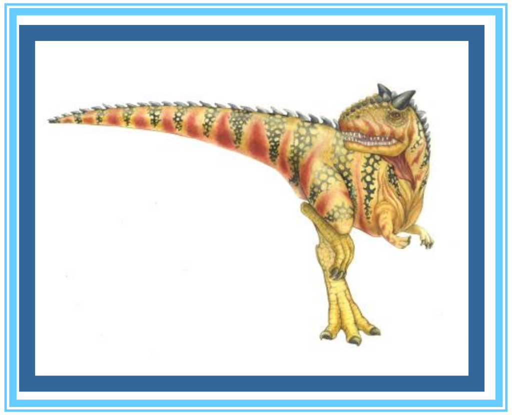
Processes
Day4: March 2022

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 vs Program

Process	**Program**
The process is basically an instance of the computer program that is being executed.	A Program is basically a collection of instructions that mainly performs a specific task when executed by the computer.
A process has a **shorter lifetime**.	A Program has a **longer lifetime**.
A Process requires resources such as memory, CPU, Input-Output devices.	A Program is stored by hard-disk and does not require any resources.
A process has a dynamic instance of code and data	A Program has static code and static data.
Basically, a process is the **running instance** of the code.	On the other hand, the program is the **executable code**.

Process in Memory

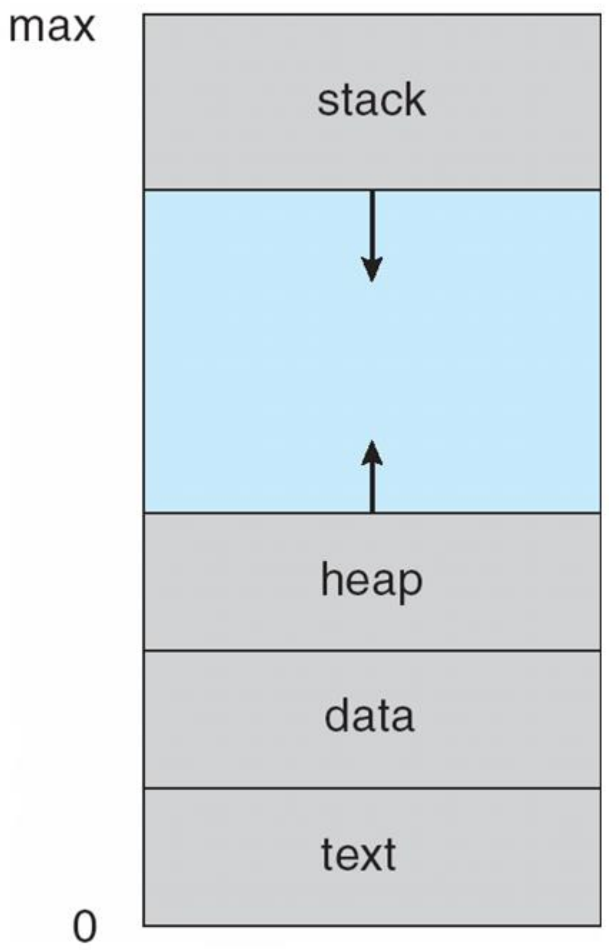
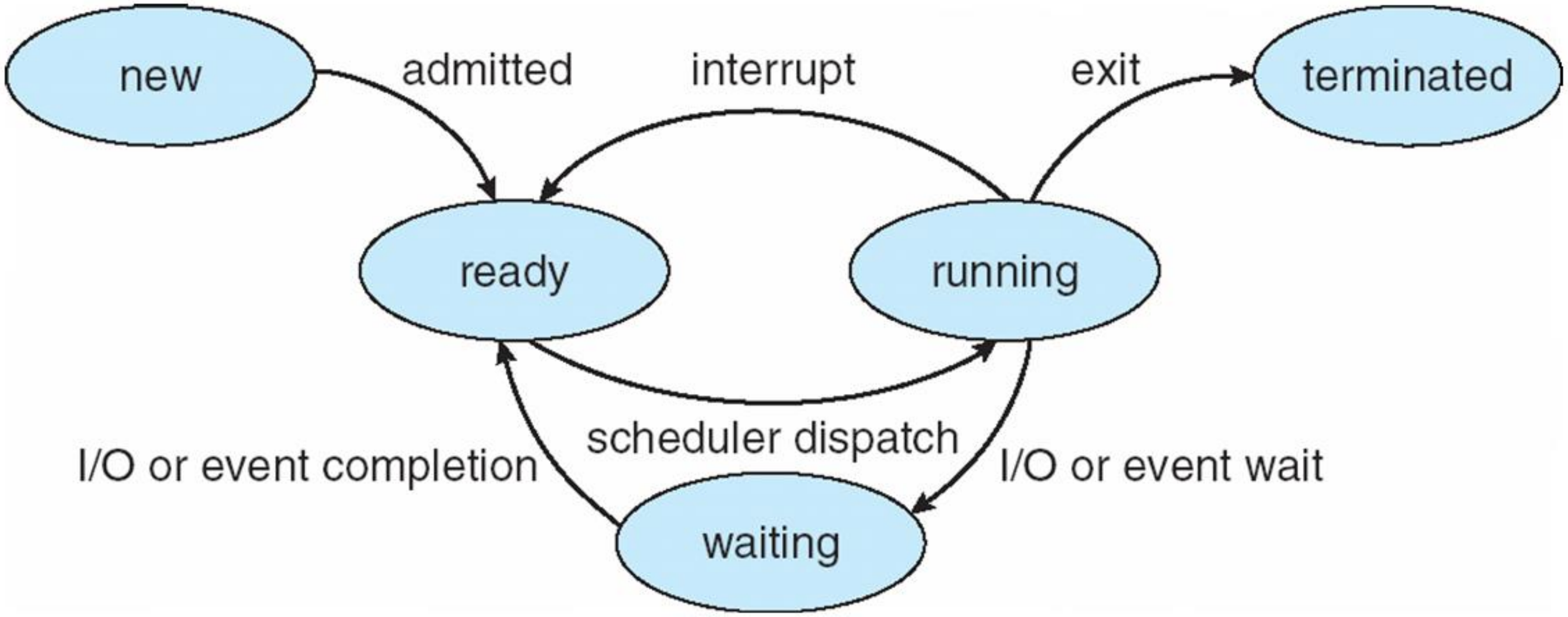


Diagram of Process State



What is Process Scheduling?

- The act of determining which process is in the ready state, and should be moved to the running state is known as **Process Scheduling**.

- The prime aim of the process scheduling system is to **keep the CPU busy all the time and to deliver minimum response time for all programs**. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

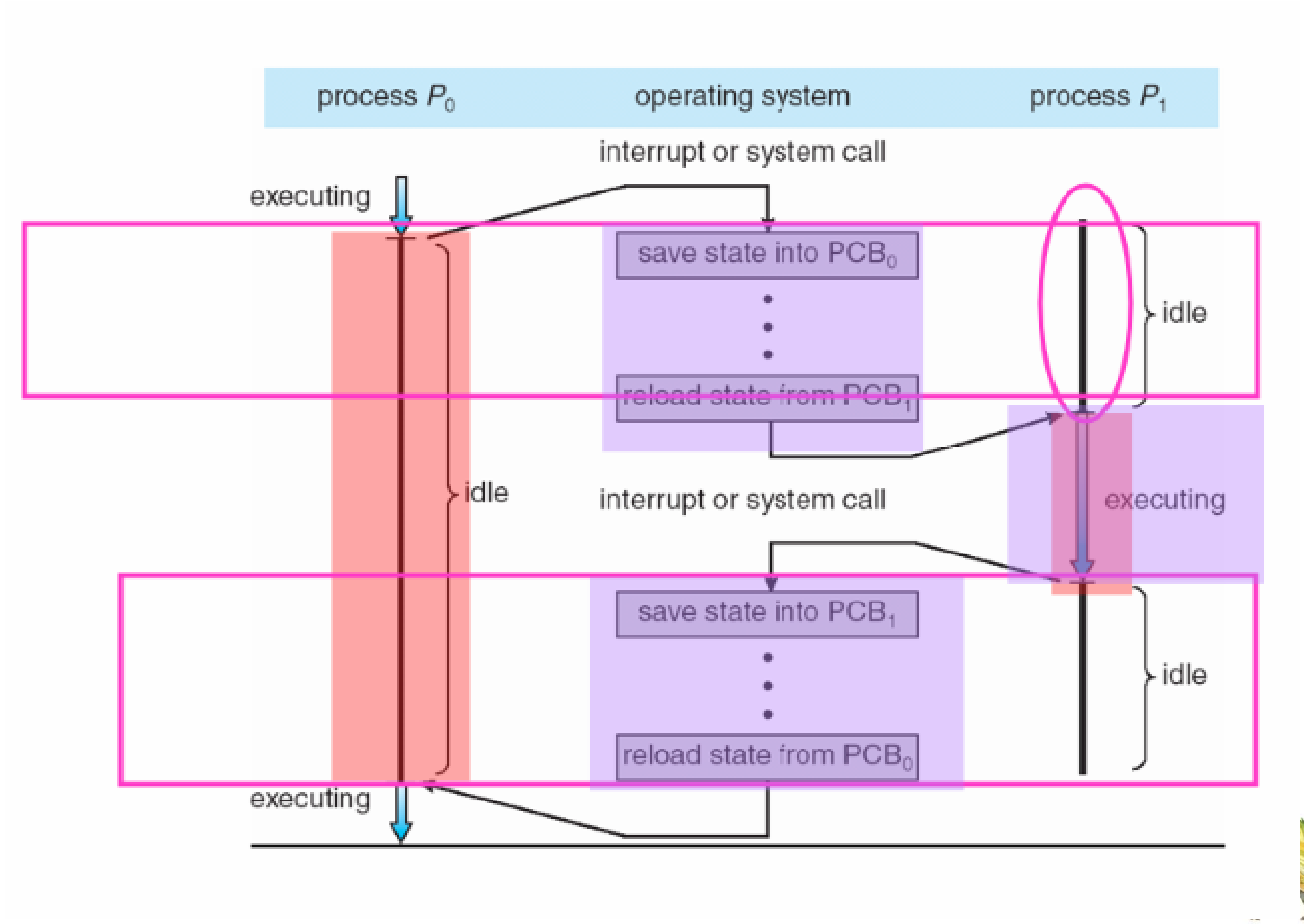
Scheduling fell into one of the two general categories:

- **Non Pre-emptive Scheduling:** When the currently executing process gives up the CPU voluntarily.
- **Pre-emptive Scheduling:** When the operating system decides to favour another process, pre-empting the currently executing process.

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.

