

Unit-1

-: Introduction to Embedded systems :-

- * our day-to-day life is becoming more and more dependent on "embedded systems" and digital techniques.
- * Embedded technologies are bonding into our daily activities even without our knowledge.
- * For example, in your vehicle itself the presence of specialised embedded systems vary from intelligent head lamp controllers, engine controllers and ignition control systems to complex air bag control systems to protect you in case of a severe accident.
- * people experience the power of embedded systems and enjoy the features and comfort provided by them.
- * Embedded systems, the various steps involved in their design and development and the major domains where they are deployed.

Definition of Embedded systems:

- * An ~~embed~~ Embedded system is an electronic/electro-mechanical system designed to perform a specific function and is combination of both hardware and software (firmware).
- * Every Embedded system is unique and the hardware as well as the firmware is highly specialized to the application domain.
- * Examples for Embedded systems are household appliances, tele communications, medical equipment, industrial control consumer products, etc.

Embedded systems vs General Computing systems

General Purpose Computing system	Embedded systems
1. A system which is a combination of a generic hardware and a General Purpose Operating System for executing a variety of applications.	1. A system which is a combination of special purpose hardware and embedded OS for executing a specific set of applications.
2. Contains General purpose Operating System (GPOS)	2. May or May not contain an operating system for functioning
3. Applications are alterable (programmable) by the user	3. The firmware of the embedded system is pre-programmed and it is non-alterable by the end-user
4. Performance is the key deciding factor in the selection of the system. Always, "Faster is Better".	4. Application-specific requirements (like performance, power requirements, memory usage, etc) are the key deciding factors.
5. Less/not at all tailored towards reduced operating power requirements, options for different levels of power management	5. Highly tailored to take advantage of the power saving modes supported by the hardware and the operating system.
6. Response requirements are not time-critical	6. For certain category of embedded systems like mission critical systems, the response time requirement is highly critical.

7. Need not be deterministic in execution behaviour

7. Execution behaviour is deterministic for certain types of Embedded systems like "Hard Real Time" systems.

History of Embedded Systems:

- * Embedded systems were built around the old Vacuum tube and the transistor technologies and the embedded algorithm was developed in low level languages.
- * Advances in semiconductor and nano-technology and IT revolution gave way to the development of miniature embedded systems.
- * The first recognised modern embedded system is the Apollo Guidance Computer (AGC) developed by the MIT Instrumentation Laboratory.
- * They ran the inertial guidance systems of both the Command Module (CM) and the Lunar Excursion Module (LEM).
 - 1) RAM size: 256, 1K, 2K words
 - 2) ROM size: 4K, 10K, 36K words
 - 3) clock frequency is 1.024 MHz
 - 4) 5000, 3-input RTL NOR gates are used.
 - 5) User interface is DSKY (display/keyboard)
- * The first-mass produced embedded system was the guidance computer for the Minuteman-I missile in 1961.
- * It was the 'Autonetics D-17' guidance computer, built using discrete transistor logic and a hard-disk for main memory.

Classification of Embedded Systems

* The classification of Embedded Systems are based on:

- 1) Based on generation
- 2) Complexity and performance requirements
- 3) Based on deterministic behaviour
- 4) Based on triggering.

* The classification based on deterministic behaviour is applicable for 'Real Time' systems.

* The application/task execution behaviour for an embedded system can be either deterministic or non-deterministic

* Based on execution behaviour, Real Time Embedded systems are classified into Hard and soft.

* Embedded systems which are 'reactive' in nature, can be classified based on the trigger. Reactive system can be either event triggered or time triggered.

i) Based on generation

The classification of Embedded systems based on generation are:

First Generation: * The early embedded systems were built around 8 bit microprocessors like 8085 and 780. and 4 bit microcontrollers.

* Simple in hardware circuits with firmware developed in Assembly Code.

* Digital telephone keypads, stepper motor control units etc. are examples.

Second Generation:

* These are embedded systems built around 16 bit microprocessors and 8 or 16 bit microcontrollers, following the 1st generation embedded systems.

- * The instruction set for 2nd generation processors/controllers were much more complex and powerful.
- * Data Acquisition Systems, SCADA systems etc are examples of 2nd generation embedded systems.

Third Generation:

* Embedded Systems built around high performance 16/32 bit Microprocessors/microcontrollers.

* A new concept of application and domain specific processors/controllers like Digital signal processors (DSP) and Application Specific Integrated Circuits (ASICs) came.

* The instruction set of processors became complex and powerful and the concept of instruction pipelining also evolved.

* Robotics, ~~Process~~ industrial process control, networking etc are the examples.

Fourth Generation:

* Embedded systems built around System on chips (SoC's) Reconfigurable processors and multicore processors.

* It brings high performance, tight integration and miniaturization into the embedded device market.

* ex: Smart phone devices, MP3 etc

(ii) Classification Based on Complexity and Performance:

* The classification is based on the Complexity and system performance requirements. According to this classification, embedded systems can be grouped into:

Small-Scale Embedded systems:

- * The embedded systems are built around low performance and low cost 8 or 16 bit microprocessors / micro controllers.
- * A small-scale embedded systems may or may not contain an operating system for its functioning.
- * It is suitable for simple applications and where performance is not time critical.

Medium-Scale Embedded systems:

- * Medium-scale Embedded systems are built around medium performance, low cost 16 or 32 bit microprocessors / micro controllers or digital signal performance.
- * There are slightly complex in hardware and firmware.
- * It may contain either general purpose or real time operating system for functioning.

Large-Scale Embedded systems / Complex systems:

- * Embedded systems built around high performance 32 or 64 bit RISC processors / controllers or Reconfigurable System on Chip (RSOC) or multi-core processors and programmable logic devices.
- * These systems involve highly complex hardware and firmware requirements.
- * They may contain multiple processors / controllers and co-units / hardware accelerators for offloading the processing

requirements from the main processor of the system.

- * Decoding/encoding of media, Cryptographic function implementation etc are examples for processing requirements which can be implemented using a co-processor/hardware accelerators.

- * Complex embedded systems usually contain a high performance Real Time Operating System (RTOS) for task scheduling, prioritization and management.

(ii) Based on deterministic behaviour:

- * It is applicable for Real Time Systems. The application/task execution behaviour for an embedded system can be either deterministic or non-deterministic.

- * These are classified into two types.

Soft Real time systems: Missing a deadline may not be critical and can be tolerated to a certain degree

Hard Real time systems: Missing a program/task execution time deadline can have catastrophic consequences (financial, human loss of life etc)

(iv) Based on triggering

- * These are classified into two types

Event triggered: Activities within the system (ex: task run-times) are dynamic and depend upon occurrence of different events.

Time triggered: Activities within the system follow a statically computed schedule (i.e; they are allocated time slots during which they can take place) and thus by nature are predictable.

Major application Areas:

The major application areas of Embedded systems are listed below:

1. Consumer electronics: Camcorders, Cameras etc.
2. Household appliances: Television, DVD players, washing machine, fridge, microwave oven etc.
3. Home automation and security systems: Air conditioners, sprinklers, intruder detection alarms, closed circuit television cameras, fire alarms etc.
4. Automotive industry: Anti-lock braking systems (ABS), engine control, ignition systems, automatic navigation system etc.
5. Telecom: Cellular telephones, telephone switches, handset multimedia applications etc.
6. Computer peripherals: Printer, scanner, fax machines etc.
7. Computer networking systems: Network routers, switches, hubs, firewalls etc.
8. Healthcare: Different kinds of scanners, EEG, ECG machines etc.
9. Measurement and Instrumentation: Digital multimeters, digital CROs, logic analyzers, PLC systems etc.
10. Banking & Retail: Automatic teller machines (ATM) and currency counters, Point of Sales (POS).
11. Card Readers: Barcode, Smart card Readers, handheld devices etc.

Purpose of Embedded Systems:

* Each Embedded system is designed to serve the purpose of anyone or a combination of the following tasks:

1. Data Collection / storage / Representation
2. Data Communication

3. Data (signal) processing

4. Monitoring

5. Control

6. Application Specific User Interface.

Data Collection / Storage / Representation:

* Embedded systems designed for the purpose of data collection performs acquisition of data from the external world.

* Data collection is usually done for storage, analysis, manipulation and transmission.

* Data can be either analog (Continuous) or digital (discrete)

* The collected data may be stored directly in the system or it may be transmitted to some other systems or it may be processed by the system or it may be deleted ~~and~~

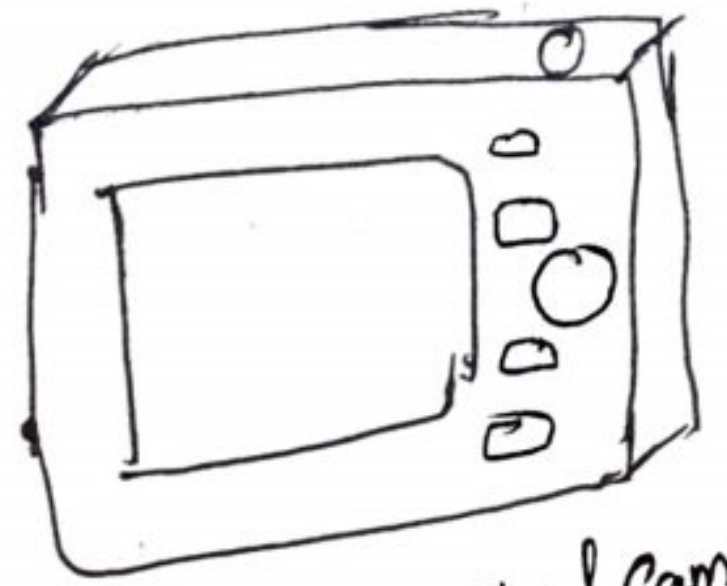
instantly after giving a meaningful representation.

Data Collection Communication

* Embedded data communication systems are deployed in applications ranging from complex satellite communication systems to simple home networking systems

* The data collected by an embedded terminal may require transferring of the same to some other system located remotely.

* The transmission is achieved either by a wire-line medium or by a wire-less medium.



Ex Digital Camera



Ex Router

- * Data can either be transmitted by analog means or by digital means.
- * The data collecting embedded terminal itself can incorporate data communication units like wireless modules (Bluetooth, ZigBee, Wi-Fi, EDGE, GPRS etc) or wire-line modules (RS-232, USB, TCP/IP, PS2 etc).
- * Network hubs, routers, switches etc are typical examples of dedicated data transmission embedded systems.

Data (Signal) Processing:

- * Embedded systems with signal processing functionalities are employed in applications demanding signal processing like speech coding, synthesis, audio video Codec, transmission applications etc.
- * A digital hearing aid is a typical example of an embedded system employing data processing.
- * Digital hearing aid improves the hearing capacity of hearing impaired persons.

Monitoring:

- * Embedded systems falling under this category are specifically designed for monitoring purpose.
- * They are used for determining the state of some variable using input sensor. They can't improve or impose control on variables.
- * The electro Cardiogram machine is an example for embedded system for monitoring the heartbeat of a patient.
- * The sensors used in ECG are the different electrodes connected to the patient's body.
- * Measuring instruments like digital CRO, digital multimeters,

logic analyzers, etc used in Control & instrumentation applications

- * They are used for knowing (monitoring) the status of some variables like current, voltage etc.

Control

- * Embedded systems with control functionalities impose control over some variables according to the changes in i/p variables.
- * A system with control functionality contains both sensors and actuators.
- * Air conditioner system used in our home to control the room temperature to a specified limit is a typical example for an embedded system for control purpose.
- * An air-conditioner contains a room temperature-sensing element (sensor) which may be a thermistor and a handheld unit for setting up (feeding) the desired temperature.
- * The air compressor unit acts as the actuator. The compressor unit is controlled according to the current room temperature and desired temperature set by the end user.

Application Specific User Interface :

- * These are embedded systems with application-specific user interfaces like buttons, switches, keypad, lights, bells, display units etc. Mobile phones is an example for this.
- * In mobile phone the user interface is provided through the keypad, graphic LCD module, system speaker, vibration alert, etc.

Purpose of Embedded systems:

Characteristics and Quality Attributes of Embedded system

a) characteristics of Embedded system

* Embedded systems possess certain specific characteristics and these characteristics are unique to each embedded system. Some of the important characteristics of an embedded system are:

1. Application and domain specific
2. Reactive and Real Time
3. Operates in harsh environments
4. Distributed
5. Small size and weight
6. Power Concerns.

Application and domain specific:

* Embedded system is having certain functions to perform and they are developed in such a manner to do the intended functions only. They can't be used for any other purpose.

Ex: The embedded control units of the microwave oven can't be replaced with AC's embedded control units of microwave oven and AC are specifically

Reactive and Real Time:

* Embedded systems are in constant interaction with the real world through sensors and user-defined input devices which are connected to the input port of the system.

* Any changes happening in the world are captured by the sensors or input devices in the Real Time and the control

algorithm running inside the unit reacts in a designed manner to bring the controlled output variables to the desired level.

- * Embedded systems produce changes in o/p in response to the changes in the input. So they are generally referred as "Reactive systems".
- * Real Time System operation means the timing behavior of the system should be deterministic; meaning the system should respond to requests or tasks in known amount of time.
- * Embedded applications or systems which are mission critical, like flight control systems, Antilock Brake system (ABS) are examples of Real Time systems.

Operates in Harsh Environment:

* The design should take care of the operating conditions of the area where the system is going to implement

* For Example, if the system needs to be deployed in a high temperature zone, then all the components used in the system should be of high temperature grade.

* Here we can't go for a compromise in cost. Also proper shock absorption techniques should be provided to systems which are going to be ~~compromised~~ commissioned in places subject to high shock.

Distributed:

* The term distributed means of embedded systems may be a part of larger systems.

* Many members of such distributed embedded systems form a single large embedded control unit.

* An automatic Vending machine is a typical example for this.

- * The vending machine contains a card reader (for pre-paid vending systems), a vending unit etc.
- * Each of them are independent embedded units but they work together to perform the overall vending function.

Small size and weight

- * Product aesthetics is an important factor in choosing a product.
- * It is convenient to handle a compact device than a bulky product.
- * The Product aesthetics (size, weight, shape, style etc) will be one of the deciding factors to choose a product.

Power Concerns:

- * Power management is another important factor that needs to be considered in designing embedded system.
- * Embedded systems should be designed in such way so that to minimize the heat dissipation by the system.
- * The production of high amount of heat demands cooling requirements like cooling fans which in turn occupies additional space and make the system bulky.

b) Quality Attributes of Embedded system:

- * Quality Attributes are the non-functional requirements that need to be documented properly in any system design.
- * The various quality attributes that needs to be addressed in any embedded system developed broadly classified into two namely 'operational Quality Attributes' and 'Non-operational Quality Attributes'.

Operational Quality Attributes:

* The operational quality attributes represent the relevant quality attributes related to the embedded system when it is in the operational mode or 'online mode'.

* The important quality attributes coming under this category are listed below,

- a) Response
- b) Throughput
- c) Reliability
- d) Maintainability
- e) Security
- f) Safety.

a) Response: * Response is a measure of quickness of the system.

* It gives you an idea about how fast your system is tracking the changes in input variables.

* Most of the embedded systems demand fast response which should be almost Real Time.

* Ex: An Embedded system deployed in flight control application should respond in a Real Time manner.

b) Throughput: * Throughput deals with the efficiency of a system.

* In general it can be defined as the rate of production or operation of a defined process over a stated period of time.

* The rates can be expressed in terms of units of products, batches produced, or any other meaningful measurements.

* In case of a Card Reader, throughput means how many transactions the Reader can perform in a minute or in a hour or in a day.

* Throughput is generally measured in terms of 'Benchmark'.

* A 'Benchmark' is a reference point by which something can be measured.

c) Reliability: * Reliability is a measure of how much percentage you can rely upon the proper functioning of the system or what is the percentage susceptibility of the system to failure.

* Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR) are the terms used in defining the system reliability.

* MTBF gives the frequency of failures in hours/weeks/months.

* MTTR specifies how long the system is allowed to be out of order following of failure

* For an embedded system with critical application need, it should be of the order of minutes.

d) Maintainability: * Maintainability deals with support and maintenance to the end user or client in case of technical issues and product failures or on the basis of a routine system Checkup.

* Reliability and maintainability are considered as two complementary disciplines.

* A more reliable system means a system with less corrective maintainability requirements and vice versa.

* Maintainability can be broadly classified into two categories. They are

1) scheduled or periodic Maintenance

2) Maintenance to unexpected failures.

e) Security: * Confidentiality, 'Integrity' and 'Availability' are the three major measures of Information security.

* Confidentiality deals with the protection of data and application from unauthorised disclosure.

- * Integrity deals with the protection of data and application from unauthorised ~~dis~~ modification.
- * Availability deals with protection of data and application from unauthorised users.

f) Safety: * Safety deals with the possible damages that can happen to the operators, public and the environment due to be the breakdown of an embedded system or due to the emission of the radioactive or hazardous materials from the embedded products.

- * The breakdown of an embedded system may occur due to hardware failure or a firmware failure.
- * Safety analysis is a mult product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of the damages to an acceptable level.
- * Some of the safety threats are sudden (like product break - down) and some of them are gradual (like hazardous emissions from the product).

Non-operational ~~Quality~~ Quality Attributes:

The important quality attributes coming under this category are listed below

- Testability and Debug-ability
- Evolvability
- Portability
- Time to Prototype and market
- Per unit and total cost.

a) Testability and Debug-ability:

- * Testability deals with how easily one can test his/her design, application and by which means she/he can test it.
- * For an embedded product, testability is applicable to both the embedded hardware and firmware.
- * Embedded hardware testing ensures that the ~~firmware~~ peripherals and the total hardware functions in the desired manner, whereas the firmware testing ensures that the firmware is functioning in the expected way.
- * Debug-ability is means of debugging the product as such for figuring out the probable source the create unexpected behaviour in the total system.
- * Debug-ability has two aspects. They are
 - 1) hardware debugging is used for figuring out the issues created by hardware problems.
 - 2) Firmware debugging is employed to figure out the probable errors that appears as result of flaws in the firmware.

b) Evolvability:

- * Evolvability is a term which is related to Biology.
- * It is referred as the non-heritable variation.
- * For an embedded system, the quality attribute 'Evolvability' refers to the ease with which the embedded product can be modified to take advantage of new firmware or hardware technologies.

c) Portability:

- * Portability is a measure of "system independence". An

Embedded product is said to be portable if the product is capable of functioning 'as such' in various environments, target processors/controllers and embedded operating systems.

- * In embedded products, the products are always flexible and portable. The term 'porting' represents the migration of the embedded firmware written for one target processor to a different target processor.

d) Time-to-prototype and Market:

- * Time-to-market is the time elapsed between the conceptualisation of a product and the time at which the product is ready for selling or use.

- * The commercial embedded product is highly competitive and time to market the product is a critical factor in the success of a commercial embedded product.
- * There may be multiple players in the embedded industry who develop products of the same category.

e) Per unit Cost and Revenue:

- * Cost is a factor which is closely monitored by both end user and product manufacturer.

- * Cost is highly sensitive factor for commercial products.

- * Any failure to position the cost of a commercial product at a nominal rate, may lead to failure of product in the market.

- * Proper market study and cost benefit analysis should be carried out before taking a decision on the per unit cost of the embedded product.

- * From a designer/product development company perspective the ultimate aim of the product is to generate the marginal profit.

Unit - II

Typical Embedded system

Elements of the Embedded system :-

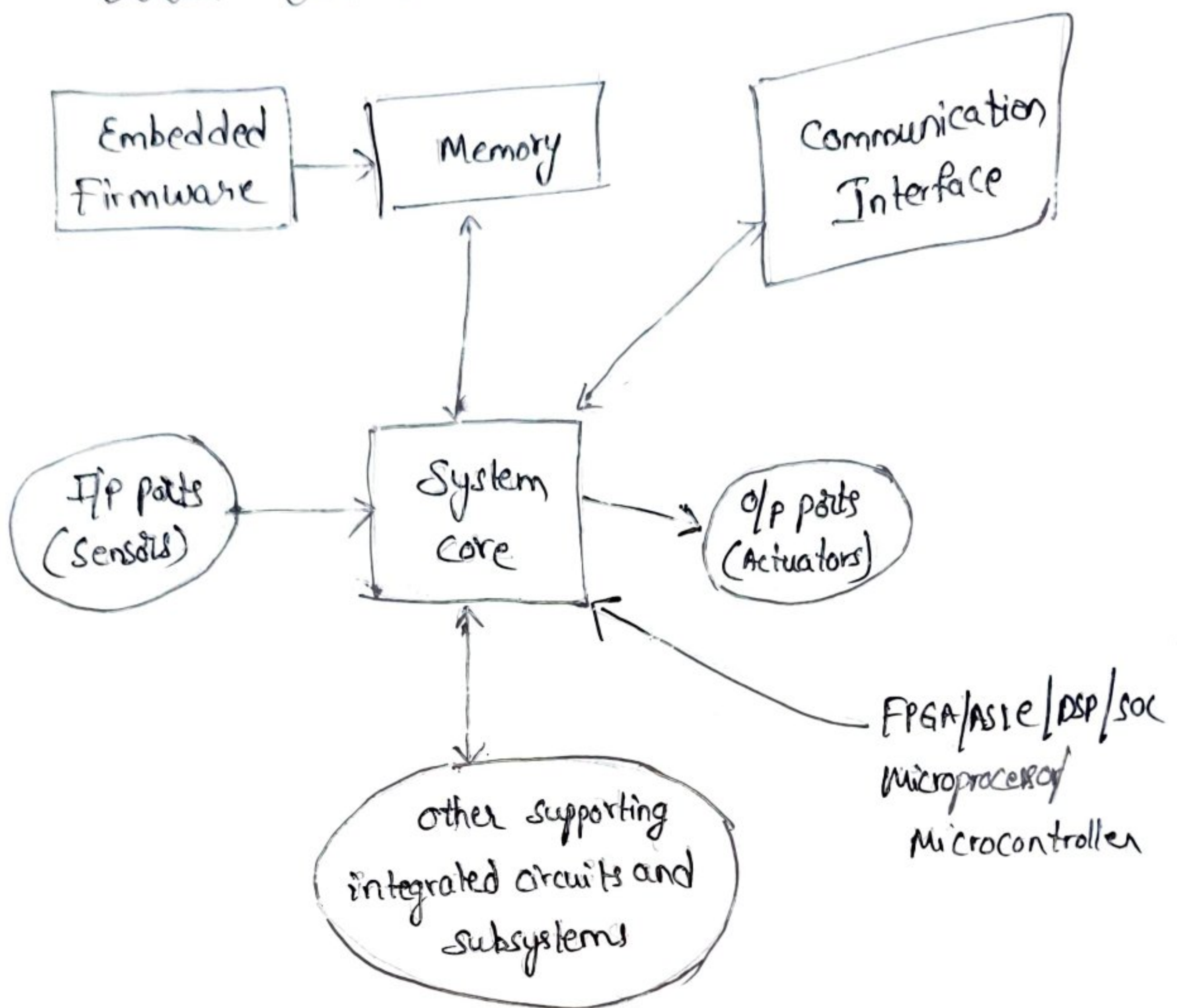


Fig: Elements of the Embedded system

- * An embedded system is a combination of 3 things. Hardware, software Mechanical Components and it is supposed to do one specific one task.
- * A typical embedded system contains a single chip controller, which acts as the master brain of the system.
- * The Controller can be Microprocessor or a Microcontroller or a Field programmable Gate Array or a Digital signal processor or an Application specific Integrated circuits.

- * Embedded hardware/software systems are basically designed to regulate a physical variable or to manipulate the state of some devices by sending some signals to the Actuators or devices connected to the o/p ports of the system, in response to the i/p signal provided by users or sensors which are connected to i/p ports.
- * Embedded system can be viewed as a reactive system.
- * The control is achieved by processing the information coming from the sensors and user interface; and controlling some actuators that regulate the physical variable.
- * Keyboards, pushbutton switches etc are examples for common user interface input devices.
- * LEDs, Liquid Crystal displays, buzzers etc are exmples of common user interface output devices.
- * Some embedded systems don't require any manual intervention for their operation.
- * They automatically sense the variations in the i/p parameters in accordance with the changes, which they are interacting through the sensors which are connected to the i/p port of the system.
- * The sensor information is passed to the processor after signal conditioning and digitisation. The core of the system performs some predefined operations on i/p data with the help of embedded firmware in the system and sends some actuating signals to the actuator ~~to~~ connected to the o/p port of the system.
- * The memory of the system is responsible for holding the Code. There are two types of memories are used in any

embedded system.

- * Fixed memory (ROM) is used for storing code or program. The user can't change the firmware in this type of memory.
- * The most common types of memories used in embedded systems for control algorithm storage are OTP, PROM, UVEPROM, EEPROM and FLASH.
- * An embedded system without code (i.e. the control algorithm) implemented memory has all the peripherals but is not capable of making decision depending on the situation as well as real world changes.
- * Memory for implementing the code may be present on the processor or may be implemented as a separate chip interfacing the processor.
- * In a Controller based embedded system, the controller may contain internal memory for storing code such controllers are called Micro-controllers with on-chip ROM, eg Atmel AT89C51.

Core of the Embedded System:

- * Embedded systems are domain and application specific and are built around a central core.
- * The core of the embedded system falls into any one of the following categories:
 1. General Purpose and Domain specific processors
 - a) Microprocessors
 - b) Micro Controller
 - c) Digital Signal processors
 2. Application Specific Integrated Circuits (ASIC)
 3. Programmable Logic Devices (PLDs)

4. Commercial off-the-shelf Components (COTS)

General Purpose and Domain Specific Processors:

* Almost 80% of the embedded systems are processor/controller based. The processor may be a microprocessor (or) a microcontroller (or) a digital signal processor, depending on the domain and application.

Microprocessors

- * A microprocessor is a silicon chip representing a central processing unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of instructions, which is specific to the manufacturer.
- * In general, the CPU contains the Arithmetic & Logic Unit (ALU), Control unit (CU) and working registers.
- * Microprocessor is a dependant unit, and it requires the combination of other hardware like Memory, Timer Unit, and Interrupt Controller etc for proper functioning.
- * Intel claims the credit for ~~the~~ developing the 1st Microprocessor unit Intel 4004, a 4 bit processor which was released in Nov 1971.

Developers for Microprocessors

- a) Intel - Intel 4004 - Nov '71 (4-bit)
- b) Intel - Intel 4040
- c) Intel - Intel 8008 - Apr 1972
- d) Intel - Intel 8080 - Apr 1974 (8-bit)
- e) Motorola - Motorola 6800
- f) Intel - Intel 8085 - 1976
- g) Ziolog - Z80 - July 1976

Microcontroller:

* A Microcontroller is a highly integrated chip that contains a CPU, scratch pad RAM, special and general purpose register arrays, on chip ROM/FLASH memory for program storage, timer and interrupt control units and dedicated

I/O ports.

* Microcontrollers can be considered as a superlat of microprocessors. Since a μ C contains all the necessary functional blocks for independent working, they found greater place in the embedded domain in the place of μ P.

* They are cheap, cost effective and readily available in the market.

* Ex: Texas Instrument's TMS 1000 is considered as the world's first μ C.

Difference between Microprocessor & Microcontroller

Microprocessor	MicroController
1) A Silicon chip representing a central processing unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of instructions.	1: A microcontroller is a highly integrated chip contains a CPU, scratch pad RAM, special & general purpose register arrays, on chip ROM/FLASH memory for program storage, timer and interrupt control units & dedicated I/O ports.
2) It is dependent unit. It requires the combination of other chips like timers, program & data memory chips, interrupt controllers etc for functioning.	2) It is self-contained unit and it doesn't require external external interrupt controller, timer, UART etc for its functioning.

3. Most of the time general purpose in design and operation.	3. Mostly application-oriented or domain specific.
4. Doesn't contain a built-in I/O port. The I/O port functionality needs to be implemented with the help of external programmable peripheral interface chips like 8255.	4. Most of the processors contain multiple built-in I/O ports which can be operated as a single 8 or 16 or 32 bit port or as individual port pins.
5. Targeted for high end market where performance is important.	5. Targeted for embedded market where performance is not so critical.
6. Limited power saving options compared to microcontrollers.	6. Includes lot of power saving features.

General purpose Processor (GPP) Vs Application-Specific Instruction Set Processor (ASIP):

- * A General purpose processor (GPP) is a processor, designed for general Computational tasks.
- * They are produced in large volumes and targeting the general market. Due to the high volume production, the per unit cost for a chip is low compared to ASIC or other specific ICs.
- * A typical general purpose processor contains an Arithmetic and logic unit (ALU) and Control unit (CU).
- * Application Specific Instruction Set processors (ASIPs) are processors with Architecture and instruction set optimised to specific-domain/application requirements like network processing, automotive, telecom, media applications, digital signal processing, Control applications, etc.

- * ASIPs fill the architectural spectrum b/w general purpose processors and Application Specific Integrated Circuits (ASICs)
- * The need for an ASIP arises when the traditional general purpose processor are unable to meet the increasing application needs.
- * Some Microcontrollers, system on chips, digital signal processors etc are examples for application specific instruction set processors (ASIPs)
- * ASIPs incorporate a processor and on-chip peripherals, demanded by the application requirement, program and data memory.

Digital Signal Processors (DSPs):

- * Digital signal processors (DSPs) are powerful special purpose 8/16/32 bit microprocessors designed specifically to meet the computational demands and power constraints of today's embedded audio, video and communications applications
- * DSPs are 2 to 3 times faster than the general purpose microprocessors in signal processing applications
- * DSPs implement algorithms in hard ware which speeds up the execution whereas general purpose processors implement the algorithm in firmware and the speed of execution depends primarily on the clock for the processors.
- * DSP can be viewed as a microchip designed for performing high speed computational operations for 'addition', 'subtraction', 'multiplication', and 'division'.

* The key units of the DSPs are:

1. Program Memory
2. Data Memory
3. Computational Engine
4. I/O unit.

RISC vs CISC processors/Controllers:

* RISC stands for Reduced Instruction Set Computing.

As the name implies, all RISC processors/controllers possess lesser number of instructions, typically in the range of 30 to 40.

* CISC stands for Complex Instruction set Computing. It is clear that the instruction set is complex and instructions are high in numbers.

RISC	CISC
1. RISC stands for Reduced Instruction set computing	1. CISC stands for Complex Instruction set Computing.
2. Lesser number of Instructions	2. Greater number of Instructions.
3. Instruction pipelining and increased execution speed.	3. Generally no instruction pipelining feature.
4. Orthogonal instruction set (Allows each instruction to operate on any register and use any addressing mode)	4. Non-orthogonal instruction set (All instructions are not allowed to operate on any register and use any addressing mode. It is instruction specific)
5. Operations are performed on registers only, the only memory operations are load & store.	5. Operations are performed on registers or memory depending on the instruction.

