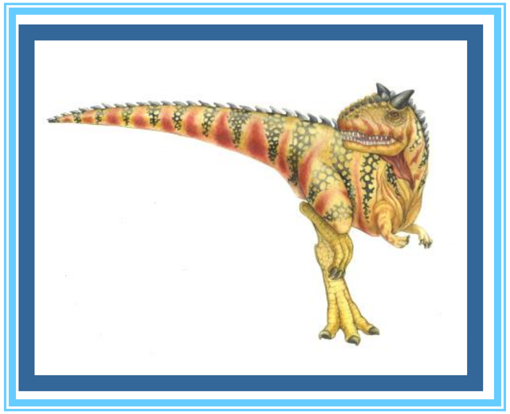
Processes
Day3: Sep 2021

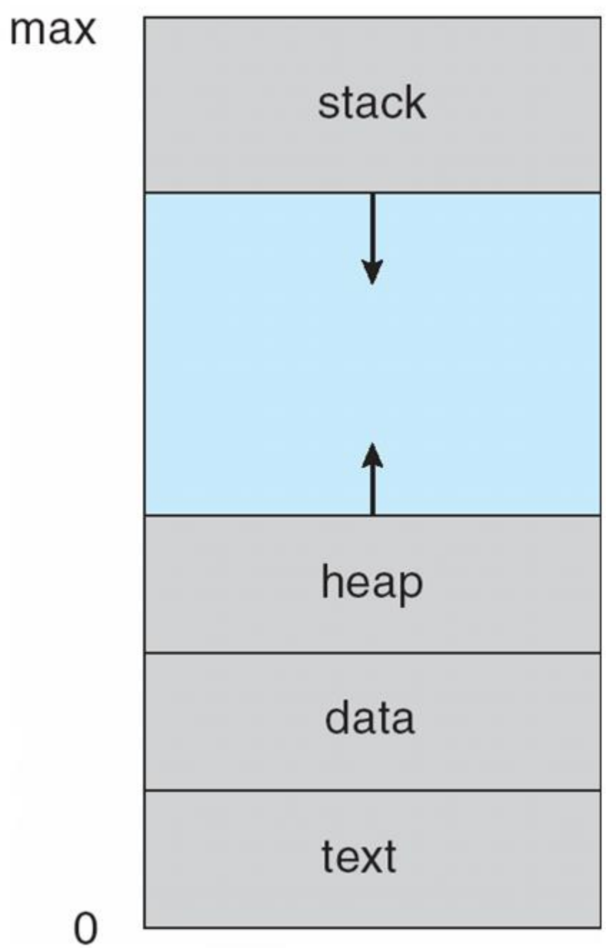
Kiran Waghmare



Agenda: Processes

- Preemptive and non preemptive
- Process mgmt
- Process life cycle
- Schedulers
- Scheduling algorithms
- Creation of fork, waitpid, exec system calls
- Orphan and zombie

Process in Memory



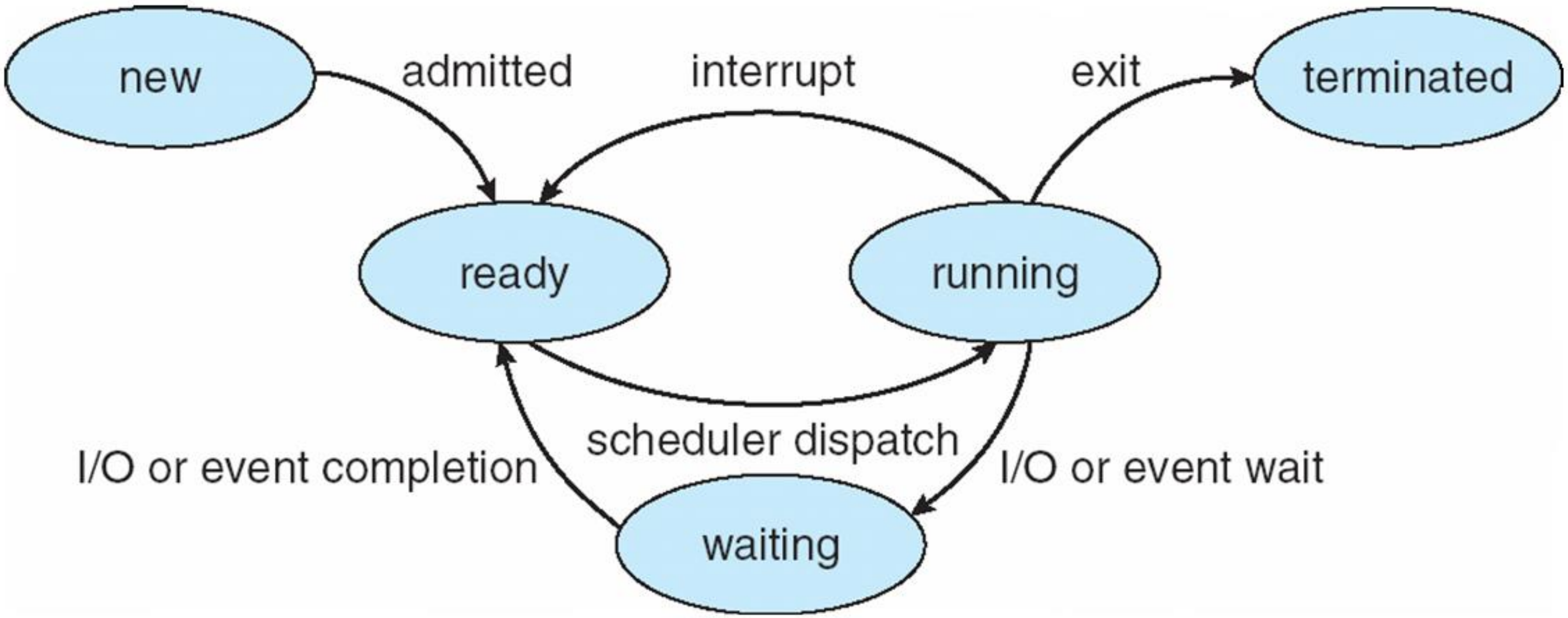
Process Concept

- 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

Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **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

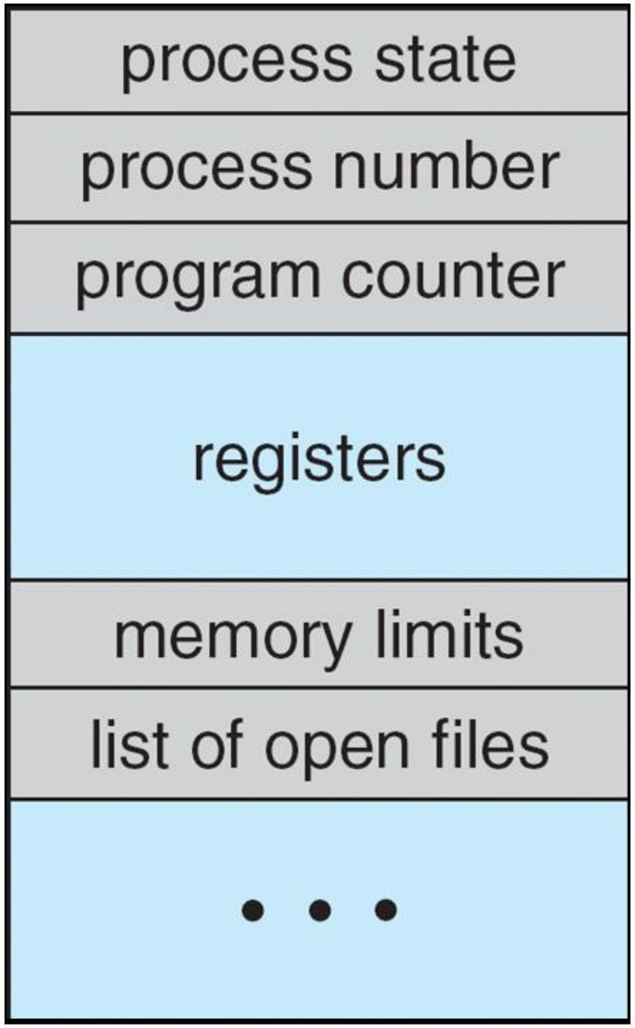


Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)



