

Operating Systems

Prepared by Dr. Alaa Mohamed

Module 2

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات

Processes



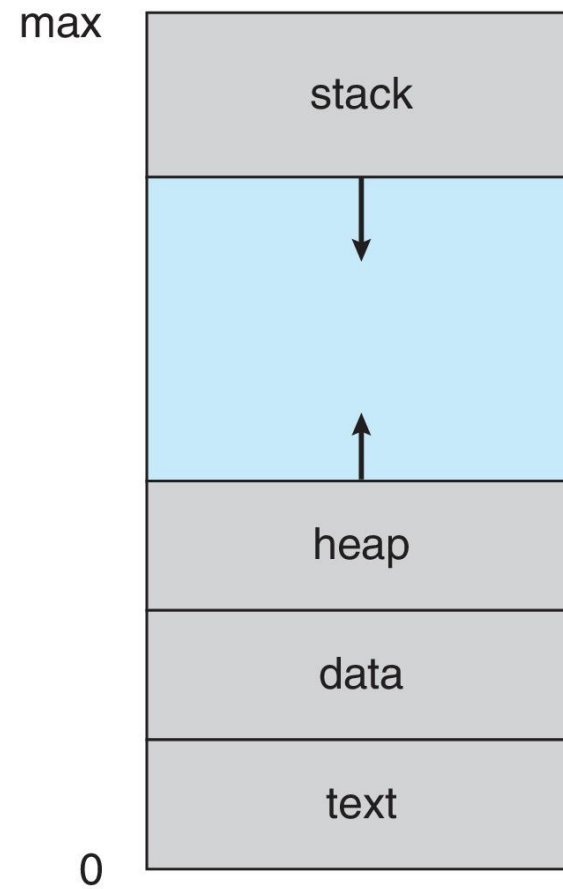
جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

- Objectives:
 - Process Concept
 - Process Scheduling
 - Operations on Processes
 - Interprocess Communication

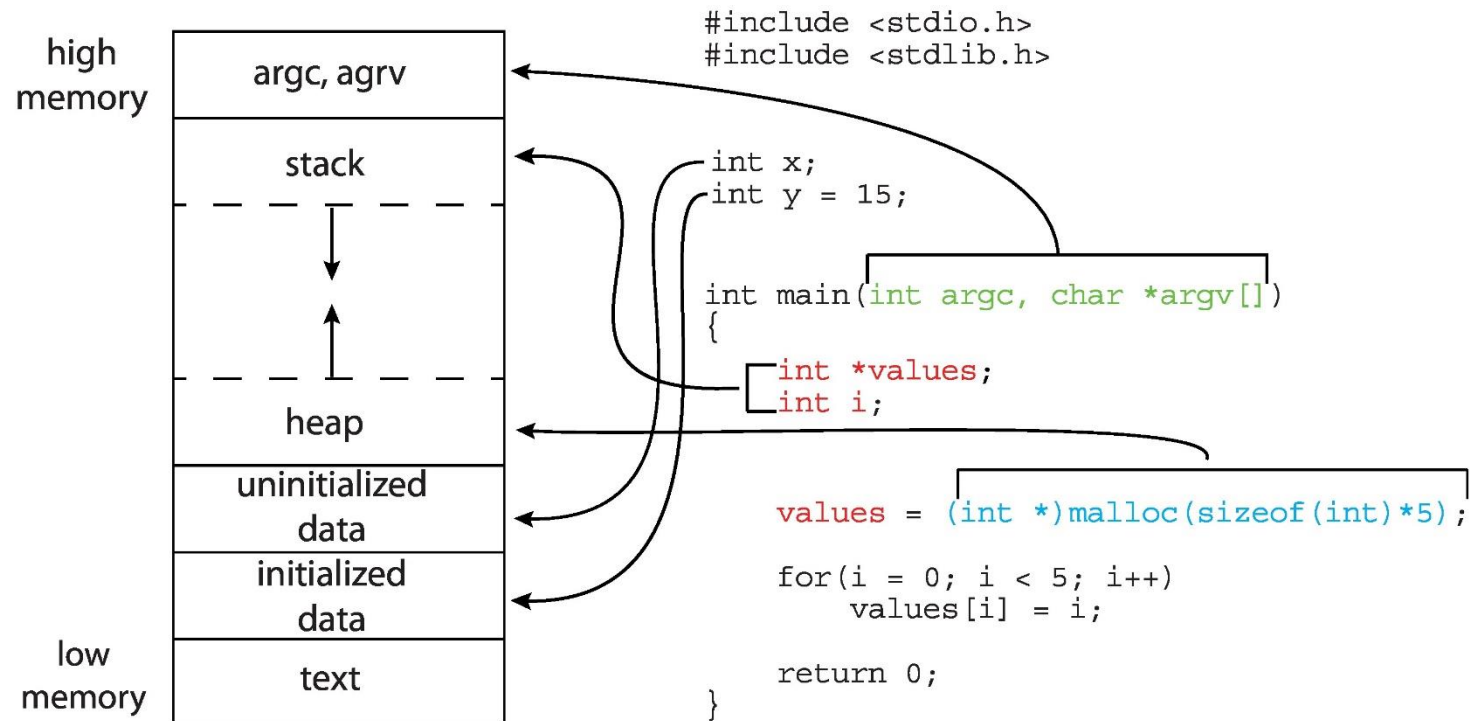
- Topics:
 - Identify the separate components of a process and illustrate how they are represented in an operating system.
 - Describe how processes are created and terminated in an operating system.
 - Describe and contrast interprocess communication using shared memory and message passing.