6. A large number of registers are available	6. Limited number of general purpose registers
7. Programmer needs to write more code to execute a task since the instructions are simpler ones.	7. Instructions are like macros in C language. A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC
8. Single, fixed length instructions	8. Variable length instructions
9. Less silicon usage and pin count	9. More silicon usage since more additional decoder logic is required to implement the complex instruction decoding
10. With Harvard Architecture	10. Can be Harvard or Von-Neumann Architecture

Harvard Vs Von-Neumann Architecture:

- * The terms Harvard & Von-Neumann refers to the processor architecture design
- * Microprocessors/controllers based on the Von-Neumann architecture shares a single common bus for fetching both instructions & data. Program instructions and data are stored in a common main memory
- * Microprocessors/controllers based on the Harvard architecture will have separate data bus & instructions bus. This allows data transfer & program fetching to occur simultaneously on both buses.

* with Harvard architecture, the data memory can be read and written while the program memory is being accessed.

* These separated data memory and code memory buses allow one instruction to execute while the next instruction is fetched.

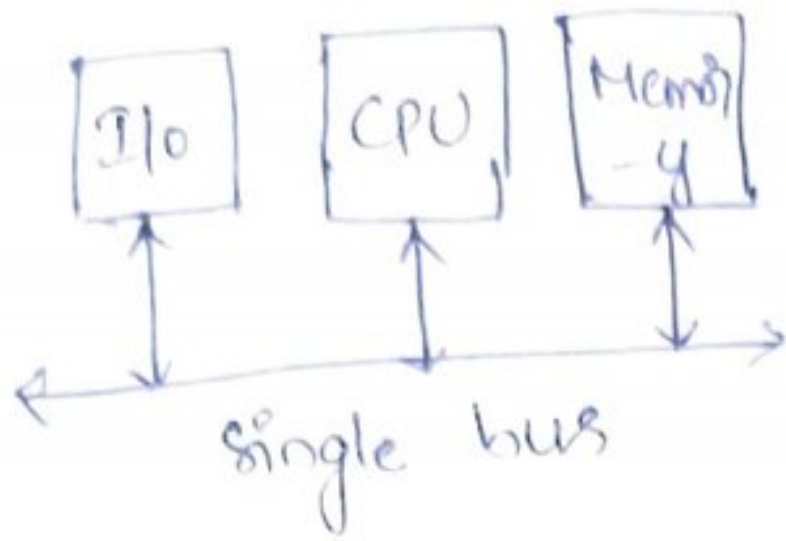


fig: Von-Neumann Architecture



fig: Harvard Architecture.

Difference between Von-Neumann and Harvard Architecture

Harvard Architecture	Von-Neumann Architecture
1. Separate buses for instruction and data fetching.	1. Single shared bus for instruction and data fetching.
2. Easier to pipeline, so high performance can be achieved.	2. Low performance compared to Harvard architecture.
3. Comparatively high cost.	3. Cheaper.
4. No memory alignment problems.	4. Allow self-modifying codes.
5. Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory.	5. Since data memory & program memory are stored physically in the same chip, chances for accidental corruption of program memory.

Big-Endian Vs Little-Endian Processors/Controllers :

* Endianness specifies the order in which the data is stored in the memory by processor operations in a multi-byte system.

* Suppose the word length is two byte then data can be stored in memory in two different ways:

1. Higher order of data byte at the higher memory and lower order of data byte at location just below the higher memory.
2. Lower order of data byte at the higher memory and high order of data byte at location just below the higher memory.

* Little-Endian: It means the lower-order byte of the data is stored in memory at the lower address and the higher-order byte at the highest address.

Base address + 0	Byte 0	Byte 0	0x20000 (Base Address)
Base address + 1	Byte 1	Byte 1	0x20001 (Base Address + 1)
Base address + 2	Byte 2	Byte 2	0x20002 (Base Address + 2)
Base address + 3	Byte 3	Byte 3	0x20003 (Base address + 3)

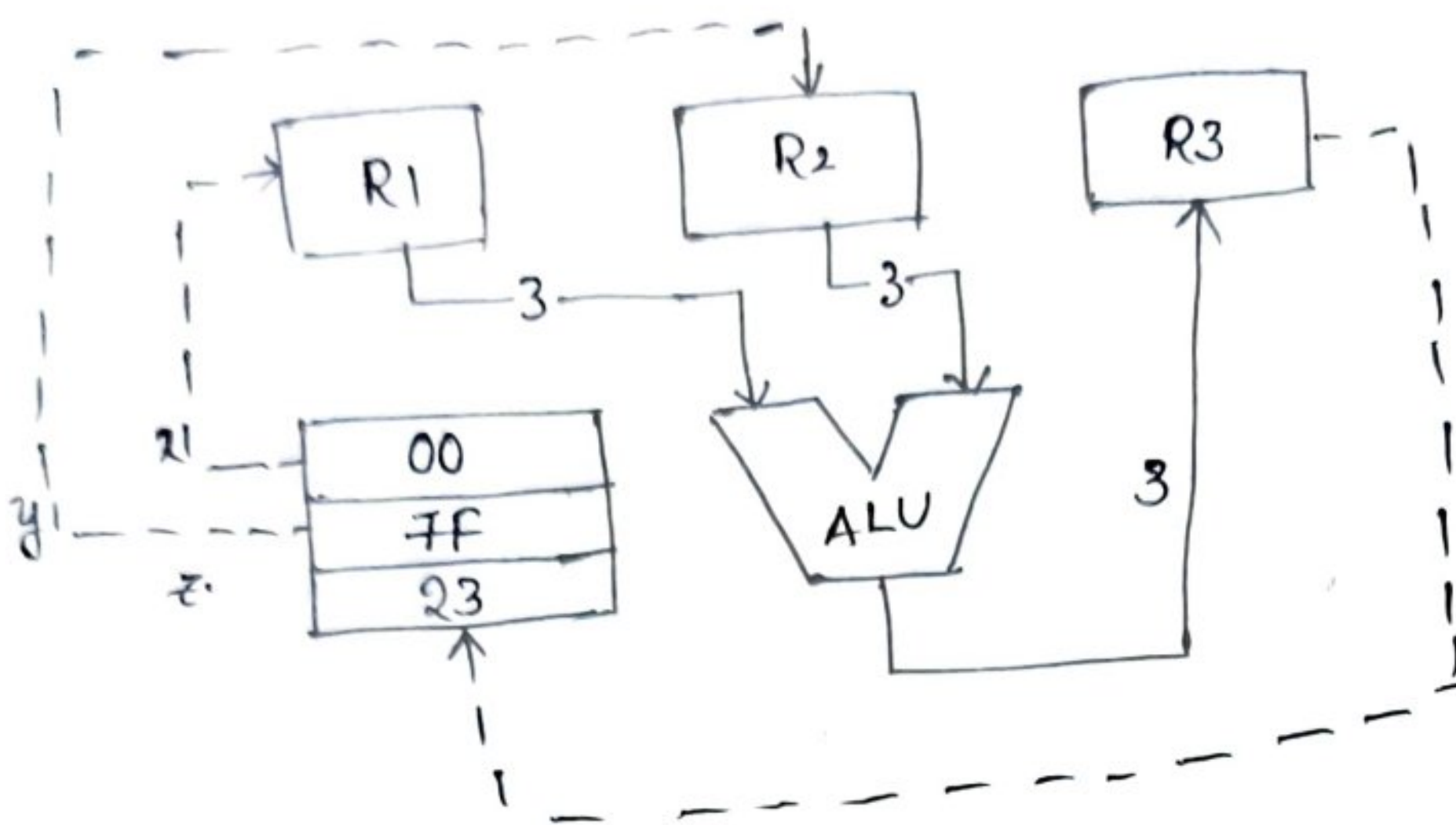
* Big-Endian: It means the higher-order byte of the data is stored in memory at the lowest address and the lower-order byte at the highest address.

Base Address + 0	Byte 3	Byte 3	0x20000 (Base Address):
Base Address + 1	Byte 2	Byte 2	0x20001 (Base Address + 1)
Base Address + 2	Byte 1	Byte 1	0x20002 (Base Address + 2)
Base Address + 3	Byte 0	Byte 0	0x20003 (Base Address + 3)

Load Store Operation and Instruction pipelining:

* The RISC processor instruction set is orthogonal, meaning it operates on registers. The memory access related operations are performed by the special instructions load & store.

- * If the operand is specified as memory location, the content of it is loaded to a register using the load instruction.
- * the instruction store stores data from a specified register to a specified memory location.



- * The conventional instruction execution by the processor follows the fetch - decode - execute sequence.
- * Where the 'fetch' part fetches the instruction from program memory or code memory and the decode part decodes the instruction to generate the necessary control signals.
- * The execute stage reads the operands, perform ALU operations and store the result.
- * In Conventional program execution, the fetch and decode operations are performed in sequence.
- * Decode operation the memory address bus is available and if it is possible to effectively utilise it for an instruction fetch, the processing speed can be increased.
- * In its simplest form instruction pipelining refers to the overlapped execution of instructions.

Application Specific Integrated Circuits (ASICs)

- * A microchip designed to perform a specific or unique application. It is used as replacement to conventional general purpose logic chips.
- * It integrates several functions into a single chip and thereby reduces the system development cost.
- * Most of the ASICs are proprietary products. As a single chip, ASIC consumes a very small area in the total system and thereby helps in the design of smaller systems with high capabilities/functionalities.
- * ASICs can be prefabricated for a special application or it can be custom fabricated by using the components from a re-usable 'building block' library of components for a particular customer applications.
- * Fabrication of ASICs requires a non-refundable initial investment for the process technology & Configuration expenses.
- * If the Non-Recurring Engineering Charges (NRE) is borne by a 3rd party and the Application Specific Integrated Circuit (ASIC) is made openly available in the market, the ASIC is referred as Application Specific Standard Product (ASSP).
- * The ASSP is marketed to multiple customers such as a general-purpose product is, but to a smaller number of customers since it is for a specific application.
- * Application Specific Integrated Circuits (ASICs) are proprietary products, the developers such chips may not

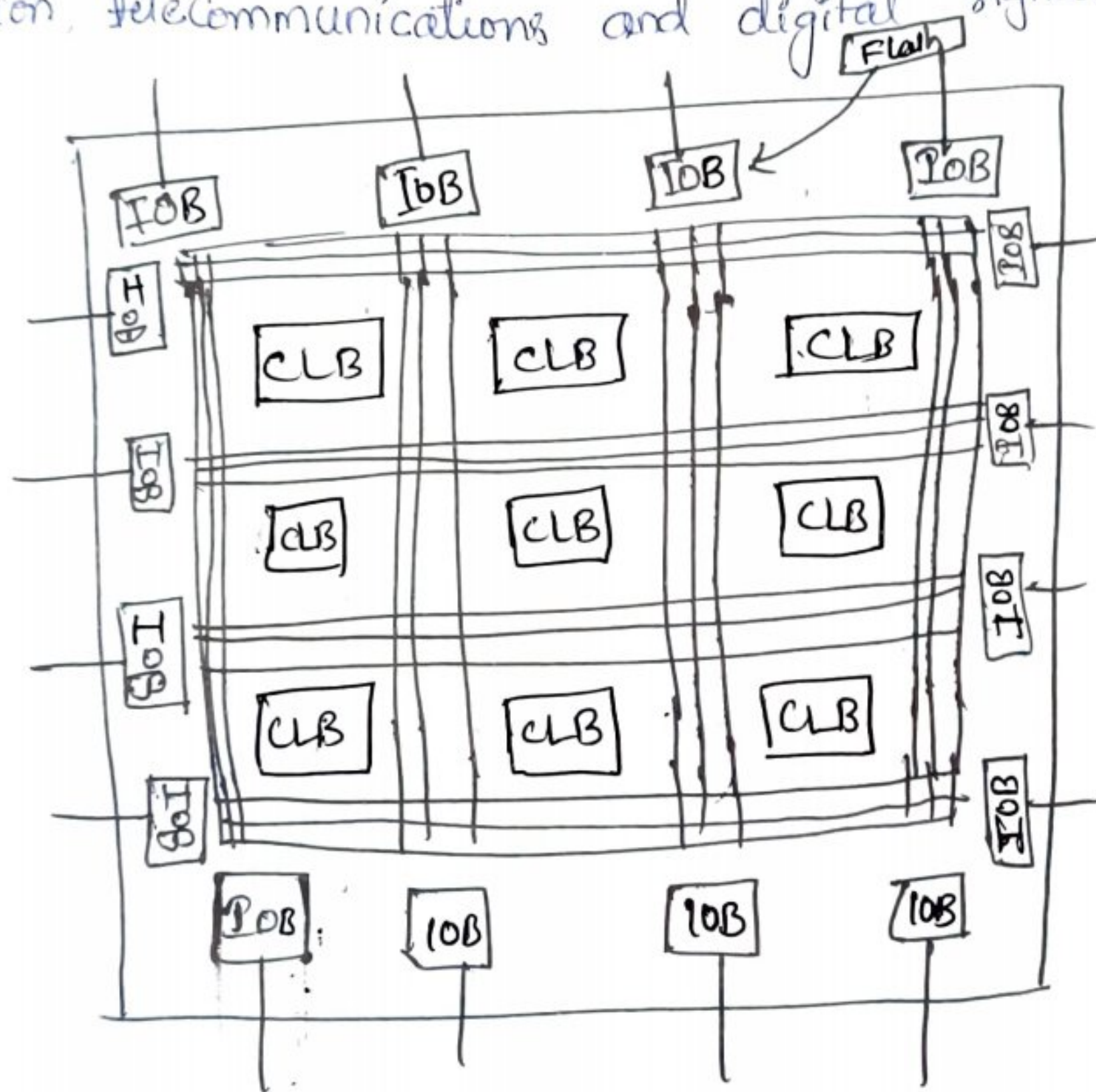
be interested in revealing the internal details.

Programmable Logic Devices (PLDs)

- * Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform.
- * Logic devices can be classified into two broad categories
 - fixed and programmable
- * As the name indicates, the ckt's in a fixed logic device are permanent, they perform one function or set of functions - Once manufactured, they can't be changed.
- * On the other hand, Programmable Logic Devices (PLDs) offers customer a wide range of logic capacity, features speed and 'Voltage Characteristics' and these devices can be re-configured to perform any number of functions at any time.
- * The Designers use inexpensive software tools to quickly develop, simulate and ~~the~~ test their designs.
- * Then, a design can be quickly programmed into a device, and immediately tested in a live circuit.
- * PLDs are based on re-writable memory technology - to change the design, the device is simply reprogrammed.
- * Complex Programmable Logic Devices (CPLDs) and Field Programmable Gate Array (FPGAs) are the two major types of programmable logic devices.

Field Programmable Gate Array (FPGA)

- * FPGA is an IC designed to be configured by a designer after manufacturing.
- * FPGAs offer the highest amount of logic density, the most features, and the highest performance.
- * Logic gate is Medium to high density ranging from 1K to 500K system gates.
- * These advanced FPGA devices also offer features such as built-in hardwired processors.
- * FPGAs are also used in wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications and digital signal processing.



Complex Programmable Logic Devices (CPLDs)

- * A Complex programmable logic Device (CPLD) is a programmable logic device with complexity between that of PALs and FPGAs, and architectural features of both.
- * CPLDs, by contrast, offer much smaller amount of logic -

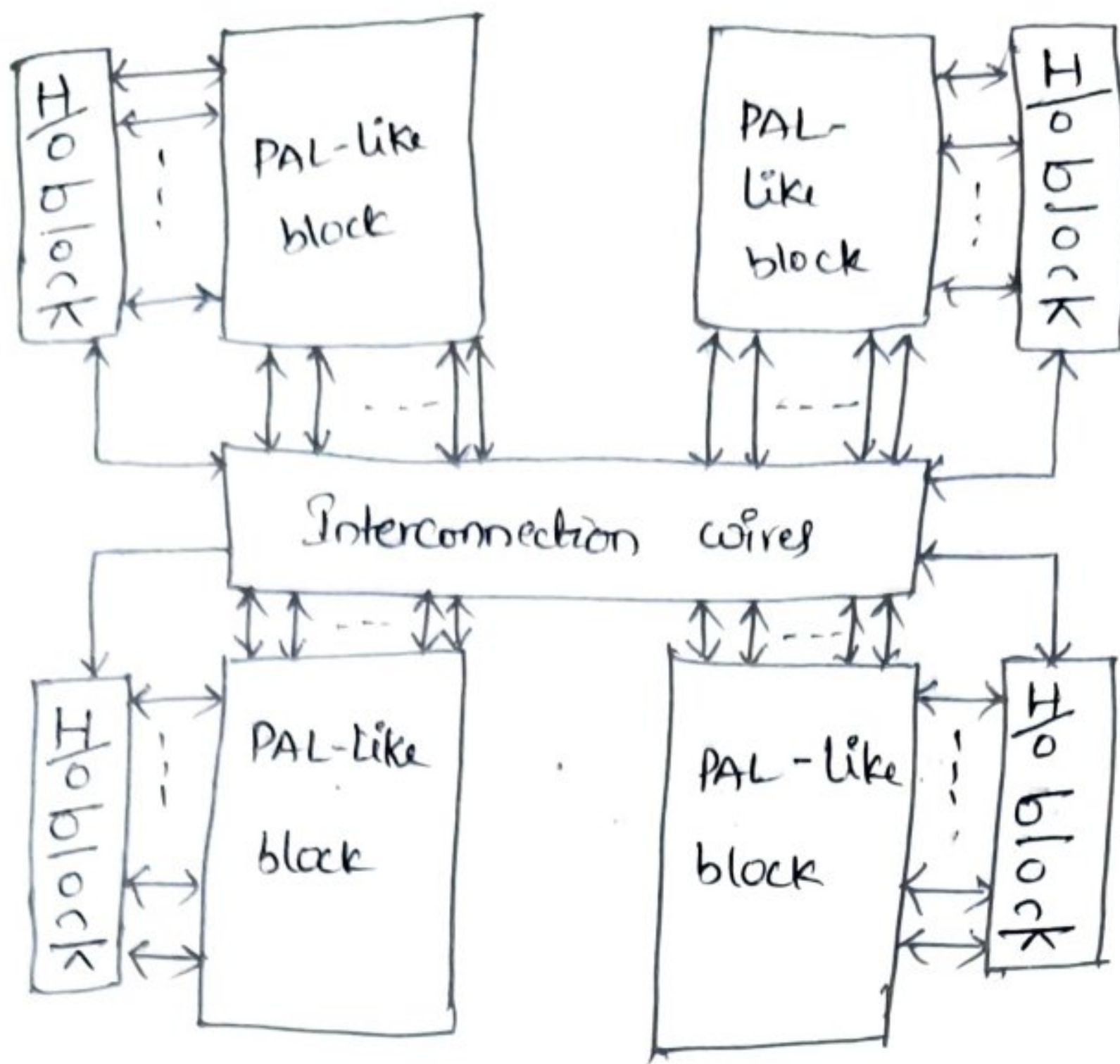


Fig: CPLD architecture

Upto about 10,000 gates.

- * CPLDs offer very predictable timing characteristics and are therefore ~~the~~ ideal for critical control applications
- * CPLDs such as the Xilinx Cool Runner series also require extremely low amounts of power and are very inexpensive, making them ideal for cost-sensitive, battery-operated, portable applications such as mobile phones and digital handheld assistants.

Advantages of PLDs:

- * PLDs offer customers much more flexibility during the design cycle.
- * PLDs do not require long lead times ~~of~~ prototypes or production parts - the PLDs are already on a distributor's shelf and ready for shipment.
- * PLDs don't require customers to pay for large NRE costs & purchase ~~an~~ expensive mask sets.
- * PLDs allow customers to order just the number of

parts they need, when they need them, allowing them to control inventory.

- * PLDs can be reprogrammed even after a piece of equipment is shipped to a customer.
- * The manufacturer able to add new features or upgrade the PLD based products that are in the field by uploading new programming file.

Commercial off-the-shelf Components (COTS)

- * A Commercial off-the-shelf (COTS) product is one which is used 'as-is'.
- * COTS products are designed in such a way to provide easy integration and interoperability with existing system components.
- * Typical examples of COTS hardware unit are remote controlled toy car control units including the RF circuitry part, high performance, high frequency microwave electronics (2-200 GHz), high bandwidth analog-to-digital converters, devices and components for operation at very high temperatures, electro-optic IR imaging arrays, UV/IR detectors etc.
- * There is no need to design this module yourself and write the firmware for the TCP/IP protocol and data transfer.
- * Everything will be readily supplied by the COTS manufacturer.
- * The major advantage of using COTS is that they are readily available in the market, are cheap and a developer can cut down his/her development time to great extent.
- * The COTS Component in turn contains a General

purpose processor (GPP) or Application Specific Instruction set processor (ASIP) or Application Specific Integrated Chip (ASIC)/ Application Specific Standard Product (ASSP) or programmable logic Device (PLD).

- * The major problem faced by the end-user is there are no operational and manufacturing standards.
- * The major drawback of using COTS Components in embedded design is that the manufacturer of the COTS Component may withdraw the product or discontinue the production of the COTS at any time if a rapid change in technology.
- * This will adversely affect a commercial manufacturer of the embedded system which makes use of the specific COTS product.

Embedded Firmware:

- * Embedded firmware refers to control algorithm (Program Instructions) and or the Configuration settings that an embedded system developer dumps into the code (Program) memory of the embedded system.
- * There are ~~the~~ various methods available for the developing the embedded firmware. They are
 - 1) Write the program in high level languages like Embedded C/C++ using an Integrated Development Environment.
 - 2) Write the program in Assembly Language using the instructions supported by your application's target processor/controller.

* There are two types of control algorithm design exist in embedded firmware.

1) Infinite loop or 'super loop' based approach, where the control flow runs from top to bottom & then jumps back to the top of the program in a conventional procedure.

2) Deals with splitting the functions to be executed into tasks and running these tasks using a scheduler.

Reset circuit

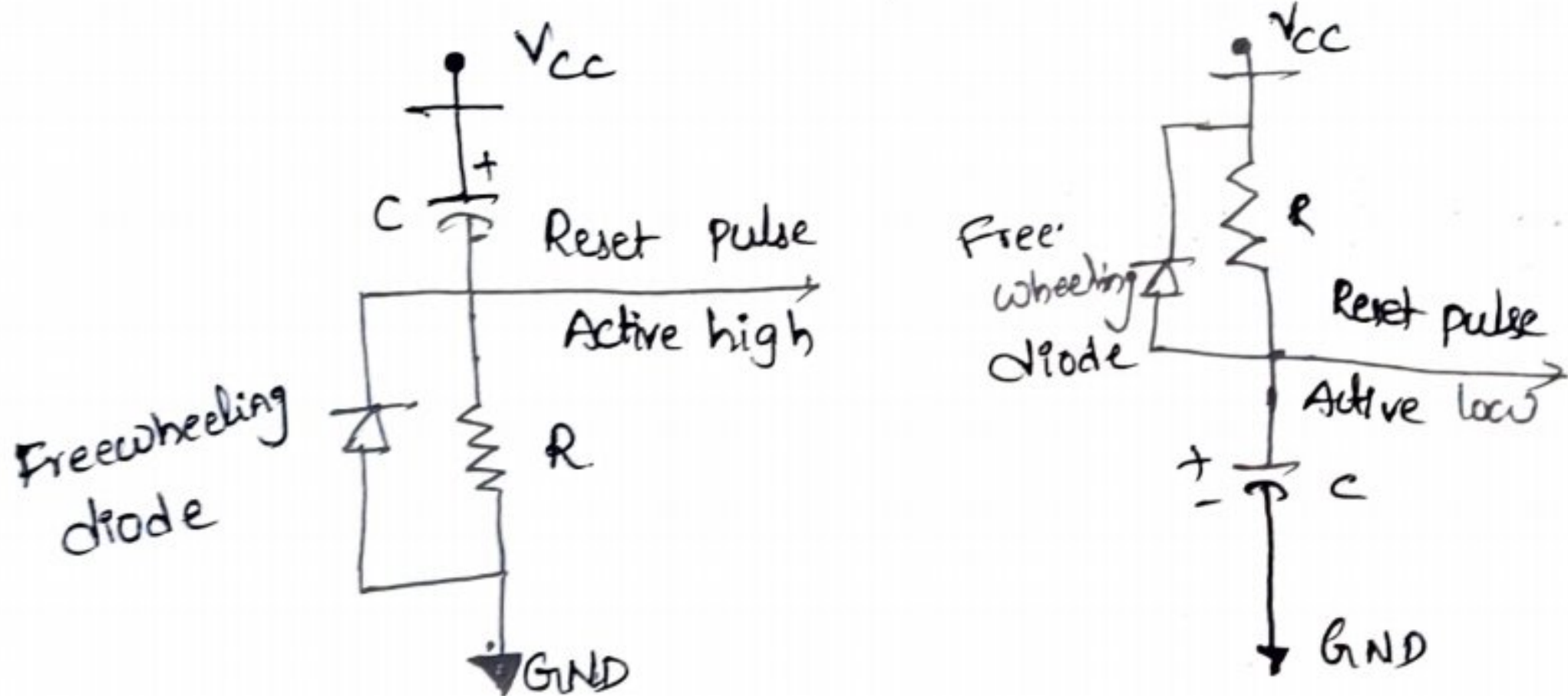


Fig: RC based reset circuit

* The reset circuit is essential to ensure that the device is not operating at a voltage level where the device is not guaranteed to operate, during the system power 'ON'.

* The reset signal brings the internal registers and the different hardware systems of the processor/controller to a known state & starts the firmware execution from the reset vector.

* The reset signal can be either active high or active

Low.

- * Since the processor operation is synchronised to a clock signal, the reset pulse should be wide enough to give time for the clock oscillator to stabilise before the internal reset state starts.
- * The reset signal to the processor can be applied at power ON through an external passive reset circuit comprising a capacitor & resistor or through a standard Reset IC.
- * Some microprocessors / microcontrollers contain built-in internal reset circuitry and they don't require external reset circuitry.
- * The reset pulse width can be adjusted by changing the resistance value R & capacitance value C .

Brown-Out Protection Circuit:

- * Brown-out protection circuit prevents the processor / controller from unexpected program execution behaviour; when the supply voltage to the processor / controller falls below a specified voltage.

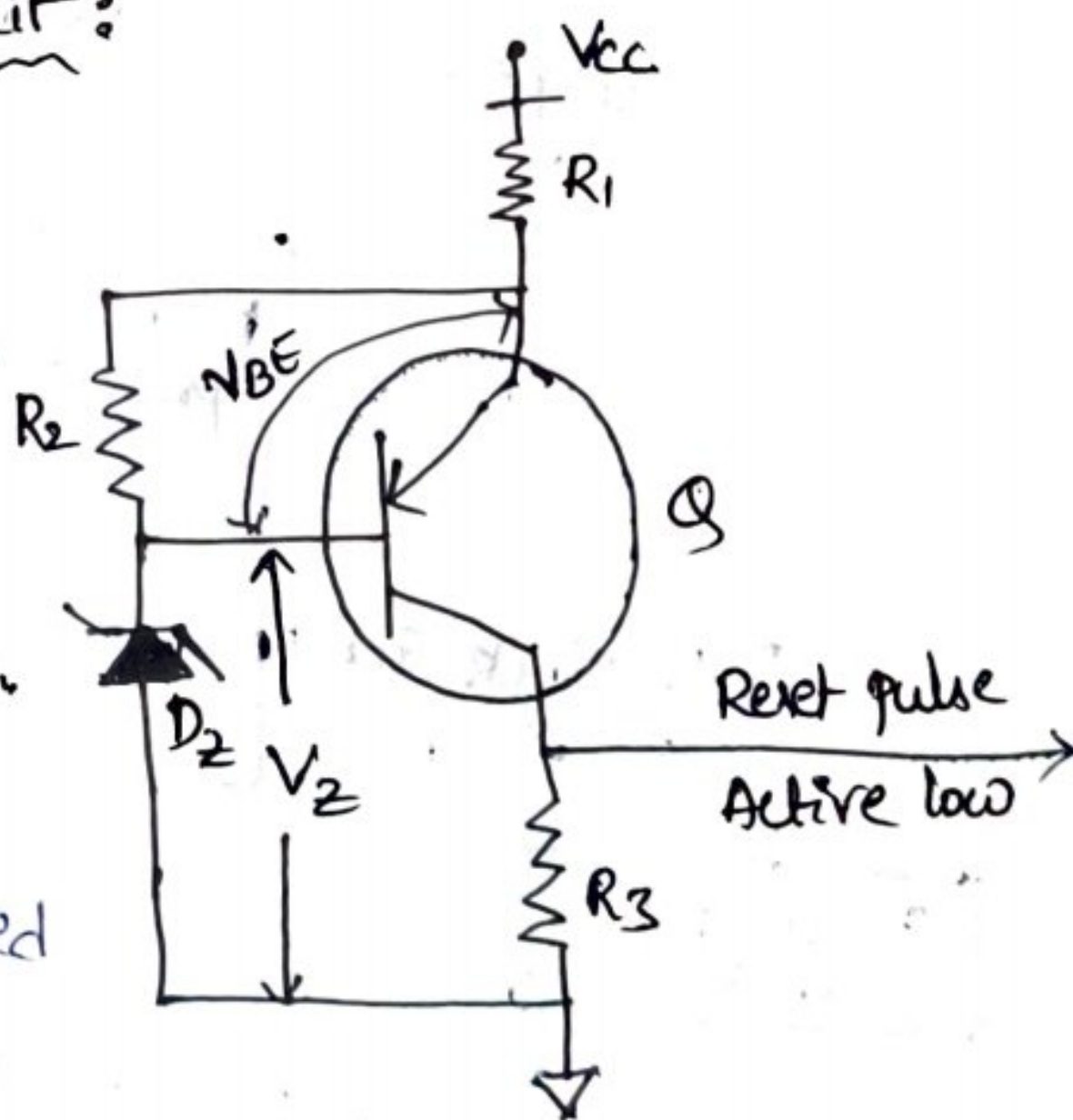


fig: Brown-out Protection ckt with Active low op.

