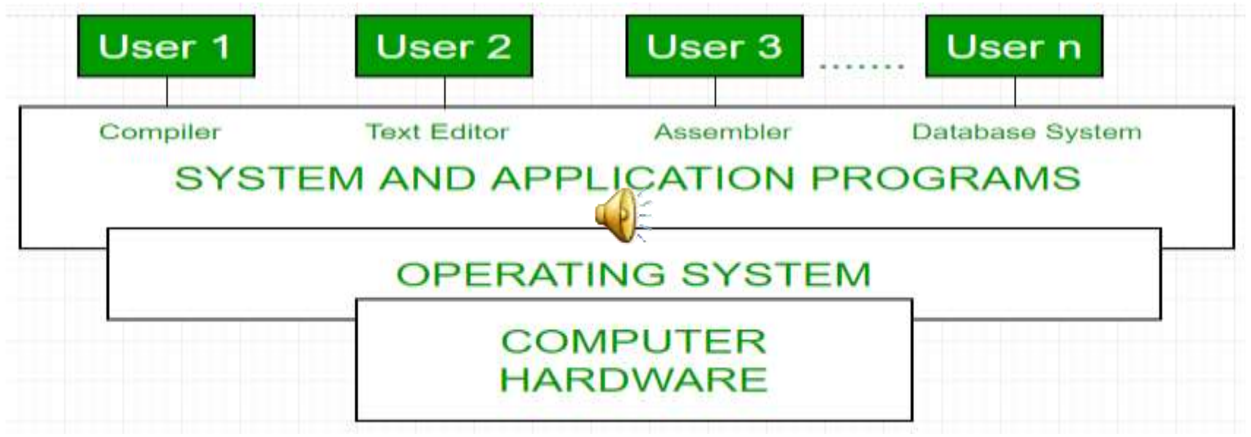
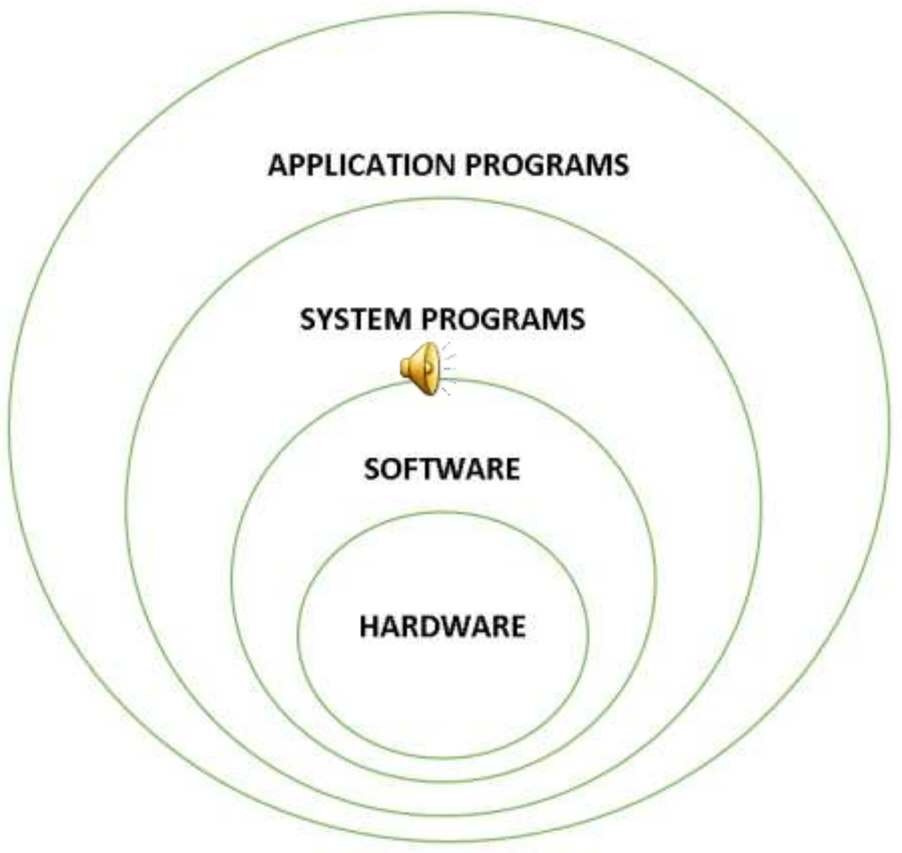
Real Time Operating Systems

Abstract View of Components of an Operating System

- An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.