- **Process** – a program in execution; process execution must progress in a specific order.
- A process refers to an entity that helps in representing the fundamental unit of work that must be implemented in any Operating system.
- In other words, we write the computer programs in the form of a text file, thus when we run them, these turn into processes that complete all of the duties specified in the program.

- A program can be divided into four pieces when put into memory to become a process:
 - The program code, also called **text section**
 - Current activity including **program counter**, processor registers
 - **Stack** containing temporary data
 - Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time

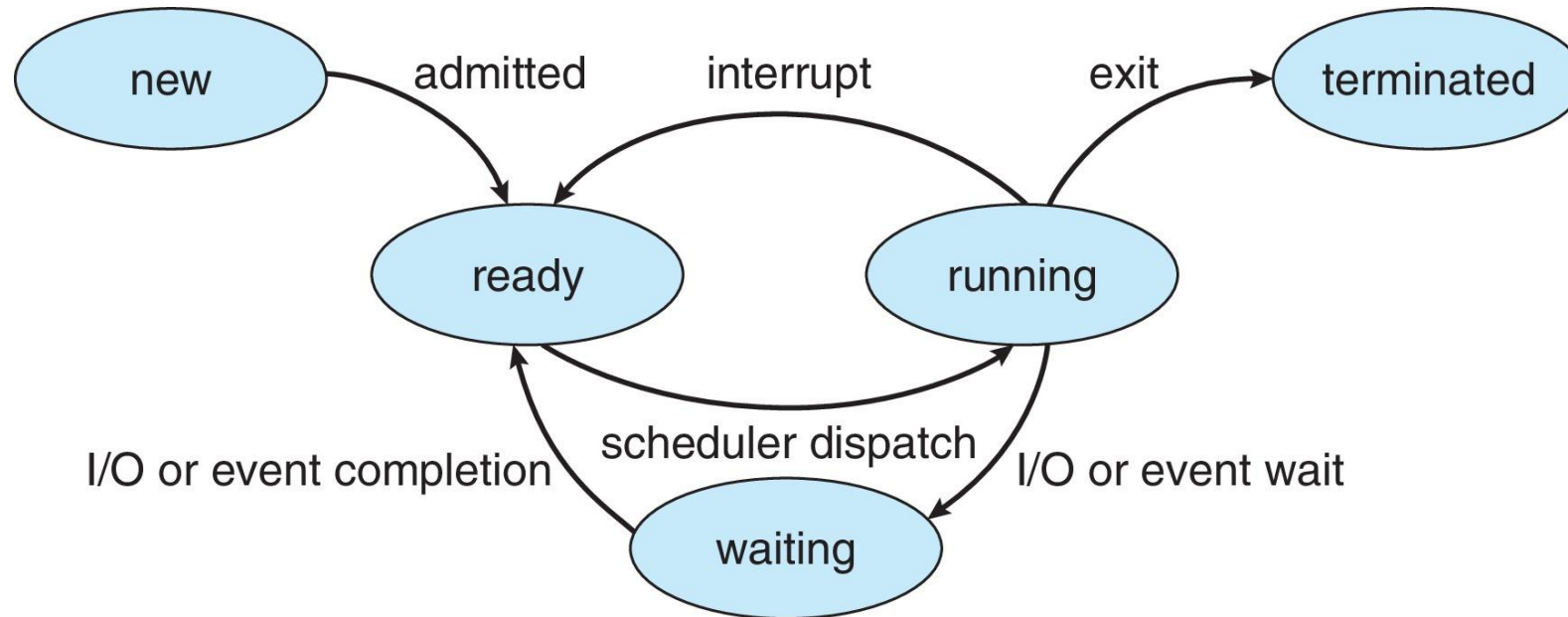


- Program is ***passive*** entity stored on disk (**executable file**); process is ***active***
 - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
 - Consider multiple users executing the same program

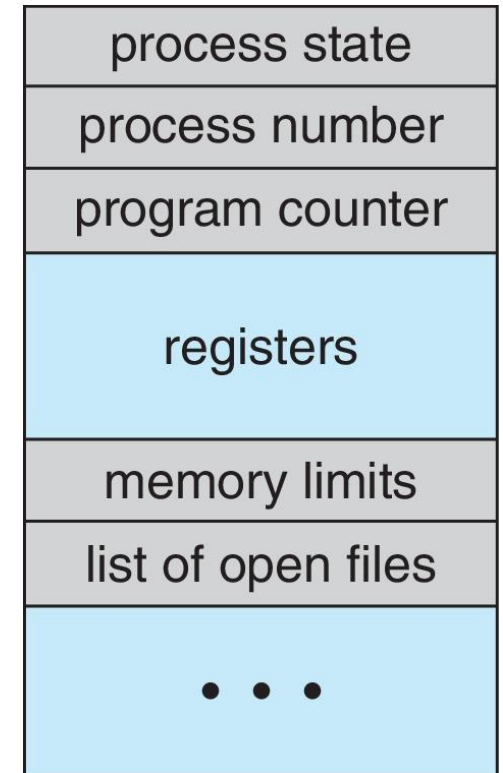


- As a process executes, it changes **state**
 - **New**: The process is being created
 - **Ready**: The process is waiting to be assigned to a processor
 - **Running**: Instructions are being executed
 - **Waiting**: The process is waiting for some event to occur
 - **Terminated**: The process has finished execution

Diagram of Process State



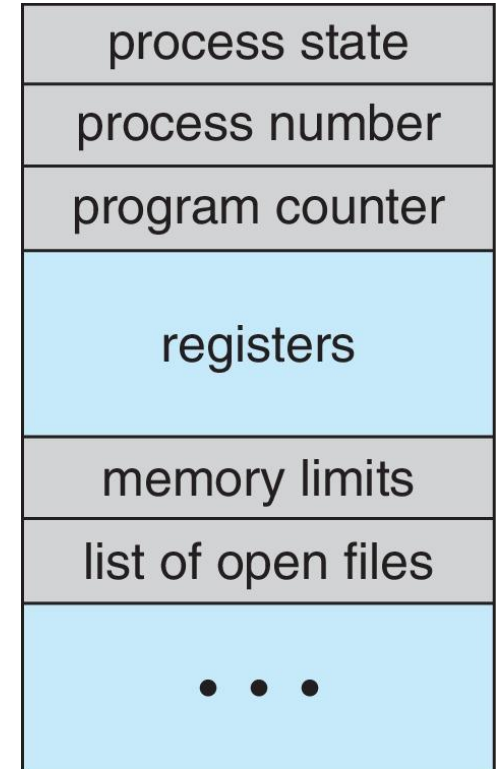
- Every process has a process control block, which is a data structure managed by the operating system.
- PCB stores all of the information required to maintain track of a process.
 - Process state – running, waiting, etc
 - Process Number (ID). Each process in the OS has its own unique identifier.