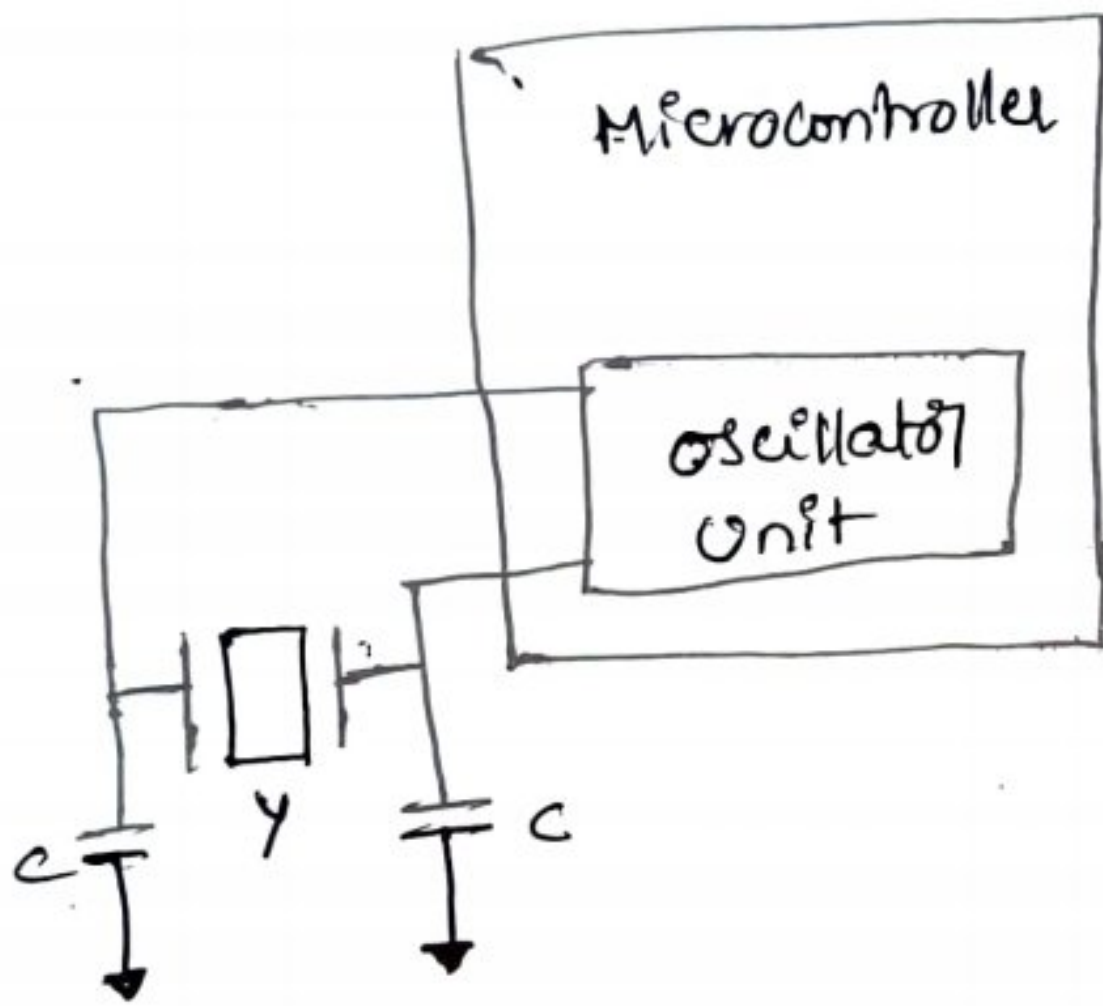
- * It holds the processor / controller in reset state, when the operating voltage falls below the threshold voltage, until it raises above that value.
- * Certain Processors / Controllers support built in brown-out

Protection circuit which monitors the supply voltage internally.

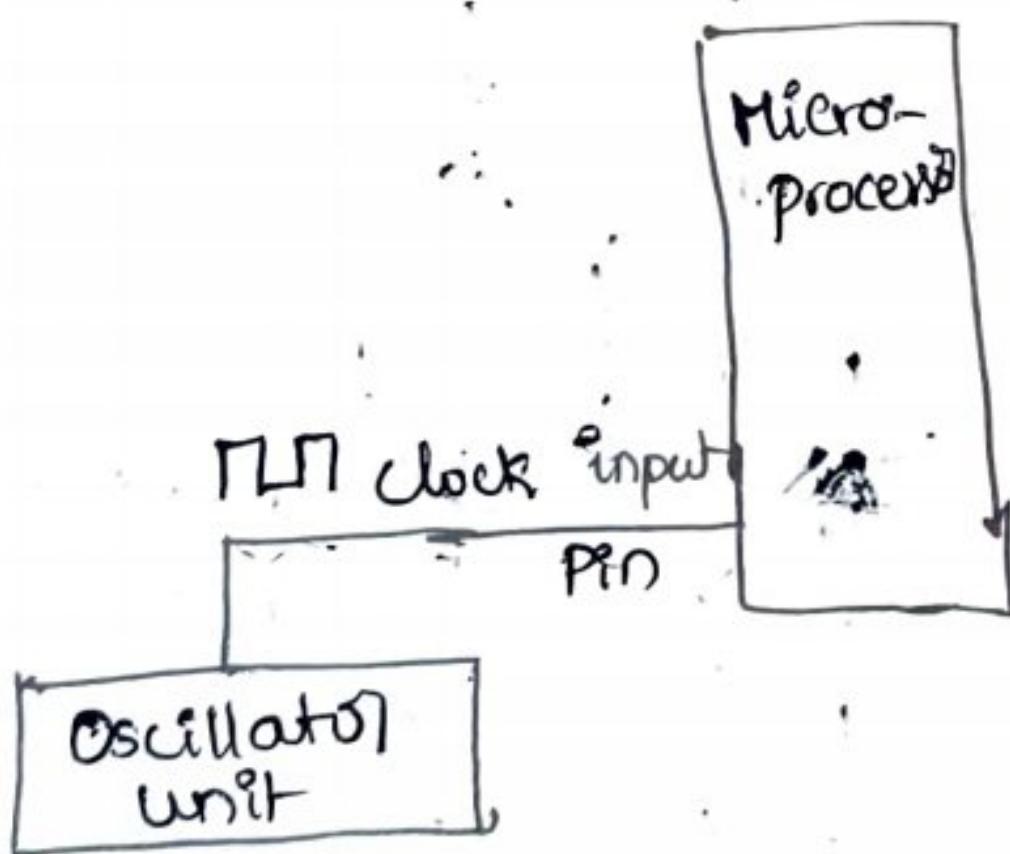
* If the processor/controller doesn't integrate a built-in brown-out protection circuit, the same can be implemented using external passive circuits.

* Figure illustrates the brown-out circuit implementation using Zener diode and transistor for processor/controller with active low Reset logic

Oscillator circuit :



fig(a): Oscillator circuitary using Quartz crystal



fig(b): Quartz crystal oscillator

* A microprocessor / microcontroller is a digital device made up of digital Combinational and sequential Circuits.

* The instruction execution of a microprocessor/ Controller occurs in sync with a clock signal.

* Processors/ Controllers integrate a built-in Oscillator unit & simply require an external Ceramic resonator/ Quartz crystal for producing the necessary clock signals.

* The speed of the operation of a processor is primarily dependent on the clock frequency.

* We can't increase the clock frequency for increasing the speed of execution.

* The Consumption of the power increases with increase the clock frequency.

* The accuracy of program execution depends on the accuracy of the clock signal & is expressed in terms of $\pm$ ppm (parts per million).

Real-Time clock (RTC):

* Real-Time Clock (RTC) is a system component responsible for keeping track of time.

* RTCs are available in the form of Integrated Circuits from different semiconductor manufacturer.

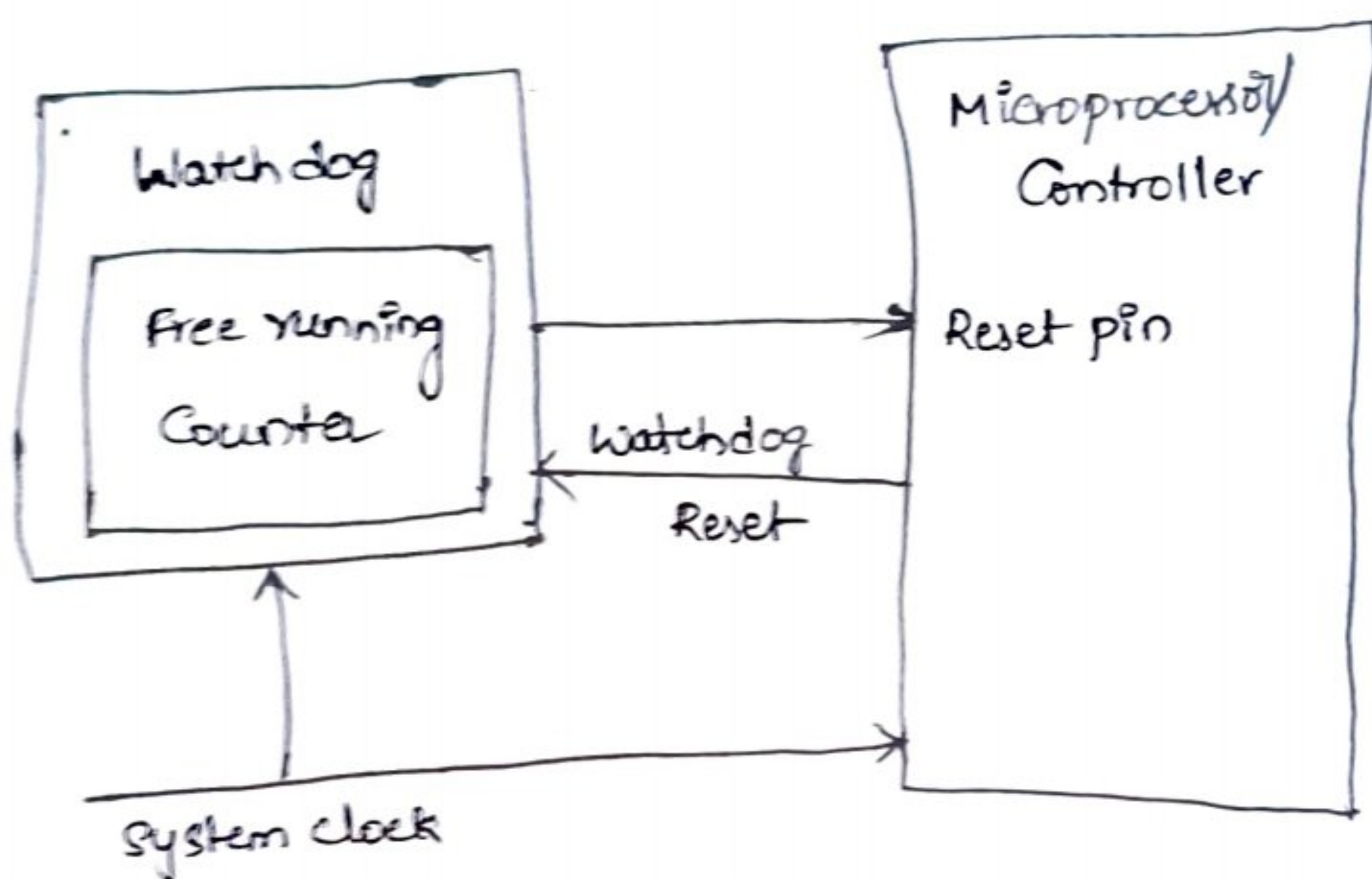
* It is intended to function even in the absence of power and holds information.

* The RTC chip contains a microchip for holding the time and date related information and backup

battery cell for functioning in the absence of power, in a single IC package.

- * The RTC can be configured to interrupt the processor at predefined intervals or to interrupt the processor when the RTC register reaches a specified value.

Watchdog Timer



- * A watchdog timer, or simply a watchdog, is a hardware timer for monitoring the firmware execution.
- * Watchdog reset the system processor/microcontroller when the program execution hangs up.
- * Depending on the internal implementation, the watchdog timer increments or decrements a free running counter with each clock pulse and generates a reset signal to reset the processor if the count reaches zero, for a down counting watchdog, or the highest count value for an upcounting watchdog.

- * If the watchdog counter is in the enabled state, the firmware execution can write a zero to it before starting the execution of a piece of code and the watchdog will start counting.
- * If the firmware execution doesn't complete due to malfunctioning, within the time required by the watchdog to reach the maximum count, the counter will generate a reset pulse and this will reset the processor.
- * If the firmware execution ~~will~~ completes before the expiration of the watchdog timer, you can reset the count by writing a '0' to the watchdog time register.
- * If the processor/controller doesn't contain a built-in watchdog timer, the same can be implemented by using an external watchdog timer IC.
- * The external watchdog timer uses hardware logic for enabling/disabling, resetting the watchdog count, etc instead of the firmware based 'writing' to the status and watchdog timer registers.
- * The Microprocessor Supervisor IC DS1232 integrates a hardware watchdog timer in it.
- ~~* The above fig illustrates the implement hardware for this handles the situation in~~
- * The above fig illustrates the implementation of an external watchdog timer based on Microprocessor supervisor circuit for a small scale embedded system.

Embedded firmware design Approaches:

- * The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed, the speed of the operation required etc.
- * There are two basic approaches are used for "Embedded firmware design. They are:
 1. Conventional Procedural Based firmware design
It is also known as 'super loop Model'.
 2. Embedded Operating System (OS) Approach.

The Super loop Based Approach:

- * The Super loop based firmware development approach is adopted for applications that are not time critical and where the response time is not so important.
- * It is very similar to a Conventional procedural programming where the code is executed task by task.
- * The task listed at the top of the program code is executed first and the tasks just below the top are executed after completing the first task. This is a true procedural one.
- * In a multiple task based system, each task is executed in serial in this approach. The firmware execution flow for this will be
 - 1) Configure the common parameters and perform initialisation for various hardware components memory, registers etc
 - 2) Start the first task and execute it.
 - 3) Execute the second task.

4) Execute the next task

5) :

6) :

7) Execute the last defined task

8) Jump back to the first task and follow the flow (same)

* From the firmware execution sequence, it is obvious that the order in which the tasks to be executed are fixed & they are hard coded in the code itself.

* Also the operation is an infinite loop based approach.

* For example, the operational sequence listed above in terms of a 'C' program code as

```
Void main()  
{  
  Configurations();  
  Initializations();  
  while(1)  
  {  
    Task1();  
    Task2();  
    :  
    Task n();  
  }  
}
```

Pros:

1. Doesn't require an operating system for task scheduling and monitoring and free from OS related overheads
2. Simple and straight forward design
3. Reduced memory foot print

Cons:

- 1) Non Real time in execution behaviour (As the no. of tasks increases the frequency at which a task gets CPU time for execution also increases)
- 2) Any issues in any task execution may effect the functioning of the product

Enhancements:

- 1) Combine Super loop based technique with interrupts
- 2) Execute the tasks with require real time attention as Interrupt routines.

Embedded Operating System (OS) Based Approach

- * The Operating System (OS) based approach contains operating systems, which can be either a General purpose operating system (GPOS) or a Real Time Operating System (RTOS) to host the user written application firmware
- * The GPOS based design is very similar to a conventional PC based application development where the device contains an operating system and you will be creating and running user applications on top of it
Ex: Hand held devices / Portable device, Point of sale terminals and Personal digital Assistants.
- * For developing applications on top of the OS the OS supported APIs are used.
- * Similar to the different hardware specific drivers, OS based applications also require 'Driver Software' for different hardware present on the board to communicate with them.

- * Real Time Operating System (RTOS) based design approach is employed in embedded products demanding Real-time response.
- * RTOS respond in a timely and predictable manner to events.
- * RTOS contains a real-time kernel responsible for performing pre-emptive multitasking, scheduler for scheduling tasks, multiple threads etc.
- * The RTOS allows flexible scheduling of system resources like the CPU and memory. It offers some way to communicate between tasks.

Ex: 'Windows CE', 'pSOS', 'Vx works', 'Thread X', 'Embedded Linux', etc

Embedded firmware Development Languages

(1) Assembly Language based Development:

- * 'Assembly language' is the human readable notation of 'machine language' whereas 'machine language' is a processor understandable language.
- * Machine language is a binary representation and it consist of 1s and 0s.
- * Assembly language & Machine languages are processor/Controller dependent and an assembly program written for one processor/controller family will not work with others.
- * Assembly language programming is the task of writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) associated data using an assembler.
- * The general format of an assembly language instruction

is an Opcode followed by Operands.

- * The Opcode tells the processor/Controller what to do and the operands provide the data & information required to perform the action specified by the opcode.
- * Some of the Opcode implicitly contains the operand & in such situation no operand is required.
- * The operand may be a single operand, dual operand and more.
- * The 8051 ASM instructions as an example,

MOV A, #30.

moves decimal value 30 to the 8051 accumulator register.

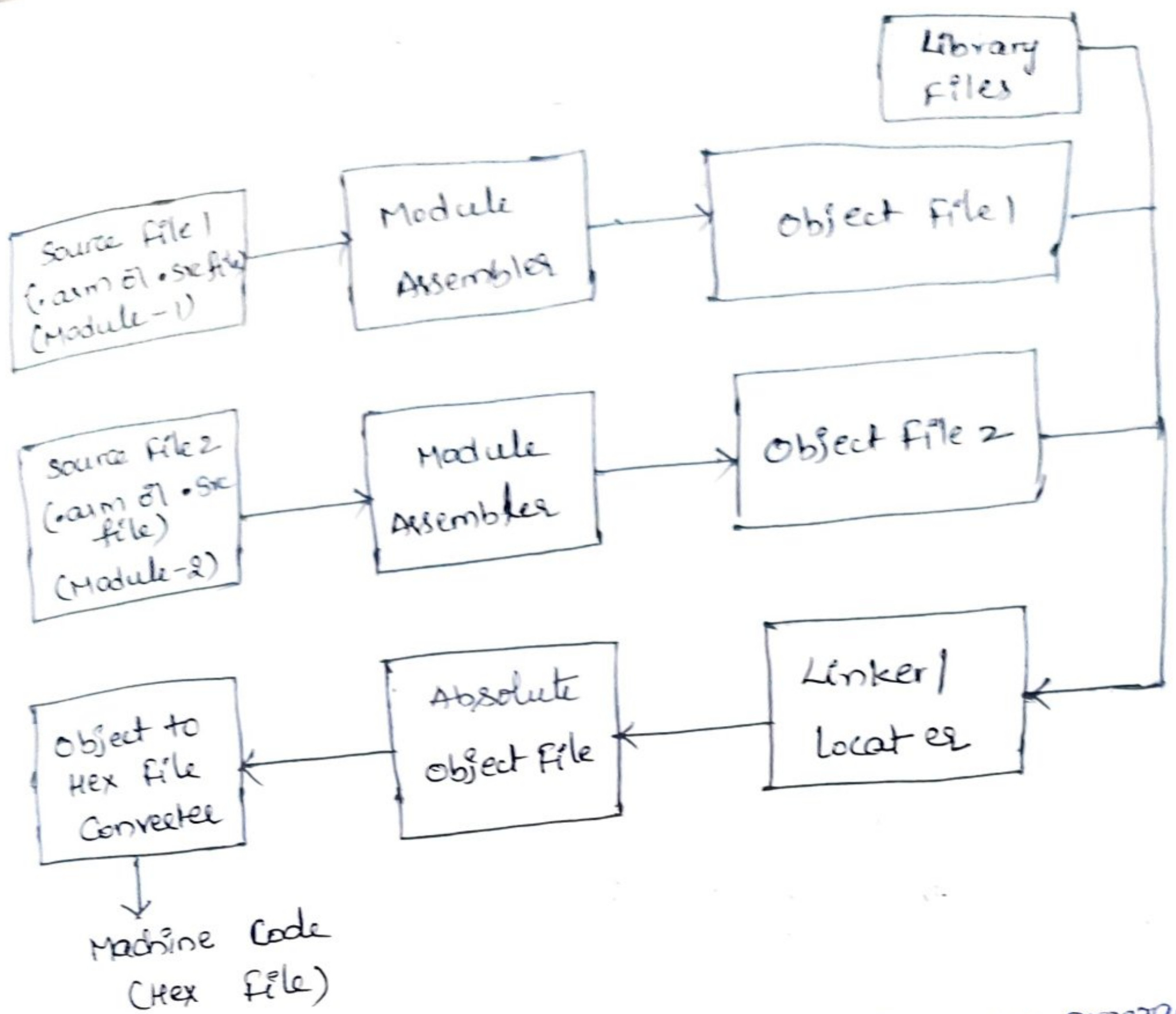
- * Here MOV A is the Opcode and 30 is the operand. The same instruction when written in machine language will look like

01110100 00011110

where the first 8 bit binary value represents the opcode MOV A and the second 8 bit binary value represents the operand 30.

a) Source File to Object File Translation:

- * Translation of assembly code to machine code is performed by assembler.
- * The assemblers for different target machines are different and it is common that assemblers from multiple vendors are available in the market.
- * A51 Macro Assembler from Keil software is a popular assembler for the 8051 family microcontroller.



- * The various steps involved in the conversion of a program written in assembly language to corresponding binary file/ machine language is illustrated as shown in fig.
- * Each source module is written in Assembly and is stored as .src file or .asm file
- * Each file can be assembled separately to examine the syntax errors and incorrect assembly instructions.
- * On successful assembling of each .src/.asm file a corresponding object file is created with extension (.obj).
- * The object file doesn't contain the absolute address of where the generated code needs to be placed on the program memory and hence it is re-located segment.

b) Library File Creation and Usage

- * Library is specially formatted, ordered program collections of object modules that may be used by the linker at a later time.
- * When the linker processes a library, only those object modules in the library that are necessary to create the program are used.
- * Library files are generated with extension '.lib'.
It is some kind of source code hiding technique.

Linker and Locator :

- * Linker and locator is another software utility responsible for 'linking' the various object modules in a multi-module project and assigning absolute address to each module."

Object to Hex File Converter :

- * This is the final stage in the conversion of Assembly language to machine understandable language.
- * Hex File is the representation of the machine code and the hex file is dumped into the code memory of the processor/controller.
- * The hex file representation varies depending on the target processor/controller make.
- * Hex file is created from the final 'Absolute Object File' using the Object to Hex File Converter utility.

Advantages

1. Efficient code Memory and Data Memory usage (Memory optimisation):

2. High performance

3. Low level Hardware Access

4. code reverse engineering

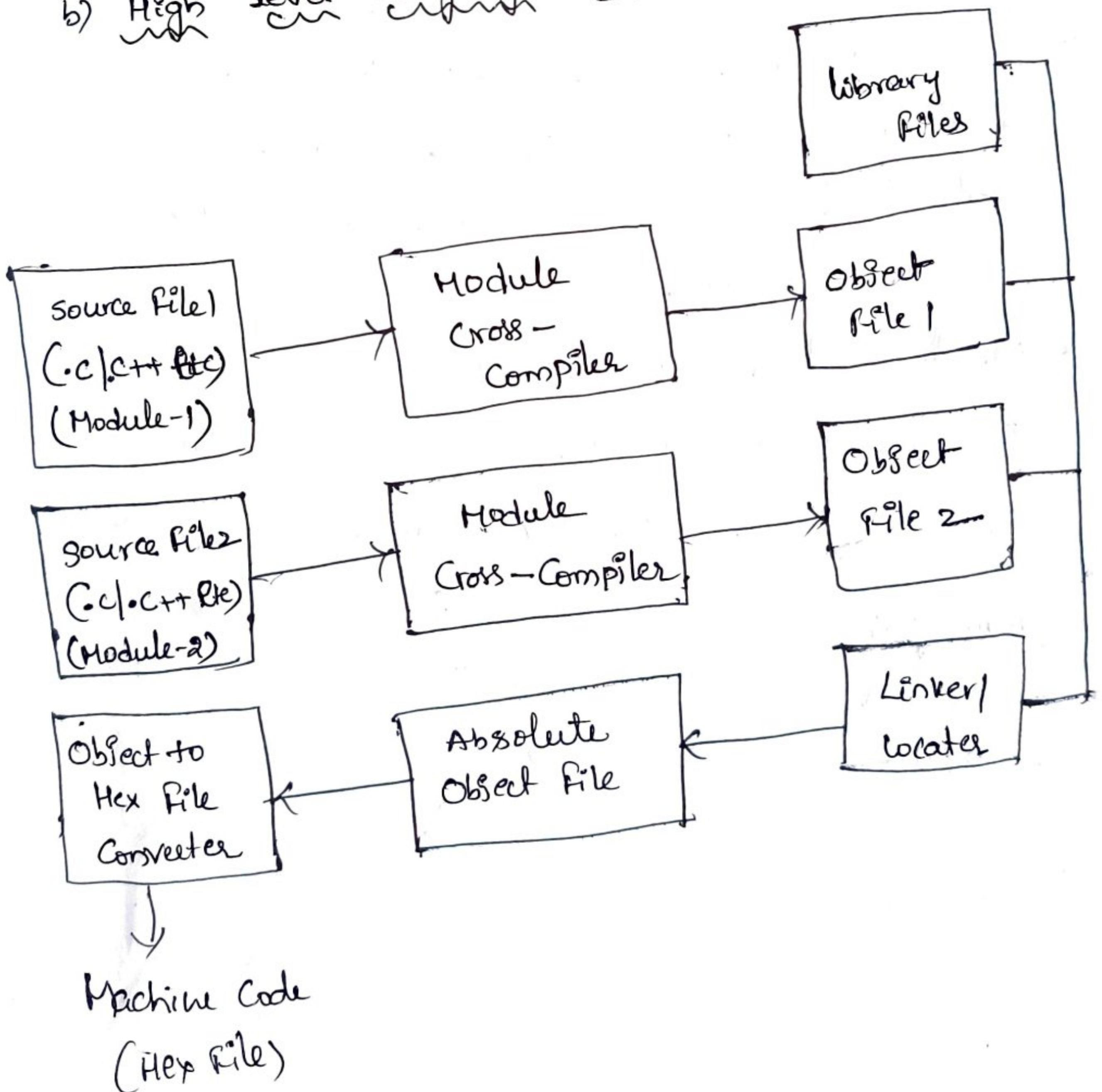
Drawbacks

1. High Development Time

2. Developer Dependency

3. Non-portable.

b) High level Language Based Development:



- * Assembly language based programming is highly time consuming, tedious and requires skilled programmers with sound knowledge of the target processor architecture.
- * Applications developed in Assembly language are not portable.
- * There, the role of high language, with a supported Cross Compiler for the target processor can be used for embedded firmware development.
- * The most commonly used high level language for embedded firmware application development is 'C'.
- * The above figure shows that the various steps involved in the conversion of a program written in high level language to corresponding binary file/machine language.
- * The text editor provided by an Integrated Development (IDE) tool for supporting the high level language in use can be used for writing the program.

Advantages:

1. Reduced Development Time
2. Developer Independency
3. Portability

Limitations

- * Some Cross-Compilers available for high level languages may not be so efficient in generating optimised target processor specific instructions.

2. Target images created by such compilers may be messy & non-optimised in terms of performance & size
3. The time required to execute a task also increases with number of instructions.
4. High level language based code snippets may not be efficient in accessing low level hardware where hardware access timing is critical.
5. The investment required for high level language based development tools is high compared to Assembly language based firmware development tools.

Mixing Assembly and High Level Language

- * Certain embedded firmware development situations may demand the mixing of high level language with Assembly & Vice versa.
- * High level languages and assembly languages are usually mixed in three ways. They are:
 - 1) mixing Assembly language with High level language
 - 2) mixing High level language with Assembly
 - 3) Inline Assembly programming.

Mixing Assembly with High level language:

- * Assembly routines are mixed with 'C' in situations where the program is written in 'C' and the cross compiler in use does not have a built in support for implementing certain features like interrupt Service Routine functions (ISR)
- * If the programmer wants to take the advantage of the speed & optimised code offered by machine code generated by hand written assembly rather than cross compiler generated machine code.

* When accessing certain low level hardware, the timing specifications may be very critical and a cross compiler generated binary may not be able to offer the required time specifications accurately.

* Writing the hardware/peripheral access routine in a processor/controller specific assembly language and invoking it from 'C' is the most advised method to handle such situations.

Mixing High level language with Assembly:

* Mixing the code written in a high level language like 'C' and Assembly language is useful in the following scenarios:

1. The source code is already available in Assembly language and a routine written in a high level language like 'C' needs to be included to the existing code.
2. The entire code is planned in Assembly code for various reasons like optimised code, optimal performance, efficient code memory utilisation and proven expertise in handling the Assembly.
3. But some portions of the code may be tedious and very difficult to code in Assembly.
4. To include built in library functions written in 'C' language provided by the Cross Compiler.

Inline Assembly

- * Inline assembly is another technique for inserting target processor/controller specific assembly instructions at any location of a source code written in a high level language 'C'.
- * This avoids the delay in calling an assembly ~~code~~ routine from a 'C' code.
- * Special keywords are used to indicate that the start and end of assembly instructions.

Unit-3

Trends in the Embedded Industry

Processor Trends in Embedded System

- * Indeed, the advances in the processor technology are the prime driving factor in the embedded development arena.
- * In the 1970s, started developing our embedded devices based on the 8 bit $\mu P/\mu C$. Later we moved to the 16 & 32 bit processor based designs, depending on our system requirement, and when they appear in the market.
- * Even today, the 8 bit microprocessors/controllers are widely used in low-end applications.
- * The today's 8 bit processors/controllers are more power and feature packed.
- * In older days, we used 'n' number of ICs for building a device, but today, with a high degree of integration and IP Core re-use techniques, processors/controllers are integrating multiple IC functionalities into a single chip (System On chip).
- * In older days, required a processor/controller, Brown out Circuit, reset circuit, watchdog timer ICs and ADC/DAC ICs separately for building a simple system, today a single microcontroller with all these components integrated is available in market which leads to miniaturisation & cost saving.

- * Reduced Instruction set Computing (RISC) and execution pipelining contributed a lot to the improvement on processor performance.
- * From the procedural execution, the processor architecture moved to the single stage instruction pipelining and today processor families like ARM supporting 8-stage pipelining with instruction and data cache.
- * The operating clock frequency was a bottleneck in processor design.
- * Now a days processors/controllers with operating frequency in GHz.
- * Another trend in the Embedded processor market is the 'Device Oriented' design of processors.

a) System on chip (SoC)

- * A system on chip (SoC) makes a system on a single chip, places multiple function 'systems'.
- * SoC changed the concept of embedded processors from general purpose processor design to device specific design.
- * Ex: Set Top Boxes, Portable media players, PDAs etc.
- * iMX31 SoC from Freescale semiconductor is a typical example for SoC targeted for multimedia applications.
- * SoCs greatly save the board space in embedded hardware development and thereby leads the concept of miniaturised.
- * SoCs are also cost effective.

b) Multicore processors / Chip level Multi Processor (CMP)

- * One way of achieving increased performance is to increase the operating clock frequency.
- * Due to this, will increase the speed of execution with the cost

of high power consumption.

- * Embedded devices are demanding battery power source for their operation

- * Multicore processors incorporate multiple processor cores on the same chip and works on same clock frequency.

- * It will increase the performance and reduces the cost of battery life.

- * Based on the number of cores, the processors are known as dual core, tri core, quad core etc

- * Multicore Processors Implement multiprocessing.

- * Each core of the CMP implements pipelining, multithreading and superscalar execution.

c) Reconfigurable Processors

- * Reconfigurable Processor is a ~~micro~~ microprocessor/controller with reconfigurable hardware features.

- * They contain an array of programming elements (PE) along with a microprocessor.

- * The PE can be either computational engine or a memory element.

- * The hardware feature of the reconfigurable processor can be changed statically or dynamically.

- * The dynamic changing of hardware configuration brings the advantage of making the chip adaptable to the firmware running on the processor.

- * Depending upon the situational need, the re-configurable processor can entirely change their functionality to adapt to the new requirements.