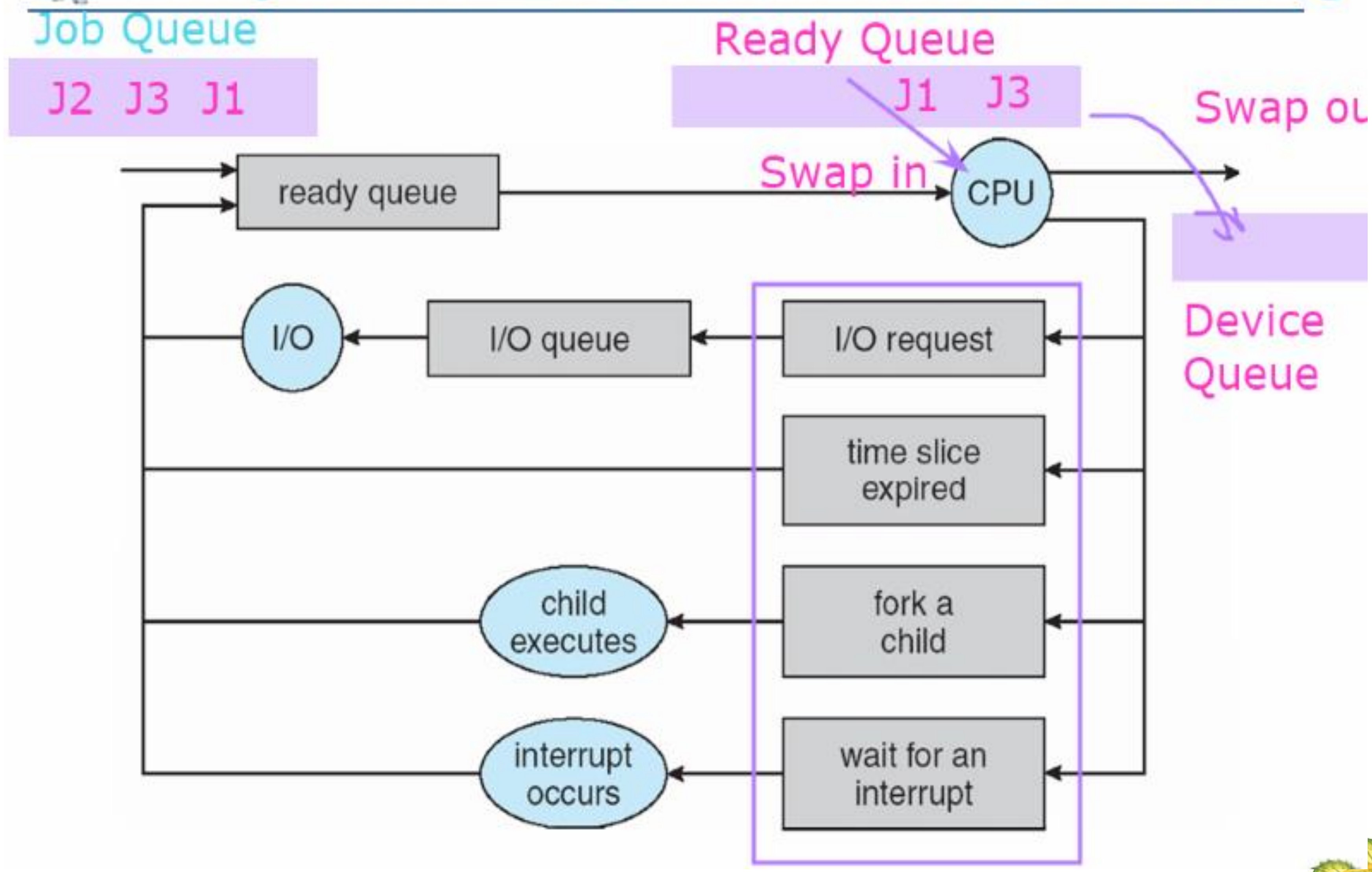
CPU Switch From Process to Process



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

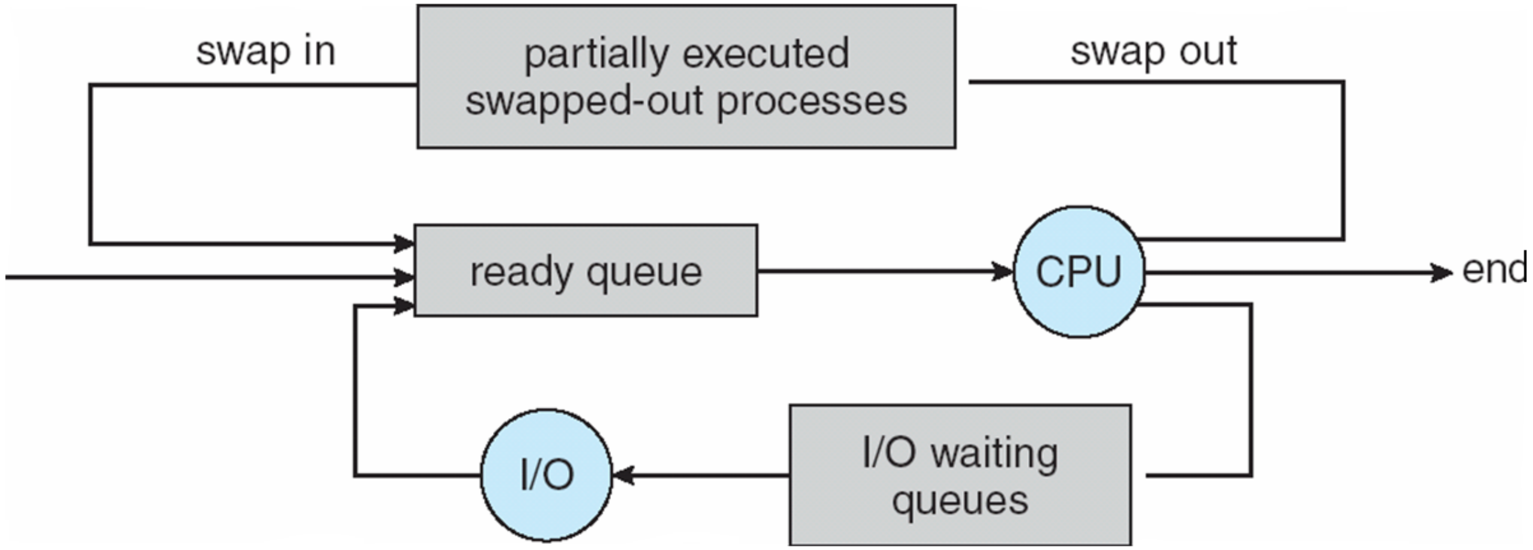


Types of Schedulers

- There are three types of schedulers available:

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

Addition of Medium Term Scheduling



Schedulers (Cont)

- Short-term scheduler **is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)**
- Long-term scheduler is **invoked very 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

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

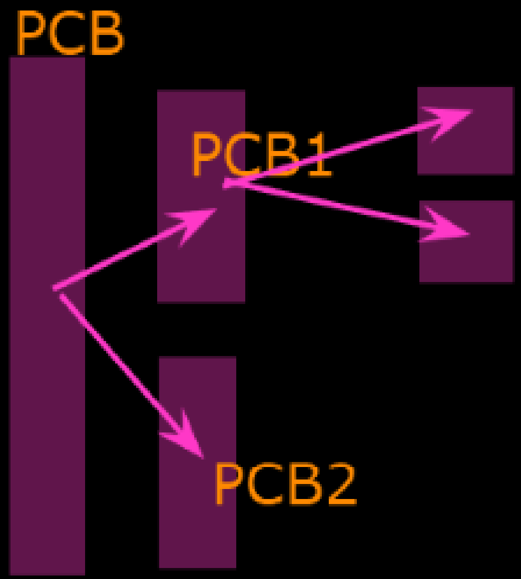
Operations on Process

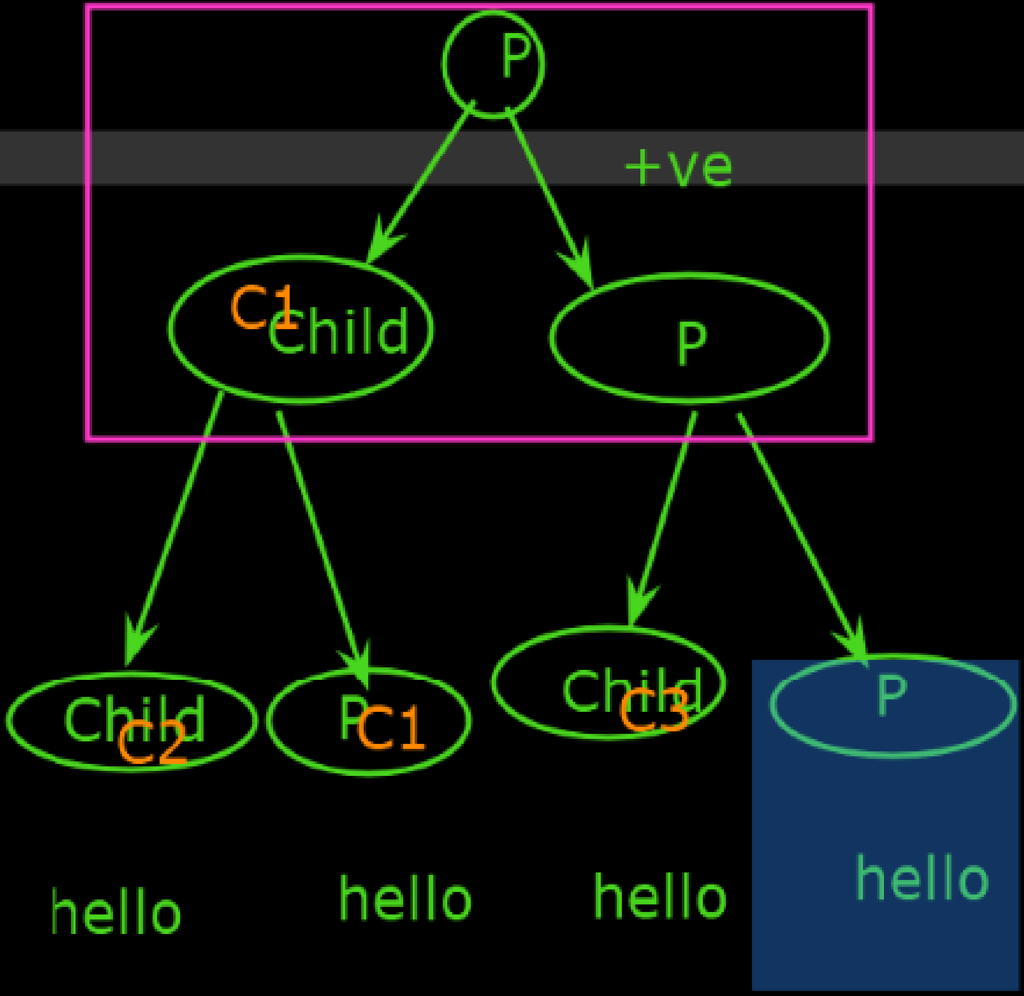
- Below we have discussed the two major operation **Process Creation** and **Process Termination**.
- **Process Creation**
- Through appropriate system calls, such as **fork or spawn**, processes may create other processes.
- The process which creates other process, is termed **the parent of the other process**, while the **created sub-process** is termed its **child.**
- Each process is given an integer identifier, termed as process identifier, or PID.
- **The parent PID (PPID**) is also stored for each process.
- On a typical UNIX systems the process scheduler is termed as sched, and is given PID 0. The first thing done by it at system start-up time is to launch init, which gives that process PID 1. Further Init launches all the system daemons and user logins, and becomes the ultimate parent of all other processes.

```
#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.

