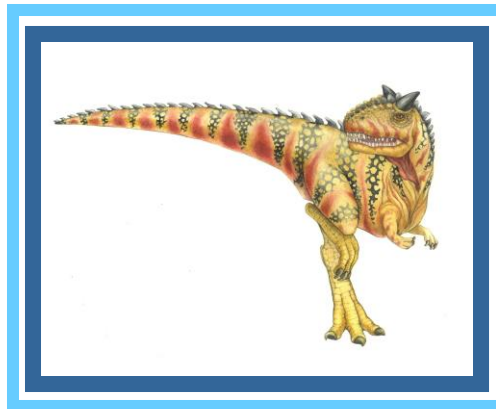


# Chapter 3: Processes

---





# Chapter 3: Processes

---

- Process Concept
- Process Scheduling
- Operations on Processes



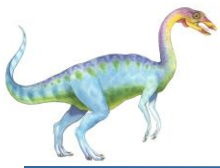


# Process Concept

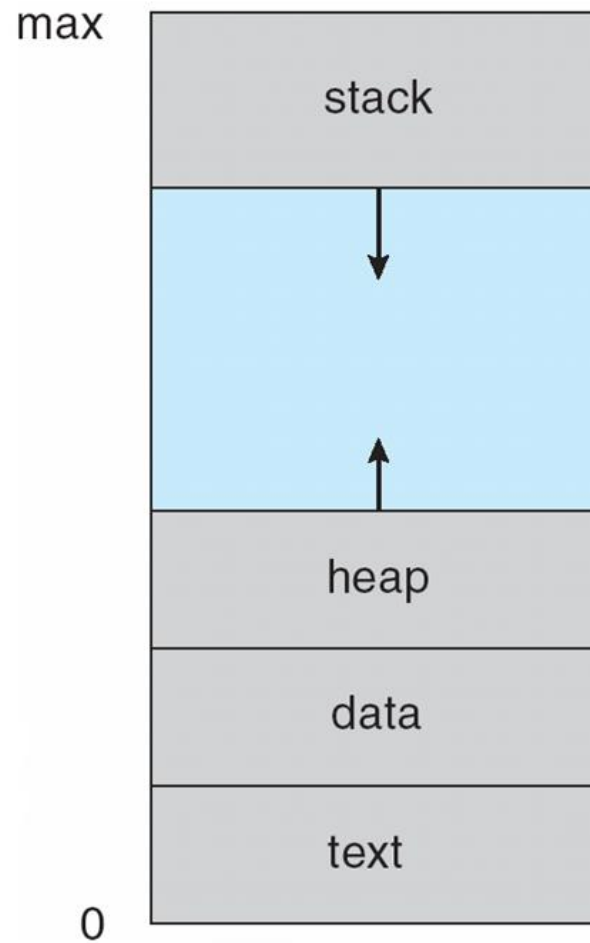
---

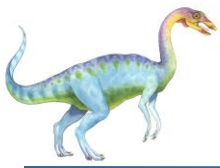
- **Process** – a program in execution; process execution must progress in sequential fashion
- Program is **passive** entity stored on disk (**executable file**), process is **active**
  - Program becomes process when executable file loaded into memory
- Multiple parts
  - The program code, also called **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





# Process in Memory





# Process State

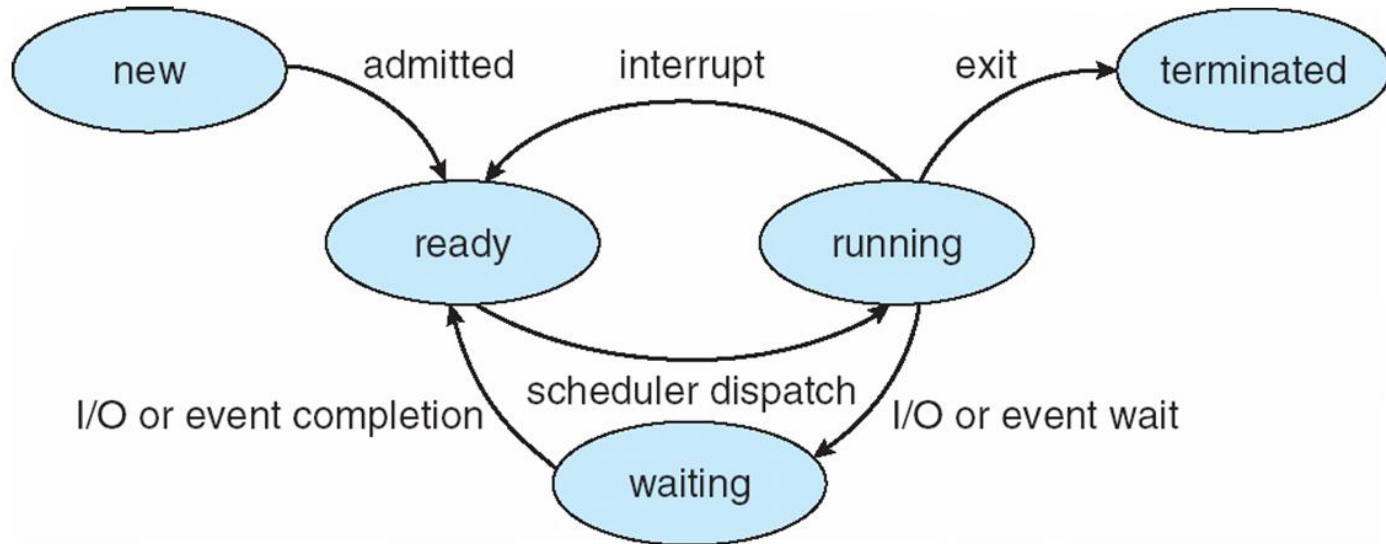
---

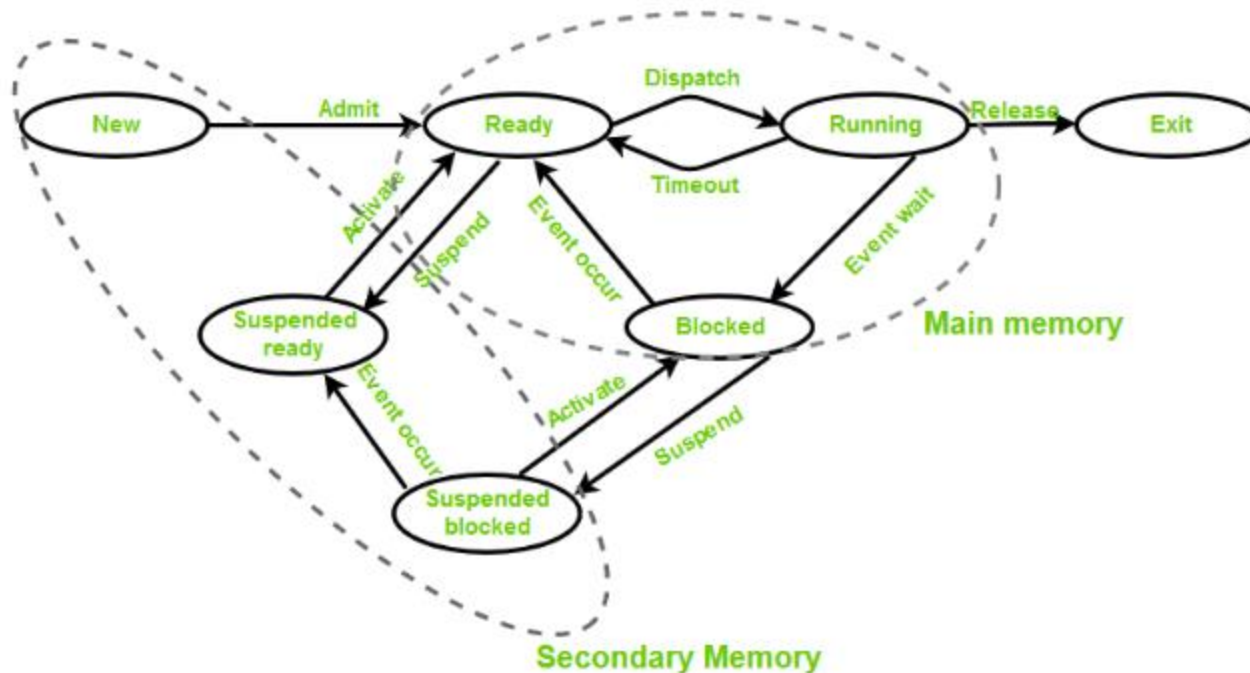
- As a process executes, it changes **state**
  - **new**: The process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked up by OS to create the process.
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution





# Diagram of Process State





•**Suspend ready** – Process that was initially in the ready state but was swapped out of main memory and placed onto external storage by scheduler is said to be in suspend ready state. The process will transition back to 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.