- Program counter – location of instruction to next execute.
- CPU registers – contents of all process-centric registers.
- CPU scheduling information- priorities.

process state
process number
program counter
registers
memory limits
list of open files
...

- Memory-management information – memory allocated to the process
- Accounting information
- I/O status information – I/O devices allocated to process, list of open files.
- *The PCB architecture is fully dependent on the operating system, and different operating systems may include different information.*



- When there are several or more runnable processes, the operating system chooses which one to run first; this is known as process scheduling.
- A **scheduler** is a program that uses a scheduling algorithm to make choices.

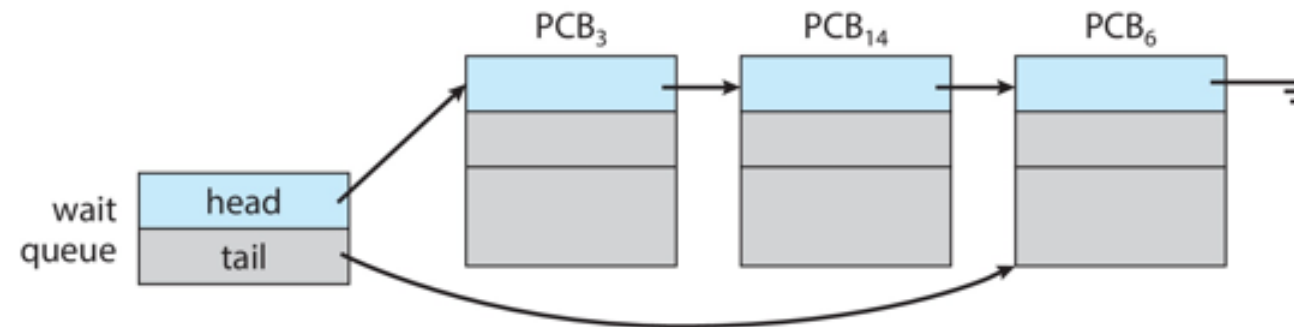
- The following are characteristics of a good scheduling algorithm:
 - For users, **response time** should be kept to minimum.
 - The total **number of jobs processed every hour** should be as high as possible.
 - **Maximize** the CPU use.
 - Each process should be given an **equal amount** of CPU time.

Maintains **scheduling queues** of processes

- All processes, upon entering the system, are stored in the **Job Queue** (set of all processes in the system)
- Processes in the Ready state are placed in the **Ready Queue** (set of all processes residing in main memory, ready and waiting to execute)