Structure of Operating System:



Operating System Architecture

Structure of Operating System:

- The structure of OS consists of 4 layers:
- Hardware Hardware consists of CPU, Main memory, I/O Devices, etc,
- Software (Operating System) Software includes process management routines, memory management routines, I/O control routines, file management routines.
- System programs This layer consists of compilers, Assemblers, linker etc.
- Application programs This is dependent on users need.

Ex. Railway reservation system, Bank database management

Goals / Function of Operating system

- Following are some of important functions of an operating System:
- Execute user programs and make solving user problems easier.
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.
- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

- Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.
- Main memory provides a fast storage that can be access directly by the CPU. So for a program to be executed, it must in the main memory.
- Operating System does the following activities for memory management:
- Keeps tracks of primary memory i.e. what part of it are in use by whom, what part are not in use.
- In multiprogramming, OS decides which process will get memory when and how much.
- Allocates the memory when the process requests it to do so.
- De-allocates the memory when the process no longer needs it or has been terminated

Managing Strategy	Explanation
Fixed-blocks Allocation	Memory address space is divided into blocks with processes having small address spaces getting a lesser number of blocks and processes with big address spaces getting a larger number of blocks.
Dynamic-blocks Allocation	Memory address space is divided into blocks with processes having small address spaces getting a lesser number of blocks and processes, with big address spaces getting a larger number of blocks to start with. The memory manager later allocates variable size blocks (in units of say 64 or 256 bytes) dynamically allocated from a free (unused) list of memory-blocks description table at the different computation phases of a process.
Dynamic Page-Allocation	Memory has fixed sized blocks called pages and the memory manager allocates the pages dynamically with a page descriptor table.
Dynamic Data Memory Allocation	The manager allocates memory dynamically to different data structures like nodes of a list, queues, and stacks.
Dynamic Address-relocation	The manager dynamically allocates the addresses initially bound to the relative addresses after adding the relative address with relocation register. The memory manager now dynamically changes only the contents of a relocation register. It takes into account a limit defining register so that the relocated addresses are within the limit of available addresses. This is also called run-time dynamic address binding.
Multiprocessor Memory Allocation	The manager adopts an allocation strategy either the memory is shared with tight coupling between two or more processors or shared with loose coupling or there is a multi segmented allocations.

Processor Management

- In multiprogramming environment, OS decides which process gets the processor when and how much time. This function is called process scheduling.
- Operating System does the following activities for processor management.
- Keeps tracks of processor and status of process. Program responsible for this task is known as traffic controller.
- Allocates the processor (CPU) to a process.
- De-allocates processor when processor is no longer required.

Process Creation

- At reset of processor in a computer system, an OS is initialized first and then a process, which can be called initial process, is created.
- Initialisation of OS means enabling the use of OIS functions, which includes the function to create the process.
- Then the OS is started and runs the initial process.
- Processes can be created hierarchically.
- The initial process creates subsequent processes.
- Creation of a process means specifying the resources for the process and address spaces (memory block) for the created process, stack, data and heap and placing the initial information at a PCB.
- The process manager allocates a PCB when it creates the process and later manages it.
- PCB (Process Control Block) is a process descriptor used by process manager.
- A PCB describes the following.

(i) Context [Processor status word, program counter, stack pointer and other CPU registers at the instant of last instruction run executed when the process was left and processor switched to other process)

(ii) Process stack pointer

(iii) Current state [Is it created, activated or spawned? Is it running? Is it blocked?]

(iv) Addresses that are allocated and that are presently in use

(v) Pointer for the parent process in case there exists a hierarchy of the processes

(vi) Pointer to a list of daughter processes (processes lower in the hierarchy)

(vii) Pointer to a list of resources, which are usable (consumed) only once. For example, input data, memory buffer or pipe, mailbox message, semaphore. [There may be producers and consumers of these resources.]

