Professional Elective-II: Embedded Systems (PECCSE601B)

Module - IV

RTOS based embedded system design

Prepared by – Dr. Susmita Biswas
Associate Professor
University of Engineering and Management, Kolkata

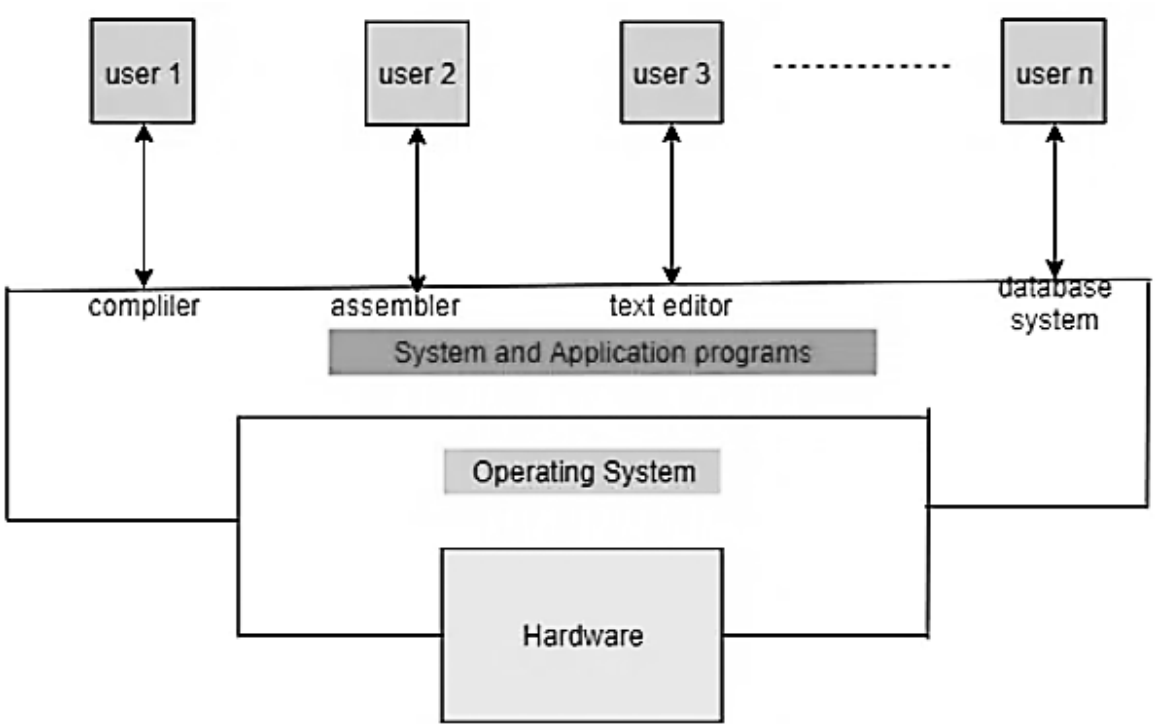
Sub topic

- Operating system basics
- types of operating systems
- need of real time operating system
- features of a real-time operating system
- commercial real-time operating systems
- tasks, process, threads, multiprocessing and multi-tasking
- architecture of kernel
- real-time task scheduling
- threads-processes- scheduling putting them together
- task communication
- task synchronization
- device drivers
- memory management
- interrupt service mechanism
- context switching

Operating System

Operating System is a collection of set of programs that manages all the resources of the computer. An operating system acts as an interface between the software and different parts of the computer or the computer hardware. The operating system is designed in such a way that it can manage the overall resources and operations of the computer.

It is an intermediator between hardware and user. The operating system helps in improving the computer software as well as hardware. Without OS, it became very difficult for any application to be user-friendly.



Functions of the Operating System

- **Resource Management:** The operating system manages and allocates memory, CPU time, and other hardware resources among the various programs and processes running on the computer.
- **Process Management:** The operating system is responsible for starting, stopping, and managing processes and programs. It also controls the scheduling of processes and allocates resources to them.
- **Memory Management:** The operating system manages the computer's primary memory and provides mechanisms for optimizing memory usage.
- **Security:** The operating system provides a secure environment for the user, applications, and data by implementing security policies and mechanisms such as access controls and encryption.
- **Job Accounting:** It keeps track of time and resources used by various jobs or users.
- **File Management:** The operating system is responsible for organizing and managing the file system, including the creation, deletion, and manipulation of files and directories.

Functions of the Operating System

- **Device Management:** The operating system manages input/output devices such as printers, keyboards, mice, and displays. It provides the necessary drivers and interfaces to enable communication between the devices and the computer.
- **Networking:** The operating system provides networking capabilities such as establishing and managing network connections, handling network protocols, and sharing resources such as printers and files over a network.
- **User Interface**: The operating system provides a user interface that enables users to interact with the computer system. This can be a Graphical User Interface (GUI), a Command-Line Interface (CLI), or a combination of both.
- **Backup and Recovery:** The operating system provides mechanisms for backing up data and recovering it in case of system failures, errors, or disasters.
- **Virtualization:** The operating system provides virtualization capabilities that allow multiple operating systems or applications to run on a single physical machine. This can enable efficient use of resources and flexibility in managing workloads.

Functions of the Operating System

- **Performance Monitoring:** The operating system provides tools for monitoring and optimizing system performance, including identifying bottlenecks, optimizing resource usage, and analyzing system logs and metrics.
- **Time-Sharing:** The operating system enables multiple users to share a computer system and its resources simultaneously by providing time-sharing mechanisms that allocate resources fairly and efficiently.
- **System Calls:** The operating system provides a set of system calls that enable applications to interact with the operating system and access its resources. System calls provide a standardized interface between applications and the operating system, enabling portability and compatibility across different hardware and software platforms.
- **Error-detecting Aids:** These contain methods that include the production of dumps, traces, error messages, and other debugging and error-detecting methods.

Objectives of Operating Systems

- **Convenient to use:** One of the objectives is to make the computer system more convenient to use in an efficient manner.
- **User Friendly:** To make the computer system more interactive with a more convenient interface for the users.
- **Easy Access:** To provide easy access to users for using resources by acting as an intermediary between the hardware and its users**.**
- **Management of Resources:** For managing the resources of a computer in a better and faster way.
- **Controls and Monitoring:** By keeping track of who is using which resource, granting resource requests, and mediating conflicting requests from different programs and users.
- **Fair Sharing of Resources:** Providing efficient and fair sharing of resources between the users and programs.

Types of Operating Systems

- **Batch Operating System:** A Batch Operating System is a type of operating system that does not interact with the computer directly. There is an operator who takes similar jobs having the same requirements and groups them into batches.
- **Time-sharing Operating System:** Time-sharing Operating System is a type of operating system that allows many users to share computer resources (maximum utilization of the resources).
- **Distributed Operating System:** Distributed Operating System is a type of operating system that manages a group of different computers and makes appear to be a single computer. These operating systems are designed to operate on a network of computers. They allow multiple users to access shared resources and communicate with each other over the network. Examples include Microsoft Windows Server and various distributions of Linux designed for servers.
- **Network Operating System:** Network Operating System is a type of operating system that runs on a server and provides the capability to manage data, users, groups, security, applications, and other networking functions.

Types of Operating Systems

•**Real-time Operating System:** Real-time Operating System is a type of operating system that serves a real-time system and the time interval required to process and respond to inputs is very small. These operating systems are designed to respond to events in real time. They are used in applications that require quick and deterministic responses, such as embedded systems, industrial control systems, and robotics.

•**Multiprocessing Operating System:** Multiprocessor Operating Systems are used in operating systems to boost the performance of multiple CPUs within a single computer system. Multiple CPUs are linked together so that a job can be divided and executed more quickly.

•**Single-User Operating Systems:** Single-User Operating Systems are designed to support a single user at a time. Examples include Microsoft Windows for personal computers and Apple macOS.

•**Multi-User Operating Systems:** Multi-User Operating Systems are designed to support multiple users simultaneously. Examples include Linux and Unix.

Types of Operating Systems

- **Embedded Operating Systems:** Embedded Operating Systems are designed to run on devices with limited resources, such as smartphones, wearable devices, and household appliances. Examples include Google's Android and Apple's iOS.
- **Cluster Operating Systems:** Cluster Operating Systems are designed to run on a group of computers, or a cluster, to work together as a single system. They are used for high-performance computing and for applications that require high availability and reliability. Examples include Rocks Cluster Distribution and Open MPI.

Real Time Operating System (RTOS)

Real-time **operating systems (RTOS)** are used in environments where a large number of events, mostly external to the computer system, must be accepted and processed in a short time or within certain deadlines.

Regular OS	Real-Time OS (RTOS)
Complex	Simple
Best effort	Guaranteed response
Don’t have Strict Timing constraints	Strict Timing constraints
Average Bandwidth	Minimum and maximum limits
Unknown components	Components are known
Unpredictable behavior	Predictable behavior
Plug and play	RTOS is upgradeable

Features of a real-time operating system

- **Correctness:** It is one of the precious parts of a real time OS.
- **Safety:** Safety is necessary for any system but real time operating system can perform for a long time without failures.
- **Time Constraints:** In real time operating system, the tasks should be completed within the given time period.
- **Embedded:** real time operating systems are embedded.

