Processes

Dr. Tarunpreet Bhatia
Assistant Professor
CSED, TU

Disclaimer

This is NOT A COPYRIGHT MATERIAL

***Content has been taken mainly from the following books**:*

Operating Systems Concepts By Silberschatz & Galvin,
Operating Systems: Internals and Design Principles By William Stallings

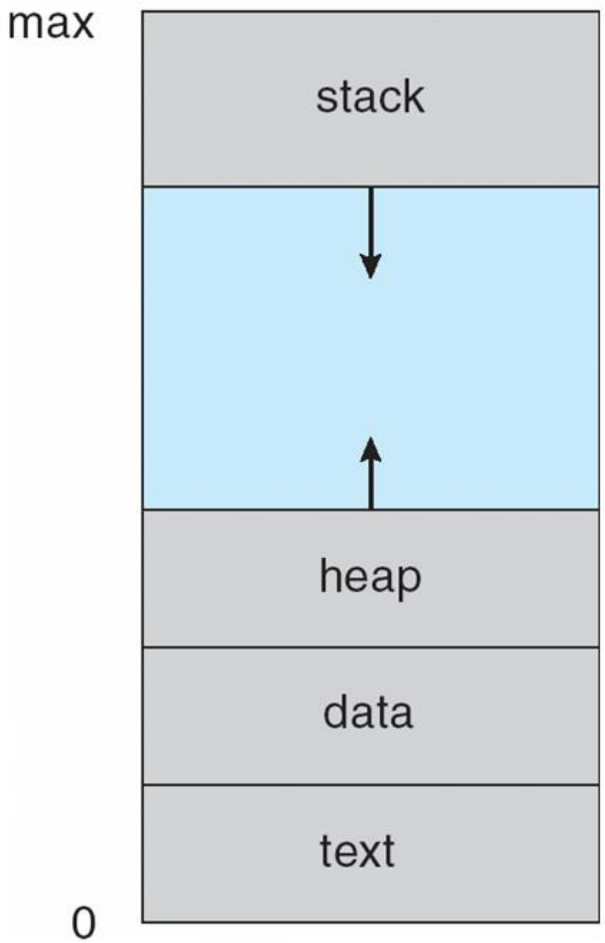
Topics to be discussed

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

Process

- A Program in Execution is called as a *PROCESS*.
- An operating system executes a variety of programs:
  - Batch System – Jobs
  - Time-Shared Systems – User Programs or Tasks
- A Process Includes:
  - Program Counter
  - Stack
  - Data Section

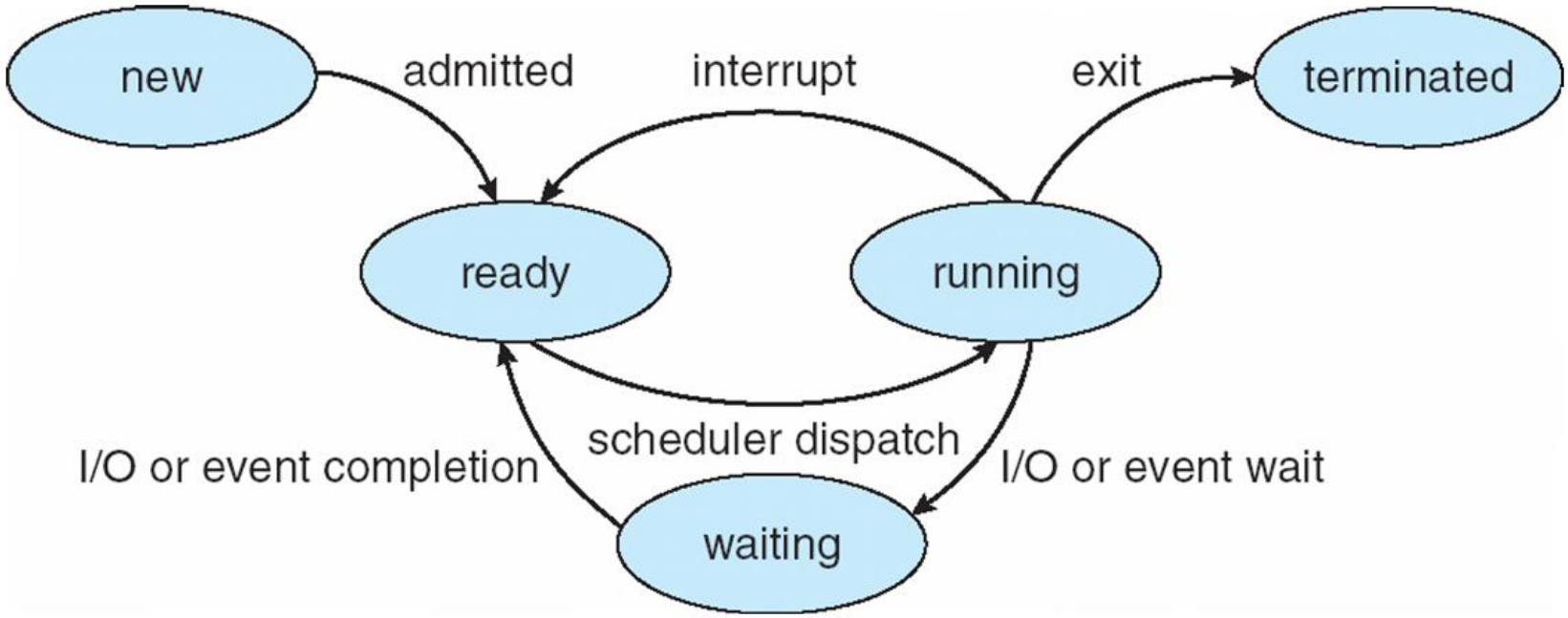
Process in Memory



Process

- A *Program* is a Passive Entity and a *Process* is an Active Entity.
- Process associates PC and a set of other resources with it.
- A Program becomes a Process when an executable file is loaded into memory.
- Two Techniques are present for loading Executable Files.
  (Double Clicking EXE, Running EXE through Command Prompt)

Process States



Process States

New State: The Process being created.

Ready State: The Process is waiting to be assigned to a Processor.

Running State: A Process is said to be running if it has the CPU.

Blocked (or Waiting) State: A Process is said to be blocked if it is waiting for some event to happen.

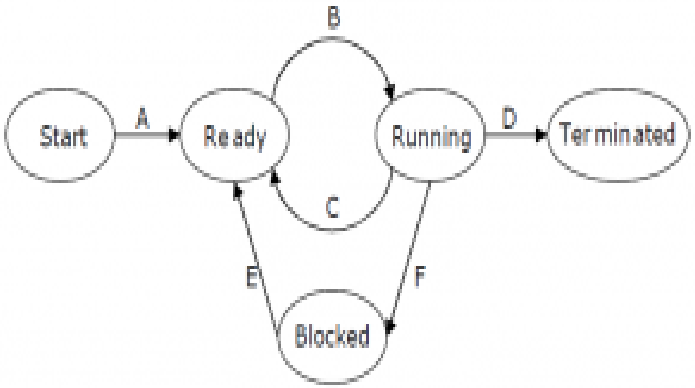
Terminated State: The Process has finished *Execution*.

Comparison

	PREEMPTIVE SCHEDULING	NON PREEMPTIVE SCHEDULING
Basic	The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Interrupt	Process can be interrupted in between.	Process can not be interrupted till it terminates or switches to waiting state.
Starvation	If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Overhead	Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Flexibility	Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Cost	Preemptive scheduling is cost associated.	Non-preemptive scheduling is not cost associative.

Question

In the following process state transition diagram for a uniprocessor system, assume that there are always some processes in the ready state: Now consider the following statements:



I. If a process makes a transition D, it would result in another process making transition A immediately.

II. A process P2 in blocked state can make transition E while another process P1 is in running state.

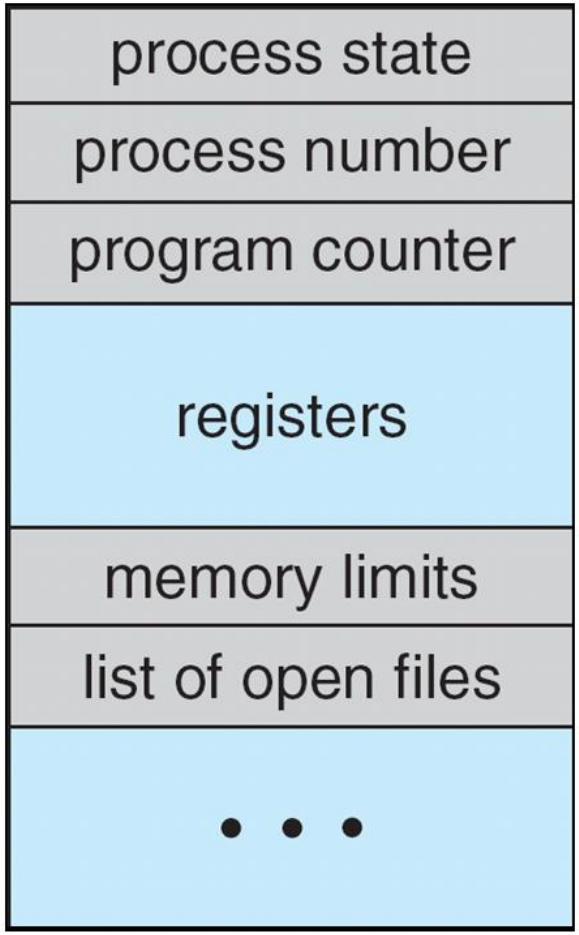
III. In RR scheduling, a process in running state makes a transition F when its quantum expires (Assume process is not requesting I/O device).

Which of the above statements are FALSE?

Process Control Block (PCB)

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

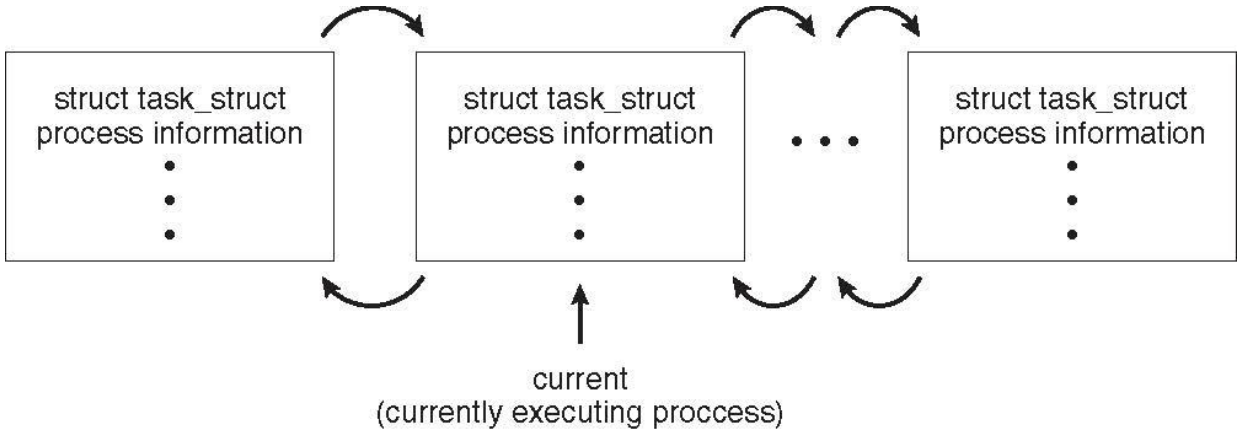
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting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



Process Representation in Linux

Represented by the C structure `task_struct`

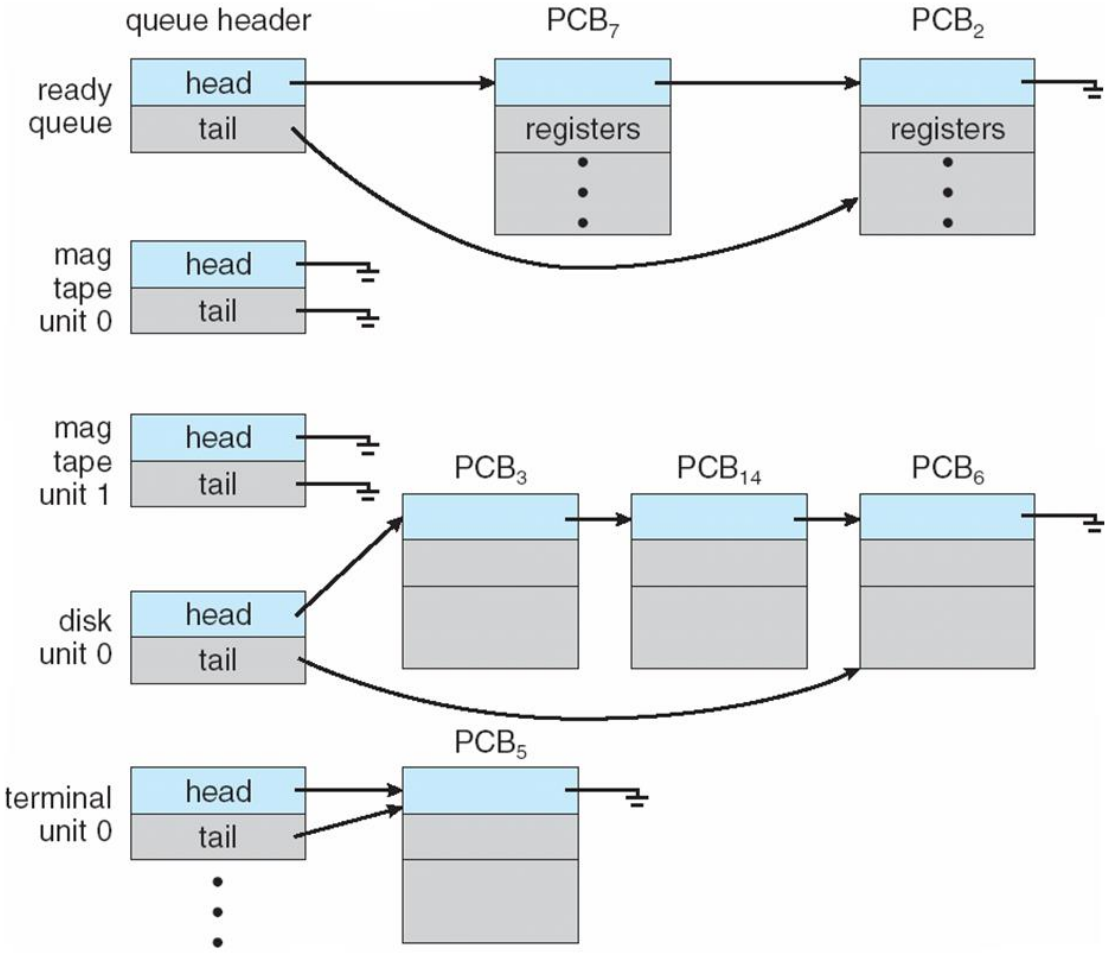
```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



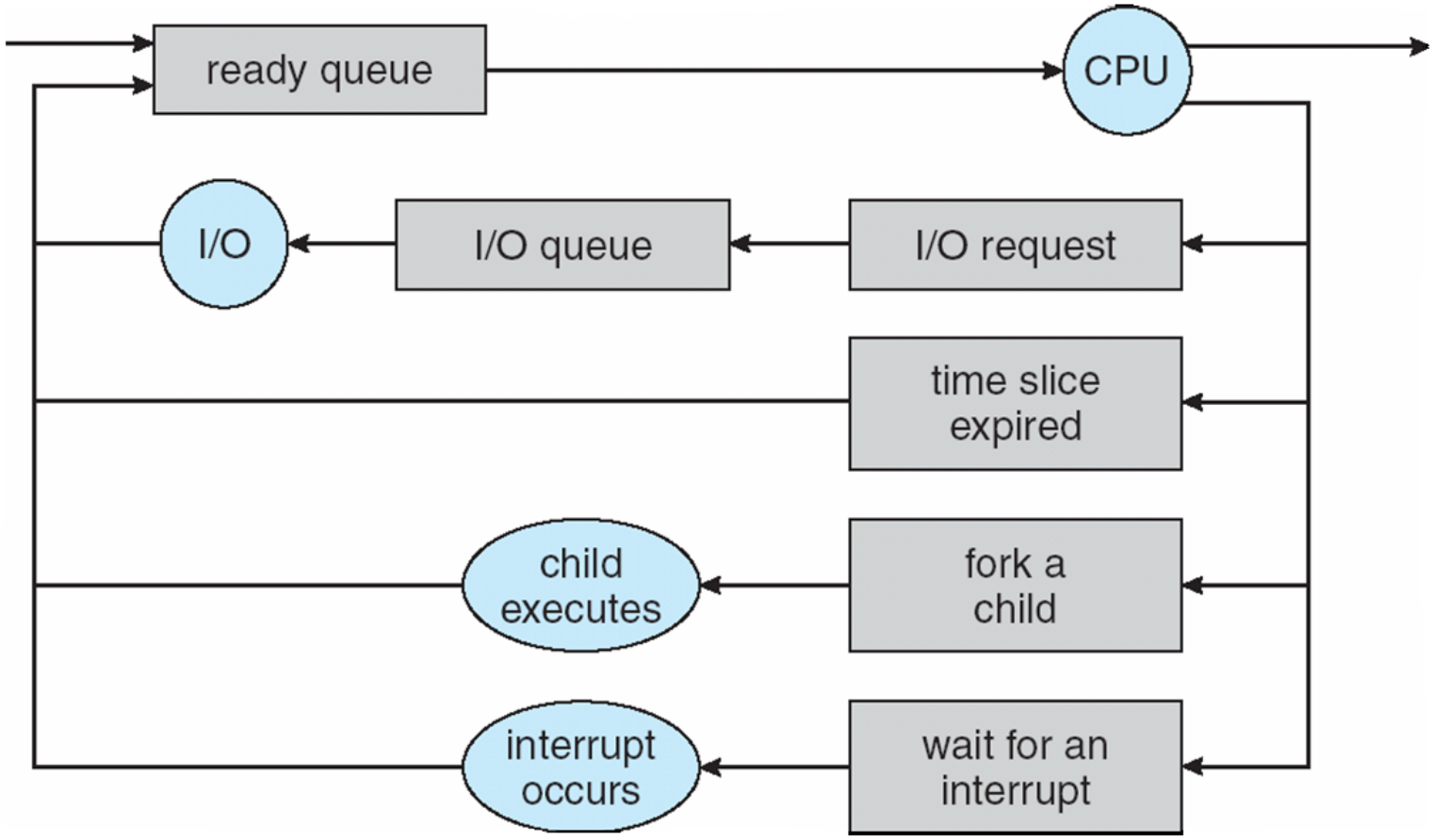
Various Queues

- 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 I/O device Queues



Process Scheduling Diagram



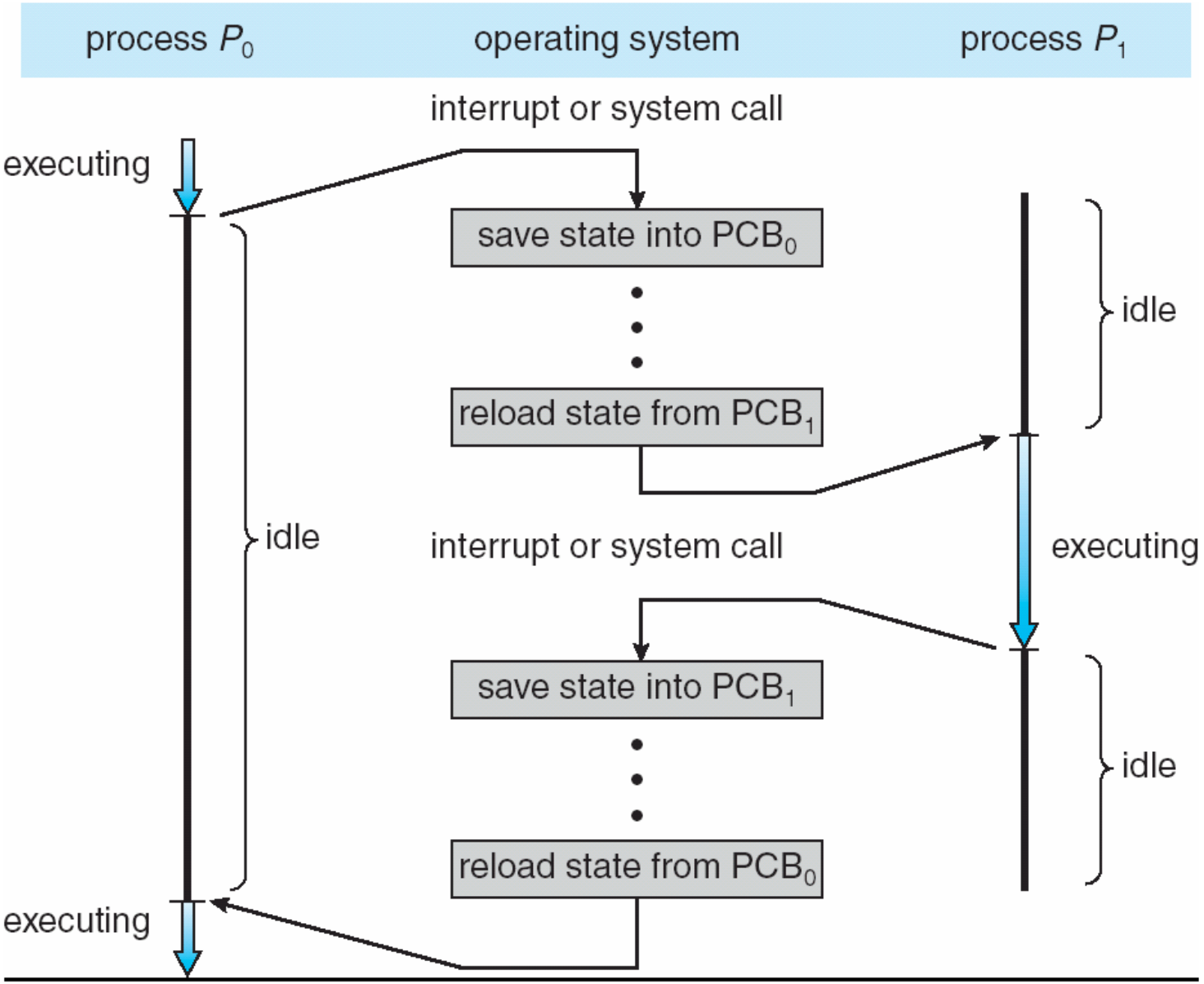
Context Switch

- 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.

- Time dependent on hardware support


process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
reload state from PCB_1
idle
idle
interrupt or system call
executing
save state into PCB_1
reload state from PCB_0
idle