```
fork():
--------
-create a child process
    -   0 : child
    -   1 : parent
```





fork() = 2^n
fork() for child = 2^n-1

Process Termination

- By making **the exit(system call),** typically returning an int, processes may request their own termination. This int is passed along to the parent if it is doing a wait(), and is typically zero on successful completion and some non-zero code in the event of any problem.

- **Processes may also be terminated by the system for a variety of reasons, including :**
- The inability of the system to deliver the necessary system resources.
- In response to a KILL command or other unhandled process interrupts.
- A parent may kill its children if the task assigned to them is no longer needed i.e. if the need of having a child terminates.

- The processes which are trying to terminate but cannot do so because their parent is not waiting for them are **termed zombies.** These are eventually inherited by init as orphans and killed off.

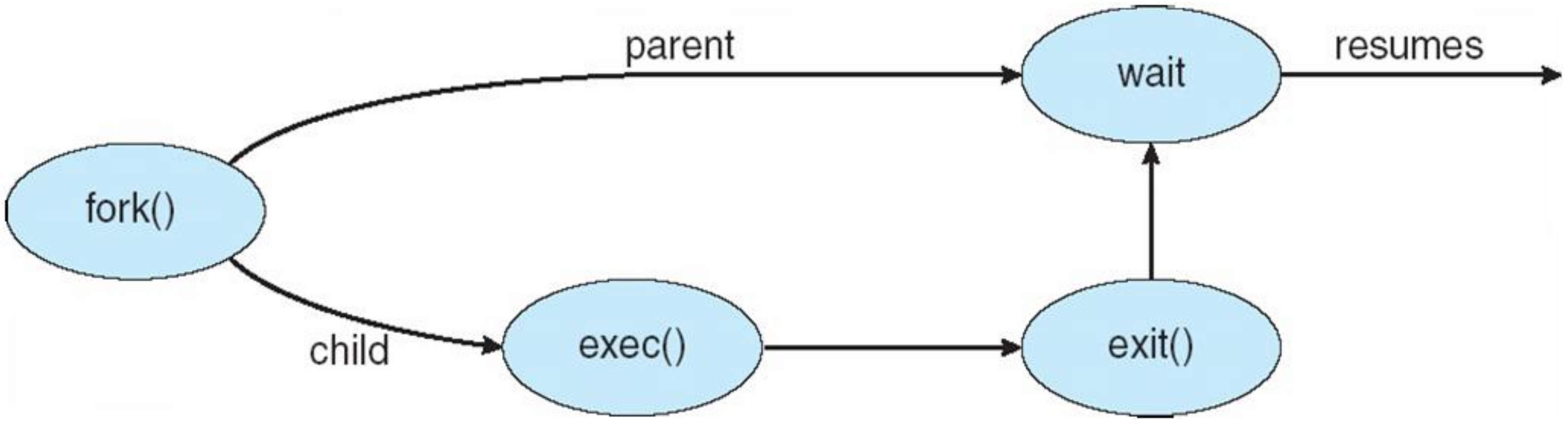
Process Creation

- **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

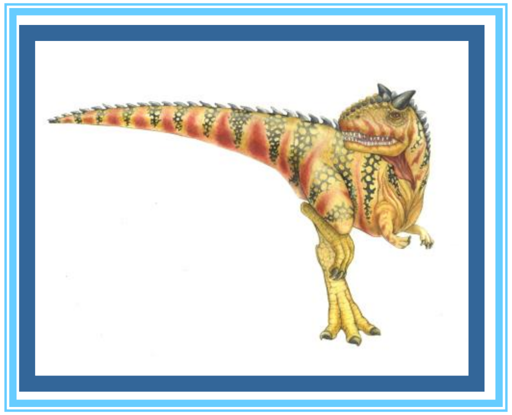
Process Creation



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**

Chapter 5: CPU Scheduling

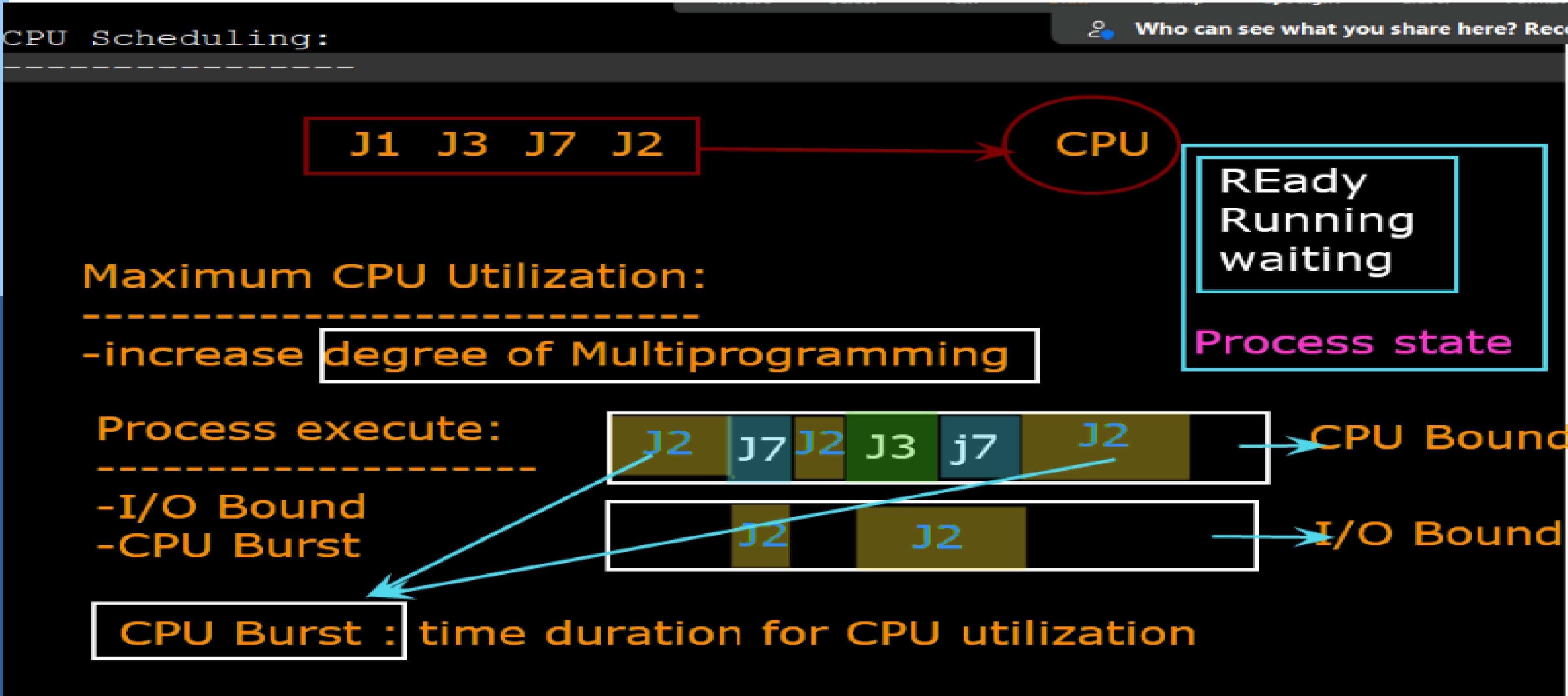


Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

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 Scheduling:

Maximum CPU Utilization:

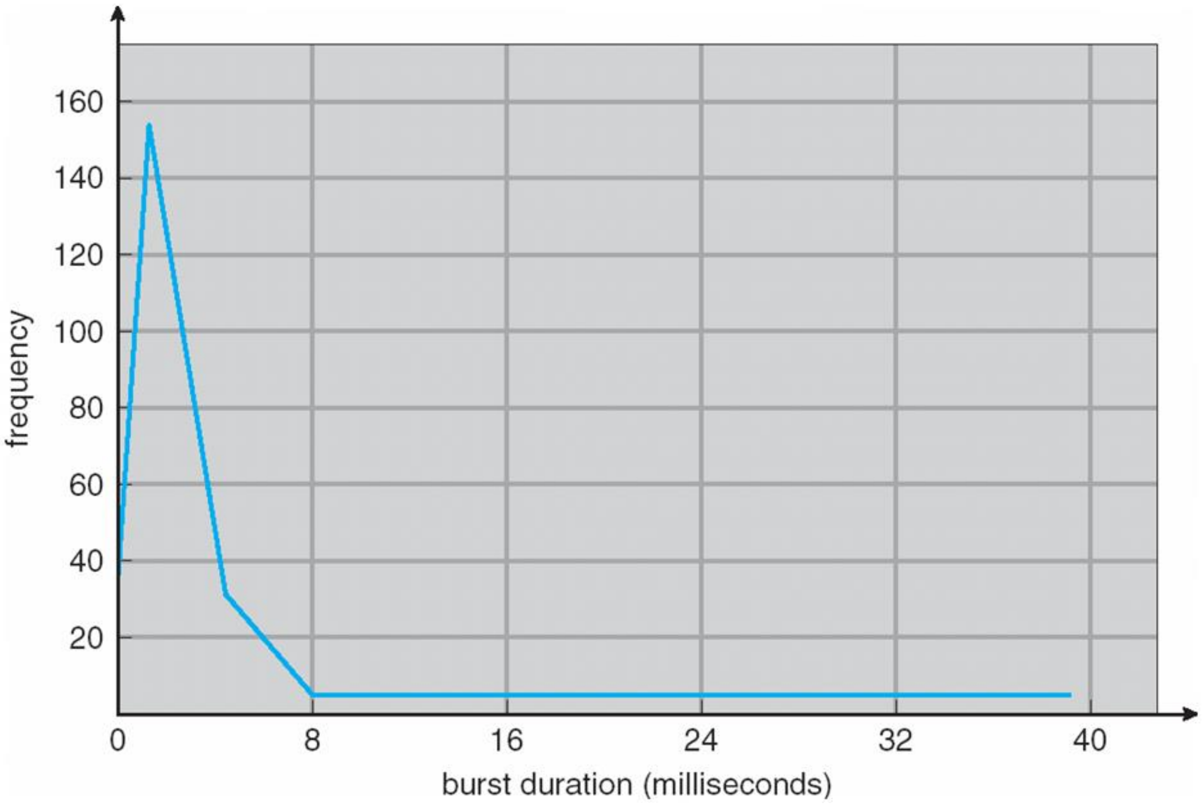
-increase degree of Multiprogramming

Process execute:

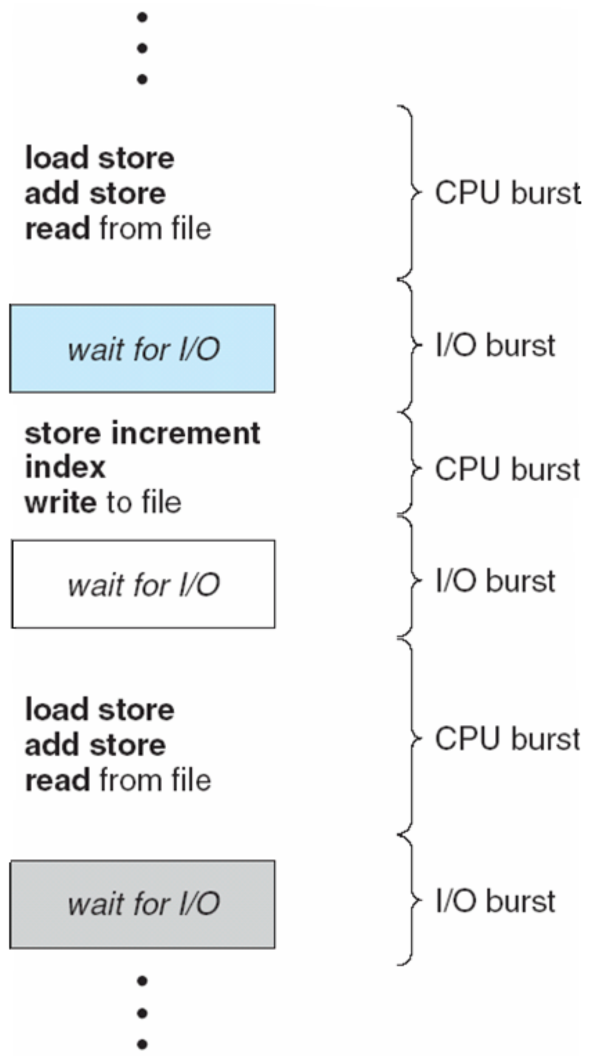
-I/O Bound
-CPU Burst

CPU Burst : time duration for CPU utilization

Histogram of CPU-burst Times



Alternating Sequence of CPU And I/O Bursts



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

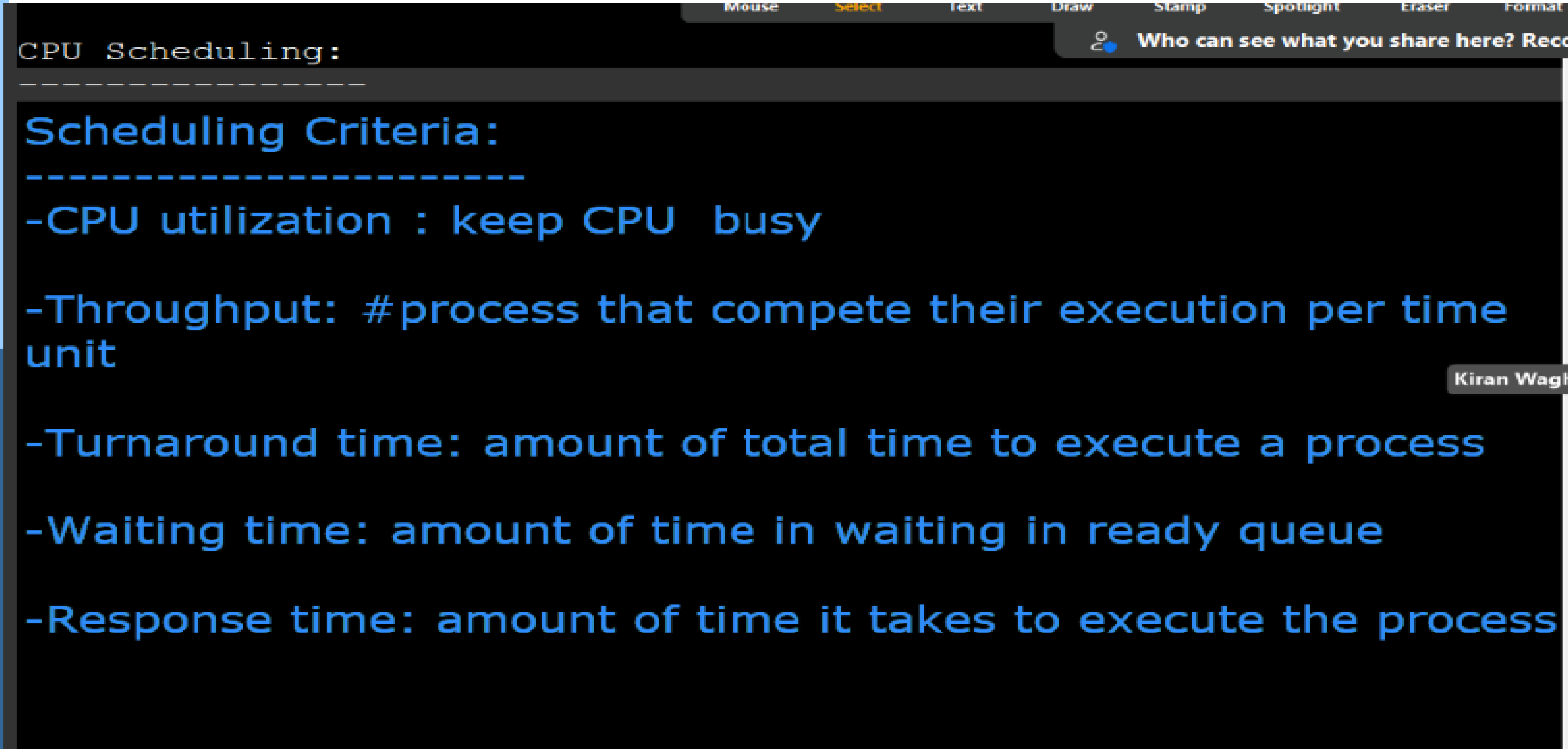
- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to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

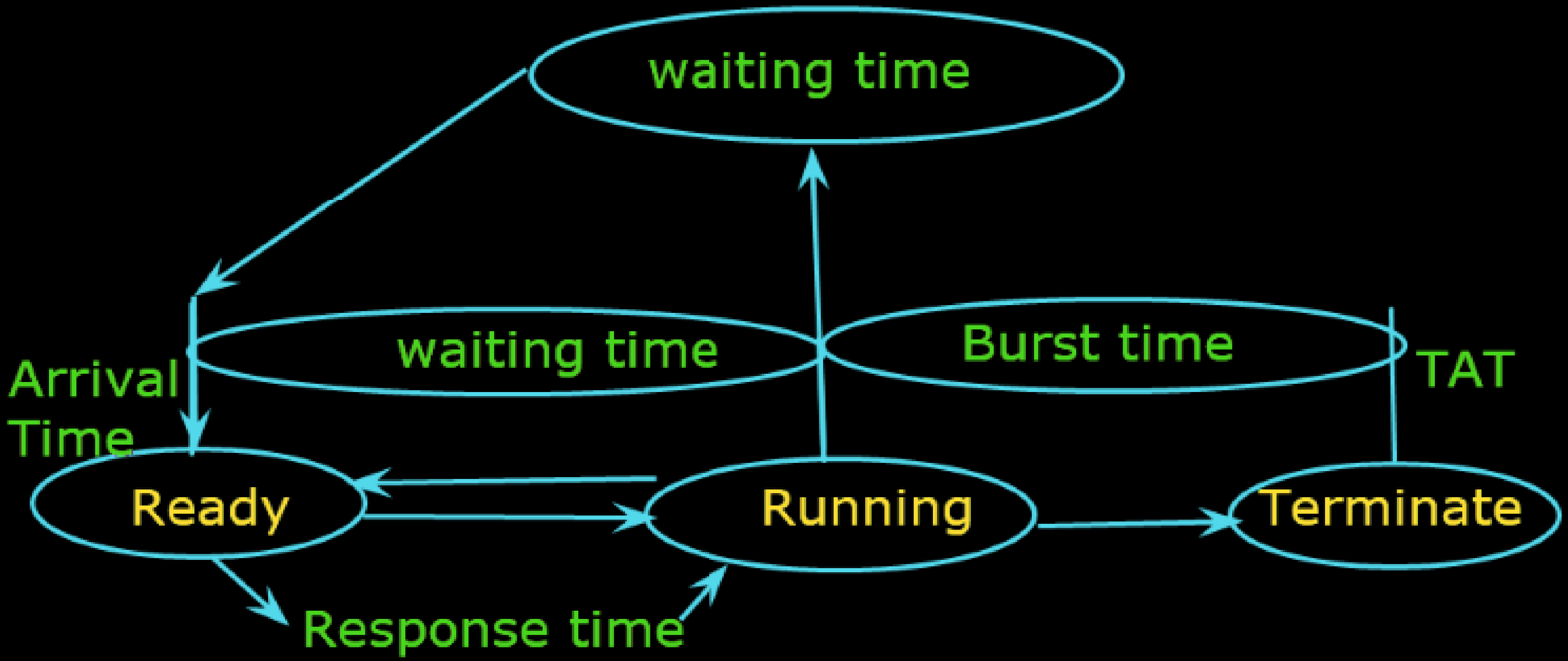
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time