(viii) Pointer to a list of resource-types usable more than once [A resource type example is a memory block. Another example is an IO port.] Each resource type will have a count of these types. For example, the number of memory blocks or the number of IO ports.

(ix) Pointer to queue of messages. It is considered as a special case of resources that are usable once. It is because messages from the OS also queue up to be controlled by a process.

(x) Pointer to Access-permissions descriptor for sharing a set of resources globally. and with another process.

(xi) ID by which identification is made by the process manager

Device Management

- OS manages device communication via their respective drivers.
- Operating System does the following activities for device management:
- Keeps tracks of all devices. Program responsible for this task is known as the I/O controller.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

- Device manger manages the physical as well as virtual devices like the pipes and sockets through a common strategy.
- Device management has three standard approaches to the three types device drivers:
- (i) Programmed I/Os by polling the service need from each device.
- (ii) Interrupt(s) from the device drivers ISR and
- (iii) Device uses DMA operation used by the devices to access the memory. Most common is the use of device driver ISRs

The functions of device manager are given below:

- Device Detection and Addition
- Device Deletion
- Device Allocation and Registration
- Detaching and DeregistrationRestricting
- Device to a specific process
- Device Sharing
- Device control
- Device Access Management
- Device Buffer Management
- Device Queue, Circular-queue or blocks of queues Management
- Device drivers updating and upload of new device-functions
- Backup and restoration

OS command functions for a device:

- **create &open-**create is for creating and open is for creating(if not created earlier) and for configuring and initializing the device
- **write** –write into the device buffer or sending output from the device
- **read -**write from the device buffer or reading input from the device
- **close &delete** –close is for deregistering the device from the system and delete id for close(if not closed earlier) and for detaching the device

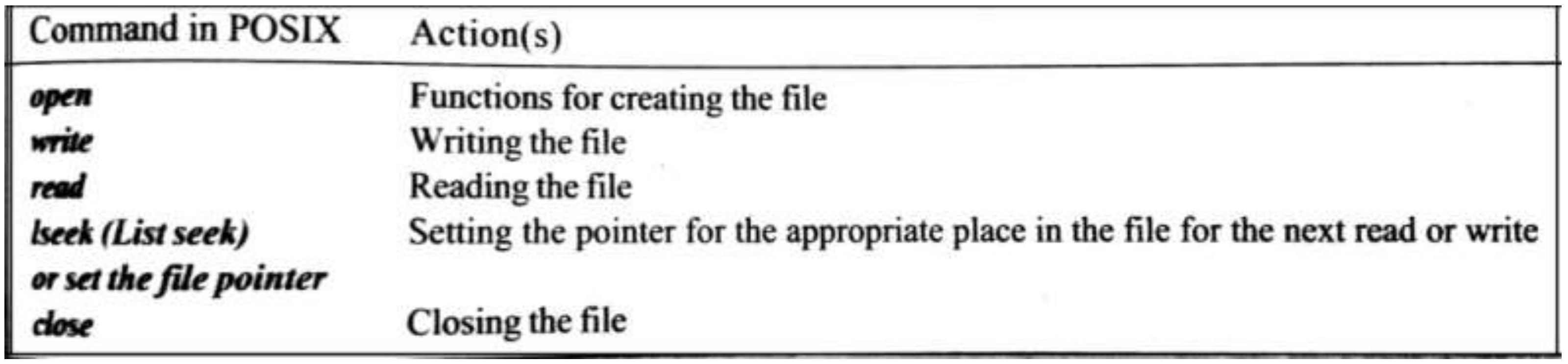
File Management

- A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.
- Operating System does the following activities for file management:
- Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

- A file is a named entity on a magnetic disc, or optical disc, or system memory or memory stick
- File contains the data, characters and text.
- **<u>Different abstractions of a file:</u>**
- A file may be a named entity that is a structured record named entity as on a disk, having random access in the system
- May be a structured record on a RAM analogous to a disk and may also be either separately called as ‘RAM disk’or simply, as a ‘file’ itself (virtual device).
- May be an unstructured record of bits or bytes
- A file device may be a pipe -like device