Process Scheduling

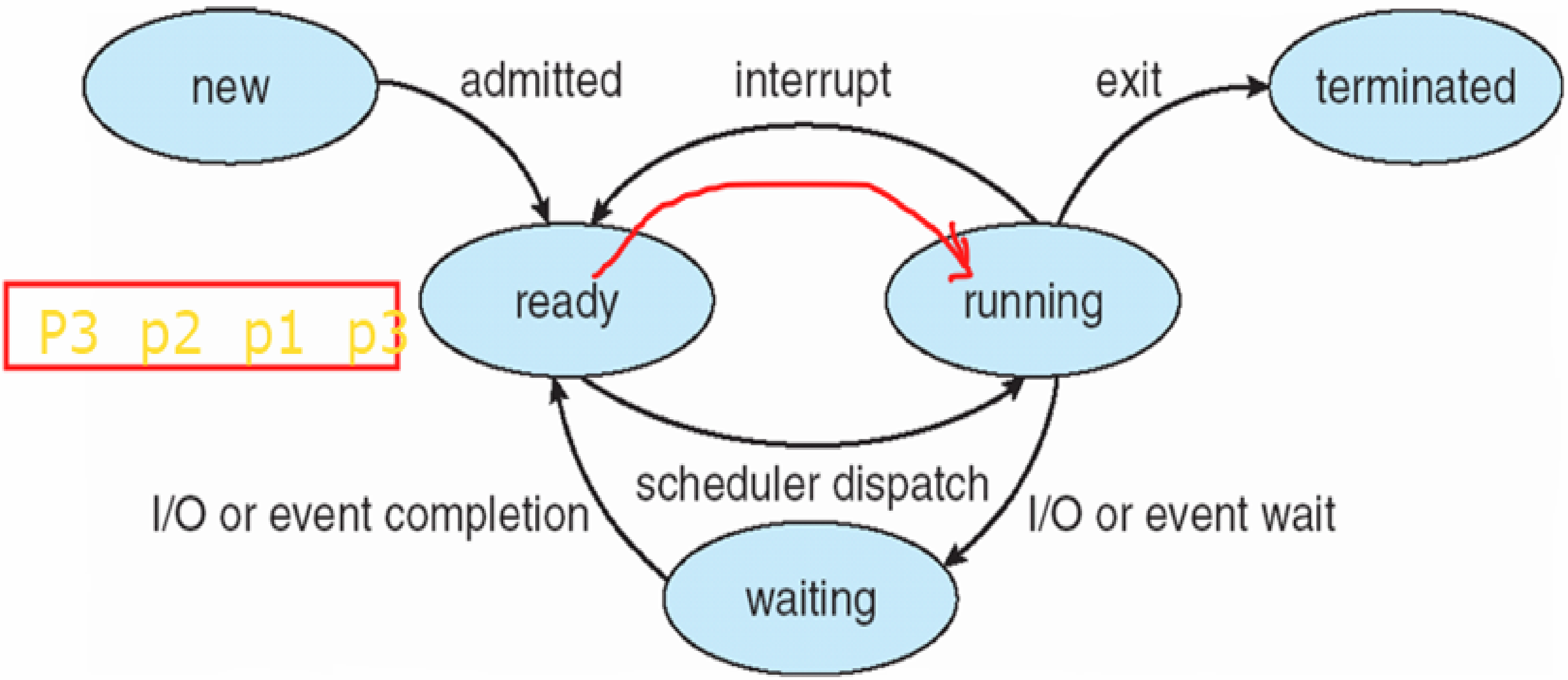
- When there are two or more runnable processes then it is decided by the Operating system which one to run first then it is referred to as Process Scheduling.
- A scheduler is used to make decisions by using some scheduling algorithm.
- Given below are the properties of a **Good Scheduling Algorithm**:
- Response time should be **minimum** for the users.
- The **number of jobs processed per hour should be maximum** i.e Good scheduling algorithm should give maximum throughput.
- The utilization of the CPU should be 100%.
- Each process should get a fair share of the CPU.