Embedded OS Trends :

- * The Embedded OS industry is also undergoing revolutionary changes to take advantage of the potential offered by the trends in processor technologies.
- * Today, we have lot of options to select from a bunch of Commercial and open source Embedded OSs for building embedded devices.
- * Most of the embedded OSs are trying to bring the Virtualization concept of the embedded device industry by adopting the microkernel architecture in place of the monolithic architecture.
- * In microkernel architecture, the kernel contains very limited and essential features/services and rest of the features or services are installed as service and they run at the user space.
- * Another trend adopting by OS suppliers is 'Device oriented' design similar to the processor trends.
- * Today OS suppliers are providing off-the-shelf OS customised for the device.

Ex :- Microsoft embedded OS

Development Language Trends :

- * There are two aspects, ~~for~~ languages for development of embedded system. They are
 - 1) System side application
 - 2) User application development.
- * The system side application is responsible for managing the embedded device, interacting with low level hardware, schedule and executing user applications, memory management etc whereas the user application runs on top of the system applications, and requires the intervention of system application

for performing system resource access.

- * In Embedded terminology the system side applications are referred as "Embedded firmware" and the user applications are known as "Embedded software".

- * The embedded firmware always comes as embedded into the program memory of the device and it is unalterable by the end user.

- * Embedded software comes either embedded with the firmware or user can install the software applications later on device.

a) Java for Embedded development

- * Java being considered as the popular language for enterprise applications due to its platform independent implementations.

- * In a basic Java development, the Java code is compiled by a Java class compiler to platform independent code called Java byte code.

- * The Java byte code is converted into processor specific object code by a utility called Java Virtual Machine (JVM). It abstracts the processor dependency from Java applications.

- * JVM performs the operation of converting the byte code into the processor specific machine code implementation.

- * During runtime, the byte code is interpreted by the JVM for execution. The interpretation technique makes Java applications slower than the other compiled applications.

- * JVM can be implemented as either a piece of software

code of hardware unit.

* The below figure illustrates the implementation of Java applications in an embedded device with operating system.

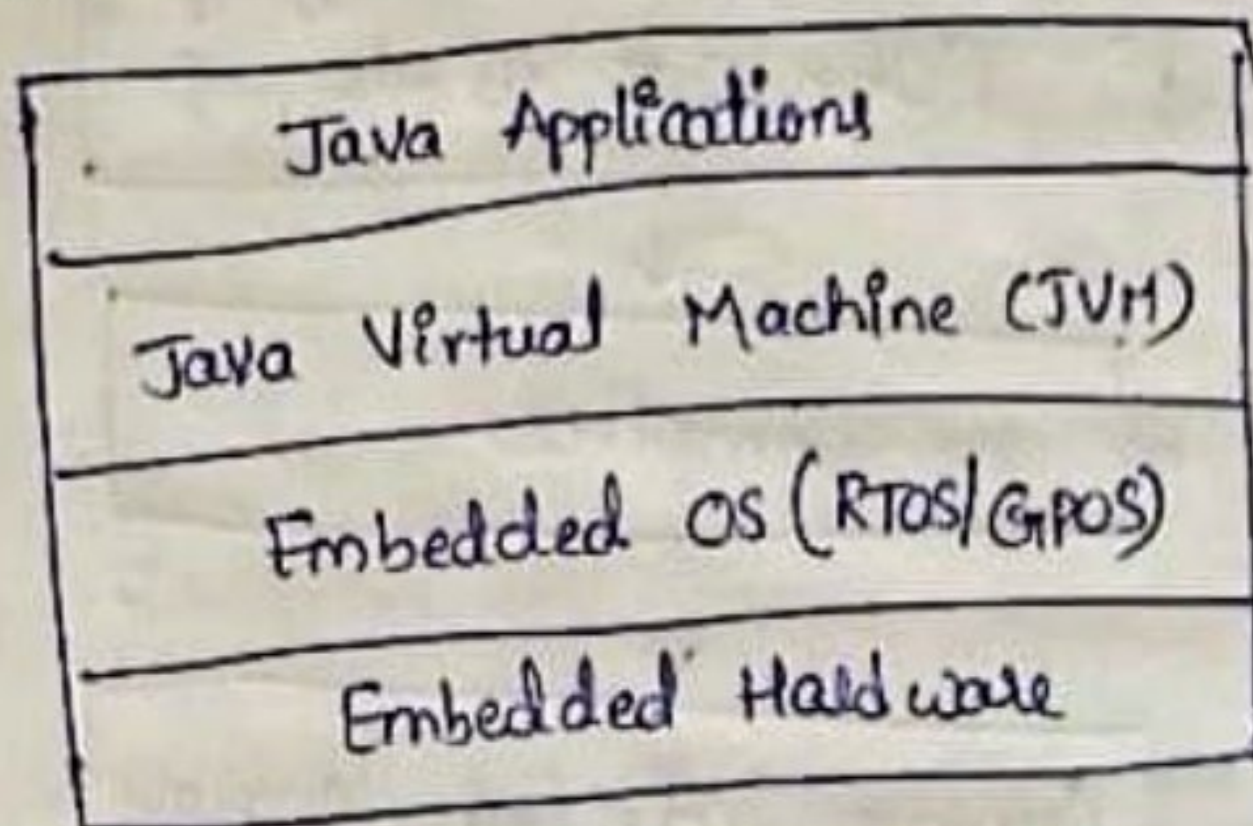
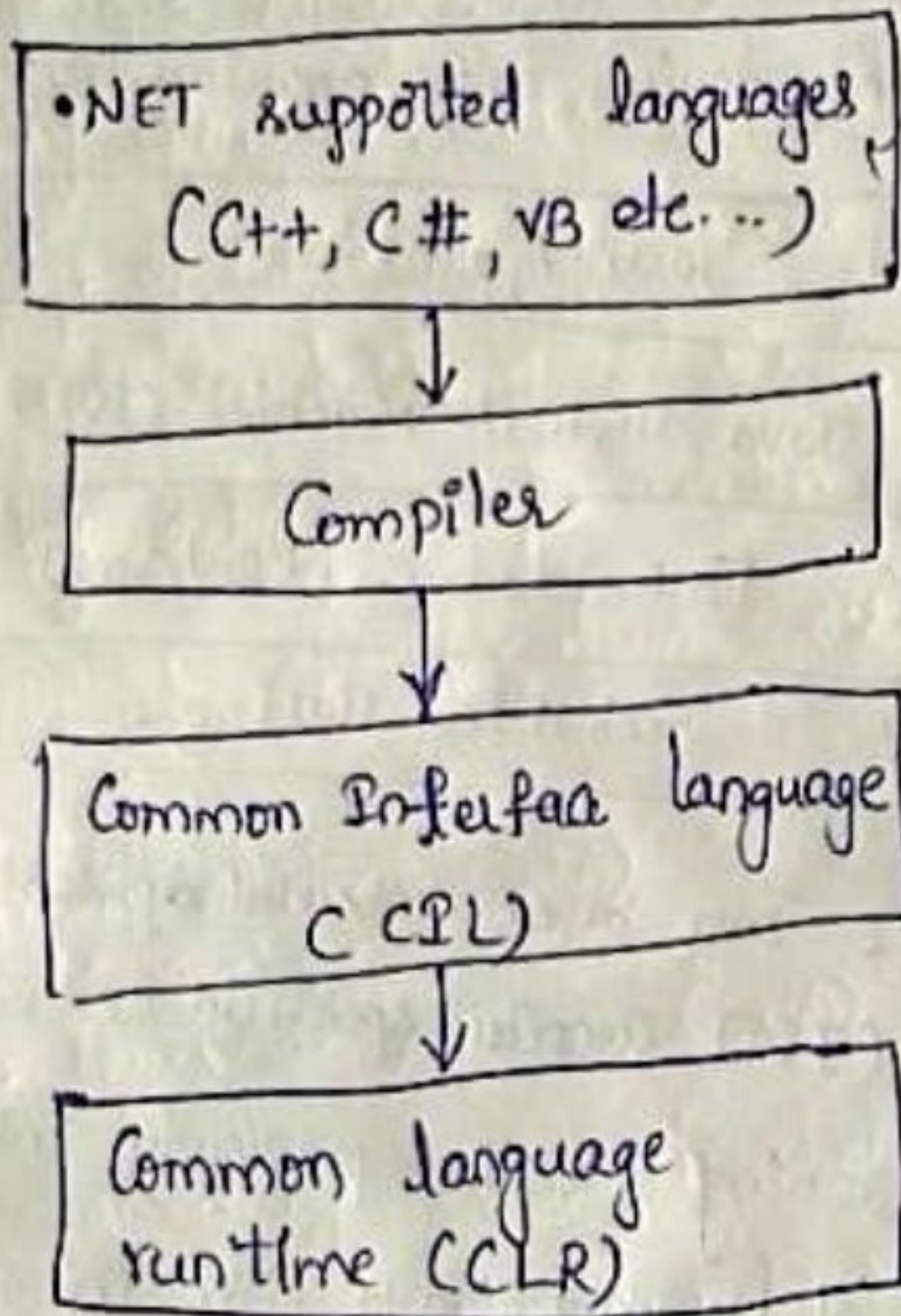


fig: Java based embedded application development

- * Just in Time (JIT) Compiler speeds up the Java program execution by caching all the previously interpreted bytecode.
- * The limitations of standard Java Code in embedded ~~platform~~ application development are:
 - 1) It is quite slower and not meeting the real time requirement demanding by most embedded systems
 - 2) Garbage collector of Java is non-deterministic in execution of behaviour. It is
 - 3) Processor which don't have a built-in JVM support or an off-the shelf software JVM, require the JVM ported for the processor architecture.
 - 4) The run-time memory requirement for JVM & Java class libraries of JIT is a bit high.
- * Java provides an interface named Java Native Interface (JNI) which can be used for invoking the functions written in other languages like C and Assembly.

b) .NET CF for Embedded development



- * .NET is a framework for application development for desktop windows operating systems from Microsoft
- * It is a collection of pre-coded libraries and it acts as the run time component for .NET supported languages.
- * It contains class libraries for user interface, database connectivity, network connectivity, image editing, cryptography etc
- * The runtime environment of .NET is known as Common Language Runtime (CLR). It resembles JVM in operation.
- * It abstracts the target processor from application programmer.
- * The CLR provides a language neutral platform for application development and execution.
- * Applications written in .NET supported languages are

Compiled to a platform neutral intermediate language called Common Intermediate Language (CIL).

- * The CIL is converted to the target processor specific machine code by the Common Language Run-time (CLR).

- * The above fig illustrates the concept of .NET framework based program execution

- * The .NET framework is targeted for enterprise application development for desktop systems.

- * It consumes significant amount of memory. When it comes to embedded application development, the .NET framework is a viable choice due to its memory requirement.

Open standards, frameworks and Alliances :

- * Certain areas of the embedded industry are highly 'hot' and competitive.

- * It demands strategic alliances to sustain in the market and bring innovations through combined R&D.

- * The combined R&D helps hardware manufacturers to come up with new designs and software developers to support the new hardware.

- * The Open Alliances and standards body are aimed towards defining and formulating standards.

- * Time to market is a critical factor in the sensational embedded market segments.

- * Open source and frameworks reduce the time to market in product development.

a) Open Mobile Alliance (OMA)

- * The Open Mobile Alliance (OMA) is a standards body for developing open standards for mobile phone industry
- * The Mission of OMA is "To create interoperable services across countries, operators, and mobile terminals by acting as the centre of the mobile service enabled specification work."

b) Open Handset Alliance (OHA)

- * The Open Handset Alliance (OHA) is a business alliance formed by various handset manufacturers, chip manufacturers, platform providers and carriers for developing standards for mobile devices.

c) Android

- * Android is an open source platform and operating system for mobile device
- * It was developed by Google corporation and maintained by the Open Handset Alliance.
- * The Android operating system is based on the Linux kernel.

d) Openmoko

- * Openmoko is a project for building open hardware & firmware (OS) standards for a family of open source mobile phones.

* The target operating system under consideration by Openmoko is Embedded Linux and it is known as Openmoko Linux.

* It also supplies hardware details of the phone as reference design for developers.

* Developers can customize the software stack and hardware to suit their product needs.

BottleNecks

* Today, the bottlenecks faced by the embedded industry are discussed below.

a) Memory performance:

* Memory performance is a real road blocker in embedded development.

* Though processor technologies are supporting high speed operations through increased clock, the current memory technology is not yet up to the mark to catchup the speed offered by the processors.

* b) Lack of standards / Conformance to standards:

* Various Alliances and open standards for specific areas of embedded system, there are no specific standards in place to ensure interoperability in all areas of embedded ~~system~~ development.

* To bring innovations and cutting edge feature rich products, it is essential to have a combined effort and

Standardization.

Lack of skilled Resources:

* Design of Embedded system is an Art" and it requires highly skilled, self motivated and talented people to meet the time Criticalities of embedded product development.

* Most of our engineering graduates are highly talented, they lack proper orientation towards the design and development of embedded systems.

Unit - II (Contd. ...)

Development Platform Trends

- * The Development Platform is an integrated platform hardware platform which incorporates a processing unit, integrated peripherals, expansion ports for connectivity with other external peripherals, USB/RS-232/TCP connectivity and debug interface for interfacing with a host computer for firmware development.
- * The various popular Hardware Development platforms are:
 - a) Arduino: Arduino is a single-board microcontroller, an open-source electronics platform based on easy-to-use hardware and software. The hardware consists of an open-source hardware board designed around an 8-bit Atmel AVR microcontroller or a 32-bit Atmel ARM processor. The open source Arduino development environment Arduino IDE for development can be used. You can also use Embedded OS like And OS, which is specifically written for Arduino, or build your own custom OS for Arduino.
 - b) BeagleBone: BeagleBone is an open hardware, community-supported single board embedded computer for ARM Cortex Processor based development. The latest BeagleBone hardware, BeagleBone Black platform is built around the TI Sitara ARM Cortex-A8 processor with onboard EMMC Flash Memory, CTI JTAG interface for debugging support, UART, USB 2.0 client and host ports, Ethernet port, HDMI Video OUT and GPIO lines for external interfacing. There are Debian Linux, Ubuntu and Android OS ports available for BeagleBone.

c) Sharks Cove: Sharks Cove is a development board for, developing device drivers and reference hardware like tablets, on Windows and Android Operating System. It is built around Intel® Atom™ processor and supports various interfaces like I2C, SPI, GPIO, I2S, UART and MPI. It gives the flexibility to integrate off the shelf modules with the existing hardware platform and write device driver for easy integration.

d) MinnowBoard MAX: MinnowBoard MAX is an open hardware embedded design platform designed with Intel Atom Processor and various support various interfaces like I2C, SPI, GPIO, I2S, UART, SDIO, USB 2.0/3.0 Host and PCI Express. It is a powerful platform for small scale low cost embedded system design. Microsoft is supporting Windows on MinnowBoard Max.

e) Raspberry PI: Raspberry PI is a single board computer design platform developed by Raspberry PI Foundation as aid in computer education across the globe. It was designed with Broadcom SOC based on an ARM11 core. The Raspberry PI B & B+ is designed to encourage Raspberry PI for embedded system design. Linux, Raspbian, RTSCOS, Debian, Free BSD and NetBSD OS ports available for Raspberry PI.

f) Intel Galileo Gen2: Intel Galileo Gen2 board is an open source hardware development board based on Intel x86 architecture and specifically designed for makers, students, educators and hobbyists. Then Intel Galileo board is based on the Intel Quark SOC X1000 Application Processor, a 32-bit Intel Pentium class SOC. It is designed to work with the Arduino line of products to deliver more advanced features. Galileo board also software compatible with the Arduino software.

Development Environment. It can run either Your Linux or Windows OS port or you can write your own custom OS or port an OS.

Cloud, Internet of Things and Embedded Systems:

- * Cloud based data storage and analytics capabilities can be leveraged extensively to make embedded systems work beyond their limited set of intended functionalities, although the initial concept of embedded system was a self-contained to serve a specific functionality or a set of functionalities.
- * Cloud is rapidly expanding the scope of embedded computing devices.
- * The 'Connected Car' concept in the Automotive Industry is a typical example where Cloud Computing can work in hand with the 'Embedded' computing units in the car and 'transmit' various statistics in real-time to cloud storage which can be used for online fault diagnostics.
- * Internet of Things (IoT) is a topology representing the ~~interconnect~~ 'interConnection' of embedded (system) computing devices, which can be identified uniquely.
- * IoT can be visualised as an enhanced version of traditional Machine-to-Machine Communication systems, with advanced capabilities.

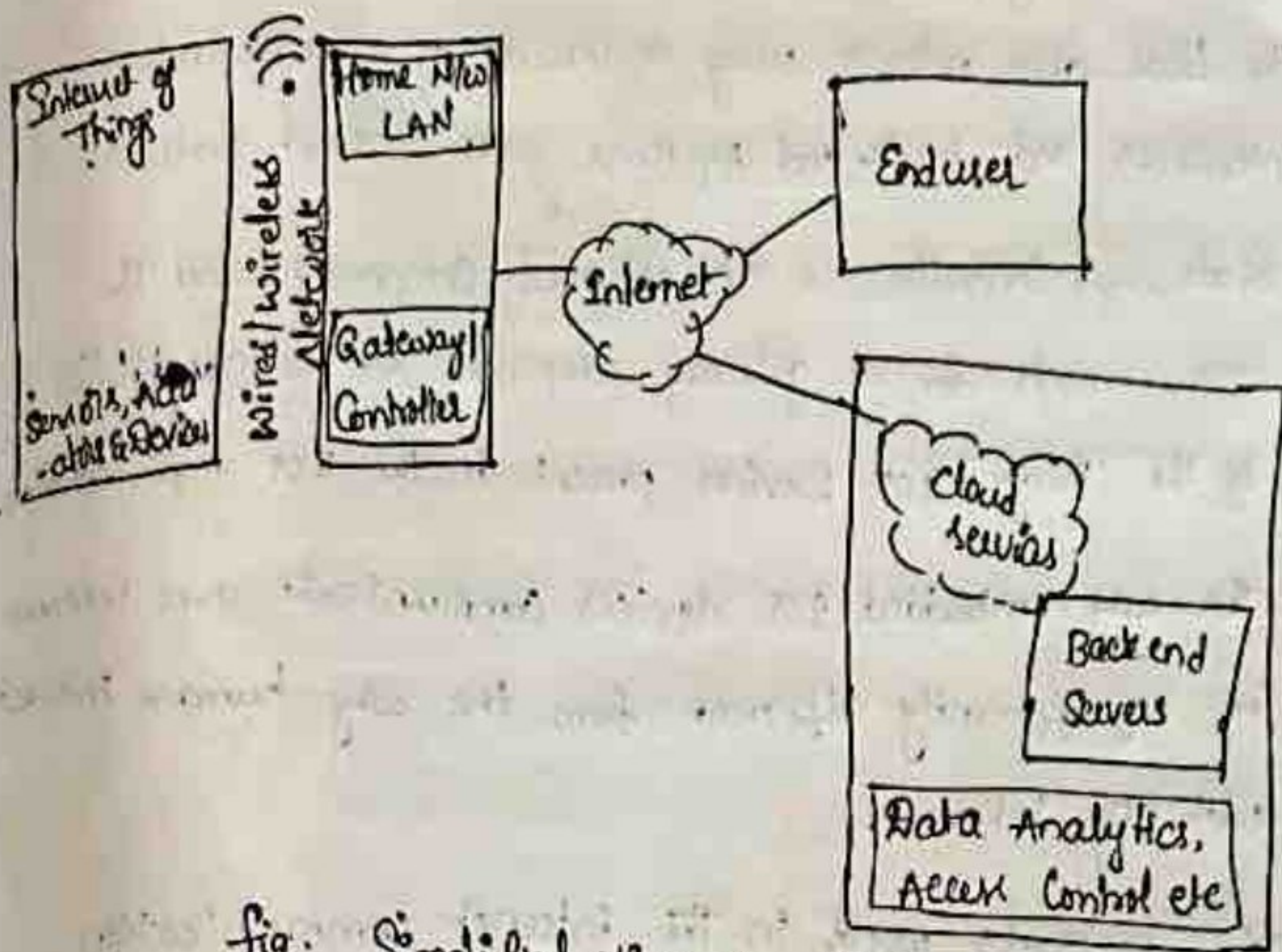


Fig: Simplified View of an IoT E2E System.

- * The above figure shows that the architectural diagram of IoT. Depending on the targeted IoT devices, it may contain the sensor systems for data collection.
- * The IoT End-to-End model is comprised of Things/IoT devices which are mostly self-contained embedded systems/smart sensor-actuator subsystems distributed across multiple locations, a local area n/w, which interfaces the things with gateway/Controller to connect to the Internet and a back-end system hosting a bunch of cloud services which connects the end users and connected devices.
- * A Thing/IoT Device is an implementation can be anything ranging from a smart sensor/actuator device to a self-contained embedded device.
- * The local area network may be built around wired interfaces/protocols like Ethernet, Modbus etc or wireless interfaces like Wi-Fi, Bluetooth etc.

- * The Gateway / controller acts as a middle man between the local area network using a variety of wired/wireless interfaces and back-end systems over Internet protocol.
- * Gateway / Controller is an optional component and the requirements for a gateway depends on the capabilities of the Things / IoT Devices present in the IoT implementation.
- * The way embedded IoT devices communicate over Internet is significantly different from the way humans interact with the Internet.
- * The protocols used in the Internet communication in IoT implementation is not as the same way as conventional communication.
- * MQTT, Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), extensible Messaging & Presence Protocol (XMPP), WebSocket and HTTP are some of the common protocols.

Communication Interface:

* Communication interface is essential for communication with various subsystems.

* The Communication Interface can be viewed in two different perspectives namely.

1. Device / board level Communication Interface (Onboard Communication Interface)

2. Product level Communication Interface (External Communication Interface)

Device / Board level or Onboard Communication Interfaces

* Communication channel which interacts / interconnects the various components with an embedded product is referred as Device / board level Communication interface (Onboard Communication Interface)

EX: Serial Interfaces like I2C, SPI, UART, 1-wire etc and parallel bus Interface.

* These are classified into

1) I2C (Inter Integrated Circuit) Bus

2) SPI (Serial peripheral Interface) Bus

3) UART (Universal Asynchronous Receiver Transmitter)

4) 1-Wires Interface.

5) Parallel Interface.

I2C (Inter Integrated Circuit) Bus:

* Inter Integrated circuit Bus (I2C) is a synchronous bi-directional half duplex Con-directional Communication at a given point of time) two wire serial interface bus.

* The Concept of I2C bus was developed by "Philips Semiconductors" in early 1980's.

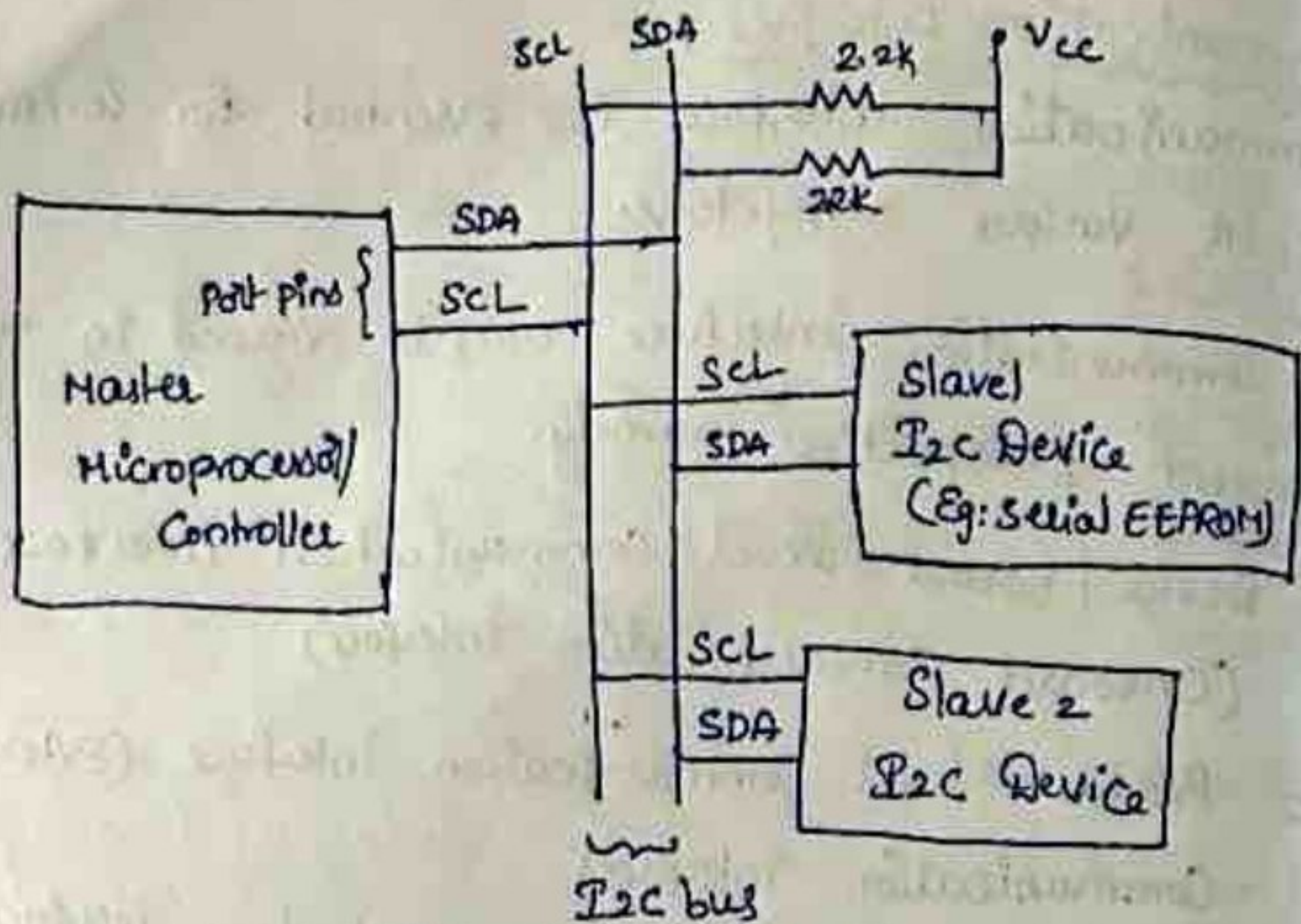


Fig: I2C Bus Interfacing

- * The original Intention of I2C was to provide an easy way of Connection between a μ p/mc system and the peripheral chips in Television sets.
- * The I2C bus is Comprised of two bus lines, namely:
Serial clock - SCL and Serial Data - SDA.

Unit-IV

RTOS based Embedded System Design

- * A RTOS, known as Real Time operating system, is a software component that rapidly switches between tasks, giving the impression that multiple programs are being executed at the same time on a single processing core.
- * A variant of OS that operates in constrained environment in which computer memory and processing power is limited. Moreover they often need to provide their services in definite amount of time.

Operating System Basics

- * The operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services.
- * The OS manages the system resources and makes them available to the user applications/tasks on a need basis.
- * The primary function of the operating system is
 - 1) Make the system convenient to use
 - 2) Organize and manage the system resources efficiently and correctly.
- * The below figure gives the basic components of an operating system and their interfaces.

The Kernel

- * Kernel is the core of the operating system and is responsible for managing the system resources and the communication.

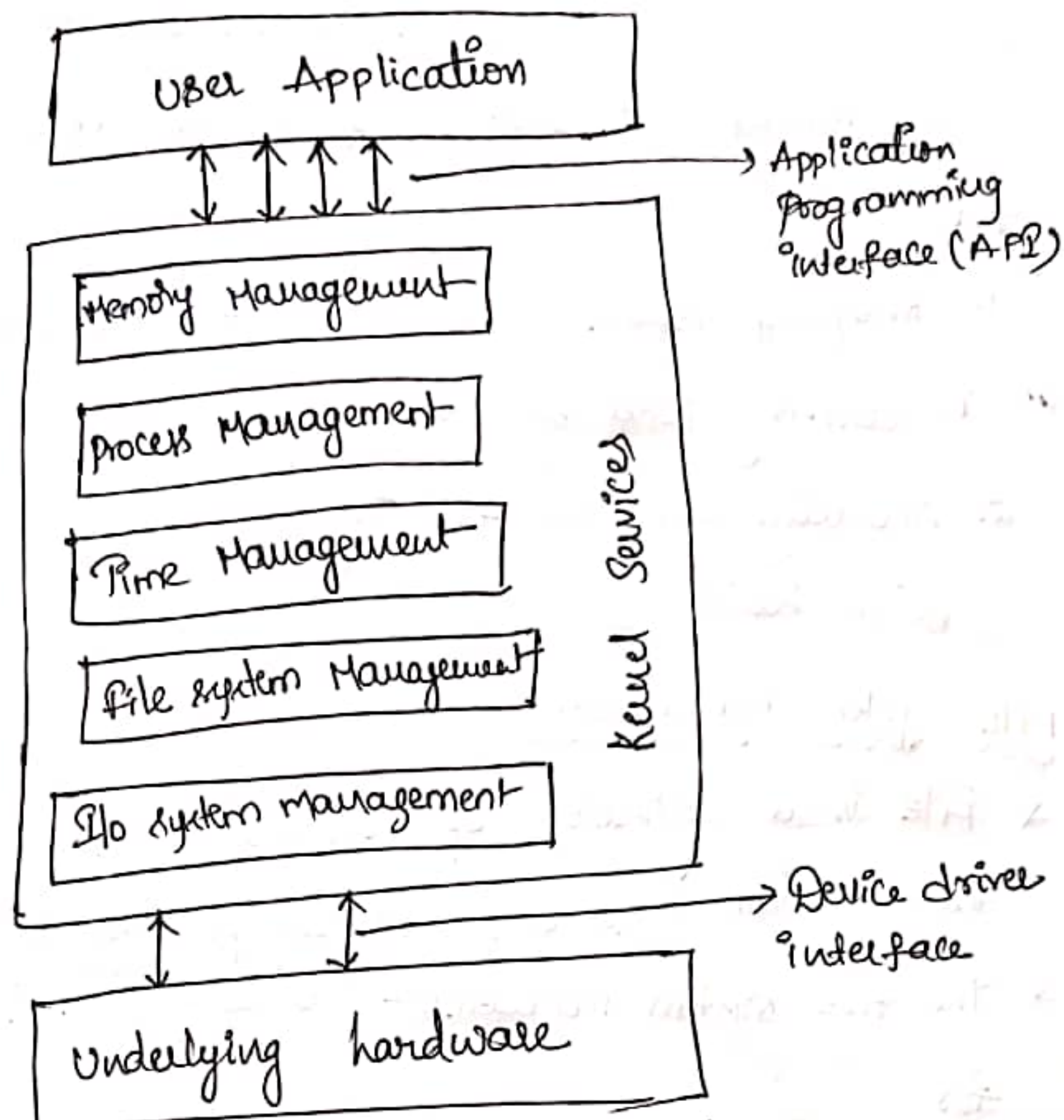


Fig: Operating System Architecture.

among the hardware and other system services.

* The kernel contains different services for handling the following.

Process Management:

* Process management deals with managing the processes/tasks. It includes setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, scheduling and managing the execution of the process, setting up and managing the process control block (PCB), InterProcess Communication and Synchronisation process termination / deletion etc.

Primary Memory Management

* Primary memory refers to the volatile memory (RAM) where processors are loaded and variables and shared

data associated with each process are stored. The memory management Unit (MMU) of the kernel is responsible for

1. keeping track of which part of the memory area is currently used by which process
2. Allocating and De-Allocating memory space on need. basis

File system Management:

* File is a collection of related information. The file operation is a useful service provided by the OS
* The file system management service of Kernel is responsible for

1. The creation, deletion and alteration of files
2. Creation, deletion and alteration of directories
3. Saving of files in the secondary storage memory
4. Providing automatic allocation of file space based on the amount of free space available
5. Providing a flexible naming convention for the files.

I/O system (Device) Management:

- * Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system.
- * The service 'Device Manager' of the kernel is responsible for handling all I/O device related operations.
- * The kernel talks to the I/O device through a set of low-

level system calls, which are implemented in a service, called device drivers.

* The Device Manager is responsible for

- a) loading & unloading of device drivers.
- b) exchanging information and the system specific control signals to and from the device.

Secondary Storage Management:

* The secondary storage Management deals with managing the secondary storage memory devices, if any, connected to the system.

* It is used as backup medium for programs and data since the main memory is Volatile.

* The secondary storage management service of kernel deals with

a) Disk storage allocation

b) Disk scheduling

c) Free Disk space Management

Protection Systems: Protection Deals with implementing the security policies to restrict the access to both user and system resources by different applications or processes or users.

* In multiuser supported operating systems, one user may not be allowed to view or modify the whole/ portions of another user's data or profile details.

* This kind of protection is provided by the protection services running within the kernel.

Interrupt Handler: Kernel provides handler mechanism for all external/ internal interrupts generated by the system.

* Depending on the type of the operating system, a kernel may contain lesser or more number of components/services

Kernel Space and User Space:

- * The applications / services are classified into two categories:
 - 1) user applications
 - 2) Kernel applications
- * The memory space at which the kernel code is located is known as 'kernel space'
- * User applications are loaded to specific area of primary memory and this memory area is referred as 'user space'. It is the memory area where user applications are loaded and executed.
- * The partitioning of memory into kernel and user space is purely operating system dependent.

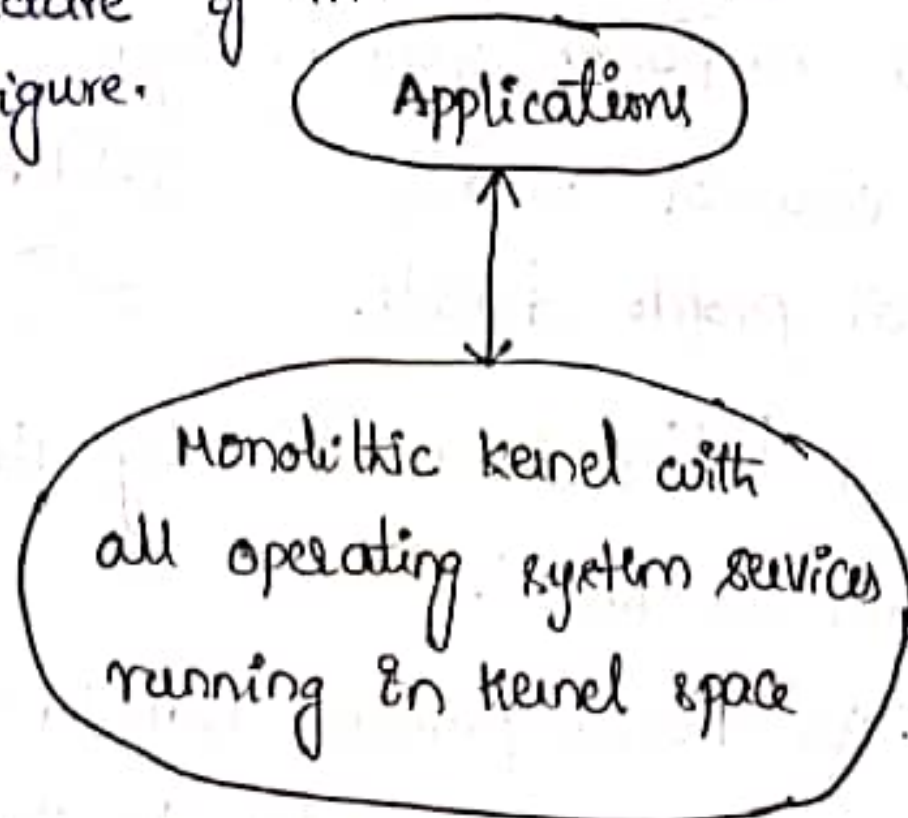
Monolithic Kernel and Micro Kernel

- * Different approaches are adopted for building an operating system kernel. The kernel forms the heart of an operating system.

- * Kernels can be classified into Monolithic and Micro

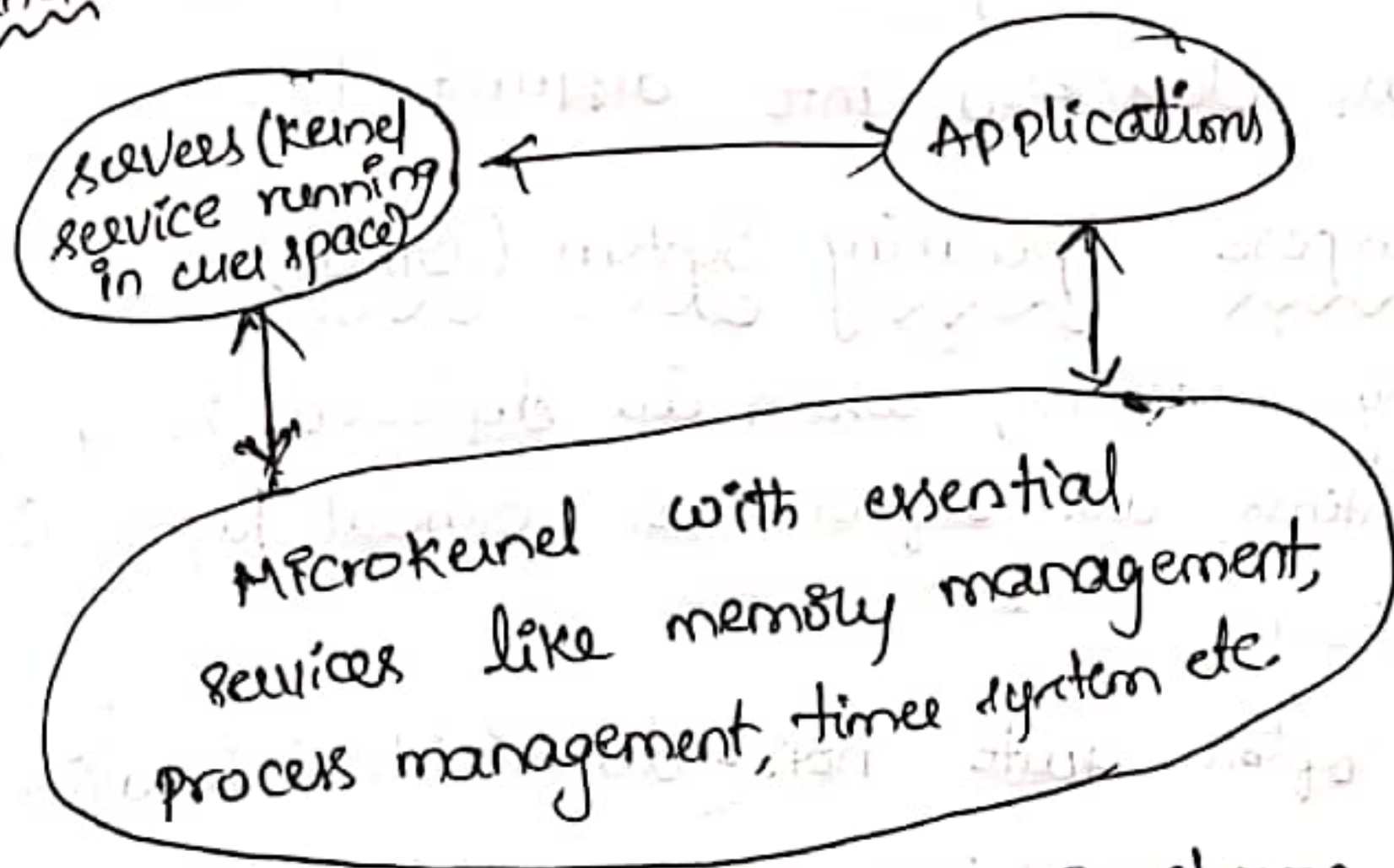
→ Monolithic kernel

The architecture of monolithic kernel is shown in the below figure.



- * In monolithic kernel architecture, all kernel services run in the kernel space. Here all kernel modules run within the same memory space under a single kernel thread.

- * The major drawback of monolithic kernel is that any error or failure in any one of the kernel module leads to the crashing of the entire kernel application.
- Micro kernel



- * The architecture of a micro kernel is shown in figure below.
- * The Microkernel design incorporates only the essential set of Operating System services into the kernel.
- * The rest of Operating System services are implemented in programs known as 'Servers' which runs in user space.
- * Memory management, process management, timer systems and interrupt handlers are the essential services; which forms the part of the Microkernel.
- * Microkernel based design approach offers the following benefits:
 - 1) Robustness:
 - 2) Configurability.

Types of Operating Systems:

- * Depending on the type of kernel and kernel services, purpose and type of computing systems where the OS is deployed and the responsiveness to applications, Operating Systems are classified into different types.

General Purpose Operating System (GPOS)

- * The Operating systems, which are deployed in general computing systems, are referred as General Purpose Operating Systems (GPOS).
- * GPOS are often quite non-deterministic in behaviour. Their services can inject random delays into application software and may cause slow responsiveness of an application at unexpected times.
- * GPOS are usually deployed in computing systems where deterministic behaviour is not an important criterion.
- * Personal Computer/ Desktop system is a typical example for a system where GPOSs are deployed.
- * Windows 10/8.x/XP/MS-DOS etc are examples for General Purpose Operating Systems.

Real-Time Operating System (RTOS)

- * Real-Time implies deterministic timing behaviour. It means the OS services consumes only known and expected amounts of time regardless the number of services.
- * RTOS implements policies and rules concerning time-critical allocation of a system's resources.
- * It decides which applications should run in which order and how much time needs to be allocated for each applica

features for
Additionally, the
light and has limited
capabilities

* Predictable performance is the hallmark of a well-designed RTOS.

* Embedded Compact, QNX, VxWorks MicroC/OS-II etc are examples of Real Time Operating Systems.

→ The Real-Time Kernel:

* The kernel of a Real-Time Operating system is referred as Real Time kernel.

* The real-time kernel is highly specialized and it contains only the minimal set of services required for running the user applications/tasks.

* The basic functions of a Real-Time kernel are listed below

1) Task/Process Management

2) Task/Process Scheduling

3) Task/Process Synchronization

4) Error/Exception handling

5) Memory management

6) Interrupt handling.

7) Time Management.

Task/Process Management:

* Deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources, setting up a Task Control Block (TCB) for the task and task/process termination/deletion. A Task Control Block (TCB) is used for holding the information corresponding.

* TCB usually contains the following set of information

1) Task ID → Task Identification Number

2) Task state → The current state of the task

* Task Type → indicates what is the type for this task.
The task can be a hard real time or soft real time or background task.

* Task priority:

* Task Context pointer: Pointer for context saving

* Task Memory pointers: Pointers to the code memory, data memory and stack memory for the task.

* Task System resource pointers: Pointers to system resources used by the task

* Task Pointers: Pointers to other TCBs.

* Other parameters: Other relevant task parameters.

→ Task Management service utilizes the TCB of a task in the following way:

1) Creates a TCB for a task on creating a task

2) Delete / remove the TCB of a task when the task is terminated or deleted

3) Reads the TCB to get the state of a task

4) Update the TCB with updated parameters on need basis

5) Modify the TCB to change the priority of the task dynamically.

Task/Process Scheduling:

* Deals with sharing the CPU among various tasks/Process.

* A kernel application called "Scheduler" handles the task scheduling.

* It is nothing but an algorithm implementation, which provides the efficient and optimal scheduling of tasks to provide

a deterministic behaviour.

Task/Process synchronization

- * Deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between the various tasks.

Error/Exception handling:

- * Deals with registering & handling the errors occurred / exceptions raised during the execution of tasks.

* Ex: Insufficient memory, timeouts, deadlocks, deadline missing, are the error / exception handling examples.

* Dead lock is an example for kernel level exception, ~~not~~ exception, whereas timeout is an example for a task level exception.

* The OS kernel gives the information about the error in the form of system call (API).

Memory Management:

* The memory management function of a RTOS kernel is slightly different from general purpose OS.

* The memory allocation time increases depending on the size of the block memory, needs to be allocated and the state of the allocated memory block.

* The predictable timing and deterministic behaviour of RTOS, it achieves by compromising the effectiveness of memory allocation.

* RTOS makes use of "block" based memory allocation technique, instead of the dynamic memory allocation techniques used by GPOS. The blocks are stored in a "Free Buffer Queue".

* To achieve the predictable timing and avoid the timing overheads, kernels allow tasks to access any of the memory block without any memory protection

* A few RTOS kernels implement "Virtual memory" concept for memory allocation if the system supports secondary memory storage

* In the block based memory allocation, a block of fixed memory is always allocated for tasks on need basis and it is taken as unit.

* The block memory concept avoids the garbage collection overhead.

Interrupt Handling

* Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.

* Interrupts can be either synchronous or asynchronous

* Interrupts which occurs synchronously with the currently executing task is known as synchronous interrupts.

It runs in the same context of the interrupting task

* Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.

* The interrupts generated by external devices connected to the processor/controller, timer overflow interrupts,

serial data reception/transmission interrupts are

examples for asynchronous interrupts.

* Priority levels can be assigned to the interrupts and each interrupt can be enabled or disabled individually

- * Most of the RTOS kernel implements 'Nested Interrupts' architecture. It allows the pre-emption of an interrupt service routine (ISR), servicing an interrupt, by a higher priority interrupt.

Time Management:

- * Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
- * Accurate time management is essential for providing precise time reference for all applications.
- * The time reference to kernel is provided by a high-resolution Real Time Clock (RTC) hardware chip (timer).
- * The hardware timer is programmed to interrupt the processor controller at a fixed rate. This timer interrupt is referred as 'Timer tick'.
- * The system time is updated based on the 'Timer tick' and the time parameters for tasks are expressed as the multiples of the 'Timer tick'.

- * ~~Whenever~~ The 'timer tick' interrupt is handled by the

'Timer interrupt' handler of kernel.

a) Hard Real Time System.

- * A Real Time Operating System which strictly adheres to the timing constraints for a task.
- * A Hard Real Time system must meet the deadlines for a task without any slippage.
- * Missing any deadline may produce catastrophic result for Hard Real Time systems, including permanent data loss & irrecoverable damages to the system/users.

- * Air bag control system and Anti-lock Brake system of vehicles are typical examples of Hard Real Time systems.

Soft Real-time System

- * Real Time Operating systems that don't guarantee meeting deadlines, but offer the best effort to meet the deadline.

- * Missing deadlines for tasks are acceptable if the frequency of deadline missing is within the Compliance limit of Quality of service.

- * Automatic Teller Machine is an example of soft Real time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.

Tasks, Processes & Threads

- * In the a) Task: operating system context, a task is defined as the program in execution and the related information maintained by the operating system for the program.

- * Task is also known as 'Job' in the operating system context. A program or part of it in execution is also called a 'process'.

- * The terms 'task', 'Job' and 'process' refer to the same entity in the operating system context and most often they are used interchangeably.

b) process

- * A process is a program or part of it in execution. Process is also known as an instance of a program in execution.

- * Multiple instances of the same program can execute.

features. Additionally, in light and has limited capabilities could be improved as a

simultaneously. A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc.

* A process is sequential in execution.

The structure of a process

* The concept of 'process' leads to concurrent execution of tasks and thereby the efficient utilisation of the CPU and other system resources.

* The concurrent execution is achieved through the sharing of CPU among the processes.

* A process mimics a processor in properties and holds a set of registers, process status, a program counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with process, and the code corresponding to the process as shown in fig.

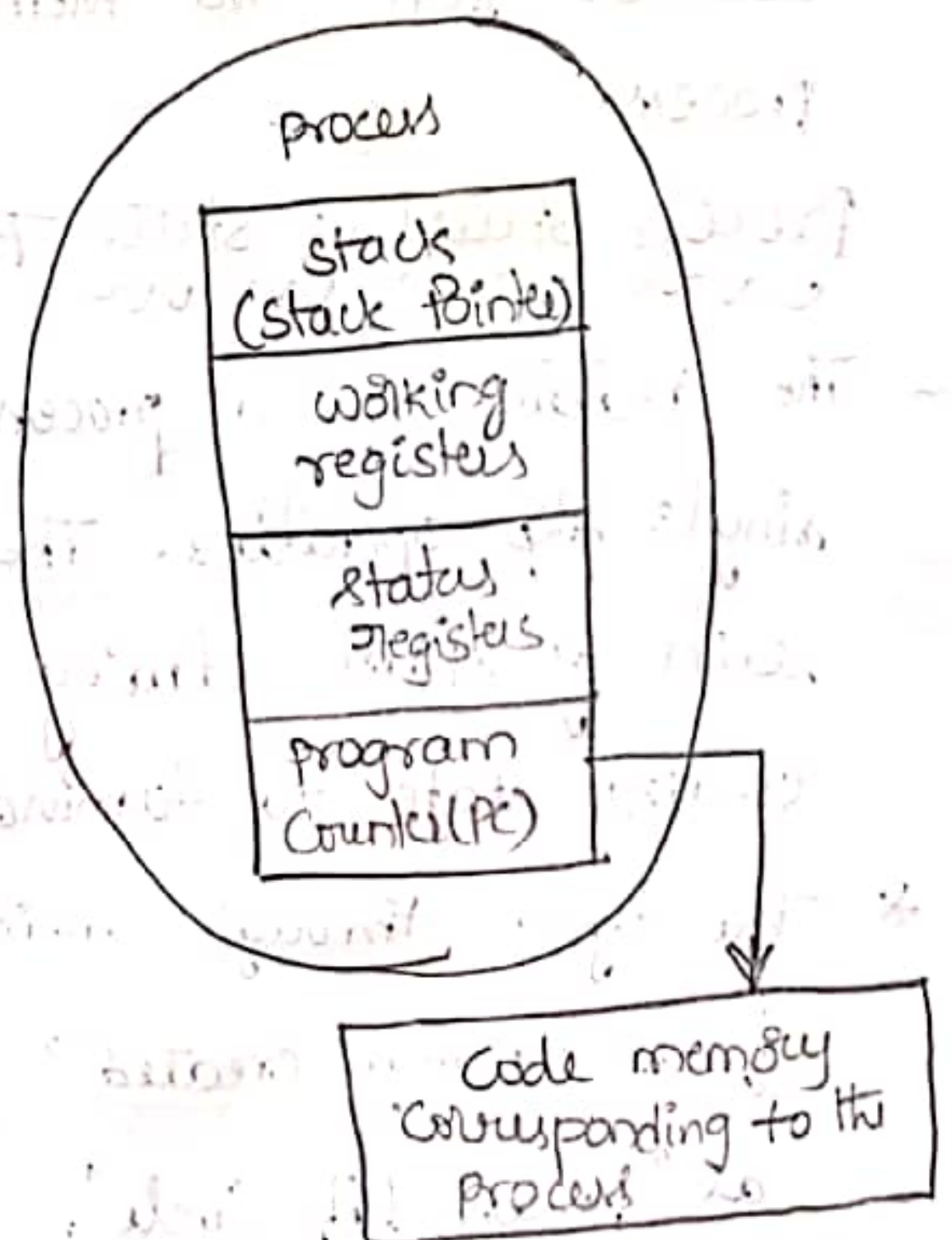
* A process which inherits all the properties of the CPU can be considered a virtual processor, awaiting its turn to have its properties switched into the physical processor.

* When the processor gets its turn, its registers and the program counter register

becomes mapped to the physical register of the CPU.

Memory Organisation of Process:

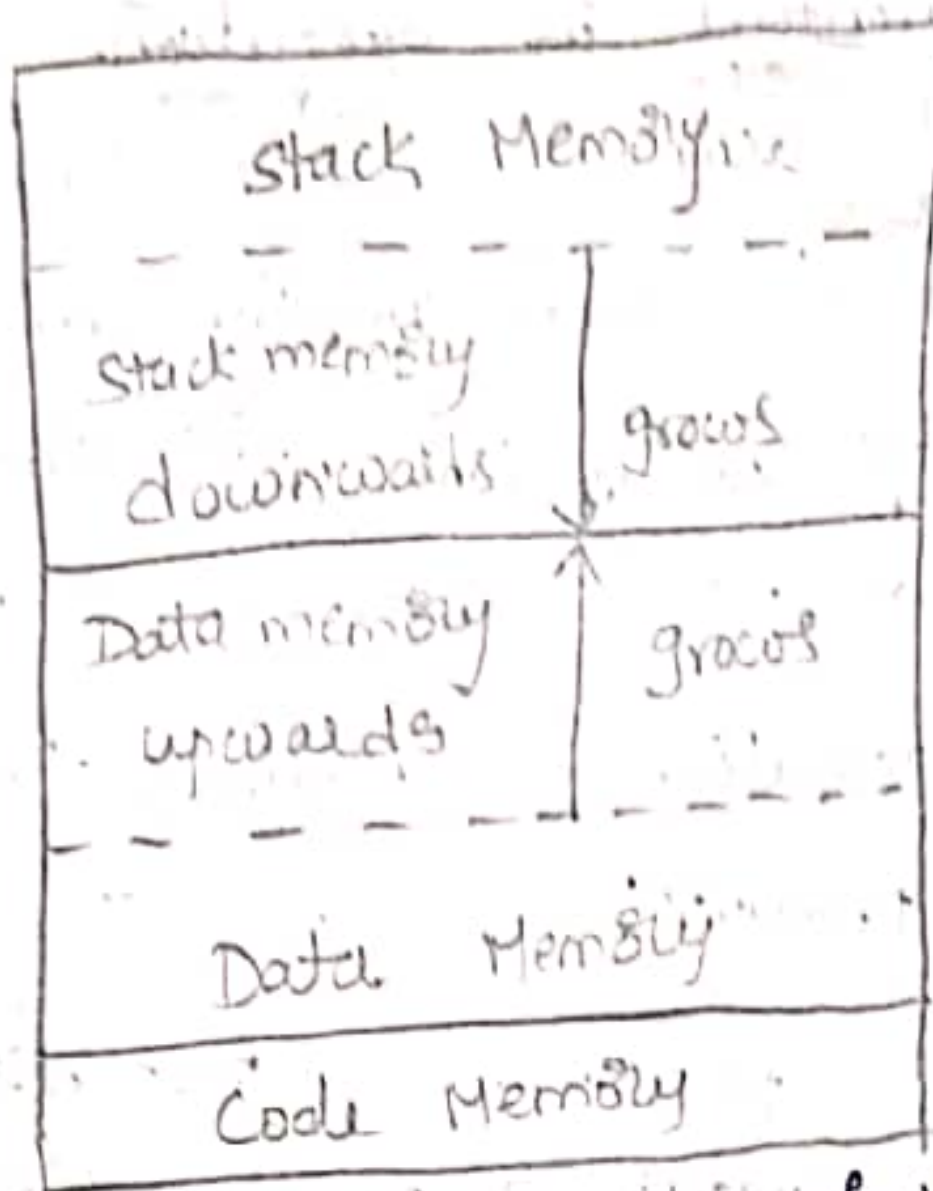
* The memory occupied by the process is segregated into 3 regions, namely, stack memory, Data memory and Code memory



* The 'stack' memory holds all temporary data such as variables local to the process.

* Data memory holds all global data for the process.

* The code memory contains the program code corresponding to the process.



* On loading a process into the main memory, a specific area of memory is allocated for the process.

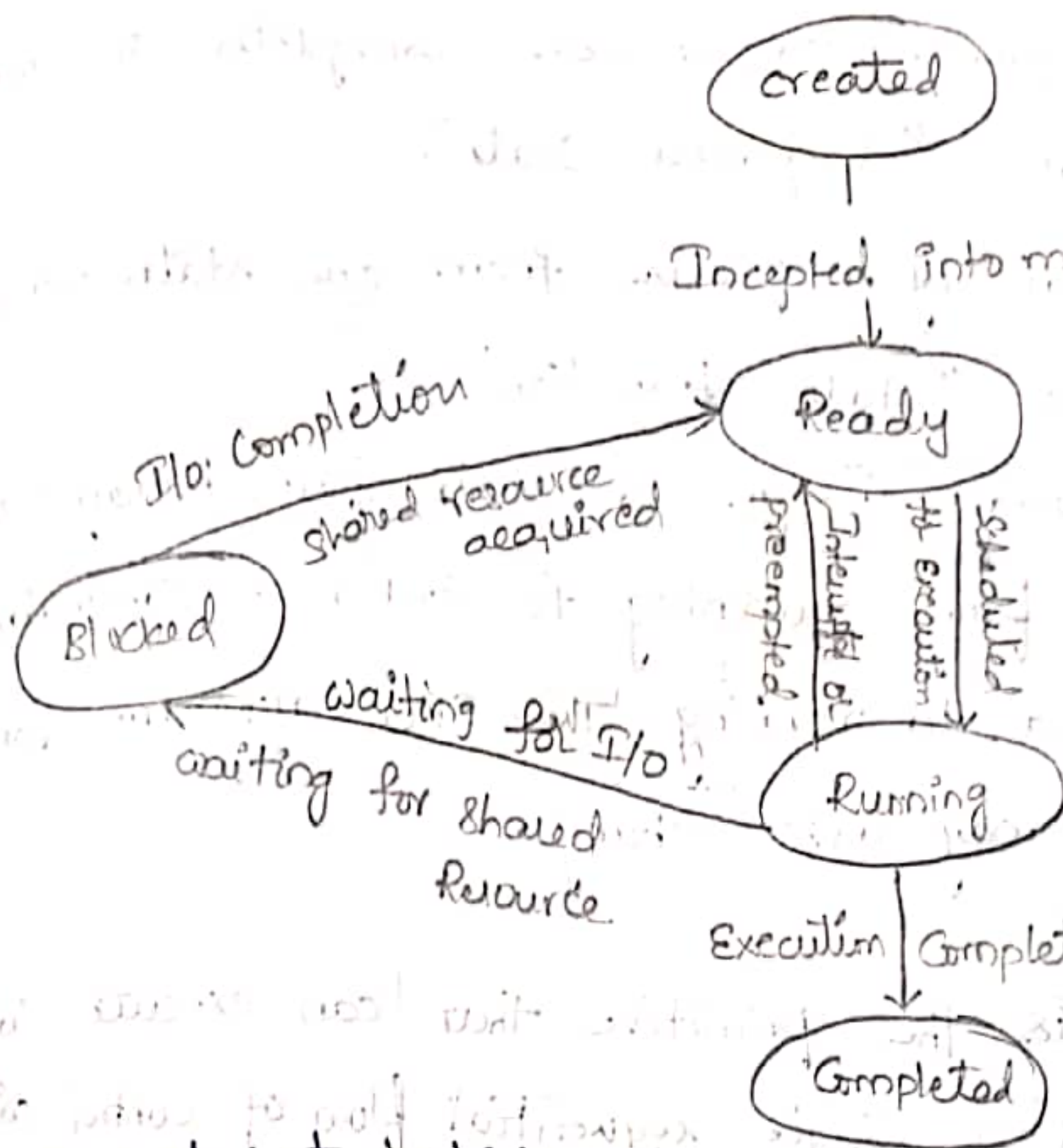
* The stack memory usually starts at the highest memory address from the memory area allocated for the process.

Process States & State Transition

* The creation of a process to its termination is not a single step operation. The process traverses through a series of states during its transition from the newly created state to terminated state.

* The cycle through which a process changes its state from 'newly created' to 'execution completed' is known as "process Life Cycle".

* The various states through which process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next.



- * The state at which a process is being created is referred as "Created state". The Operating System recognises a process in the "Created state" but no resources are allocated to the process.
- * The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as "Ready state". At this stage, the process is placed in the "Ready list" queue maintained by the OS.
- * The state where in the source code instructions corresponding to the process is being executed is called "Running state". Running state is the state at which the process execution happens.
- * "Blocked state/wait state" refers to a state where a running process is temporarily suspended from execution and doesn't have immediate access to resources.
- * The blocked state might be invoked by various conditions like: the process enters a wait state for an event to occur or waiting for getting access to a shared resource.

* A state where the process completes its execution is known as "Completed State".

* The transition of a process from one state to another is known as "state transition".

* When a process changes its state from Ready to running or from running to blocked or terminated from blocked to running, the CPU allocation for the process may also change.

Thread

* A thread is the primitive that can execute code.

* A thread is a single sequential flow of control within a process.

* "Thread" is also known as lightweight process. A process can have many threads of execution.

* Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area.

* Threads maintain their own thread status, program counter and stack. The memory model for a process and its threads are given in fig.

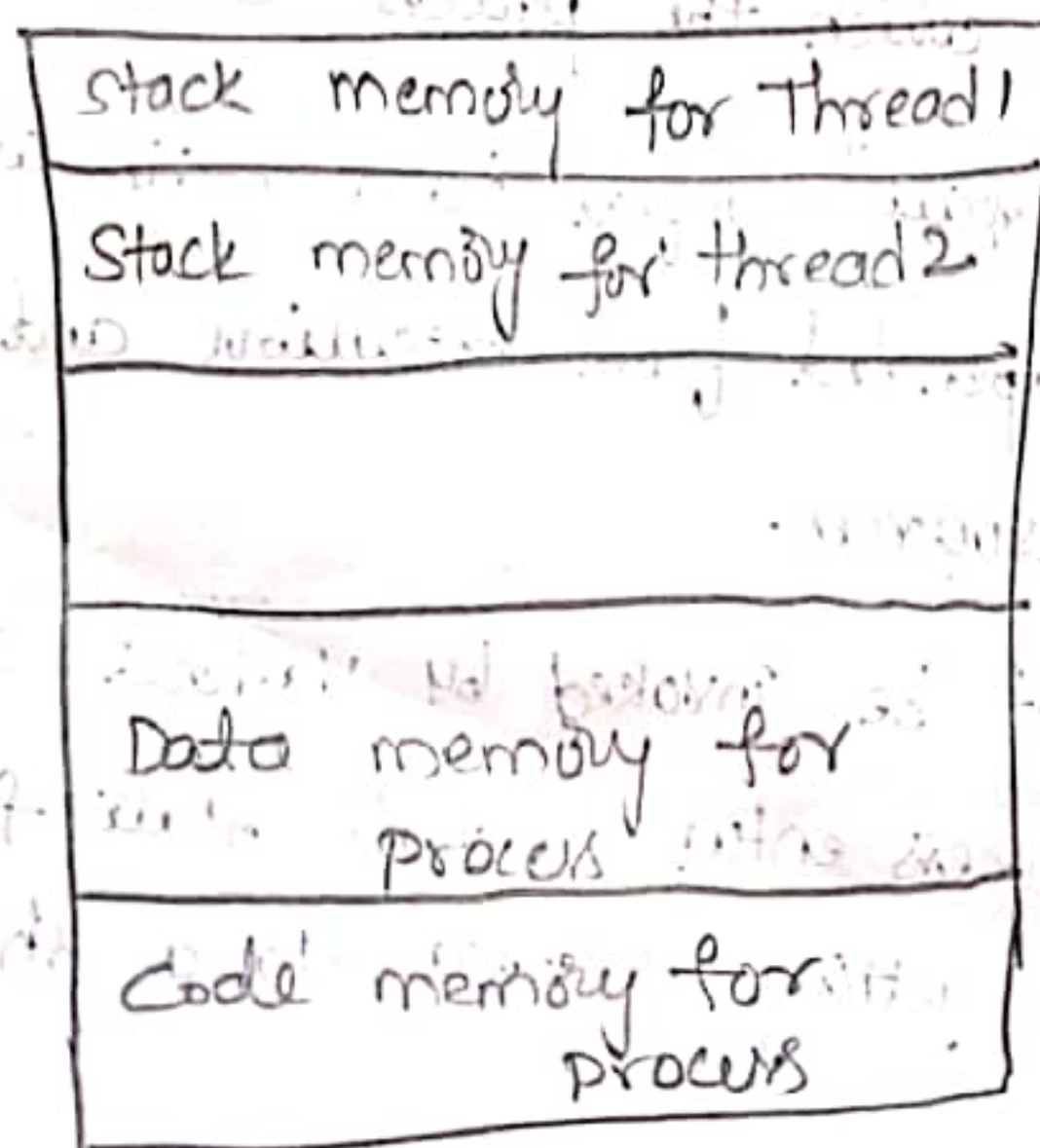


fig: Memory organization of a process and its threads

Multithreading:
 * Use of multiple threads to execute a process brings the following advantage

1) Better memory utilisation. Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.

2) Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilised by the threads of the process that do not require the event, which the other thread is waiting for processing. This speeds up the execution of the process.

* Efficient CPU utilisation. The CPU is engaged all time.

Task/Process

Code Memory		
Data Memory		
Stack	Stack	Stack
Registers	Registers	Registers
Thread 1	Thread 2	Thread 3

Thread Vs Process

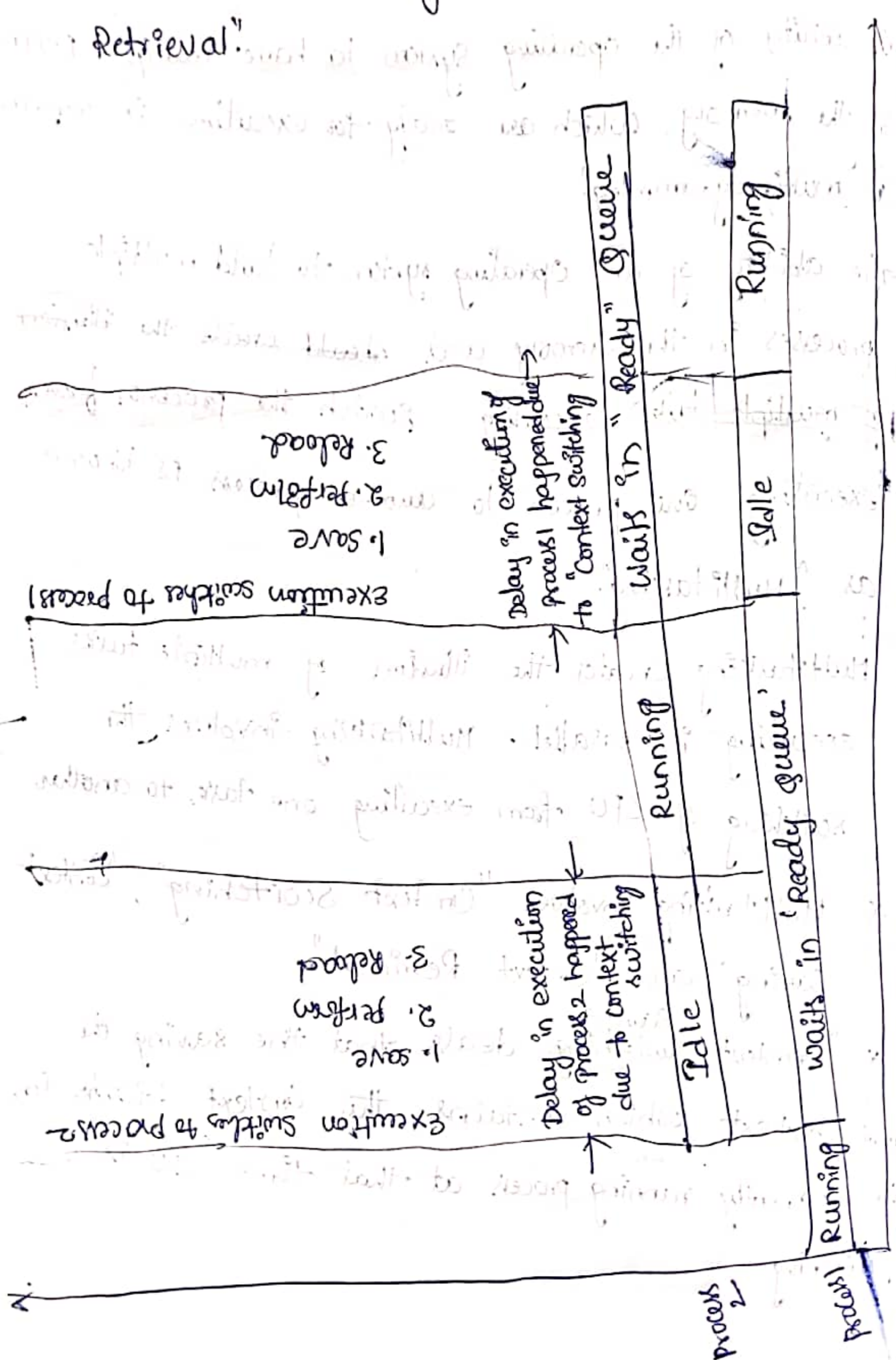
Threads	process
Thread is single unit of execution & is part of process	process is program in execution and contains one or more threads
A thread does not have its own data and heap memory. It shares the data memory & heap memory with other threads of the same process.	process has its own code memory, data memory and stack memory.
A thread can't live independently; it lives within the process.	A process contains at least one thread.
There can be multiple threads in a process. The first thread calls the main function and occupies the start of the stack memory of the process.	Threads within a process can share the code, data & heap memory. Each thread holds separate memory area for stack.
Threads are very inexpensive to create.	Processes are very expensive to create. Involves many OS overhead.
Context switching is inexpensive & fast.	Context switching is complex and involves lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If the process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also die.

capabilities Multiprocessing and Multitasking:-

- * The ability to execute multiple processes simultaneously referred as 'multiprocessing'.
- * Systems which are capable for performing multiprocessing are known as 'multiprocessor systems'.
- * Multiprocessor systems possess multiple CPUs and can execute multiple processes simultaneously.
- * The ability of the operating system to have multiple programs in the memory, which are ready for execution is referred as 'multiprogramming'.
- * The ability of an operating system to hold multiple processes in the memory and ~~should create the illusion~~ ~~of multiple tasks executing~~ switch the processor from 'executing one process to another process' is known as 'Multitasking'.
- * Multitasking creates the illusion of multiple tasks executing in parallel. Multitasking involves the switching of CPU from executing one task to another.
- * Multitasking involves "Context Switching", "Context Saving" and "Context Retrieval".
- * "Context ^{Saving} Switching" deals with the saving of the current context, which contains the context details for the currently running process at that time. CPU is switching.

* "Context Switching" refers that the act of Switching CPU among the processes or changing the current execution Context.

* The process of retrieving the saved Context details for a process, which is going to be executed due to CPU switching is known as "Context Retrieval".



Multitasking

- * Multitasking involves the switching of execution among multiple tasks. Depending on how the switching act is implemented, multitasking classified into 3 types.

Co-operative Multitasking:

- * Co-operative multitasking is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU.

- * Any task/process can hold the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, is known as co-operative multitasking.

- * If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU.

Pre-emptive Multitasking

- * Pre-emptive multitasking ensures that every task/process gets a chance to execute.

- * When and how much time a process gets is dependent on the implementation of pre-emptive scheduling.

- * The pre-emption of task may be based on time slots or task/process priority.

Non-pre-emptive Multitasking:

- * The process/task, which is currently given the CPU time, is allowed to execute until it terminates or enters the "Blocked/wait" state, waiting for an I/O.

* The co-operative and non-preemptive multitasking differs in their behaviour when they are "Blocked/ wait state".

* In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the 'Block/wait' state, waiting for an I/O, or a shared resource access or an event to occur where as in non-preemptive multitasking, the currently executing task relinquishes the CPU when it waits for an I/O.

Task scheduling:

- * In a multi-tasking system, there should be some mechanism in place to share the CPU among the different tasks to decide which process/task is to be executed at a given point of time.
- * Determining which task/process is to be executed at a point of time is known as task/process scheduling.
- * Task scheduling forms the basis of multitasking. Scheduling policies forms the guidelines for determining which task is to be executed when.
- * The scheduling policies are implemented in an algorithm and it is run by the kernel as a service.
- * The kernel service/application which implements the scheduling algorithm, is known as "scheduler".
- * The process scheduling decision may take place when a process switches to its state to
 - 1) 'Ready' state from 'Running state'

2. 'Blocked/wait' state from 'Running' state
3. 'Ready' state from 'Blocked/wait' state
4. 'Completed' state.

Task scheduling - scheduler selection

* The selection of a scheduling criterion/algorithm should consider the following factors:

→ CPU Utilization: The scheduling algorithm should always make the CPU utilisation high. It is ~~for~~ a direct measure of how much percentage of the CPU is being utilized.

→ Throughput: This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.

→ Turnaround Time: It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, and the time spent in execution. The turnaround time should be a minimal for a good scheduling algorithm.

→ Waiting Time: It is the amount of time spent by a process in the "Ready" queue waiting to get the CPU time for execution. The waiting time should be minimal for a good scheduling algorithm.

→ Response Time: It is the time elapsed between the submission of a process and the first response. For a good scheduling algorithm, the response time should be as least as possible.

Task scheduling - Queues

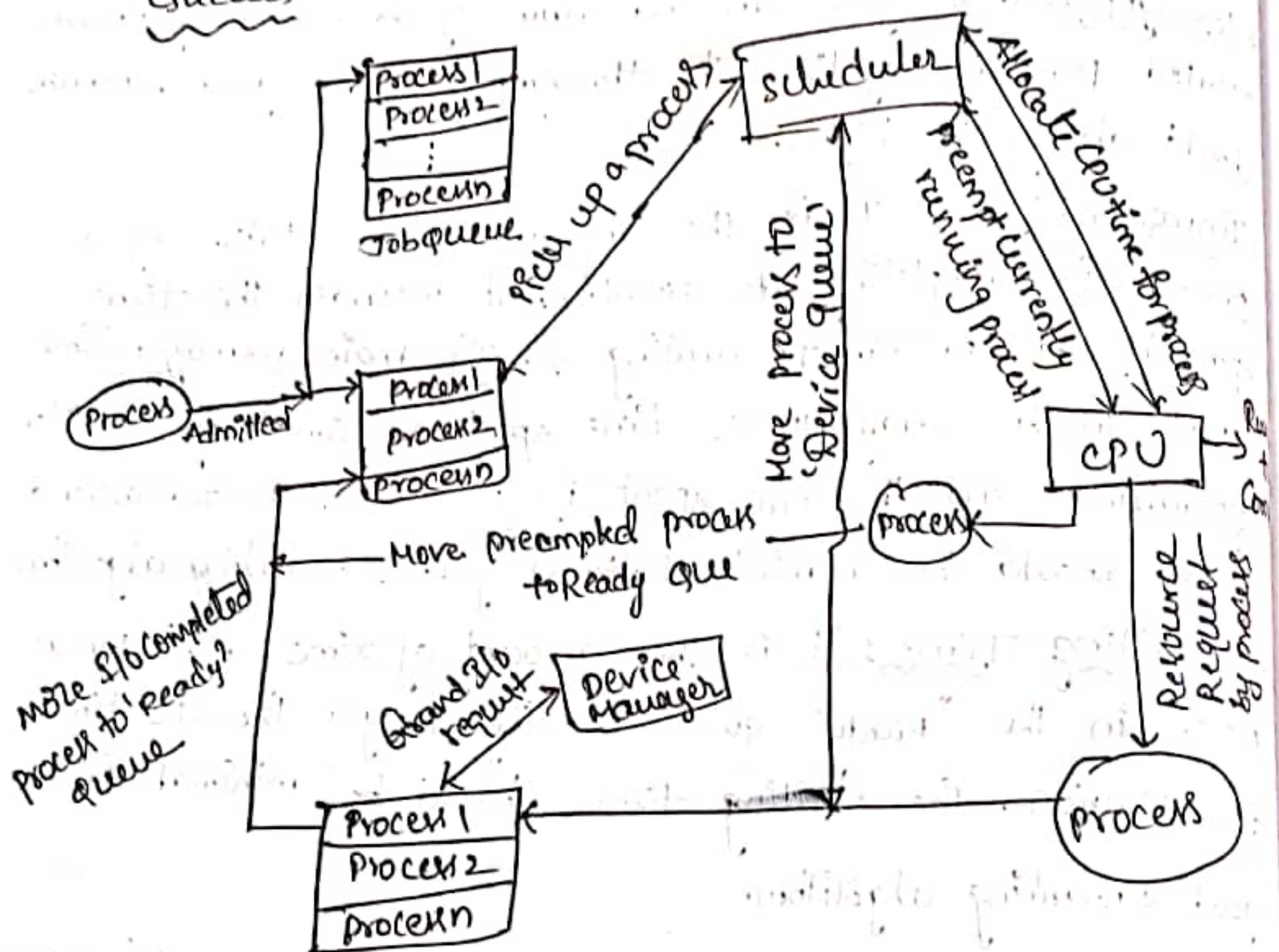
* The various queues maintained by OS in association with CPU scheduling are:

→ Job queue: Job queue contains all the processes in the system.

Ready Queue: Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution. The ready queue is empty when there is no process ready for running.

Device Queue: Contains the set of processes, which are waiting for an I/O Device.

Task Scheduling - Task transition through Various Queues



Non preemptive scheduling - First Come First Served (FCFS) / First In First Out (FIFO) scheduling

- * Allocates CPU time to the processes based on the order in which they enter the 'Ready' state.
- * The first entered process is served first.
- * It is the same as any real world application.

where queue systems are used. Eg: Ticketing

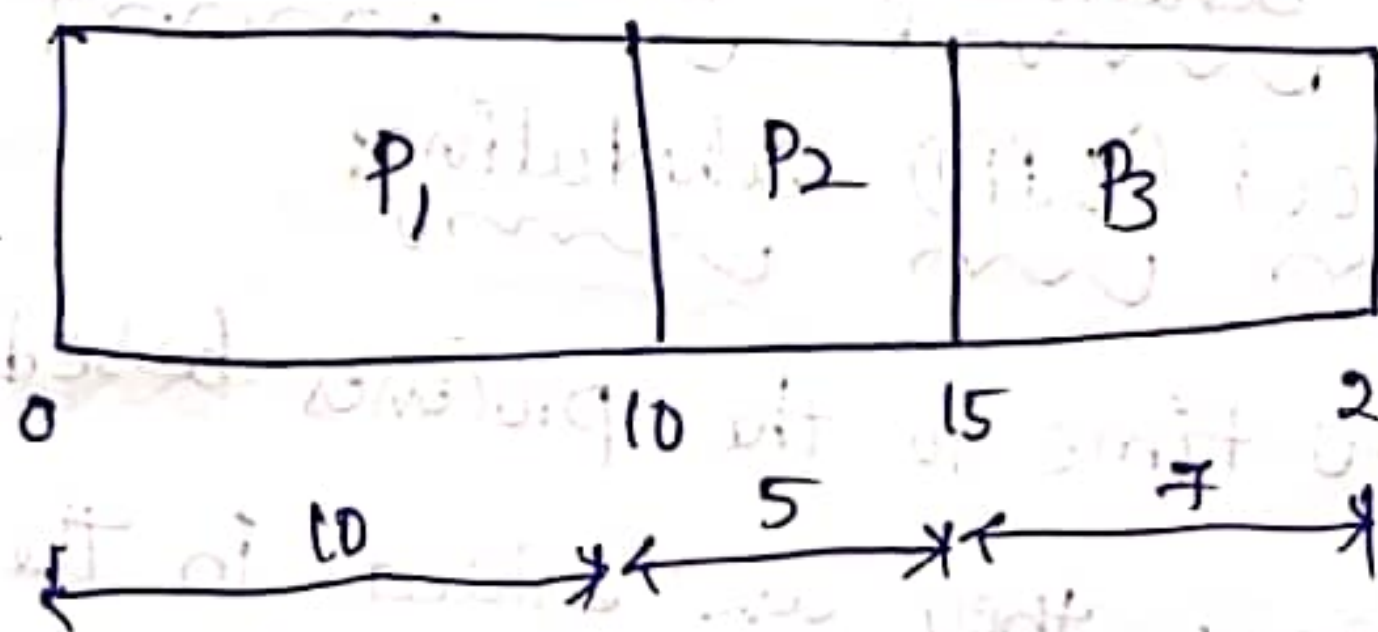
Drawbacks:

1. favors monopoly of process. A process, which doesn't contain any I/O operation, continues its execution until it finishes its task.
 2. In general, FCFS favors CPU bound processes & I/O bound processes may have to wait until the completion of CPU bound process; if the currently executing process is a CPU bound process. This leads to poor device utilization.
- * The avg waiting time is not minimal for FCFS scheduling algorithm.

Example: Three Processes with process IDs P_1, P_2, P_3 with estimated completion time 10, 5, 7 ms respectively enter the ready queue together in the order P_1, P_2, P_3 .

Calculate the waiting time and Turn Around Time for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes)

Sol The sequence of execution of the process by the CPU is represented as



Assuming the CPU is readily available at the time of

arrival of P_1, P_2 starts executing without any waiting in the "Ready" queue. Hence the waiting time for P_1 is zero.

Waiting Time for $P_1 = 0\text{ms}$ (P_1 starts executing first)

Waiting Time for $P_2 = 10\text{ms}$ (P_2 starts executing after completing P_1)

Waiting Time for $P_3 = 15\text{ms}$ (P_3 starts executing after completing P_1 & P_2)

Average waiting time = (waiting time for all processes) / No. of processes

$$= (\text{waiting time for } (P_1 + P_2 + P_3)) / 3$$

$$(0 + 10 + 15) / 3 = 8.33\text{ms}$$

Turn Around Time (TAT) for $P_1 = 10\text{ms}$ (Time spent in Ready queue + Execution Time)

$$\text{TAT for } P_2 = 10 + 5 = 15\text{ms}$$

$$P_3 = 15 + 7 = 22\text{ms}$$

Average Turn Around Time =

(Turn Around Time for all processes) / No. of processes

$$= (\text{TAT for } (P_1 + P_2 + P_3)) / 3$$

$$= (10 + 15 + 22) / 3 = 47 / 3$$

$$= 15.66\text{ms}$$

Non preemptive Scheduling - Last Come First served (LCFS)

Last in first out (LIFO) Scheduling:

* Allocates CPU time to the processes based on the order in which they are entered in the 'Ready' queue.

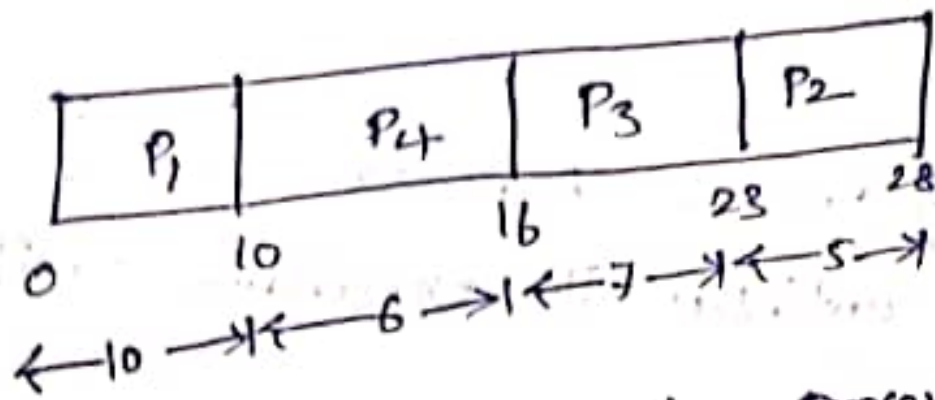
- * The last entered process is serviced first
- * LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the 'Ready' queue, is serviced first

Drawbacks:

1. Favors monopoly of process. A process, which doesn't contain any I/O operation, continues its execution until it finishes its task.
 2. In general, LCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process. This leads to poor device utilization.
 3. The avg waiting time is not minimal for LCFS scheduling algorithm.
- Ex Problem Three processes with process IDs, P_1, P_2, P_3 with estimated completion time 10, 5, 7 ms respectively enters the ready queue together in the order P_1, P_2, P_3 (Assume only P_1 is present in the 'Ready' queue when the scheduler picks up it and P_2, P_3 entered 'ready' queue after that). Now a new process P_4 with estimated completion time 6ms enters the Ready queue after 5ms of scheduling P_1 . Calculate the waiting time and Turn Around Time for each process and the Average waiting Time and Turn Around time (Assuming there is no I/O waiting for the process). Assume all the processes contain only CPU operation and no I/O operations are involved.

Sol Initially there is only P_1 available in the Ready queue and the scheduling sequence will be P_1, P_3, P_2, P_4

enters the queue during the execution of P_1 & becomes the last process entered the 'Ready' state. Now the order of execution changes to P_1, P_4, P_3 & P_2 as given below



The waiting time for all the processes are given as
 waiting Time for $P_1 = 0\text{ms}$ (P_1 starts executing first)
 waiting Time for $P_4 = \text{Execution start time} - \text{Arrival Time}$
 $= 10 - 5 = 5$

(P_4 starts executing after completing P_1 . But P_4 arrived after 5ms of execution of P_1).

waiting Time for $P_3 = 16\text{ms}$ (P_3 starts executing after completing P_1 & P_4)

$P_2 = 23\text{ms}$

~~$P_2 = 23\text{ms}$~~

Avg time for waiting = $\frac{\text{waiting time for all processes}}{\text{No. of processes}}$

= $\frac{\text{waiting time for } (P_1 + P_4 + P_3 + P_2)}{4}$

$$= \frac{0 + 5 + 16 + 23}{4} = \frac{44}{4} = 11\text{ms}$$

Turn-Around Time (TAT) for $P_1 = 10\text{ms}$

$P_4 = 11\text{ms}$ (Time spent in Ready queue)

Execution Time = (Execution

start Time - Arrival Time) +

Estimated Execution Time.

$$= 10 - 5 + 6 = 5 + 6$$

$$P_3 = 10 \text{ ms}$$

$$P_4 = 2.8 \text{ ms}$$

$$\text{Avg} = \frac{\text{TAT for } (P_1 + P_2 + P_3 + P_4)}{4} = \frac{10 + 11 + 23 + 2.8}{4}$$

$$= \frac{46.8}{4} = 11.7 \text{ ms}$$

Non-preemptive scheduling - shortest Job first (SJF)
Scheduling:

* Allocates CPU time to the processes based on the execution completion time for tasks

* The avg waiting time for a given set of processes is minimal in SJF scheduling

* Optimal compared to other non-preemptive scheduling like FCFS

Drawbacks:

* A process whose estimated execution completion time is high may not get a chance to execute if more & more

processes with least estimated execution

* The avg waiting time for a given set of processes is minimal in SJF scheduling

* Optimal compared to other non-preemptive scheduling like FCFS

Drawbacks:

1) A process whose estimated execution completion time is high may not get a chance to execute if more & more processes with least estimated execution time enter the

"Ready" queue before the process with longest estimated execution time starts its execution.

2) May lead to the "starvation" of processes with high estimated completion time.

3) Difficult to know in advance the next shortest process in the 'Ready' queue for scheduling since new processes with different execution time keep entering the 'Ready' queue at any point of time.

Non-preemptive scheduling - priority based scheduling

- * A priority, which is unique or same is associated with each task.

- * The priority of a task is expressed in different ways like a priority number, the time is required to

complete the execution etc

- * In number based priority assignment the priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent.

- * Windows CE supports 256 levels of priority (0 to 255)

- * The priority is assigned to the task on creating it.

It can also be changed dynamically

- * The non-preemptive priority based scheduler sorts the "Ready queue" based on the priority and picks the process with the highest level of priority for execution.

Drawbacks:

- * Similar to SJF scheduling algorithm, non-preemptive priority based algorithm also poses the drawback of

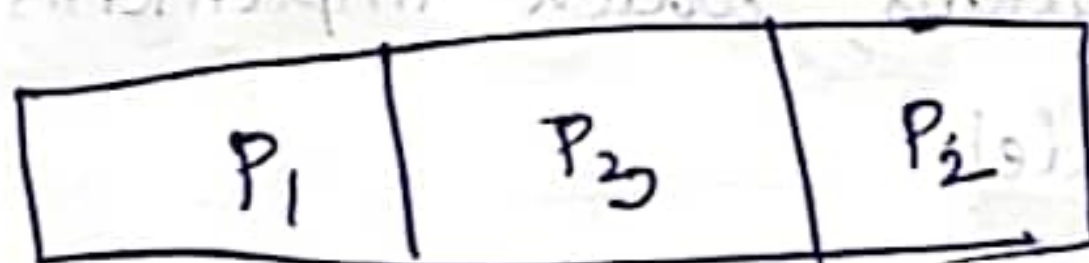
and has limited abilities could...

'starvation' where a process whose priority is low may not get a chance to execute if more and more processes with higher priorities enter the 'Ready' queue before the process with lower priority starts its execution.

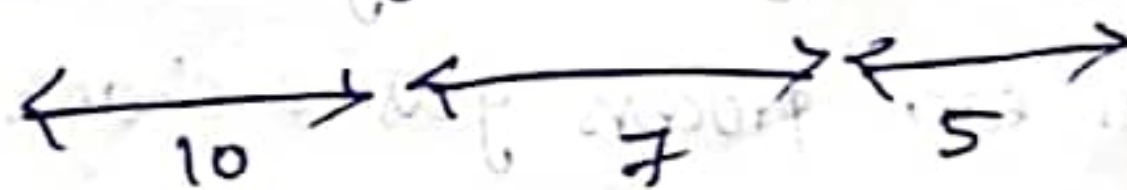
- * 'starvation' can be effectively tackled in priority based non-preemptive scheduling by dynamically raising the priority of the low priority task/process which is under starvation
- * The technique of gradually raising the priority of the processes which are waiting in the 'Ready' queue as time progresses, for preventing 'starvation' is known as 'Aging'

Problem: Three processes with process IDs P_1, P_2, P_3 with estimated Completion 10, 5, 7 ms. and priorities 0, 3, 2 (0 - highest priority, 3 - lowest priority) respectively enter the ready queue together. Calculate the waiting time and Turn around Time (TAT) for each process and the Average waiting Time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.

Sol



0 10 17 22



- * The scheduler sorts the Ready queue based on the priority and schedules the process with the highest priority and schedules the process with the highest priority (P_1 with priority number 0) first and the next high priority process (P_3 with priority number 2) as second and soon. The order in which the processes are scheduled for execution is represented as

The waiting time for all the processes are given
 waiting time for $P_1 = 0\text{ms}$ (P_1 starts executing first)
 $P_3 = 10\text{ms}$ (P_3 " " after P_1)
 $P_2 = 17\text{ms}$ (P_2 " " after P_3)

$$\text{Avg waiting time} = \left(\frac{\text{waiting time for all processes}}{\text{No. of processes}} \right)$$

$$= \frac{(P_1 + P_2 + P_3)}{3} = \frac{0 + 17 + 10}{3} = 9\text{ms}$$

* Turn Around time (TAT) for $P_1 = 10\text{ms}$ (Time spent in Ready queue + Execution Time)

$$\text{for } P_3 = 17\text{ms} (10 + 7)$$

(Time spent in Ready queue + Execution Time)

$$\text{for } P_2 = 17 + 5 = 22\text{ms}$$

Average Turn Around Time = (Turn Around time for all processes) / No of processes

$$= \frac{10 + 17 + 22}{3} = \frac{49}{3}$$

$$= 16.33\text{ms}$$

Preemptive Scheduling

* Employed in systems, which implements preemptive multitasking model.

* Every task in the 'Ready' queue gets a chance to execute.

When & how often each process gets a chance to execute

is dependent on the type of preemptive scheduling algorithm used for scheduling the processes.

* The scheduler can preempt (stop temporarily) the currently executing task/process & select another from the 'Ready' queue for execution.

* When to pre-empt a task and which task is to be picked up from 'Ready' Queue for execution after preempting the current task is purely dependent on the scheduling Algorithm.

* A task which is preempted by the scheduler is moved to the 'Ready' Queue. The act of moving a "Running" process/task into the 'Ready' Queue by the scheduler, without the processes requesting for it is known as "preemption".

* Time based pre-emption & Priority based Preemption are two important approaches.

Preemptive SJF scheduling / shortest Remaining Time (SRT)

* The non-preemptive SJF scheduling algorithm sorts the 'Ready' queue only after completing the execution of the current process or when the process enters "wait" state, whereas the preemptive SJF scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated time for the currently executing process.

* If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for algorithm.

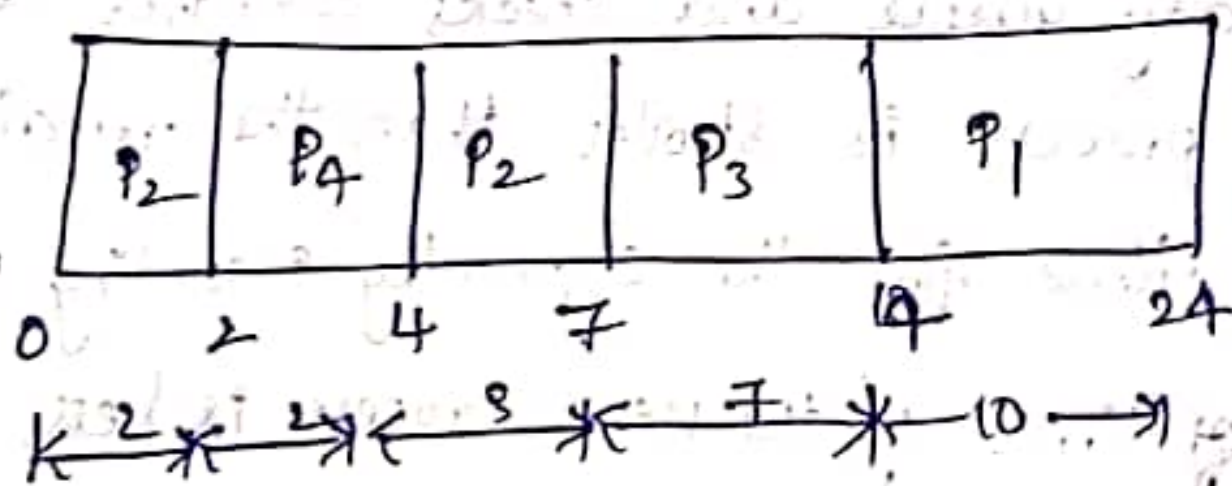
* Always compares the execution completion time of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution.