```
CPU Scheduling:
------------------
Scheduling Criteria:
-----------------------
-CPU utilization : keep CPU  busy

-Throughput: #process that compete their execution per time
unit

-Turnaround time: amount of total time to execute a process

-Waiting time: amount of time in waiting in ready queue

-Response time: amount of time it takes to execute the process
```

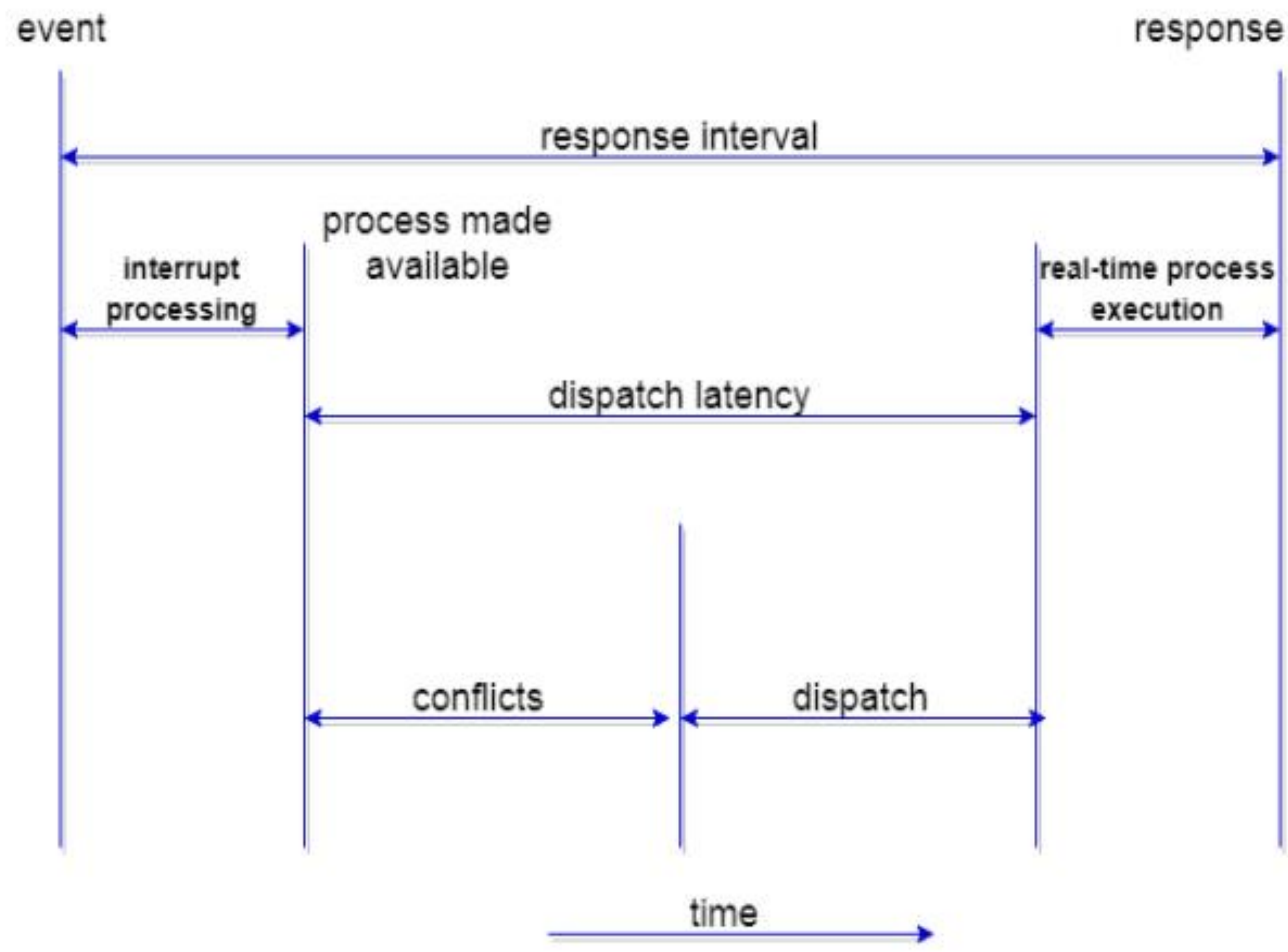
```
CPU Scheduling:
-----------------
```

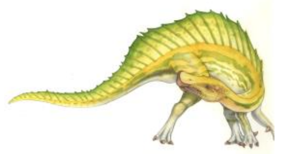
TAT = CT - AT
TAT = WT + BT
WT = TAT - BT
RT = BT - AT



CPU Scheduling: Dispatcher

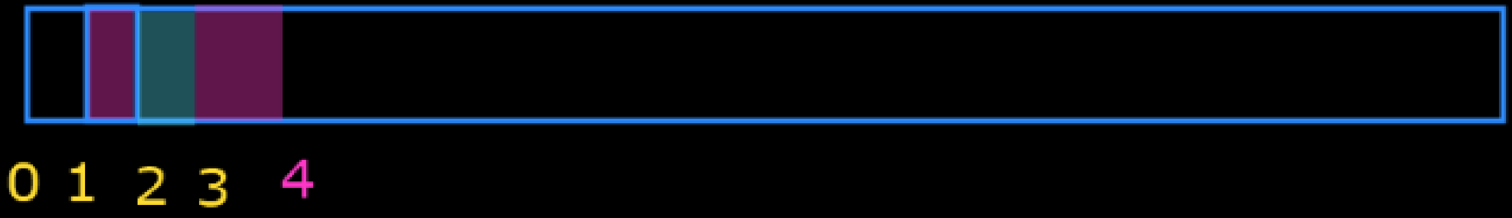
- Another component involved in the CPU scheduling function is the **Dispatcher**.
- The dispatcher is the module that gives control of the CPU to the process selected by the **short-term scheduler**. This function involves:
  - Switching context
  - Switching to user mode
- Jumping to the proper location in the user program to restart that program from where it left last time.
- The dispatcher should be as fast as possible, given that it is invoked during every process switch.
- The time taken by the dispatcher to stop one process and start another process is known as the **Dispatch Latency**.
- Dispatch Latency can be explained using the below figure:


event
response to event
response interval
process made available
interrupt processing
real-time process execution
dispatch latency
conflicts
dispatch
time

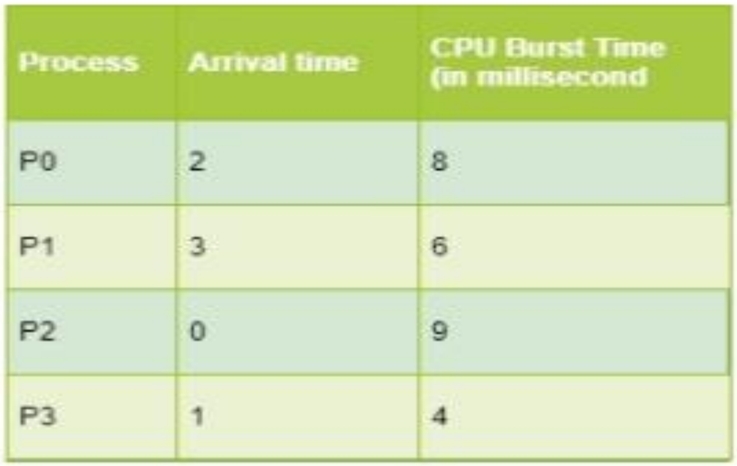


Process	CB	Prio	AT
P1	21	3	5
P2	3	2	3
P3	4	1	4
~~P4~~	~~1~~	~~1~~	~~2~~
P5	~~6~~ 4	1	1