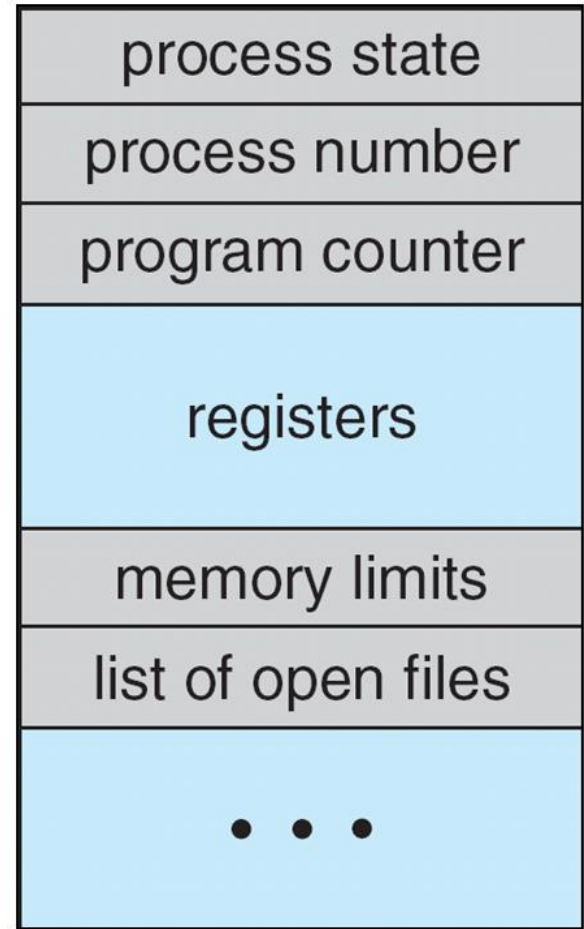




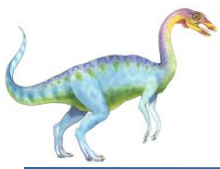
# Process Control Block (PCB)

Information associated with each process  
(also called **task control block**)

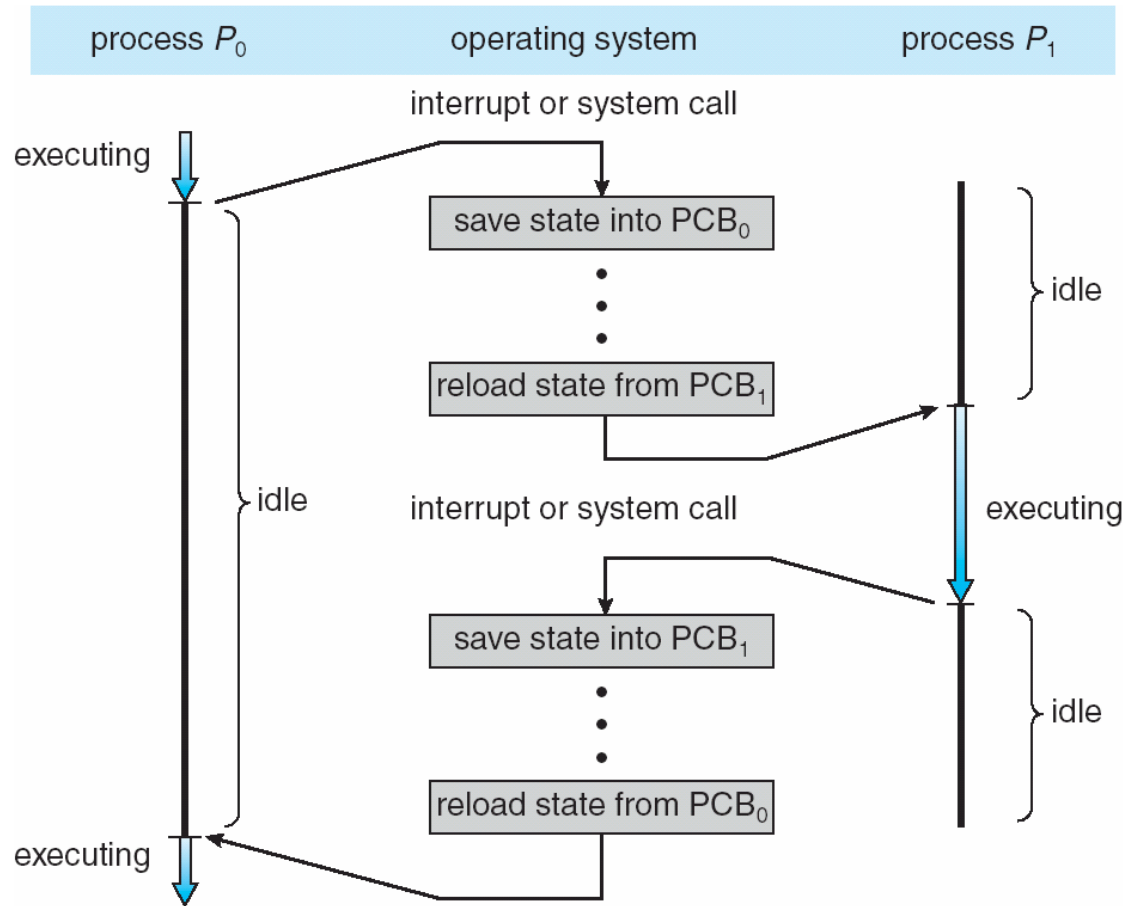
- ❑ Process state – running, waiting, etc
- ❑ Program counter – location of instruction to next execute
- ❑ CPU registers – contents of all process-centric registers
- ❑ CPU scheduling information- priorities, scheduling queue pointers
- ❑ Memory-management information – memory allocated to the process
- ❑ Accounting information – CPU used, clock time elapsed since start, time limits
- ❑ I/O status information – I/O devices allocated to process, list of open files







# CPU Switch From Process to Process





# Process Scheduling

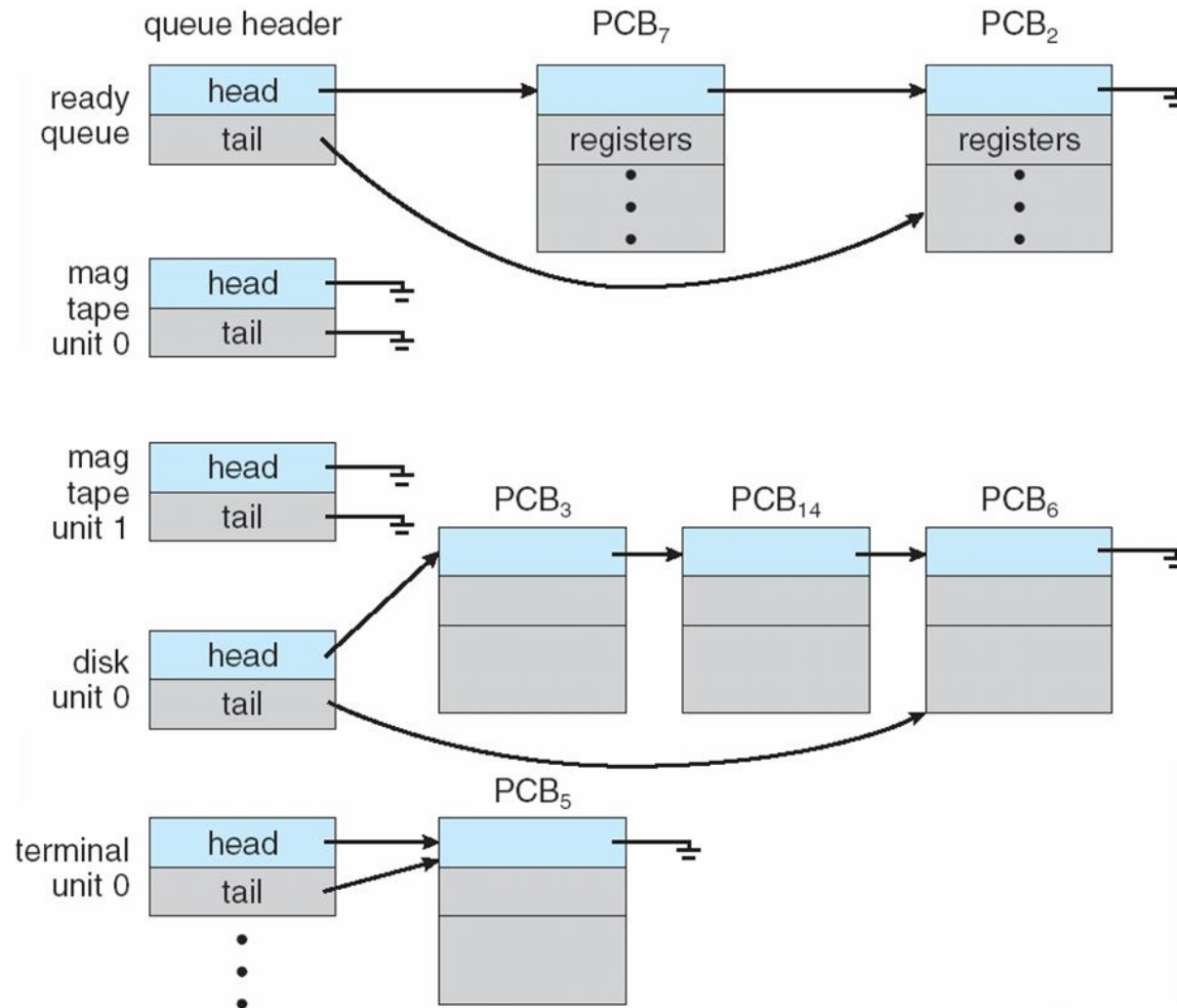
---

- ❑ Maximize CPU use, quickly switch processes onto CPU for time sharing
- ❑ **Process scheduler** selects among available processes for next execution on CPU
- ❑ Maintains **scheduling queues** of processes
  - ❑ **Job queue** – set of all processes in the system
  - ❑ **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - ❑ **Device queues** – set of processes waiting for an I/O device
  - ❑ Processes migrate among the various queues





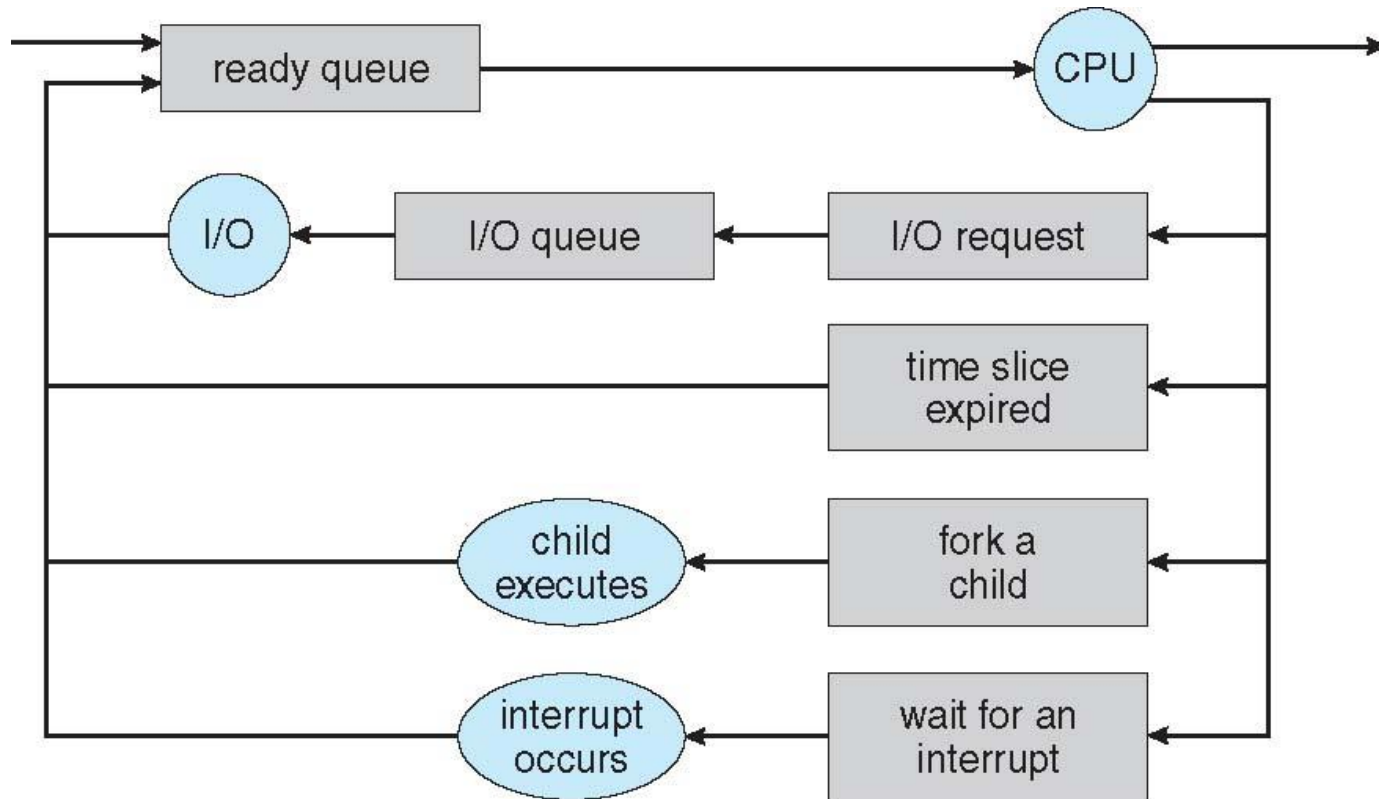
# Ready Queue And Various I/O Device Queues





# Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

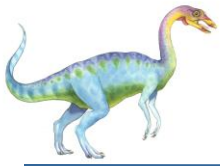




# Schedulers

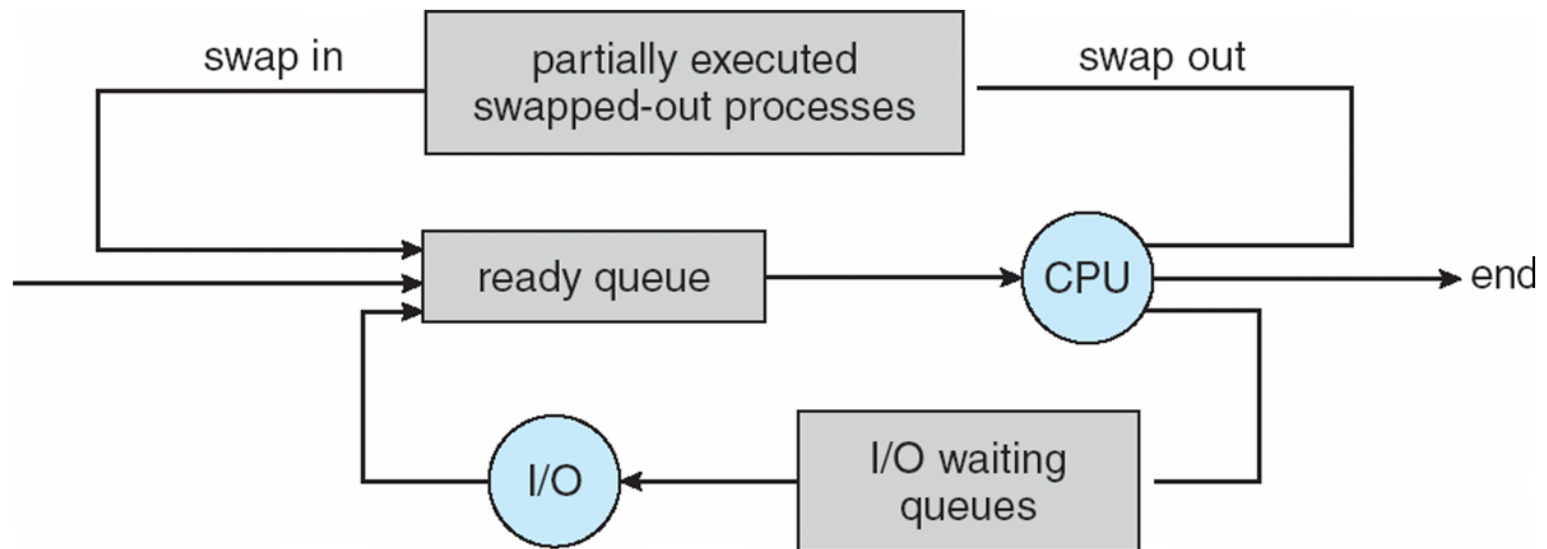
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds)  $\Rightarrow$  (must be fast)
- **Long-term scheduler** (or **job scheduler**)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