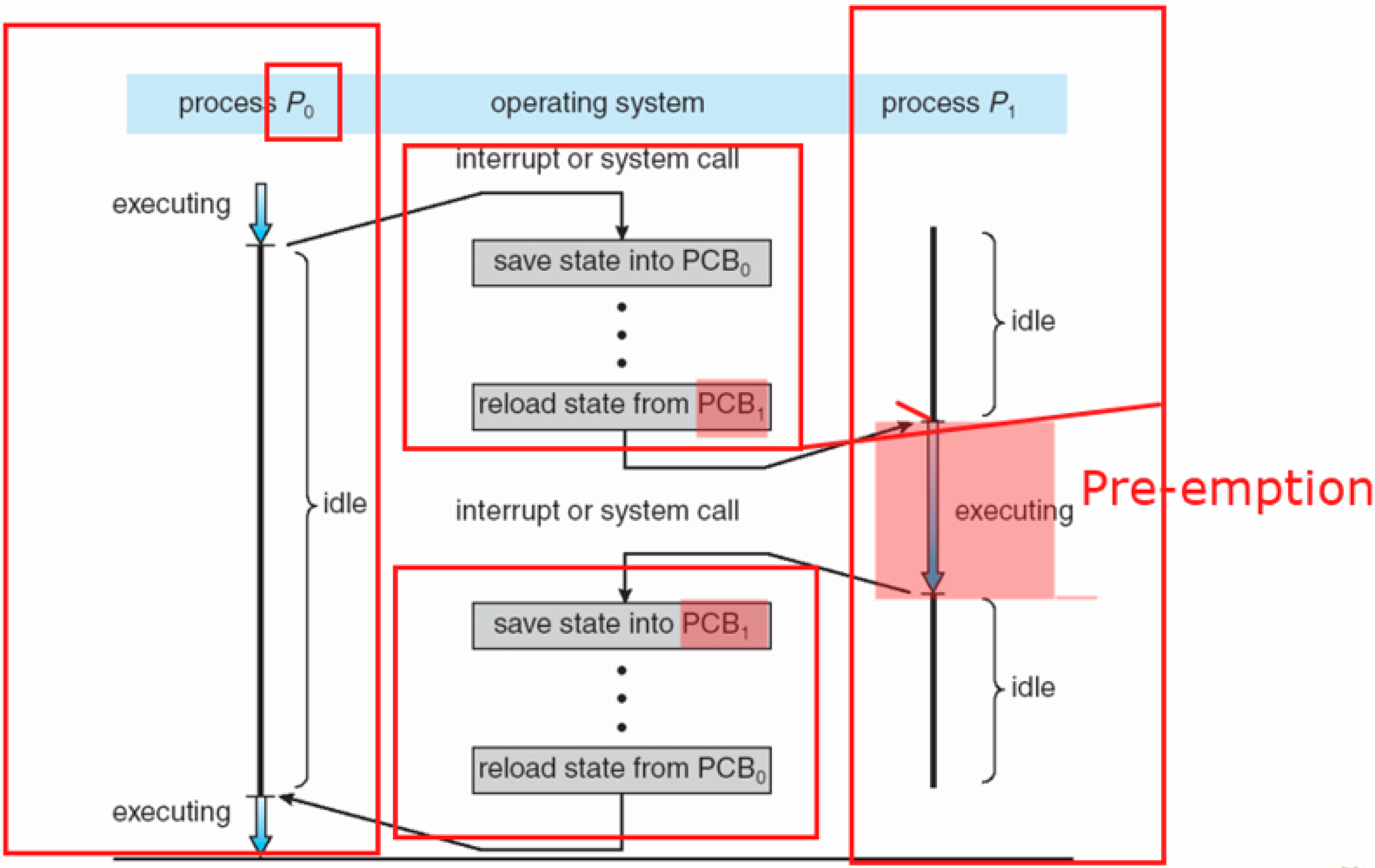
Diagram of Process State

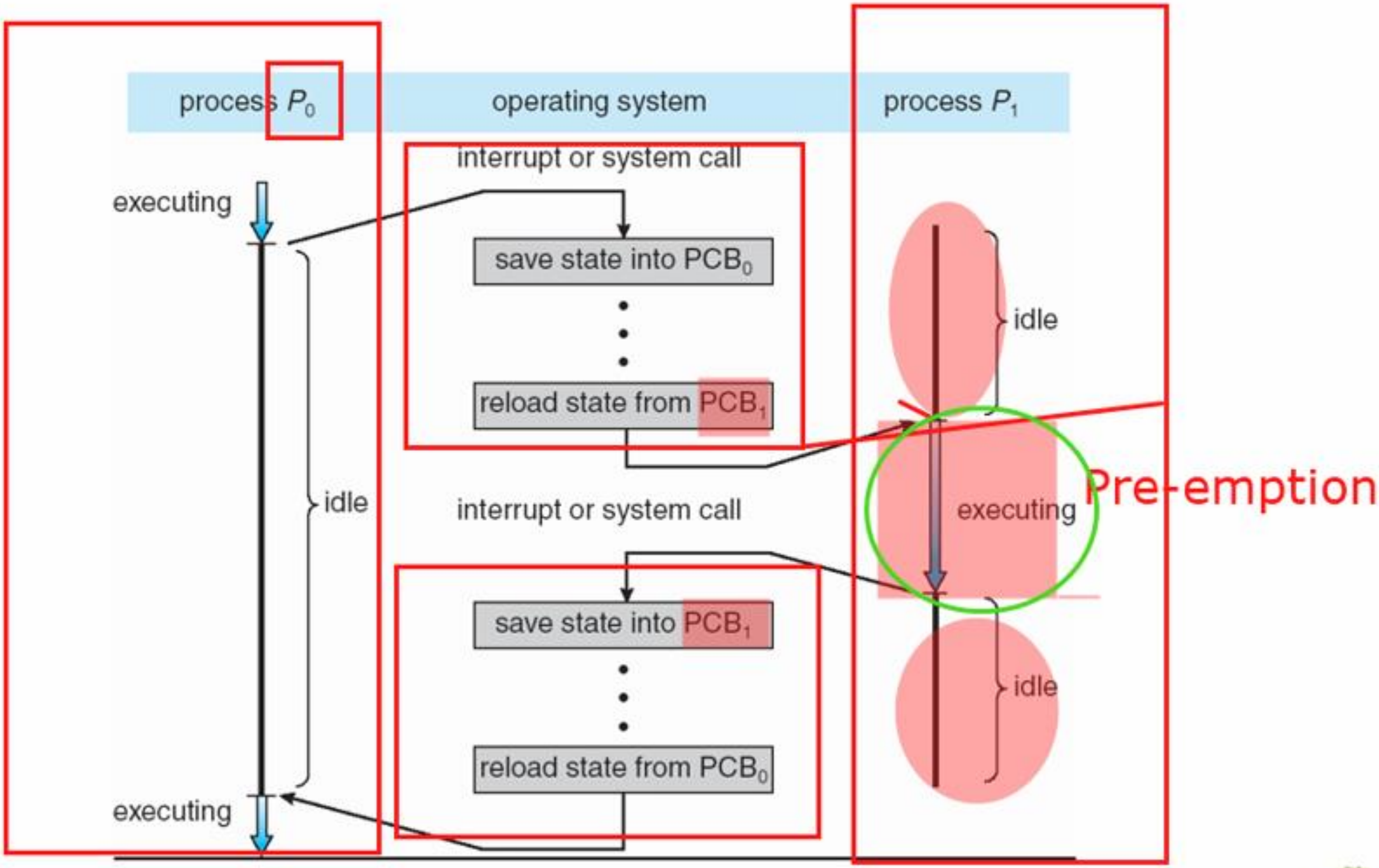
Scheduling Categories

Pre-emptive

Non- preemptive

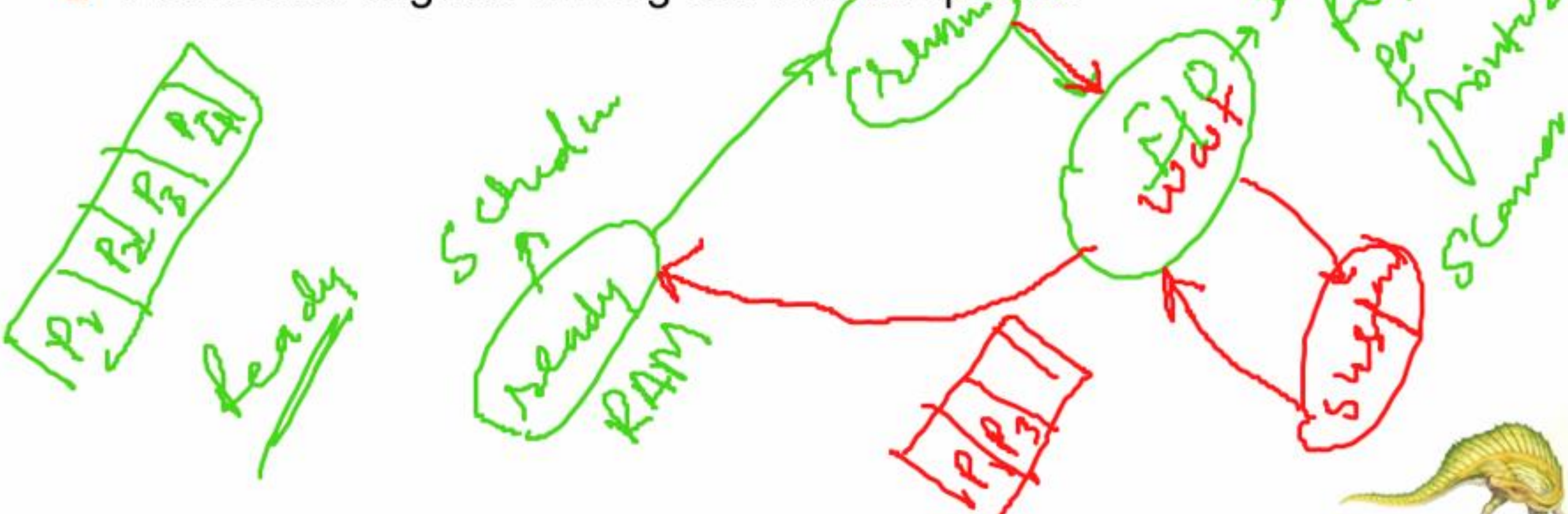



process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
idle
reload state from PCB_1
Pre-emption
idle
interrupt or system call
executing
save state into PCB_1
idle
reload state from PCB_0
executing

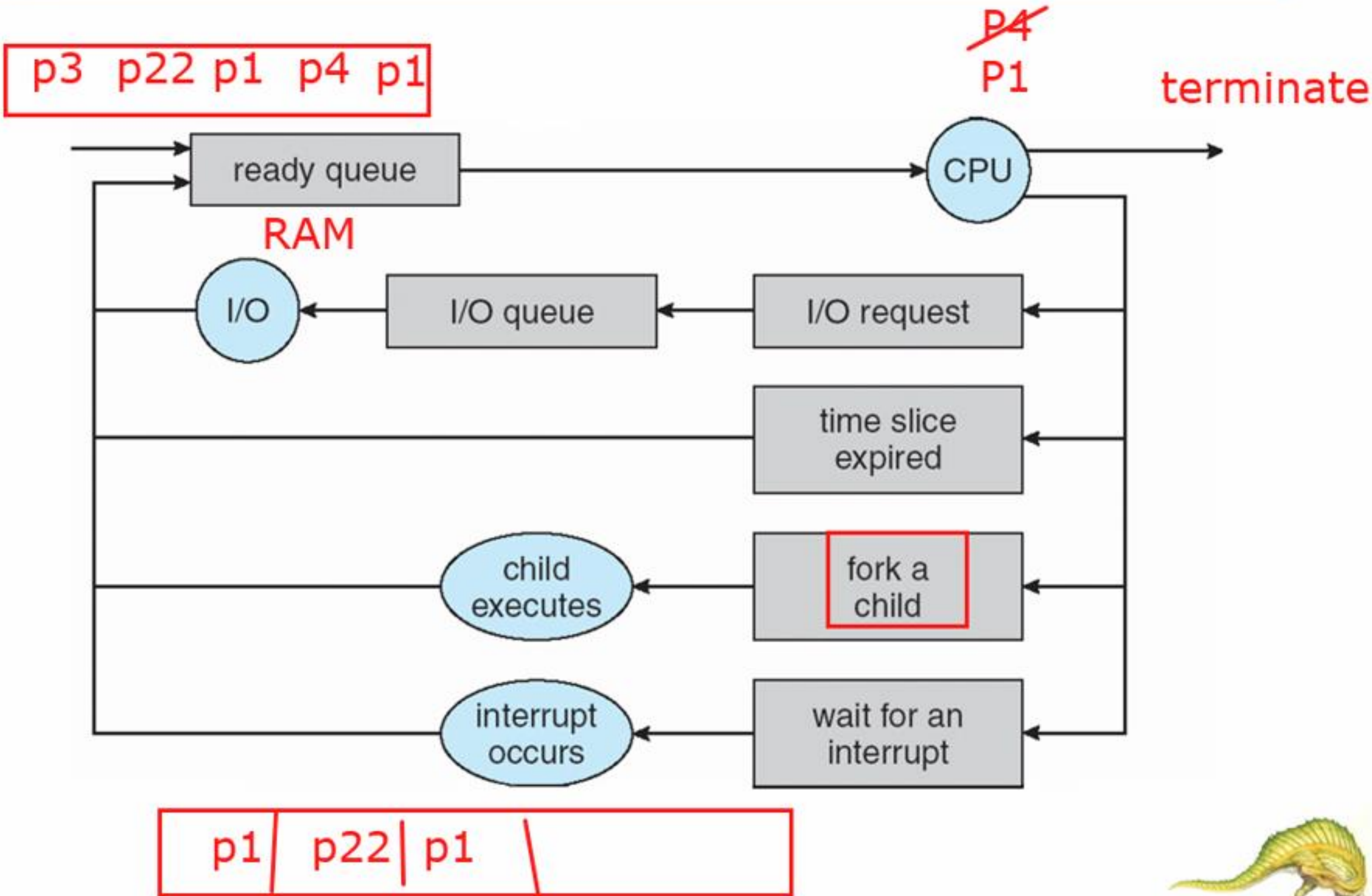

process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
reload state from PCB_1
idle
idle
executing
Pre-emption
interrupt or system call
save state into PCB_1
idle
reload state from PCB_0
executing

Process Scheduling Queues

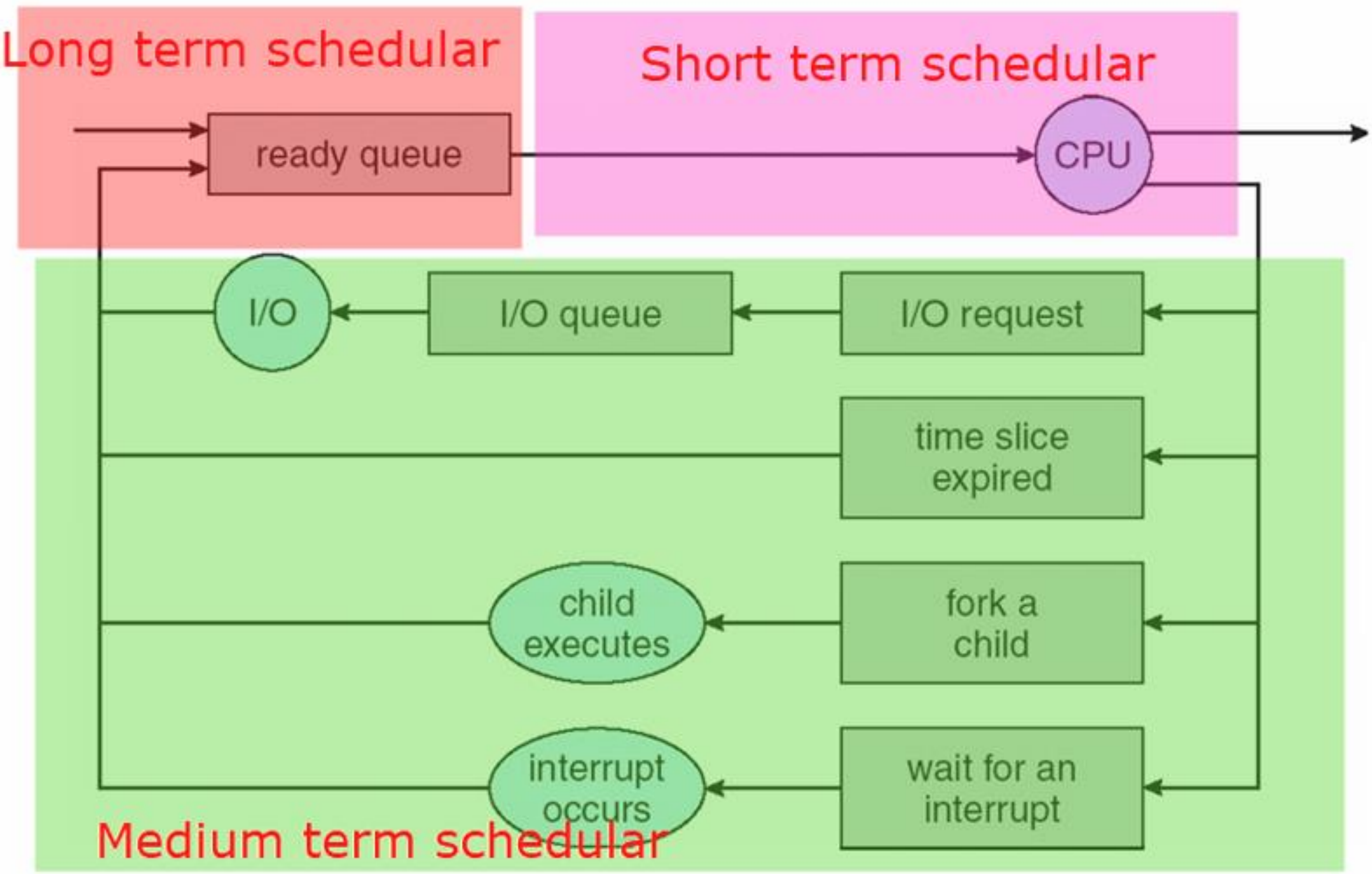
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes **residing in main memory**, ready and waiting to execute
- **Device queues** – set of processes **waiting for an I/O** device
- Processes migrate among the various queues