Types of RTOS

1. Hard Real-Time Operating System: These operating systems guarantee that critical tasks are completed within a range of time.

For example, a robot is hired to weld a car body. If the robot welds too early or too late, the car cannot be sold, so it is a hard real-time system that requires complete car welding by the robot hardly on time., scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

Types of RTOS

2. Soft real-time operating system: This operating system provides some relaxation in the time limit.

For example – Multimedia systems, digital audio systems, etc. Explicit, programmer-defined, and controlled processes are encountered in real-time systems. A separate process is changed by handling a single external event. The process is activated upon the occurrence of the related event signaled by an interrupt.

Multitasking operation is accomplished by scheduling processes for execution independently of each other. Each process is assigned a certain level of priority that corresponds to the relative importance of the event that it services. The processor is allocated to the highest-priority processes. This type of schedule, called, priority-based preemptive scheduling is used by real-time systems.

Types of RTOS

3. Firm Real-time Operating System: RTOS of this type have to follow deadlines as well. In spite of its small impact, missing a deadline can have unintended consequences, including a reduction in the quality of the product. Example: Multimedia applications.

4.Deterministic Real-time operating System: Consistency is the main key in this type of real-time operating system. It ensures that all the task and processes execute with predictable timing all the time,which make it more suitable for applications in which timing accuracy is very important. **Examples:** INTEGRITY, PikeOS.

Commercial real-time operating systems

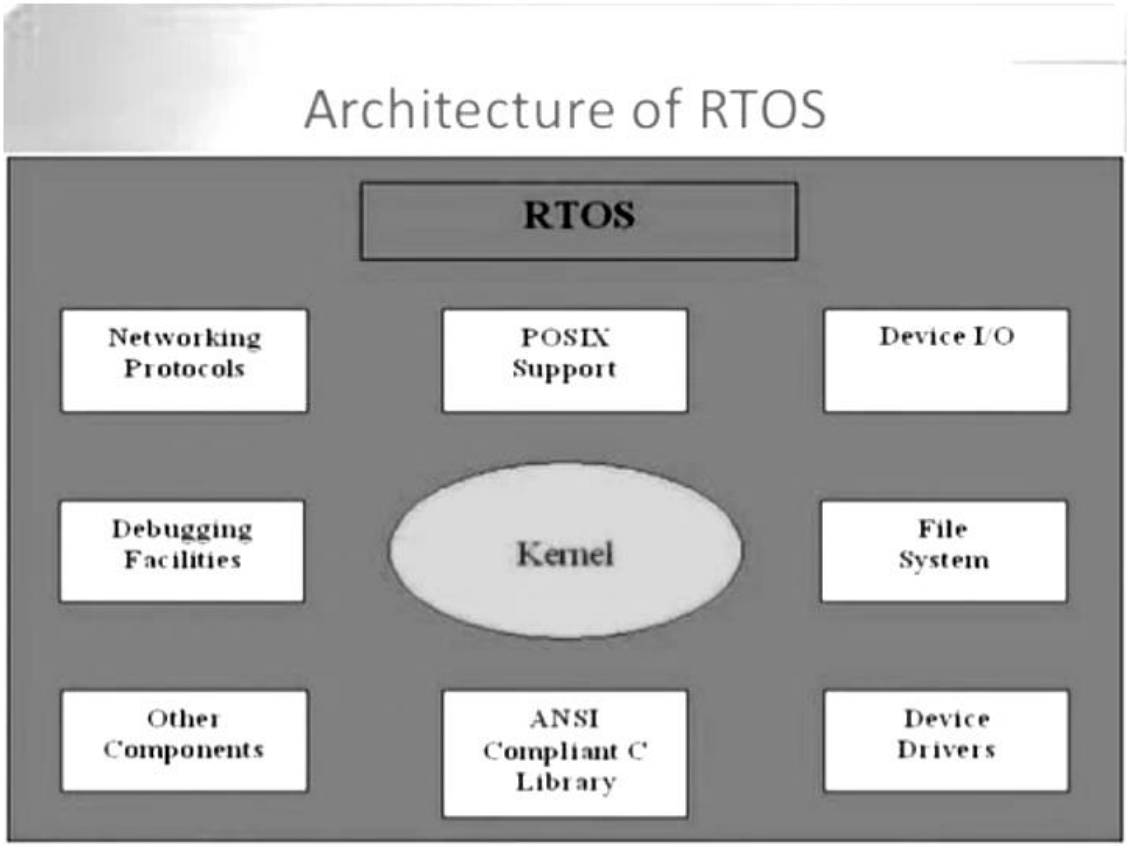
- **uKOS**
- **ARTOS (Locamation)**
- **ARTCOS (Robotu)**
- **Atomosher**
- **PSOS**

Architecture of kernel

The kernel is a computer program at the core of a computer's operating system and has complete control over everything in the system. It manages the operations of the computer and the hardware.

There are five types of kernels :

A micro kernel, which only contains basic functionality; A monolithic kernel, which contains many device drivers; Hybrid Kernel; Exokernel; Nanokernel



Architecture of kernel

The difference between kernel services and user services is

Kernel services	**User services**
Kernel services are low-level services that run in kernel mode, which is the privileged mode where the process has unrestricted access to system resources like hardware, memory, etc.	User services are higher-level services that run in user mode, which is the normal mode where the process has limited access and needs to request kernel services to access hardware, I/O or kernel data.
Kernel services are kept under kernel address space	User services are kept in user address space, which reduces the size of kernel and operating system.

Architecture of kernel

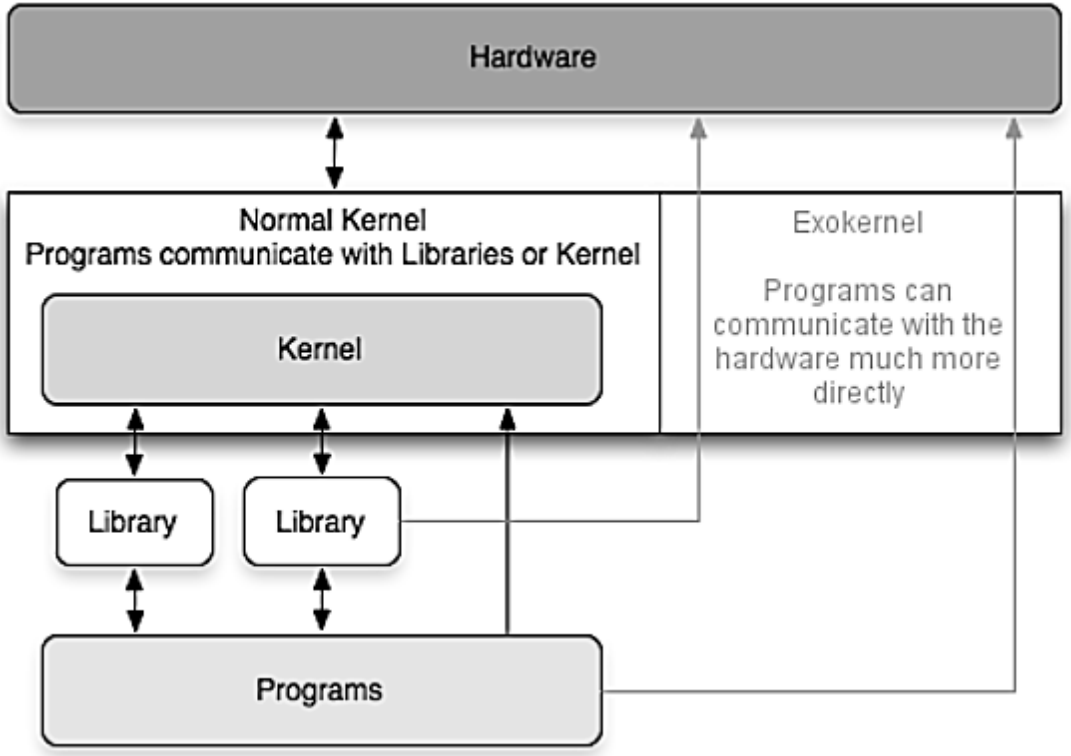
1. Microkernel:
In microkernel the user services and kernel services are implemented in different address space. The user services are kept in user address space, and kernel services are kept under kernel address space. Microkernel are smaller in size. Execution speed is low. Debugging is simple. **Example :** Mac OS X.

2. Monolithic kernel:
In Monolithic kernel, the entire operating system runs as a single program in kernel mode. The user services and kernel services are implemented in same address space. Monolithic kernel is larger than microkernel. Execution speed is high. Debugging is difficult. **Example :** Microsoft Windows 95.