# Addition of Medium Term Scheduling

- **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**





Long Term	Short Term	Medium Term
It is a job scheduler.	It is a CPU scheduler.	It is swapping.
Speed is less than short term scheduler.	Speed is very fast.	Speed is in between both
It controls the degree of multiprogramming.	Less control over the degree of multiprogramming.	Reduce the degree of multiprogramming.
Absent or minimal in a time-sharing system.	Minimal in a time-sharing system.	Time-sharing system uses a medium-term scheduler.
It selects processes from the pool and load them into memory for execution.	It selects from among the processes that are ready to execute.	Process can be reintroduced into the meat and its execution can be continued.
Process state is (New to Ready).	Process state is (Ready to Running)	—
Select a good prccess, mix of I/O bound, and CPU bound.	Select a new process for a CPU quite frequently.	—





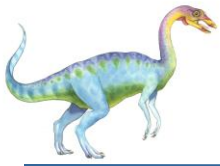
# Context Switch

---

- 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
- Context-switch time is overhead; the system does no useful work while switching







# Operations on Processes

---

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next





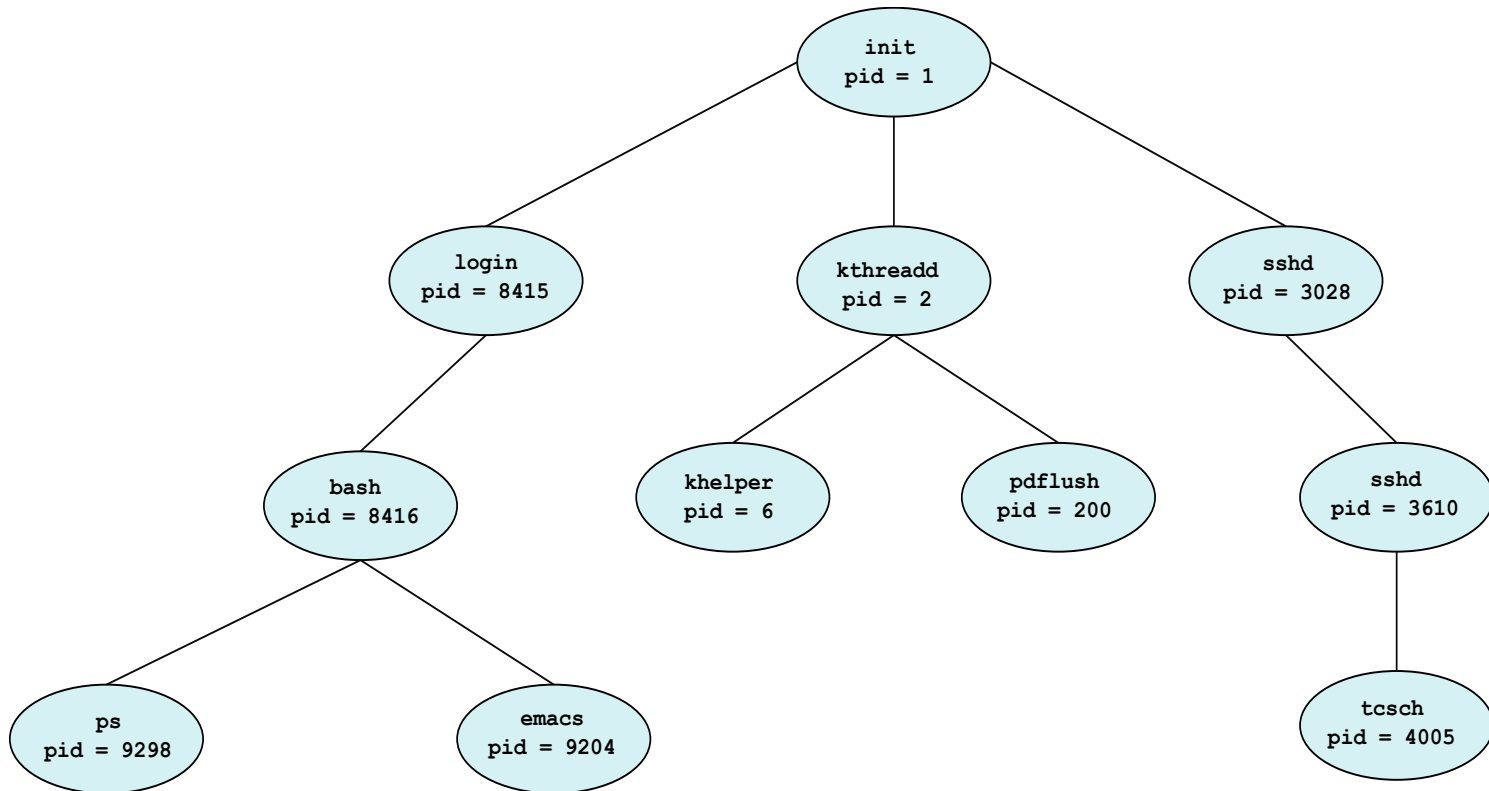
# Process Creation

- ❑ **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
  - ❑ Parent waits until children terminate





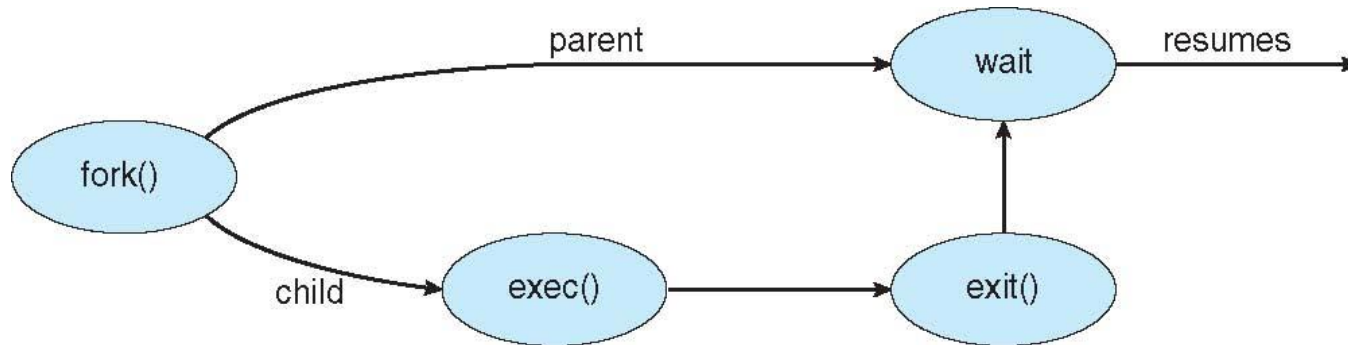
# A Tree of Processes in Linux





# Process Creation (Cont.)

- Address space
  - 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





# C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

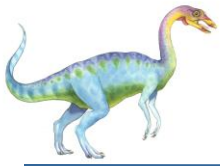
int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```



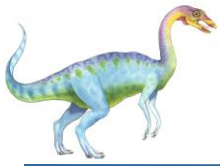


# Process Termination

---

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
  - Returns status data from child to parent (via **wait()**)
  - 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





# Process Termination

- 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

```
pid = wait(&status);
```
- If no parent waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait`, process is an **orphan**



# End of Chapter 3

---