Representation of Process Scheduling



Representation of Process Scheduling

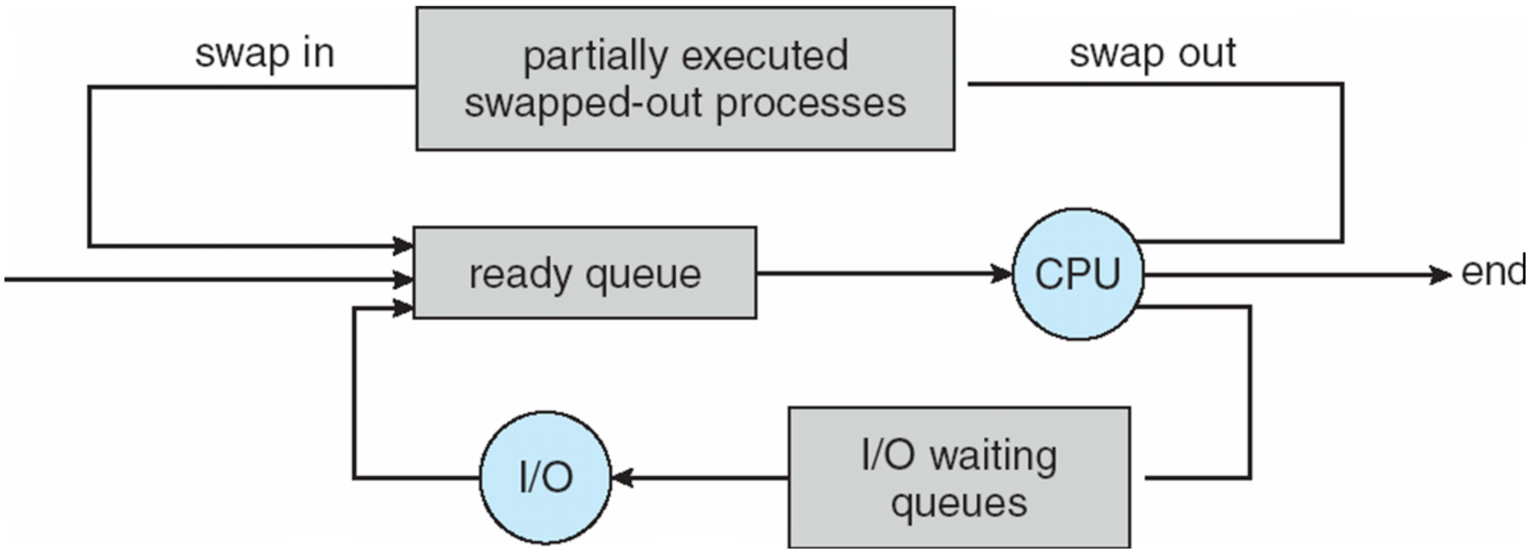


Types of Schedulers

- There are three types of schedulers available:

- Long Term Scheduler
- Short Term Scheduler
- Medium Term Scheduler

Addition of Medium Term Scheduling



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
- Time dependent on hardware support

```c
#include<stdio.h>

void main(int argc, char *argv[])
{
    int pid;

    /* Fork another process */
    pid = fork();

    if(pid < 0)
    {
        //Error occurred
        fprintf(stderr, "Fork Failed");
        exit(-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");
        exit(0);
    }
}
```

GATE Numerical Tip: If fork is called for n times, the number of child processes or new processes created will be: 2^n – 1.

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
  - 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 children terminate

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

Process Operations

-1. Process creation

-fork,spawn

-2. Process Termination

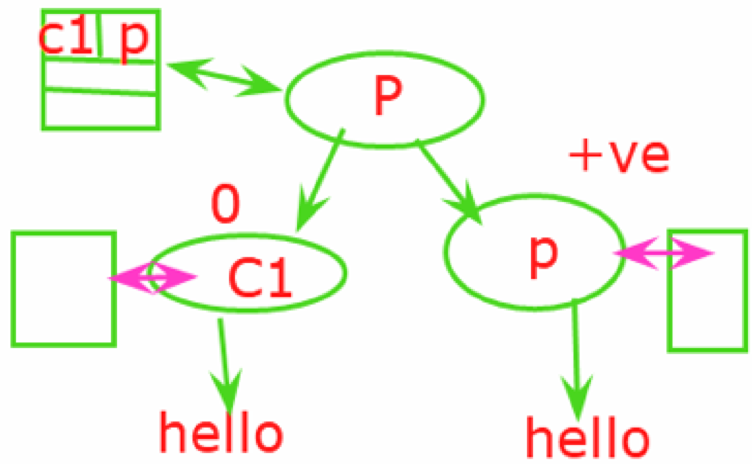
fork()

-copy of parent, child

-child : 0

-Paarent : +ve

```
main()
{
fork();
printf("hello");
}
```



Process Operations

-1. Process creation

-fork,spawn

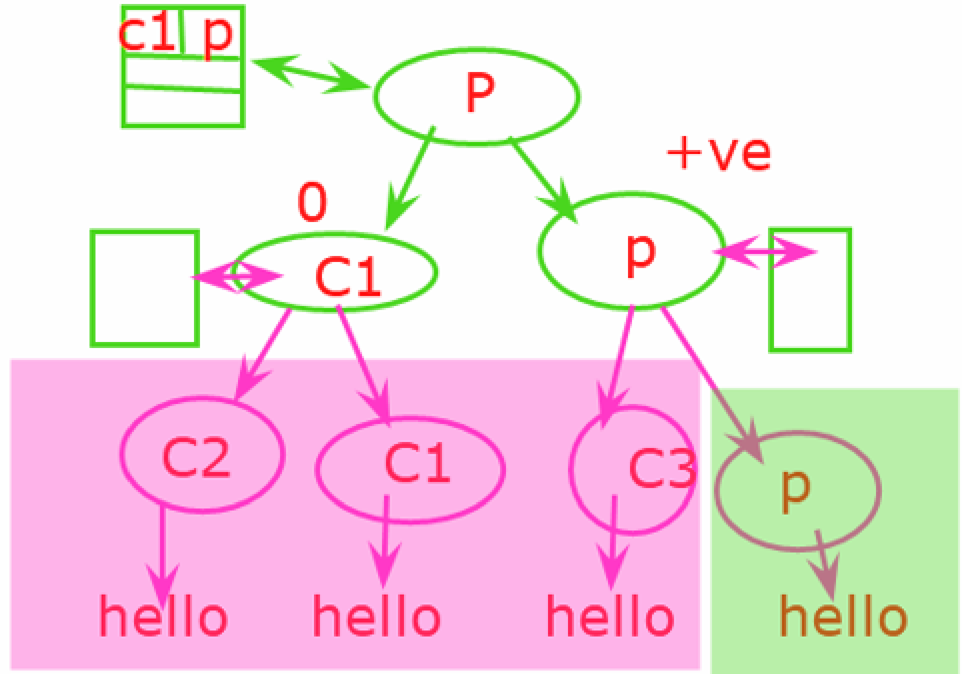
-2. Process Termination

fork()

-copy of parent, child

-child : 0

-Paarent : +ve



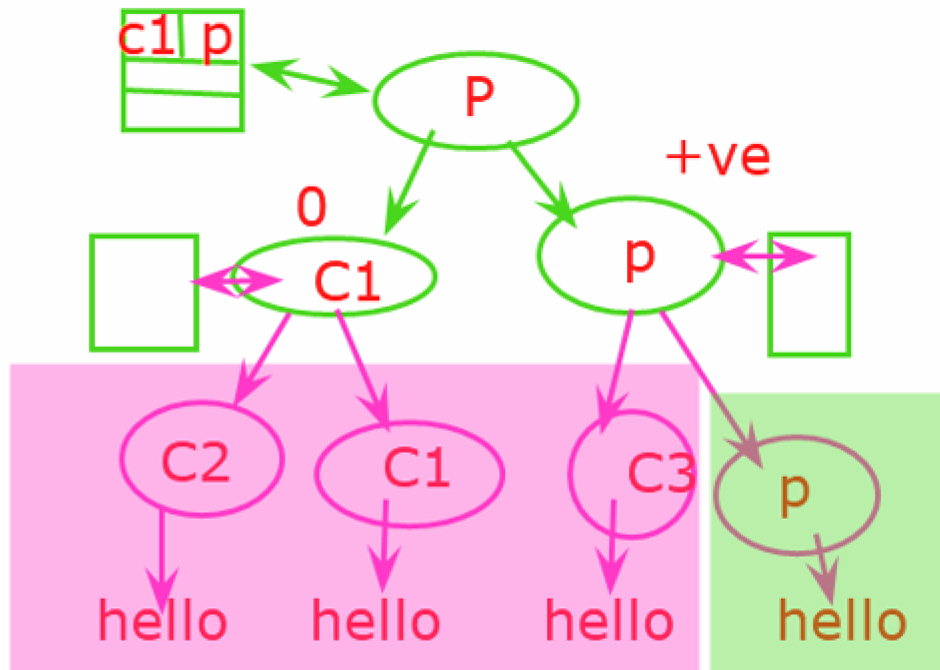

```
main()
{
fork();
 fork() ;
fork();
printf("hello");
}
```