3. Hybrid Kernel: A hybrid kernel is operating system kernel architecture that attempts to combine aspects and benefits of Microkernel and Monolithic kernel architectures used in operating systems.

Architecture of kernel

4. Exokernel: An Exokernel OS design pushes the boundaries of minimalism. Applications are given direct access to hardware resources, allowing them to manage resources and make decisions that the kernel previously made.



5. Nanokernel: A nanokernel is a small kernel that offers hardware abstraction, but without system services. Larger kernels are designed to offer more features and manage more hardware abstraction.

Functions of real-time operating systems

- **Task scheduling**
- **Interrupt handling**
- **Inter-task communication and synchronization**
- **Memory management**
- **Device driver management**

Tasks, process, threads

Tasks could also be considered operations that include input and output, while a process is a set of instructions that don't need any input.

Processes run in the background and might take input from the operating system, while tasks might be considered instructions with user input.

Thread is the smallest segment of instructions that can be handled independently by a scheduler.

Multiprocessing and multi-tasking

Multitasking: When a single resource is used to process multiple tasks then it is multitasking.

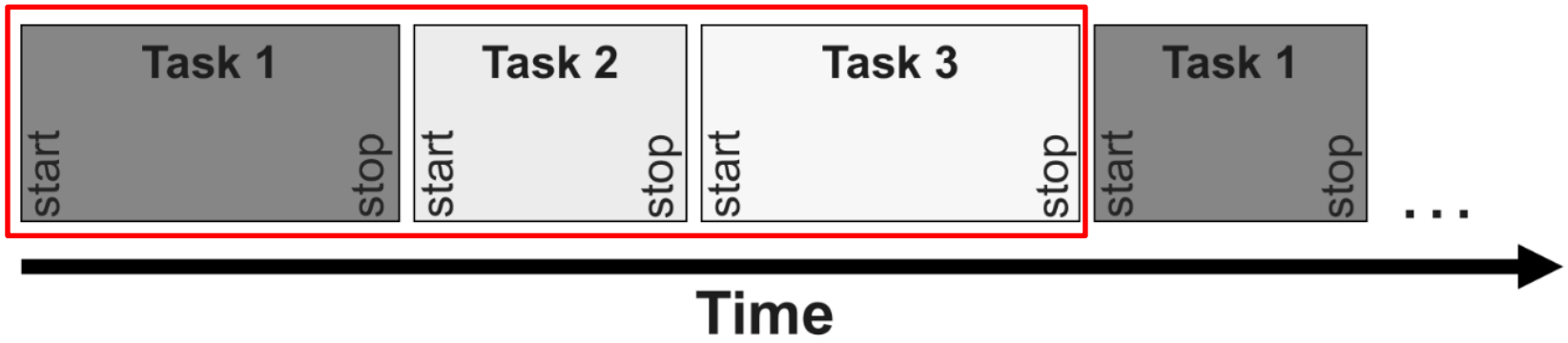
Multithreading: It is an extended form of multitasking.

Multiprocessing: When more than one processing unit is used by a single computer then it is called multiprocessing.

Task scheduling

Run to completion [RTC]

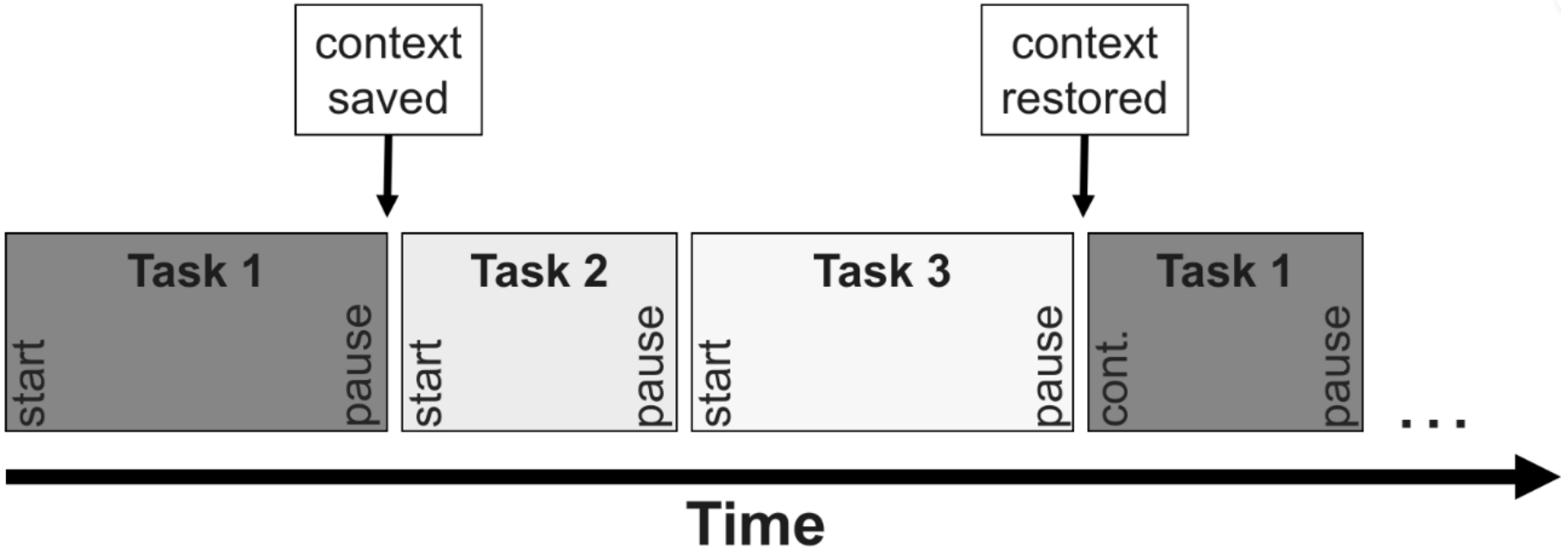
An RTC scheduler is very simple. **The idea is that one task runs until it has completed its work, then terminates. Then the next task runs similarly.** And so forth until all the tasks have run, when the sequence starts again.



Task scheduling

Round robin [RR]

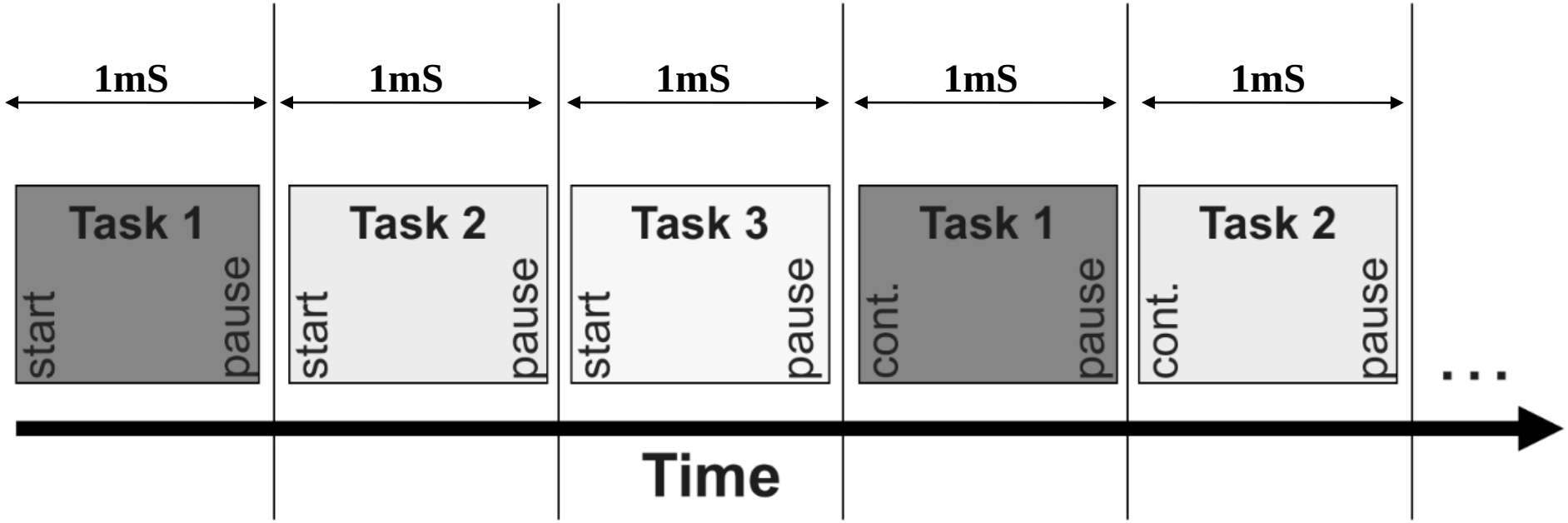
An RR scheduler is the next level of complexity. **Tasks are run in sequence in just the same way** (**with task suspend being a possibility**), **except that a task does not need to complete its work, it just relinquishes the CPU when convenient to do so. When it is scheduled again, it continues from where it left off.**



Task scheduling

Time slice [TS]