Ex problem:

Three processes with process IDs P_1, P_2, P_3 with estimated completion time 10, 5, 7 ms respectively enter the ready queue together. A new process P_4 with estimated completion time 2 ms enters the 'Ready' queue after 2 ms. Assume all the processes contain only CPU operation and no I/O operations are involved.

Sol At the beginning, there are only three processes (P_1, P_2 & P_3) available in the 'Ready' queue and the scheduler picks up the process with the shortest remaining time for execution completion (In this example, P_2 with remaining time 5 ms) for scheduling the execution sequence diagram for this is as follows:

Now P_4 process with estimated execution completion time 2 ms enters the 'Ready' queue after 2 ms of start execution of P_2 .



$$\text{Waiting Time } P_2 = 0 + (4 - 2) \text{ ms} \\ = 2 \text{ ms}$$

(P_2 starts executing 1st and is interrupted by P_4 and has to wait till the completion of P_4 to get the next CPU slot)

Time for $P_4 = 0 \text{ ms}$ [P_4 starts execute by preempting P_2 execution time for completion P_4 less than that Remaining time for completion 5 ms]

$P_3 = 7 \text{ ms}$

Waiting time for $P_1 = 14 \text{ ms}$.

$$\text{Average wait time} = \frac{\text{waiting time for all the processes}}{\text{No. of process}}$$

$$= \frac{0 + 2 + 7 + 14}{4} = 5.75 \text{ ms.}$$

Turn Around time for $P_2 = 7 \text{ ms}$

(Time spent in Ready queue + Execution time)

$$\text{TAT for } P_4 = 2 \text{ ms} \quad \left[\begin{array}{l} \text{Time spent in Ready queue + Execution Time} \\ = (\text{Execution start time} - \text{Arrival time}) + \\ \text{Estimated Execution time} = ((2 - 2) + 2) = 2 \end{array} \right]$$

TAT for $P_3 = 14 \text{ ms}$

TAT for $P_1 = 24 \text{ ms}$.

$$\text{Avg TAT} = \frac{7 + 2 + 14 + 24}{4}$$

$$= \frac{47}{4} = 11.75 \text{ ms.}$$

Round Robin (RR) scheduling

* Each process in the 'Ready' queue is executed for a pre-defined slot.

* The execution starts with picking up the first process in the 'Ready' queue. It is executed for a pre-defined time.

* when the pre-defined time elapses

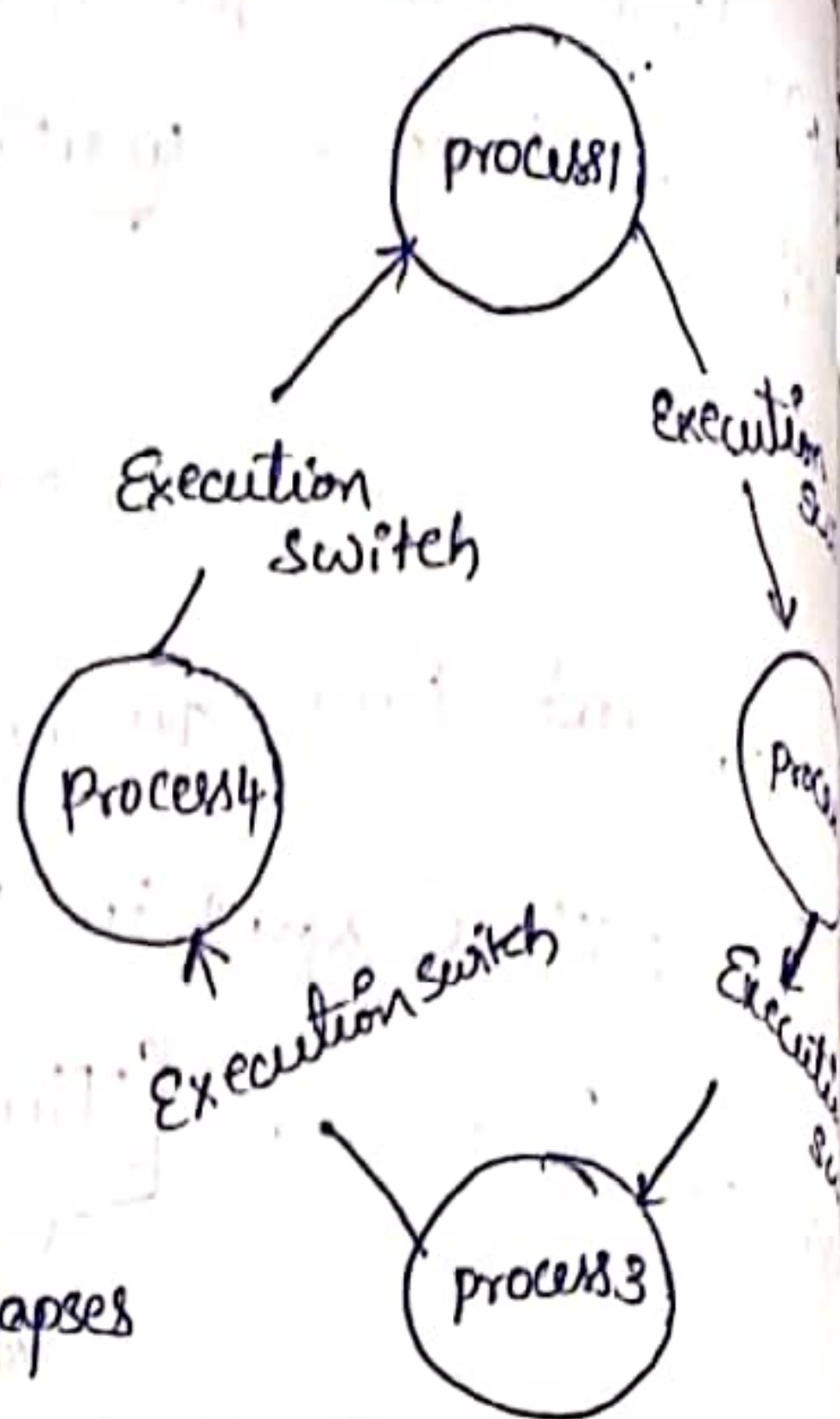
or the process completes the next process in the 'Ready' queue is selected for execution.

* This is repeated for all the processes in the 'Ready' queue. Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution.

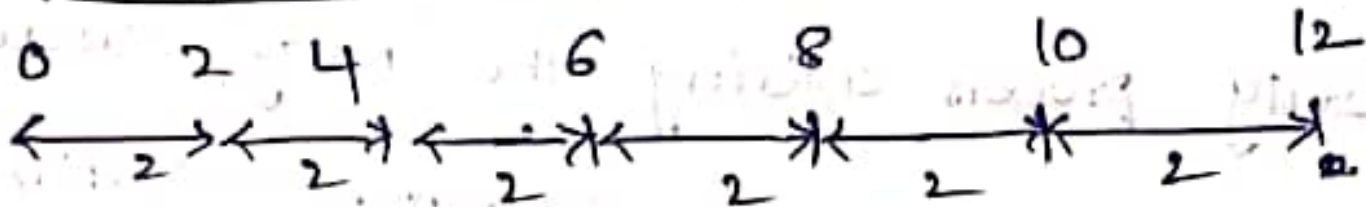
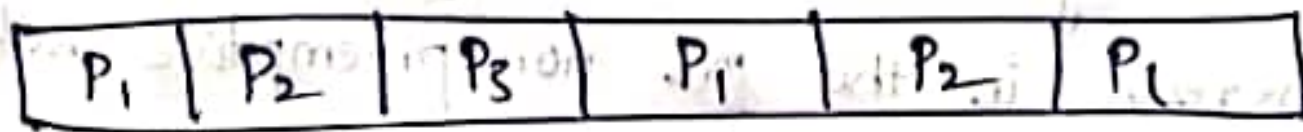
* The 'Ready' queue can be considered as a circular queue in which the scheduler picks up the process for execution and moves to the next till end of the queue and then comes back to the beginning of the queue to pick up the first process.

Ex problem:

Three processes with process IDs P_1, P_2, P_3 with execution completion time 6, 4, 2 ms respectively, enter the ready queue together in the order P_1, P_2, P_3 . Calculate the waiting time and Turn Around time for each process and the Average waiting Time & Average Turn around time in RR with Time slice = 2ms.



sol. The scheduler sorts the 'Ready' queue based on the FCFS policy & picks up the process P_1 from the 'Ready' queue & executes it for the time slice 2ms. When the time slice is expired; P_1 is preempted and P_2 is scheduled for execution. The time slice expires after 2ms of execution of P_2 . Now P_2 is preempted and P_3 is picked up for execution. P_3 completes its execution within the time slice and the scheduler picks P_1 again.



waiting time for $P_1 = 0 + (6-2) + (10-8) = 0 + 4 + 2 = 6 \text{ ms.}$

$P_2 = (2-0) + (8-4) = 2 + 4 = 6 \text{ ms.}$

$P_3 = (4-0) = 4 \text{ ms}$

Avg waiting time = $\frac{6+6+4}{3} = \frac{16}{3} = 5.33 \text{ ms.}$

Turn Around time for $P_1 = 12 \text{ ms}$ (Time spent Ready queue + Execution time)

$P_2 = 10 \text{ ms}$

$P_3 = 6 \text{ ms}$

Avg Turn Around time = $\frac{12+10+6}{3} = \frac{28}{3} = 9.33 \text{ ms}$

Priority based scheduling:

* Priority based preemptive scheduling algorithm is same as that of the non-preemptive priority based scheduling except for the switching of execution between tasks.

* In pre-emptive scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the ~~pre~~ non-preemptive scheduling high priority process entering the 'Ready' queue is scheduled ~~for~~ only after the currently executing process completes its execution.

* Completes the execution or only when it is voluntarily relinquishes the CPU.

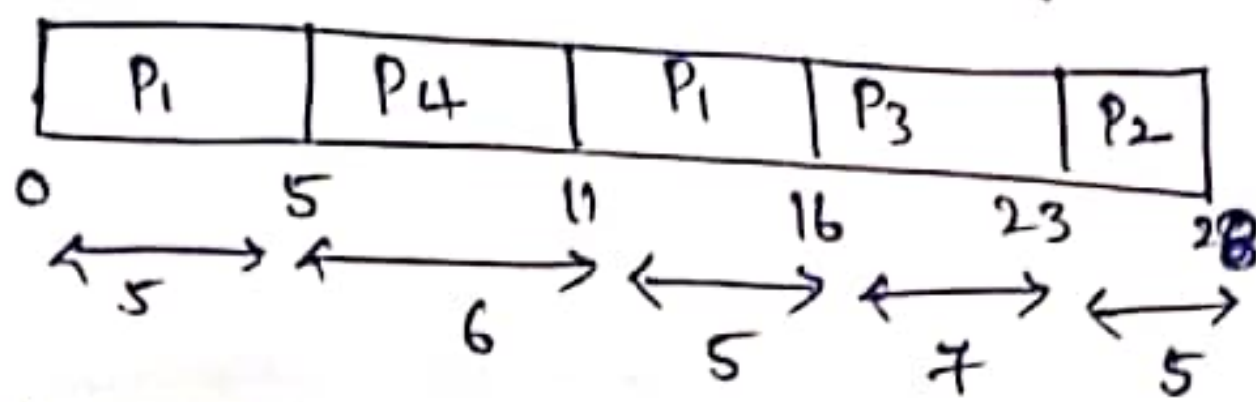
* The priority of task/process in preemptive scheduling is indicated in the same way as that of the mechanism adopted for non-preemptive multitasking.

Ex problem

Three processes with process IDs P_1, P_2, P_3 with estimated completion time 10, 5, 7 ms and priorities (1 - highest, 3 - lowest priority) respectively enter the ready queue together. A new process P_4 with estimated completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P_1 . Assume all the processes contains only CPU operation and no I/O operations are involved.

Sol: At the beginning, there are only 3 processes (P_1, P_2, P_3) available in the 'Ready' queue and the scheduler picks up the process after 5 ms start of execution

with highest priority for scheduling (P_1 with priority 0)
 * Now process P_4 with estimated execution completion time 6ms and the priority 0 enters the 'Ready' queue after 5ms of start of execution of P_1 . Now, the following order



waiting time for $P_1 = 0 + (11 - 5) = 6\text{ms}$

(P_1 starts executing 1st & gets preempted by P_4 after 5ms and again gets the CPU time after completion P_4)

waiting time for $P_4 = 0\text{ms}$

" $P_3 = 16\text{ms}$

" $P_2 = 23\text{ms}$

$$\text{Avg waiting time} = \frac{6 + 0 + 16 + 23}{4} = \frac{45}{4} = 11.25\text{ms}$$

Turn Around Time for $P_1 = 16\text{ms}$

$P_2 = 28\text{ms}$

$P_3 = 23\text{ms}$

$P_4 = 6\text{ms}$

~~Avg TAT~~

$$\begin{aligned} &\left[\begin{array}{l} \text{Time spent in Ready Queue} + \\ \text{Execution Time} = (\text{Execution} \\ \text{start time} - \text{Arrival Time} + \\ \text{Estimated execution time} \\ = 5 - 5 + 6 = 6 \end{array} \right] \end{aligned}$$

$$\text{Avg TAT} = \frac{16 + 28 + 23 + 6}{4}$$

$$= \frac{73}{4} = 18.25\text{ms}$$

TASK COMMUNICATION

* In a multitasking system, multiple tasks/processes run concurrently and each process may or may not interact between. Based on the degree of interaction, the processes running on an OS are classified as

Co-operating processes: In the co-operating interaction model one process requires the help from other processes to complete its execution.

Competing Processes: Don't share anything among themselves but they share the system resources.

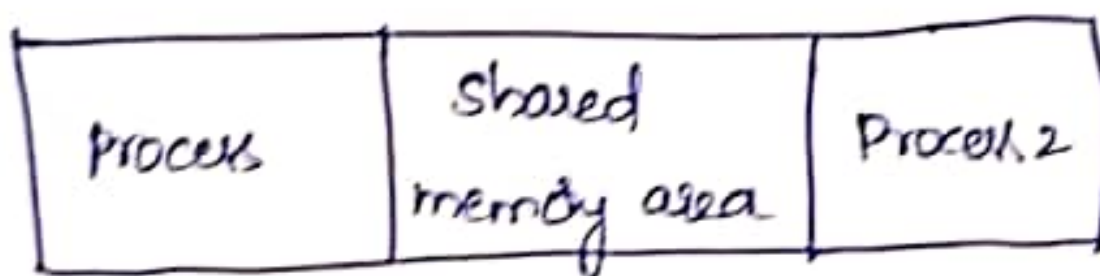
Ex: file, display device etc

* Co-operating processes exchange information and communicate through the following methods.

Co-operation through sharing: Exchange data through some shared resources.

Co-operation through Communication No data is shared between the processes. But they communicate for synchronization.

Shared memory



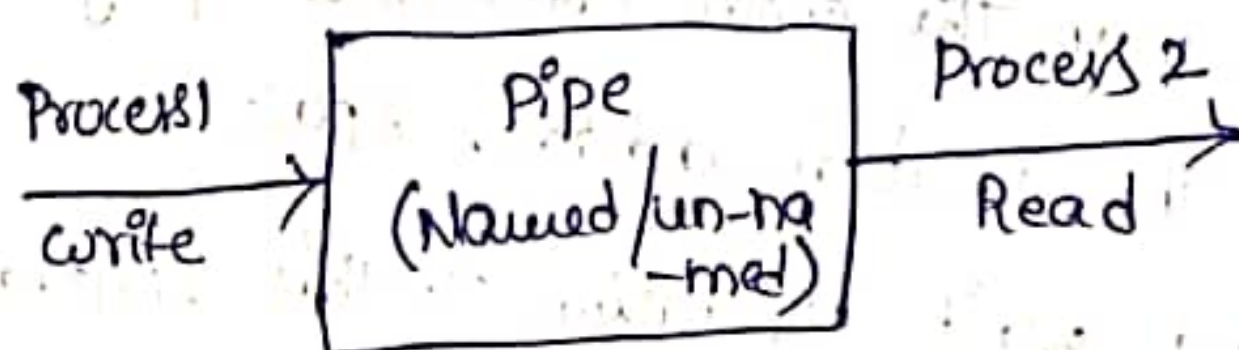
* Processes share some area of the memory to communicate among them as shown in the fig.

* Information to be communicated by the process is written

- to the shared memory area.
- * Other processes which require this information can read the same from the shared memory area.
- * The implementation of shared memory concept is kernel dependent. Different mechanisms are adopted by different kernels for implementing them.
- Ex: "Notice Board" is used by Corporate to publish the information among the employees.
- * A few among them are:

a) Pipes: 'pipe' is a section of the shared memory used by processes for communicating.

- * Pipes follow the client-server architecture. A process which creates a pipe is known as a pipe server and a process which connects to a pipe is known as pipe client.
- * A pipe can be considered as a conduct for information flow and two conceptual ends.
- * It can be unidirectional, allowing information flow in one direction (a) bidirectional, allowing bi-directional information flow.



- * A unidirectional pipe allows the process connecting at one end of the pipe to write to the pipe and the process connected at the other end of the pipe to read the data, whereas a bi-directional pipe allows both reading and writing at one end.
- * The implementation of 'pipes' is also OS dependent.

* There are two types of 'pipe' for Inter Process Communication.
They are:

Anonymous pipes: These pipes are unnamed, unidirectional pipes used for data transfer between two processes.

Named pipes: Unidirectional or bi-directional pipe for data exchange between processes.

* The process which creates the named pipe is known as pipe server. A process which connects the named pipe is known as pipe client.

* With named pipes, any process can act as both client and server allowing point-to-point communication.

* Pipe is a special implementation of message queues.

b) Memory mapped objects:

* Memory mapped objects are a shared memory technique adopted by certain Real-Time Operating systems for allocating a shared block of memory which can be accessed by multiple processes simultaneously.

* In this approach, a mapping object is created and physical storage for it is reserved and committed.

* A process can map the entire committed physical area or a block of it to its virtual address space.

* All read and write operation to this virtual address space by a process is directed to its committed physical area.

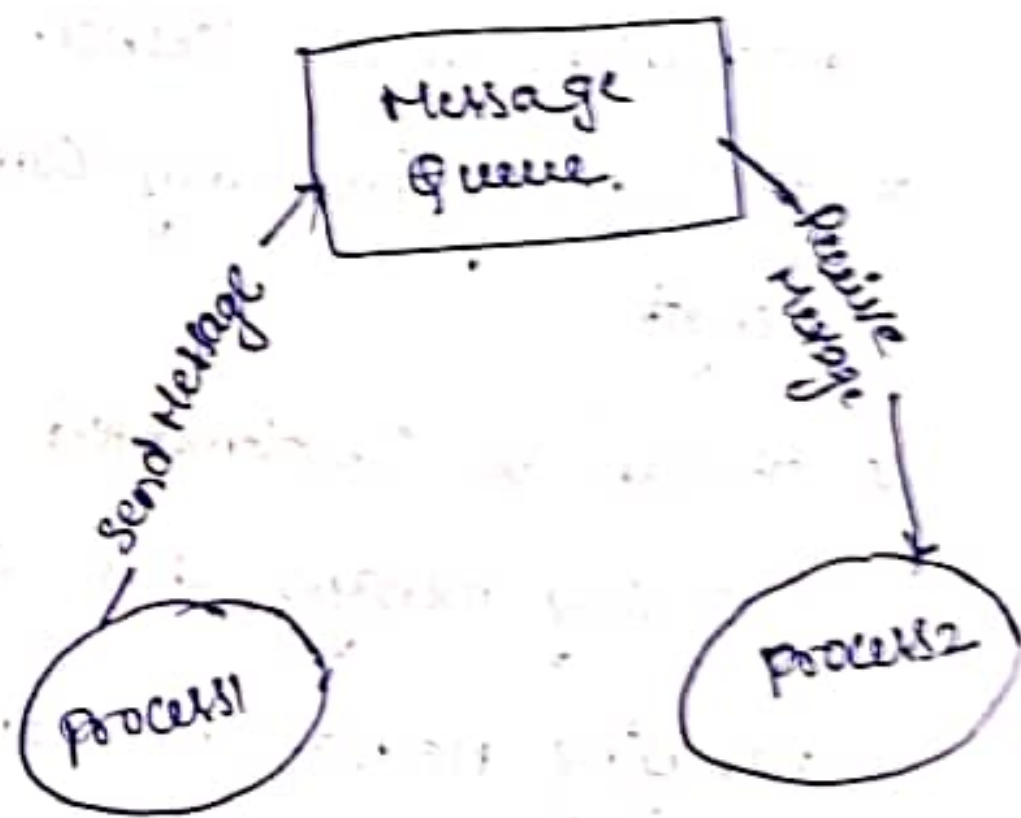
* Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.

Message passing:

- * Message passing is an asynchronous information exchange mechanism used for Inter process/thread Communication.
- * The major difference between shared memory and message passing technique is that, through shared memory lots of data can be shared whereas only limited amount of info/data is passed through message passing.
- * Message passing is relatively fast & free from the synchronisation overheads, Compared to shared memory.
- * Based on the message passing operation between the processes, Message passing is classified into
 - 1) Message Queue
 - 2) Mail Box
 - 3) Signalling.

a) Message Queue:

- * The process which wants to talk to another process posts the message to a first-in-first-out queue called message queue; which stores the messages



- temporarily in a system defined memory object, to pass it to desired process.
- * The messages are sent and received through send and receive methods.
- * The messages are exchanged through a message queue.
- * The implementation of the message queue, send and receive methods are OS kernel dependent.

* The kernel picks up the message from the system message queue one at a time, and examines the message finding the destination thread and then posts the message to the message queue of the corresponding thread.

* The messaging mechanism is classified into synchronous and asynchronous based on the behaviour of the message posting thread.

* In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance from the thread to which the message is posted.

* In Synchronous messaging, the thread which posts a message enters waiting state and waits for the message result from the thread to which the message is posted.

* 'Message queue' is the primary inter-task communication mechanism under kernel.

* It supports two-way communication of messages of variable lengths.

* It can be implemented using one message queue for incoming messages and another one for outgoing messages.

* Messaging Mechanism can be used for task-to-task and task to interrupt service routine communication.

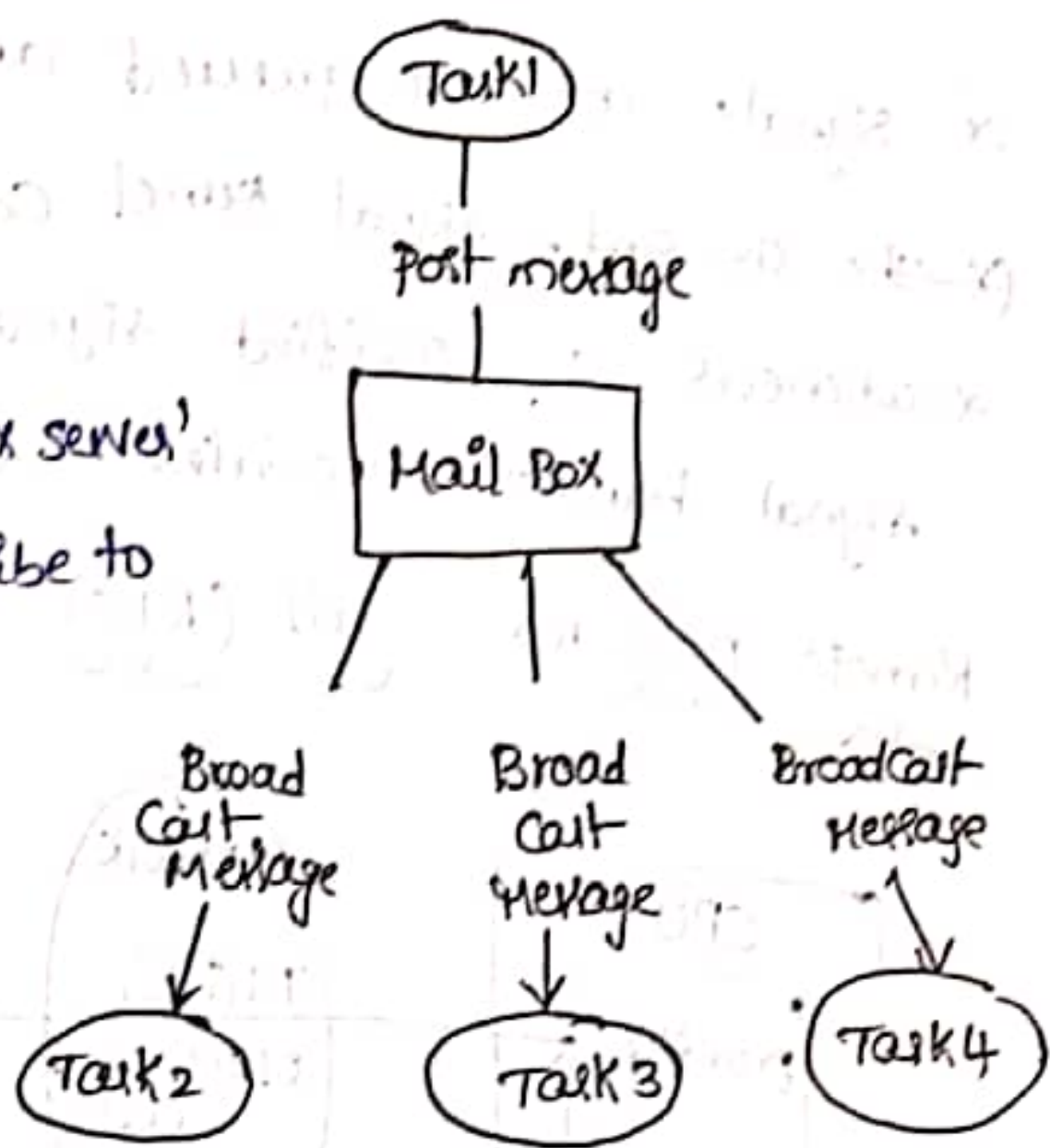
b) Mailbox

* Mailbox is an alternate form of 'message queues' and it is used in certain Real-Time Operating Systems for IPC.

* It is usually used for one-way messaging. The task/threads which want to send a message to other task/

threads creates a mail box for posting the messages.

* The thread which creates the mail box is known as 'mail box server' and the threads which subscribe to the mail box are known as 'Mailbox clients'.



* The mail box server post messages to the mail box and notifies it to clients which are subscribed to mail box. The client read the message from the mailbox on receiving the notification.

* The mail box creation, subscription, message reading, and writing are achieved through OS kernel provided API calls.

* Mail Box and Message queues are the same functionality. The only difference is in the messages supported by them.

* Mail box is used for exchanging a single message b/w tasks or b/w ISR & a task.

* It associates a pointer pointing a mail box and a wait list to hold the tasks waiting for message to appear in the mail box.

* The implementation of mail box is OS kernel dependent.

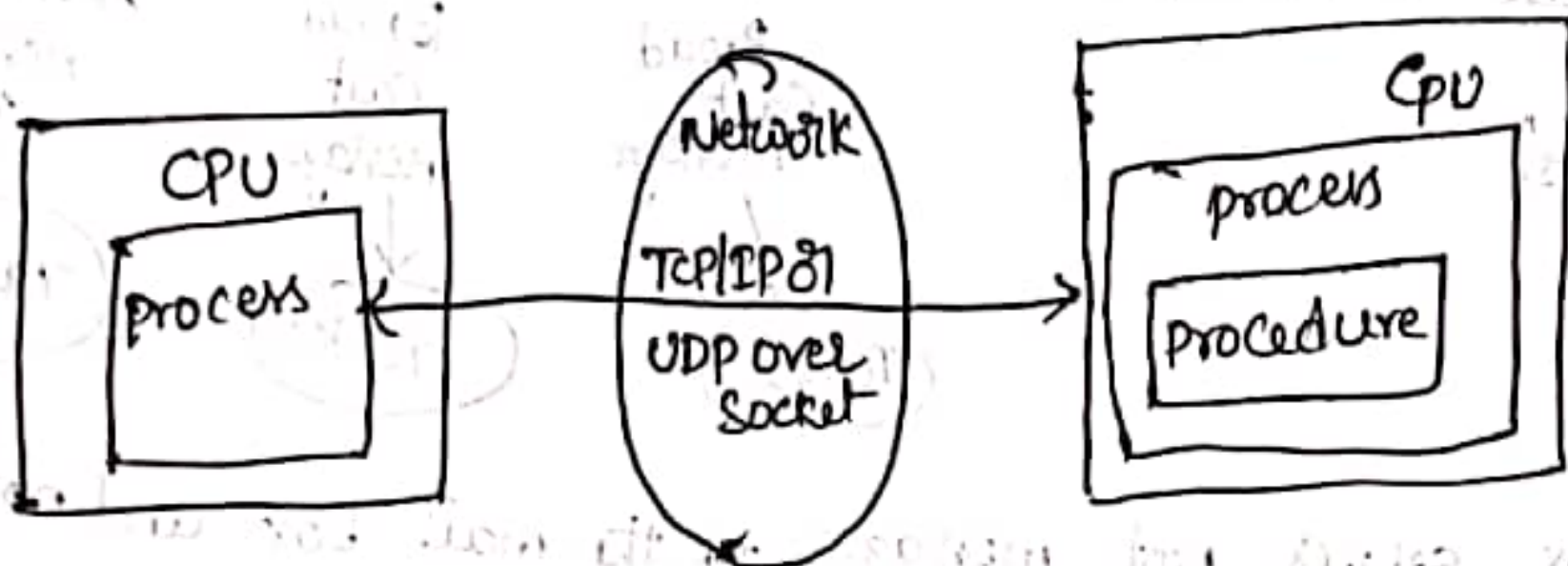
Signalling:

* Signalling is a primitive way of communication between Processes/ threads.

