemons.
- Process terminates due to one of the following conditions:
  - Normal exit – voluntary

- Error exit – voluntary
- Fatal error - involuntary
- Killed by another process - involuntary

- A process is a program in execution.
- A process includes process stack, heap, data, text, and heap segment. Process state diagram shows the different states of transition
- Program Control Block (PCB) is used to represent a process in the operating system.
- Four scenarios in which a process is created are:
  - System initialization - (booting)
  - Execution of process creation system call
  - User request to create a new process
  - Initiation of a batch job
- Processes that stay in the background to perform some activity, such as web pages, printing, and so on, are known as daemons.
- Process terminates due to one of the following conditions:
  - Normal exit – voluntary
  - Error exit – voluntary
  - Fatal error - involuntary
  - Killed by another process - involuntary
- A process can be of two types:
  - Independent process.
  - Co-operating process.
- independent process - the execution of other processes does not affect another process co-operating process. The execution of other processes can be affected by other executing processes
- Processes can communicate with each other through the following approaches.
  - Shared memory
  - Message passing

- Shared memory - Processes can then exchange information by reading and writing data to the shared region.
- Message passing - communication takes place by means of messages exchanged between the cooperating processes.
- Inter-Process communication is the method of how the processes communicate between each other.
- Purposes of IPC are
    - Data transfer
    - Sharing data
    - Event notification
    - Resource sharing
    - Synchronization
    - Process control
- Following are the methods in IPC.
    1. Pipes
    2. Named Pipes
    3. Message Queuing
    4. Semaphores
    5. Shared memory
    6. Sockets
- Race condition is a situation where- The final output produced depends on the execution order of instructions of different processes.
- Several processes compete with each other.
- Mutual exclusion is some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.
    - That part of the program where the shared memory is accessed is called the critical section.
- Another name for the critical section is critical region.
- Process synchronisation is a method for avoiding inconsistent results production while sharing the same resources by multiple resources

Objective Type Questions

1. What is a program in execution?
2. Which part of a process contains temporary data?
3. Which component contains global variables?
4. Which is the dynamically allocated memory of a process during runtime?
5. Which component shows the different states and transitions a process makes during its execution?
6. What is the state in which a process is created?
7. What indicates that instructions are being executed?
8. What indicates that the process is waiting for some event to occur?
9. What is the state where a process is waiting to be assigned a new process or to a processor?
10. What indicates that the process has finished execution?
11. What is used to represent a process in an operating system?
12. What is another name for a process control block?
13. What part of a process helps to distinguish a process from other processes?
14. Which component contains the address of the instruction being executed at the current time?
15. What is used to link one PCB to another?
16. Where is context data present while the process is executing?
17. Which includes the amount of processor time and clock time used, time limits, account numbers?
18. What are the basic operations of a process?
19. How are the processes that stay in the background to perform some activity such as web pages, printing, and so on are known?
20. What is the name of the exact replica of the parent created when the fork() system call is invoked?

21. In the case of a thread, which segment is replicated?

22. Which system call is used by most operating systems for process termination?

23. In order for a process to create a new process during the course of execution, which system call is used?

24. How is a process classified?

25. In which case does the execution of other processes not affect another process?

26. With what the execution of other processes can be affected by other executing processes?

27. What are the advantages of the cooperating process?

28. How can processes communicate with each?

29. In which model, a region of memory that is shared by cooperating processes is established?

30. In which model communication takes place by means of messages exchanged between the cooperating processes?

31. Which is a set of interfaces which is usually programmed in order for the programs to communicate between a series of processes?

32. Which allows the concurrent running of programs in an operating system?

33. Which allows the flow of data in one direction only?

34. Which is used for two-way communication in IPC?

35. Which allows the interchange of data through a defined area of memory?

36. What is used to communicate over a network between a client and a server?

37. What are the situations where two or more processes are reading or writing some shared data, and the final result depends on who runs precisely when called.

38. What is a way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing?

39. What is the part of the program where the shared memory is accessed?

Answers to Objective Type Questions

1. Process
2. Process stack
3. Data section
4. Heap
5. Process state diagram
6. New
7. Running state
8. Waiting state
9. Ready
10. Terminated state
11. Process control block
12. Task control block.
13. Unique identifier
14. Program counter
15. Memory pointers
16. Registers
17. Accounting information
18. Process creation, Process termination
19. Daemons
20. Child
21. Stack
22. exit()
23. fork()

24. Independent process, Co-operating process.

25. Independent process

26. Cooperating process

27. Computational speed, convenience

28. Shared memory, Message passing

29. Shared memory

30. Message passing

31. Interprocess communication (IPC)

32. Interprocess communication

33. Pipes

34. Named pipe

35. Shared memory

36. Sockets

37. Race condition.

38. Mutual exclusion

39. Critical section / critical region.

Assignments

1. Explain the process state diagram.
2. Explain the process control block.
3. Explain different interprocess communications.
4. Describe race conditions with an example.

Suggested Reading

1. William Stallings – Operating Systems, Sixth Edition, Prentice Hall of India, Pearson

2. Abraham Silberschatz, Peter Galvin and Greg Gagne – Operating System Principles, Seventh edition, John Wiley.



Introduction to Process Synchronization

Learning Outcomes

By the completion of the unit, the learner will be able to:

- understand the need for process synchronization
- identify the types of processes based on synchronization.
- comprehend the concept of race conditions.
- explain Critical Section Problem and the available solution

Prerequisites

We have already seen the situation of race condition and the problems related to it. A race condition is a type of concurrency problem that occurs when the behaviour of a software system depends on the relative timing of events, such as the sequence and timing of threads or processes that share resources. In simpler terms, it happens when two or more threads or processes access shared data simultaneously, and the final outcome depends on the order of execution, which can lead to unpredictable and erroneous results.

To prevent race conditions, it is crucial to control access to shared resources and ensure that only one thread or process can manipulate the shared data at a time. Let's find a solution to the previous banking transaction problem where Alice and Bob withdraw money simultaneously, resulting in incorrect results.

Here's a solution using a lock to synchronize access to the joint account balance:

Initial Balance: The account balance is ₹1000.

Alice's Withdrawal:

Alice's transaction requests a lock on the account balance.

The lock is granted, and Alice's transaction reads the balance as ₹1000.

Alice requests to withdraw ₹500.

The system calculates the new balance as ₹1000 - ₹500 = ₹500.

Alice's transaction updates the balance to ₹500.

Alice's transaction releases the lock.

Bob's Withdrawal:

Bob's transaction requests a lock on the account balance.

The lock is granted after Alice's transaction is completed.

Bob's transaction reads the balance as ₹500.

Bob requests to withdraw ₹500.

The system calculates the new balance as ₹500 - ₹500 = ₹0.

Bob's transaction updates the balance to ₹0.

Bob's transaction releases the lock.

Both withdrawals are successful, and the final balance is correctly updated to ₹0.

Similarly, the operating system provides several process synchronization mechanisms to avoid errors related to the concurrent simultaneous access of shared resources by many processes. We are going to discuss all those mechanisms in this section.

Keyterms

Process synchronization, independent processes, co-operating processes, Producer-consumer problem, Peterson's Solution, Dining Philosophers Problem, The Readers Writers Problem

Discussion

5.2.1 Introduction to process synchronization

Processes Synchronization or Synchronization is the way by which processes that share the same memory space are managed in an operating system. It helps maintain the consistency of data by using variables or hardware so that only one process can make changes to the shared memory at a time. There are various solutions for this, such as semaphores, mutex locks, synchronization hardware, etc.

Process Synchronization was introduced to handle problems that arose during multiple process executions. Process is categorized into two types on the basis of synchronization, and these are given below:

- Independent Process

- Cooperative Process

5.2.1.1 Independent Processes

Two processes are said to be independent if the execution of one process does not affect the execution of another process. They do not share data or resources with other processes. Each process operates independently, with its own memory space and state, without any need for synchronization or communication with other processes.

5.2.1.2 Cooperative Processes

Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronised so that the order of execution can be guaranteed.

5.2.2 Process synchronization

As you have seen with race conditions, when multiple processes share the same resources, inconsistency will occur. Process synchronization is a method for avoiding inconsistent results production while sharing the same resources by multiple resources. Process synchronization controls the execution of processes running concurrently to ensure consistent results are produced. Synchronization is classified into two. They are:

1. Competition synchronization
2. Cooperation Synchronization

5.2.2.1 Competition Synchronization

Competitive synchronization is required between two processes when both require the use of the same resources that cannot be simultaneously used.

5.2.2.2 Cooperation Synchronization

Cooperation synchronization is required between two processes when the second process must wait for the first process's tasks to finish execution before it may proceed.

Cooperation synchronization can be compared to a real-world example of building a house. In the case of building a house, the structure of the house needs to be completed before the tile work can start.

In the same way, as with the above example, P2 needs to wait for P1 to complete execution.

Note:

Lack of synchronization may lead to:

1. Inconsistency
2. Loss of data
3. Deadlock

5.2.3 Critical Section Problem

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes. The entry to the critical section is mainly handled by the wait( ) function, while the exit from the critical section is controlled by the signal( ) function.

```
do
  {
        // Entry section
              Critical section
        //Exit section
              Remainder section
  }while(TRUE);
```

5.2.3.1 Entry Section

In this section, the process requests for its entry are mainly in the Critical Section.

Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion:**

If a process is executing in its critical section, then no other process is allowed to execute in the critical section.

- **Progress:**

If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter the critical section next. The selection cannot be postponed indefinitely.

- **Bounded Waiting:**

A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

5.2.3.2 Exit Section

This section is followed by the critical section. The Exit section enables other processes waiting in the Entry section to enter the Critical section.

5.2.4 The solution to the Critical Section Problem

The critical section plays an important role in Process Synchronization, so the problem must be solved. Some widely used methods to solve the critical section problem are as follows:

5.2.4.1 Peterson's Solution

Peterson's Solution is a classical software-based solution to the critical section problem. In Peterson's solution, we have two shared variables:

- boolean flag[i]: Initialized to FALSE; initially no one is interested in entering the critical section
- int turn: The process whose turn is to enter the critical section.

```
do
{
        flag[i] = TRUE;
        turn = j;
        while(flag[j] && turn = = j);
        //Critical section
        flag [i] == FALSE;
        // remainder section
}while(TRUE);
```

Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's solution

- It involves busy waiting. (In Peterson's solution, the code statement- "while(flag[j] && turn == j);" is responsible for this. Busy waiting is not favoured because it wastes CPU cycles that could be used to perform other tasks.)
- It is limited to 2 processes.
- Peterson's solution cannot be used in modern CPU architectures.

```
do
{
        flag[i] = TRUE;
        turn = j;
        while(flag[j] && turn = = j);
        //Critical section
        flag [i] == FALSE;
        // remainder section
}while(TRUE);
```

The above shows the structure of the process Pi in Peterson's solution.

- Suppose there are N processes (P1, P2, ... PN), and at some point in time, every process required to enter the Critical Section
- A FLAG[ ] array of size N is maintained here, which is false by default. Whenever a process requires an entry in the critical section, it has to set its flag as true. Example: If Pi wants to enter, it will set FLAG[i]=TRUE.
- Another variable is called TURN and is used to indicate the process number that is currently waiting to enter into the critical section.
- The process that enters into the critical section while exiting would change the TURN to another number from the list of processes that are ready.
- Example: If the turn is 3 then P3 enters the Critical section, and while exiting turn=4 and therefore P4 breaks out of the wait loop.

5.2.4.2 Synchronization Hardware

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disable interrupts to occur while a shared variable or resource is being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling interrupts in a multiprocessor environment can be time-consuming as the message is passed to all the processors. This message transmission lag delays the entry of threads into the critical section, and the system efficiency decreases.

5.2.4.3 Mutex Locks

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. Mutex is a mutual exclusion

object that synchronizes access to resources. It is created with a unique name at the start of a program. The mutex locking mechanism ensures only one thread can acquire the mutex and enter the critical section. This thread only releases the mutex when it exits in the critical section, as shown in Fig 5.2.1

In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside the critical section, and in the exit section, that LOCK is released. As the resource is locked while a process executes its critical section, no other process can access it.

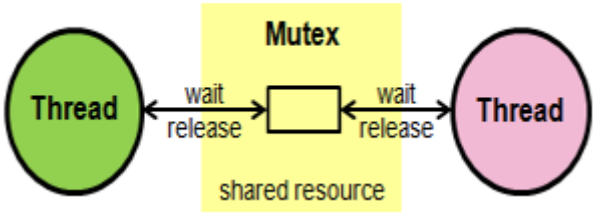


Fig 5.2.1 Mutex Lock

5.2.4.4 Semaphores

A semaphore is a signalling mechanism, and a thread that is waiting on a semaphore can be signalled by another thread. This is different from a mutex as the mutex can be signalled only by the thread that is called the wait function. A Semaphore is simply a regular integer. It cannot be negative, with its minimum value being zero (0). The maximum value of a Semaphore can be any number. It uses two atomic operations, wait and signal, for process synchronization. A Semaphore is an integer variable which can be accessed only through two operations: wait( ) and signal( ). Fig 5.2.2 shows the working of the semaphore. There are two types of semaphores: Binary Semaphores and Counting Semaphores.

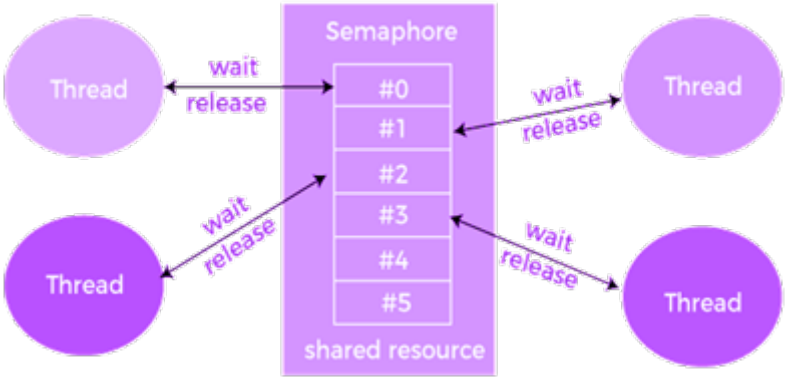


Fig 5.2.2 Semaphore

Atomic operations are those operations that cannot be interrupted. An operation acting on shared memory is atomic if it completes in a single step relative to other threads.

Definition of wait and signal processing are:

Wait:

Wait operation decrements the value of its argument S by 1, if it is positive. If it is

negative or zero, no operation is performed.

```
wait( )
    {
        while (S<=0);
        S- -;
}
```

Signal :

Signal operation increments the value of argument S.

```
signal( )
    {
      S+ +;
     }
```

Here, if the Semaphore value is greater than zero (positive), the process can enter the critical section. If the Semaphore value is zero, the process must wait until another process exits the critical section. The P function or Wait operation is only relevant until a process enters the critical section. Once the process enters, the P function or Wait operation is no longer needed. When a process exits the critical section, the Semaphore value should be decreased.

5.2.5 Classical Problems of Synchronization

1. Producer Consumer Problem
2. Readers and Writers Problem
3. Dining Philosophers Problem

5.2.5.1 Producer Consumer Problem

Producer Consumer Problem is a classical synchronization problem.

Before we discuss producer-consumer problems, let us introduce the term ‘buffer’.

A buffer is a region of physical memory storage that is used to temporarily store data while it is being moved from one place to another. A buffer may be used to move data between processes within a computer.

In the case of producer-consumer problems, a producer produces an item and stores it in the fixed-size buffer. The consumer consumes the item stored in the fixed-size buffer. The condition for synchronization is such that the producer should not produce an item while the consumer is consuming an item from the buffer and vice versa. Hence, at a time, either the producer or the consumer should access the buffer. Semaphores are used

to resolve the producer consumer problem. Semaphores are integer variables used to avoid critical section problems.

5.2.5.2 Readers and Writers Problem

Readers Writers problem can be related to objects such as files that are shared between multiple processes. Some of these processes are readers, i.e. they only want to read the data from the object, and some of these processes are writers i.e. they want to write into the object. However, multiple readers may read the data simultaneously i.e. concurrently. Only one writer can write the data at any time i.e. no reader should be present. A reader and a writer cannot access data simultaneously.

Solution is when a writer writes, it has exclusive access to the data, i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time, i.e., if a reader is in the critical section, any number of readers could enter, irrespective of whether or not the writer is waiting to enter the critical section. This can be implemented using semaphores.

5.2.5.3 Dining Philosophers Problem

Dining Philosophers is a classical synchronization problem. It states that there are five philosophers sitting around a circular table. Their job is to think or eat alternatively. There is a bowl of noodles placed at the center of the table along with five chopsticks. Figure 5.1.6 shows five Philosophers sitting around the table with their plates and chopsticks. In order for a philosopher to eat, he requires both left and right chopsticks to be available at the same time. In case if the immediate left and right chopsticks are not available at the same time, the philosopher puts down their chopstick and starts thinking again.

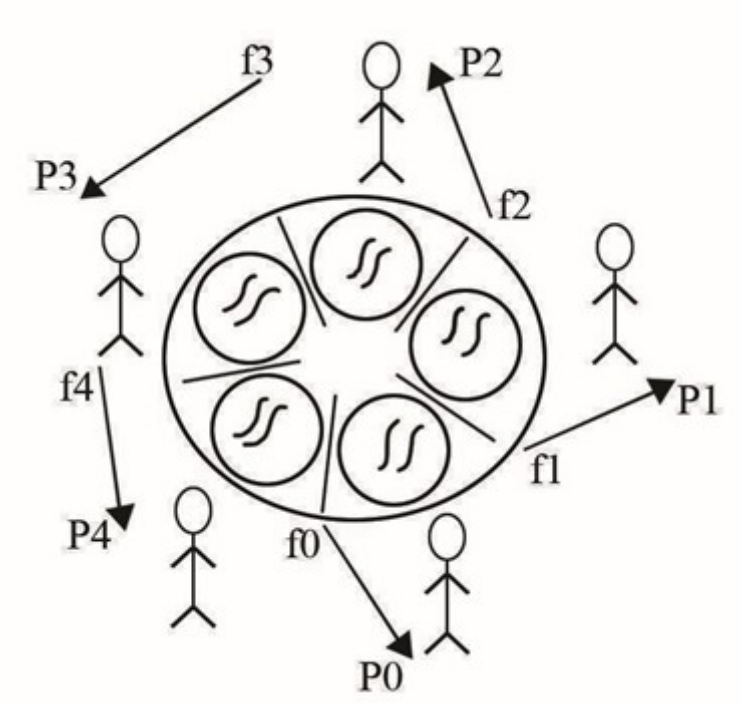


Fig. 5.2.3 Five Philosophers sitting around the table with plates and chopsticks

Fig. 5.2.3 Five Philosophers sitting around the table named as P0, P1, P2, P3 and P4 and Chopsticks named as f0, f1, f2, f3 and f4.

Initially, all the philosophers take one chopstick in the Left to Right direction, i.e. P0 takes f0, P1 takes f1, P2 takes f2, P3 takes f3, and P4 takes f4. But in order for any philosopher to eat, say P0, requires f0 and f1. But f1 is already occupied by P1. The same is the situation for all the processes. i.e. a resource required by one process is already occupied by another process. In another process, P1 requires a resource, say f2, that is already occupied by another process, P2 and so on; finally, P4 requires f0, which is already occupied by P0, thus resulting in circular wait and hold and wait. Such a condition is known as deadlock.

A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does.