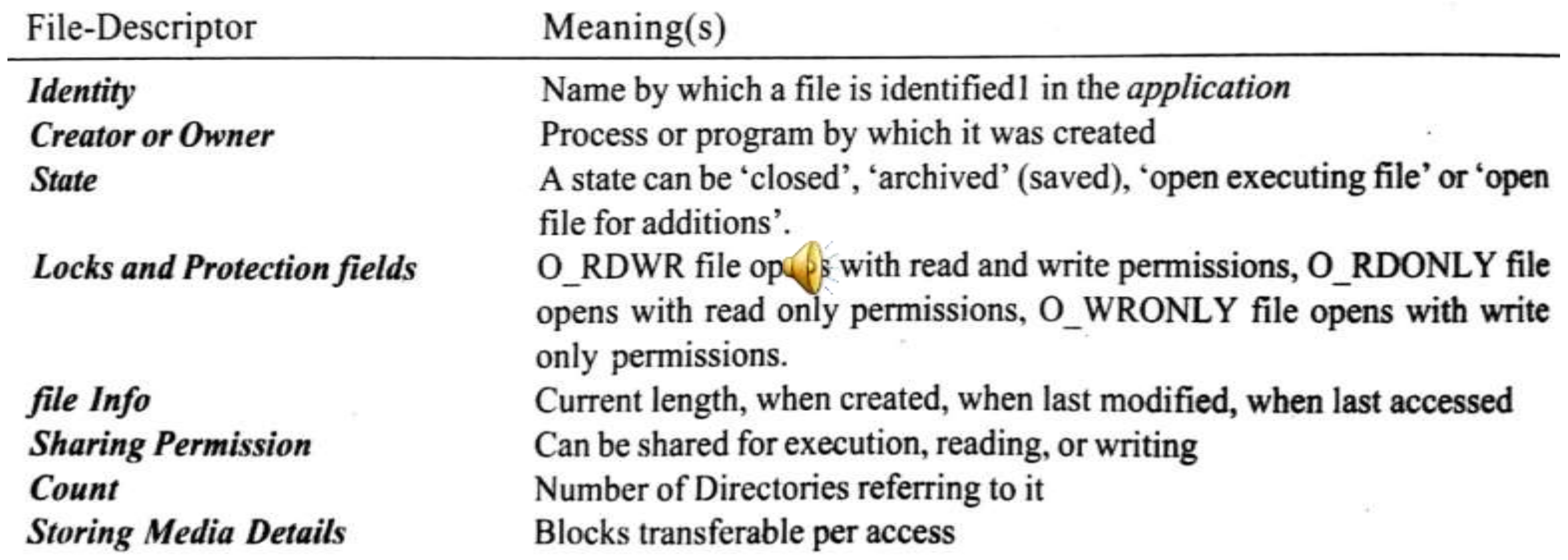
- **<u>File organization in a system</u>**

- File is organized in a way according to a file system, which has set of command functions for operations on the file.
- Table below gives these functions for POSIX file system.



Command in POSIX	Action(s)
open	Functions for creating the file
write	Writing the file
read	Reading the file
lseek (List seek) or set the file pointer	Setting the pointer for the appropriate place in the file for the next read or write
close	Closing the file

- A file system has a data structure called file descriptor as per the table below:



File-Descriptor	Meaning(s)
Identity	Name by which a file is identified1 in the *application*
Creator or Owner	Process or program by which it was created
State	A state can be 'closed', 'archived' (saved), 'open executing file' or 'open file for additions'.
Locks and Protection fields	O_RDWR file opens with read and write permissions, O_RDONLY file opens with read only permissions, O_WRONLY file opens with write only permissions.
file Info	Current length, when created, when last modified, when last accessed
Sharing Permission	Can be shared for execution, reading, or writing
Count	Number of Directories referring to it
Storing Media Details	Blocks transferable per access