```
CPU Scheduling:
----------------

Non-Preemptive Algorithms:
----------------------------
    -FCFS : First Come First Serve
    -SJF  : shortest Job First
    -LJF  : Longest Job First
    -HRRN : Highest Response Ratio Next
    -Multi level Queue
    -Priority

Preemptive Algorithms:
-----------------------
    -SRTF : Shortest Remaining Time First
    -LRTF : Longest REmaining Time First
    -Round Robin
    -Priority based
```



-P5-P4-P5-



Process	Arrival time	CPU Burst Time (in millisecond
P0	2	8
P1	3	6
P2	0	9
P3	1	4

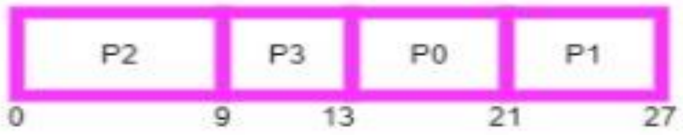
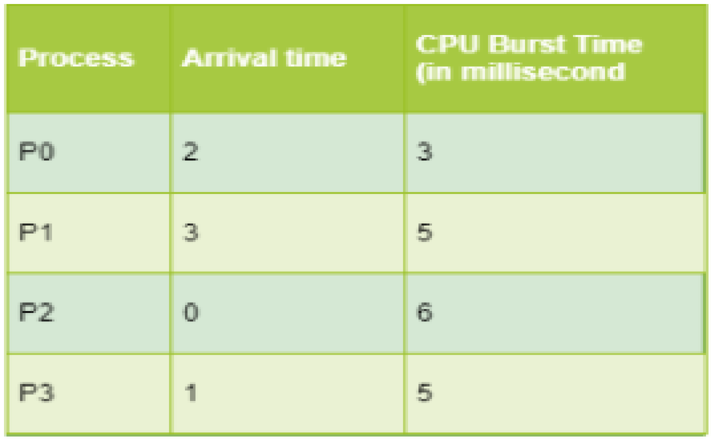


Figure: Non-Preemptive Scheduling



Process	Arrival time	CPU Burst Time (in millisecond
P0	2	3
P1	3	5
P2	0	6
P3	1	5

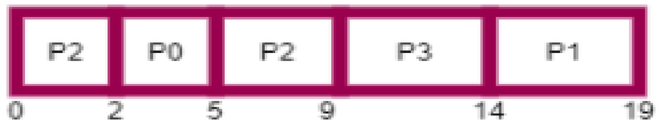


Figure: Preemptive Scheduling

CPU Scheduling: Scheduling Criteria

- There are many different criteria to check when considering the **"best"** scheduling algorithm, they are:
- **CPU Utilization**
- To make out the best use of the CPU and not to waste any CPU cycle, the CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
- **Throughput**
- It is the total number of processes completed per unit of time or rather says the total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.
- **Turnaround Time**
- It is the amount of time taken to execute a particular process, i.e. The interval from the time of submission of the process to the time of completion of the process(Wall clock time).
- **Waiting Time**
- The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.
- **Load Average**
- It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.
- **Response Time**
- Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).
- In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Scheduling Algorithms

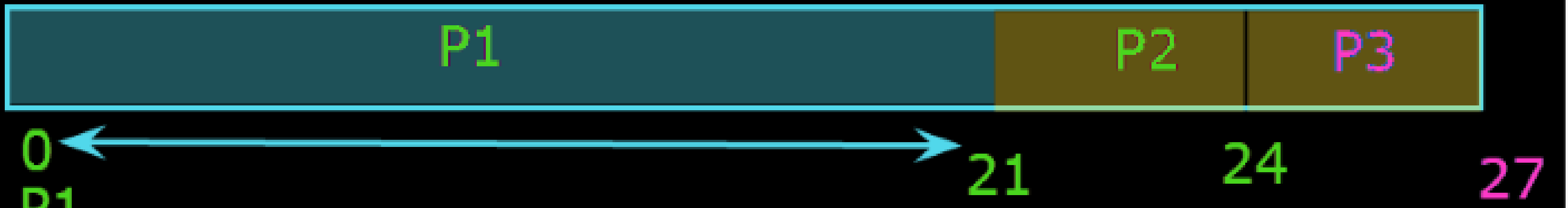
- First Come First Serve(FCFS) Scheduling
- Shortest-Job-First(SJF) Scheduling
- Priority Scheduling
- Round Robin(RR) Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling
- Shortest Remaining Time First (SRTF)
- Longest Remaining Time First (LRTF)
- Highest Response Ratio Next (HRRN)

- Here we have simple formulae for calculating various times for given processes:
- **Completion Time:** Time taken for the execution to complete, starting from arrival time.
- **Turn Around Time:** Time taken to complete after arrival. In simple words, it is the difference between the Completion time and the Arrival time.
- **Waiting Time:** Total time the process has to wait before it's execution begins. It is the difference between the Turn Around time and the Burst time of the process.
- For the program above, we have considered the arrival time to be 0 for all the processes, try to implement a program with variable arrival times.

First Come First Serve

Process	BT	CT	TAT	WT	RT
P1	21	21	21	0	0
P2	3	24	24	21	21
P3	3	27	27	24	24

Sequence: P1-P2-P3



P1
P2
P3

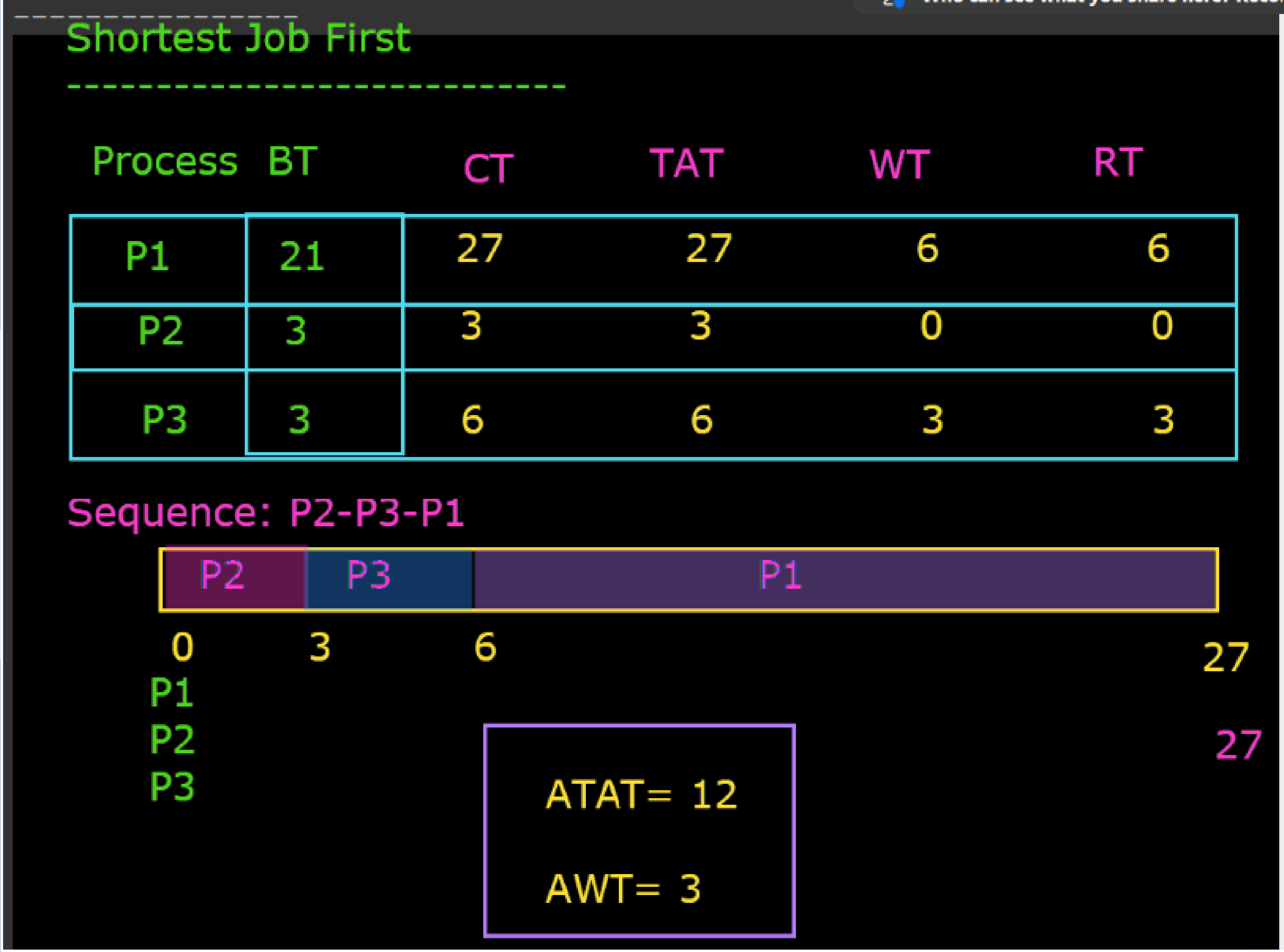
AWT=(0+21+24)/3=15
ATAT=(21+24+27)/3 = 24

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

Shortest Job First(SJF) Scheduling

- Shortest Job First scheduling works on the process with the shortest burst time or duration first.
- This is the best approach to minimize waiting time.
- This is used in Batch Systems.
- It is of two types:
  - Non Pre-emptive
  - Pre-emptive
- To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time.
- This scheduling algorithm is optimal if all the jobs/processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all)



Shortest Job First

Process	BT	CT	TAT	WT	RT
P1	21	27	27	6	6
P2	3	3	3	0	0
P3	3	6	6	3	3

Sequence: P2-P3-P1