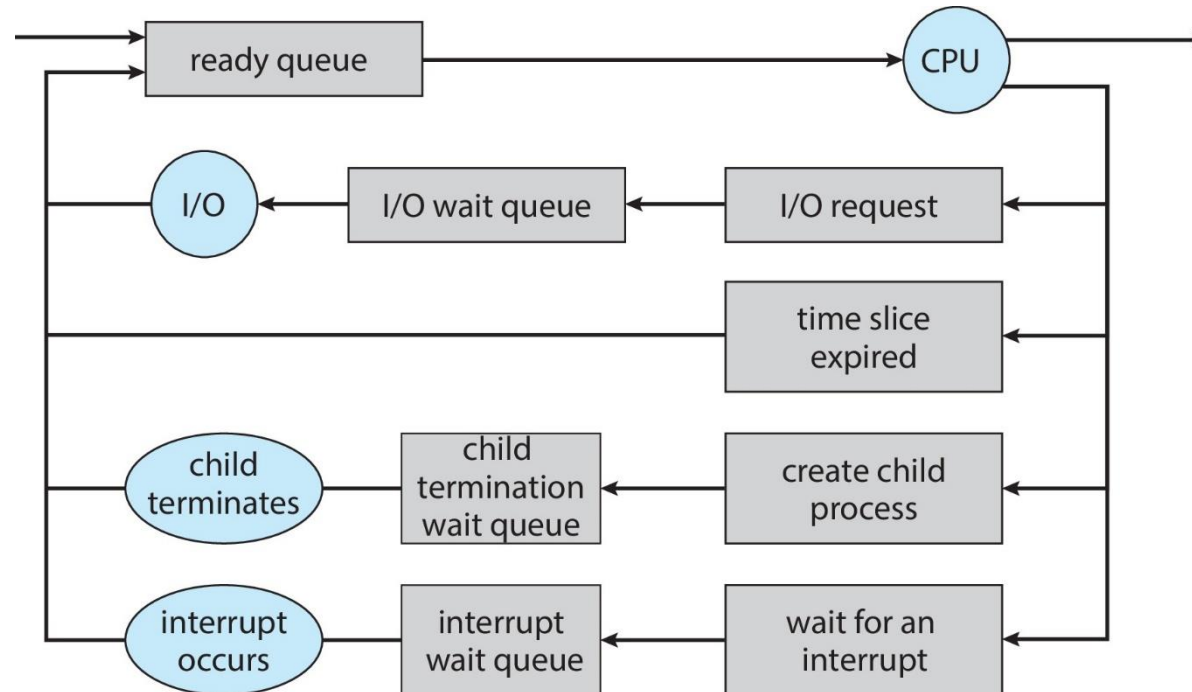
- Processes waiting for a device to become available are placed in **Device Queues**. There are unique device queues available for each I/O device.

Processes migrate among the various queues



A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution(or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

- The process could issue an I/O request, and then be placed in the **I/O queue**.
- The process could create a new subprocess and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.



- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
 - Sometimes the only scheduler in a system
 - Short-term scheduler is invoked frequently (milliseconds)
⇒ (must be fast)

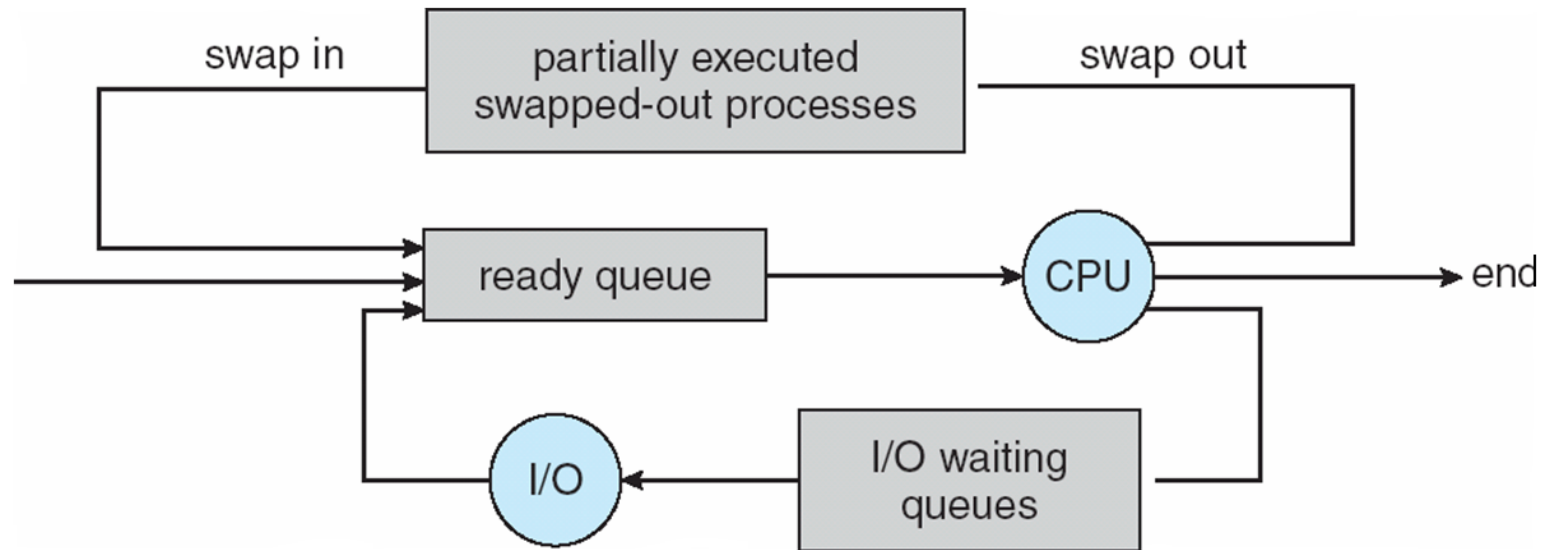
Long-term scheduler (or **job scheduler**) – selects which processes should be brought into the ready queue

Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)

The long-term scheduler controls the **degree of multiprogramming**

- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good *process mix*

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
 - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

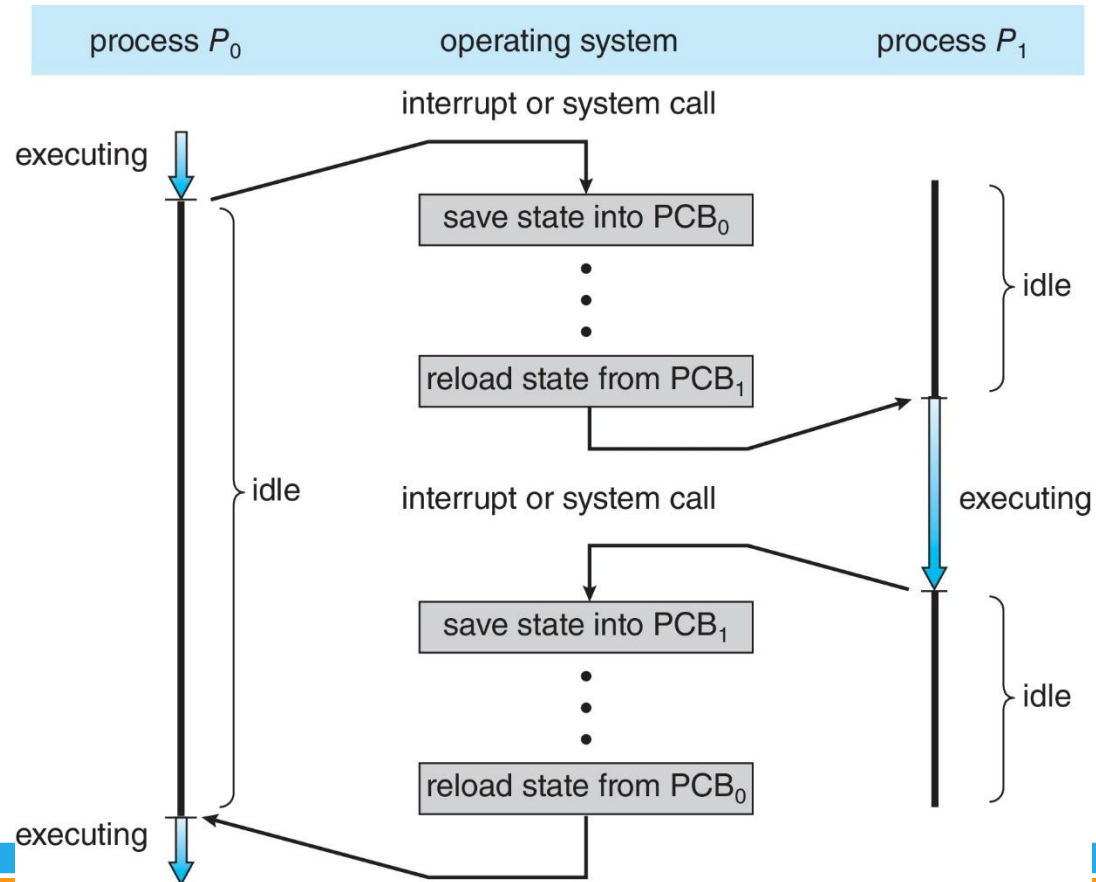


- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB. When a context switch occurs, the Kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

- Context-switch time is overhead; the system does no useful work while switching
 - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

CPU Switch From Process to Process

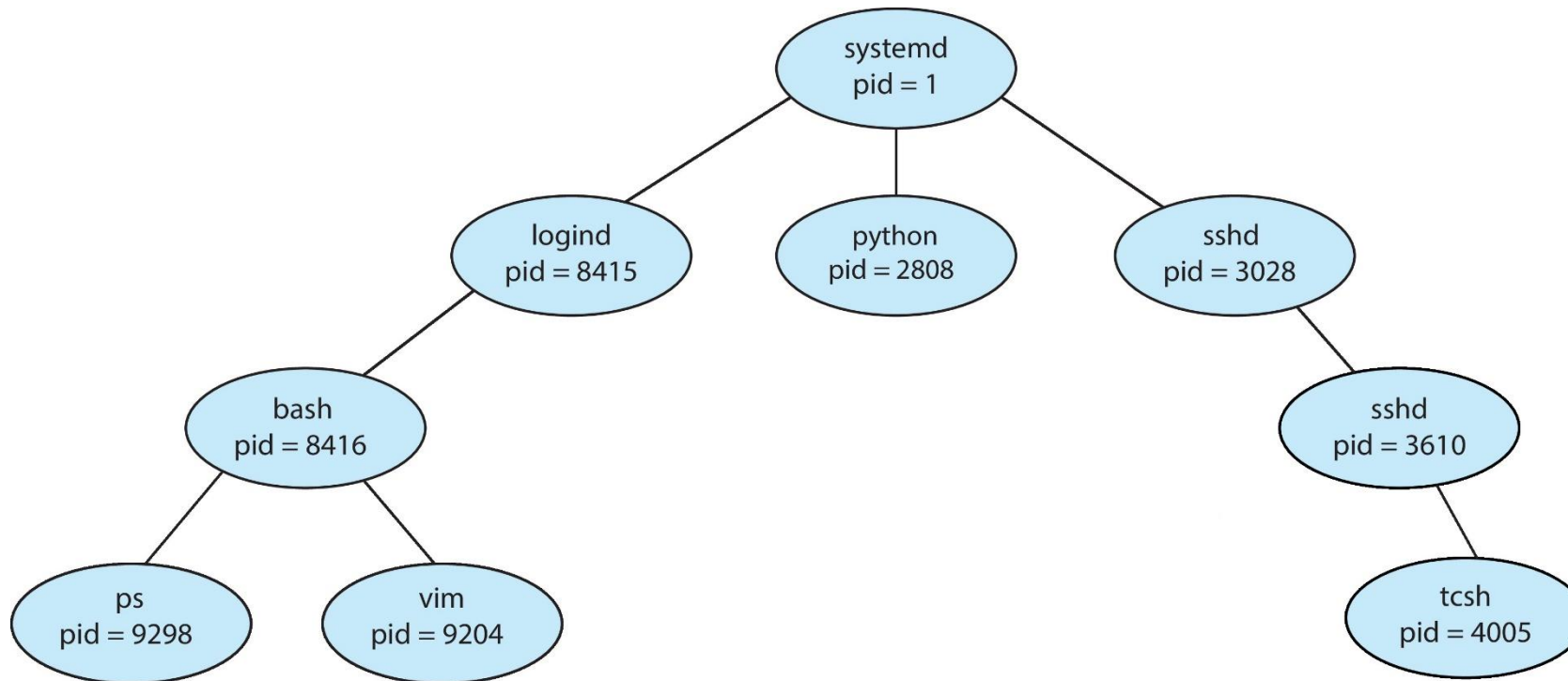
A **context switch** occurs when the CPU switches from one process to another.



- System must provide mechanisms for:
 - process creation
 - process termination

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources

- Execution options
 - Parent and children execute concurrently.
 - Parents Wait for the child process to terminate before proceeding. Parent process makes a `wait()` system call, for either a specific child process or for any particular child process, which causes the parent process to block until the `wait()` returns.