Other Important Activities

- Following are some of the important activities that Operating System does.
- **Security** -- By means of password and similar other techniques, preventing unauthorized access to programs and data.
- **Control over system performance** -- Recording delays between request for a service and response from the system.
- **Job accounting** -- Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** -- Production of dumps, traces, error messages and other debugging and error detecting aids.
- **Coordination between other software and users** -- Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

- **Operating System handles**
- Memory Addressing & Management
- Interrupt & Exception Handling
- Process & Task Management
- File System
- Timing
- Process Scheduling & Synchronization
- **Examples of Operating Systems**
- RTOS – Real-Time Operating System
- Single-user, Single-task: example PalmOS
- Single-user, Multi-task: MS Windows and MacOS
- Multi-user, Multi-task: UNIX

Micro C/OS-II RTOS (MUCOS)

- One of the popular RTOS for an embedded system is μC/OS-II.
- μC/OS-II is a free ware for non commercial use.
- J. Labrosse designed it in 1992
- μC/OS-II name derives from MicroController Operating System
- Preemptive RTOS
- Multitasking
- Deterministic
- Portable as ROM image
- Scalable –only needed OS functions become part of application code.
- Different Platforms support

System Level Functions

- **void OSInit (void)**- is used to initiate the OS. Use is compulsory before calling any OS kernel functions
- **void OSStart (void)-**is used to start the initiated operating system and created tasks Its use is compulsory for the multitasking OS kernel operations
- **void OSIntEnter (void)** -used at the start of ISR .For sending a message to RTOS kernel for taking control.compulsory to let OS kernel control the nesting of the ISRs in case of occurrences of multiple interrupts of varying priorities
- **void OSIntExit (void)** -used just before the return from the running ISR ─ For sending a message to RTOS kernel for quitting control of presently running ISR
- **OS_ENTER_CRITICAL** ─is used at the start of the ISR. Macro to disable interrupts before a critical section .It is for sending a message to RTOS kernel for disabling interrupts .

- **OS_EXIT_CRITICAL** ─ Macro to enable interrupts. [ENTER and EXIT functions form a pair in the critical section] is used just before the return from the critical section. It is for sending a message to RTOS kernel for quitting control from the section .

- **void OSTickInit (void)** ─ is used to initiate the system clock ticks and interrupts at regular intervals as per OS_TICKS_PER_SEC predefined when defining configuration of MUCOS

Task Service Functions

- Service functions mean the functions to task create, suspend and resume, and time setting and time retrieving functions.
- 1. Function for Creating a task:
- **Unsigned byte OSTaskCreate(void(*task)(void*taskPointer),void*pmdata,OS_STK*taskStackPointer, unsigned byte taskPriority)**

***taskPointer** ─ task is a pointer to the task code

***pmdata**─ is a pointer to an argument that is passed to your task when it starts executing,

***taskStackpointer** -is a pointer to the top of the stack that is assigned to the task

task priority- is the desired task priority.

- This function returns:
- OS_NO_ERR when creation succeeds
- OS_PRIO_EXIST if priority value that passed already exist
- OS_NO_ MORE_TCB when no more memory block for task control is available
- OS_PRIO_INVALID if the priority value that is passed is more than the given range