P2	P3	P1

0 3 6 27

P1
P2
P3

27

ATAT= 12

AWT= 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

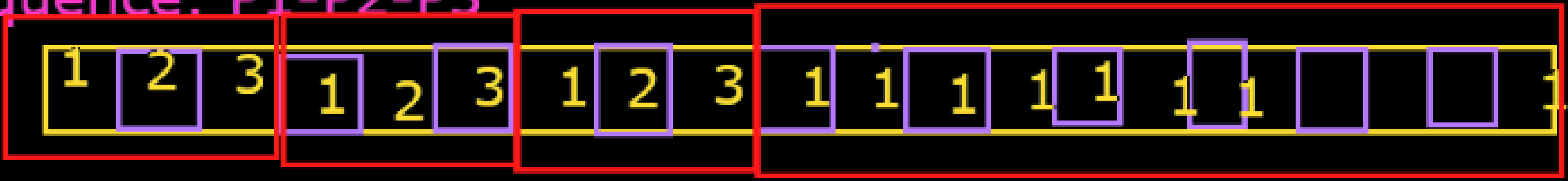
Round Robin Scheduling

- Round Robin(RR) scheduling algorithm is mainly **designed for time-sharing systems**. This algorithm is similar to FCFS scheduling, but in Round Robin(RR) scheduling, preemption is added which enables the system to switch between processes.
- A fixed time is allotted to each process, called a **quantum**, for execution.
- Once a process is executed for the given time period that process is preempted and another process executes for the given time period.
- **Context switching is used to save states of preempted** processes.
- This algorithm is simple and easy to implement and the most important is thing is this **algorithm is starvation-free** as all processes get a fair share of CPU.
- It is important to note here that the length of time quantum is generally from 10 to 100 milliseconds in length.

Round Robin: time slice (quantum) = 1

Process	BT	CT	TAT	WT	RT
P1	21	27	27	6	6
P2	~~3~~	3	3	0	0
P3	~~3~~	6	6	3	3

Sequence: P1-P2-P3



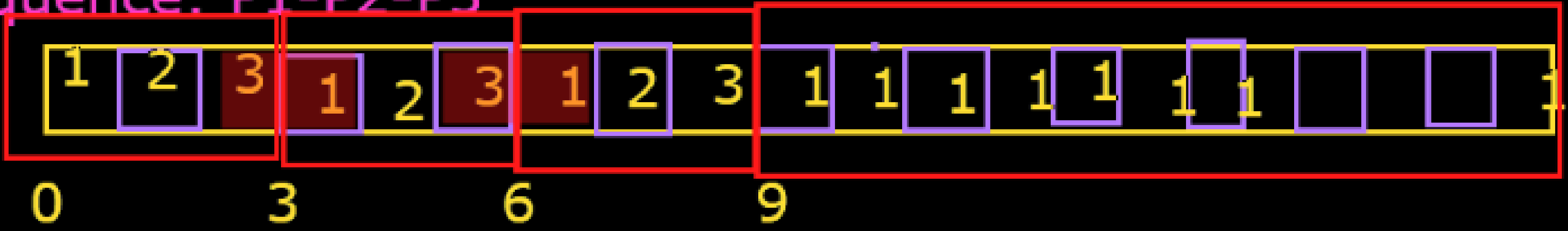
P1
P2
P3

Quantum=2

Round Robin: time slice (quantum) = 1

Process	BT	CT	TAT	WT	RT
P1	21	27	27	6	0
P2	~~3~~	8	8	5	1
P3	~~3~~	9	9	6	2

Sequence: P1-P2-P3



Some important characteristics of the Round Robin(RR) Algorithm are as follows:

1. Round Robin Scheduling algorithm resides under the category of Preemptive Algorithms.
2. This algorithm is one of the oldest, easiest, and fairest algorithm.
3. This Algorithm is a real-time algorithm because it responds to the event within a specific time limit.
4. In this algorithm, the time slice should be the minimum that is assigned to a specific task that needs to be processed. Though it may vary for different operating systems.
5. This is a hybrid model and is clock-driven in nature.
6. This is a widely used scheduling method in the traditional operating system.

Important terms

1. **Completion Time** It is the time at which any process completes its execution.
2. **Turn Around Time** This mainly indicates the time Difference between completion time and arrival time. The Formula to calculate the same is: **Turn Around Time = Completion Time – Arrival Time**
3. **Waiting Time(W.T):** It Indicates the time Difference between turn around time and burst time. And is calculated as **Waiting Time = Turn Around Time – Burst Time**

Advantages of Round Robin Scheduling Algorithm

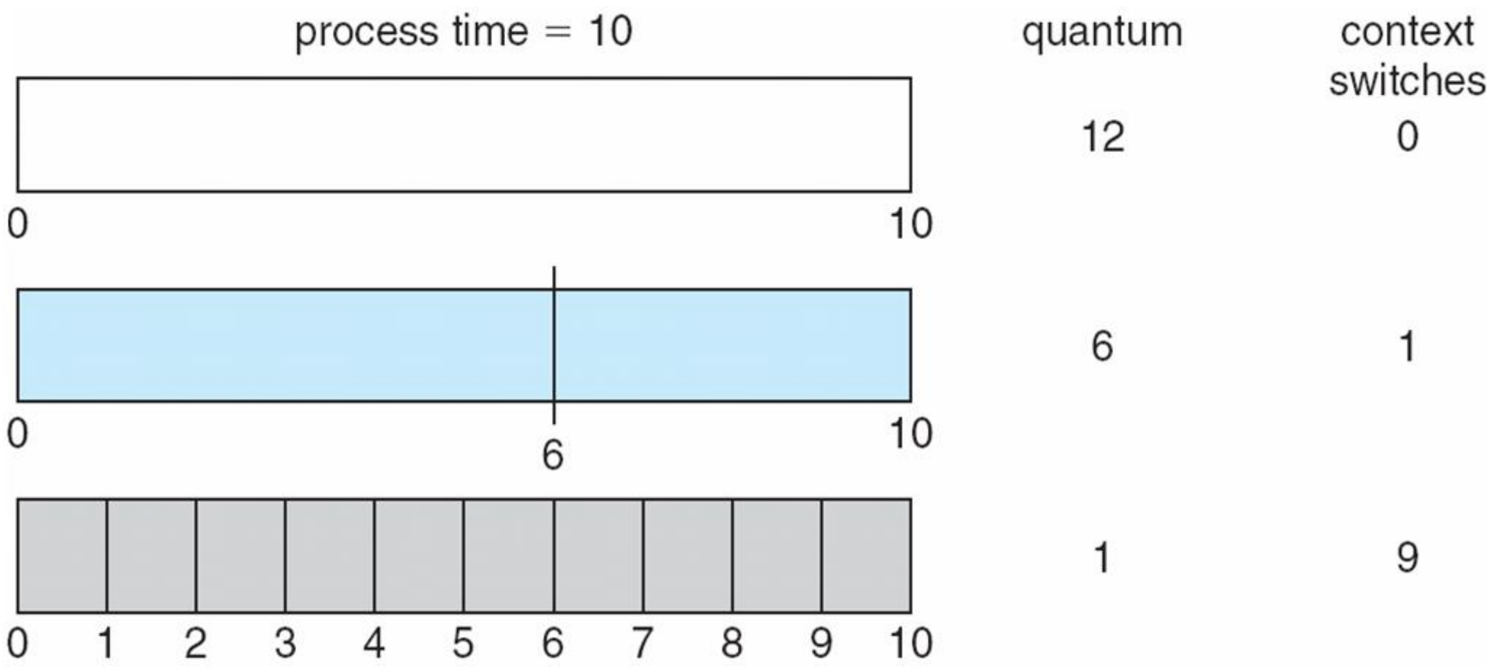
- Some advantages of the Round Robin scheduling algorithm are as follows:
- While performing this scheduling algorithm, a **particular time quantum is allocated** to different jobs.
- In terms of average response time, this **algorithm gives the best performance.**
- With the help of this algorithm, **all the jobs get a fair allocation** of CPU.
- In this algorithm, there are **no issues of starvation or convoy effect**.
- This algorithm deals with all processes without any priority.
- This algorithm is **cyclic in nature.**
- In this, the newly created process is added to the end of the ready queue.
- Also, in this, **a round-robin scheduler generally employs time-sharing** which means providing each job a time slot or quantum.
- In this scheduling algorithm, each process gets a chance to reschedule after a particular quantum time.

Disadvantages of Round Robin Scheduling Algorithm

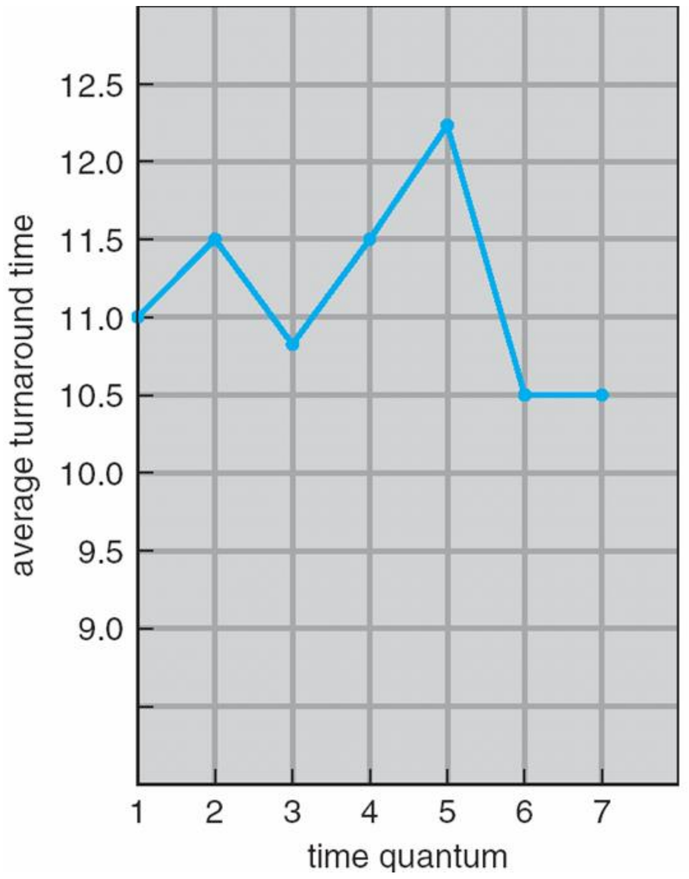
- Some disadvantages of the Round Robin scheduling algorithm are as follows:

- This algorithm spends **more time on context switches**.