One way to avoid the deadlock in the Dining philosopher's problem is for all the philosophers except the last philosopher to take the chopsticks from Left to Right and the last philosopher to take the chopsticks from Right to Left. Another solution is for all the even philosophers to pick the chopsticks from Left to right and the odd philosophers to pick the chopsticks from Right to Left. These solutions can be implemented using Semaphores.

Recap

- Synchronization is the effort of executing processes such that no two processes have access to the same shared data.
- Four elements of program/data are:
  - Entry section
  - Critical section
  - Exit section
  - Reminder section
- The critical section is a portion of code that a single process can access at a specific moment in time.
- Three essential rules that any critical section solution must follow are as follows:
  - Mutual Exclusion
  - Progress
  - No Starvation (Bounded waiting)

- Solutions to critical section problems are:
  - Peterson's solution
  - Synchronization hardware
  - Mutex Locks
  - Semaphore

Objective Type Questions

1. What is it called when several processes access the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place?
2. If a process is executing in its critical section, then no other processes can be executing in their critical section. What is this condition called?
3. Mutual exclusion can be provided by _____
4. Process synchronisation can be done on _____.
5. Concurrent access to shared data may result in _____.
6. A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which access takes place is called _____.
7. What does it imply if a process is executing in its critical section, then no other process must be executing in its critical sections?

Answers to Objective Type Questions

1. Race condition
2. Mutual exclusion
3. Both mutex locks and binary semaphores
4. Both hardware and software level
5. Data inconsistency
6. Race condition
7. Mutual exclusion

Assignments

1. Describe the process state diagram.
2. Explain the process control block.
3. Discuss different interprocess communications.
4. Illustrate various Classical problems of synchronisation.

Suggested Reading

1. Silberschatz, Galvin, and Gagne. Operating System Concepts, 7th Edition, Wiley, 2005. ISBN 0-471-69466-5
2. Crowley, Charles. Operating systems: a design-oriented approach. McGraw-Hill Professional, 1996.
3. Harris, J. Archer Archer, and John Cordani. Schaum's outline of operating systems. McGraw-Hill, Inc., 2001.
4. Andrew, S. Tanenbaum. "Modern operating systems Pearson." (2014).



UNIT 3 Semaphores and Monitors

Learning Outcomes

By the completion of the unit, the learner will be able to:

- introduce the solutions to critical section problems using semaphores
- present binary and counting semaphores
- make aware of monitors as a tool for solving critical section problems
- examine the solution of dining philosophers, and readers-writers problem using semaphore and monitor

Prerequisites

We've already seen that processes can execute concurrently or in parallel. We have seen the role of process scheduling and described how the CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process.

Consider the code for the producer process:

```
while (true)
   {
     /* produce an item in next produced */
    while (counter == BUFFER SIZE );
      /* do nothing */
     buffer[in] = next produced;
     in = (in + 1) % BUFFER SIZE ;
     counter++;
   }
```

```
while (true)
   {
        while (counter == 0);
             /* do nothing */
        next consumed = buffer[out];
        out = (out + 1) % BUFFER SIZE ;
        counter--;
          /* consume the item in the next consumed */
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A situation like this, where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. To guard against the race condition above, we need to ensure that only one process at a time can manipulate the variable counter. To make such a guarantee, we require that the processes be synchronised in some way. Semaphores and monitors are used for this.

Keyterms

Semaphores, wait( ),signal( ), monitors, race condition, process synchronization

Discussion

5.3.1 Semaphores

Semaphore is a more robust tool that can provide more sophisticated ways for processes to synchronize their activities. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait( ) and signal( ) .

The wait( ) operation was originally termed P (from the Dutch proberen, “to test”); signal( ) was originally called V (from verhogen, “to increment”). The definition of wait( ) is as follows:

```
wait(S)
{
```

while (S <= 0); // busy wait

S--;

}

The definition of signal( ) is as follows:

signal(S)

{

S++;

}

All modifications to the integer value of the semaphore in the wait() and signal( ) operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.

5.3.1.1 Semaphore usage

Operating systems often distinguish between counting and binary semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead to provide mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized based on the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will be blocked until the count becomes greater than 0. We can also use semaphores to solve various synchronization problems.

For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2 . Suppose we require that S2 be executed only after S1 has been completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch initialized to 0. In process P1, we insert the statements.

S1 ;

signal(synch);

In process P2 , we insert the statements

wait(synch);

S2 ;

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked

signal(synch) , which is after statement S1 has been executed.

5.3.1.2 Bounded Buffer Problem Using Semaphore

In our problem, the producer and consumer processes share the following

data structures:

Buffer Pool: This is a shared resource that can hold a certain number of items (buffers). The size of this pool is n.

Semaphores:

mutex: Ensures mutual exclusion, meaning only one process can access the buffer at a time. Initialized to 1.

empty: Counts the number of empty buffers available. Initialized to n (all buffers are empty at the start).

full: Counts the number of full buffers available. Initialized to 0 (no buffers are full at the start).

Now, consider the variable initialization:

int n;

semaphore mutex = 1;

semaphore empty = n;

semaphore full = 0

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process :

```
do
 {
. . .
/* produce an item in next produced */
. . .
wait(empty);
wait(mutex);
```

. . .

/* add next produced to the buffer */

. . .

signal(mutex);

signal(full);

} while (true);

The producer's job is to produce items and place them in the buffer. Here's the producer's code explained:

1. Produce an item: The producer generates an item to be placed in the buffer.
2. wait(empty): The producer waits until there is at least one empty buffer available.
3. wait(mutex): The producer waits for exclusive access to the buffer to avoid race conditions.
4. Add item to buffer: The produced item is placed in the buffer.
5. signal(mutex): The producer releases exclusive access to the buffer.
6. signal(full): The producer signals that a new buffer is full and available for the consumer.

The code for the consumer process.

do {

wait(full);

wait(mutex);

. . .

/* remove an item from buffer to next consumed */

. . .

signal(mutex);

signal(empty);

. . .

/* consume the item in the next consumed */

. . .

} while (true);

The consumer's job is to consume items from the buffer. The consumer's code is explained here:

1. wait(full): The consumer waits until there is at least one full buffer available.
2. wait(mutex): The consumer waits for exclusive access to the buffer to avoid race conditions.
3. Remove item from buffer: The consumer takes an item from the buffer.
4. signal(mutex): The consumer releases exclusive access to the buffer.
5. signal(empty): The consumer signals that a new buffer is empty and available for the producer.
6. Consume the item: The consumer uses the item.

Note the symmetry between the producer and the consumer.

- Both processes wait for the appropriate semaphore (empty for the producer and full for the consumer) to ensure there is space to work.
- Both processes use mutex to ensure exclusive access to the buffer.
- Both processes signal the other semaphore (full for the producer and empty for the consumer) to update the state of the buffer.

5.3.2 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect since these errors happen only if particular execution sequences take place and these sequences do not always occur.

To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterwards. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we examine the various difficulties that may result. Note that these difficulties will arise even if a single process is not well-behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a process interchanges the order in which the wait( ) and signal( ) operations on the semaphore mutex are executed, resulting in the following execution:

signal(mutex);

...

critical section

...

wait(mutex);

In this situation, several processes may be executed in their critical sections simultaneously, violating the mutual exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

wait(mutex);

...

critical section

...

wait(mutex);

In this case, a deadlock will occur.

- Suppose that a process omits the wait(mutex) , or the signal(mutex) , or both. In this case, either mutual exclusion is violated, or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem.

To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the monitor type.

5.3.2.1 Monitor usage

An abstract data type or ADT encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. A monitor type is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The syntax of a monitor type is shown as follows.

monitor monitor name

{

/* shared variable declarations */

function P1 (. . .)

{

. . .

}

```
function P2 ( . . . )
{
. . .
}
.
.
.
function Pn ( . . . )
{
. . .
}
initialization code ( . . . )
{
}
}
```

The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly. However, the monitor construct, as defined so far, is not sufficiently powerful for modelling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:

```
condition x, y;
```

The only operations that can be invoked on a condition variable are wait( ) and signal( ). The operation x.wait( ); means that the process invoking this operation is suspended until another process invokes x.signal( );

The x.signal( ) operation resumes exactly one suspended process. If no process is suspended, then the signal( ) operation has no effect; that is, the state of x is the same as if the operation had never been executed. Contrast this operation with the signal( ) operation associated with semaphores, which always affects the state of the semaphore. Now suppose that, when the x.signal( ) operation is invoked by a process P, there exists

a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signalling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that conceptually, both processes can continue with their execution. Two possibilities exist:

1. Signal and wait. P either waits until Q leaves the monitor or waits for another condition.

2. Signal and continue. Q either waits until P leaves the monitor or waits for another condition.

There are reasonable arguments in favour of adopting either option. On the one hand, since P was already executing in the monitor, the signal-and-continue method seems more reasonable. On the other hand, if we allow thread P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold. A compromise between these two choices was adopted in the language Concurrent Pascal. When thread P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.

5.3.2.2 Dining-Philosophers Solution Using Monitors

Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

1. **States of Philosophers**: Each philosopher can be in one of three states:
   - **THINKING**: The philosopher is thinking.
   - **HUNGRY**: The philosopher wants to eat.
   - **EATING**: The philosopher is eating.
2. **Conditions**: We use conditions to make philosophers wait when they cannot pick up chopsticks.

Data Structures

- state[5]: Array to keep track of the state of each philosopher.
- self[5]: Condition variables for each philosopher to wait and be signalled.

Monitor Definition

The monitor DiningPhilosophers manages the chopsticks and philosopher states.

```
monitor DiningPhilosophers {
   enum { THINKING, HUNGRY, EATING };
```

```
int state[5];
condition self[5];
// Philosopher i tries to pick up chopsticks
void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING) {
        self[i].wait();
    }
}
// Philosopher i puts down chopsticks
void putdown(int i) {
    state[i] = THINKING;
    test((i + 4) % 5);
    test((i + 1) % 5);
}
// Test if philosopher i can start eating
void test(int i) {
    if (state[(i + 4) % 5] != EATING && state[i] == HUNGRY && state[(i + 1) % 5]
!= EATING) {
        state[i] = EATING;
        self[i].signal();
    }
}
// Initialization code
initialization() {
    for (int i = 0; i < 5; i++) {
        state[i] = THINKING;
    }
```

```
    }
}
```

Explanation

1. pickup(int i): When a philosopher wants to eat, they change their state to HUNGRY and call test(i). If they can't eat immediately, they wait.

2. putdown(int i): When a philosopher i finishes eating, they change their state to THINKING. Then, they check if their neighbours can start eating by calling tests on their left and right neighbours.

3. test(int i): This function checks if the philosopher can start eating. Philosopher i can eat if both their left and right neighbors are not eating, and if i is HUNGRY.

4. initialization(): This function initializes all philosophers' states to THINKING at the beginning.

It is easy to show that this solution ensures that no two neighbours are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death.

Recap

- Mutual exclusion must be provided to ensure that a critical section of code is used by only one process or thread at a time.
- Semaphores can be used to solve various synchronization problems
- Integer variable accessed through atomic operations: wait() and signal().
- **wait()**: Originally termed P; decreases semaphore value if it's positive.
- **signal()**: Originally termed V; increments semaphore value
- Monitors provide a synchronization mechanism for sharing abstract data types
- A condition variable provides a method by which a monitor function can block its execution until it is signaled to continue.
- Use of semaphore variable mutex initialized to 1.
- wait(mutex) before, signal(mutex) after critical section.

- Replacing signal(mutex) with wait(mutex): Causes deadlock.
- Monitor Solution: Ensures no two neighboring philosophers eat simultaneously; prevents deadlock.
- Three states (THINKING, HUNGRY, EATING); self[i] condition allows philosophers to wait when necessary.

Objective Type Questions

1. What is an uninterruptible unit in a process?
2. What type of variable is semaphore?
3. What is an integer variable used to solve the critical section problem?
4. What are the two atomic operations permissible on semaphores?
5. The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0, S2 = 0.

```
Process P0
while(true)
{
   wait(S0);
   print '0';
   release(S1);
   release(S2);
}

Process P1
wait(S1);
release(S0);
Process P2
wait(S2);
release(S0);
```

How many times will P0 print '0'?

6. Each process Pi, i = 0,1,2,3,……,9 is coded as follows.

repeat

P(mutex)

{Critical Section}

V(mutex)

forever

The code for P10 is identical, except that it uses V(mutex) instead of P(mutex). What is the largest number of processes that can be inside the critical section at any moment (the mutex is initialized to 1)?

7. Give an example of a high-level synchronization construct.

8. What is characterized by a set of programmer defined operators?

Answers to Objective Type Questions

1. Atomic
2. Integer
3. Semaphore
4. Wait and signal
5. At least twice
6. 3
7. Monitor
8. Monitor

Assignments

1. Explain the semaphore.
2. Explain the monitor.
3. Find a solution to the Dining Philosophers problem using semaphore.

References

1. William Stallings - Operating Systems, Sixth Edition, Prentice Hall of India, Pearson

2. Milan Kovic - Operating Systems, Second Edition, TMH .

Suggested Reading

1. Abraham Silberschatz, Peter Galvin and Greg Gagne - Operating System Principles, Seventh Edition, John Wiley .



Deadlock in Operating System

Learning Outcomes

By the completion of this unit, the learner will be able to:

- know about the strategy for resource utilization.
- understand the description of deadlocks and the necessary conditions for occurring.
- present a number of different methods for preventing or avoiding
- deadlocks in a computer system

Prerequisites

Imagine you're the manager of a busy restaurant. Your main job is to ensure that all your customers have a pleasant dining experience, which involves efficiently managing both your kitchen staff and the resources they need to prepare meals.

For that, you need to allocate kitchen equipment like stoves and ovens, manage recipes and an order system. You have to ensure that non-shareable resources like the oven are used by one chef at a time, while shareable resources like recipes can be accessed by multiple chefs simultaneously. The OS similarly allocates and manages resources to keep multiple programs running smoothly, providing non-shareable resources to one process at a time and allowing simultaneous usage of shareable resources.

Resources are the hardware or software required by a program to complete its execution.

For example, if we want to print a document or image that requires a printer, the printer is also a resource. The main job of an operating system is to allocate and manage resources.

For example, if there are multiple programs running on my computer, it's the job of my operating system to provide the resources to the program. Now, this allocation can be of two types: If the resource is not shareable, the OS will allocate it to one process for a time, take it out and give it to another. This process will continue till all the needs of resources are over. In the case of shareable resources, simultaneous usage of resources occurs.

Examples of hardware resources are: CPU, Memory, I/O devices, Registers and Cache

Examples of Software resources are: Device Drivers, Semaphores, and Monitors.

Keywords

Resource-allocation graph, Wait-for graph, safe state, Mutual Exclusion

Discussion

5.4.1 The normal operation mode of a resource

Request: A process asks for a resource. If the resource is currently being used by another process, the requesting process must wait until the resource becomes available.

Use: Once the process acquires the resource, it can perform operations with it. For instance, if the resource is a printer, the process can proceed with printing.

Release: After finishing the operation, the process releases the resource, making it available for other processes to use.

5.4.2 The Deadlock view

It is a situation where a set of processes is blocked because each process is holding a resource and waiting for another resource acquired by some other process. That is where each of the computer processes waits for a resource that is assigned to another process. In this situation, none of the processes gets executed since the resource it needs is held by some other process, which is also waiting for some other resource to be released, as shown in Fig.5.4.1

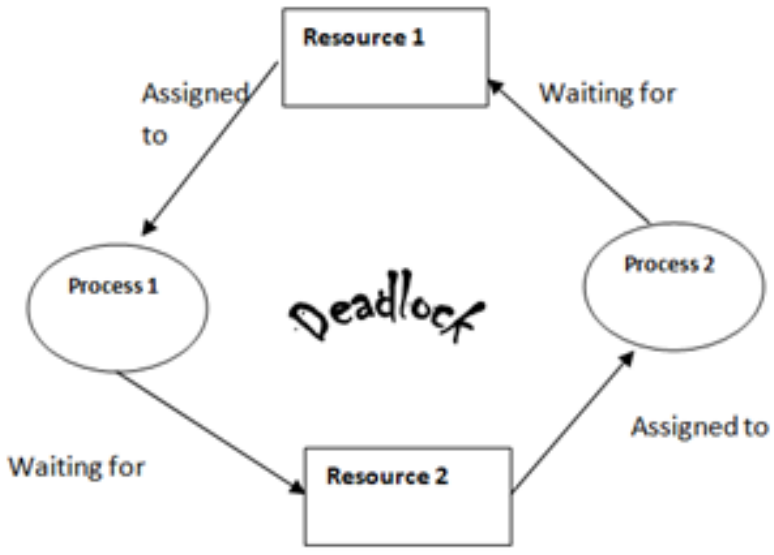


Fig 5.4.1 Deadlock

For example, in a traffic island junction, all the vehicles are blocked, and no vehicles can move forward, as shown in Fig 5.4.2

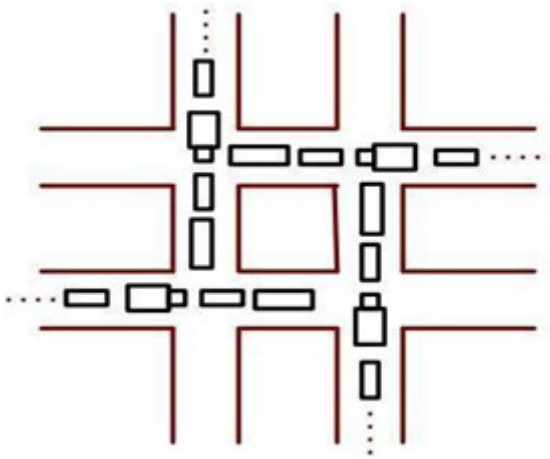


Fig 5.4.2 Deadlock

5.4.2.1 Necessary conditions for deadlock

A deadlock situation can arise if the following four conditions hold simultaneously

in a system

1. Mutual Exclusion.
2. No Pre-emption.
3. Hold and wait
4. Circular wait

1. Mutual Exclusion

Mutual exclusion occurs in the case of resources that are in non-sharable mode. Non-sharable means only one process can use the resource at a time. Other processes need to be used after the usage of others.

2. No Pre-emption

A resource cannot be forcefully released during usage. It will be released only voluntarily by the process holding it.

3. Hold and wait

A process is holding at least one resource and waiting for other resources.

4. Circular wait:

Each process must be waiting for a resource that is being held by another process, which in turn is waiting for the first process to release the resource. A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

5.4.2.2 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system

resource-allocation graph. This graph consists of a set of vertices V, and a set of edges E. The set of vertices V is partitioned into two different types of nodes: P = {P1, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi → Rj; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to process Pi is denoted by Rj → Pi; it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi → Rj is called a request edge; a directed edge Rj → Pi is called an assignment edge.

If there is a cycle in the resource allocation graph, it means that the system is deadlocked. Look at fig 5.4.3 of the resource allocation graph without deadlock.

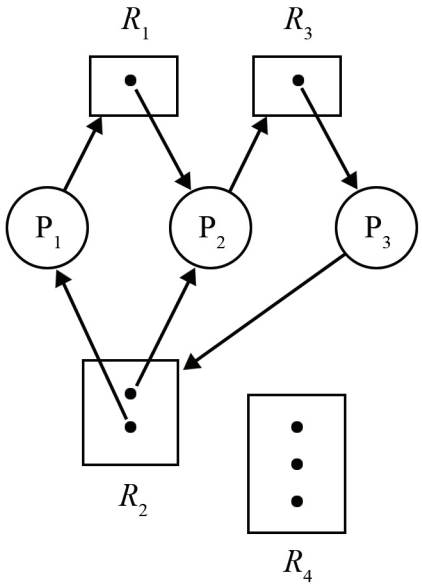


Fig. 5.4.3 Resource Allocation Graph

R = {R1, R2, R3, R4}

E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

Process states:

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.

- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3.

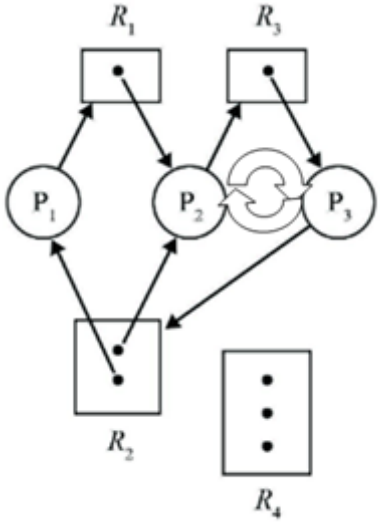


Fig 5.4.4 Resource Allocation Graph with deadlock

P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

As shown in Fig 5.4.4, processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

5.4.3 Methods for handling deadlock

We can deal with the deadlock problem in one of three ways:

- Using a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- Allowing the system to enter a deadlocked state, detect it, and recover.
- Ignoring the problem altogether and pretending that deadlocks never occur in the system.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme. Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

With this additional knowledge, the operating system can decide whether or not the process should wait for each request. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

5.4.3.1 Deadlock Prevention

By ensuring that at least one of the following conditions cannot hold, we can prevent the occurrence of a deadlock.

1. Mutual exclusion
2. No preemption
3. Hold-and-wait
4. Circular wait

5.4.3.2 Deadlock Avoidance

Deadlock-prevention algorithms prevent deadlocks by imposing restrictions on how resource requests can be made, ensuring that at least one of the necessary conditions for a deadlock cannot occur. However, this method can lead to low device utilization and reduced system throughput. An alternative method for avoiding deadlocks is to require additional information about resource requests.

For instance, consider a system with one tape drive and one printer. The system needs to know that process P will request the tape drive first and then the printer, while process Q will request the printer first and then the tape drive. With knowledge of the complete sequence of requests and releases for each process, the system can decide whether a process should wait to avoid a potential deadlock. Each request requires the system to consider the currently available resources, the resources allocated to each process, and the future requests and releases of each process.

Different algorithms using this approach vary in the amount and type of information required. The simplest and most practical model requires each process to declare the maximum number of resources of each type it may need. With this information, an algorithm can ensure that the system will never enter a deadlocked state. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource allocation state is defined by the number of available and allocated resources and the maximum demands of the processes. In the following sections, we will explore two deadlock-avoidance algorithms.

Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j <i. In this situation, if the resources that Pi needs are not immediately

available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain the resources it needs, and so on. If no such sequence exists, then the system state is said to be unsafe. A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 7.6). An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs.

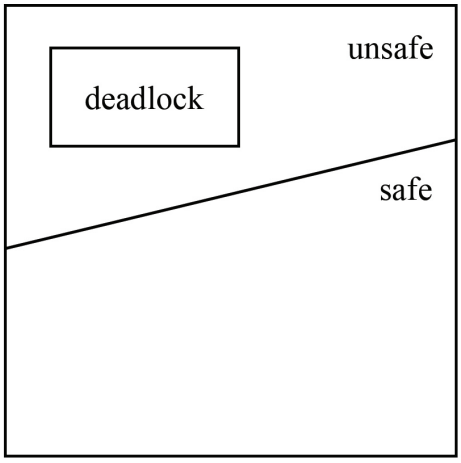


Fig 5.4.5 Safe, unsafe, and deadlocked state spaces

Resource-Allocation-Graph Algorithm

In a resource-allocation system where each resource type has only one instance, we can use an enhanced resource-allocation graph for deadlock avoidance. Along with the request and assignment edges, we introduce a new type of edge called a claim edge. A claim edge, represented as a dashed line from process Pi to resource Rj (i.e., Pi→Rj), indicates that process Pi may request Rj in the future. When Pi actually requests Rj, the claim edge is converted to a request edge. Conversely, when Rj is released by Pi, the assignment edge Rj→Pi is reverted to a claim edge Pi→Rj.

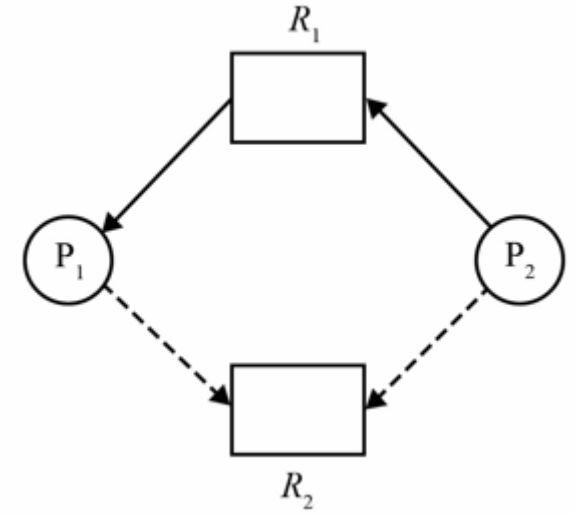


Fig 5.4.6 Resource allocation graph for deadlock avoidance

If the process Pi requests resource Rj, the request is granted only if converting the claim edge to an assignment edge does not create a cycle in the resource-allocation graph. To ensure this, we use a cycle-detection algorithm. Detecting a cycle in this graph requires $O(n^2)$ operations, where n is the number of processes in the system. If no cycle is detected, the resource allocation keeps the system in a safe state. If a cycle is found, the system will be in an unsafe state, and process Pi must wait for its request to be fulfilled.

1. Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe here is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Consider a bank with 'n' account holders and a total cash reserve of 'T'. When an account holder applies for a loan, the bank first deducts the loan amount from the total cash and checks if the remaining cash is greater than 'T'. This ensures that the bank can handle other loan applications or withdrawals without disrupting its operations.

This concept parallels how an operating system function. When a new process is created, it must provide the operating system with information about its upcoming processes, resource requests, counts, and delays. The operating system uses this information to determine the sequence in which processes should be executed or paused, preventing deadlocks. This approach is known as a deadlock avoidance algorithm or deadlock detection in operating systems.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

2. Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted.

Let Requesti be the request vector for process Pi . If Requesti [j] == k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1. If Request ≤ Need, go to step 2. Otherwise, raise an error condition since the process has exceeded its maximum claim.

2. If Request ≤ Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available – Request ;

Allocation = Allocation + Request ;

Need = Need –Requesti ;

If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Requesti, and the old resource-allocation state is restored.

5.4.3.3 Deadlock Detection

Deadlock detection and recovery are critical components of operating system design and management, impacting system stability and performance. The selection of a deadlock detection and recovery method hinges on the system's specific needs and the trade-offs between performance, complexity, and risk tolerance. The operating system must carefully balance these factors to ensure that deadlocks are efficiently identified and resolved.

This section discusses Deadlock Detection and Recovery techniques.

1. **If resources have a single instance –** In this case, for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph. The presence of a cycle in the graph is a sufficient condition for deadlock.

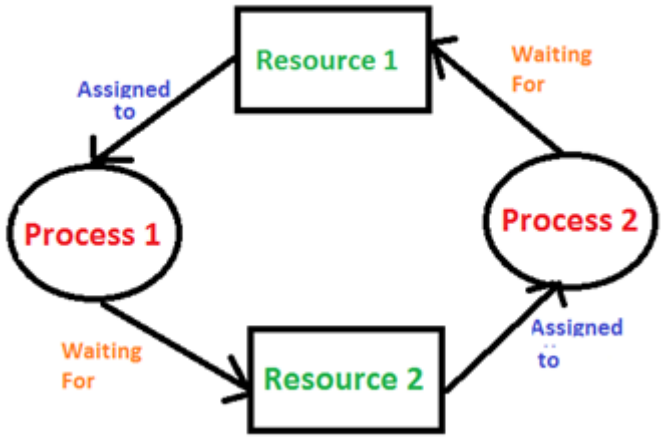


Fig 5.4.7 A deadlock condition

In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is Confirmed.

2. If there are multiple instances of resources – Detection of the cycle is necessary but not a sufficient condition for deadlock detection. In this case, the system may or may not be in a deadlock, which varies according to different situations.

3. **Wait-For Graph Algorithm –**

The Wait-For Graph Algorithm is a deadlock detection algorithm used to detect deadlocks in a system where resources can have multiple instances. The algorithm works by constructing a Wait-For Graph, which is a directed graph that represents the dependencies between processes and resources.

5.4.3.4 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

1. Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- Abort all deadlocked processes. This method clearly will break the deadlock cycle but at great expense. The deadlocked processes may have been computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it would leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to the correct state before printing the next job. If the partial termination method is used, then we must determine which deadlocked processes (or processes) should be terminated.

Many factors may affect which process is chosen, including:

1. What is the priority of the process?
2. How long has the process been computed?
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources does the process need in order to complete?
5. How many processes will need to be terminated?
6. Whether the process is interactive or batch?

2. Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed**:**

1. Selecting a victim

Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

2. Rollback

If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resources. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. Starvation

How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Recap

- A request occurs when a process asks for a resource
- After finishing the operation, the process releases the resource
- Deadlock is a situation where a set of processes are blocked
- A deadlock situation can arise if any of the four conditions hold simultaneously in a system
- **Mutual exclusion:** Only one process can use the resource at a time.
- **No Pre-emption:** A resource cannot be forcefully released during usage.
- **Hold and Wait:** A process is holding at least one resource and waiting for other resources.
- **Circular Wait:** A situation where each process is waiting for a resource held by the next process in the set, forming a circular chain.
- Deadlocks can be described more precisely in terms of resource-allocation graph.
- Deadlock prevention methods
- Deadlock avoidance methods
- Deadlock detection methods

Objective Type Questions

1. What type of resource can be used by multiple processes simultaneously?
2. Which is the necessary condition for occurring deadlock other than No-preemption, Hold and wait, or Circular wait?
3. What is the name of the state in which the system can allocate resources to each process in some order and still avoid a deadlock?
4. What is the purpose of the banker's algorithm?
5. What is the name of the problem encountered in multitasking when a process is perpetually denied necessary resources?
6. What is the visual way to find deadlock occurrences?

7. What is the condition where a process is holding at least one resource and waiting for other resources?

8. What is the name of the condition where a resource cannot be forcefully released during usage?

9. In the resource allocation graph, what is the name of the edge that indicates a process may request a resource in the future?

10. What type of graph is used in systems with single instances of each resource type to detect deadlocks?

Answers to Objective Type Questions