Process Operations
-1. Process creation
-fork,spawn

-2. Process Termination

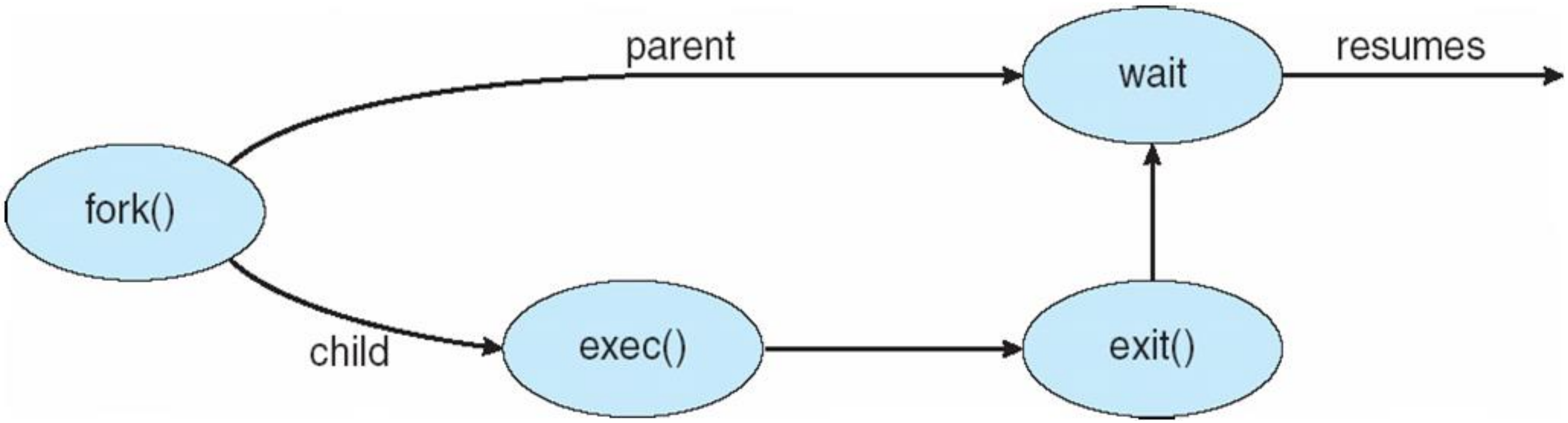
fork()
-copy of parent, child
-child : 0
-Paarent : +ve

```
main()
{
fork();
 fork() ;
fork();
printf("hello");
}
```



2^n-1

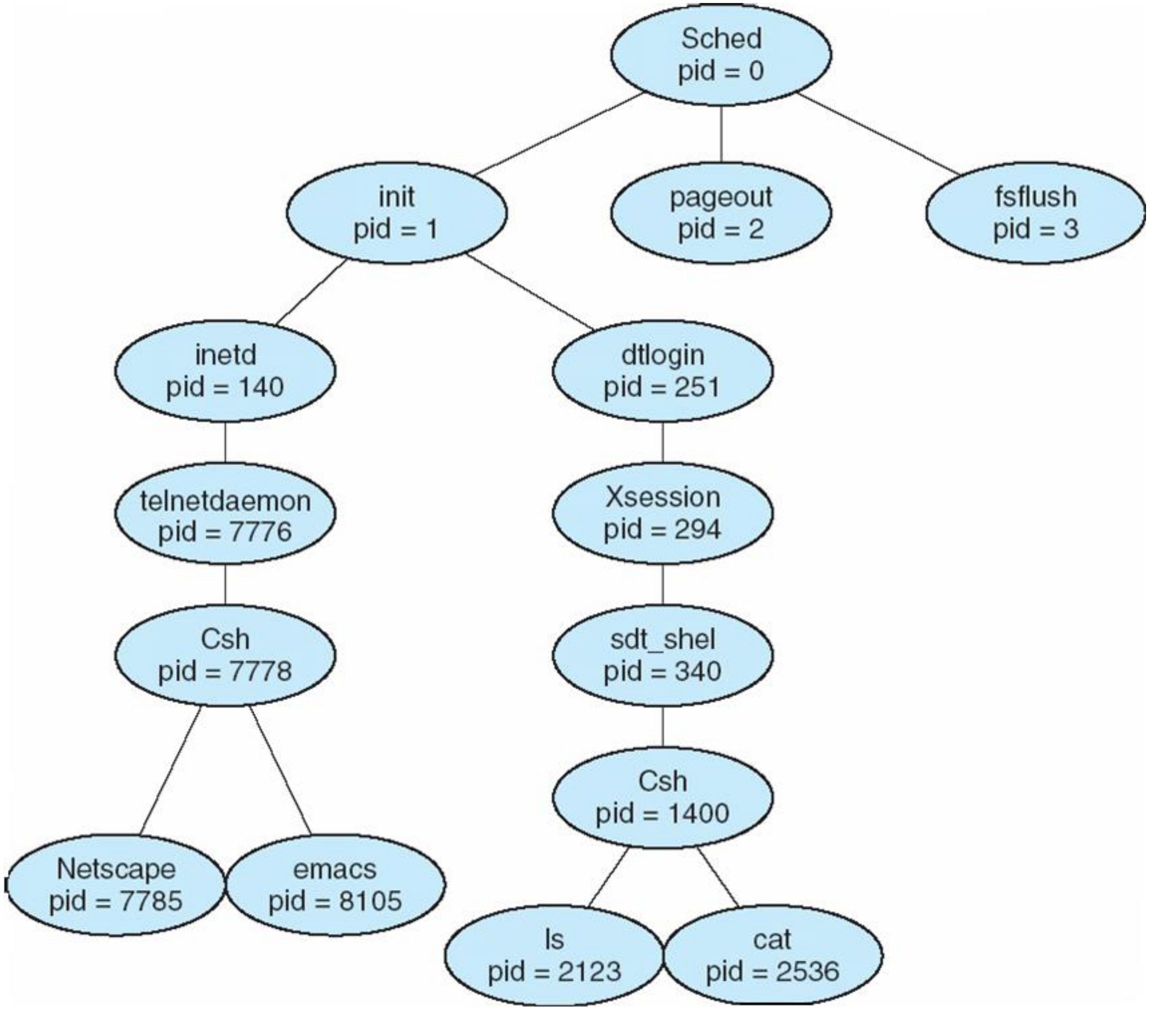
Process Creation



C Program Forking Separate Process

```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-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");
          exit(0);
    }
}
```

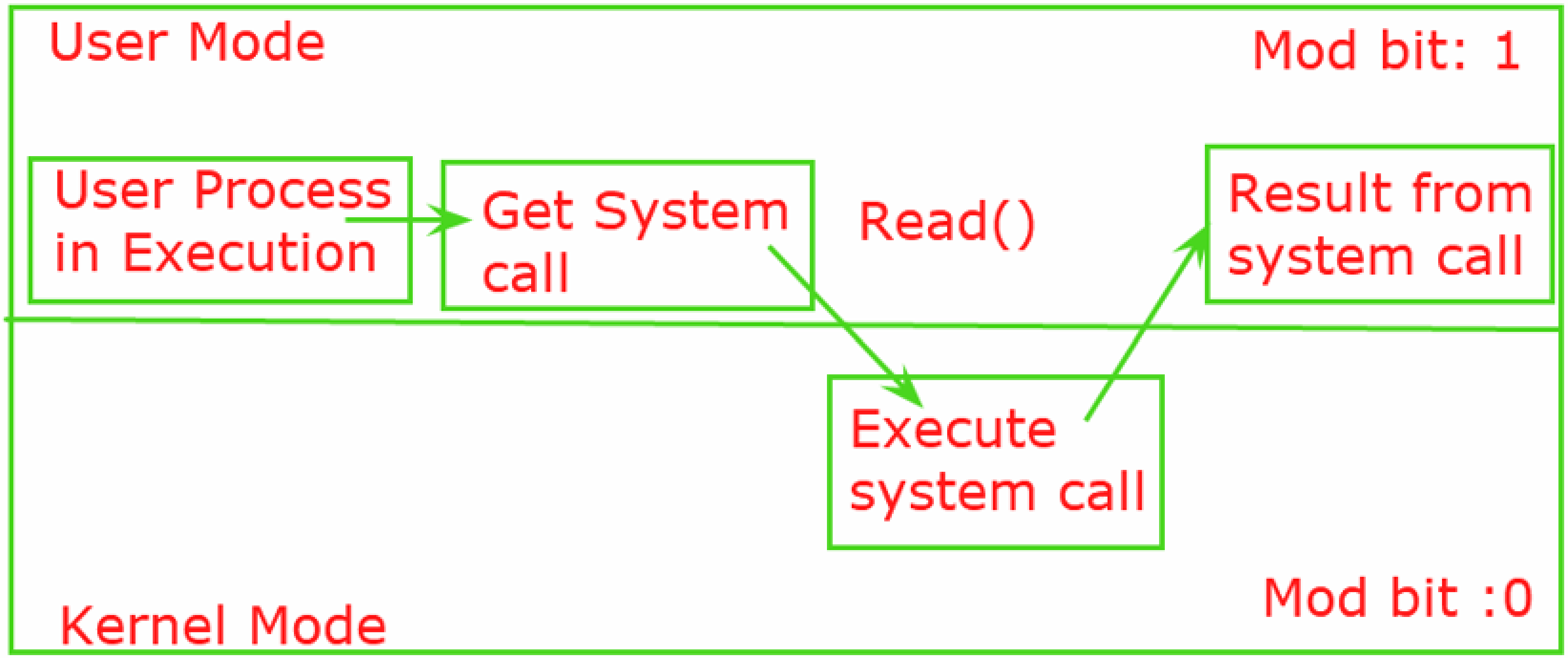
A tree of processes on a typical Solaris



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

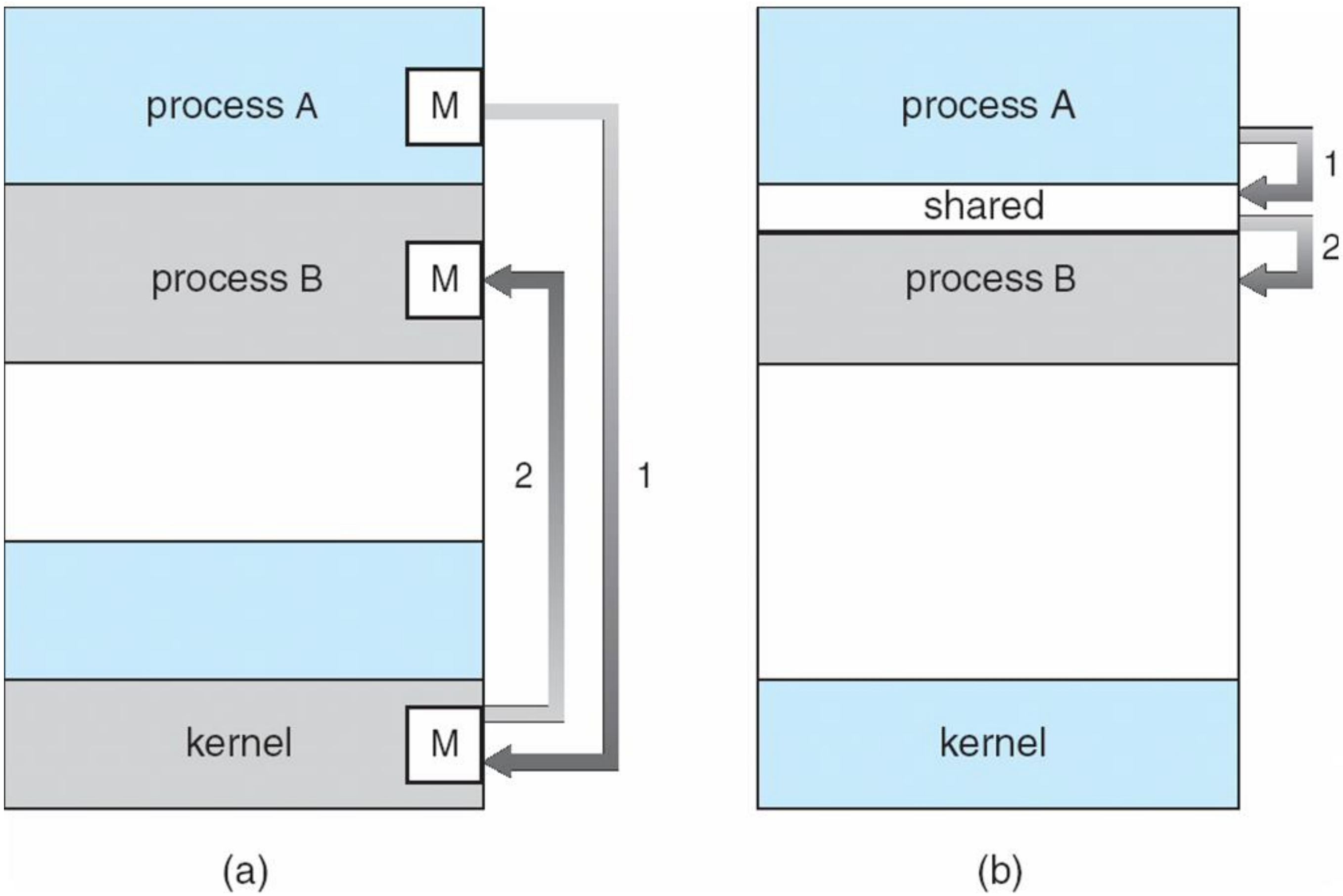
User Vs Kernel Mode



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

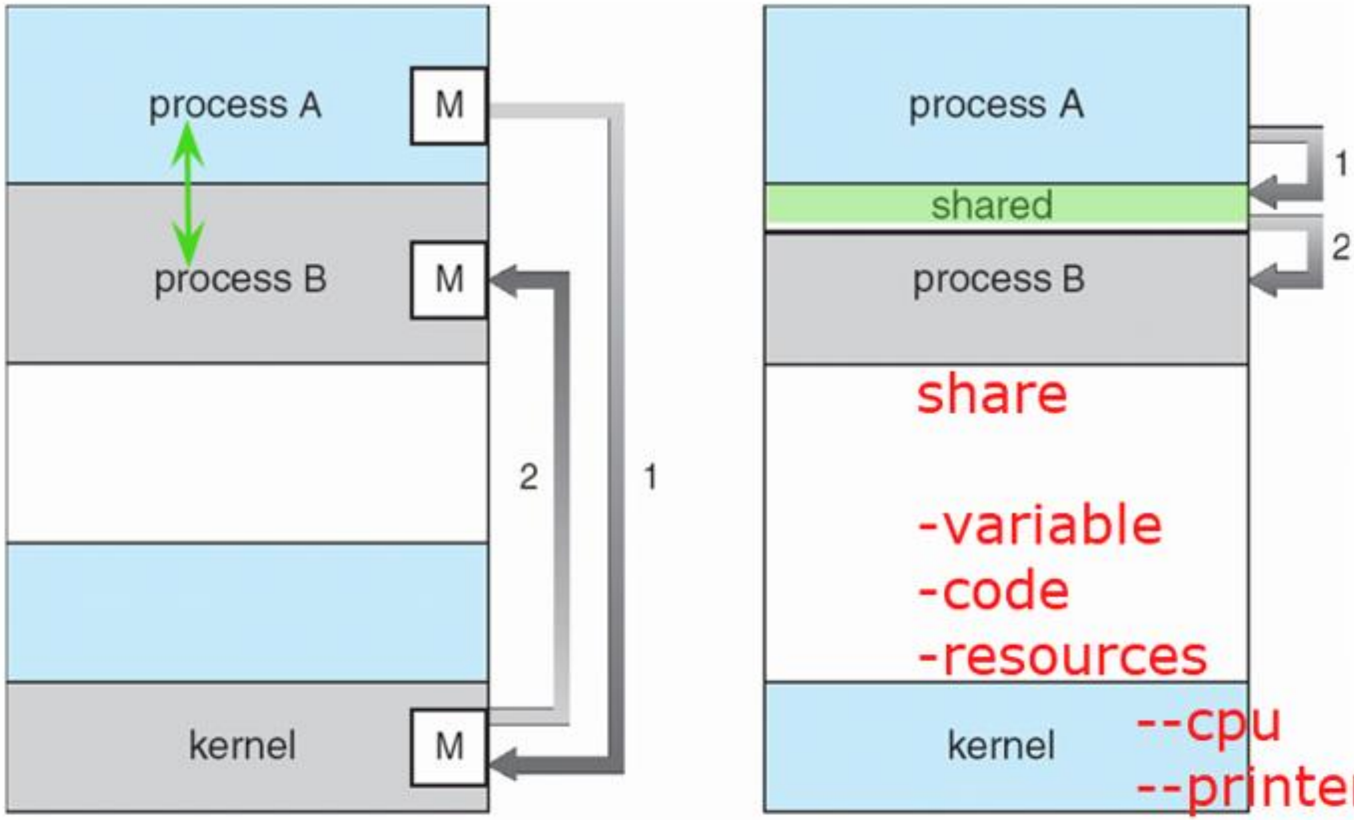
Communications Models



Communications Models

Independent process

Cooperative process



share

-variable
-code
-resources

--cpu
--printer
--scanner

(a) (b)

Soln : Synchronization

Process A

Process B

Ask for same resource-->Racing cond

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

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

Producer-Consumer Problem

n=6, 0-5

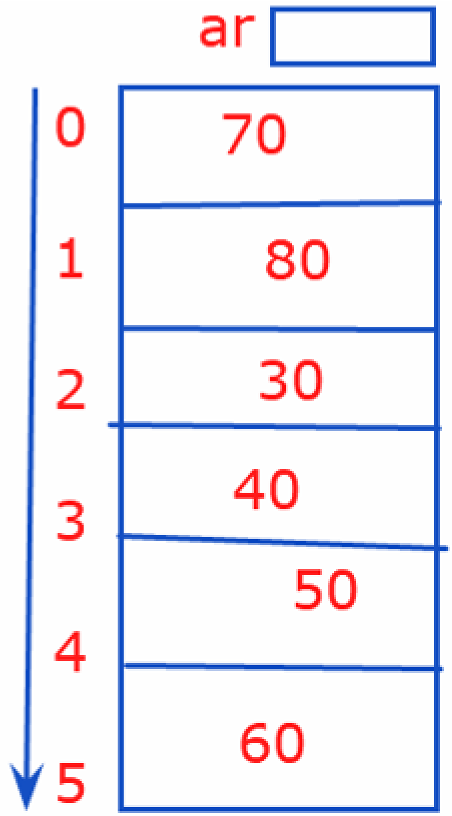
Consumer

while(ar == n)

10

20

ar= ar-1



Buffer

Producer

while(ar == n)

10

20

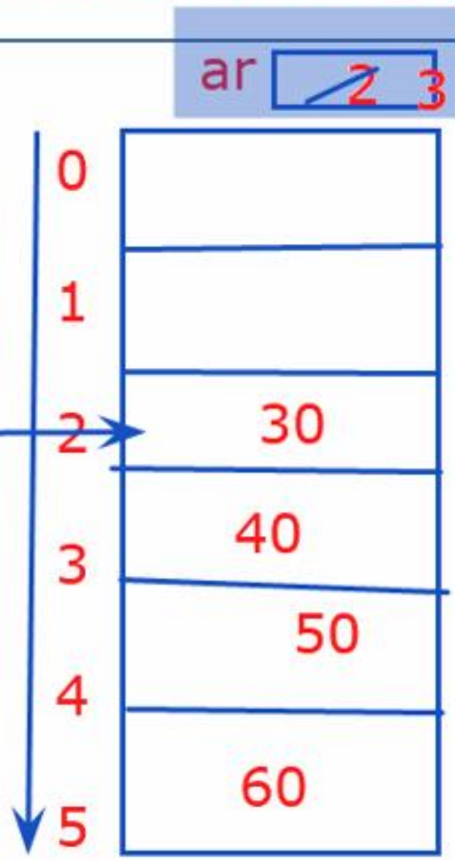
30

ar=ar+1

40

Producer-Consumer Problem

n=6, 0-5



Buffer

Consumer

while(ar == n)

20

ar= ar-1

Producer

while(ar == n)

10

20

30

ar=ar+1

40

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 – Producer

```
while (true) {
   /* Produce an item */

  while (((in = (in + 1) % BUFFER SIZE count)  == out)

     ;   /* do nothing -- no free buffers */

    buffer[in] = item;

    in = (in + 1) % BUFFER SIZE;

}
```

Bounded Buffer – Consumer

```
while (true) {
    while (in == out)
          ; // do nothing -- nothing to consume

     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
 return item;
}
```

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*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

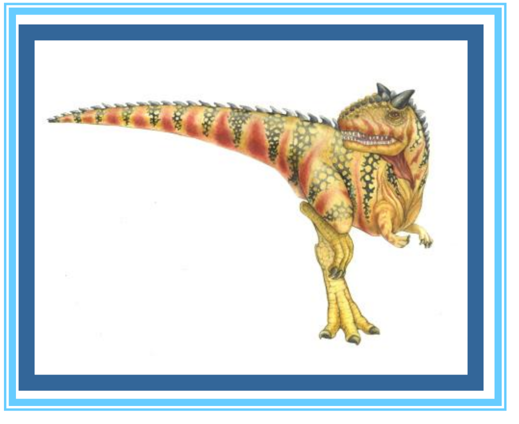
Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

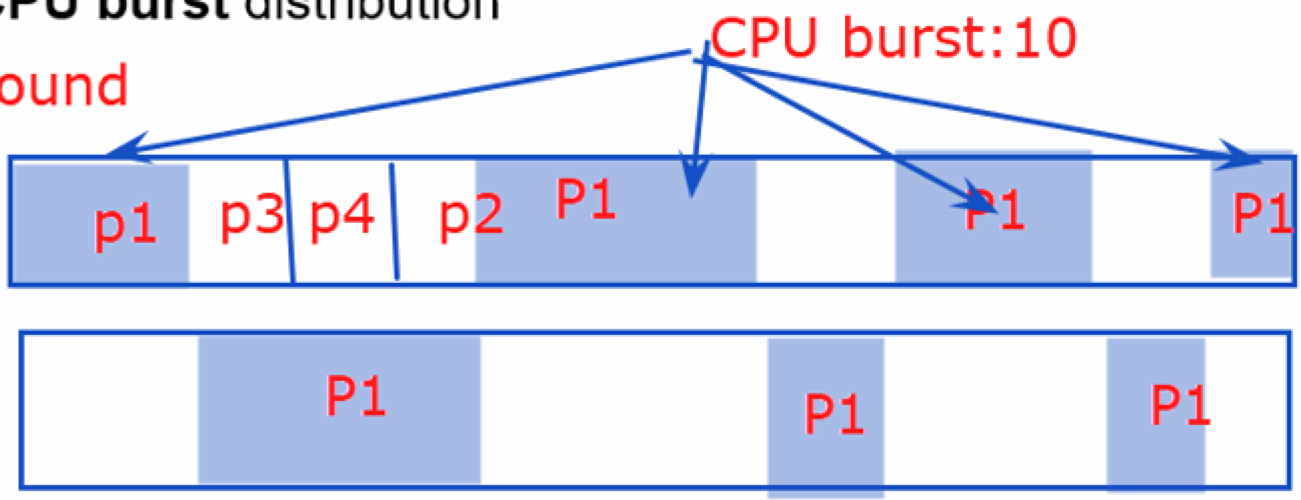
CPU Scheduling



Basic Concepts

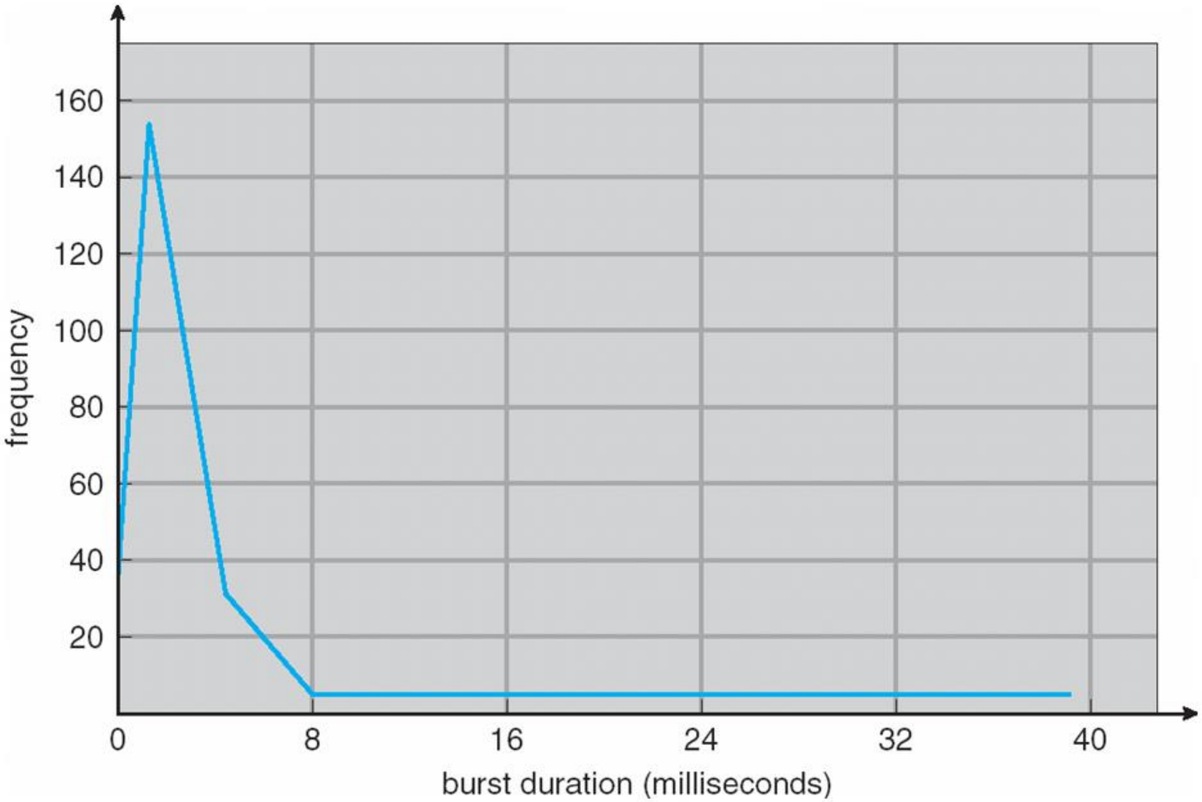
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution

CPU bound

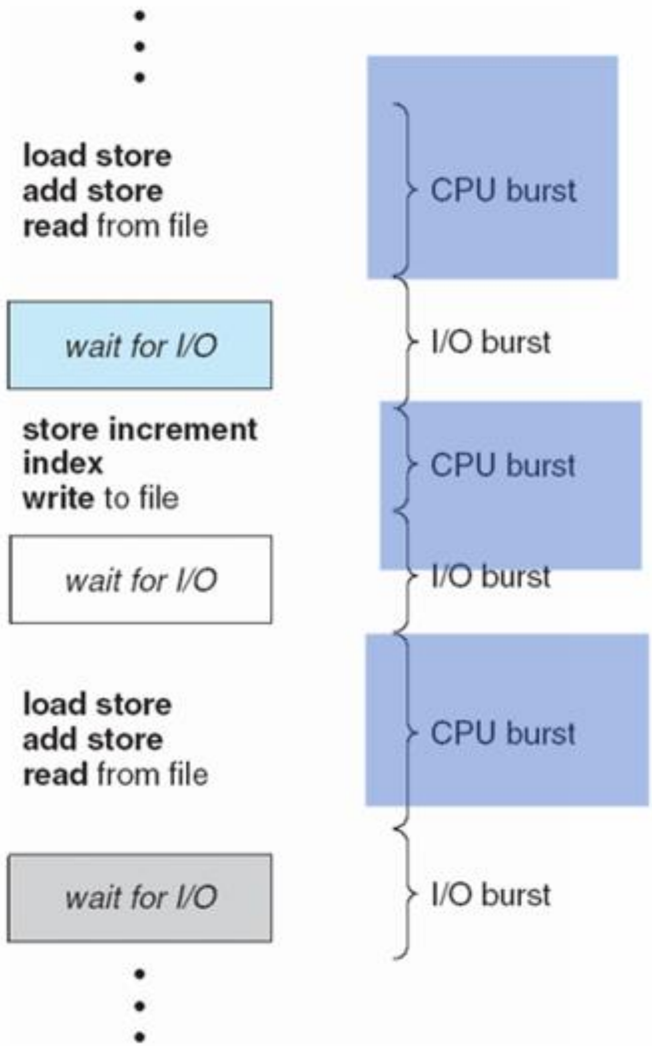


I/o bound

Histogram of CPU-burst Times



Alternating Sequence of CPU And I/O Bursts



1. Running -> waiting
2. Running -> Ready
3. waiting -> Ready

Scheduling

CPU Scheduler

- Selects from among the **processes in memory that are ready to execute**, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from **running to waiting state**
  2. Switches from **running to ready state**
  3. Switches from **waiting to ready**
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Arrival Time :

Burst Time :

Completion Time:

Turn around Time: {Completion time -arrival time}

Waiting Time : {Turn around - burst time}

Response time : {Process with 1st cpu allotment -arrival time}

Scheduling Algorithm

Pre-emptive

SRTF

LRTF

Round Robin

Priority based

Non- pre-emptive

FCFS

SJF

LJF

HRRN

Multilevel Queue

Priority

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time	comp.T	TAT	WT	Resp T
~~P_1~~	~~24~~	24	24	0	0
~~P_2~~	~~3~~	27	27	24	24
~~P_3~~	~~3~~	30	30	27	27

Sequence: ~~P1~~-~~P2~~-P3

P1	P2	P3

0 24 27 30

AWT: (0+24+27)/3=17

ATAT : 27

Res T : 17

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

comp.T	TAT	WT	Resp T
24	24	0	0
27	27	24	24
30	30	27	27

Sequence: P2-P3-P1

Arrival time : 0

P1	P2	P3
0 – 24	24 – 27	27 – 30

Gantt Chart

AWT: (0+24+27)/3=17

ATAT : 27

Res T : 17

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

comp.T		TAT		WT		Resp T	
24	30	24	30	0	6	0	6
27	3	27	3	24	0	24	0
30	6	30	6	27	3	27	3