- For **small quantum**, it is time-consuming scheduling.
- This algorithm **offers a larger waiting time and response time**.
- In this, there is **low throughput.**
- If time quantum is less for scheduling then its Gantt chart seems to be too big.

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

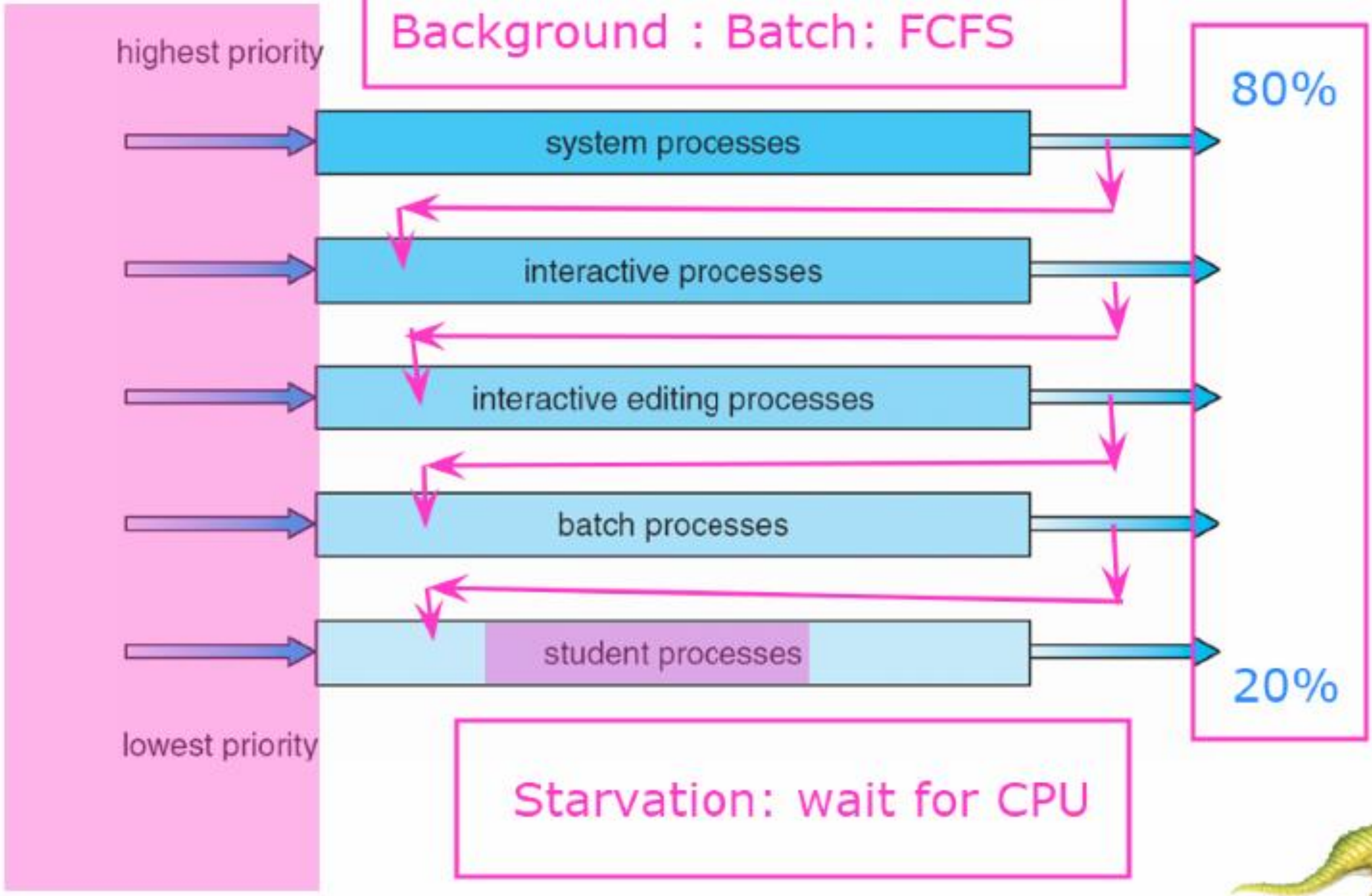
Multilevel Queue

- 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

Foreground : interactive: RR

Background : Batch: FCFS



Starvation: wait for CPU

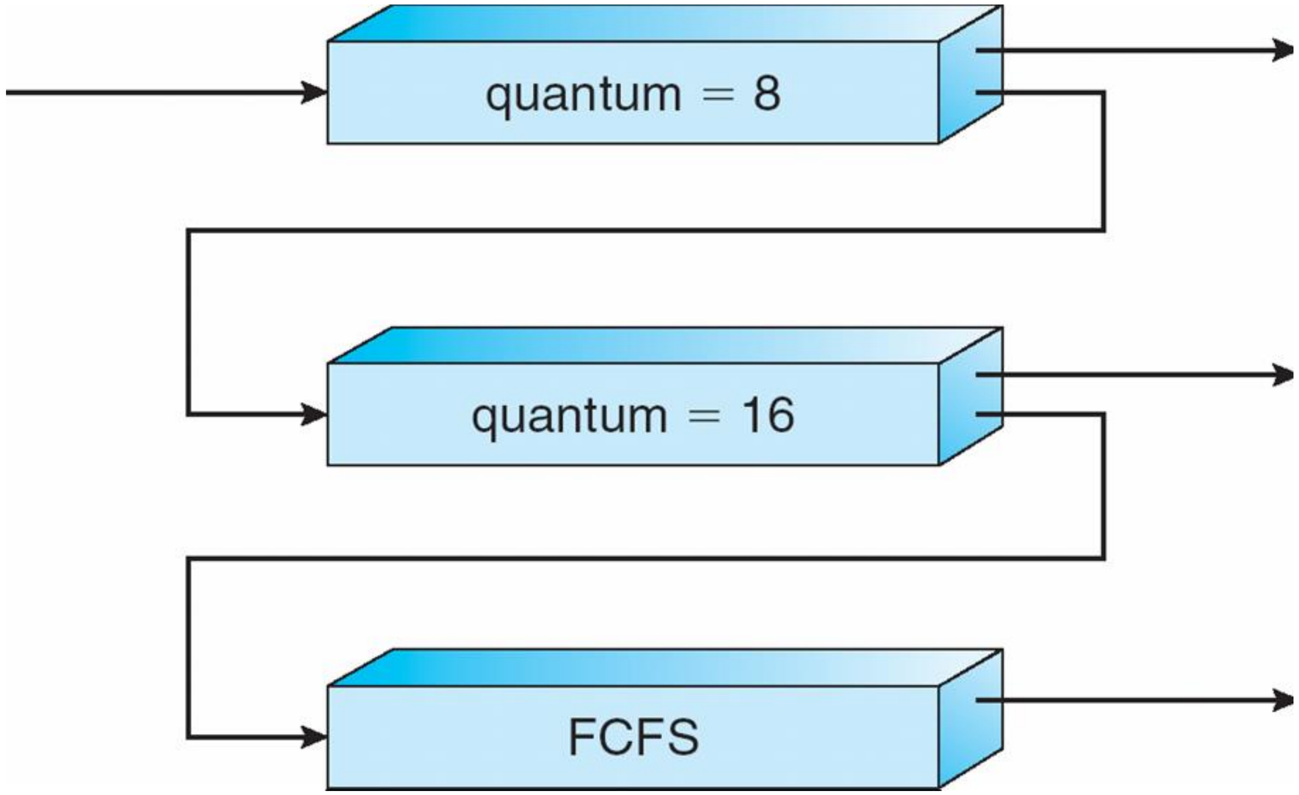
Multilevel Feedback Queue

- 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

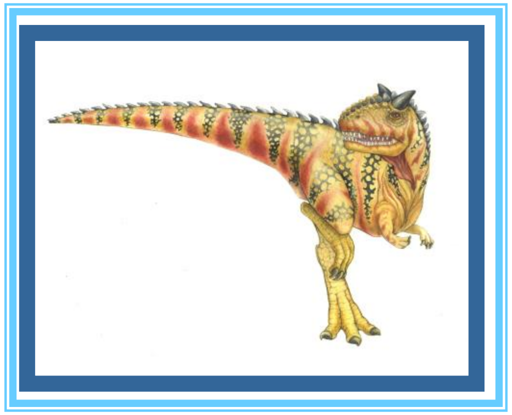
Example of Multilevel Feedback Queue

- 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



Chapter 6: Process Synchronization



Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Process Synchronization

- In this tutorial, we will be covering the concept of Process synchronization in an Operating System.
- Process Synchronization was introduced to handle problems that arose while multiple process executions.
- Process is categorized into two types on the basis of synchronization and these are given below:
  - Independent Process
  - Cooperative Process

Independent Processes

- Two processes ar**Cooperative Processes**
- e said to be independent if the execution of one process does not affect the execution of another process.

Cooperative Processes

- Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

- It is the task phenomenon of coordinating the **execution of processes in such a way that no two processes can have access to the same shared data and resources.**
- It is a procedure that is involved in **order to preserve the appropriate order of execution** of cooperative processes.
- In order to **synchronize the processes, there are various synchronization mechanisms.**
- Process Synchronization is mainly needed in a multi-process system when **multiple processes are running together, and more than one processes try to gain access to the same shared resource** or any data at the same time.

Race Condition

- At the time when more than one process is either executing the same code or accessing the same memory or any shared variable;
- In that condition, there is **a possibility that the output or the value of the shared variable is wrong so for that purpose all the processes are doing the race to say that my output is correct**. This condition is commonly known as a **race condition**.
- As several processes access and process the manipulations on the same data in a concurrent manner and due to which the outcome depends on the particular order in which the access of data takes place.

Mainly this condition is a situation that may **occur inside the critical section**.

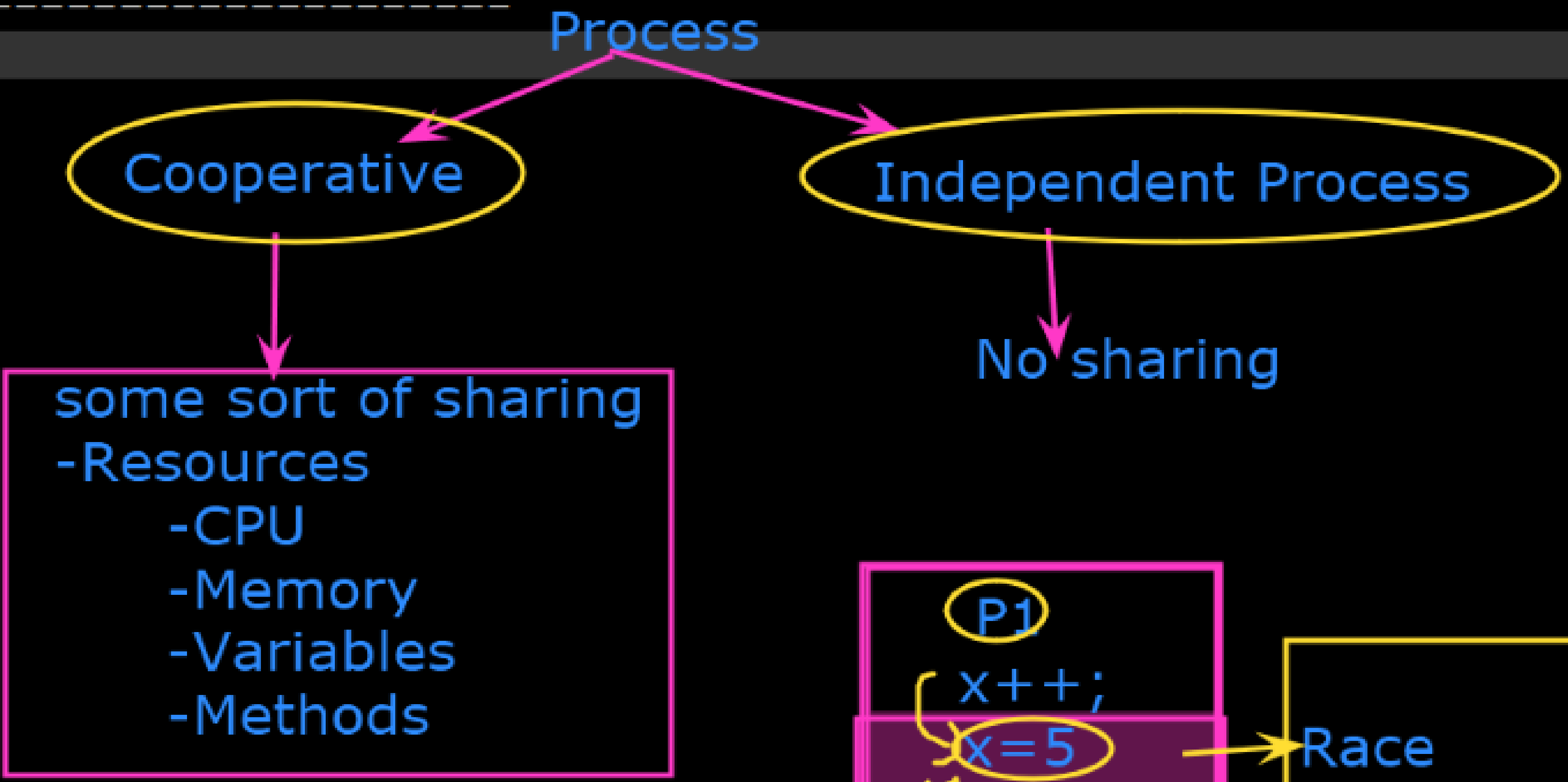
Race condition in the critical section happens when the result of multiple thread execution differs according to the order in which the threads execute.

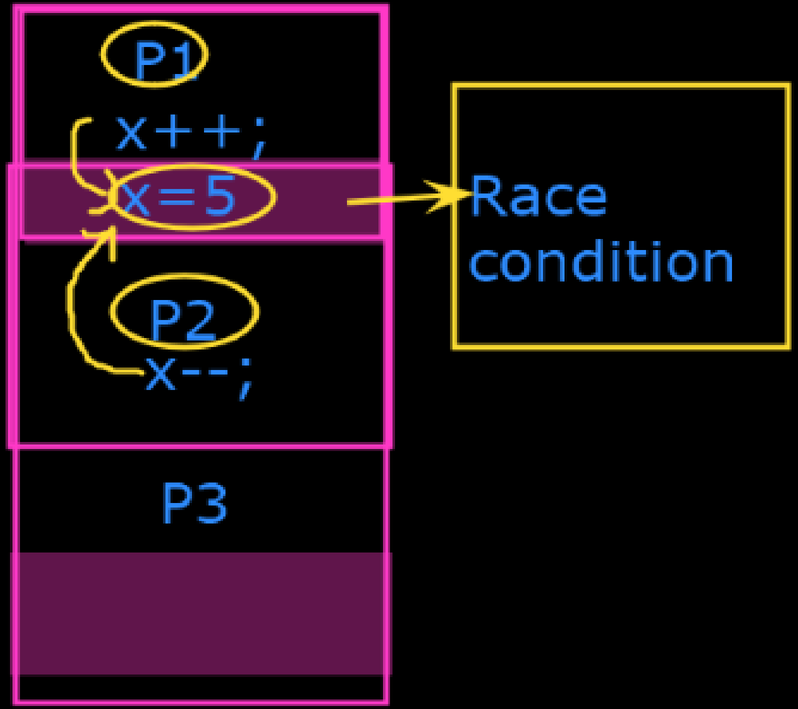
But this condition is critical sections can be avoided if the critical section is treated as an atomic instruction. Proper thread synchronization using locks or atomic variables can also prevent race conditions.

Critical Section Problem

- A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section.
- If any other process also wants to execute its critical section, it must wait until the first one finishes. The entry to the critical section is mainly handled by wait() function while the exit from the critical section is controlled by the signal() function.