executing

Question

Consider a process PCB of size 100kb. The context load rate is 1 mbps with an extra latency of 8 ms. What should be an acceptable time quantum for effective CPU utilization?

A) 0.216 sec B) 2.048 sec

C) 0.108 sec D) 0.124 sec

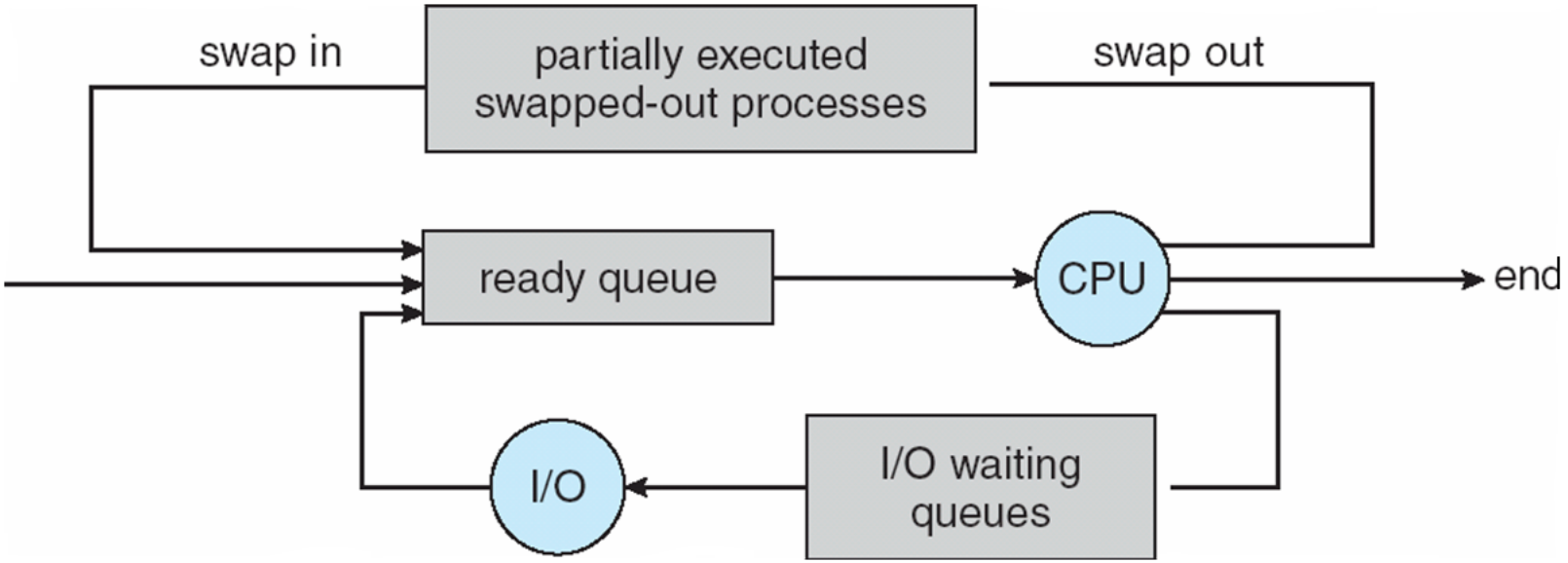
LSM Term Scheduler

- Long-Term Scheduler (or Job scheduler) – Selects which Processes should be brought into the Ready queue

- Short-Term Scheduler (or CPU scheduler) – Selects which Process should be executed next and allocates CPU

- Medium-Term Scheduler – Intermediate Level of Scheduling. Limit the Multiprogramming and executes Swapping.

Medium Term Scheduler



Comparison among scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Parent & Child Process

- **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
  - Parent and Children share all Resources
  - Children share subset of parent's Resources
  - Parent and child share no Resources

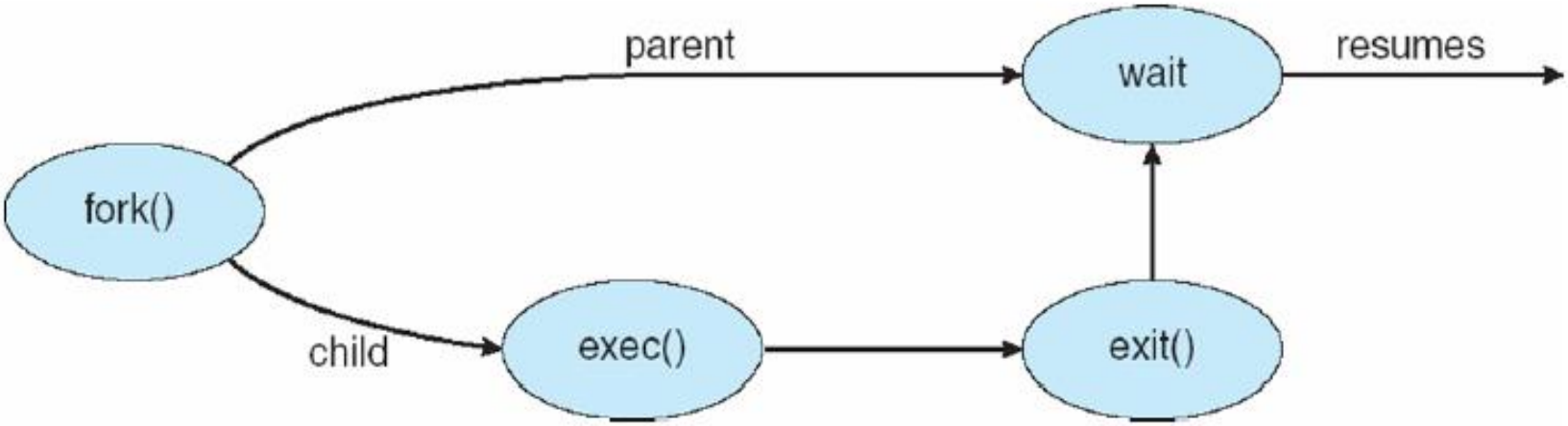
- Execution
  - Parent and Children execute Concurrently
  - Parent waits until some or all of its children have terminated

Process Creation (Unix/Linux)

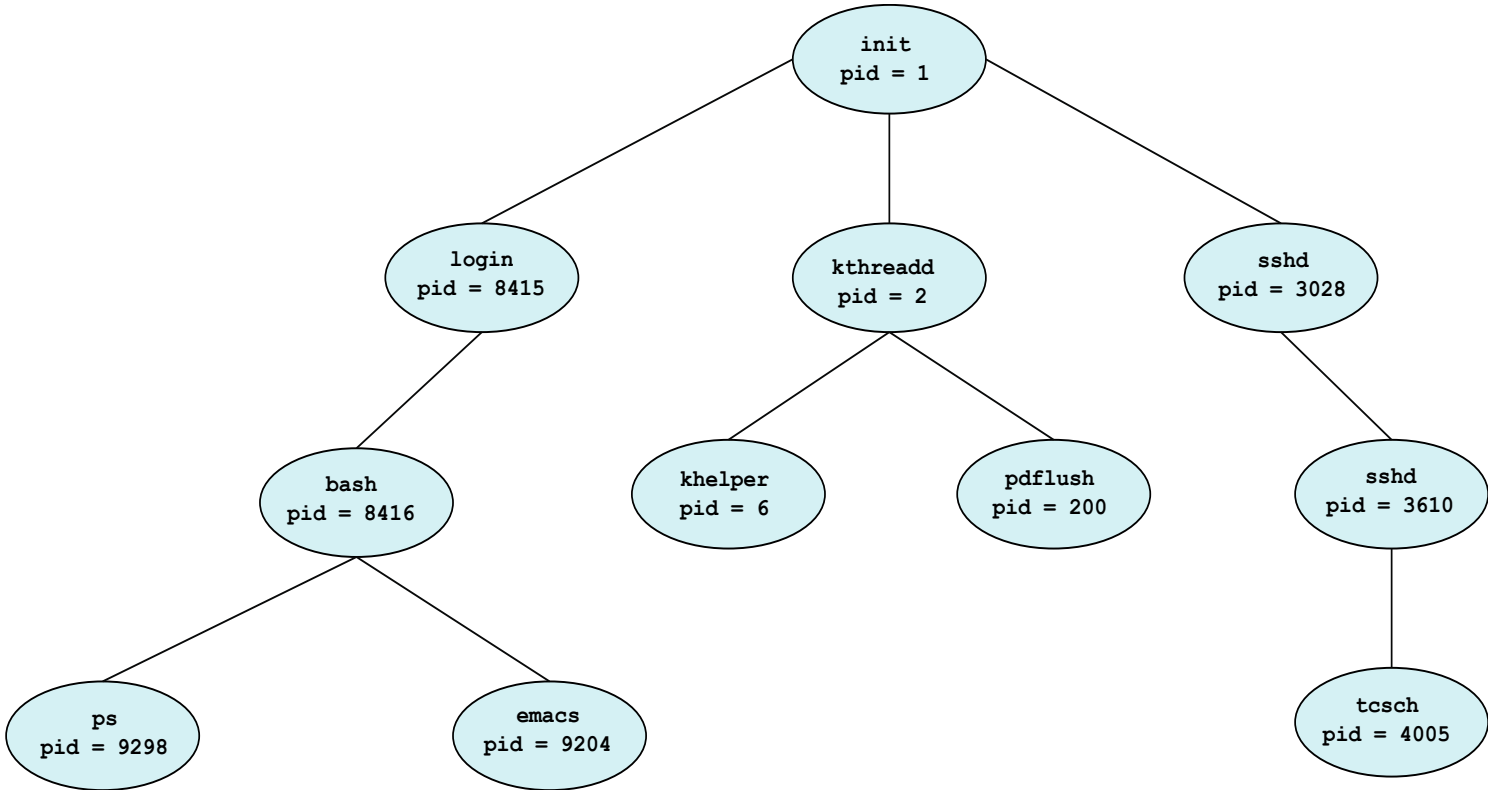
- ***Fork*** System Call creates new Process.
  - It creates a new process that starts on the following instruction after the original parent process. The parent process also continues on the following instruction, as well.
  - The fork call returns a pid_t (an integer) to both process es. The parent process gets a return value that is the pid of the child process.The child process gets a return valu e of 0, indicating that it is the child.

- ***Exec*** System Call used after a fork to replace the process' memory space with a new Program.

Process Creation (Unix)



A Tree of Processes in Linux



C Program Forking Separate Process

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
pid_t  pid;
    /* fork another 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 */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
```

Using the program below, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

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

int main()
{
pid_t pid, pid1;

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

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       pid1 = getpid();
       printf("child: pid = %d",pid); /* A */
       printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
       pid1 = getpid();
       printf("parent: pid = %d",pid); /* C */
       printf("parent: pid1 = %d",pid1); /* D */
       wait(NULL);
    }

    return 0;
}
```

Including the initial parent process, how many processes are created by the program shown below:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Predict the Output

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

void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

Predict the Output

```
int main()
{
        if(fork()||fork())
        {
                fork();
        }
cout<<”hello”;
return 0;
}
```

Predict the Output

```
#include<iostream>
int main()
{
    if(fork())
    {
        if(!fork())
        {
            fork();
            cout<<"1";
        }
        else
        {
            cout<<"2";
        }

    }
    else
    {
        cout<<"3";
    }
    cout<<"4";
    return 0;
}
```

How many times “UCS303 Quiz” will be printed?

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int doYourOwnWork(){
        fork();
        fork();
        printf("UCS303 Quiz\n");
}
int main() {
        doYourOwnWork();
        printf("UCS303 Quiz\n");
}
```

A. 2
B. 4
C. 8
D. 16

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main() {
  if(fork()==0)
  {
    if(fork())
        printf("UCS303 Quiz\n");
    printf("UCS303 Quiz\n");
  }
}
```

A. 1
B. 2
C. 3
D. 4

Predict the Output

```
int main()
{
        if (fork() && (!fork())) {
                if (fork() || fork()) {
                        fork();
                }
        }
        printf("2 ");
        return 0;
}
```

Predict the Output

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

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      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 exists 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 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**

Process Termination

Zombie process

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        pid_t child_pid = fork();
        if (child_pid > 0)
                sleep(50);
        else
                exit(0);
        return 0;
}
```

Orphan process

```
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        int pid = fork();
        if (pid > 0)
                printf("in parent process");
        else if (pid == 0)
        {
                sleep(30);
                printf("in child process");
        }

        return 0;
}
```

Zombie Process

```
//zombie.c
#include<stdio.h>
#include<unistd.h>
int main()
{
   pid_t t;
   t=fork();
   if(t==0)
   {
      printf("Child having id %d\n",getpid());
    }
    else
    {
       printf("Parent having id %d\n",getpid());
       sleep(15); // Parent sleeps. Run the ps command during this time
    }
}
```

Output

$gcc zombie.c

$./a.out &

Task

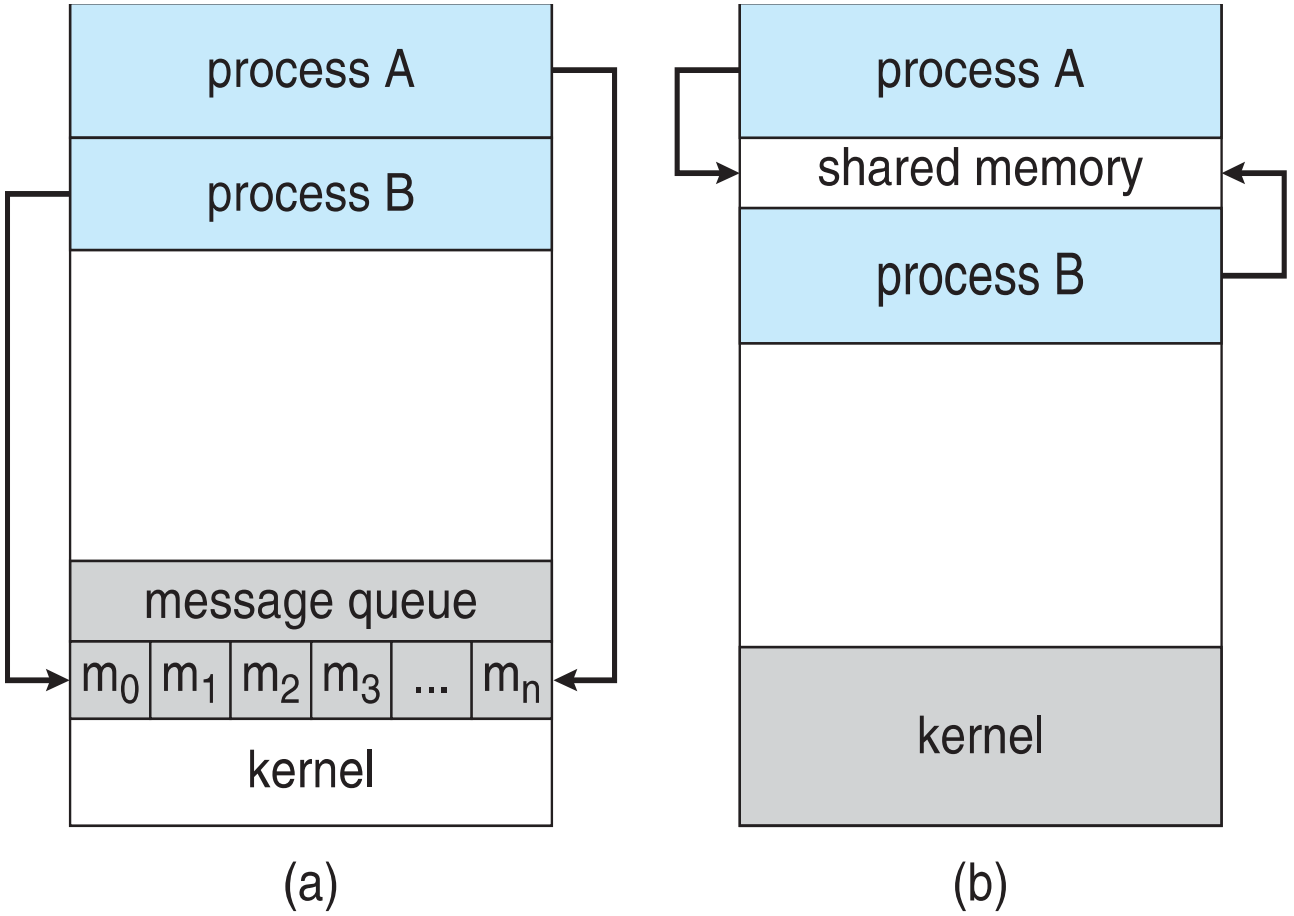
- To find the maximum number of Zombie processes that a system can handle.
- Finding Zombie and Orphan Processes in Linux
- Killing zombie process in Linux

InterProcess Communication

- 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 Inter Process Communication (IPC)
- Two models of IPC - Shared memory & Message passing

Communications Models

(a) Message passing. (b) shared memory.



Interprocess Communication – Shared Memory

- 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.

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded Buffer using Shared Memory

Producer process

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Consumer process

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

Bounded Buffer using Shared Memory

Producer process

```
while(true)
{
    /* produce an item in next produced */
    while(counter = = buffer-size);
    buffer[int] = next produced;
    in = (in+1) % buffer- size;
    counter ++;
}
```

Consumer process

```
while(true)
{
    while (counter = = 0);
    next consumed = buffer[out];
    out= (out+1) % buffer size;
    counter--;
}
```

Program 1

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
        int i;
        void *shared_memory;
        char buff[100];
        int shmid;
        shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
        printf("Key of shared memory is %d\n",shmid);
        shared_memory=shmat(shmid,NULL,0);
        printf("Process attached at %p\n",shared_memory);
        printf("Enter some data to write to shared memory\n");
        read(0,buff,100);
        strcpy(shared_memory,buff);
        printf("You wrote : %s\n",(char *)shared_memory);
}
```

Program 2

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0);
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

Output

- Program 1

```
tarun@ubuntu:~$ ./p1
Key of shared memory is 24
Process attached at 0x7f6071035000
Enter some data to write to shared memory
Hi How are you
You wrote : Hi How are you
```

- Program 2

```
tarun@ubuntu:~$ ./p2
Key of shared memory is 24
Process attached at 0x7f5336688000
Data read from shared memory is : Hi How are you

tarun@ubuntu:~$
```

Interprocess Communication – Message Passing

- 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

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P*, *message*) – send a message to process P
  - **receive**(*Q*, *message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  send(*A*, *message*) – send a message to mailbox A

  receive(*A*, *message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- 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

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Producer-consumer using message passing

```
message next_produced;
        while (true) {
            /* produce an item in next produced */
        send(next_produced);
        }


message next_consumed;
         while (true) {
            receive(next_consumed);

            /* consume the item in next consumed */
         }
```

Program 1 send.c

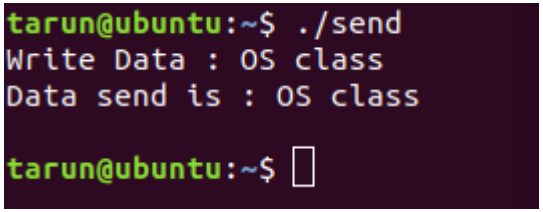
```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10
struct mesg_buffer {
        long int mesg_type;
        char mesg_text[100];
} message;
int main()
{
        key_t key;
        int msgid;
        key = ftok("progfile", 65);
        msgid = msgget(key, 0666 | IPC_CREAT);
        message.mesg_type = 1;
        printf("Write Data : ");
        fgets(message.mesg_text,MAX,stdin);
        msgsnd(msgid, &message, sizeof(message), 0);
        printf("Data send is : %s \n", message.mesg_text);
        return 0;
}
```

Program 2 recv.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct mesg_buffer {
        long int mesg_type;
        char mesg_text[100];
} message;
int main()
{
        key_t key;
        int msgid;
        key = ftok("progfile", 65);
        msgid = msgget(key, 0666 | IPC_CREAT);
        msgrcv(msgid, &message, sizeof(message), 1, 0);
        printf("Data Received is : %s \n",message.mesg_text);
        msgctl(msgid, IPC_RMID, NULL);
        return 0;
}
```

Output

- Program 1



- Program 2