- There are also two possibilities in terms of the address space of the new process:
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork()** system call creates new process
 - **exec()** system call used after a **fork()** to replace the process' memory space with a new program
 - Parent process calls **wait()** for the child to terminate

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
- Process' resources are deallocated by operating system

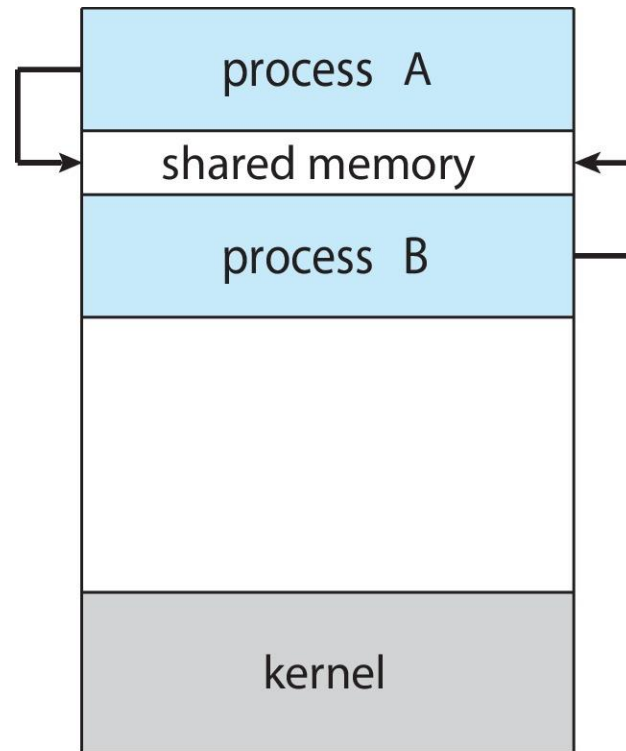
- Parent may terminate the execution of children processes using the **abort ()** system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call. The call returns status information and the pid of the terminated process

- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience

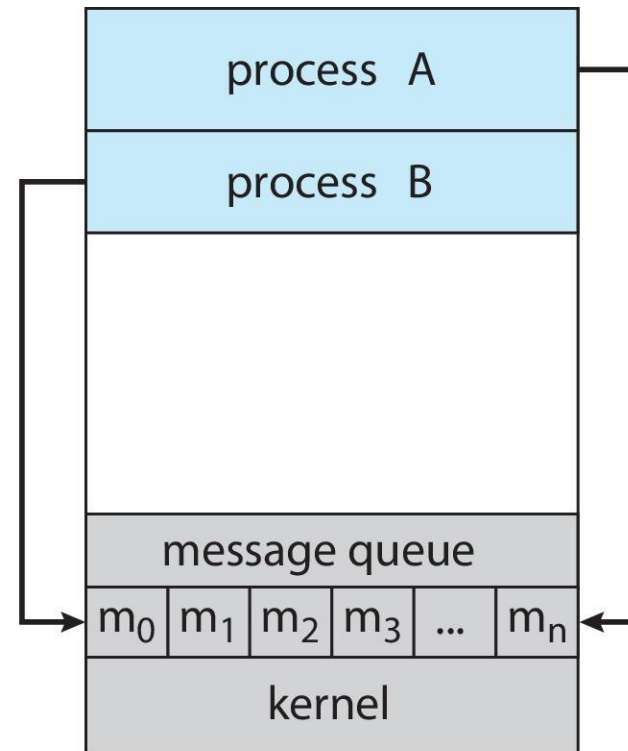
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**

(a) Shared memory.



(a)

(b) Message passing.



(b)

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Module 4

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send**(*message*)
 - **receive**(*message*)
- The *message* size is either fixed or variable

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** -- the sender is blocked until the message is received
 - **Blocking receive** -- the receiver is blocked until a message is available

- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** -- the sender sends the message and continue
 - **Non-blocking receive** -- the receiver receives:
 - A valid message, or
 - Null message

Different combinations possible

If both send and receive are blocking, we have a **rendezvous**

End of Module 2