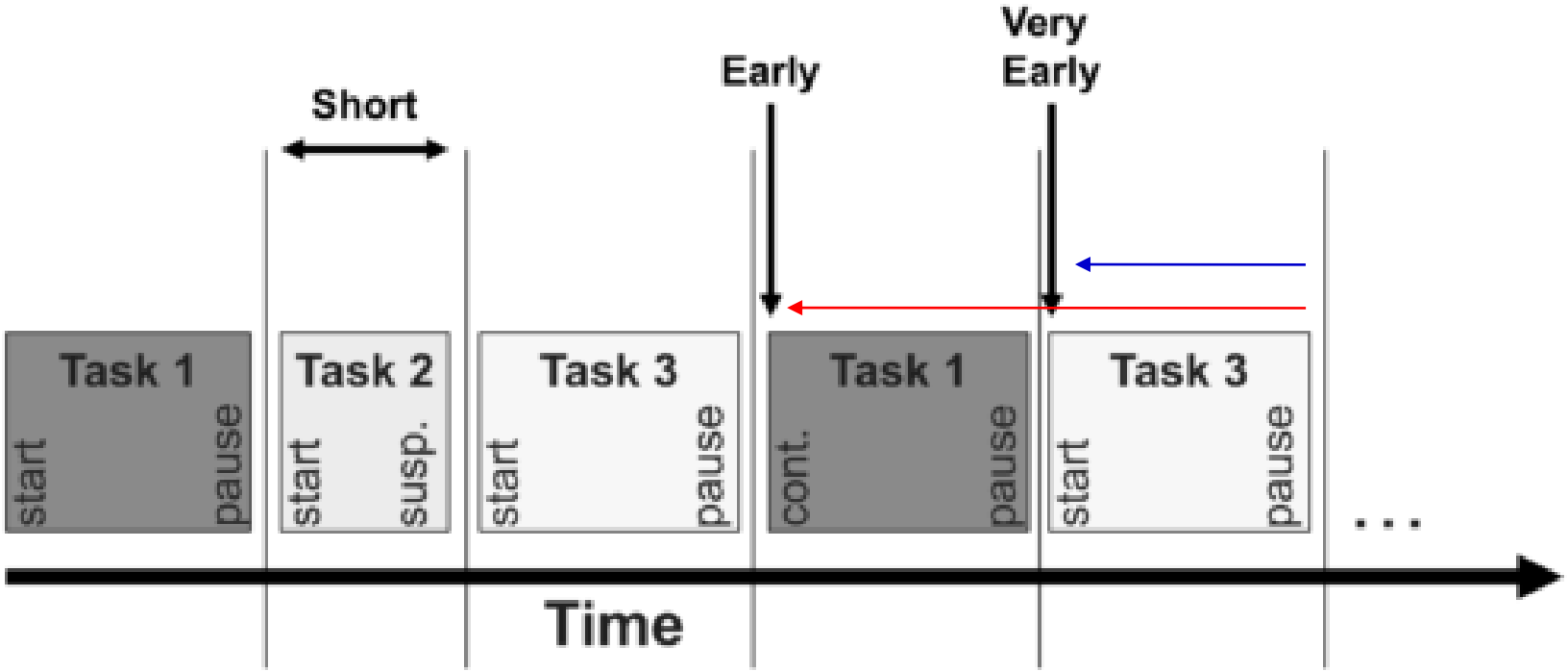
A TS scheduler is a straightforward example of "preemptive multitasking". **The idea is to divide time into "slots", each of which might be, say, 1mS. Each task gets to run in a slot. At the end of its allocated time, it is interrupted and the next task run.**



Task scheduling

Time slice with background task [TSBG]

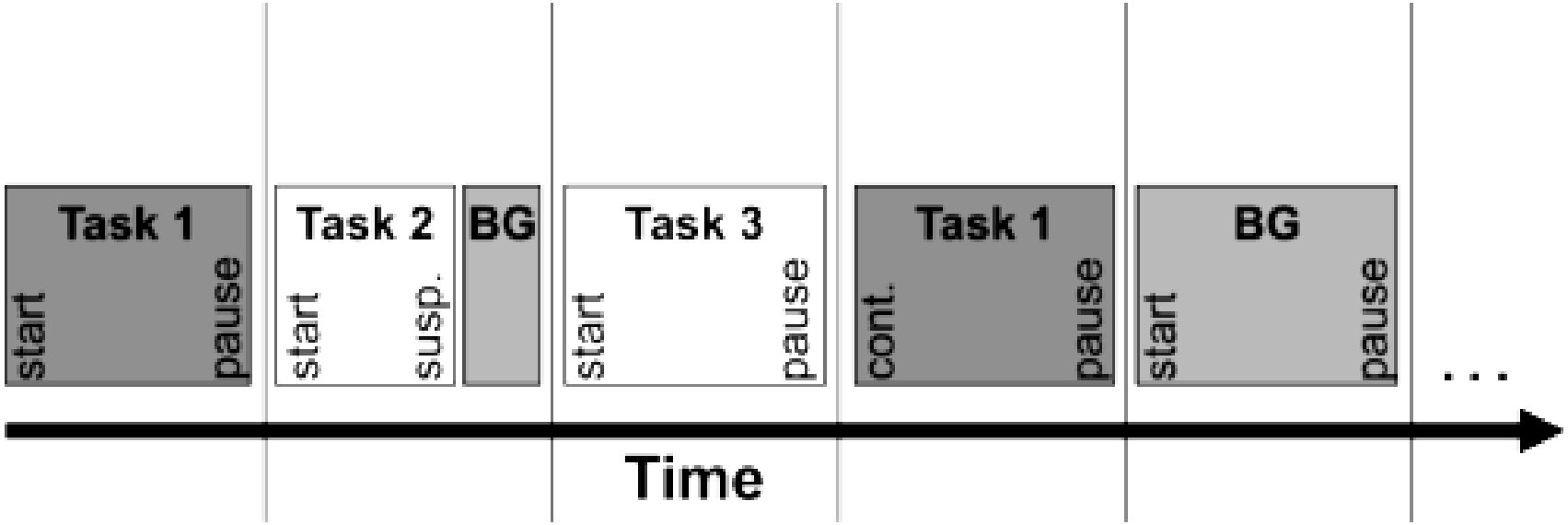
Although a **TS scheduler is neat and tidy, there is a problem. If a task finds that it has no work to do, its only option is to loop – burning CPU time – until it can do something useful**. This means that it might waste a significant proportion of its slot and an indefinite number of further slots. Clearly, **the task might suspend itself (to be woken again when it is needed), but this messes up the timing of the other task**



Task scheduling

Time slice with background task [TSBG]

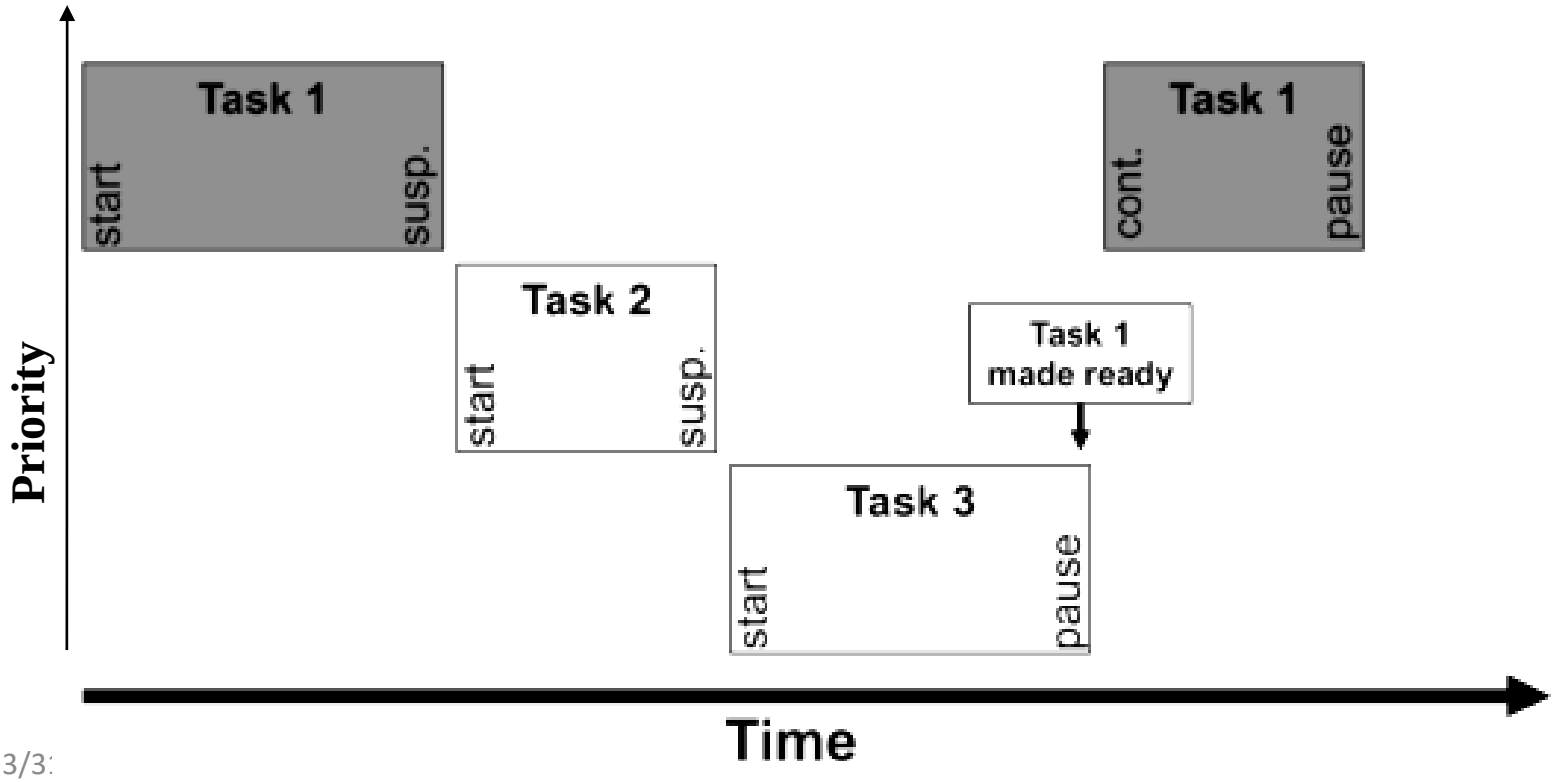
A solution is to enhance the scheduler so that, **if a task suspends itself, the remainder of its slot is taken up by a "background task**"; **this task would also use the full slots of any suspended tasks**. What the background task actually does depends on the application, but broadly it must be non-time-citical code – like self-testing. **This restores the timing integrity.**