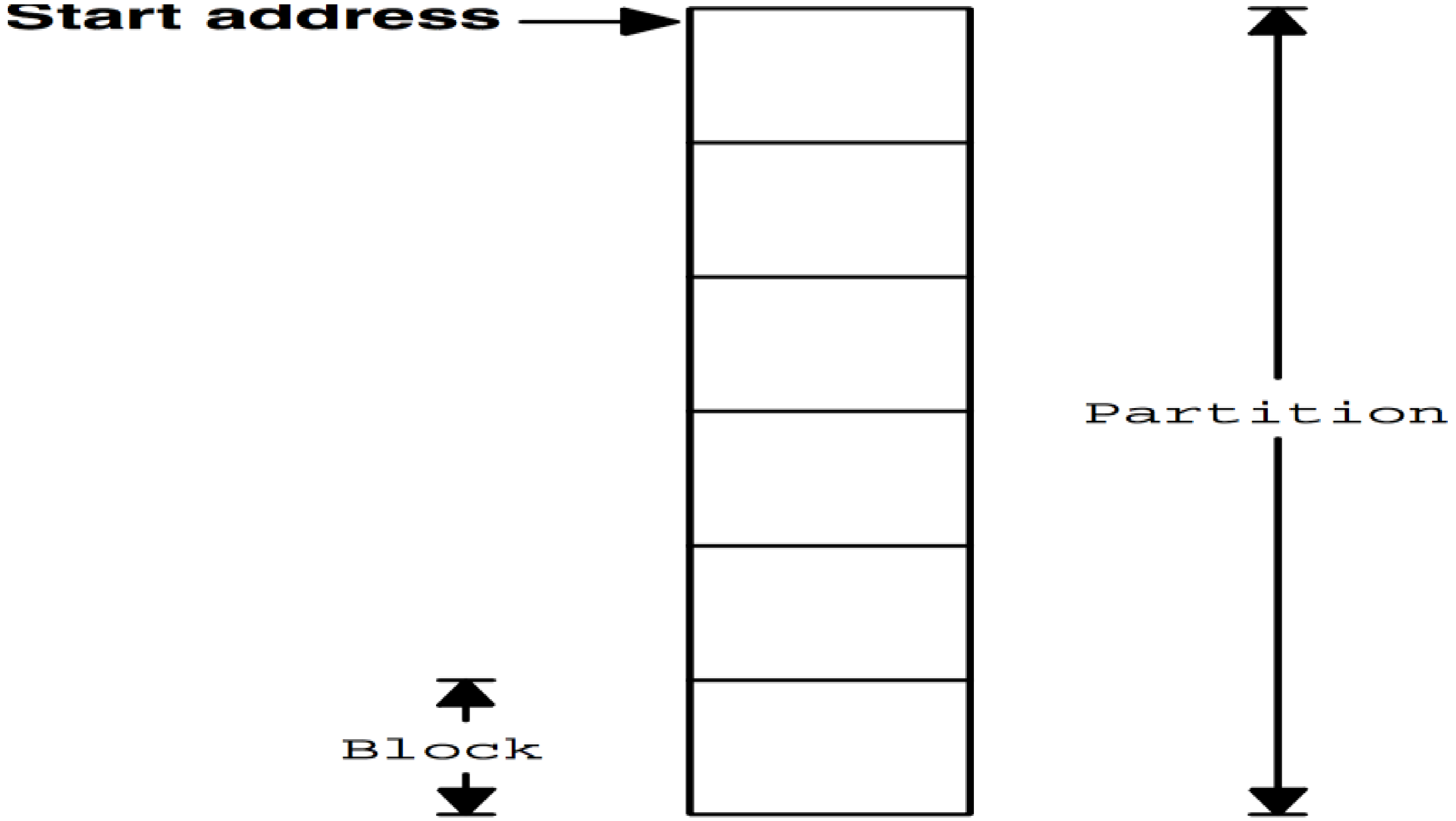
- **unsigned byte OSTaskSuspend (unsigned byte taskPriority)**
- — Called for blocking a task
- Task priority -Priority of the task to be suspended
- This function returns:
- OS_NO_ERR when blocking succeeds
- OS_TASK_SUSPEND_PRIO if priority value that passed already does not exist
- OS_TASK_SUSPEND_IDLE if attempting to suspend an idle task that is illegal
- OS_PRIO_INVALID if the priority value that is passed is more than the given range

- **unsigned byte OSTaskResume (unsigned byte taskPriority)**
- — Called for resuming a blocked task
- Task priority -Priority of the task to be resumed
- This function returns:
- OS_NO_ERR when unblocking succeeds
- OS_TASK_RESUME_PRIO if priority value that passed already does not exist
- OS_TASK_NOT_SUSPENDED if attempting to resume a not suspended task
- OS_PRIO_INVALID if the priority value that is passed is more than the given range
- **Function void OSTimeSet(unsigned int count)**
- — is used for setting the system clock