Sequence: P2-P3-P1

Arrival time : 0

case 1

Gantt Chart

P1	P2	P3

0 24 27 30

AWT: (0+24+27)/3=17

ATAT : 27

Res T : 17

Convoy Effect

case 2

P2	P3	P1

0 3 6 30

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

comp.T		TAT		WT		Resp T	
24	30	24	30	0	6	0	6
27	3	27	3	24	0	24	0
30	6	30	6	27	3	27	3

Sequence: P2-P3-P1

Arrival time : 0

case 1

Gantt Chart

P1	P2	P3

0 24 27 30

AWT: (0+24+27)/3=17

ATAT : 27

Res T : 17

Convoy Effect

AWT : (6+0+3)/3=3

case 2

P2	P3	P1

0 3 6 30

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request

Case 2

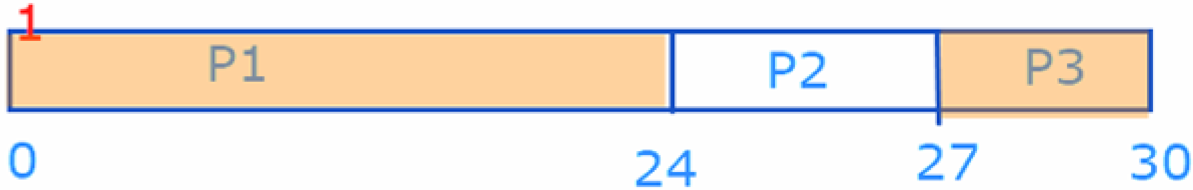
Process	Burst Time
P_1	24
P_2	3
P_3	3

comp.T		TAT		WT		Resp T	
24	30	24	30	0	6	0	6
27	3	27	3	24	0	24	0
30	6	30	6	27	3	27	3

Sequence: P2-P3-P1

Arrival time : 0

case 1



Gantt Chart

AWT: (0+24+27)/3=17

ATAT : 27

Res T : 17

Convoy Effect

AWT : (6+0+3)/3=3

case 2

Example of SJF

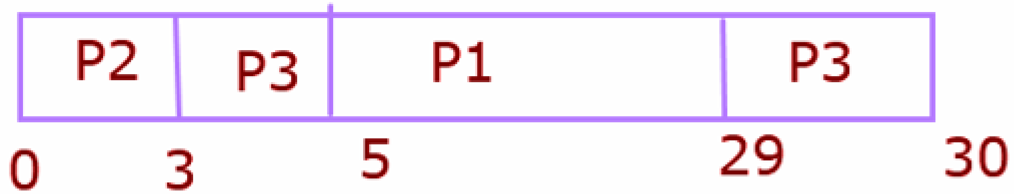
Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Priority Scheduling

	Process		Burst Time
1	P_1	5	24
2	P_2	0	3
3	P_3	1	3



Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

	Process	Burst Time
1	P_1	~~24~~ ~~21~~ ~~18~~ 15
2	P_2	~~3~~
3	P_3	~~3~~

Round Robin : time quantam

3 sec

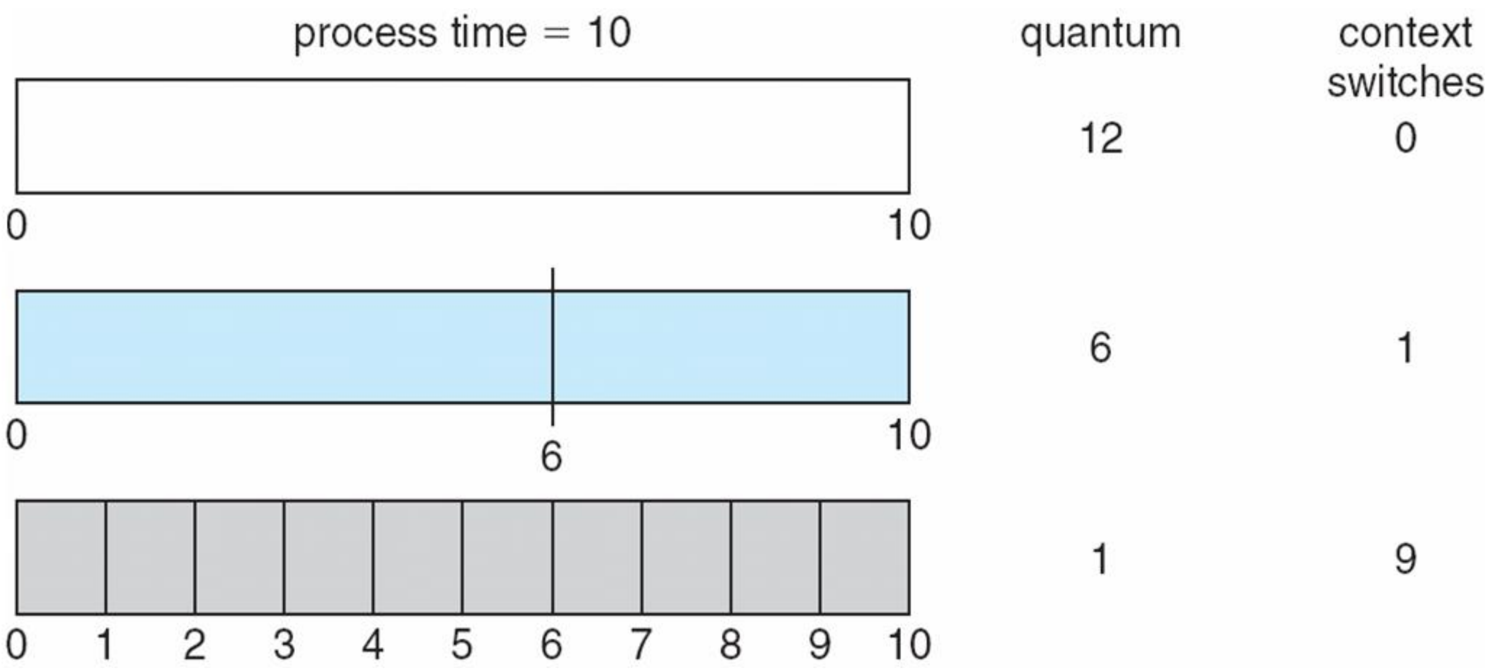
P2	P3	P1	P3

0 3 5 29 30

P1	P2	P3	P1	P1	P1	P1	P1	P1	P1

0 3 6 9 12 30

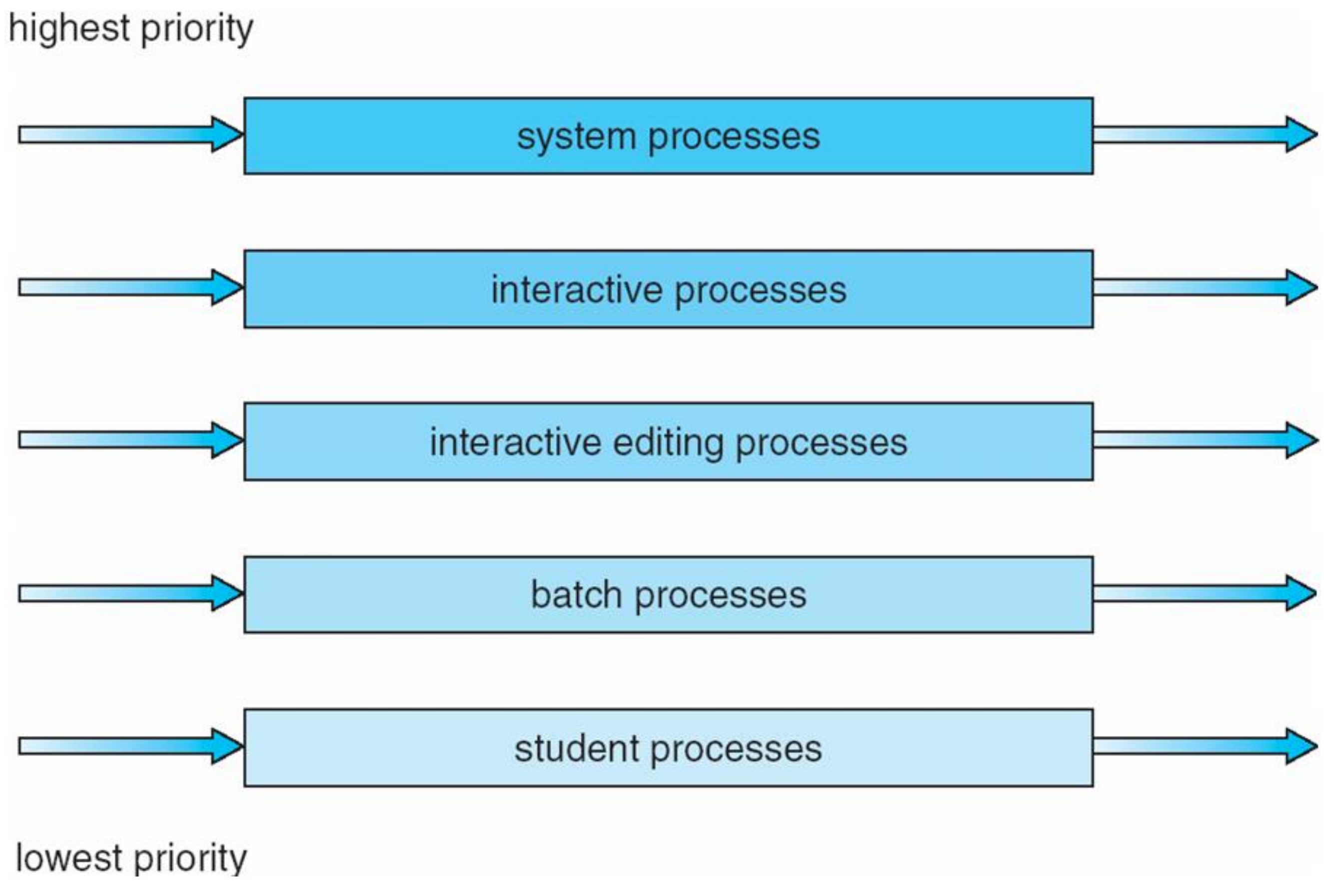
Time Quantum and Context Switch Time



Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues