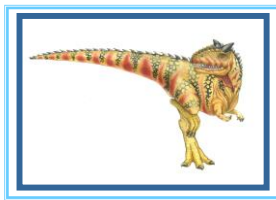


# Processes

---



## Processes

---

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-process Communication
- Examples of IPC Systems
- Communication in Client-Server Systems



## Objectives

---

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To describe communication in **client-server** systems



## Process Concept

---

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms job and process almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section [data + heap]

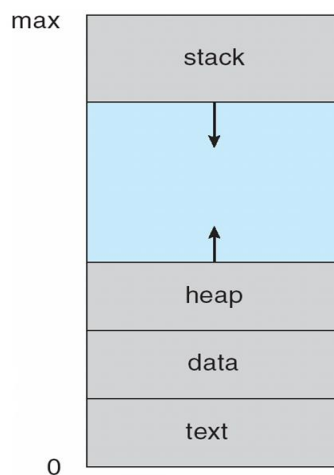


## Process vs. Program

- Program is a **passive** entity
  - It usually found on hard drives or magnetic disks
- Process is an **active** entity
  - The action starts when a program file **loaded into memory**
- Execution of program started via
  - **GUI** event (GUI = Graphic User Interfaces)
  - **Command line** entry of its name (cmd.exe, xterm, putty, ...)
  - **Exec system calls** (exec\*(), CreateProcess, ...)
- One program can start many processes
  - Consider 10 instances of FireFox process (10 tabs)
  - Consider multiple users executing the same program



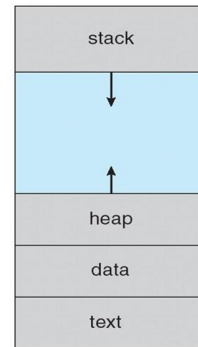
## Process in Memory





# The Process

- **Text section (machine code!!)**
- **program counter, processor registers**
- **Data section**
  - Consists of global and static variables that are initialized by the programmer (like C++ const/global declarations, Java Final...)
  - Does not change at run-time
- **Heap**
  - Dynamic memory, allocated during run time
  - data is freed with delete, delete[], or free()
  - (this is where memory leaks happen ...)
- **Stack containing temporary data**
  - Function arguments
  - Return values (usually pointers to structures on the heap)
  - local variables (C uses the stack to store local variables)



# Example

```
double PI = 3.14159 // data or text?
unsigned int u = 27 // data section
char * str = "No changes allowed"; // data section

int foo()
{
    char *pBuffer; // nothing allocated yet (excluding the pointer itself,
                  // which is allocated here on the stack).
    bool b = true; // Allocated on the stack
    if(b)
    {
        long int x, y, z ; // Create 3 longs on the stack! (local vars)
        char buffer[500]; // Create 500 bytes on the stack! (local var)
        pBuffer = new char[500]; // Create 500 bytes on the heap! (array of char objects)
    } // buffer is deallocated here, pBuffer is not!
} // oops there's a memory leak, should have called:
  // delete[] pBuffer;
```



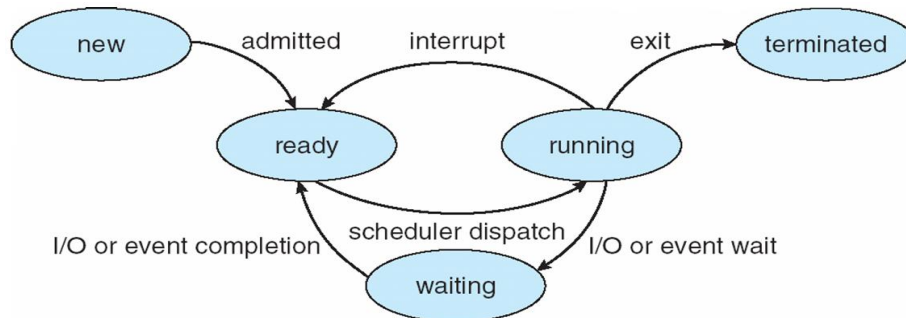
## Process State

During its lifetime, process changes *states*:

- **New**      The process is being created  
The process has been launched and is loaded to memory
- **Ready**      The process is waiting to be assigned to a processor  
It is in memory and ready to run (scheduling)
- **Running**      Instructions are being executed  
CPU control was given to the process and it now the CPU master
- **Waiting**      The process is waiting for some event to occur  
wait for data write, data read, network response, child process to finish work, ...
- **Terminated**      The process has finished execution



## Diagram of Process State





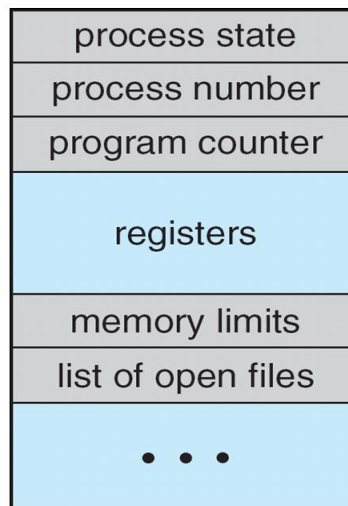
## Process Control Block (PCB)

Data structure holding process information

- **Process state** (ready, waiting, running, ...)
- **Program counter**
- **CPU registers**
- **CPU scheduling information** (priority, queues)
- **Memory-management information** (base, limit)
- **Accounting information** (run times, reads, writes, ...)
- **I/O status information** (open files tables)

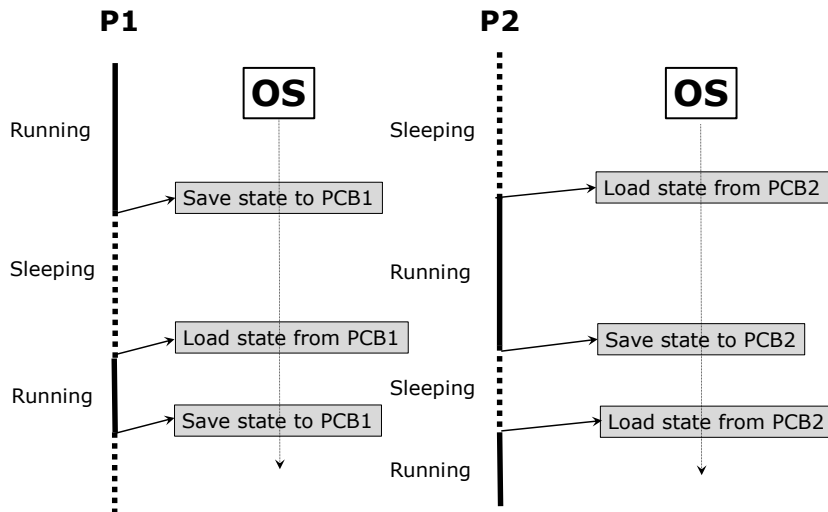


## Process Control Block (PCB)





## CPU Switch From Process to Process



## Process Scheduling

- Maximize CPU usage
- Optimize process time sharing by quick switches
- **Process scheduler** role is to decide 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 (per device)
- Processes migrate among the various queues



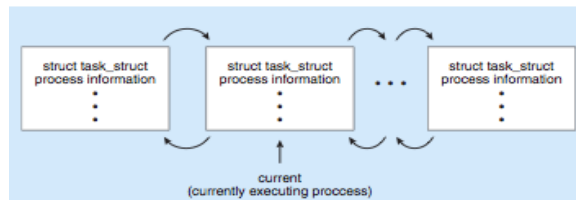
## Process Representation in Linux

- Represented by the C structure `task_struct`

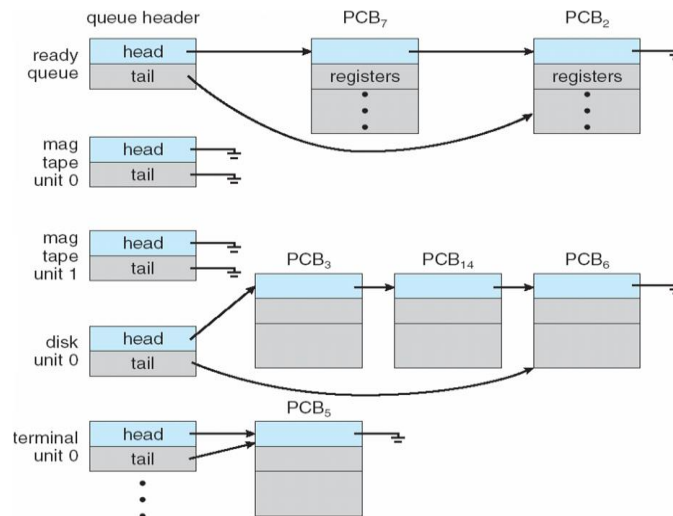
```

pid_t pid;                /* process identifier */
long state;               /* state of the process */
unsigned int time_slice   /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm;     /* memory management info */
struct task_struct *p_opptr, *p_pptr, *p_cptra, *p_ysptr, *p_osptr;
/* op=original parent, p=parent, c=youngest child, ys=youngest siebling,
   os=older siebling */

```



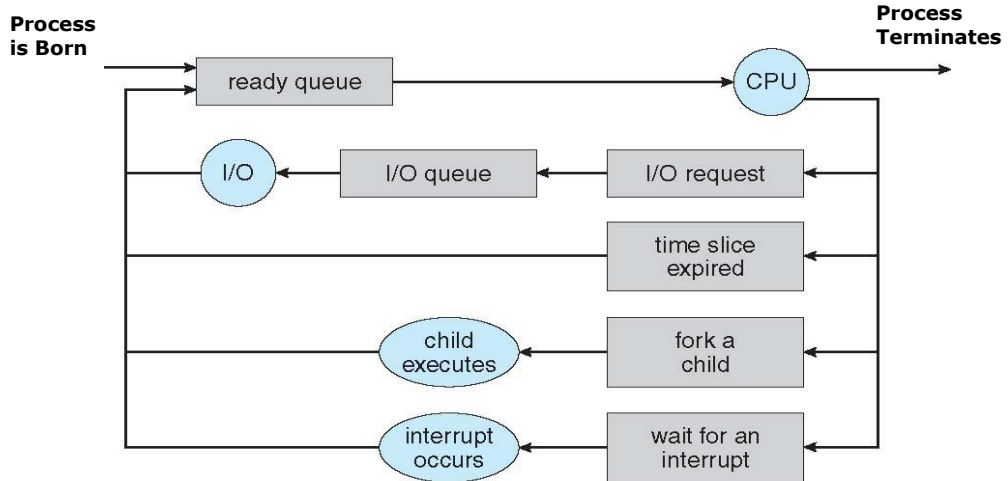
## Ready Queue And Various I/O Device Queues







## Representation of Process Scheduling

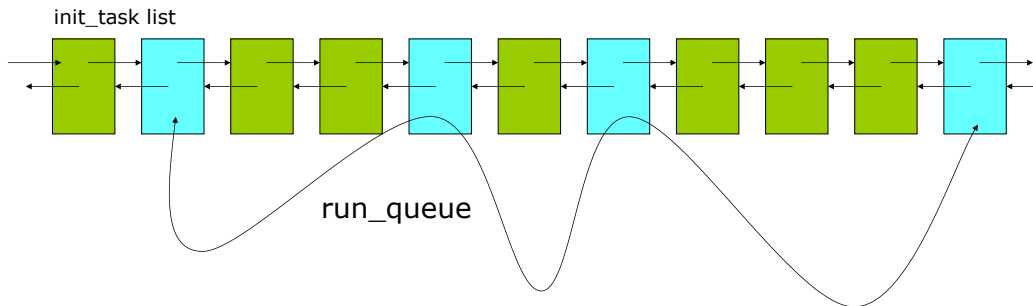


## Schedulers

- **Long-term scheduler** (or job scheduler)
  - Selects which processes should be brought into the ready queue
  - Selects which processes be swapped to disk
- **Short-term scheduler** (or CPU scheduler)
  - selects which process will run next
  - Sometimes the only scheduler in a system



## Some tasks are 'ready-to-run'



Those tasks that are **ready-to-run** comprise a sub-list of all the tasks, and they are arranged on a queue known as the '**run-queue**'

Those tasks that are **blocked** while awaiting a specific event to occur are put on alternative sub-lists, called '**wait queues**', associated with the particular event(s) that will allow a blocked task to be unblocked



## Schedulers (Cont.)

- Short-term scheduler is invoked very frequently
  - Typically 15-60 milliseconds
  - Must be fast!
- Long-term scheduler is invoked very infrequently
  - Seconds, minutes, or hours
  - Could be slow (disk swap is very slow ...)
- Processes that run for days, or even sleep for days but hold large memory segments. The long-term scheduler may swap them to disk
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts and long I/O bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts



## Communications in Client-Server Systems

---

- Pipes
- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)



## Sockets

---

- A **socket** is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets



## Pipes

- Acts as a conduit allowing two processes to communicate
- **Issues**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes? (usable between processes with a common ancestor)
  - Can the pipes be used over a network?



## Simplex, half-duplex, full-duplex

- A simplex line permits data to flow only in one direction
- doesn't support switching direction!
- A half duplex line can alternately send or receive data but only one at a time
- A full duplex line can send and receive data simultaneously

- <http://stackoverflow.com/questions/3557327/what-is-the-difference-between-full-duplex-half-duplex-and-simplex-tcp-ip-operations>

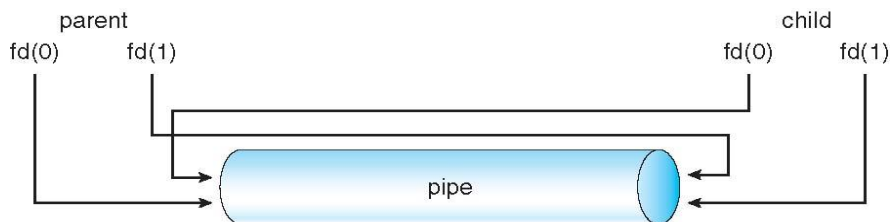


## Ordinary Pipes

- **Ordinary Pipes** allow communication in standard **producer-consumer style**
- **Producer** writes to one end (the *write-end* of the pipe)
- **Consumer** reads from the other end (the *read-end* of the pipe)
- Ordinary pipes are therefore **unidirectional**
- Require parent-child relationship between communicating processes



## Ordinary Pipes





## Named Pipes (FIFO)

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems



## Socket Communication