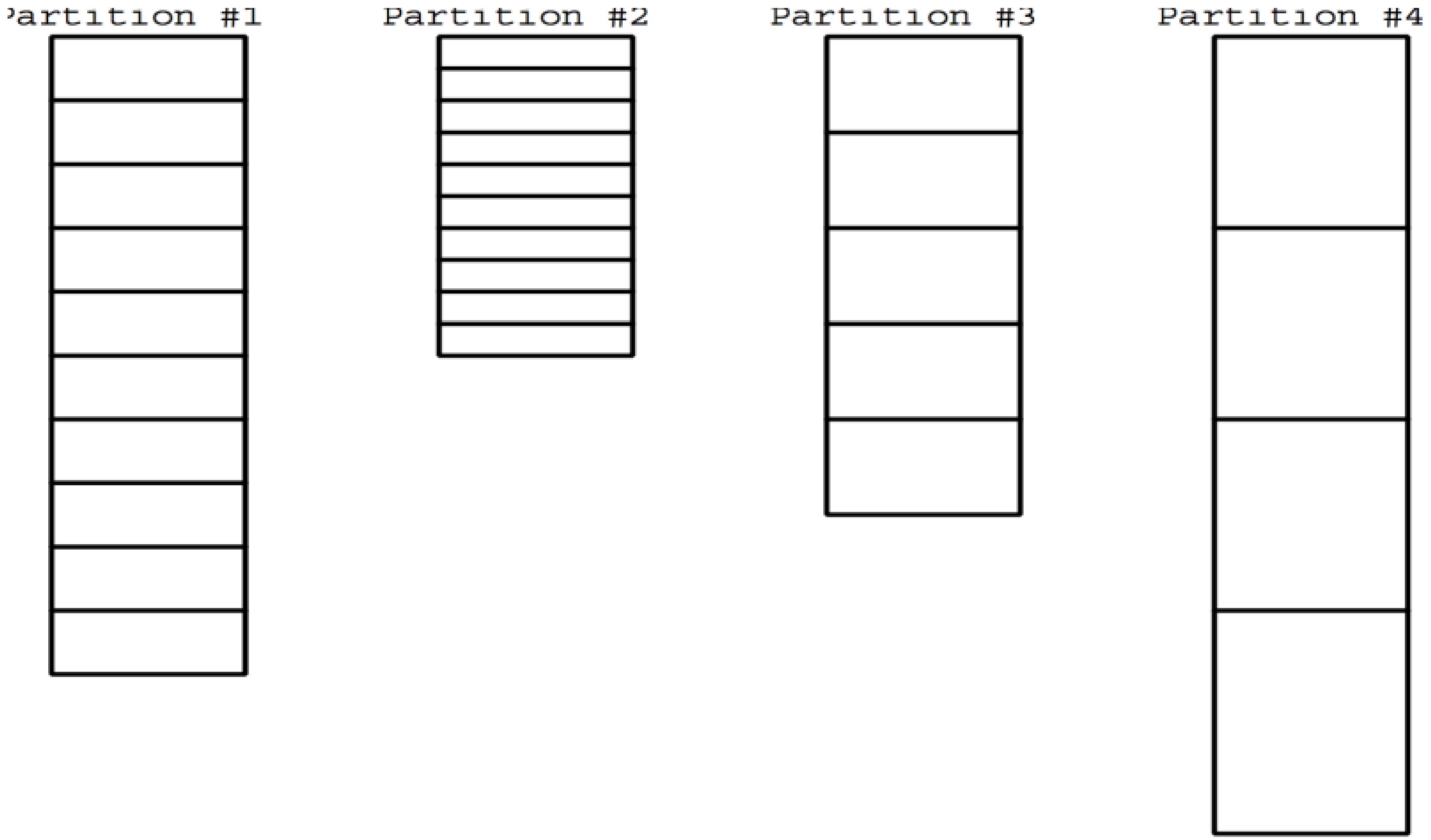
Memory Allocation Related Functions

- Creating Memory blocks at a memory address:
- **OS_MEM *OSMemCreate (void *memaddr, MEMTYPE numBlocks, MEMTYPE blocksize, unsignedbyte *memErr)**
- Is an OS function which partitions memory from an address with partitions in the block
- the beginning address of the memory partition,
- the number of blocks to be allocated from this partition,
- the size (in bytes) of each block
- and a pointer to a variable that contains an error code.