1. Sharable resources
2. Mutual Exclusion
3. Safe state
4. Deadlock Avoidance
5. Starvation
6. Resource allocation graph
7. Hold and wait
8. No pre-emption
9. Claim edge
10. Wait-for graph

Assignments

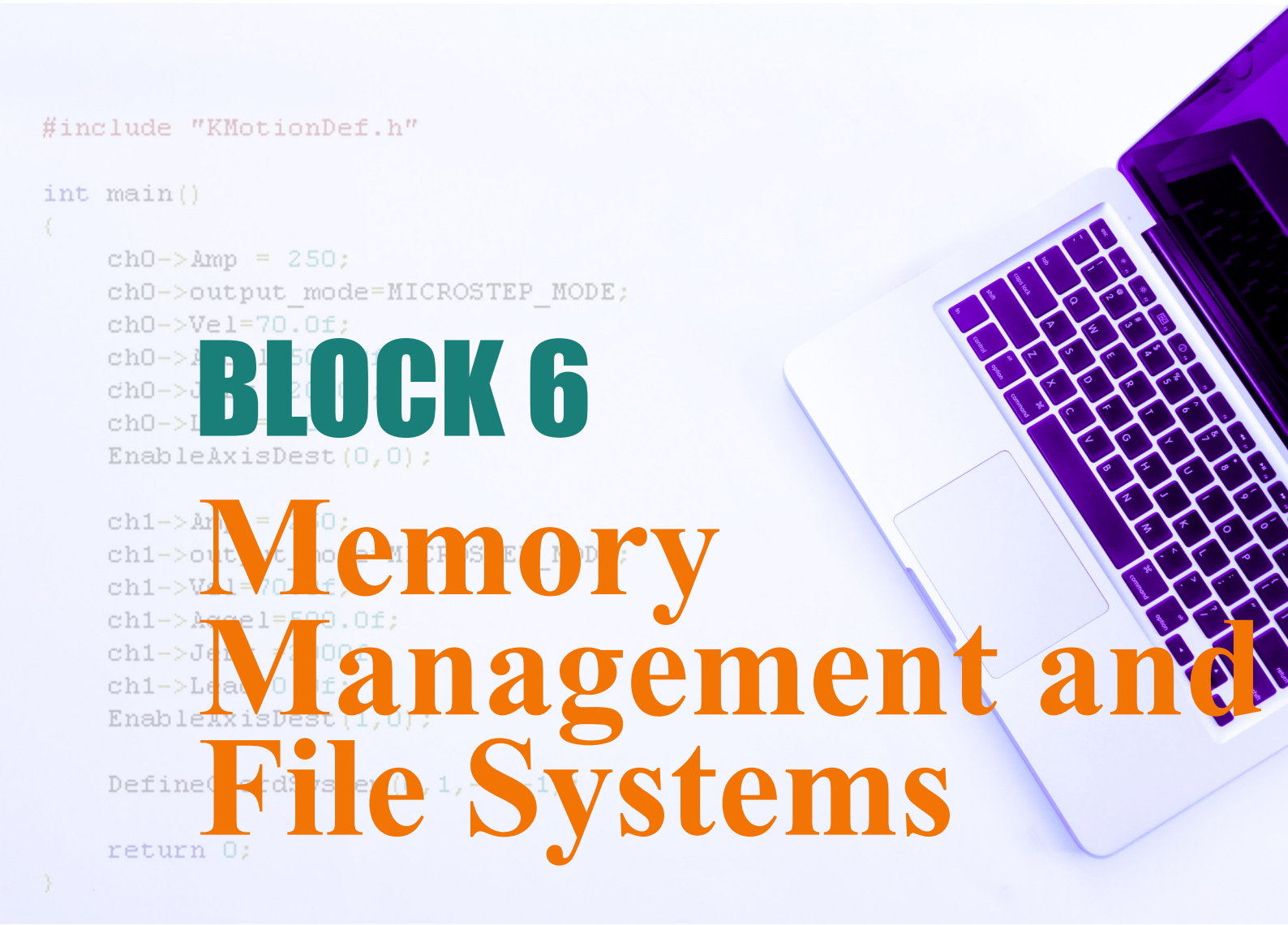
1. Describe the resource allocation graph.
2. Explain the various deadlock prevention methods.
3. Discuss Banker's Algorithm for deadlock avoidance.
4. How can a deadlock be recovered?

Suggested Reading

1. Abraham Silberschatz, Peter Galvin and Greg Gagne - Operating System Principles, Seventh Edition, John Wiley

2. William Stallings - Operating Systems, Sixth Edition, Prentice Hall of India, Pearson

3. Milan Kovic - Operating Systems, Second Edition, TMH



BLOCK 6

Memory Management and File Systems



Memory Management Strategies

Learning Outcomes

By the completion of this unit, the learner will be able to:

- familiarise with memory management strategies of Operating System
- understand contiguous and non-contiguous memory allocation
- identify the binding of instruction and data to memory
- familiarized with dynamic memory allocation

Prerequisites

Memory

Computer memory refers to the electronic components that store data and programs. Two main types of memory are primary and secondary memory. Primary memory, also termed main memory, primarily uses semiconductor electronic circuits to store data temporarily while a computer is running. This type of memory is volatile, meaning its contents are lost when the power is turned off. Primary memory includes high-speed memory that requires a power supply to maintain its data, such as RAM (Random Access Memory), which is further divided into DRAM (Dynamic Random Access Memory) and SRAM (Static Random Access Memory), typically used in CPU cache memory. Another memory called cache memory placed between RAM and CPU to speed up the access of data that the CPU frequently uses.

On the other hand, secondary memory provides long-term storage for large amounts of data and programs. It is non-volatile, meaning its contents are retained even when the power is turned off. Secondary memory includes flash memory, read-only memory (ROM), secondary storage devices such as magnetic disks, optical disks (DVD and CD), and flash memory devices. In summary, primary memory temporarily stores data while a computer runs, whereas secondary memory provides long-term storage for more significant amounts of data.

Memory Management

Memory management is the process of managing a computer's physical memory, or Random Access Memory (RAM), to optimize performance. The primary purpose of memory management is to dynamically distribute memory across all running pro-

cesses to ensure optimal performance. Primary memory contains a limited storage area. However, the Central Processing Unit (CPU) can access primary memory for running applications. If there are more processes for running, it cannot entertain all the running processes simultaneously. Basically, this is the process of controlling and coordinating computer memory by assigning some of the portion (blocks) of the main memory to the running process or application. The program under execution requests the block of main memory, and then the memory management module assigns a memory block of the required size to the program, ensuring protection and security for other processes residing in main memory, including OS.

Keywords

Memory Management Techniques, Types of Memory, Management Requirements, Contiguous Memory, Management Strategies

Discussion

6.1.1 Memory management techniques

Now, we are going to discuss techniques for memory management. Generally, we classify them as Real memory management and Virtual memory management.

6.1.1.1 Real Memory Management

In real memory management, operating systems are responsible for managing the space of the main memory (RAM). Real memory management can be done using either mono-programming or multi-programming. In mono programming systems, one program is executed at a time. Memory is divided into two parts: one part saves or persists OS and the other saves programs. Another approach is a Multiprogramming System in which more than one program is executed at the same time, and the memory space is shared by several processes. When the processes are executed completely, they are taken out of the memory.

6.1.1.2 Virtual Memory Management

Virtual memory management is a technique that allows processes to execute via a transparent scheme even when their entire data and code are not loaded into physical memory. One of the main advantages of this memory is that it provides an illusion that the program can be larger than the physical memory. This creates the illusion of a larger memory space, enabling efficient multitasking and resource utilisation. There are three techniques of virtual memory management. They are Paging Method, Segmentation Method and Paging with Segmentation Method.

This unit provides a detailed discussion of Real Memory Management Strategies.

6.1.2 Memory Management Requirements

As we discussed, the OS is primarily responsible for memory management. The memory management system associated with OS should satisfy the following requirements: Relocation, Protection, Sharing, Logical organisation and Physical organisation.

Relocation

When we write a program, it uses many variables. All these variables are residing in some locations in RAM. When we execute our program, we do not have any idea about the memory block that will be assigned to the program by the OS. All the memory-specific references, such as variables, are to be referred to with actual memory references in the main memory. This arrangement is termed relocation.

We learned about blocking processes in Unit 1 of Block III. When the OS chooses the blocked processes for execution again, there is no guarantee that a blocked process will be placed in the same memory block that was allocated to it before it is moved to the blocked state. In this case, the memory management module has to again relocate the blocked process to the newly allocated block.

Protection

OS will be present in the main memory. There will be different processes residing in the memory. However, it is necessary that the memory occupied by the process should not be tampered with by any other processes. Each process should be protected against unwanted interference (such as reading or writing to a specific memory variable) by another process. Mechanisms that support relocation should also support protection requirements.

Sharing

We mentioned that the memory occupied by each process should be protected from other processes. Then, is it possible to share data and variables associated with a process? While enforcing protection, memory management should also provide the facility to access a portion of memory by many processes. Suppose a number of processes are executing the same program. Each process can use the entire copy of the program (including code and data) and get executed in different portions of the memory. In this case, memory is to be allocated to each of these processes separately, even if they execute the same code. However, if each of these processes shares the code area and makes its own copies of the data alone, a lot of memory can be saved. This is an example of sharing. In such a situation, the program is shared among different processes without affecting protection.

Logical Organization

We know that a computer system has main memory and secondary memory. As we mentioned earlier, the main memory is the "working space" for the computer, and secondary memory is the "storage area" of the computer. How is information stored in memory? Memory consists of a series of locations placed one after the other. Each of these locations has an "address." Hence, whatever may be the content, it is stored in the

memory in a linear manner.

But when we think about programs, we consider them as a single file. Even a program can have modules. We consider each of these modules to be a small portion of the main program. It has many advantages. They are

- Modules can be written and compiled independently.
- It is possible to give different degrees of protection to modules, such as read-only and execute-only.
- It is possible to share modules among processes.

Now, how can a program with a set of modules be placed in the main memory? It is done by dividing the main memory into segments. A program and modules in it, along with its data, can be kept in this segment – hence, we will get a feeling of a single unit of memory allocated to our program. This is the spirit of logical organisation.

Physical Organization

Main memory is relatively high-speed and costly when compared to secondary memory. However, the main memory is volatile. Thus, secondary memory is used for storing data on a long-term basis, while the main memory holds data for currently running programs. How are information (data and programs) copied from primary to secondary memory? How is the data being stored and used as and when needed? All these concerns will be taken care of by the memory management module.

6.1.3 Memory Access

We can imagine RAM as "locations" with specific addresses placed one after the other. How can the CPU access these memory locations? Let us discuss different physical and logical concepts that are associated with access to memory locations.

6.1.3.1 Logical and Physical Address Space

We have learned that an executable program resides in secondary memory. When it is to be executed, the program is transferred to the main memory, which is then called a process. In order to simplify memory access, the OS uses the concept of logical and physical address. Here, the address refers to the location of data in the memory. Physical address refers to the physical location of the data in a memory. Hence, we can say that when a program is transferred to the main memory for execution, it is transferred to the physical memory – RAM. A logical address or virtual address is the address of a location generated by the CPU during process execution. When the CPU executes an instruction, it gets the logical address from the instructions being executed. It is referred to as a virtual address because it does not physically exist.

The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses is a physical address space.

The logical address is mapped to the corresponding physical address by an address

mapping arrangement. We will learn about this arrangement in this unit when we discuss memory management schemes.

6.1.4 Base and limit registers

Base and limit registers are the basic hardware elements used to accomplish the mapping from physical to logical address. When a portion of a program being executed is transferred to the main memory, the OS keeps two information:

(i) The starting address of the main memory block to which the portion of the program being executed is transferred, and it is called the base address

(ii) The size of the main memory block allocated to the portion of this program is termed the limit.

The base address and limit of each memory block are stored by the OS using registers, and they are called base and limit registers, respectively. A base register is also called a relocation register.

What is the use of the base and limit registers? Every memory access performed by the process should be between the base register and limit registers. Hence, the memory management module of the OS continuously checks whether a process demands memory access beyond its allocated limits. Do you remember the protection requirements we discussed earlier?

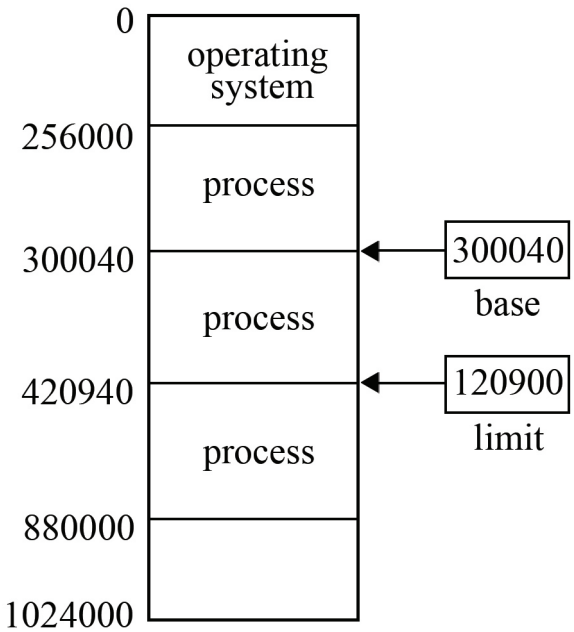


Fig 6.1.1 Illustration of Base and Limit registers

The base and limit registers are essential hardware components used in memory management to ensure safe and efficient access to memory by processes. Here's how they function:

Base Register:

This register holds the starting address of a process's allocated memory space. When a process accesses memory, the address generated by the CPU is added to the base address to get the actual physical address in memory.

Limit Register:

This register defines the size of the memory allocation for the process. It specifies the range of valid addresses that the process can access. Any access attempt beyond this limit results in an error or exception, preventing unauthorised memory access.

Figure 6.1.1 illustrates the base and limit register. In Figure 6.1.1, the second process starts at the address 300040 (base), and the size (or limit) of this block is 120900. So, during its execution, the base register contains the value 300040, and the limit register will contain the value 120900.

6.1.5 Dynamic relocation using base register

Let us see how the base register and limit register can be used to access a memory location. Recall that the logical address is the address generated by the CPU. We have learned that the program counter (PC) in the CPU will have the address of the current instruction being executed. When the CPU executes the current instruction, let us assume it generated a logical address 346 (Refer to Figure 6.1.2). Address location 346 is the logical address. This address will be mapped to the physical address in the main memory; for this, a base register will be used. The value of the base register (14000) will be added to the logical address 346 (Refer to Figure 6.1.2). This will map the logical address 346 to the physical address 14346. Now, what will happen if the logical address leads to a physical address that will not fit the limit register for that block? The address mapping hardware will check this, and such attempts will not be allowed.

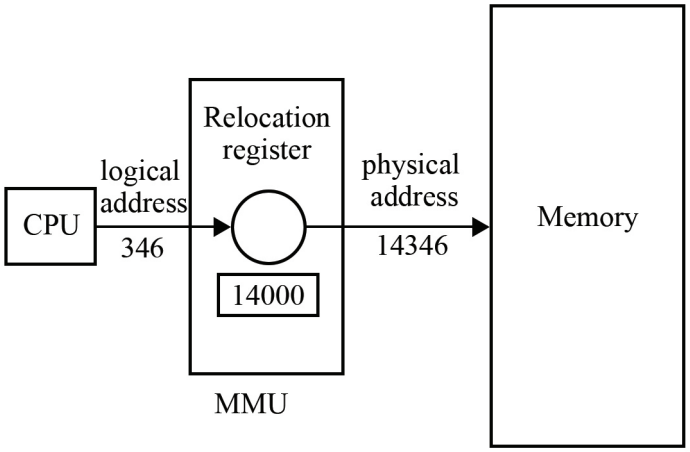


Fig 6.1.2 Relocation of physical address to logical address

This process of mapping a logical address to the corresponding physical address is termed dynamic relocation. The OS module that performs address mapping with the help of a relocation register is called a memory management unit (MMU). We have

learned that the relocation requirement for memory management modules, and here it is satisfied.

6.1.6 Binding of Instruction and Data to Memory

In semester 1, we completed a course on Problem Solving and Programming in C. We used at least one variable in our C program. We have learned that each variable in our program is a memory location in the main memory. Here, we have an issue! When writing the program, we were not sure about the memory location in which it would be allotted or assigned, which was "binded"! Who will do this binding, and when will this be done? Is it during the program execution or compilation time?

Address Binding refers to mapping computer instructions and data to physical memory locations. That is the process of allocating an address for symbolic references such as variables in a high-level program. Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any of the following steps:

Compile time: Suppose the details of the main memory block allocated to a program are known in advance. In this case, we can do the address binding (i.e., binding of the actual address of a variable with the main memory location) at the time of compilation. Hence, in this scheme, all variables will be allocated their actual addresses at the time of compilation. However, if the start address of the main memory block changes later, the program must be recompiled.

Load time: If the main memory block being allocated for the program is not known at compile time, then the address binding is to be done at the time of transferring the executable code to the main memory. This is called load time address binding.

Execution time: Usually, a process will be assigned a set of memory blocks for execution. The process will complete its execution while remaining in that memory block. In some cases, processes will move to other memory segments during execution. In such cases, address binding will be done during execution – here, the addresses of variables will change from one memory segment to another. This is called execution time binding.

6.1.8 Swapping

We have learned that all programs being executed (i.e. processes) must be in main memory. They will continue to be in main memory till their execution is over. We have also learned that a process is having a run state and a blocked state. A process in the run state will be executed by the CPU. A process in the blocked state will not be executed, but it will remain in the main memory.

Here, we have a typical issue. Suppose that in a system, there are several processes – some of these processes are in the run state, and some of them are in the blocked state. As we mentioned earlier, all these processes are in the main memory. Let us assume that OS was executing Process P1 and OS would like to execute Process P2. When the OS checks the main memory, it detects there is no space available for P2. However, how

can the OS allocate the additional blocks required for the P2 process? In such situations, the memory management unit of the OS will undergo a process called swapping.

Swapping is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that the main memory can be made available for other processes. It is used to improve main memory utilisation. In secondary memory, the place where the swapped-out process is stored is called swap space.

The concept of swapping has been divided into two more concepts: Swap-in and Swap-out.

- Swap-out is a method of removing a process from RAM and adding it to the hard disk.
- Swap-in is a method of removing a process from a hard disk and putting it back into the main memory or RAM.

In the secondary storage device, an area called a swap area will be created. OS then chooses one of the blocked processes and is moved temporarily out of the main memory to the swap area (also called a backing store). Hence, there will be some space free in the main memory. Now, the OS can satisfy the demand by process P2 (Refer to Figure 6.1.3). Here, process P1 is "swapped out" from main memory, and P2 is "swapped in". OS will swap P1 again to the main memory as and when P1 is to be executed.

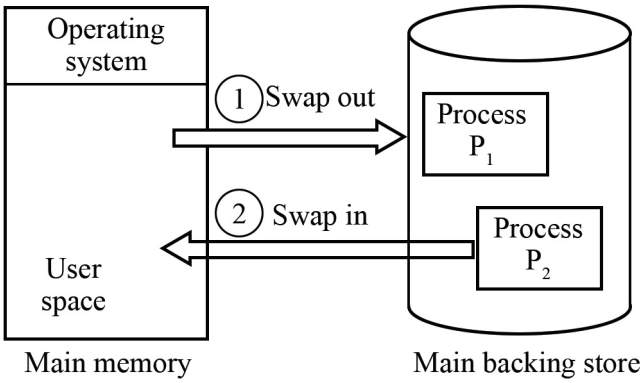


Fig. 6.1.3 Swapping of two processes using disk as a backing store

6.1.8 Memory Management Strategies

The principle operation of memory management is to allocate memory blocks to processes for execution. We know that RAM contains OS and other processes. The part of RAM that is not used by the OS is to be allocated to all the processes currently in memory and for the newly created processes. Different kinds of memory allocation schemes are available and can be broadly classified into two:

1. Contiguous Allocation
2. Noncontiguous allocation

Contiguous memory allocation is further classified into two. They are Fixed Partitioning and Dynamic Partitioning. Noncontiguous memory allocation is further classified into paging and segmentation. In this unit, we will discuss contiguous memory allocation.

6.1.8.1 Contiguous memory allocation

In this allocation scheme, consecutive memory locations are allotted for a process. Accessing information from contiguous memory locations is similar to reading consecutive pages of a book to get information. Let us discuss the distribution of consecutive memory locations to a process. It is divided into two: they are

- Fixed partitioning
- Dynamic partitioning

6.1.8.2 Fixed Partitioning or Static Partitioning

In this method, as its name suggests, RAM is divided into fixed-size partitions. The word "fixed" says that once the RAM locations are partitioned, they remain fixed without changing the size of each partition. The size of each partition may or may not be the same.

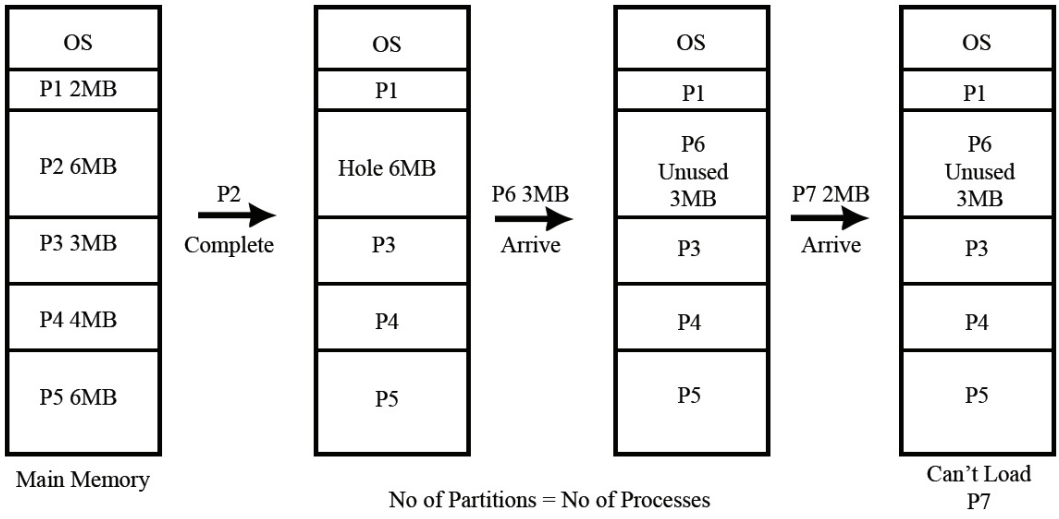


Fig 6.1.4 Illustrates the fixed-partition scheme

In a fixed-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests. In Figure 6.1.4, after the completion of P2, another process, P6, with size 3KB, is allocated to partition2 of size 6 KB, and then 3KB will remain there without filling. Here, there is an unused area of memory within an allocated block. This wastage of memory is called internal fragmentation.

Internal fragmentation: When a process is allotted a memory block, and if the process

requires a smaller amount of memory than the requested block, a free space is created inside the given memory block. Because of this, the free space of the memory block will be unused, which causes internal fragmentation. Also, it is only possible to include up to five processes because the memory contains exactly five partitions. This will limit the degree of multiprogramming. Even though it has these limitations, it is the simplest and easiest method to implement. This method was originally used by the IBM 05/360 operating system (called MFT); it is no longer in use.

6.1.8.3 Dynamic partitioning or variable partitioning technique

We discussed the approach of contiguous allocation of memory. It is the simplest memory allocation method. However, it suffers from internal fragmentation. Using dynamic partitioning, it is possible to alleviate or mitigate the problem faced by fixed partitioning.

In contrast with fixed partitioning, partitions are not made before the execution or during system configuration. Initially, RAM is empty, and partitions are made during the run-time according to the process's need instead of partitioning during system configuration. The size of the partition will be equal to the incoming process. The partition size varies according to the need of the process so that internal fragmentation can be avoided to ensure efficient utilization of RAM. The number of partitions in RAM is not fixed and depends on the number of incoming processes and the size of the main memory. Figure 6.1.5 illustrates the dynamic partitioning of memory. In Figure 6.1.5, a 32 MB RAM is divided into five partitions according to the size of the incoming process. Five processes, P1, P2, P3, P4 and P5, of size 2MB, 6MB, 3MB, 4MB and 5MB, respectively, are allocated a block of memory similar to their size.

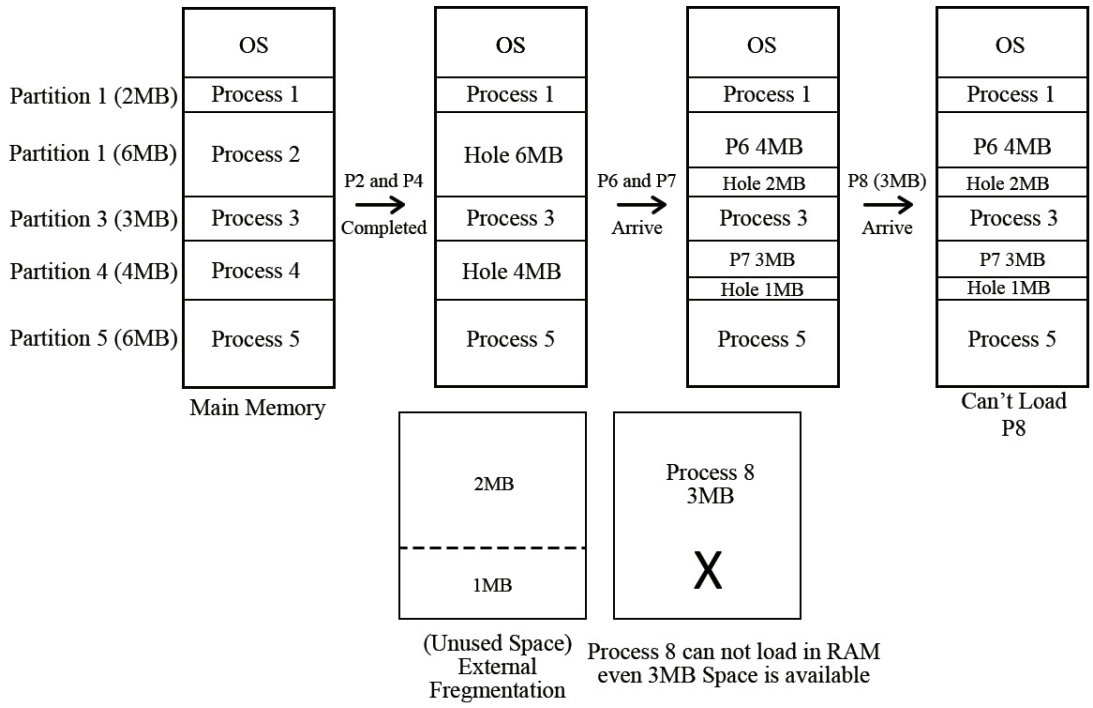


Fig 6.1.5 Dynamic partitioning of memory

Advantages of Dynamic Partitioning

1. No Internal Fragmentation: In the variable partitioning approach, space in the main memory is allocated strictly according to the needs of the process. Hence, there is no chance of internal fragmentation.

2. No restriction on the Degree of Multiprogramming: More number of processes can be accommodated due to the absence of internal fragmentation. A process can be loaded until the memory is empty.

3. No Limitation on the size of the process: In Fixed Partitioning, the process with a size greater than the size of the largest partition could not be loaded. However, in variable Partitioning, the process size can't be restricted since the partition size is decided according to the process size.

Disadvantages of Dynamic Partitioning

1. Difficult to Implement: Implementing variable partitioning is difficult compared to fixed partitioning as it involves the allocation of memory during run-time rather than during system configuration.

2. External Fragmentation: There will be external fragmentation in spite of the absence of internal fragmentation. For example, consider Figure 6.1.6. It shows a RAM containing five processes: P1, P2, P3, P5 and P5. Out of these five processes, two processes, P1 (2MB) and P3 (3MB), completed their execution. Hence, 5MB is left. Let's suppose a process P6 of size 5MB comes. The empty space in memory cannot be allocated as no spanning is allowed in contiguous allocation. There is no way to divide the 5MB process to occupy empty spaces of 2MB and 3MB, respectively. It results in a wastage of memory called external fragmentation.

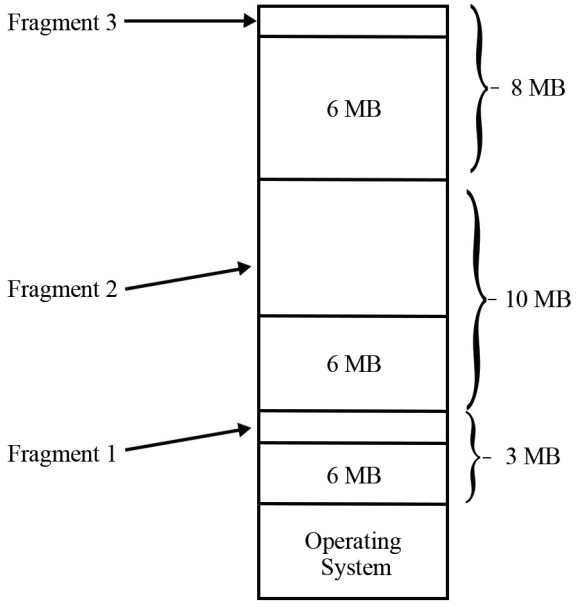


Fig 6.1.6 Illustration of external fragmentation

Compaction is a technique used for overcoming external fragmentation.

Compaction is a technique to collect all the free memory present in the form of fragments into one large chunk of free memory, which can be used to run other processes. From time to time, the OS shifts the process so that they are contiguous. Thus, it arranges all the free memory in one block. It is a time-consuming procedure, and it wastes processor time.

When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes. This procedure is a particular instance of the general dynamic storage allocation problem.

6.1.9 Dynamic Memory Allocation Algorithms

The most commonly used algorithms for dynamic memory allocation are the first-fit, best-fit, next-fit and worst-fit strategies.

First fit: Allocate the first hole that is big enough. Searching can start at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Next fit: In the first fit, if searching starts from the last allocated hole, it is called the next fit.

Best fit: Allocate the smallest hole that is big enough. We must search the entire list unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst fit: Allocate the largest hole. Again, we must search the entire list unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Suppose a process P3 of size 2KB wants memory location to be allocated.

According to the **first fit**, it will be allocated the first free hole that is big enough to accommodate P3. So, it is allocated to Hole1, which is 6KB in size.

According to the **next fit** algorithm, P3 is allocated to Hole2 of size 4KB, which is next to P2.

According to the algorithm **best fit**, in that case, search the entire list to allocate the smallest hole, then the Hole3, which is big enough to accommodate a 2KB process P3.

According to the **worst fit,** P3 is allocated the largest hole by searching all the free holes that are available. Here, Hole4 8KB is the largest hole, then P3 is allocated to Hole4.

Recap

- Memory Management is the process of controlling and coordinating Random Access Memory (RAM), the primary memory of a computer. It is one of the services offered by the Operating System.
- Various requirements that a Memory Management System should satisfy include Relocation, Protection, Sharing, Logical organization and Physical organization.
- The concept of a logical and physical address space is central to proper memory management.
- Logical address – address generated by the CPU; also referred to as virtual address.
- Physical address – address seen by the memory unit.
- The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses is a physical address space.
- Base and limit registers are the basic hardware elements used to map from logical to physical addresses. These registers provide starting and ending address location blocks of memory allotted for a particular process that is residing in RAM.
- The mechanism used for mapping a logical address to the corresponding physical address is called dynamic relocation. The module of the OS that performs address mapping with the help of relocation registers is called the memory management unit (MMU).
- Address Binding is the process of allocating an address for symbolic references such as variables in a high-level program. Each binding is a mapping from one address space to another. Address binding of instructions and data to memory addresses can happen at three different stages: at compile time, load time or Execution time.
- OS uses different strategies for the allocation of primary memory among processes when the CPU wants to execute them. These strategies are broadly classified into two: Contiguous allocation and noncontiguous allocation.
- In contiguous allocation, a process allocates consecutive memory locations.
- In noncontiguous allocation, a process is allocated to a nonconsecutive memory location.
- Contiguous allocation is further classified into two: Fixed partitioning and dynamic partitioning.
- In contiguous fixed partitioning, memory locations are partitioned into equal-sized blocks and are allocated to a process.

- In contiguous dynamic partitioning, memory locations are partitioned into unequal-sized blocks and are allocated to a process.
- The main drawback of contiguous memory allocation is that it faces the problem of fragmentation.
- In the dynamic partition scheme, RAM partitions are made during the run-time according to the process's need instead of partitioning during system configuration. The size of the partition will be equal to the incoming process.
- The main advantage of the Dynamic partition scheme is that there is no internal fragmentation and no limit to the degree of multiprogramming.
- However, in most cases, Dynamic partition suffers from external fragmentation.
- The most commonly used algorithms for dynamic memory allocation are the first-fit, best-fit, next-fit and worst-fit strategies.

Objective Type Questions

1. What is the need for memory management?
2. What is the logical address?
3. What is a physical address?
4. Name the categorisation of memory management.
5. What do you understand about relocation?
6. What is address binding?
7. What are the different methods of address binding?
8. Who defines a logical address space?
9. What is Swapping?
10. What is the need for swapping?
11. Name different memory partitioning strategies.
12. Differentiate contiguous memory allocation and noncontiguous memory allocation.
13. What are the disadvantages of contiguous memory allocation?
14. What are the different dynamic contiguous memory allocation algorithms?

15. What are the disadvantages of a fixed partitioning memory allocation scheme?

16. What is internal fragmentation?

17. What is external fragmentation?

18. What is the disadvantage of dynamic contiguous memory allocation schemes?

19. What is compaction?

Answers to Objective Type Questions

1. Dynamically distribute memory across all running processes to ensure optimal performance.

2. Address generated by the CPU during program execution.

3. It is the actual address in the main memory where data is stored.

4. Real memory management and virtual memory management.

5. All the memory-specific references such as variables, are to be referred to with actual memory references in main memory, which is called relocation.

6. The Address Binding refers to the mapping of computer instructions and data to physical memory locations.

7. Compile time binding, load time binding and execution time binding.

8. CPU

9. Swapping involves the temporary swapping of processes from main memory to secondary memory so that the main memory can be made available for other processes.

10. Swapping is used to improve main memory utilisation.

11. Contiguous memory allocation and Non-contiguous allocation.

12. Contiguous memory allocation uses a single memory space for tasks, while non-contiguous allocation assigns tasks to separate memory sections at different locations.

13. Internal fragmentation and external fragmentation.

14. First fit, next fit, best fit, worst fit.

15. Internal fragmentation

16. When a process is allocated to a memory block smaller than the requested size, unused free space within the block causes internal fragmentation.

17. External fragmentation occurs when sufficient memory space exists to meet process requirements, but it is non-contiguous and cannot be utilised effectively.

18. External fragmentation.

19. Compaction is a technique that consolidates fragmented free memory into a single large block, allowing it to be used for other processes.

Assignments

1. Explain different memory allocation schemes.

2. Suppose we have five processes with memory sizes of 300K, 25K, 125K, and 50K, respectively. Try to allocate memory for these processes using dynamic memory allocation algorithms' first fit and best fit. Which algorithm is most suitable? Analyse.

3. Explain internal and external fragmentation.

4. Explain first fit, next fit, best fit and worst fit with example.

5. What is compaction? How compaction is performed in memory?

6. Explain need of swapping in multiprogramming.

7. What is address binding? Explain address binding with example.

References

1. Silberschatz, Abraham, James L. Peterson, and Peter B. Galvin. Operating system concepts. Addison-Wesley Longman Publishing Co.

2. Operating system Concepts: Milan Malinkovic, TMH, 2nd ed.

3. Stallings, William. Operating Systems: Internals and Design Principles.

Suggested Reading

1. Silberschatz, Abraham, Peter Galvin, and Greg Gagne. Operating System Principles. 7th ed., John Wiley.

2. Tanenbaum, Andrew S., and Herbert Bos. Modern Operating Systems.



Paging and Segmentation

Learning Outcomes

By the completion of the unit, the learner will be able to:

- recall the definition of paging and its role in memory management.
- identify what segmentation is and how it differs from paging.
- list the key characteristics of paging and segmentation.
- point out the types of fragmentation (internal and external) associated with paging and segmentation.

Prerequisites

In the first unit, we discussed different contiguous memory allocation schemes. The main disadvantage of a contiguous allocation scheme is the wastage of memory space due to internal and external fragmentation. What are the solutions for the fragmentation of main memory? The answer to this question is the non-contiguous allocation of main memory. In this unit, we will discuss different non-contiguous memory allocation schemes, such as Paging and Segmentation.

Keywords

Non-contiguous memory allocation, Paging, Segmentation

Discussion

6.2.1 Paging

Paging is a noncontiguous memory allocation scheme used for Real Memory management. In this scheme, we assume that the main memory and the processes are divided into small fixed-sized chunks of the same size. The partition in main memory is called **frames**, and partitions in the process are called **pages**. Then, pages can be assigned to available memory frames. High paging activity is called **Thrashing.**

For example, a process named A is divided into four equal-sized partitions (blocks) of 2KB (Kilo Bytes) called **pages**. Similarly, main memory is also divided into equal-sized partitions of the same size, 2KB and is called **frames**. During the execution of process A, instead of transferring the whole process, only necessary pages are transferred to the main memory. A data structure called Page Table is used to map the page number referenced by the CPU to the frame number where that page is stored.

A.0 (2KB) A.1 (2KB) A.2 (2KB) A.3 (2KB)

Process A

A.0 (2KB)

A.1 (2KB)

A.2 (2KB)

A.3 (2KB)

Main Memory

6.2.2 Basic Method of Paging

The implementation of paging consists of a set of hardware. We learned that the CPU executes every instruction of a computer program. For the execution of instructions, it must be available in Main Memory, RAM. At the time of execution, the CPU generates the logical address of the required instruction. However, it is different from the actual address of the data in the main memory (RAM). So, mapping from logical address to physical address is essential. This address translation is performed with the help of the hardware part, and the process is explained below.

The address generated by the CPU is termed a "logical address" or "virtual address." This address is used by programs to access memory locations. This is translated into a physical address by the memory management unit (MMU) before being used to access the physical memory (RAM). The logical address is divided into two parts:

- **Page number (p) –** used as an index in a page table that contains the base address of each page in physical memory.
- **Page offset (d) –** combined with base address to define the physical memory address that is sent to the memory unit.

Translation of Logical Address into Physical Address

The following steps are followed to translate a logical address into a physical address

Step1: Generation of Logical Address

CPU generates a logical address consisting of two parts

- **Page Number (p):** This part specifies the specific page of the process from which the CPU wants to read or write data.
- **Page Offset (d):** This part specifies the specific word or byte on the page that the CPU wants to access.

Page number	offset os
p	d
m-n	n

For the given example, logical address space is 2^m and page size is 2^n

Step 2: Look Up in Page Table

- The logical address is sent to the Memory Management Unit (MMU).
- The MMU uses the **Page Number (p)** to index into the **Page Table**. The Page Table is a data structure maintained by the operating system, which holds the mapping between logical pages and physical frames.

Step 3: Extract Frame Number

- The Page Table provides the corresponding frame number (base address of the frame) where that page is stored in the main memory, as shown in Fig 6.2.1. A special type of register called Page Table Base Register (PTBR) is used as an index to the page table, as shown in page table 6.2.1 b

Step 4: Formulate Physical Address

The frame number combined with the page offset forms the required physical address, as shown in fig 6.2.2.

- **Frame Number -** specifies the specific frame where the required page is stored.
- **Page Offset -** specifies the specific word that has to be read from that page.

Physical Address = Frame Number + Page Offset

Frame number	offset os
f	d

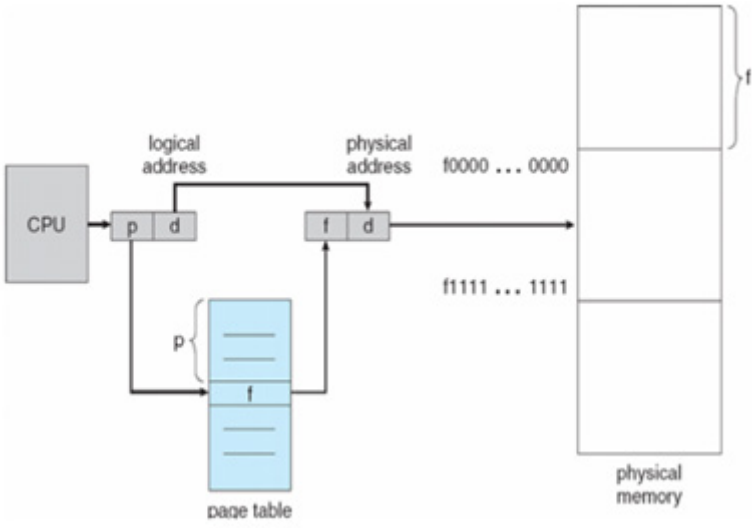


Fig 6.2.1a Page Table

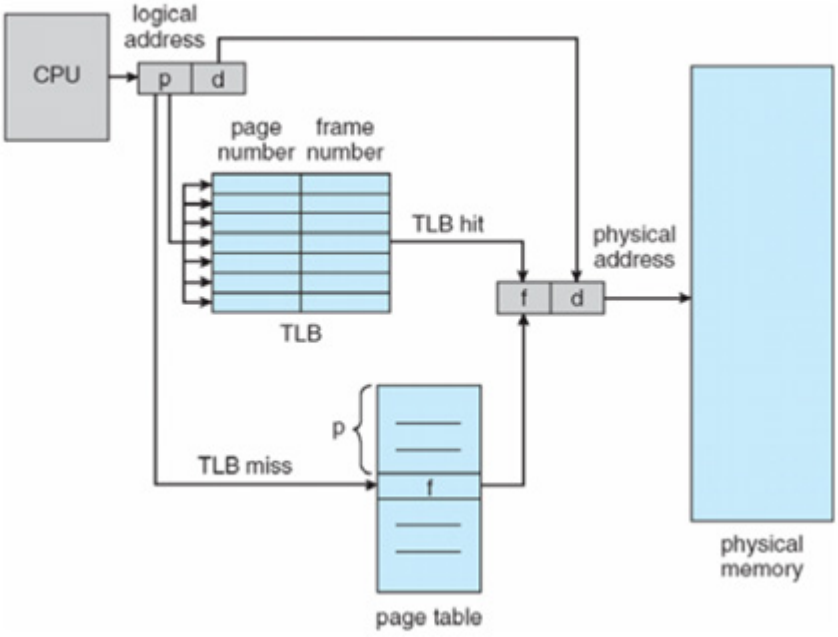


Fig 6.2.1b Page Table Base Register (PTBR)

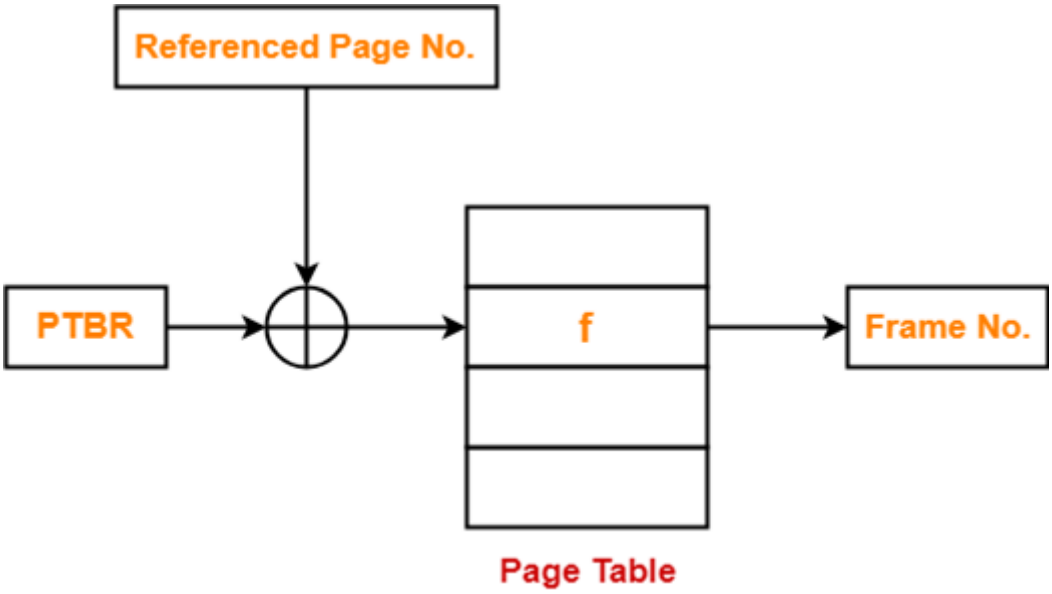


Fig 6.2.2 Computing Frame Number using Page Table

Example

Suppose we have a computer system with a physical memory size of 8KB and a page size of 2 KB. In this scenario, the physical memory can be divided into 2KB pages, resulting in a total of 4 frames (8 KB/2 KB).

The operating system maintains a data structure called a page table, which keeps track of the mapping between the logical addresses used by the process and the physical addresses in memory. The page table contains an entry for each page, indicating its

corresponding physical frame.

For example, the page table might look like this:

Page Table

Page	Frame
1	2
2	3
3	1

In the page table, we can see that Page 1 is mapped to Frame 2, Page 2 is mapped to Frame 3, and Page 3 is mapped to Frame 1.

- **Physical Memory Size:** 8 KB
- **Page Size:** 2 KB
- **Total Number of Frames:** 4 frames (since 8 KB / 2 KB = 4)
- **Process Memory Requirement:** 10 KB

Paging Process

Since the process requires 10 KB of memory and each page is 2 KB in size, the process needs to be divided into pages:

- Number of Pages Needed for the Process:

10 KB / 2 KB = 5 pages

Memory Allocation

Given that the physical memory can only accommodate 4 frames (4 pages of 2 KB each), the process of 5 pages cannot fit entirely into physical memory all at once. Therefore, the operating system will use a technique called **paging** to manage the allocation of memory.

- **Step-by-Step Allocation:**

1. The first 4 pages of the process will be loaded into the 4 available frames in physical memory.
2. The remaining 1 page will be kept in secondary storage (like a disk).

Therefore, mapping may be done as Page 0 —> Frame 0

Page 1 —> Frame 1

Page 2 —> Frame 2

Page 3 —> Frame 3 (Page 4 Cannot be loaded)

When a page that is not currently in memory is needed, the operating system will use a **page replacement algorithm** (such as Least Recently Used or First-In-First-Out) to swap one of the pages in physical memory with the required page from secondary storage.

We will study the page replacement strategies in detail in the next unit, which comes under virtual memory management.

6.2.3 Advantages and Disadvantages of Paging

Advantages

- It allows storing parts of a single process in a non-contiguous fashion.
- It solves the problem of external fragmentation.

Disadvantages

- It suffers from internal fragmentation.
- There is an overhead of maintaining a page table for each process.
- Need more time to fetch the instructions from the main memory.

6.2.4 Segmentation

Like Paging, Segmentation is another non-contiguous memory allocation technique. In segmentation, the process is not simply divided into fixed-size pages. Instead, the process is divided into modules for better visualisation.

Characteristics of segmentation

- Segmentation is a variable-size partitioning scheme.
- In segmentation, secondary memory and main memory are divided into partitions of unequal size.
- The size of partitions depends on the length of the modules.
- The partitions of a process are called segments.

Segment table

- Segment table is a table that stores the information about each segment of the process.
- It has two columns.
- The first column stores the size or length of the segment.
- The second column stores the base address or starting address of the segment in the main memory.
- The segment table is stored as a separate segment in the main memory.

- A special type of register called Segment table base register (STBR) stores the base address of the segment table.

The limit indicates the length or size of the segment. Base refers to the base address or starting address of the segment in the main memory. In accordance with the above segment table, the segments are stored in the main memory.

Translation of logical address to physical address

We know that the CPU always generates a logical address. We need to map logical addresses into physical addresses to access the main memory. The different steps involved in the mapping of logical address to physical address are given below.

Step 1: Generation of Logical Address

The CPU generates a logical address consisting of two parts: Segment Number and Segment Offset.

- **Segment Number (s):** Segment Number specifies the specific segment of the process from which the CPU wants to read the data.
- **Segment Offset (d):** Segment Offset specifies the specific word in the segment that the CPU wants to read.

Step 2: Accessing the Segment Table

- The logical address is sent to the Memory Management Unit (MMU).
- The MMU uses the Segment Number (s) to index into the Segment Table.
- The Segment Table is a data structure maintained by the operating system, which holds the base address and limit (size) for each segment.

The corresponding entry for the generated segment number is located in the segment table. Then, the segment offset is compared with the limit (size) of the segment. Now, two cases are possible.

Case-01: Segment Offset >= Limit

If segment offset is found to be greater than or equal to the limit, a trap is generated.

Case-02: Segment Offset < Limit

If segment offset is found to be smaller than the limit, then the request is treated as a valid request.

The segment offset must always lie in the range [0, limit-1]. Then, the segment offset is added with the base address of the segment. The result obtained after addition is the address of the memory location storing the required word.

Advantages of segmentation

- It allows the program to be divided into modules, which provides better visualization.

- Segment tables consume less space than page Tables in terms of paging.
- It solves the problem of internal fragmentation.

The disadvantages of segmentation are-

- There is an overhead of maintaining a segment table for each process.
- The time taken to fetch the instruction has increased since two memory accesses are now required.
- Segments of unequal size are not suited for swapping.
- It suffers from external fragmentation as the free space gets broken down into smaller pieces, and the processes are loaded and removed from the main memory.

Recap

- Paging is a noncontiguous memory allocation scheme used for Real Memory management.
- In paging, process and main memory are divided into equal fixed-size partitions. The partitions of the process are called pages. The partitions of the main memory are called frames. Page size and frame size are always equal.
- Paging uses a data structure called page table for the mapping of CPU generates logical address to physical address.
- In Operating Systems, Segmentation is a memory management technique in which the memory is divided into variable size parts. Each part is known as a segment, and it can be allocated to a process.
- A data structure called a segmentation table stores information about the segments of a process.
- With the help of the segmentation table, the logical address generated by the CPU is converted to a physical address during segmentation
- Using segmentation, we can eliminate internal fragmentation

Objective Type Questions

1. What is the name of high paging activity?
2. Which memory is broken into fixed-sized blocks called frames?
3. Which memory is broken into blocks of the same size called Pages?
4. What is the recommended methodology for specifying the size of a page?
5. What is used as an index into the page table?
6. What are the divisions of address generated by the CPU in paging?
7. With paging, which fragmentation can be solved?
8. Which are the factors used to specify address in segmentation?
9. Which table is associated with the factor segment limit?
10. Which factor is represented by the segment base?
11. What is represented using segment limits?
12. If there are 32 segments, each of size 1Kb, how many bits are there in the logical address?

Answers to Objective Type Questions

1. Thrashing
2. Physical
3. Logical
4. Power of two
5. Page number
6. Page number and page offset
7. External fragmentation
8. A segment number and offset

9. Segment table
10. Starting physical address of segment
11. Segment length
12. 15 bits

Assignments

1. Differentiate physical address and logical address.
2. Explain different types of memory management techniques.
3. What is paging? Explain advantages and disadvantages of paging.
4. Explain framing in memory management.
5. What is segmentation. Explain with example.
6. Explain data structures used in segmentation and paging.
7. Explain advantages and disadvantages of segmentation.

References

1. Silberschatz, Abraham, James L. Peterson, and Peter B. Galvin. Operating system concepts. Addison-Wesley Longman Publishing Co.
2. Operating system Concepts: Milan Malinkovic, TMH, 2nd ed.
3. Stallings, William. Operating Systems: Internals and Design Principles.

Suggested Reading

1. Silberschatz, Abraham, Peter Galvin, and Greg Gagne. Operating System Principles. 7th ed., John Wiley.
2. Tanenbaum, Andrew S., and Herbert Bos. Modern Operating Systems.



UNIT 3 Virtual Memory Management

Learning Outcomes

By the completion of the unit, the learner will be able to:

- describe the benefits of a virtual memory system.
- explain the concepts of demand paging.
- discuss page-replacement algorithms.
- examine the allocation of page frames.

Prerequisites

The memory-management algorithms previously discussed are essential because of a fundamental requirement: instructions being executed must reside in physical memory. The initial method to satisfy this requirement is to place the entire logical address space into physical memory. While dynamic loading can alleviate this restriction, it typically necessitates special precautions and additional effort from the programmer.

The need for instructions to be in physical memory for execution seems both necessary and reasonable. Still, it is also limiting because it restricts a program's size to that of the physical memory. However, studies of real programs reveal that the entire program is often not needed in main memory at all times. Even when a full program is required, not all parts are needed simultaneously. Executing a program that is not fully in memory would be advantageous for both the system and the user.

Virtual memory addresses this issue by separating logical memory, as perceived by users, from physical memory. This separation enables the provision of a substantially larger virtual memory for programmers, even when only a smaller physical memory is available.

Keyterms

Virtual memory, Virtual address space, Demand Paging, Page replacement, Belady's anomaly

Discussion

6.3.1 Virtual Address Space

The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address, say, address 0 and exists in contiguous memory, as shown in Figure 6.3.1

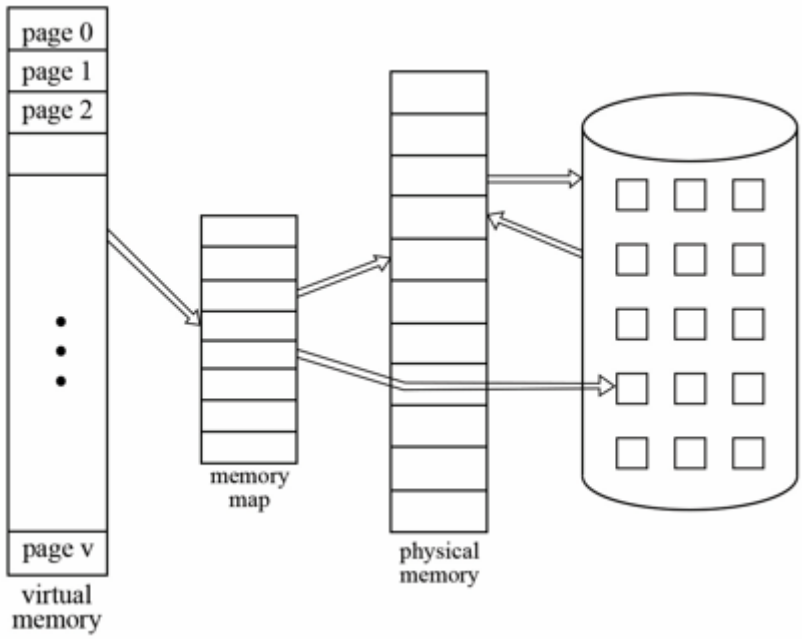


Fig 6.3.1 Diagram showing virtual memory, larger than physical memory

Physical memory can be organized into page frames, and these frames assigned to a process may not be contiguous. The memory-management unit (MMU) is responsible for mapping logical pages to physical page frames in memory.

In Figure 6.3.2, the heap is allowed to grow upward in memory with dynamic memory allocation, while the stack is allowed to grow downward in memory with successive function calls. The large blank space, or hole, between the heap and the stack, is part of the virtual address space. This space will only require actual physical pages if the heap or stack expands into it.

Virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:

- System libraries can be shared by several processes through the mapping

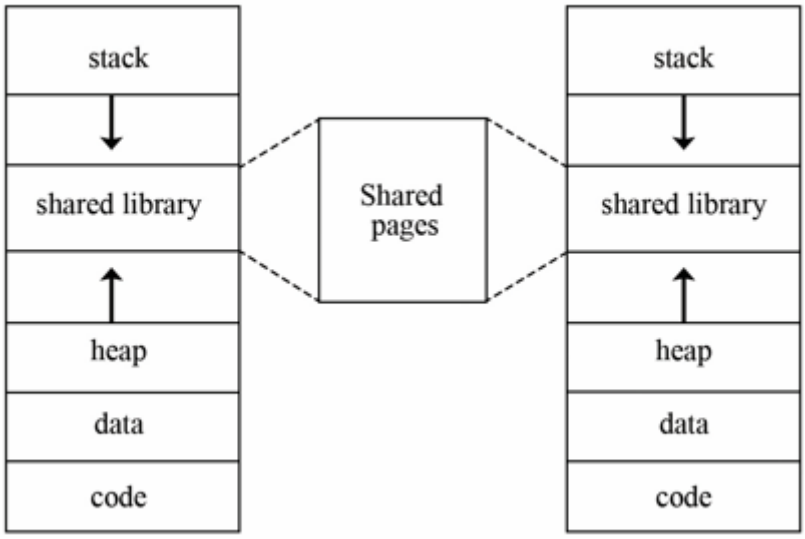


Fig 6.3.2 Shared library using virtual memory

of the shared object into a virtual address space. Although each process considers the libraries to be part of its virtual address space, all the processes share the actual pages where the libraries reside in physical memory. Typically, a library is mapped read-only into the space of each process that is linked with it.

- Similarly, processes can share memory. Two or more processes can communicate through the use of shared memory.
- Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared.
- Pages can be shared during process creation with the fork() system call, thus speeding up process creation.

6.3.2 Demand Paging

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether or not an option is ultimately selected by the user. An alternative strategy is to load pages only as they are needed. This technique is known as demand paging and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory.

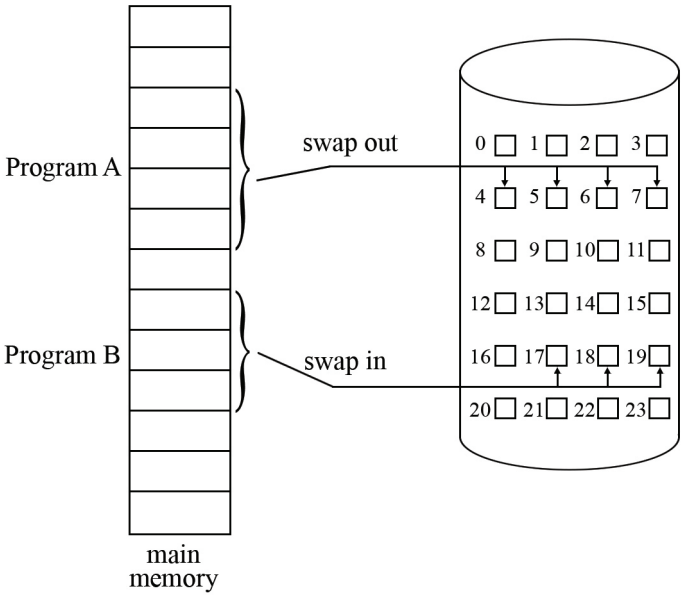


Fig 6.3.3 Transfer of a paged memory to contiguous disk space

A demand-paging system is similar to a paging system with swapping, where processes are stored in secondary memory, typically a disk, as illustrated in Figure 6.3.3. When we want to execute a process, it is swapped into memory. However, instead of swapping the entire process into memory, we use a lazy swapper. A lazy swapper only swaps a page into memory if it is needed. In a demand-paging system, the term "swapper" is technically incorrect. A swapper handles entire processes, while a pager deals with individual pages of a process. Therefore, in the context of demand paging, we use the term "pager" instead of "swapper."

6.3.2.1 Basic Concepts

When a process is to be swapped in, the pager predicts which pages will be used before the process is swapped out again. Instead of bringing the entire process into memory, the pager only loads the predicted pages. This approach avoids loading pages that won't be used, reducing swap time and the amount of physical memory required.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid-invalid bit scheme can be used for this purpose. This time, however, when this bit is set to “valid,” the associated page is both legal and in memory. If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is depicted in Figure 6.3.4.

Marking a page as invalid does not affect the process if it never tries to access that page. If the pager correctly predicts and loads only the needed pages, the process will run as if all pages were loaded into memory. While the process executes and accesses pages that

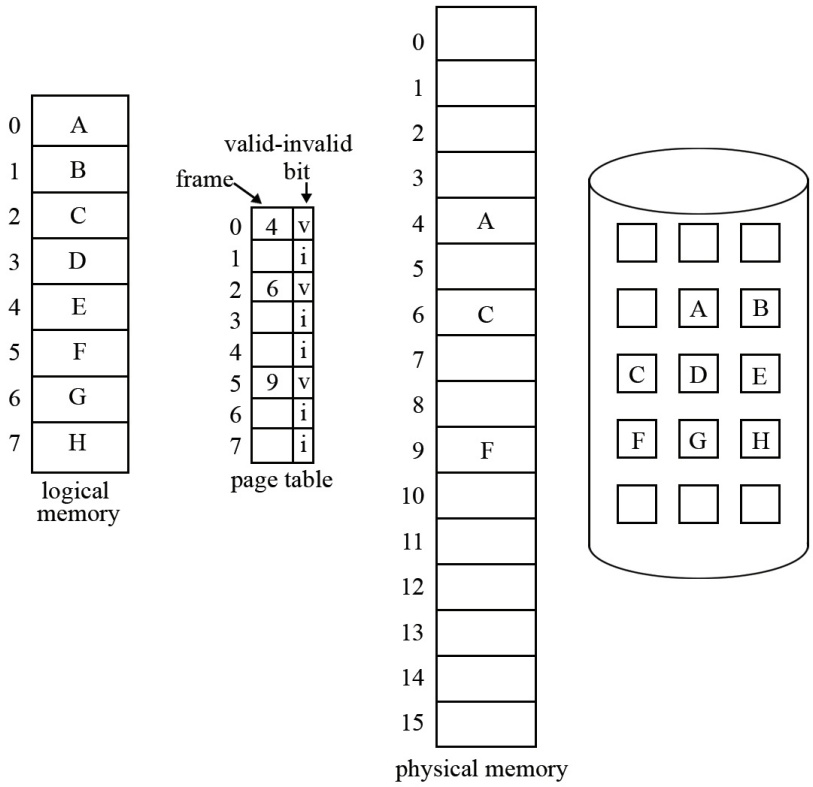


Fig 6.3.4 Page table when some pages are not in main memory

are in memory, it continues to run normally. This approach optimises memory usage and execution efficiency by only loading necessary pages.

If a process tries to access a page that is not in memory, it causes a page fault. The paging hardware, upon encountering the invalid bit in the page table, triggers a trap in the operating system. This trap occurs because the desired page was not loaded into memory. The operating system then handles the page fault by loading the required page into memory and updating the page table to reflect this change. Once the page is loaded, the process can resume normal execution. This procedure ensures that only necessary pages are loaded, optimizing memory usage. The procedure for handling a page fault, as illustrated in Figure 6.3.5, is explained below:

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or invalid memory access.

2. If the reference is invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.

3. We find a free frame (by taking one from the free-frame list, for example).

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. We restarted the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In extreme cases, we can start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is pure demand paging: never bring a page into memory until it is required.

Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behaviour is exceedingly unlikely. Programs tend to have a locality of reference, which results in reasonable performance from demand paging.

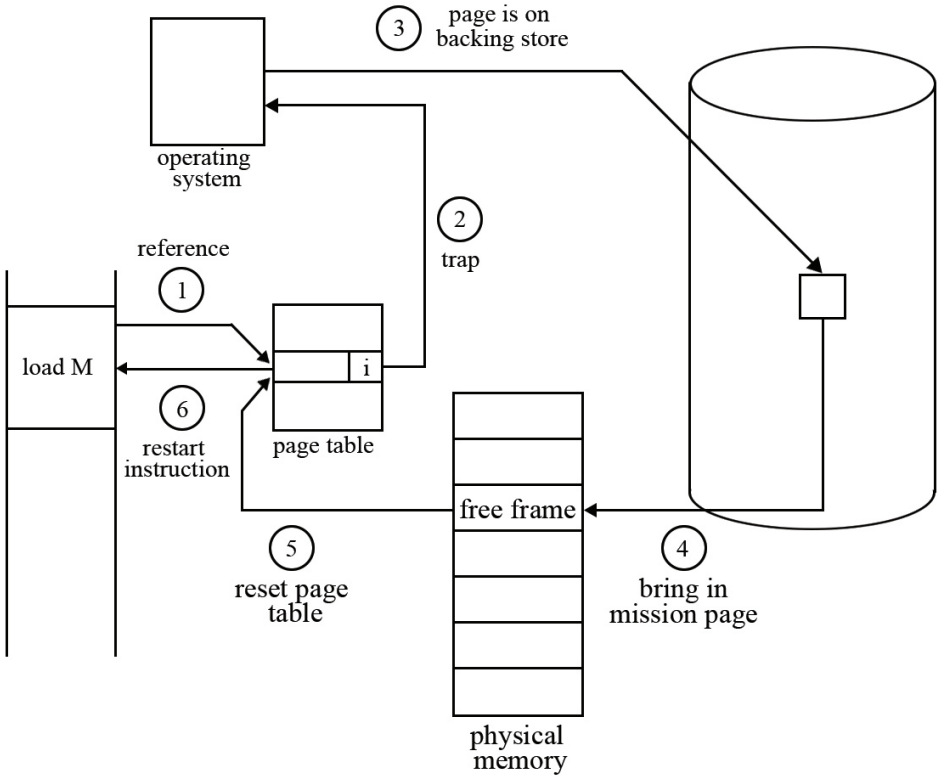


Fig 6.3.5 Steps in handling a page fault

The hardware to support demand paging is the same as the hardware for paging and swapping:

Page table. This table has the ability to mark an entry invalid through a valid-invalid bit or a special value of protection bits.

Secondary memory. This memory holds those pages that are not present in the main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the disk section used for this purpose is known as the swap space.

6.3.3 Page Replacement

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in.

6.3.3.1 Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory, as shown in Figure 6.3.6. We can now use the freed frame to hold the page for which the process is faulted. We modify the page-fault service routine to include page replacement:

- Find the location of the desired page on the disk.
- Find a free frame:
    - If there is a free frame, use it.
    - If there is no free frame, use a page-replacement algorithm to select a victim frame.
    - Write the victim frame to the disk; change the page and frame tables accordingly.
- Read the desired page into the newly freed frame; change the page and frame tables.
- Continue the user process from where the page fault occurred.

Notice that if no frames are free, two-page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a modified bit (or dirty bit). When this scheme is used, each page or frame has a modified bit associated with it in the hardware. The modified bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified.

When we select a page for replacement, we examine its modification bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this

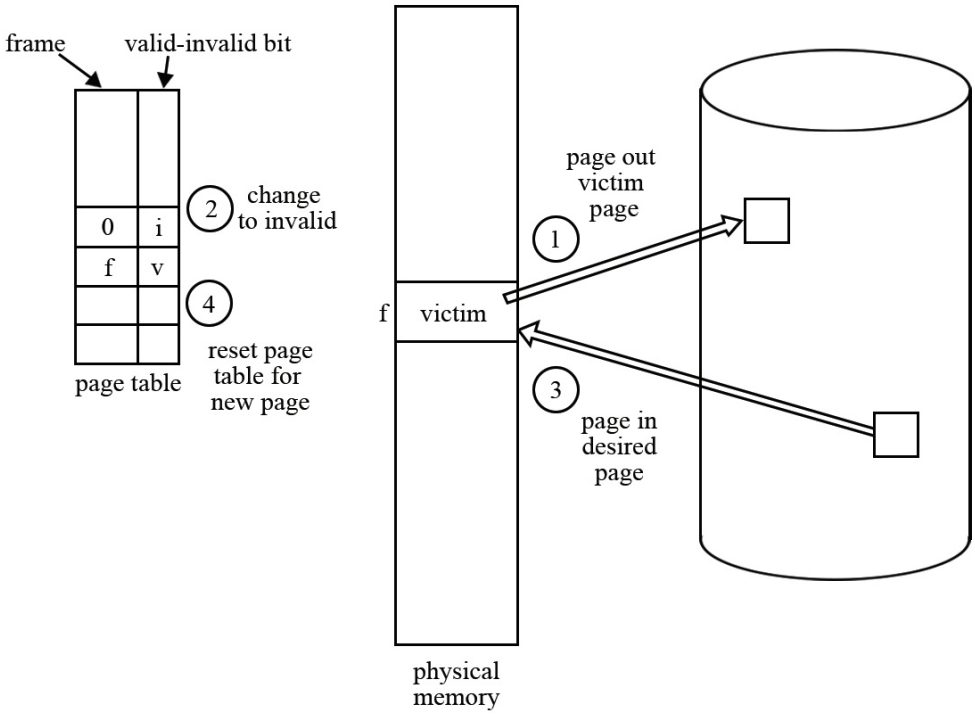


Fig 6.3.6 Page Replacement

case, we must write the page to the disk. If the modified bit is not set, however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault since it reduces I/O time by one-half if the page has not been modified.

We must solve two major problems to implement demand paging: we must develop a frame-allocation algorithm and a page-replacement algorithm. That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process, and when page replacement is required, we must select the frames that are to be replaced.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a reference string.

We next illustrate several page-replacement algorithms. In doing so, we use the reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.

6.3.3.2 FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7 because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in the replacement of page 0 since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 6.3.7. Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

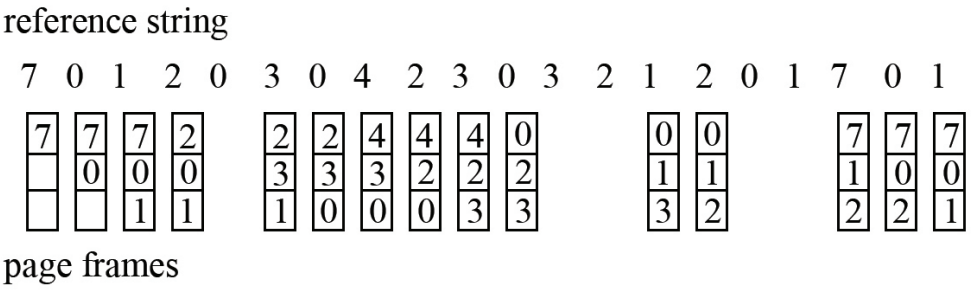


Fig 6.3.7 FIFO page-replacement algorithm

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.

Notice that even if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new one, a fault occurs almost immediately when retrieving the active page. Some other pages must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not, however, cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Figure 6.3.8 shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)! This most unexpected result is known as Belady's anomaly: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. We would expect that giving

more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

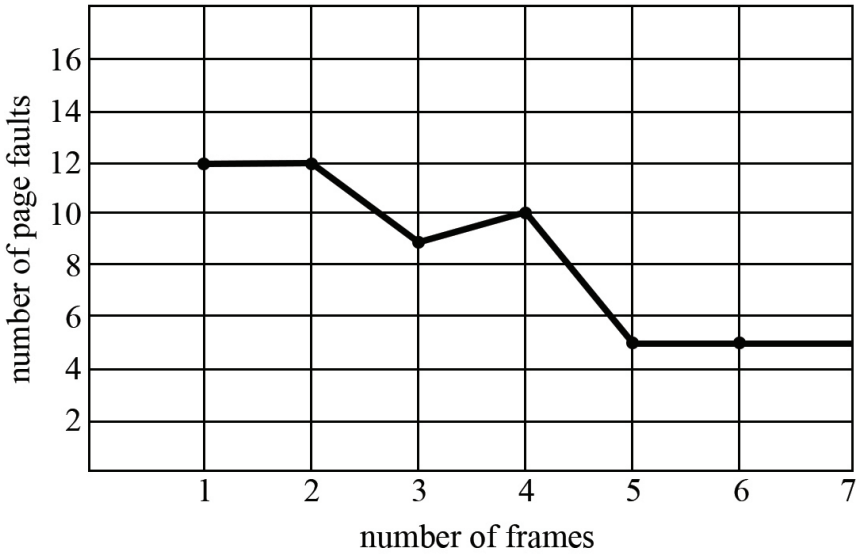


Fig 6.3.8 Page-fault curve for FIFO replacement on a reference string

6.3.3.3 Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm—the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN.

It is simply this:

Replace the page that will not be used for the longest period of time.

The use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine-page faults, as shown in Figure 3.9. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7 because page 7 will not be used until reference 18, whereas page 0 will be used at 5 and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine-page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies.

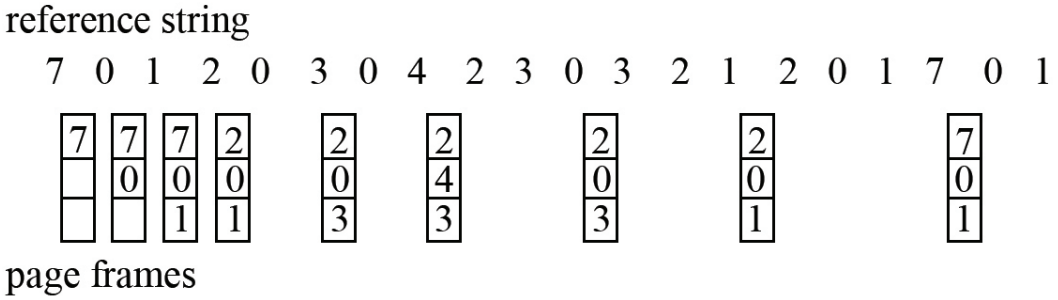


Fig 6.3.9 Optimal page-replacement algorithm

6.3.3.4 LRU Page Replacement

If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period. This approach is the least recently used (LRU) algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backwards in time rather than forward.

The result of applying LRU replacement to our example reference string is shown in Figure 6.3.10. The LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.

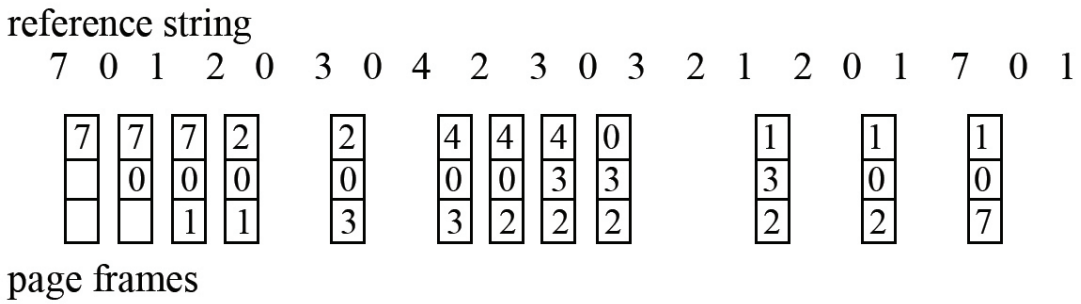


Fig 6.3.10 LRU page-replacement algorithm

Recap

- Virtual memory is a technique that enables us to map a large logical address space onto a smaller physical memory.
- Virtual memory allows us to run extremely large processes and to raise the degree of multiprogramming, increasing CPU utilization.
- Virtual memory is commonly implemented by demand paging.
- Pure demand paging never brings in a page fault until that page is referenced.
- If total memory requirements exceed the capacity of physical memory, then it may be necessary to replace pages from memory to free frames for new pages.
- FIFO page replacement is easy to program but suffers from Belady's anomaly.
- Optimal page replacement requires future knowledge.
- LRU replacement replaces the page that has not been used for the longest period of time.

Objective Type Questions

1. What is the concept in which a process is copied into the main memory from the secondary memory according to the requirement?
2. Where does swap space exist?
3. What occurs when a program tries to access a page that is mapped in address space but not loaded in physical memory?
4. Which page is replaced in the FIFO page replacement algorithm?
5. Which algorithm chooses the page that has not been used for the longest period of time whenever the page is required to be replaced?
6. What a virtual memory miss is called?
7. What memory is called the separation of user logical memory and physical memory?
8. What is the best page replacement algorithm?
9. Is logical Address space larger than physical address space?
10. How can virtual memory be implemented?

Answers to Objective Type Questions

1. Paging
2. Secondary memory
3. Page fault
4. Oldest page
5. least recently used algorithm
6. Page fault
7. Virtual memory
8. Optimal page replacement algorithm
9. Yes
10. Demand Paging

Assignments

1. Discuss the hardware support required to support demand paging.
2. What are the advantages of demand paging.
3. Explain virtual address space.
4. What is page fault? How it is affected in demand paging?
5. Explain FIFO page replacement.
6. What is Belady's anomaly?
7. Explain optimal page replacement.
8. Explain least recently used page replacement.

References

1. Silberschatz, Abraham, James L. Peterson, and Peter B. Galvin. Operating system concepts. Addison-Wesley Longman Publishing Co.

2. Operating system Concepts: Milan Malinkovic, TMH, 2nd ed.

3. Stallings, William. Operating Systems: Internals and Design Principles.

Suggested Reading

1. Silberschatz, Abraham, Peter Galvin, and Greg Gagne. Operating System Principles. 7th ed., John Wiley.

2. Tanenbaum, Andrew S., and Herbert Bos. Modern Operating Systems.



File allocation and Management

Learning Outcomes

After the completion of the unit, the learner will be able to:

- explain the attributes of files
- describe the details of the directory structure
- discuss different methods of file allocation

Prerequisites

Since main memory is usually too small to accommodate all the data and programs permanently, the computer system must provide secondary storage to back up main memory. Modern computer systems use disks as the primary online storage medium for information (both programs and data). The file system provides the mechanism for online storage and access to both data and programs residing on the disks. A file is a collection of related information defined by its creator. The files are mapped by the operating system onto physical devices. Files are normally organized into directories for ease of use.

The direct-access nature of disks gives us flexibility in the implementation of files. In almost every case, many files are stored on the same disk. In this unit we will discuss various attributes and properties of files, directories, how files are stored in a disk and how to allocate space for files in a disk.

Keyterms

File attributes, Directories, Contiguous allocation, Linked allocation, Indexed allocation

Discussion

6.4.1 Files

A file is a collection of related information that is recorded on secondary storage. In other words, a file is a collection of logically related entities. From the user's perspective, a file is the smallest allotment of logical secondary storage.

The name of the file has two parts, as shown below:

- Name
- extension, separated by a period

6.4.1.1 File attributes

A file's attributes vary across different operating systems, but they typically include the following:

1. **Name**: The symbolic file name, which is the only information stored in a human-readable format.
2. **Identifier**: A unique tag, usually a number, that identifies the file within the file system. This serves as the non-human-readable name for the file.
3. **Type**: Information about the file type, which is necessary for systems that support multiple file types.
4. **Location**: A pointer indicating the device and the file's location on that device.
5. **Size**: The current size of the file, measured in bytes, words, or blocks, and possibly the maximum allowed size.
6. **Protection**: Access-control information that determines who can read, write, execute, and perform other operations on the file.
7. **Time, Date, and User Identification**: Information on creation, last modification, and last use, which is useful for protection, security, and usage monitoring purposes.

6.4.2 Directories

The collection of files is a file directory. The directory contains information about the files, including attributes, location and ownership. Much of this information, especially that is concerned with storage, is managed by the operating system. The directory is itself a file, accessible by various file management routines.

The information contained in a device directory typically includes the following details:

1. **Name**: The file name.
2. **Type**: The type of file.
3. **Address**: The location of the file on the device.
4. **Current Length**: The current size of the file.
5. **Maximum Length**: The maximum allowed size of the file.
6. **Date Last Accessed**: The last date the file was accessed.
7. **Date Last Updated**: The last date the file was modified.
8. **Owner ID**: The identification of the file's owner.
9. **Protection Information**: Details regarding access control and permissions.

Operations performed on a directory typically include:

1. **Search for a File**: Locating a specific file within the directory.
2. **Create a File**: Adding a new file to the directory.
3. **Delete a File**: Removing a file from the directory.
4. **List a Directory**: Displaying the files and subdirectories contained within the directory.
5. **Rename a File**: Changing the name of an existing file.
6. **Traverse the File System**: Navigate through the directory structure to access files and subdirectories.

The advantages of maintaining directories are:

- Efficiency: A file can be located more quickly.
- Naming: It becomes convenient for users as two users can have the same name for different files or may have different names for the same file.
- Reduced redundancy.
- Grouping: Logical grouping of files can be done by properties, e.g. all Java programs, all games, etc.

6.4.3 Types of Directories

6.4.3.1 Single-level directories

In this, a single directory is maintained for all the users. An example of a single-level

directory is shown in Figure 6.4.1

The drawbacks of single-level directories are:

- Naming problem: Users cannot have the same name for two files.
- Grouping problem: Users cannot group files according to their needs.

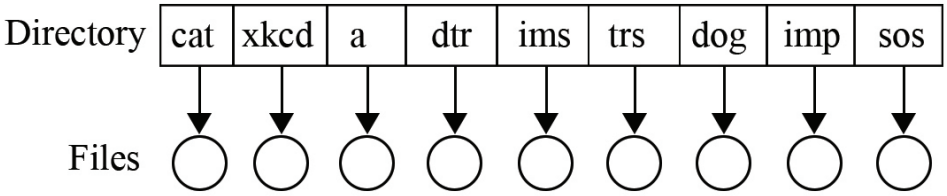


Fig 6.4.1 Single Level Directory

6.4.3.2 Two level directories

In this, separate directories for each user are maintained. An example of two level directory is shown in Fig. 6.4.2.

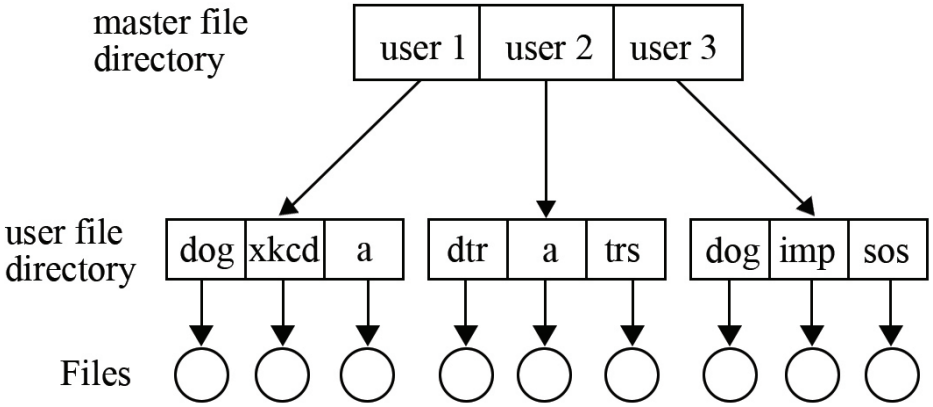


Fig 6.4.2 Two-level directory

- Path name: Due to two levels, there is a path name for every file to locate that file.
- Now, we can have the same file name for different users.
- Searching is efficient in this method.

6.4.3.3 Tree-structured directory

The directory is maintained in the form of a tree. Searching is efficient, and also there is a grouping capability. We have absolute or relative path names for a file, as shown in Fig 6.4.3.

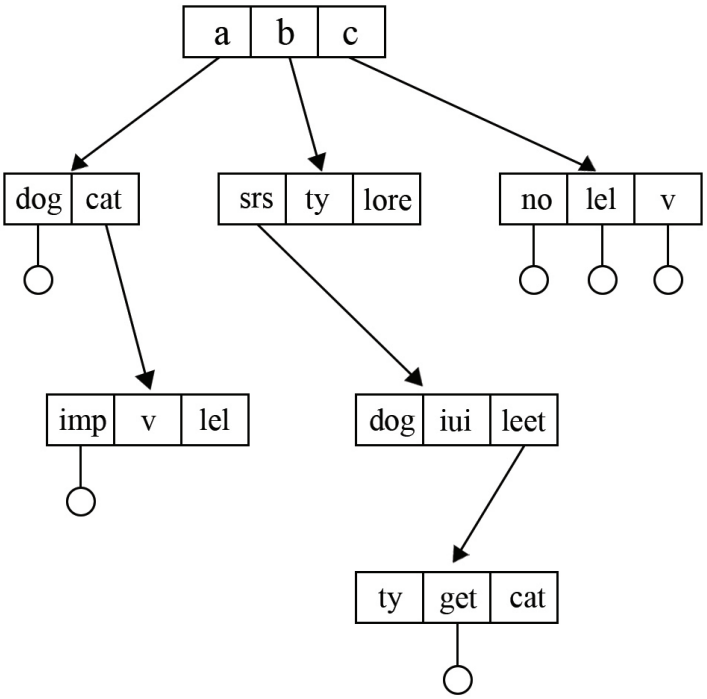


Fig. 6.4.3 Tree Structured directory

6.4.4 File Allocation

The direct-access nature of disks gives us flexibility in the implementation of files. In almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are: contiguous, linked, and indexed way of allocating disk space.

6.4.4.1 Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block b +1 after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head needs to move only from one track to the next. Thus, the number of disks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed.

The contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ..., b + n − 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file, as shown in Figure 6.4.4

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when

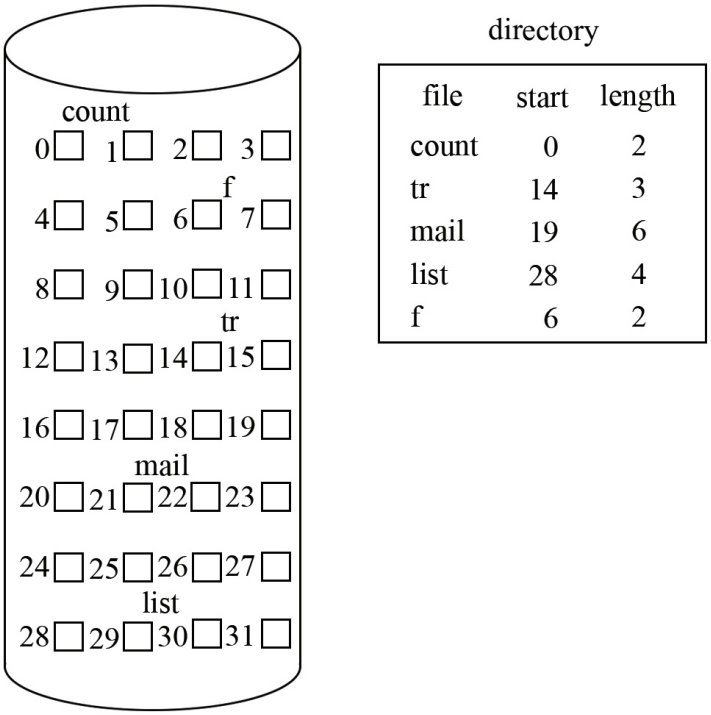


Fig 6.4.4 Contiguous allocation of disk space

necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block b + i. Thus, both sequential and direct access can be supported by contiguous allocation.

Disadvantages of contiguous allocation are:

- ♦ External fragmentation will occur, making it difficult to find contiguous blocks of space of sufficient length. A Compaction algorithm will be necessary to free up additional space on the disk.
- ♦ Also, with pre-allocation, it is necessary to declare the size of the file at the time of creation.