Task scheduling

Priority [PRI]

A common, more sophisticated scheduling scheme is PRI, which is **used in many [most] commercial RTOS products**. **The idea is that each task has a priority and is either "ready" [to run] or "suspended**". **The scheduler runs the task with the highest priority that is "ready". When that task suspends, it runs the one with the next highest priority.**



Inter-task communication and synchronization

In multi-tasking model, we have seen that each task is a quasi-independent program. Although tasks in an embedded application have a degree of independence, it does not mean that they have no "awareness" of one another.

Although some tasks will be truly isolated from others, the requirement for communication and synchronization between tasks is very common.

This represents a key part of the functionality provided by an RTOS. The actual range of options offered by different RTOS may vary quite widely.

Inter-task Communication and Synchronization Options In Embedded/RTOS Systems

There are three broad standards for inter-task communications and synchronization:

Task-owned facilities – attributes that an RTOS imparts to tasks that provide communication (input) facilities. The example we will look at some more is signals.

Kernel objects – facilities provided by the RTOS which represent stand-alone communication or synchronization facilities. Examples include: event flags, mailboxes, queues/pipes, semaphores and mutexes.

Message passing – a rationalized scheme where an RTOS allows the creation of message objects, which may be sent from one to task to another or to several others. This is fundamental to the kernel design and leads to the description of such a product as being a "message passing RTOS"

Activities involved in Inter-task Communication and Synchronization

- **Shared Variables or Memory Areas:** A simplistic approach to inter-task communication is to just have variables or memory areas which are accessible to all the tasks concerned.
- **Signals:** Signals are probably the simplest inter-task communication facility offered in conventional RTOSes. They consist of a set of bit flags – there may be 8, 16 or 32, depending on the specific implementation – which is associated with a specific task. A signal flag (or several flags) may be set by any task using an OR type of operation. Only the task that owns the signals can read them. The reading process is generally destructive – i.e. the flags are also cleared.
- **Event Flag Groups:** Event flag groups are like signals in that they are a bit-oriented inter-task communication facility. They may similarly be implemented in groups of 8, 16 or 32 bits. They differ from signals in being independent kernel objects; they do not "belong" to any specific task. Any task may set and clear event flags using OR and AND operations. Likewise, any task may interrogate event flags using the same kind of operation.

Activities involved in Inter-task Communication and Synchronization

- **Semaphores:** Semaphores are independent kernel objects, which provide a flagging mechanism that is generally used to control access to a resource. Any task may attempt to obtain a semaphore in order to gain access to a resource. If the current semaphore value is greater than 0, the obtain will be successful, which decrements the semaphore value. In many OSes, it is possible to make a blocking call to obtain a semaphore; this means that a task may be suspended until the semaphore is released by another task. Any task may release a semaphore, which increments its value.
- **Mailboxes:** Mailboxes are independent kernel objects, which provide a means for tasks to transfer messages. Any task may send to a mailbox, which is then full. If a task then tries to send to send to a full mailbox, it will receive an error response. In many RTOSes, it is possible to make a blocking call to send to a mailbox; this means that a task may be suspended until the mailbox is read by another task. Any task may read from a mailbox, which renders it empty again. If a task tries read from an empty mailbox, it will receive an error response.
- **Queues:** Queues are independent kernel objects, that provide a means for tasks to transfer messages. They are a little more flexible and complex than mailboxes. The message size depends on the implementation, but will normally be a fixed size and word/pointer oriented.

Interrupt handling

Interrupts are signals that pause the normal execution of a program to handle specific events. To handle interrupts in an embedded system, follow these steps:

1. **Enabling Interrupts:** Configure the microcontroller to enable specific interrupts. This typically involves setting control registers and enabling the interrupt controller.
2. **Writing Interrupt Service Routines (ISRs):** Define ISRs for each enabled interrupt. ISRs are functions that handle the interrupt event. They are executed when the corresponding interrupt occurs.
3. **Priority and Nesting:** Assign priorities to interrupts to determine the order in which they are serviced when multiple interrupts occur simultaneously. Consider the nesting capability of the interrupt controller, which determines whether interrupts can interrupt other interrupts.
4. **Interrupt Context:** Be mindful of the context in which ISRs execute. ISRs have limitations in terms of memory usage, stack depth, and execution time. Avoid complex operations within ISRs and focus on handling the event promptly.
5. **Interrupt Handler Design:** Design interrupt handlers to be efficient and time-critical. Minimize the time spent in an interrupt handler to ensure timely responsiveness to other events.

Memory management

In a multiprogramming computer, the Operating System resides in a part of memory, and the rest is used by multiple processes. The task of subdividing the memory among different processes is called Memory Management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and updates status.

Why Memory Management is Required?

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process

Memory management: Logical and Physical Address Space

- **Logical Address Space:** An address generated by the CPU is known as a “Logical Address”. It is also known as a Virtual address. Logical address space can be defined as the size of the process. A logical address can be changed.

- **Physical Address Space:** An address seen by the memory unit (i.e the one loaded into the memory address register of the memory) is commonly known as a “Physical Address”. A Physical address is also known as a Real address. The set of all physical addresses corresponding to these logical addresses is known as Physical address space. A physical address is computed by MMU. The run-time mapping from virtual to physical addresses is done by a hardware device Memory Management Unit(MMU). The physical address always remains constant.

Memory management: Static and Dynamic Loading

- Loading a process into the main memory is done by a loader. There are two different types of loading :

- **Static Loading:** Static Loading is basically loading the entire program into a fixed address. It requires more memory space.

- **Dynamic Loading:** The entire program and all data of a process must be in physical memory for the process to execute. So, the size of a process is limited to the size of physical memory. To gain proper memory utilization, dynamic loading is used. In dynamic loading, a routine is not loaded until it is called. All routines are residing on disk in a relocatable load format. One of the advantages of dynamic loading is that the unused routine is never loaded. This loading is useful when a large amount of code is needed to handle it efficiently.

Memory management: Static and Dynamic Linking

- To perform a linking task a linker is used. A linker is a program that takes one or more object files generated by a compiler and combines them into a single executable file.

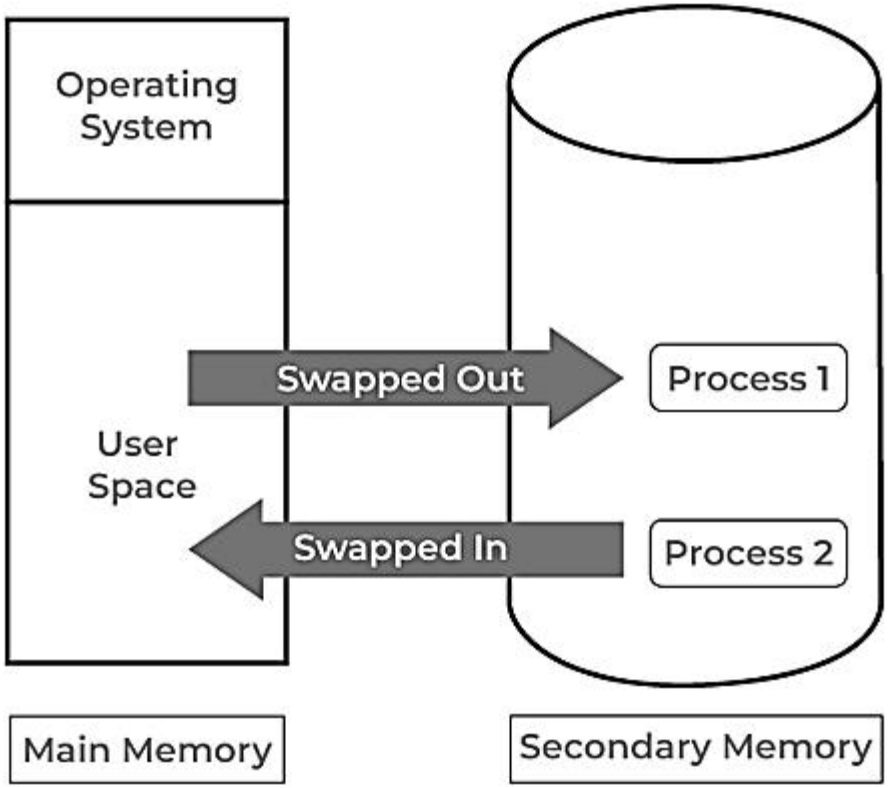
- **Static Linking:** In static linking, the linker combines all necessary program modules into a single executable program. So there is no runtime dependency. Some operating systems support only static linking, in which system language libraries are treated like any other object module.

- **Dynamic Linking:** The basic concept of dynamic linking is similar to dynamic loading. In dynamic linking, “Stub” is included for each appropriate library routine reference. A stub is a small piece of code. When the stub is executed, it checks whether the needed routine is already in memory or not. If not available then the program loads the routine into memory.

Memory management: Swapping

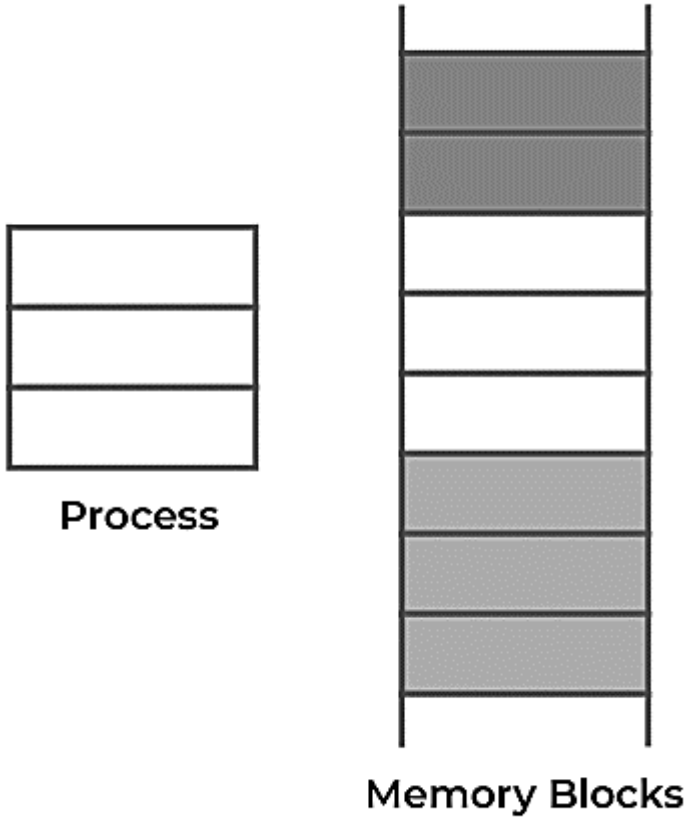
- When a process is executed it must have resided in memory. Swapping is a process of swapping a process temporarily into a secondary memory from the main memory, which is fast compared to secondary memory. A swapping allows more processes to be run and can be fit into memory at one time. The main part of swapping is transferred time and the total time is directly proportional to the amount of memory swapped. Swapping is also known as roll-out, or roll because if a **higher priority process arrives and wants service, the memory manager can swap out the lower priority process and then load and execute the higher priority process. After finishing higher priority work, the lower priority process swapped back in memory and continued to the execution process.**

Memory management: Swapping



Memory management: Contiguous Memory Allocation

- The main memory should accommodate both the operating system and the different client processes. Therefore, the allocation of memory becomes an important task in the operating system. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We normally need several user processes to reside in memory simultaneously. Therefore, we need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In adjacent memory allotment, each process is contained in a single contiguous segment of memory.



Memory Allocation

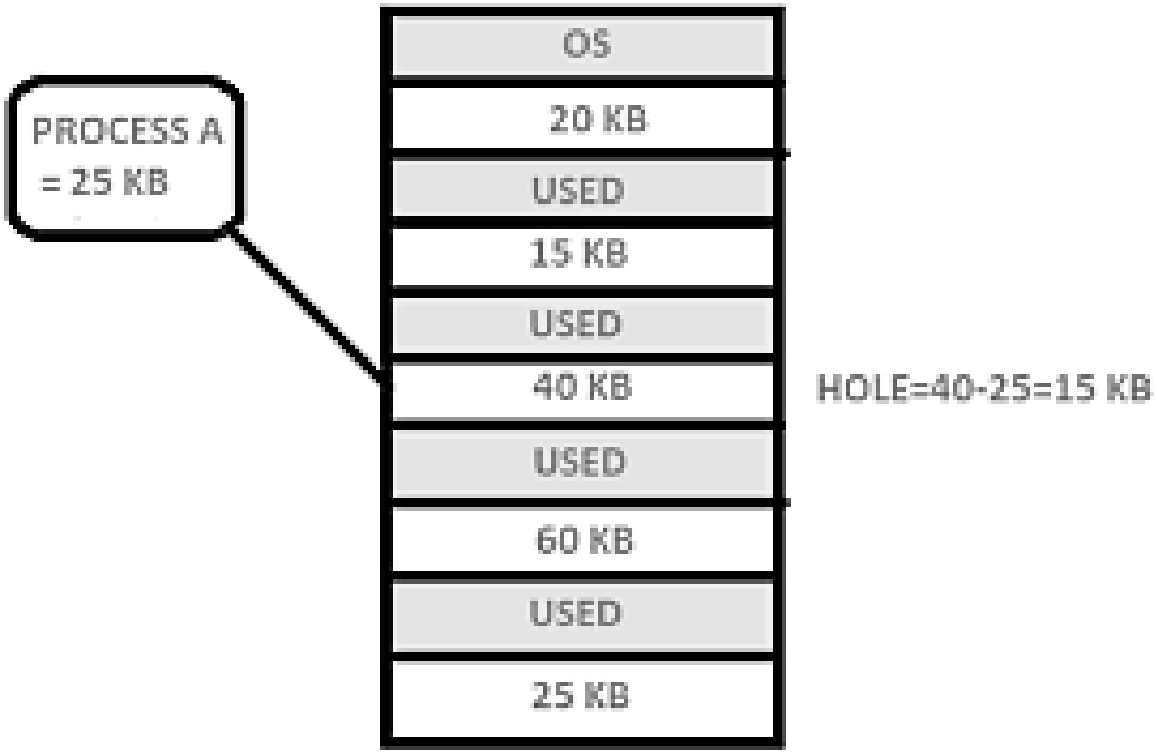
- To gain proper memory utilization, memory allocation must be allocated efficient manner. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions and each partition contains exactly one process. Thus, the degree of multiprogramming is obtained by the number of partitions.

- **Multiple partition allocation:** In this method, a process is selected from the input queue and loaded into the free partition. When the process terminates, the partition becomes available for other processes.

- **Fixed partition allocation:** In this method, the operating system maintains a table that indicates which parts of memory are available and which are occupied by processes. Initially, all memory is available for user processes and is considered one large block of available memory. This available memory is known as a “Hole”. When the process arrives and needs memory, we search for a hole that is large enough to store this process. If the requirement is fulfilled then we allocate memory to process, otherwise keeping the rest available to satisfy future requests. While allocating a memory sometimes dynamic storage allocation problems occur, which concerns how to satisfy a request of size n from a list of free holes. There are some solutions to this problem:

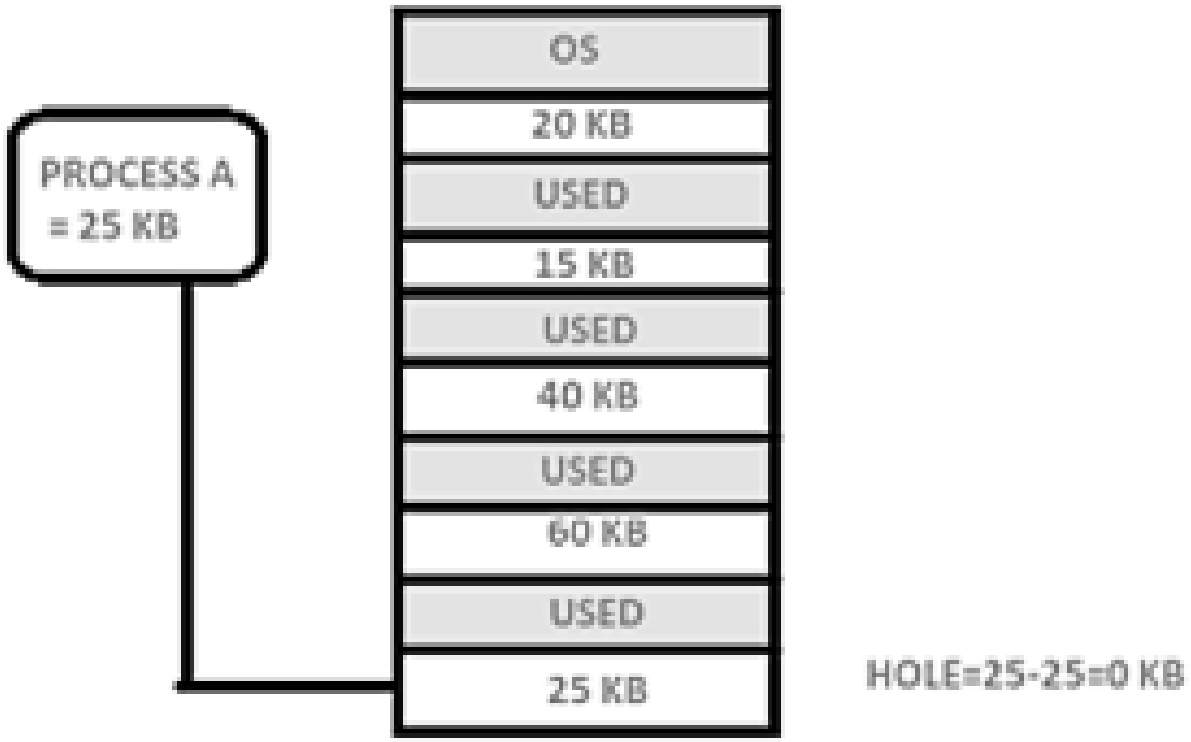
First Fit

In the First Fit, the first available free hole fulfil the requirement of the process allocated.



Here, in this diagram, a 40 KB memory block is the first available free hole that can store process A (size of 25 KB), because the first two blocks did not have sufficient memory space.

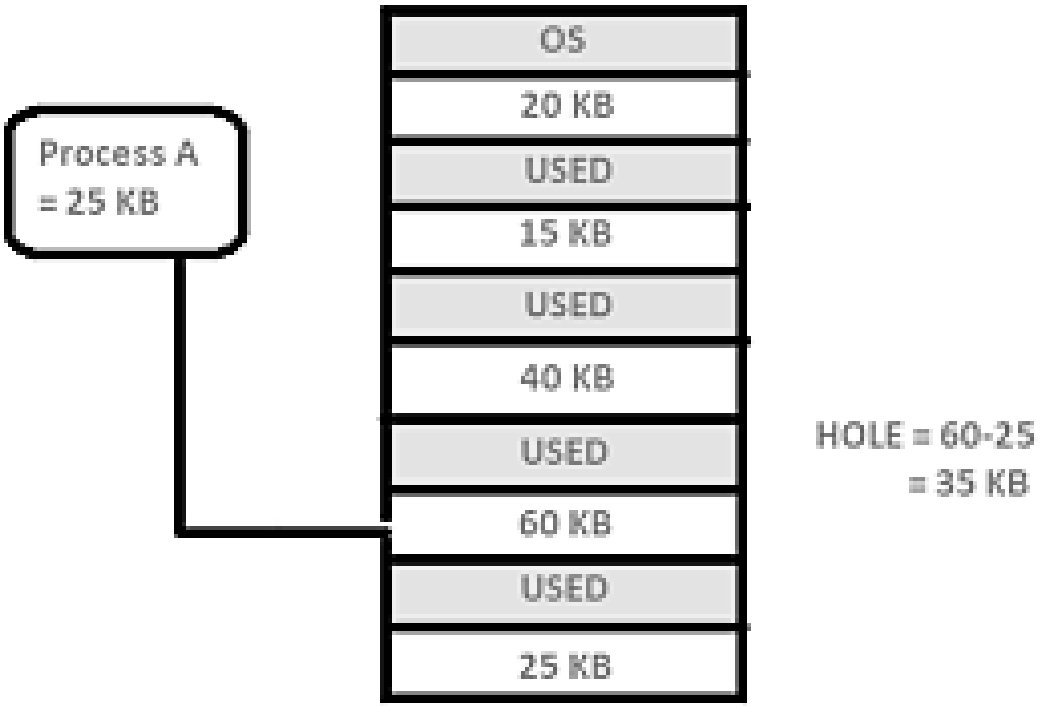
Best Fit



In the Best Fit, allocate the smallest hole that is big enough to process requirements. For this, we search the entire list, unless the list is ordered by size.

Here in this example, first, we traverse the complete list and find the last hole 25KB is the best suitable hole for Process A(size 25KB). In this method, memory utilization is maximum as compared to other memory allocation techniques.

Worst Fit



In the Worst Fit, allocate the largest available hole to process. This method produces the largest leftover hole.

Here in this example, Process A (Size 25 KB) is allocated to the largest available memory block which is 60KB. Inefficient memory utilization is a major issue in the worst fit.

Fragmentation

- Fragmentation is defined as when the process is loaded and removed after execution from memory, it creates a small free hole. These holes can not be assigned to new processes because holes are not combined or do not fulfill the memory requirement of the process. To achieve a degree of multiprogramming, we must reduce the waste of memory or fragmentation problems. In the operating systems two types of fragmentation:

- **Internal fragmentation:** Internal fragmentation occurs when memory blocks are allocated to the process more than their requested size. Due to this some unused space is left over and creating an internal fragmentation problem. **Example:** Suppose there is a fixed partitioning used for memory allocation and the different sizes of blocks 3MB, 6MB, and 7MB space in memory. Now a new process p4 of size 2MB comes and demands a block of memory. It gets a memory block of 3MB but 1MB block of memory is a waste, and it can not be allocated to other processes too. This is called internal fragmentation.

- **External fragmentation:** In External Fragmentation, we have a free memory block, but we can not assign it to a process because blocks are not contiguous. **Example:** Suppose (consider the above example) three processes p1, p2, and p3 come with sizes 2MB, 4MB, and 7MB respectively. Now they get memory blocks of size 3MB, 6MB, and 7MB allocated respectively. After allocating the process p1 process and the p2 process left 1MB and 2MB. Suppose a new process p4 comes and demands a 3MB block of memory, which is available, but we can not assign it because free memory space is not contiguous. This is called external fragmentation.

Fragmentation

- Both the first-fit and best-fit systems for memory allocation are affected by external fragmentation. To overcome the external fragmentation problem Compaction is used. In the compaction technique, all free memory space combines and makes one large block. So, this space can be used by other processes effectively.

- Another possible solution to the external fragmentation is to allow the logical address space of the processes to be noncontiguous, thus permitting a process to be allocated physical memory wherever the latter is available.

Device management

Device Administration within An operating system controls every piece of hardware and virtual device on a PC or computer. Input/output devices are assigned to processes by the device management system based on their importance. Depending on the situation, these devices may also be temporarily or permanently reallocated.

Usually, systems are hardware or physical devices like computers, laptops, servers, cell phones, etc. Additionally, they might be virtual, like virtual switches or machines. A program may require a variety of computer resources (devices) to go through to the end. It is the operating system's responsibility to allocate resources wisely. The operating system is alone in charge of determining if the resource is available. It deals not only with device allocation but also with deallocation, which means that a device or resource must be removed from a process once its use is over.

Functions of device management

1. Keeps track of all devices and the program that is responsible for performing this is called the I/O controller.
2. Monitoring the status of each device such as storage drivers, printers, and other peripheral devices.

Enforcing preset policies and making a decision on which process gets the device when and for how long.

3. Allocates and deallocates the device efficiently.

Features of Device Management in OS

- The operating system is responsible in managing device communication through their respective drivers.
- The operating system keeps track of all devices by using a program known as an input-output controller.
- It decides which process to assign to CPU and for how long.
- O.S. is responsible in fulfilling the request of devices to access the process.
- It connects the devices to various programs in an efficient way without error.
- Deallocate devices when they are not in use.

Types of Devices

1. Dedicated Device

Certain devices are assigned to only one task at a time in device management until that task releases them. Plotters, printers, tape drives, and other similar devices require this kind of allocation method because sharing them with numerous users at the same time will be inconvenient. The drawback of these devices is the inefficiency that results from assigning the device to a single user throughout the entirety of the task execution process, even in cases when the device is not utilized exclusively.

2. Shared Device

There are numerous processes that these devices could be assigned to. Disk-DASD could be shared concurrently by many processes by interleaving their requests. All issues must be resolved by pre-established policies, and the Device Manager closely monitors the interleaving.