Memory Partition



Multiple Memory Partition



- This function returns:
- a NULLpointer if OSMemCreate() fails
- on success returns a pointer to the allocated memory control block
- MEMTYPE is the data type according to the memory whether 16 bit or 32 bit memory address are there.

- Obtaining a memory block at a memory address
- **void *OSMemGet (OS_MEM *memCBPointer, unsignedbyte *memErr)**
- Is to retrieve a memory block from the partition created earlier.
- This function returns:
- a pointer to the memory control block for the partitions it returns NULL if no block exists there

- Obtaining Status of a memory partition
- **Unsigned byte OSMemQuery (OS_MEM *memCBPointer, OS_MEMDATA *memData)**
- Is to query and return error code and pointers for the memory partition
- is used to obtain information about a memory partition. Specifically, your application can determine how many memory blocks are free, how many memory blocks have been used (i.e., allocated), the size of each memory block (in bytes), etc
- This function returns:
- an error code which is an unsigned byte .the code is OS_NO_ERR=1 when querying succeeds else 0

- Returning a memory block into a partition
- **Unsignedbyte OSMemPut (OS_MEM *memCBPointer, void *memBlock)**
- the address of the memory control block (memCBPointer) to which the memory block belongs (memBlock).
- This function returns:
- OS_NO_ERR when the memory block returned to the memory partition
- OS_MEM_FULL when the memory block cannot be put into the partition as it is full

Semaphore Functions

- Function:
- **OS_Event OSSemCreate (unsigned short semVal)**

  To create and initialize semaphores

  then attempts to obtain an ECB

  If there is an ECB available, the ECB type is set to OS_EVENT_TYPE_SEM
- This function returns:
- returns a pointer to the ECB.
- If there are no more ECBs, OSSemCreate() returns a NULL pointer.

- **void OSSemPend (OS_Event *eventPointer, unsigned short timeOut, unsigned byte *SemErrPointer)**
- Is for letting a task wait till the release event of a semaphore

  To check whether semaphore is pending or not pending (0 or >0).

  If pending (=0), then suspend the task till >0 (released).

  If >0, decrement the value of semaphore and run the waiting codes
- This function returns:
- OS_NO_ERR when the semaphore search succeeds
- OS_TIMEOUT if the semaphore did not release during the ticks defined for the timeout
- OS_ERR_EVENT_TYPE, if *eventPointer is not pointing to the semaphore.

- **unsigned short OSSemAccept (OS_EVENT *eventPointer)**
- then gets the current semaphore count to determine whether the semaphore is available (i.e., a nonzero value).
- The count is decremented only if the semaphore was available
- To check whether semaphore value > 0 and if yes, then retrieve and decrement. Used when there is no need to suspend a task, only decrease it to 0 if value is not already zero

- **unsigned byte OSSemPost (OS_EVENT*eventPointer)**
- then checks to see if any tasks are waiting on the semaphore
- If there are no tasks waiting on the semaphore, the semaphore count simply gets incremented
- Increment makes the semaphore again not pending for the waiting tasks
- This function returns:
- OS_NO_ERR when the semaphore signalling succeeds
- OS_SEM_OVF, when semVal overflows .
- OS_ERR_EVENT_TYPE, if *eventPointer is not pointing to the semaphore

- **unsigned byte OSSemQuery (OS_EVENT *eventPointer, OS_SEM_DATA *SemData)**

  -To get semaphore information

- OSSemQuery() then copies the wait list from the OS_EVENT structure to the OS_SEM_DATA structure.
- Finally, OSSemQuery() copies the current semaphore count from the OS_EVENT structure to the OS_SEM_DATA structure.
- OS_SEM_DATA contains the current semaphore count and the list of tasks waiting on the semaphore.