6.4.4.2 Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25, as shown in Figure 6.4.5. Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The

size field is also set to 0. A write-to-the-file method causes the free-space management system to find a free block, and this new block is written to and linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

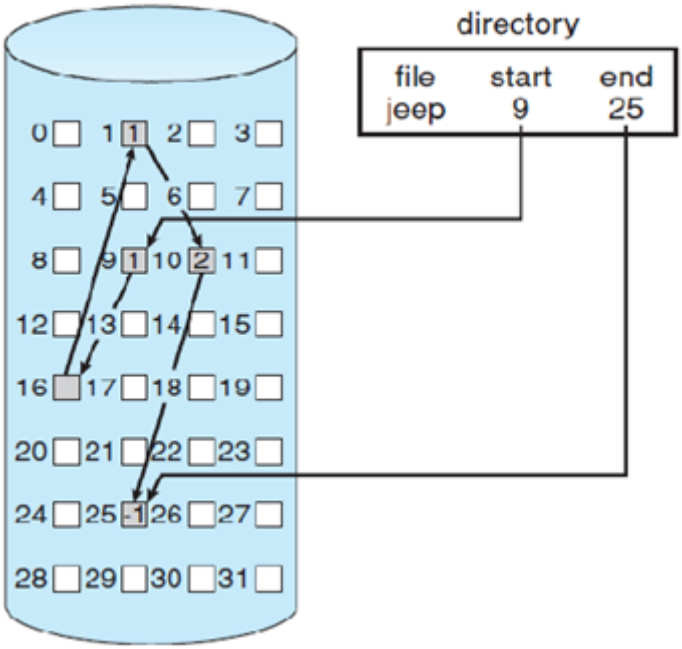


Fig 6.4.5 Linked Allocation of Disk Space

The disadvantages of linked allocation are:

- Internal fragmentation exists in the last disk block of the file.
- There is an overhead of maintaining the pointer in every disk block.
- If the pointer of any disk block is lost, the file will be truncated.
- It supports only the sequential access of files.

6.4.4.3 Indexed Allocation

Linked allocation solves the external fragmentation and size-declaration problems of contiguous allocation. However, linked allocation comes with a downside: it does not support efficient direct access to file blocks without a File Allocation Table (FAT). This inefficiency arises because the pointers to the blocks are distributed across the disk, interspersed with the data blocks, making it necessary to retrieve each pointer sequentially to access a specific block.

However, in the absence of a FAT, linked allocation cannot support efficient direct access since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. Indexed allocation solves this problem by bring-

ing all the pointers together into one location: the index block.

Each file has its own index block, which is an array of disk-block addresses. The i-th entry in the index block points to the i-th block of the file. The directory contains the address of the index block, as shown in Figure 6.4.6. To find and read the i-th block, we use the pointer in the i-th index-block entry.

When the file is created, all pointers in the index block are set to null. When the i-th block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.

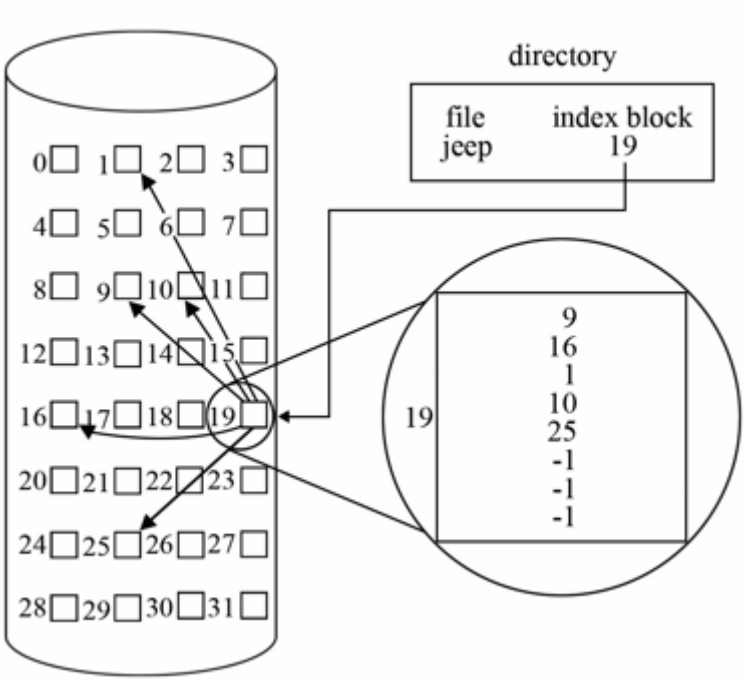


Fig. 6.4.6 Indexed allocation of disk space

Indexed allocation offers the significant advantage of supporting direct access without suffering from external fragmentation, as any free block on the disk can be utilised to satisfy a request for additional space. However, this method does entail a drawback: potential wasted space due to pointer overhead within the index block. This overhead is generally more substantial than that in linked allocation.

6.4.4.4 Pointer Overhead in Indexed Allocation

- **Small File Scenarios**: The inefficiency becomes particularly evident for files consisting of only one or two blocks. In linked allocation, each block includes only one pointer to the next block, thus minimising space wastage. Contrarily, indexed allocation necessitates allocating an entire index block, even if only one or two pointers within it will be used, leading to significant unused space.

6.4.4.5 Determining Index Block Size

The size of the index block is a critical factor that influences the efficiency of indexed allocation:

- **Small Index Blocks**:
  - **Advantages**: Minimizes wasted space for small files, as fewer pointers remain unused.
  - **Disadvantages**: Limited capacity to accommodate large files. If a file's size exceeds the capacity of a small index block, additional mechanisms are required to manage the overflow of pointers.
- **Large Index Blocks**:
  - **Advantages**: Adequately supports large files without necessitating additional structures.
  - **Disadvantages**: Increases wasted space for small files due to a higher number of unused pointers.

6.4.4.6 Advanced Indexing Techniques

To efficiently manage large files without resorting to excessively large index blocks, the following strategies can be implemented:

Linked Scheme

In a **linked scheme,** large files are divided into blocks, and these blocks are linked together using pointers. This method is a combination of linked allocation and indexed allocation. Here's how it works:

1. **Dynamic Linking of Index Blocks**: Instead of using one large index block, the linked scheme links multiple smaller index blocks together. Each index block contains pointers to data blocks and a pointer to the next index block in the sequence.
2. **Efficient Use of Space**: This method allows the file system to grow dynamically, accommodating large files without the need for a single, excessively large index block from the start. It uses additional index blocks only as needed.
3. **Scalability**: The chain of index blocks can expand as the file size increases, enabling efficient storage and retrieval of large files.
4. **Advantages**: This scheme avoids the limitations of fixed-size index blocks and reduces the amount of wasted space, as it allocates space only when necessary.

The linked scheme involves linking two or more index blocks together to store pointers. Each index block contains a pointer or address to the next index block. This method allows the file system to handle files that require more pointers than a single index block can accommodate. The chain of index blocks can grow dynamically as needed, supporting large files without requiring excessively large index blocks initially.

Multilevel Index

The multilevel index scheme uses a hierarchical approach to indexing. In this policy, a first-level index block points to second-level index blocks, which in turn point to the disk blocks occupied by the file. This method can be extended to three or more levels, depending on the maximum file size that needs to be supported. For instance:

- **First-level index block**: Contains pointers to second-level index blocks.
- **Second-level index block**: Contains pointers to actual data blocks or to third-level index blocks.
- **Third-level index block**: Contains pointers to actual data blocks.

This structure allows for efficient management of very large files by providing a scalable indexing system.

Combined Scheme

The combined scheme employs a special block called the Inode (information Node), which contains all the information about the file, such as its name, size, and permissions. The remaining space in the Inode is used to store disk block addresses that contain the actual file data. The goal is to balance between direct access and hierarchical/multilevel

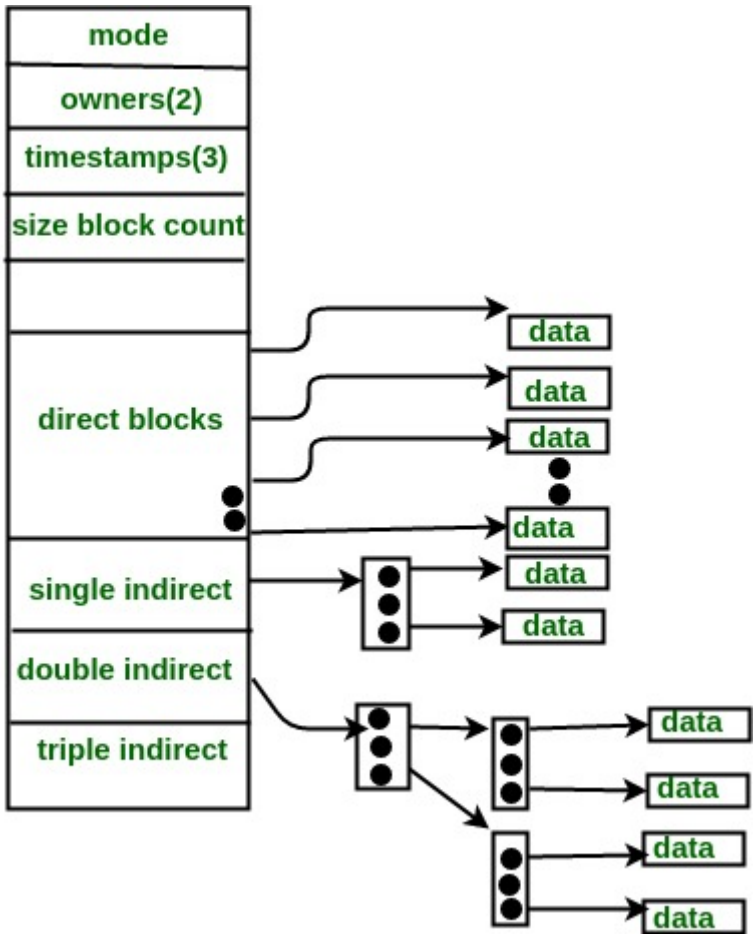


Fig 6.4.7 Advanced Indexing Technique

indexing to handle various file sizes effectively. Figure 6.4.7 shows Advanced Indexing Techniques (combined scheme)

- **Direct Blocks**: The first few pointers in the Inode point directly to the disk blocks containing the file data.
- **Single Indirect Block**: A pointer in the Inode points to a disk block that does not contain file data but contains addresses of the blocks that hold the file data.
- **Double Indirect Block**: A pointer in the Inode points to a disk block that contains addresses of single indirect blocks, which in turn contain addresses of the actual data blocks.
- **Triple Indirect Block**: A pointer in the Inode points to a disk block that contains addresses of double indirect blocks, which contain addresses of single indirect blocks, which then point to the actual data blocks.

Recap

- A file is a collection of related information that is recorded on secondary storage.
- The directory contains information about the files, including attributes, location and ownership.
- The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently.
- The various files can be allocated space on the disk in three ways: through contiguous, linked, or indexed allocation.
- Contiguous allocation can suffer from external fragmentation.
- Direct access is very inefficient with linked allocation.
- Indexed allocation may require substantial overhead for its index block.

Objective Type Questions

1. In which file allocation method must each file occupy a set of contiguous blocks on the disk?
2. In which file allocation method is each file a linked list of disk blocks?
3. In which file allocation method are all the pointers to scattered blocks placed together in one location?
4. In the linked allocation, the directory contains a pointer to which block?
5. In which file allocation method there is no external fragmentation?
6. In which file allocation method there is only sequential access?
7. Disadvantage of contiguous allocation.
8. Disadvantages of linked allocation

Answers to Objective Type Questions

1. Contiguous allocation
2. Linked allocation
3. Indexed allocation
4. First and last
5. Contiguous
6. Linked
7. External fragmentation
8. Internal fragmentation

Assignments

1. Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used to decide which strategy is best utilised for a particular file?
2. What is a file? Explain file allocations in memory.
3. Explain different types of directories.
4. Explain file allocation- contiguous and non-contiguous.
5. Explain different indexing scheme.
6. Explain advantages and disadvantages of linked allocation.

References

1. Silberschatz, Abraham, James L. Peterson, and Peter B. Galvin. Operating system concepts. Addison-Wesley Longman Publishing Co.
2. Operating system Concepts: Milan Malinkovic, TMH, 2nd ed.
3. Stallings, William. Operating Systems: Internals and Design Principles.

Suggested Reading

1. Silberschatz, Abraham, Peter Galvin, and Greg Gagne. Operating System Principles. 7th ed., John Wiley.
2. Tanenbaum, Andrew S., and Herbert Bos. Modern Operating Systems.

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: ……

Reg. No : .....

Name : …….....

MODEL QUESTION PAPER Set-I

End Semester Examination 2024

BACHELOR OF COMPUTER APPLICATION

BCA2021

B21CA03DC: COMPUTER ARCHITECTURE AND OPERATING SYSTEMS

(CBCS - UG)

2023-24 - Admission Onwards

Time: 3 Hours Max Marks: 70

Section A

Answer any ten questions. Each carries one mark (10x1=10 marks)

1. What is the primary function of the Control Unit in a computer?
2. Which addressing mode directly includes the operand as part of the instruction
3. What happens when a HALT instruction is executed?
4. Name the register typically used to hold an address for the memory unit.
5. What does DMA stand for in I/O systems?
6. Who is considered the father of vector processing and supercomputing?
7. What does SIMD stand for ?
8. "Cycle time of the processor is reduced" is one of the disadvantages of Pipelining". True/False
9. What is the first program that runs when a computer starts?
10. Which multiprocessor system contains a master slave relationship?
11. What does the PCB stand for?

12. What are the two atomic operations permissible on semaphores?

13. What is the purpose of the wait() operation in semaphores?

14. What does MMU stand for in the context of memory management?

15. What is the primary role of the operating system in memory management?

Section B

Answer any five questions. Each carries two marks (5x2=10 marks)

16. Explain the role of the Arithmetic and Logic Unit (ALU) in a computer.

17. Describe the purpose of the NOP instruction.

18. Describe what happens during the "Fetch Cycle" of micro-operations.

19. Mention the various phases in executing an instruction.

20. Mention the various types of pipelining.

21. Mention the different operating system services.

22. Explain the purpose of the "Program Counter" field in a PCB.

23. Explain the 'hold and wait' condition.

24. How does demand paging improve memory efficiency?

25. Explain the role of conditional branch instructions in program control.

Section C

Answer any five questions. Each carries four marks (4x5=20 marks)

26. Explain the difference between primary and secondary memory

27. Define addressing modes and explain why they are important in computer architecture.

28. Explain the difference between synchronous and asynchronous I/O.

29. What is pipelining? Define processor cycle in pipelining.

30. State the need for Instruction Level Parallelism.

31. Differentiate preemptive and non preemptive scheduling algorithms.

32. Explain the concept of deadlock with an example.

33. Describe the steps involved in the Banker's Resource-Request Algorithm.

34. How does the best-fit algorithm work for dynamic memory allocation?

35. Explain the difference between Direct and Indirect Addressing Modes.

Section D

Answer any two questions. Each carries fifteen marks (2x15=30 marks)

36. Explain the functional units of a computer according to the Neumann model. Discuss how each unit contributes to the overall functioning of a computer.

37. Describe the architecture of a common bus system using multiplexers, explaining how data is transferred between multiple registers and how control signals determine the selected register.

38. Consider the following table. And find average waiting time using the FCFS algorithm.

Process(P)	AT	BT
P1	0	4
P2	2	5
P3	3	3
P4	4	2
P5	5	1

39. Explain the various methods of Inter-Process Communication (IPC) in detail. Explain how the Dining Philosophers Problem demonstrates a deadlock situation.

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: ……

Reg. No : .....

Name : …….....

MODEL QUESTION PAPER Set-II

End Semester Examination 2024

BACHELOR OF COMPUTER APPLICATION

BCA2021

B21CA03DC: COMPUTER ARCHITECTURE AND OPERATING SYSTEMS

(CBCS - UG)

2023-24 - Admission Onwards

Time: 3 Hours Max Marks: 70

Section A

Answer any ten questions. Each carries one mark (10x1=10 marks)

1. What are the two types of control signals generated by the control unit?
2. What does the Program Counter (PC) keep track of?
3. What does the term "RTL" stand for in digital circuits?
4. How many bits are used in ASCII code?
5. Name two types of I/O interfaces.
6. What are uniprocessing computing devices called?
7. In which type of processing are several instructions executed simultaneously?
8. What is the name of a program in execution?
9. Which scheduler is invoked every time the CPU requires a new process for execution?
10. Which scheduler helps in swapping?
11. What does IPC stand for?

12. What does the signal() operation do in semaphores?

13. Name any one technique used for virtual memory management.

14. What does a page fault indicate?

15. What is the term for high paging activity?

Section B

Answer any five questions. Each carries two marks (5x2=10 marks)

16. What is the purpose of the Control Memory Address Register (CMAR) in a micro-programmed control unit?

17. What is the role of a multiplexer in a common bus system?

18. What is the purpose of a status command in an I/O interface?

19. What is meant by the pipeline bubble?

20. What is the Booting Process?

21. What happens during the fork() system call?

22. What is the purpose of a Page Table?

23. How does the compaction technique address external fragmentation?

24. Explain different system calls used for process control.

25. Describe the purpose of the NOP instruction.

Section C

Answer any five questions. Each carries four marks (4x5=20 marks)

26. What are the key components of a hardwired control unit? Briefly describe their functions.

27. Describe how RTL can be used to represent the transfer of data between two registers with an example.

28. How does Direct Memory Access (DMA) improve system performance in data transfer operations?

29. Compare UMA and NUMA multiprocessors.

30. Explain how the operating system performs process management.

31. What is time sharing operating system?

32. Illustrate the lifecycle of a process using a state diagram. Include all major states and transitions.

33. Explain the concept of logical and physical address space in memory management.

34. Discuss the various file allocation methods: contiguous, linked, and indexed. Include their advantages and disadvantages.

35. What is the significance of Indexed Addressing Mode, and how does it function?

Section D

Answer any two questions. Each carries fifteen marks (2x15=30 marks)

36. Evaluate Flynn's classification of parallel processing with necessary diagrams.

37. Analyze the role of the I/O interface in managing data transfer, control signals, data conversion, and synchronization between the CPU and peripherals, using examples from different types of I/O systems.

38. a) Compare the memory organization scheme paging and segmentation.

    b) Calculate the number of page faults for the following reference string with three-page frames using the following algorithms

    9,2,3,4,1,2,3,5,1,0,5,9,9,0,1

    i)FIFO ii)Optimal iii)LRU

39. Consider the following table. And find the average waiting time using the SJF algorithm.

Process(P)	AT	BT
P1	1	7
P2	2	5
P3	3	1
P4	4	2
P5	5	8

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura
Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in

COMPUTER ARCHITECTURE AND OPERATING SYSTEMS

COURSE CODE: B21CA03DC

ISBN 978-81-978764-5-5

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841