3. Virtual Device

Virtual devices are dedicated devices that have been converted into shared devices, making them a hybrid of the two types of devices. For instance, a spooling programme that routes all print requests to a disc can turn a printer into a sharing device. A print job is routed to the disc and not delivered straight to the printer until it is ready with all the necessary formatting and sequencing, at which time it is sent to the printers. The method can increase usability and performance by turning a single printer into a number of virtual printers.

What are the Various Techniques for Accessing a Device?

Polling: In this instance, a CPU keeps an eye on the status of the device to share data. Busy-waiting is a drawback, but simplicity is a plus. In this scenario, when an input/output operation is needed, the computer simply keeps track of the I/O device's status until it's ready, at which time it is accessed. Stated differently, the computer waits for the device to be ready.

Interrupt-Driven I/O: Notifying the associated driver of the device's availability is the device controller's job. One interrupt for each keyboard input results in slower data copying and movement for character devices, but the advantages include more effective use of CPU cycles. A block of bytes is created from a serial bit stream by a device controller. It also does error correction if needed. It consists of two primary parts: a data buffer that an operating system can read or write to, and device registers for communication with the CPU.

DMA(Direct Memory Access) : Data motions are carried out by using a second controller. This approach has the benefit of not requiring the CPU to duplicate data, but it also has the drawback of preventing a process from accessing data that is in transit.

Double Buffering: This mode of access has two buffers. One fills up while the other is utilised, and vice versa. In order to hide the line-by-line scanning from the viewer, this technique is frequently employed in animation and graphics.

Context switching

Context switching in an operating system involves saving the context or state of a running process so that it can be restored later, and then loading the context or state of another. process and run it.

Context Switching refers to the process/method used by the system to change the process from one state to another using the CPUs present in the system to perform its job.

Context switching

Example of Context Switching

Suppose in the OS there (N) numbers of processes are stored in a Process Control Block(PCB). like The process is running using the CPU to do its job. While a process is running, other processes with the highest priority queue up to use the CPU to complete their job.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds. Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch

Context switching

Need of Context Switching

Context switching enables all processes to share a single CPU to finish their execution and store the status of the system's tasks. The execution of the process begins at the same place where there is a conflict when the process is reloaded into the system.

The operating system's need for context switching is explained by the reasons listed below. One process does not directly switch to another within the system. Context switching makes it easier for the operating system to use the CPU's resources to carry out its tasks and store its context while switching between multiple processes.

Context switching enables all processes to share a single CPU to finish their execution and store the status of the system's tasks. The execution of the process begins at the same place where there is a conflict when the process is reloaded into the system.

Context switching only allows a single CPU to handle multiple processes requests parallelly without the need for any additional processors.

Context switching

Context Switching Triggers

The three different categories of context-switching triggers are as follows.
Interrupts
Multitasking
User/Kernel switch

Interrupts: When a CPU requests that data be read from a disc, if any interruptions occur, context switching automatically switches to a component of the hardware that can handle the interruptions more quickly.

Multitasking: The ability for a process to be switched from the CPU so that another process can run is known as context switching. When a process is switched, the previous state is retained so that the process can continue running at the same spot in the system.

Kernel/User Switch: This trigger is used when the OS needed to switch between the user mode and kernel mode.
When switching between user mode and kernel/user mode is necessary, operating systems use the kernel/user switch.

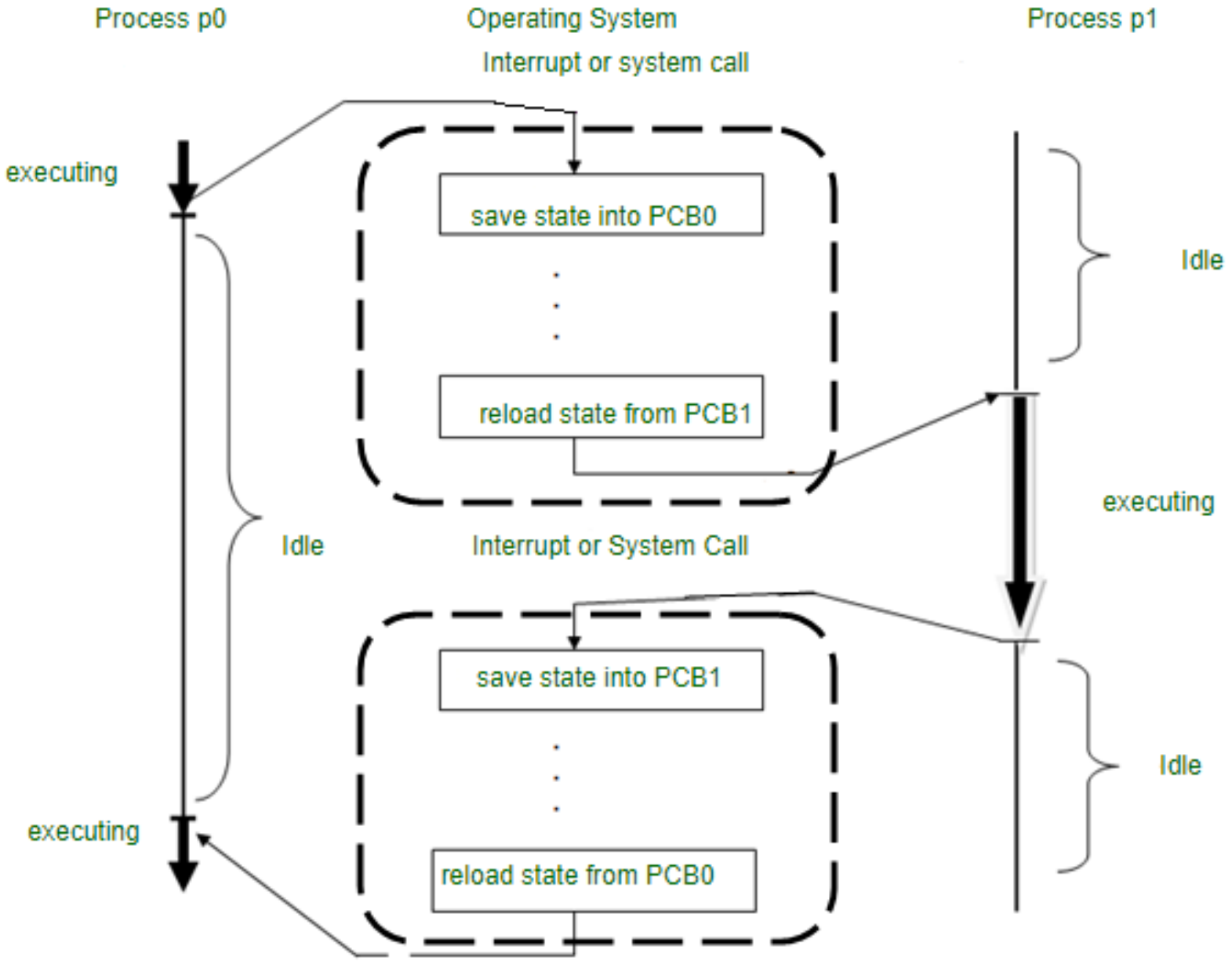
Context switching

What is Process Control Block(PCB)?

The Process Control block(PCB) is also known as a Task Control Block. It represents a process in the Operating System. **A process control block (PCB) is a data structure used by a computer to store all information about a process.** It is also called the descriptive process. When a process is created (started or installed), the operating system creates a process manager.

Context of a process needs to store in PCB: Content of program counter, content of register, I/O devices related to that particular process, files accessed by this particular process.

State Diagram of Context switching



Activities that are involved in context switching

- The context switching of two processes, the priority-based process occurs in the ready queue of the process control block. These are the following steps.
- The state of the current process must be saved for rescheduling.
- The process state contains records, credentials, and operating system-specific information stored on the PCB or switch.
- The PCB can be stored in a single layer in kernel memory or in a custom OS file.
- A handle has been added to the PCB to have the system ready to run.
- The operating system aborts the execution of the current process and selects a process from the waiting list by tuning its PCB.
- Load the PCB's program counter and continue execution in the selected process.
- Process/thread values can affect which processes are selected from the queue, this can be important.

Process life cycle

Process life cycle includes several stages that it passes through from beginning to end.

- **New (Create):** In this step, the process is about to be created but not yet created. It is the program that is present in secondary memory that will be picked up by OS to create the process.
- **Ready:** New -> Ready to run. After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue called ready queue for ready processes.
- **Run:** The process is chosen from the ready queue by the CPU for execution and the instructions within the process are executed by any one of the available CPU cores.
- **Blocked or Wait:** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already acquired) it enters the blocked or waits state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.
- **Terminated or Completed:** Process is killed as well as PCB is deleted. The resources allocated to the process will be released or deallocated.

Process life cycle

- **Suspend Ready:** Process that was initially in the ready state but was swapped out of main memory(refer to Virtual Memory topic) and placed onto external storage by the scheduler is said to be in suspend ready state. The process will transition back to a ready state whenever the process is again brought onto the main memory.
- **Suspend wait or suspend blocked:** Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.