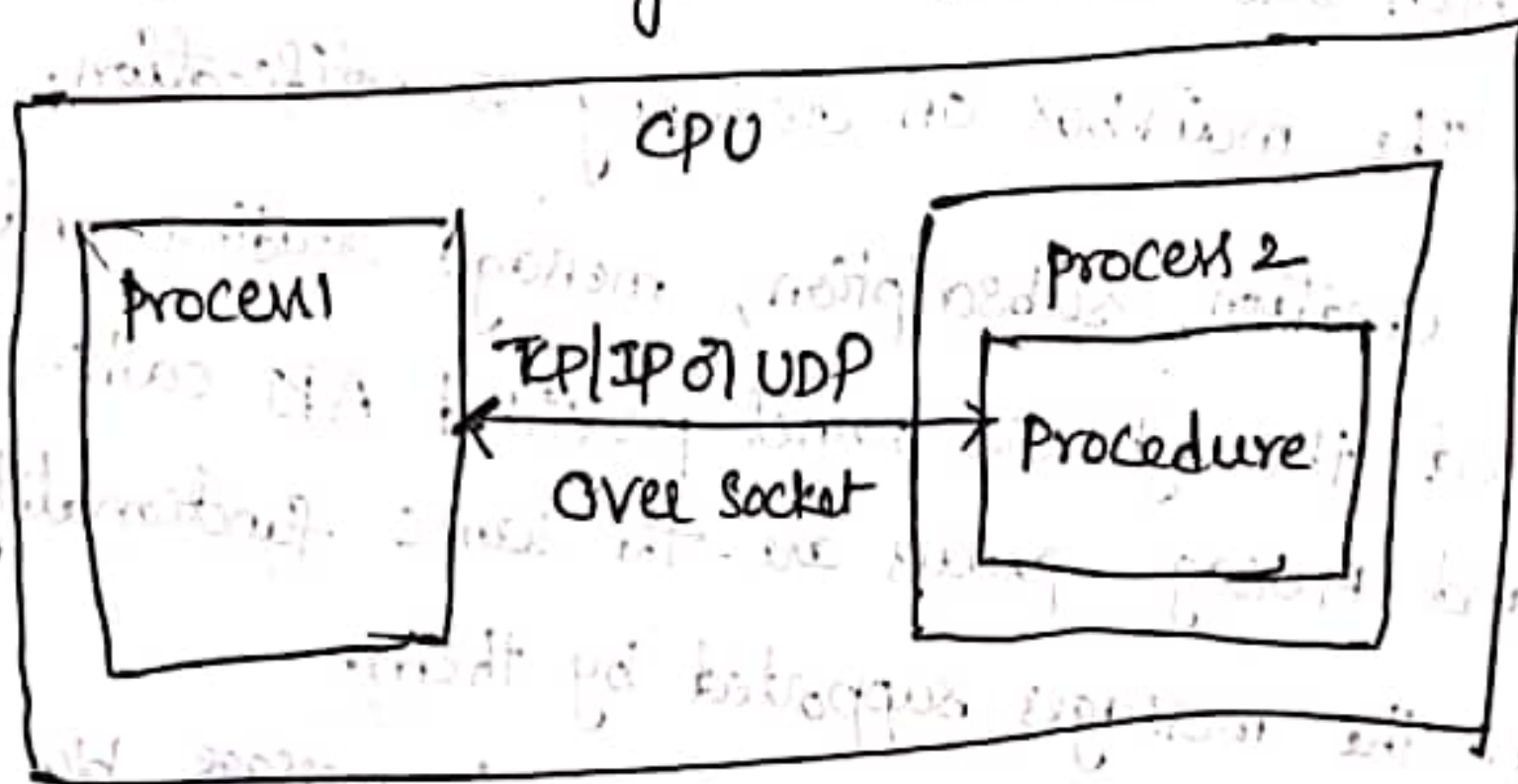
* Signals are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a scenario which the other process/thread is waiting.

- * Signals are not queued and they don't carry any data.
- ~~(*) The OS send signal kernel call under)~~
- * whenever a specified signal occurs it is handled by a signal handler associated with the signal.

Remote procedure call (RPC) and sockets;



fig(a): Process running on different CPUs which are not



fig(b): Process running on same CPU.

- * Remote Procedure Call or RPC is the inter process communication (IPC) mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in network.
- * RPC is mainly used for distributed applications like client server applications. It is possible to communicate over a heterogeneous n/w.
- * The CPU/process containing the procedure which needs to be invoked remotely known as server.
- * The CPU/process which initiates an RPC request is known as client.

* The RPC Communication can be either Synchronous (Blocking) or Asynchronous (Non-blocking)

* In Synchronous Communication, the procedure which calls the remote procedure is blocked until it receives a response back from the other process.

* In Asynchronous RPC calls, the calling process continues its execution while the remote process performs the execution of the procedure.

* RPC employs authentication mechanisms to protect the systems against vulnerabilities.

* Authentication mechanisms like IDs, public key cryptography etc. are used. ~~It is~~ by the client

Socket → Sockets are used for RPC Communication. It is a logical end point in a two-way communication link between two applications running a network.

* A port number is associated with a socket so that the network layer of the communication channel can deliver the data to the designed application.

* Types of Sockets are : Internet Sockets (INET), UNIX Sockets.

* INET Sockets work on Internet Communication protocol.

* TCP/IP, UDP are the communication protocols used by INET Sockets.

* They are classified into two types. They are

a) Stream Sockets

b) Datagram Sockets

* Stream Sockets are connection oriented and they use TCP to establish a reliable connection.

* Datagram Sockets rely on UDP for establishing a connection. It is unreliable when compared to TCP.

- * The client-server Communication model uses a socket at the client side and a socket at the server side.
- * The implementation of socket is OS kernel dependent. Different types of OSs provide different socket interfaces.

Task Synchronization

- * The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as 'Task/process Synchronization'.

- * Various synchronisation issues may arise in a multitasking environment if processes are not synchronised properly.

Task Synchronisation/Communication Issues

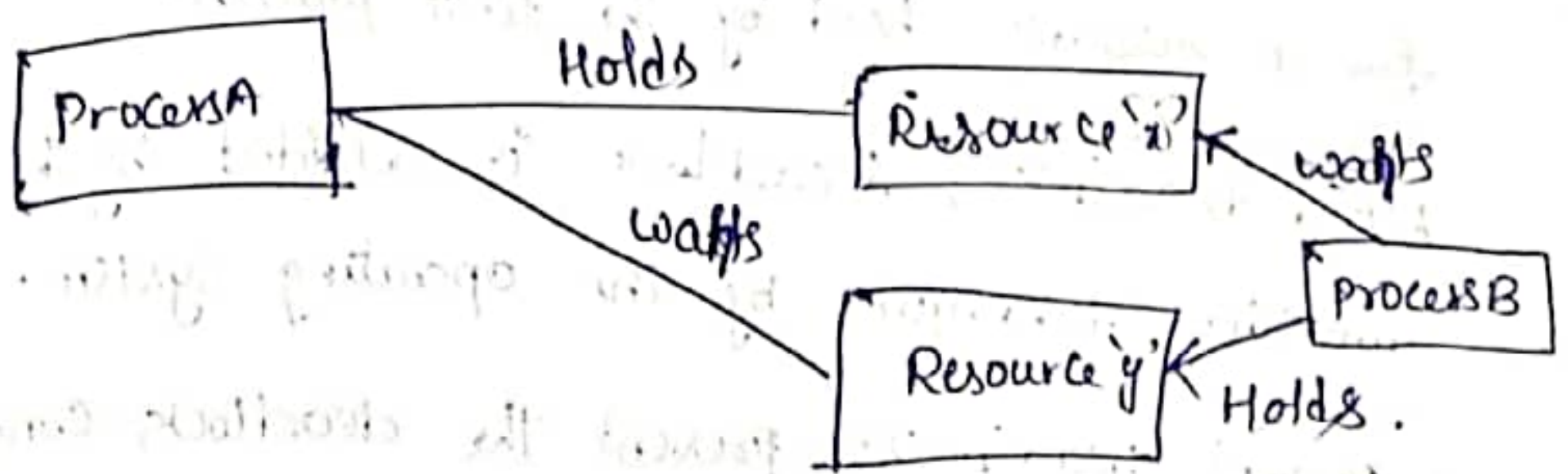
a) Race Condition (Racing)

- * Race (or) Racing Condition is the situation in which multiple processes compete (race) each other to access and manipulate the shared data concurrently.
- * In a Race Condition, the final value of the shared data depends on the process which acted on the data finally.
- * A race Condition produces incorrect results.

b) Dead lock:

- * A Deadlock Condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlock processes.

* In its simplest form 'deadlock' is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by first process.



* From above fig: Process A holds a resource 'x' and it wants a resource 'y'; process B holds a resource 'y' and it wants a resource 'x'.

[* Resource 'y' is held by process B. Resource 'x' held by process A.]

* They compete each other to get the resources and held by the respective processes. The result of competition is dead lock.

* The different conditions favoring a deadlock situation

a) Mutual Exclusion: The criteria that only process can hold a resource at a time. They should access ~~shared~~ shared resources with mutual exclusion.

Ex: The accessing of display hardware is an embedded device.

b) Hold and wait: A process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other process.

c) No Resource preemption: Operating system can't take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process.

holding it.

d) Circular wait: A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process.

e) Avoid Deadlocks: Dead lock is avoided by the resource allocation techniques by the operating system.

f) Prevent Deadlocks: prevent the deadlock condition by negating one of the four conditions favouring the Dead lock situation.

i) Ensure that a process doesn't hold any other resources when it requests a resource. This can be achieved by

implementing the set of rules/guidelines in allocating resources to processes.

a) A process must request all its required resources at once. The resources should be allocated before its process begins its execution.

b) Grant resource allocation requests from processes

only if the process doesn't hold a resource currently.

ii) Ensure the resource preemption is possible at operating system level. This can be achieved by implementing the

following set of rules/guidelines in resources allocation and releasing.

a) Release all the ~~process~~ resources currently held by a process if a request made by the process for a new resource is not able to fulfil immediately.

b) Add the resources which are preempted to a resource list describing the resource which the process requires to complete its execution.

c) Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

The Dining Philosopher's Problem:

Scenario 1: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed since two forks are required for eating the spaghetti present in plate. Philosopher 1 thinks that philosopher 2 sitting to the right of him/her will put the fork down and waits for it. Philosopher 2 thinks that philosopher 3 sitting to the right of him/her will put the fork down and waits for it and so on. This forms a circular chain of un-granted requests. If the philosopher continue in this state waiting for the forks from the philosopher sitting to right of each, they will not make any progress in eating and this will result in starvation of the philosophers and deadlock.

Scenario 2: All the philosophers start brainstorming together. One of the philosophers is hungry and he/she picks up the left fork. When the philosopher is about ~~to~~ to pick up the right fork, the philosopher sitting to his right also become hungry and tries to grab the left fork which the right fork of his neighbouring philosopher who is trying to lift it, resulting in a Race around condition.

Scenario 3: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the forks and unable to proceed, since two forks are required for eating. Each of them anticipates that the adjacently

sitting philosopher will put his/her fork down and let it go for a fixed duration & after this puts the fork down. Each of them again tries to lift the fork after duration of time. All philosophers are trying to lift the fork at the same time, none of them will be able to grab two forks. This leads to livelock and starvation Condition.

Producer - Consumer / Bounded buffer problem:

* It is a common data sharing problem where two processes concurrently access a shared buffer with fixed size. A process which produces data is called "producer process" and process which consumes data is called consumer process. The situation under which the producer process more data rate and consumer process consumes at slower data rate, then it will lead to buffer overrun. Similarly if the consumer consumes the data at a faster rate and the data is produced at lesser rate then this will lead to buffer underrun.

* The different situation which may arise based on scheduling of the producer threads and consumer threads are listed as below.

- a) Producer threads are scheduled more frequently than consumer threads
- b) Consumer threads are scheduled more frequently than producer threads.

Readers - writers problem: This is a common issue observed in processes competing for limited shared resources. They are characterised by multiple shared processes trying to read and write shared data concurrently.

Priority Inversion: It is the byproduct of the combination of blocking based process synchronisation and pre-emptive priority scheduling. Priority inversion is the condition in which a high priority task and low priority task and medium priority task which doesn't require the shared resource continue its execution by preempting the low priority task.

Priority Inheritance: A low priority task that is concurrently accessing a shared resource requested by high priority task temporarily 'inherit' the priority of the high priority task, from the moment that the high priority task arises the request.

Priority Ceiling: In priority ceiling, a priority is associated with each shared resource. The priority associated to each resource is the priority of highest priority task which uses this shared resources. This priority is called ceiling priority. Whenever a task access a shared memory resources, the scheduler elevates the priority of the task to that of the priority of the resources.

Task Synchronization Techniques:

* Process / Task synchronization is essential for

1. Avoiding conflicts in resource access in a multitasking environment.

2. Ensuring the proper sequence of operation across processes. The producer-consumer problem is a typical example for processes requiring proper sequence of operation. In Producer-consumer problem, accessing

the shared buffer by different processes is not the issue, the issue is writing. A process should write to the shared buffer only if the buffer is not full and the consumer thread should not read from the buffer if it is empty.

3. Communicating between processes.

The code memory area which holds the program instructions for accessing a shared resource is known as 'critical section'. In order to synchronize the access to shared resources, the access to the critical section should be exclusive. The exclusive access to critical section of code is provided through mutual exclusion mechanism.

Mutual Exclusion through Busy waiting / Spin Locks

* "Busy waiting" is the simplest method for enforcing mutual exclusion.

- * The 'Busy waiting' technique uses a lock variable for mutual exclusion.
- * Each process/thread checks the lock variable before entering the critical section.
- * The lock is set to '1' by a process/thread, if the process/thread is already in its critical section, otherwise the lock is set to '0'.
- * The major challenge in implementing the lock variable based on synchronisation is the non-availability of a single atomic instruction which combines the reading, comparing and setting of the lock variable.
- * Most often the three different operations related to locks, viz; the operation of reading the lock variable, checking its present value and setting it are achieved with multiple low level instructions.
- * The low level implementation of the 'Busy waiting' code snippet which is discussed earlier, under windows.

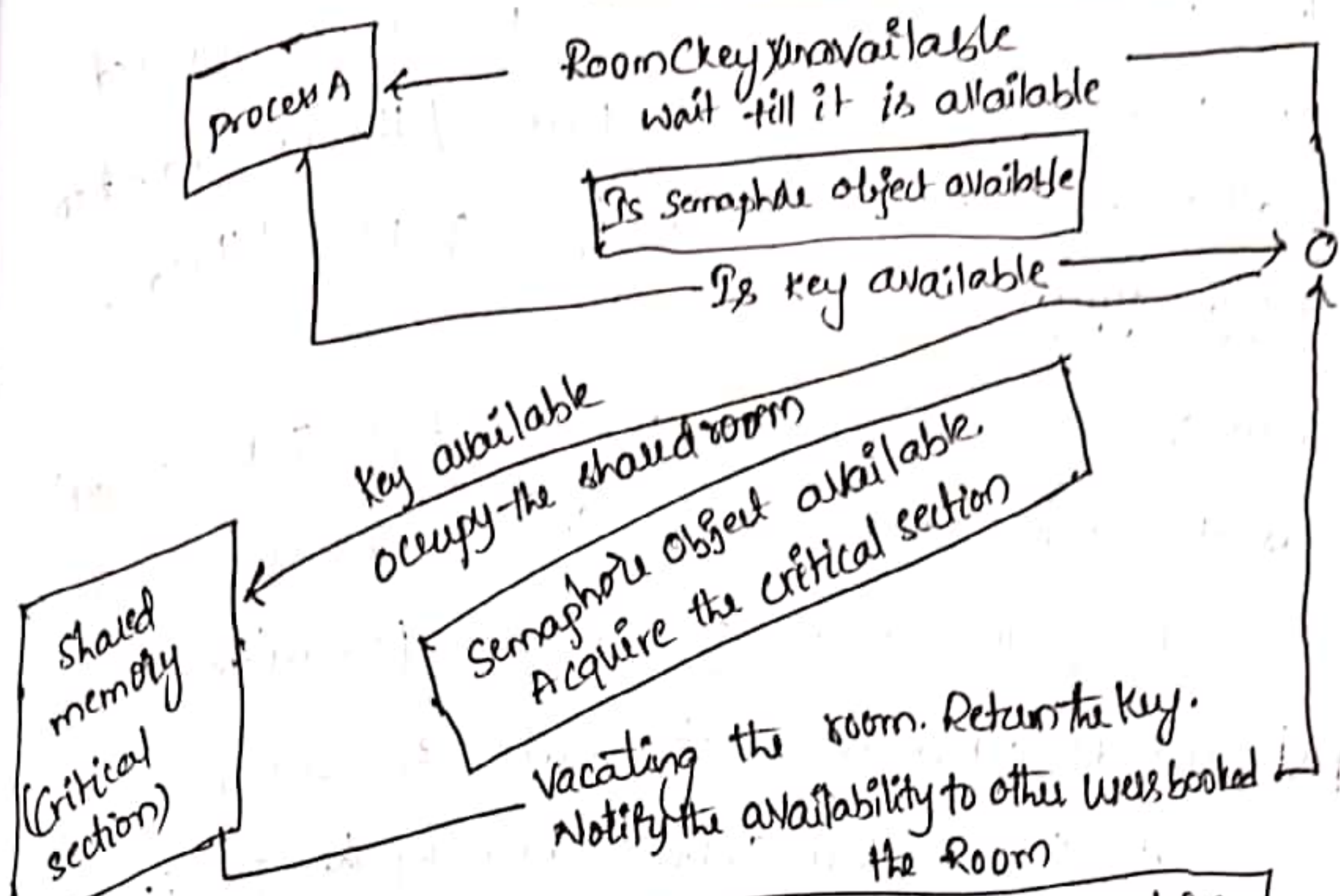
Mutual Exclusion through Sleep & Wakeup

- * The 'Busy waiting' mutual exclusion enforcement mechanism used by processes makes the CPU always busy by checking the lock to see whether they can proceed. This results in the wastage of CPU time and leads to high power consumption.
- * An alternative to 'busy waiting' is the 'sleep & wakeup' mechanism. When a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes 'sleep' and enters the 'blocked' state.
- * The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section.
- * The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section.

Semaphore:

- * Semaphore is a sleep and wakeup based mutual exclusion for shared resource access.
- * Semaphore is a system resource and the process which wants to access the shared resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it.

semaphore object unavailable. Sleep till semaphore object is available

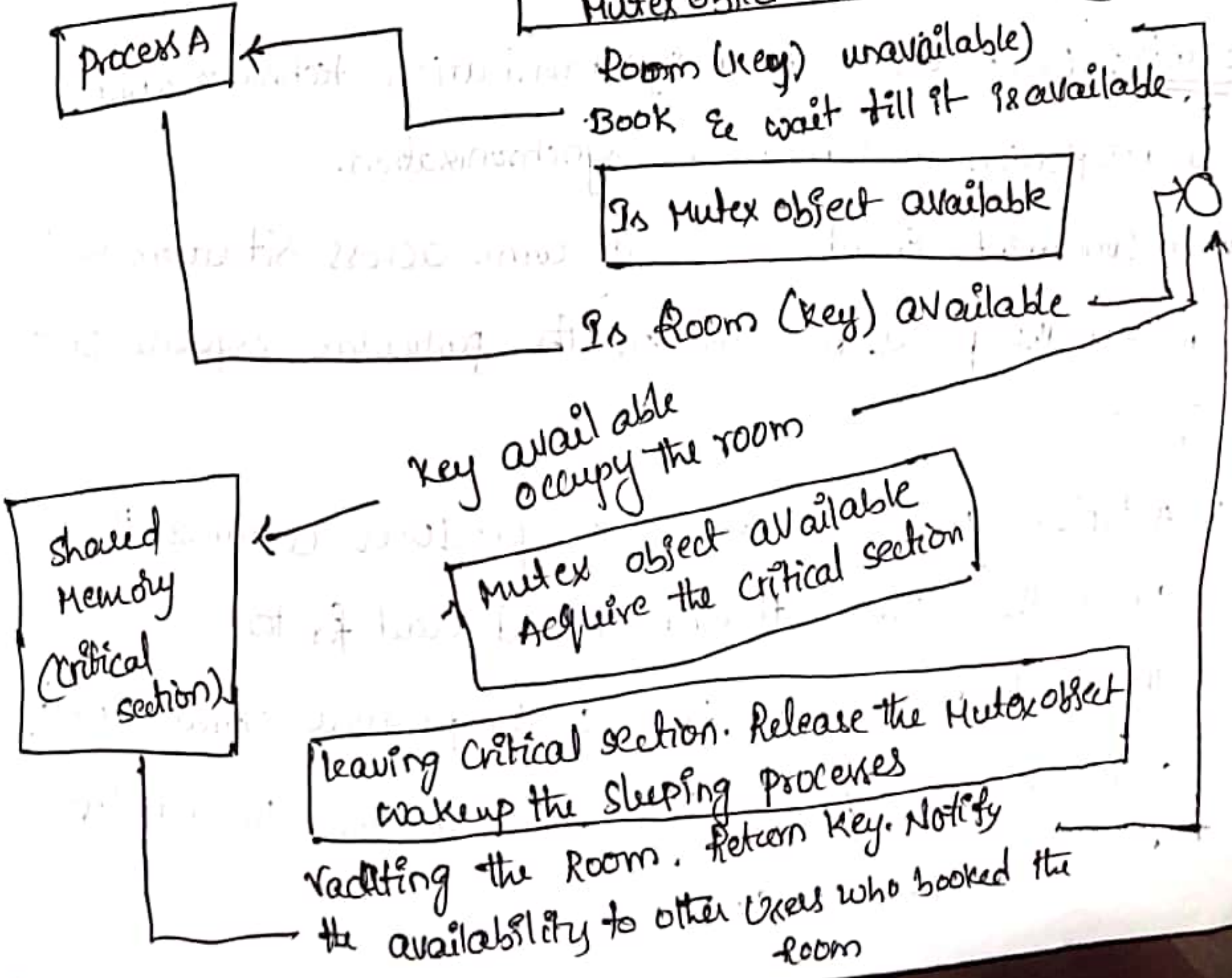


Leaving critical section. Release the semaphore object. Wakeup the sleeping process.

Binary semaphore: Binary semaphore (Mutex) is a synchronization object provided by OS for process/thread synchronization.

* Any process/thread can create a 'mutex object' and other processes/threads of the system can use this "mutex object" for synchronizing the access to the critical section

Mutex object unavailable. Sleep till Mutex object is available



- * Only one process/thread can own the "mutex object" at a time. The state of a mutex object is set to signalled when it is not owned by any process/thread and non-signalled when it is owned by any process/thread.

Critical Section objects:

- * In Windows Embedded Compact, the "Critical Section object" is same as the "mutex object" except that "Critical Section object" can only be used by the threads of a single process (single process).
- * The piece of code which needs to be made as "Critical Section" is placed at the "Critical Section" area by the process.
- * The memory area which is to be used as the "Critical Section" is allocated by process.
- * The process creates a "Critical Section" area by creating a variable of type `CRITICAL_SECTION`.
- * The Initialize Critical Section API initializes the critical section pointed by the pointer `lpCriticalSection` to the Critical Section.

Events: Event object is a synchronisation technique which uses the notification mechanism for synchronisation.

- * In Concurrent Execution we may come across situations which demand the process to wait for the particular sequence of operations.
- * A typical example of this is the producer consumer threads, where the consumer thread should wait for the producer thread to produce the data & the producer thread should wait for the consumer thread to consume the data before producing fresh data.

Device Drivers

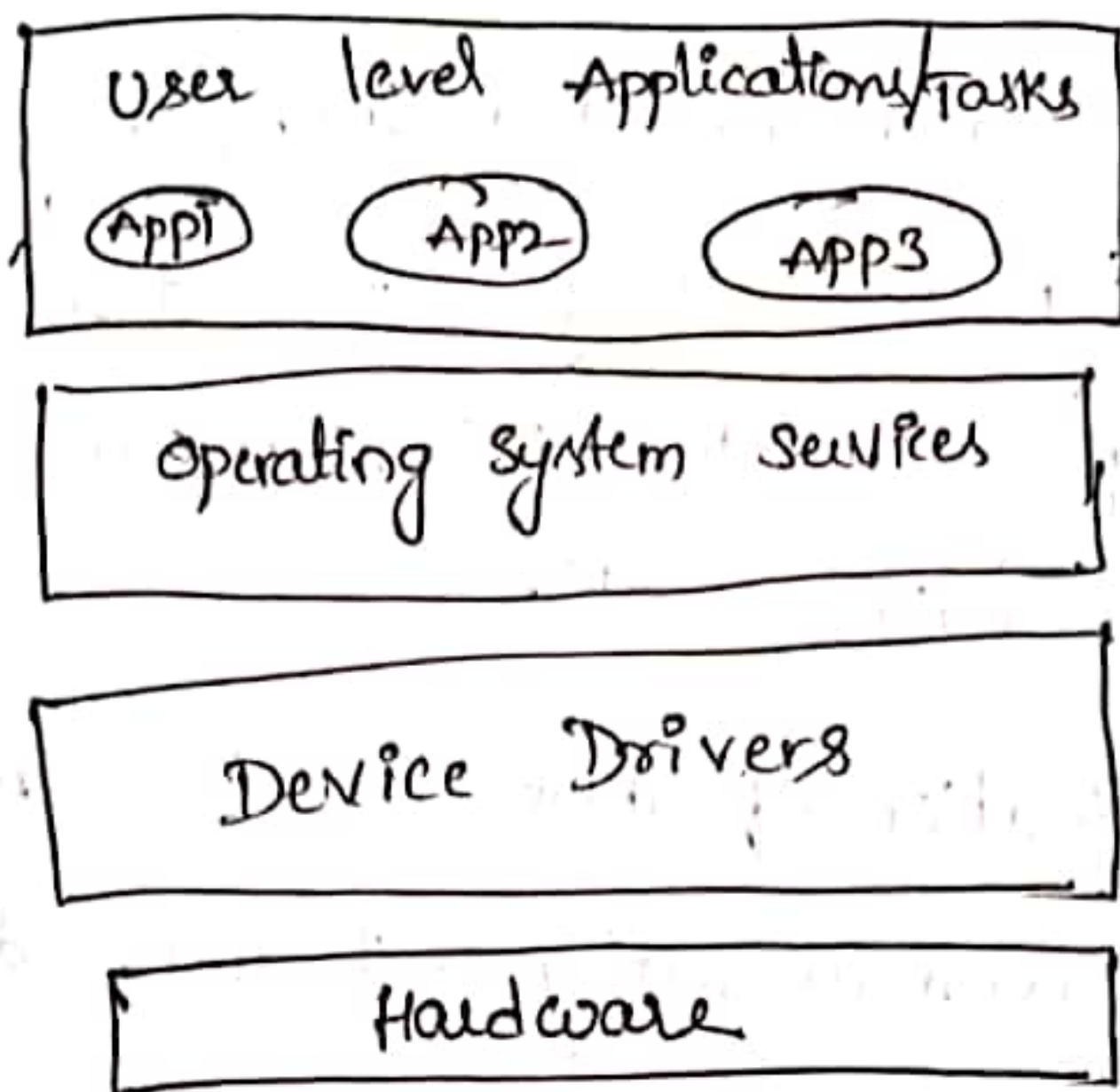


Fig: Role of Device Driver in Embedded OS based product.

- * Device driver is a piece of software that acts as a bridge between the operating system and the hardware.
- * The user applications talk to the Operating System^{Kernel}, for all necessary information exchange including communication with the hardware peripherals.
- * The architecture of OS kernel will not allow direct device access from the user application.
- * All the device related access should flow through the OS kernel and the kernel routes it to the concerned hardware peripheral.
- * OS provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware.
- * The device driver abstracts the hardware from user Application.
- * Device drivers are responsible for initiating and managing the communication with the hardware peripherals.

- * They are also responsible for establishing the connection, initialising the hardware and transferring data.
- * An embedded product may contain different types of hardware components like Wi-Fi module, file systems, storage device interface etc.
- * The initialization of these devices and the protocols required for communicating with these devices may be different.
- * All these requirements are implemented in drivers and a single driver will not be able to satisfy all these. Hence, each driver requires a unique driver component.
- * Certain drivers come as a part of the OS kernel & certain drivers need to be installed on the fly.
- * Device drivers can run on either userspace or kernel space.
- * Device drivers which run in userspace is known as "user mode drivers" and the driver which run in kernel space are known as "kernel mode drivers".
- * User mode drivers are safer than kernel mode drivers.
- * If an error or exception occurs in user mode driver it won't affect the services of the kernel.
- * If an exception occurs in the kernel mode, it may lead to kernel crash.

* A Device Driver implements the following

- 1) Device Initialization and Interrupt Configuration
- 2) Interrupt Handling & processing
- 3) Client Interfacing.

* The Basic Interrupt involves the following.

1. Set the interrupt type of Level Triggered, enable the interrupt & set the interrupt ~~clock~~ priorities
2. Bind the Interrupt with an Interrupt request (IRQ)
3. Register an Interrupt service Routine (ISR) with an Interrupt Request (IRQ).

How to choose an RTOS:

* The decision of choosing RTOS for an embedded design is very crucial. A lot of factors needs to be analysed carefully before making a decision on the selection of an RTOS.

* These factors can be either functional or non-functional.

Functional Requirements:

a) Processor Support : It is not necessary that all RTOS support all kinds of processor architecture. It is essential to ensure the processor supported by RTOS.

b) Memory requirements : The OS requires ROM memory for the OS files and it is normally in non-volatile memory like FLASH. OS also requires working memory RAM for loading the OS services. Since embedded systems are memory constrained, it is essential to evaluate the minimal ROM & RAM requirements for the OS under consideration.

- c) Real time Capabilities: It is not mandatory that the all embedded systems need to be Real-time and all embedded Operating Systems are 'Real time behaviour'. The task/scheduling policies plays an important role in the 'Real time behaviour' of an OS.
- d) Kernel and Interrupt Latency: The kernel of the OS may be disable interrupts while executing certain services and may lead to interrupt latency. For an embedded system whose response requirements are high, this latency should be minimal.
- e) Inter process Communication and Task synchronization: The implementation of Inter process Communication and synchronization is OS kernel dependent. Kernels implement policies for avoiding priority inversion issues in resource sharing.
- f) Modularisation Support: Most of the Operating systems provide a bunch of features. ~~at and~~ It is very useful if the OS supports modularisation where in which the developer can choose the essential modules.
- g) Support for Networking & Communication: The OS kernel may provide stack implementation and Driver support for bunch of communication interfaces and n/wing. Ensure the OS under consideration provides support for all the interfaces required by the embedded product.
- h) Development Language Support: Certain OS may include run time libraries required for the running applications in languages like Java & C#.

* The OS may include these components as built in components. If not, check the availability of the same from the 3rd party.

Non functional Requirements

- a) Custom Developed or Off the Shelf: It is possible to go for the complete development of an operating system suiting the embedded system needs or use an off-the-shelf, readily available operating system. It may be possible to build the required features by customising the open source OS. The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.
- b) Cost: The total cost for developing or buying the OS and maintaining in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.
- c) Development and Debugging Tools Availability: The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design. Certain operating system may be superior performance, but the availability of tools for supporting the development may be limited.
- d) Ease of use: It is easy to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.
- e) After Sales: For a commercial embedded RTOS, after sales in the form of e-mail, on-call services etc; for bug fixes, critical patch updates and support for production issue etc should be analysed.