```
Process Synchronization:
----------------------------
```



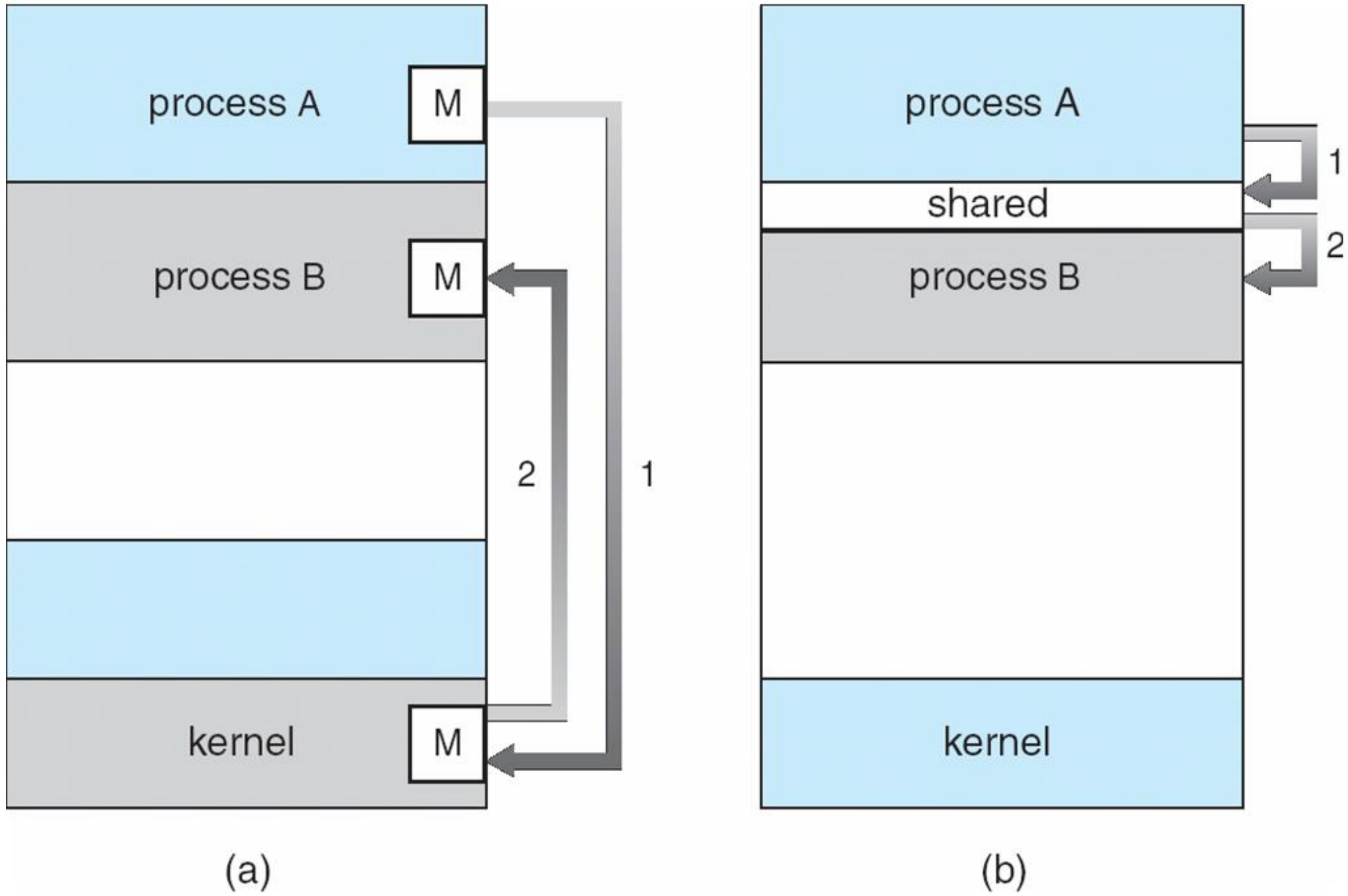


Solution:
Process Synchronization

Critical Section Problem

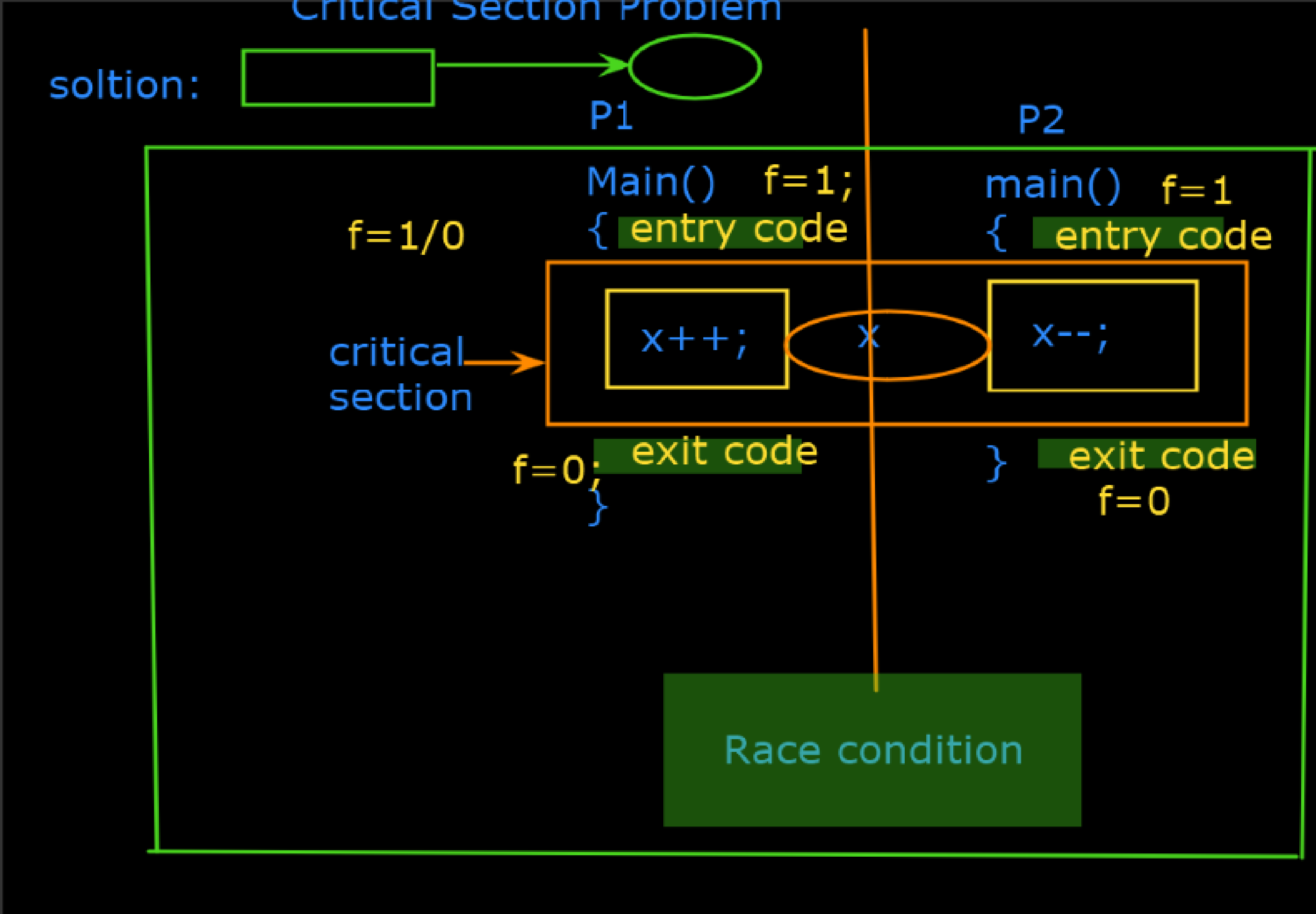
Solution:

Communications Models

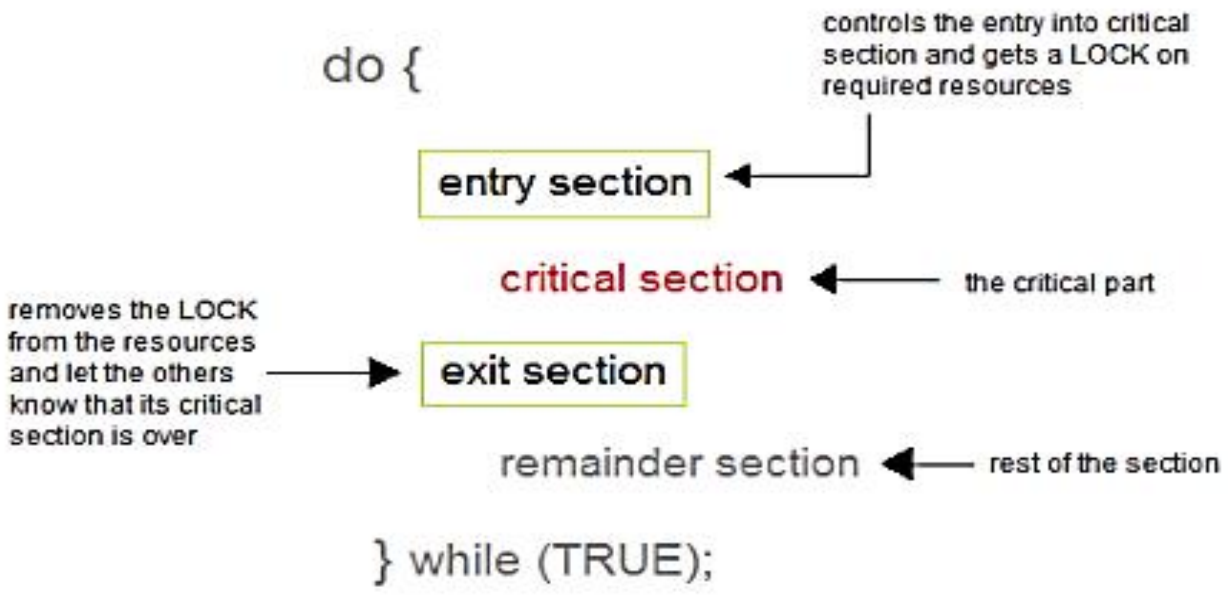


Process Synchronization:

Critical Section Problem



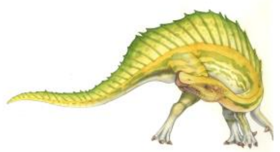
- **Entry Section**
- In this section mainly the process requests for its entry in the critical section.
- Exit Section
- This section is followed by the critical section.



do {
entry section
controls the entry into critical section and gets a LOCK on required resources
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

The solution to the Critical Section Problem

- A solution to the critical section problem must satisfy the following three conditions:
- **1. Mutual Exclusion**
- Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.
- **2. Progress**
- If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.
- **3. Bounded Waiting**
- After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.



Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is **executing in its critical section,** then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes