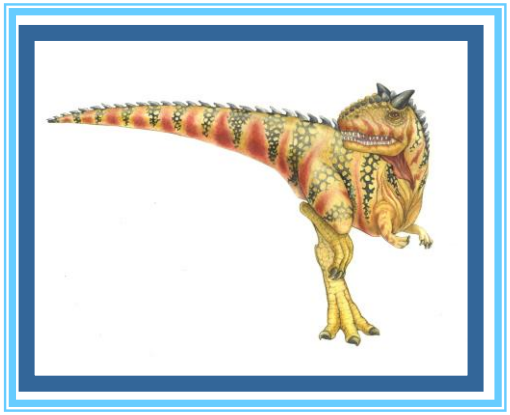
Processes
Day3: Sep 2021

Kiran Waghmare



Types of Schedulers

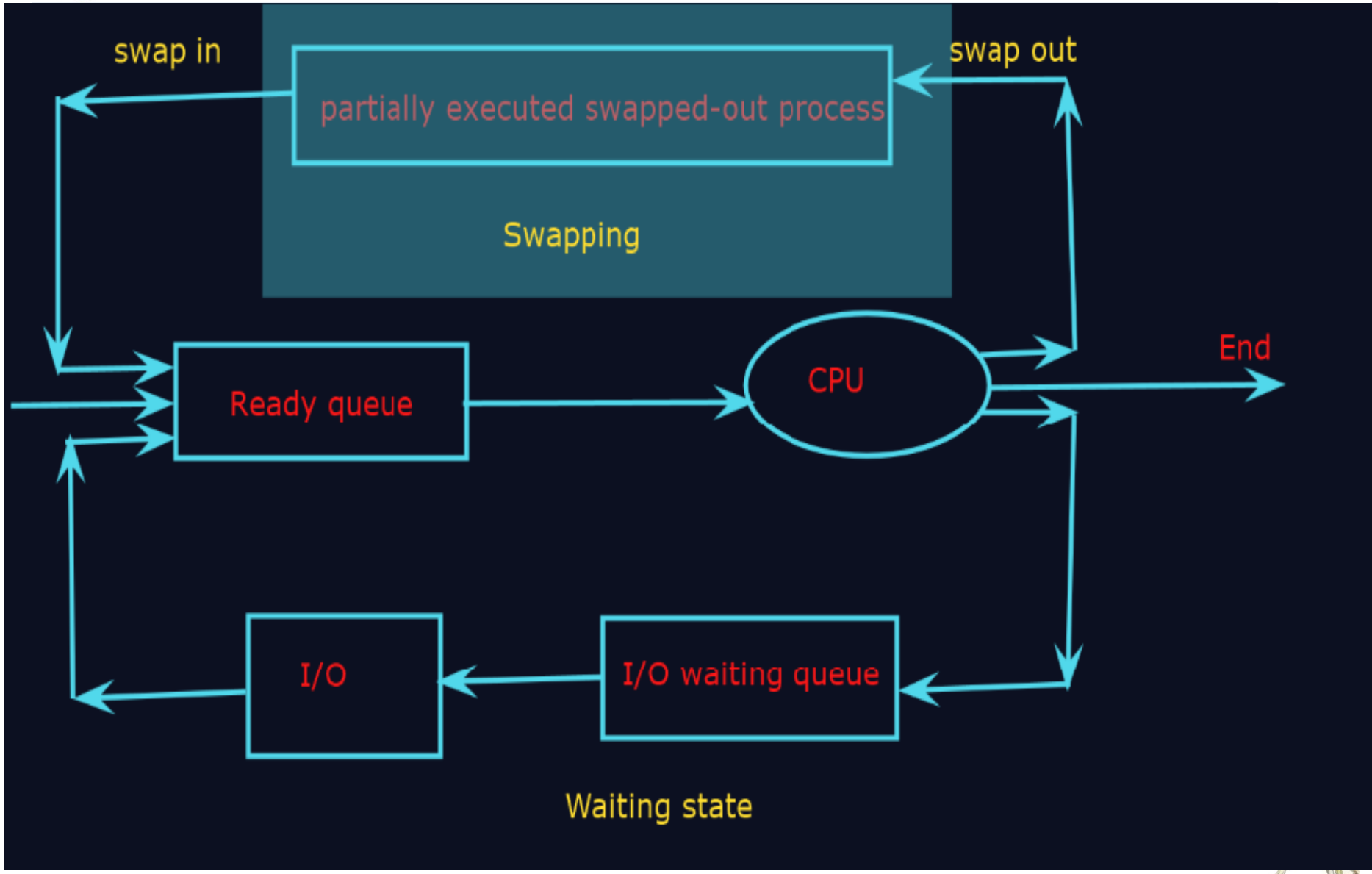
- There are three types of schedulers available:

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

Schedulers

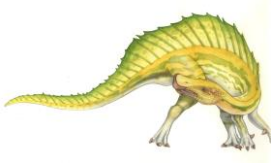
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Addition of Medium Term Scheduling



Schedulers (Cont)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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.

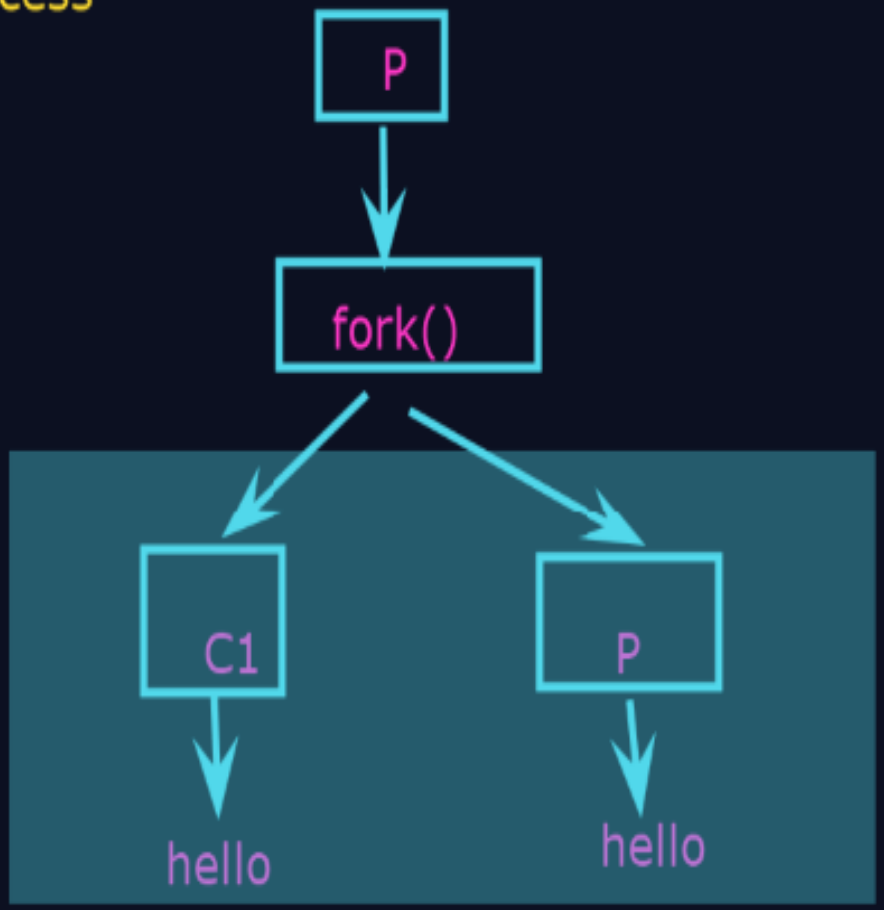
- On a typical UNIX systems the process scheduler is termed as sched, and is given PID 0. The first thing done by it at system start-up time is to launch init, which gives that process PID 1. Further Init launches all the system daemons and user logins, and becomes the ultimate parent of all other processes.

fork() → use to create a child process

0 : child
+1 : Parent
-1 : child copy

```
case 1:

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



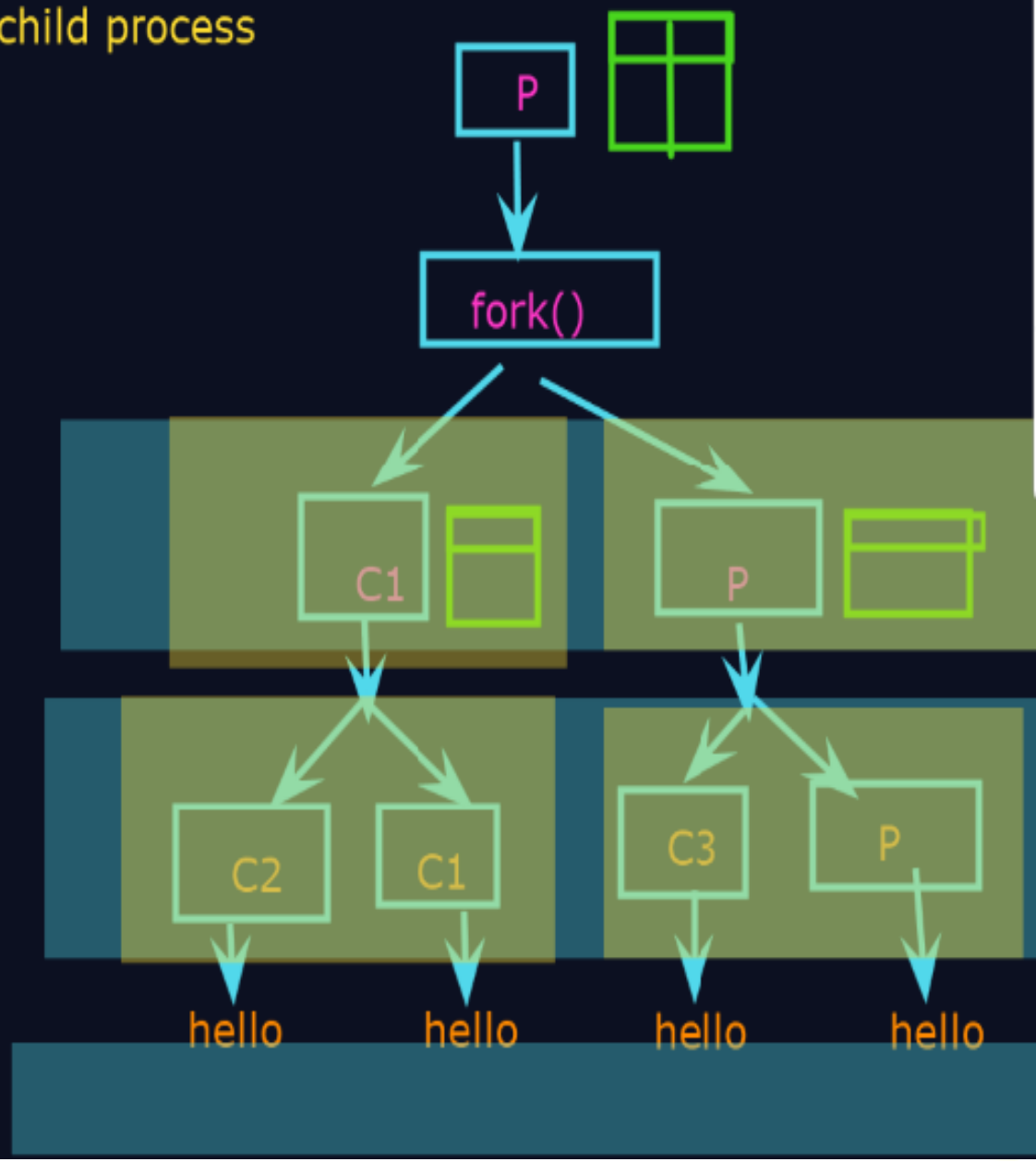
fork() → use to create a child process

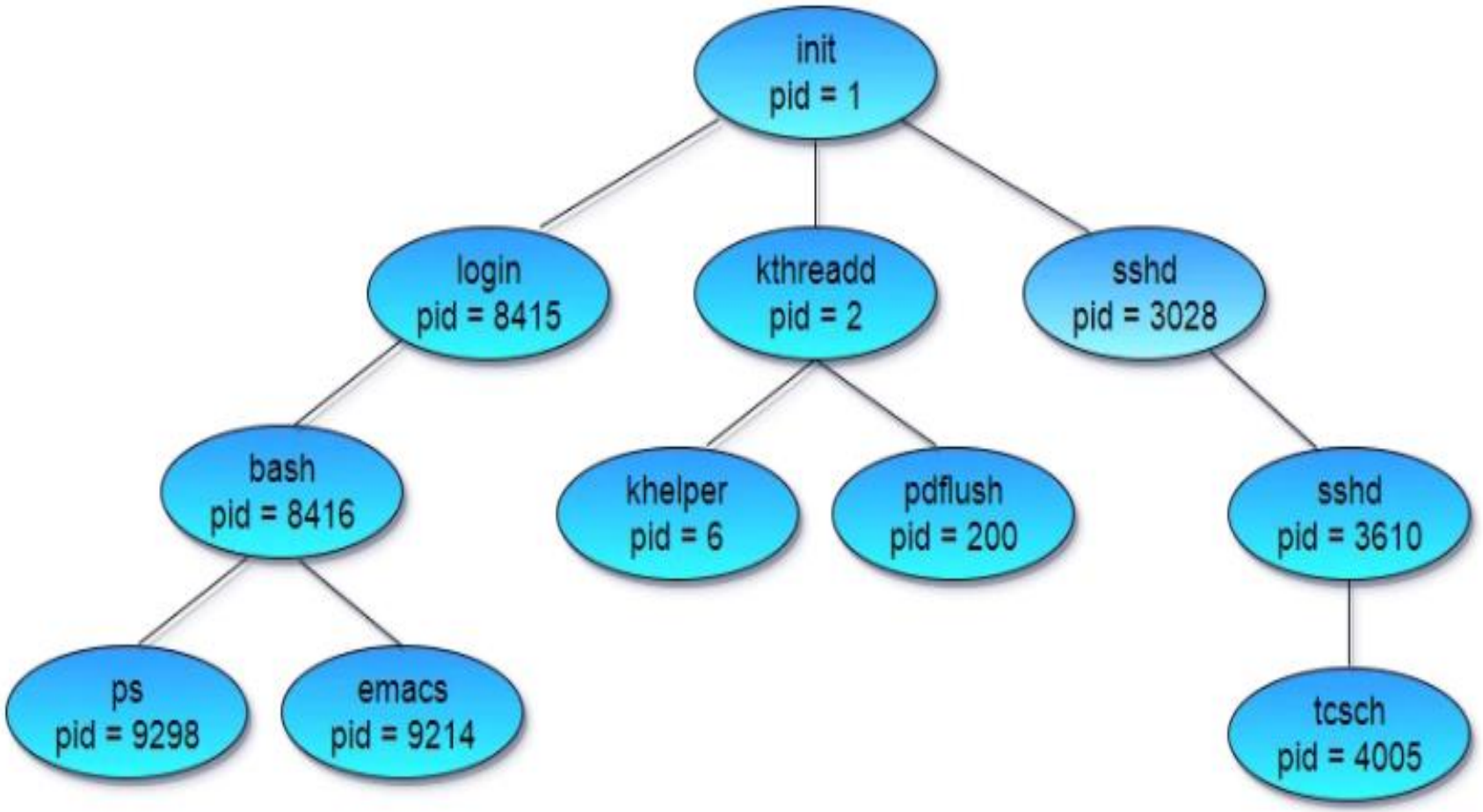
0 : child
+1 : Parent
-1 : child copy

```
case 2:

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

P
c1
C2





```
#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: 2n – 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.
- If the parent exits, the system may or may not allow the child to continue without a parent (In UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them.)
- When a process ends, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process already became an orphan.

- 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.

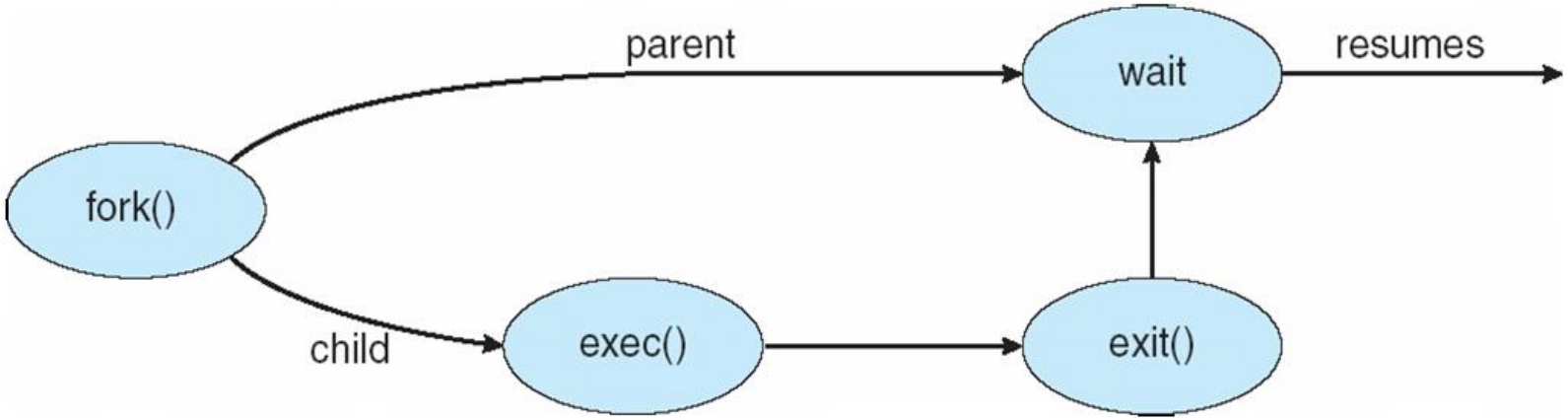
Process Creation

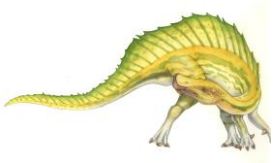
- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's 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

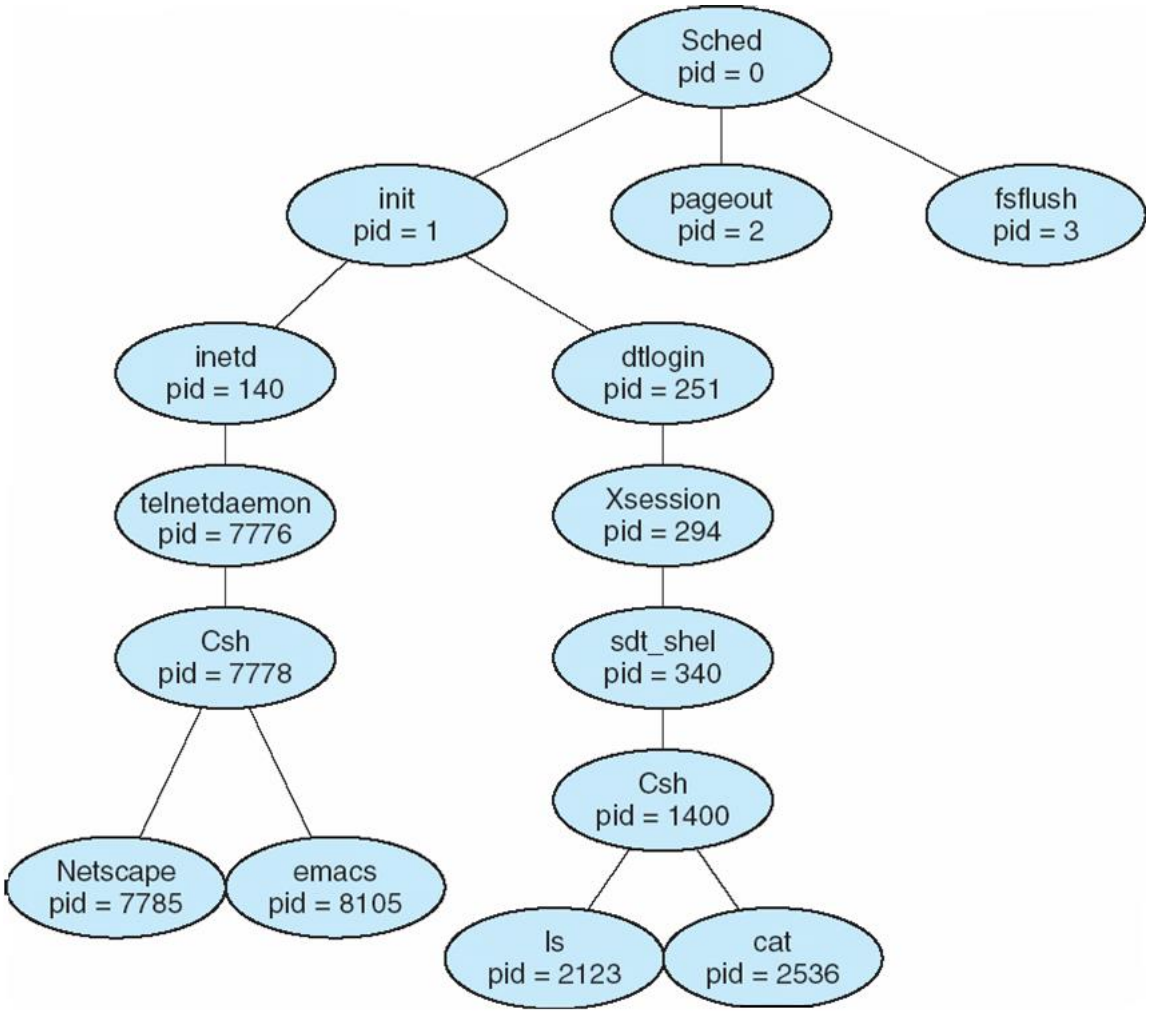




C Program Forking Separate Process

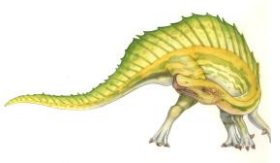
```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent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

A tree of processes on a typical Solaris

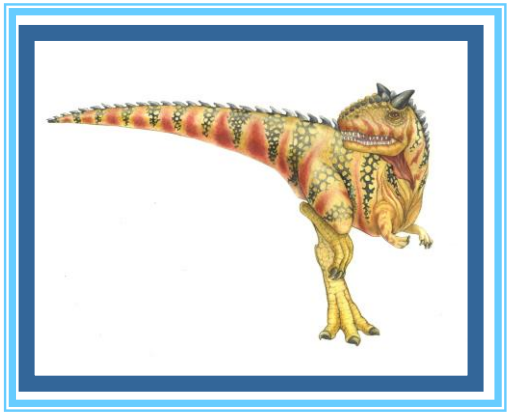


Process Termination

- 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**

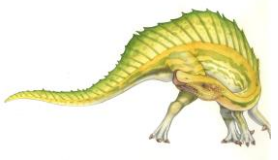


Chapter 5: CPU Scheduling



Chapter 5: CPU Scheduling

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



Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

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

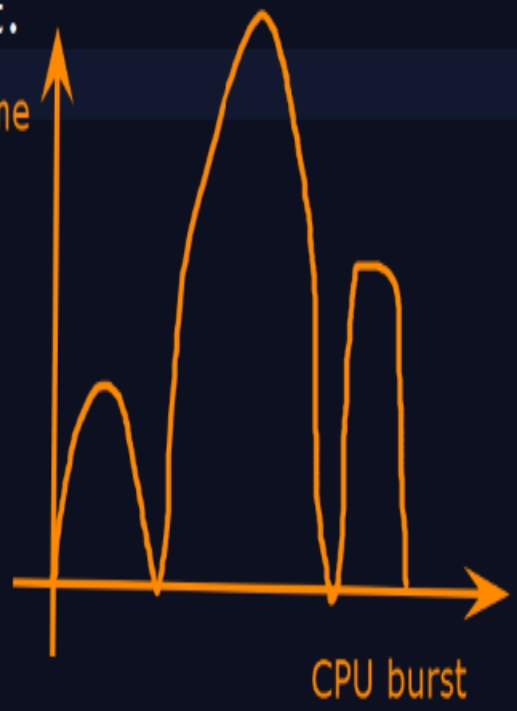
CPU scheduling:

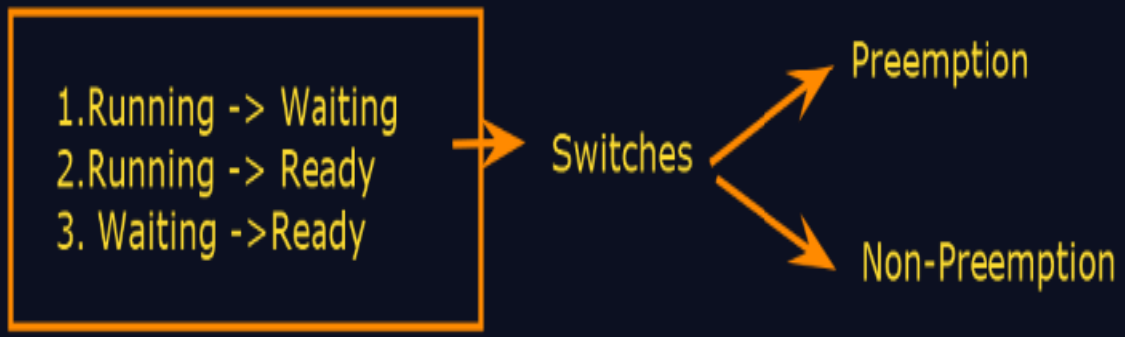
---

1. Maximum CPU utilization
2. CPU Burst and I/O Burst

- Process execution consist of a cycle of CPU execution and I/O wait.

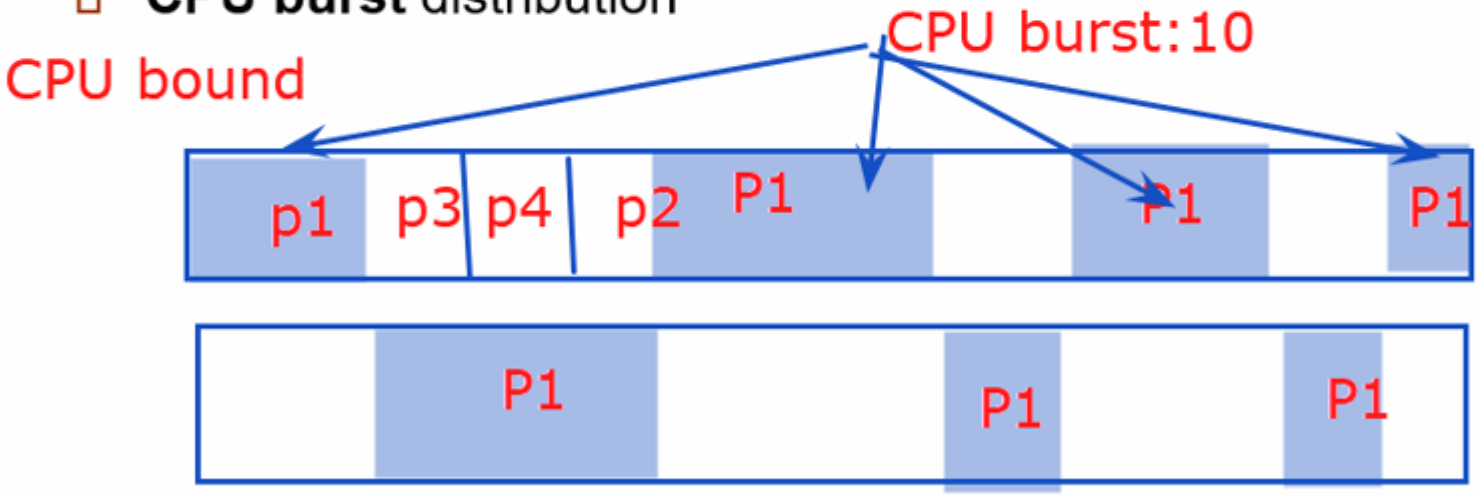
3.CPU burst distribution (Eficiently)



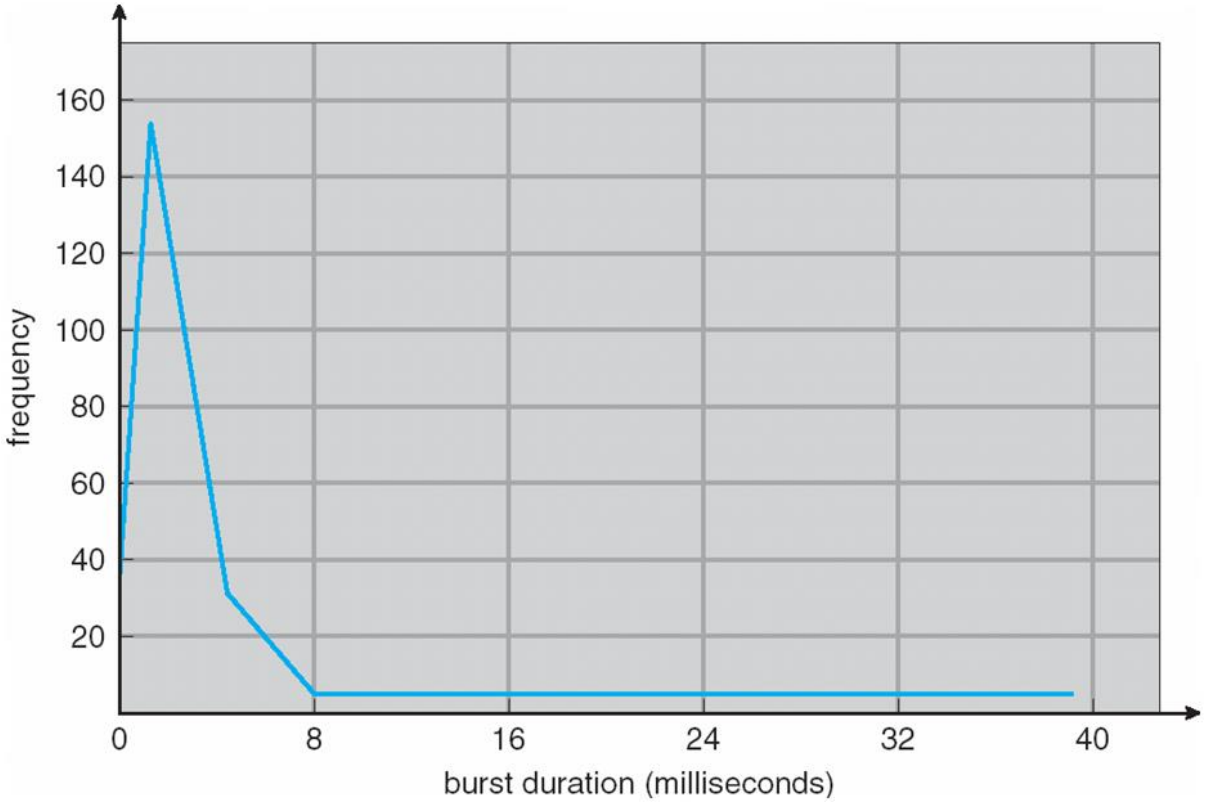


Basic Concepts

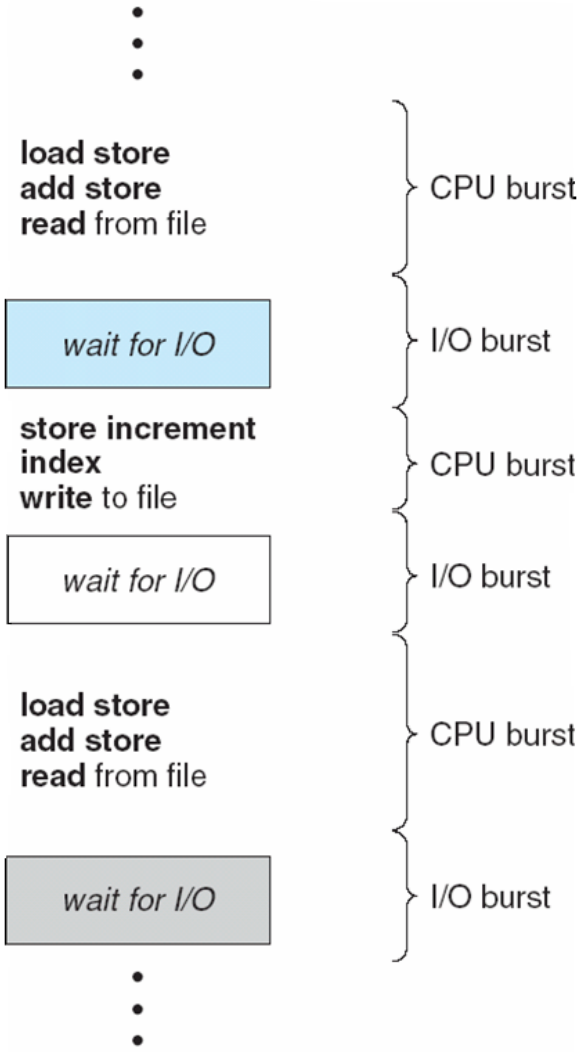
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution



Histogram of CPU-burst Times



Alternating Sequence of CPU And I/O Bursts



CPU Scheduler

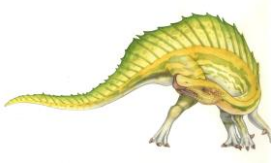
- Selects from among the **processes in memory that are ready to execute**, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from **running to waiting state**
  2. Switches from **running to ready state**
  3. Switches from **waiting to ready**
  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)



Arrival time: time at which process enter the ready queue/ready state.

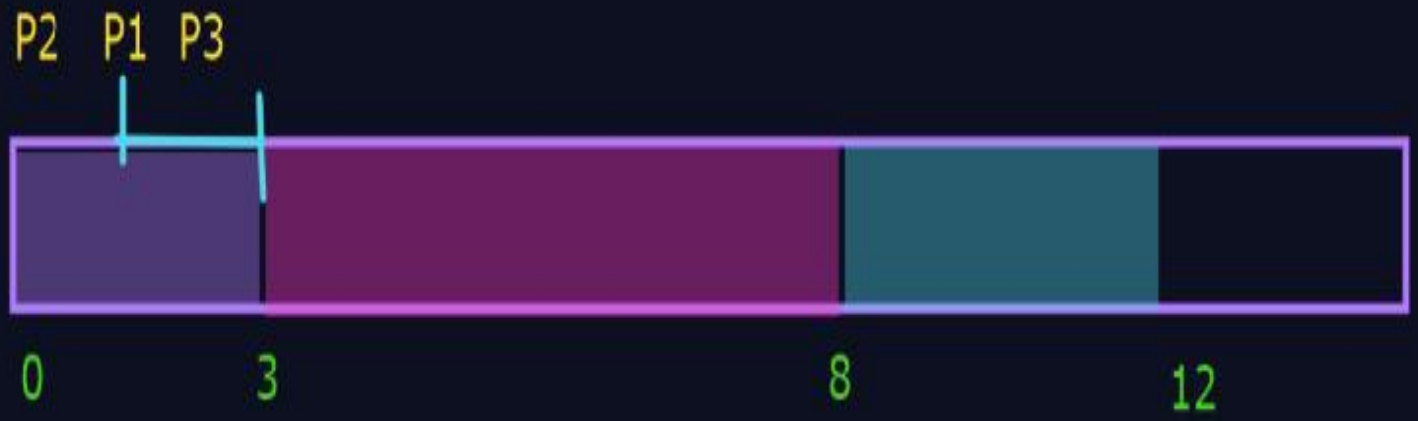
Burst time: Time required by a process to get executed by CPU.

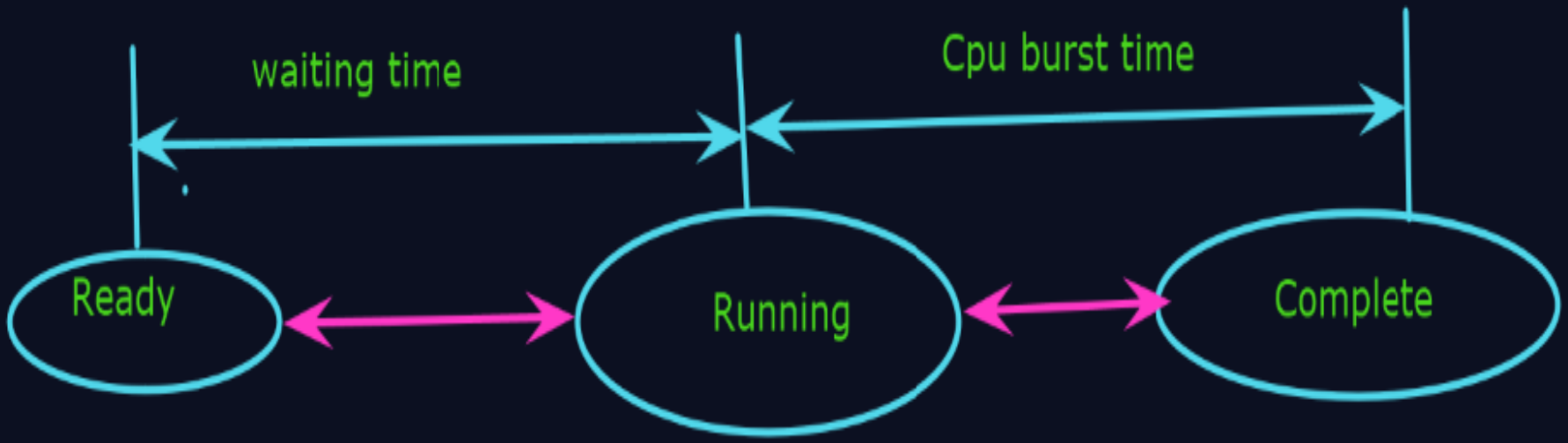
Completion time:Time at which process comple its execution.

Turn around time: {Completio time - Arrival time}

Waiting time: {Turn around time - Burst time}

Response time:{Time at which a processor get CPU - Arrival time}

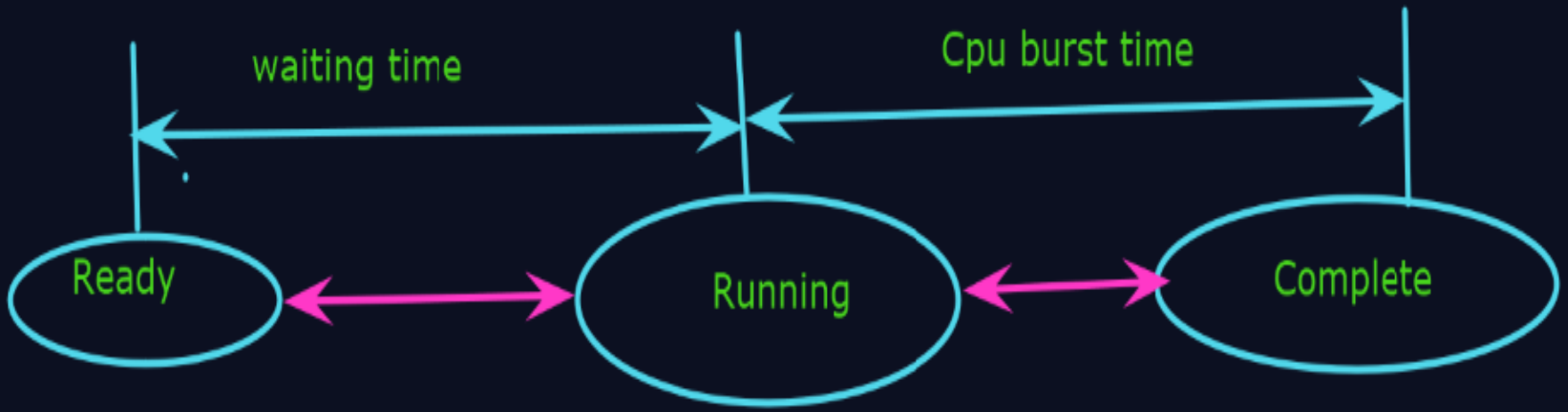




CT-AT =WT+CBT

TAT=CT -AT

WT = TAT - CBT



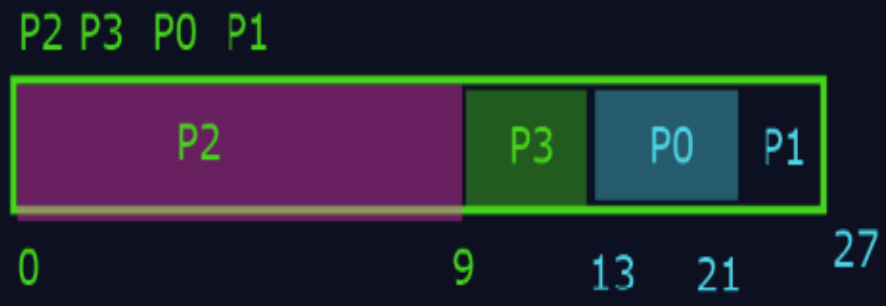
Scheduling Algorithm

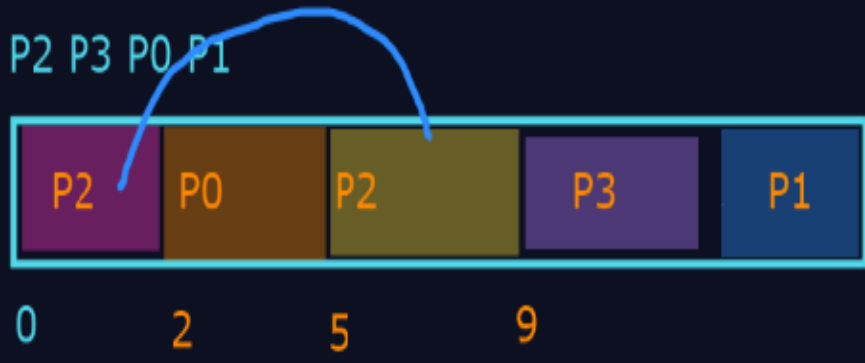
- Preemptive : SRTF, LRTF, RR, Priority based
- Non-preemptive : FCFS,SJF, LJF,RR, HRRN, Multilevel Queue, Priority

Scheduling Algorithm:

Process	AT	CBT
P0	2	8
P1	3	6
P2	0	9
P3	1	4

Process	AT	CBT	
~~P0~~	2	~~3~~	1
P1	3	5	3
P2	0	~~6~~ 4	4
P3	1	5	2





Non-Preemptive scheduling

Preemption scheduling

FCFS, SJF , Priority, RR

Scheduling Algorithm:

Process	CBT	CT	TAT	WT	RT
P1	24	24	24	0	0
P2	3	27	27	24	24
P3	3	30	30	27	27

Sequence: P1-P2-P3--->

Average TAT=(24+27+30)/3=27

Average WT= 17

Average RT= 17

Scheduling Algorithm: First Come First Serve

Process	CBT	CT	TAT	WT	RT
P1	24	30	30	6	6
P2	3	6	6	3	3
P3	3	3	3	0	0

Sequence: P3-P2-P1--->

Average TAT=13

Average WT= 3

Average RT= 3

P1

First-Come, First-Served (FCFS) Scheduling

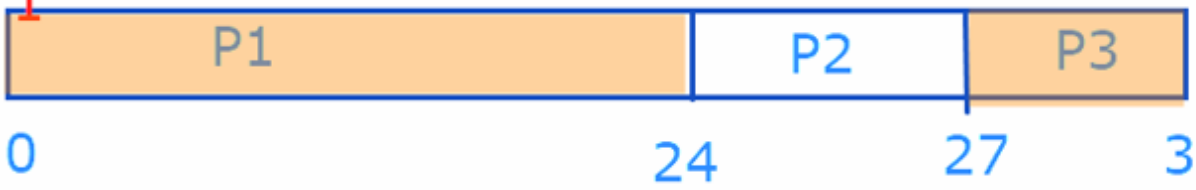
Process	Burst Time
P_1	24
P_2	3
P_3	3

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

Sequence: P2-P3-P1

Arrival time : 0

case 1



Gantt Chart

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

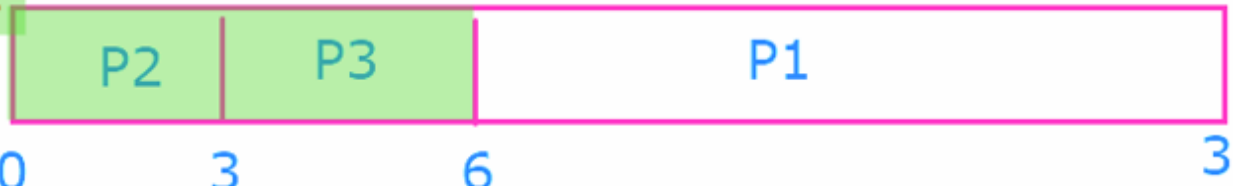
ATAT : 27

Res T : 17

Convoy Effect

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

case 2

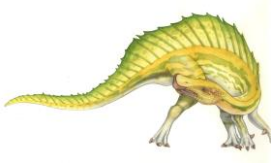


Problems with FCFS Scheduling

- Below we have a few shortcomings or problems with the FCFS scheduling algorithm:

1. It is **Non Pre-emptive** algorithm, which means the **process priority** doesn't matter.If a process with very least priority is being executed, more like **daily routine backup** process, which takes more time, and all of a sudden some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.
2. Not optimal Average Waiting Time.
3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, and hence poor resource(CPU, I/O etc) utilization.

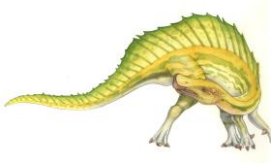
- **What is Convoy Effect?**
- Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time.
- This essentially leads to poort utilization of resources and hence poor performance.



- 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.

Shortest-Job-First (SJF) Scheduling

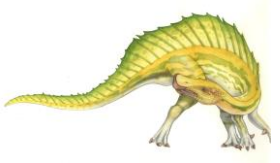
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request



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)



Non Pre-emptive Shortest Job First

- Consider the below processes available in the ready queue for execution, with arrival time as 0 for all and given burst times.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

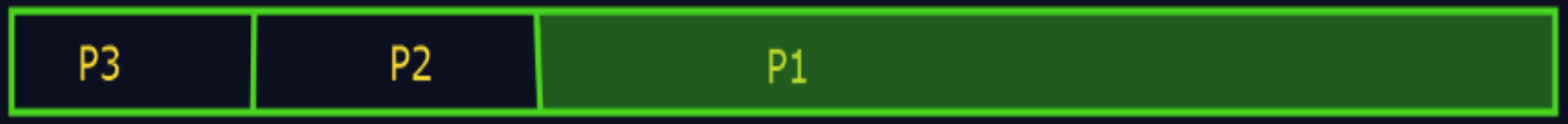
Hence the GANTT chart will be following :

Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

Shortest Job First

Process	AT	CBT	CT	TAT	WT	RT
P1	0	24	30	30	6	6
P2	0	3	6	6	3	3
P3	0	3	3	3	0	0

Sequence: P3-P2-P1--->



Average TAT=13

Average WT= 3

Average RT= 3

Problem with Non Pre-emptive SJF

- If the arrival time for processes are different, which means all the processes are not available in the ready queue at time 0, and some jobs arrive after some time, in such situation, sometimes process with short burst time have to wait for the current process's execution to finish, because in Non Pre-emptive SJF, on arrival of a process with short duration, the existing job/process's execution is not halted/stopped to execute the short job first.

- This leads to the problem of **Starvation**, where a shorter process has to wait for a long time until the current longer process gets executed. This happens if shorter jobs keep coming, but this can be solved using the concept of **aging**.

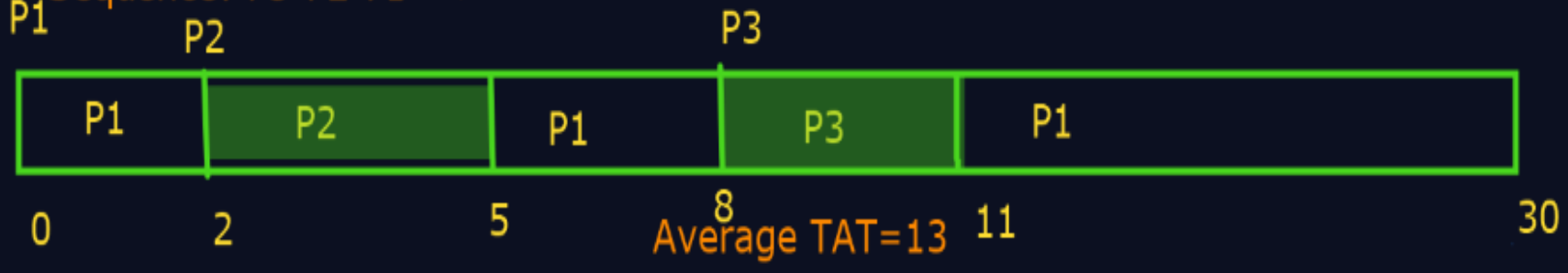
Shortest Job First

P1	21
P2	3
P3	6
P4	2

p4
p5
p6
p7

Process	AT	CBT	CT	TAT	WT	RT
P1	0	~~24~~ ~~22~~ 19	30	30	6	0
P2	2	3	5	3	0	0
P3	8	3	11	3	0	0

Sequence: P3-P2-P1--->



Average TAT=13

Average WT= 3

Average RT= 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 CPU Scheduling

In this tutorial we will understand the priority scheduling algorithm, how it works and its advantages and disadvantages.

In the Shortest Job First scheduling algorithm, the priority of a process is generally the inverse of the CPU burst time, i.e. the larger the burst time the lower is the priority of that process.

In case of priority scheduling the priority is not always set as the inverse of the CPU burst time, rather it can be internally or externally set, but yes the scheduling is done on the basis of priority of the process where the process which is most urgent is processed first, followed by the ones with lesser priority in order.

Processes with same priority are executed in FCFS manner.

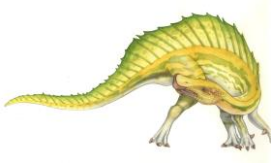
The priority of process, when internally defined, can be decided based on memory requirements, time limits ,number of open files, ratio of I/O burst to CPU burst etc.

Whereas, external priorities are set based on criteria outside the operating system, like the importance of the process, funds paid for the computer resource use, makrte factor etc.

Types of Priority Scheduling Algorithm

- Priority scheduling can be of two types:

1. **Preemptive Priority Scheduling**: If the **new process arrived at the ready queue has a higher priority** than the currently running process, the CPU is preempted, which means the processing of the current process is stoped and the incoming new process with higher priority gets the CPU for its execution.
2. **Non-Preemptive Priority Scheduling**: In case of non-preemptive priority scheduling algorithm if a **new process arrives with a higher priority than the current running process, the incoming process is put at the head of the ready queue,** which means after the execution of the current process it will be processed.



Priority algorithm +Preemption with SJF

P1	21
P2	3
P3	6
P4	2

p4
p5
p6
p7

Process	P	CBT	CT	TAT	WT	RT
P1	3	24 29 19	30	30	6 ✓	0
P2	2	3	5	3	0	0
P3	1	3	11	3	0	0

Sequence: P3-P2-P1--->

Priority algorithm +Preemption with SJF

P1	21
P2	3
P3	6
P4	2

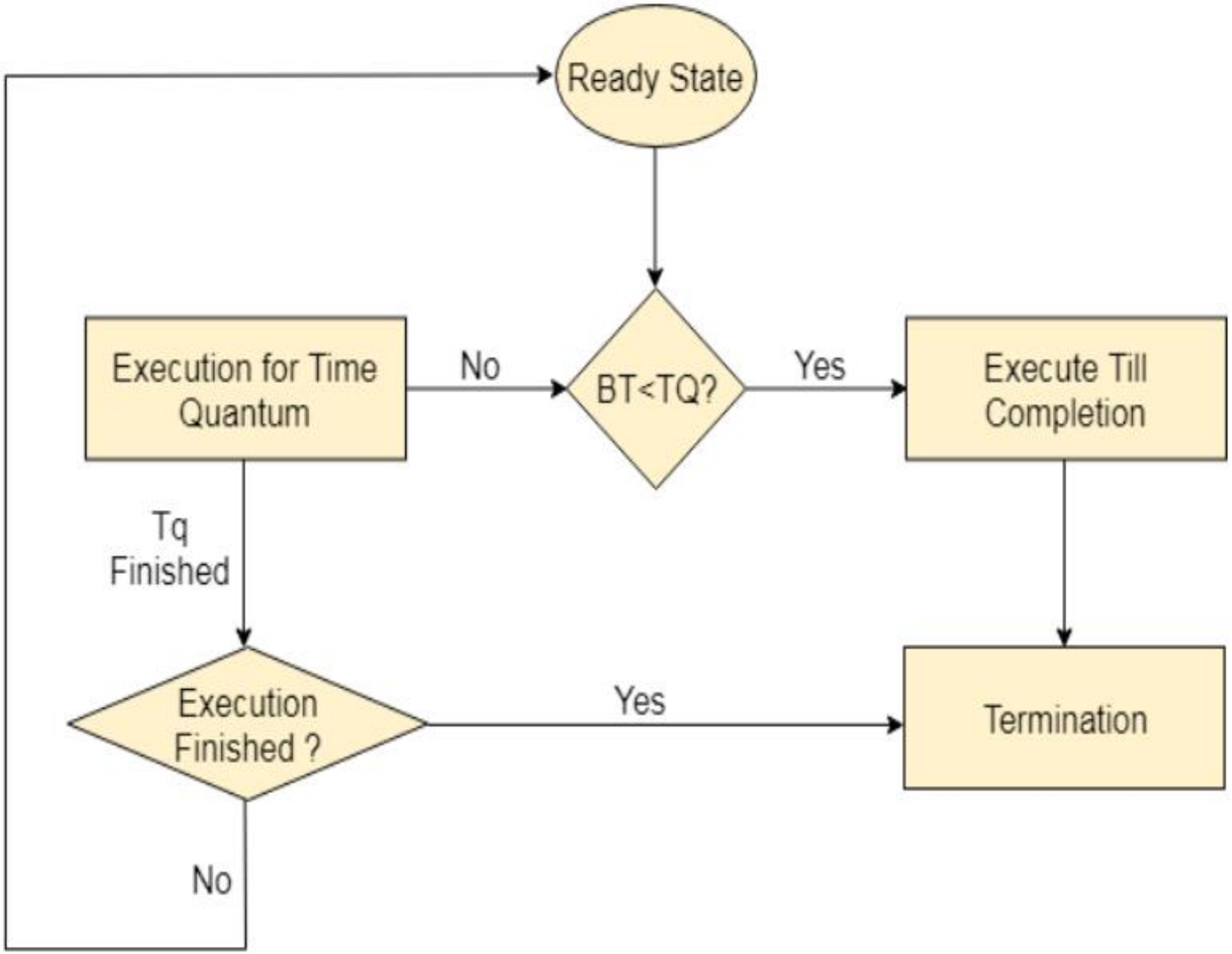
Process	AT	P	CBT	CT	TAT	WT	RT
P1	0	3	~~24~~ ~~22~~ 9	30	30	6 ✓	0
P2	2	2	3	5	3	0	0
P3	12	1	3	11	3	0	0

p4
p5
p6
p7

Sequence: P3-P2-P1--->

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.



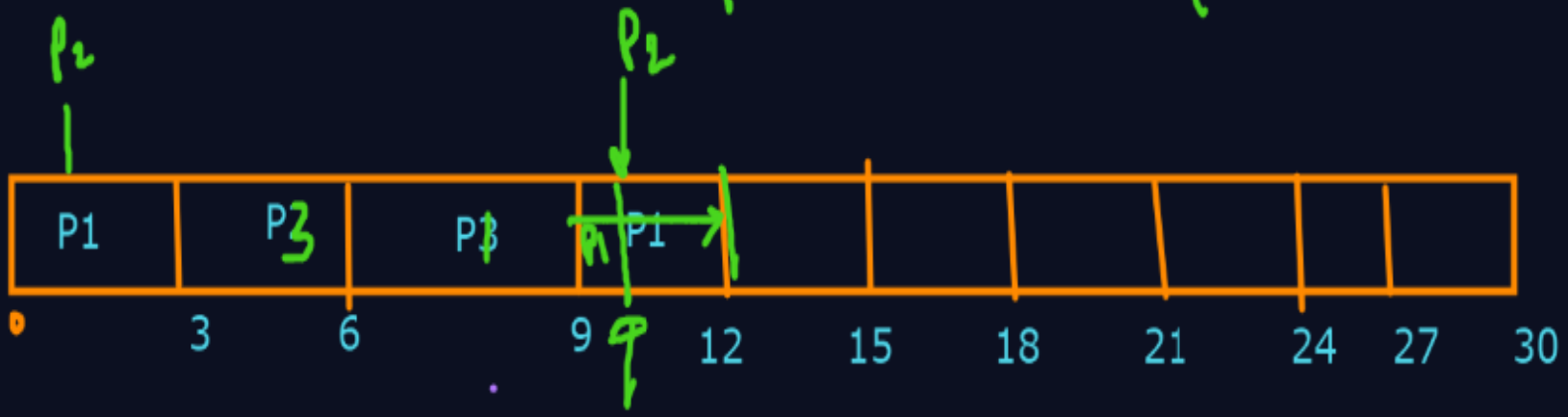
Some important characteristics of the Round Robin(RR) Algorithm are as follows:

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.

Round Robin : time slice: quantum =3s

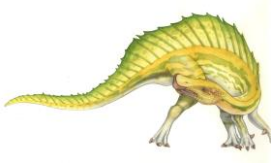
- CS

Process	AT	CBT	CT	TAT	WT	RT
P1	0	~~24~~ 21	30	30	6 ✓	0
P2	~~10~~ 0	~~3~~	5	3	0	0
P3	0	~~3~~ 2	11	3	0	0



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

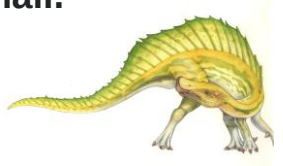
- Some disadvantages of the Round Robin scheduling algorithm are as follows:
- This algorithm spends **more time on context switches**.
- For **small quantum**, it is time-consuming scheduling.
- This algorithm **offers a larger waiting time and response time**.
- In this, there is **low throughput.**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

Some Points to Remember

- **1.Decreasing value of Time quantum**
- With the decreasing value of time quantum
  - The number of context switches increases.
  - The Response Time decreases
  - Chances of starvation decrease in this case.
- For the **smaller value of time quantum,** it becomes better in terms of **response time.**

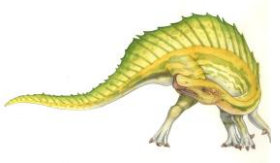
- **2.Increasing value of Time quantum**
- With the increasing value of time quantum
  - The number of context switch decreases
  - The Response Time increases
  - Chances of starvation increases in this case.
- For the higher value of time quantum, it becomes better in terms of the **number of the context switches.**

- 3. If the value of **time quantum is increasing** then Round Robin Scheduling tends to **become FCFS Scheduling.**
- 4.In this case, when the value of time quantum **tends to infinity** then the Round Robin Scheduling **becomes FCFS Scheduling.**
- 5. Thus the performance of Round Robin scheduling mainly depends on the **value of the time quantum.**
- 6.And the value of the **time quantum** should be such that it is neither **too big nor too small.**



Round Robin (RR)

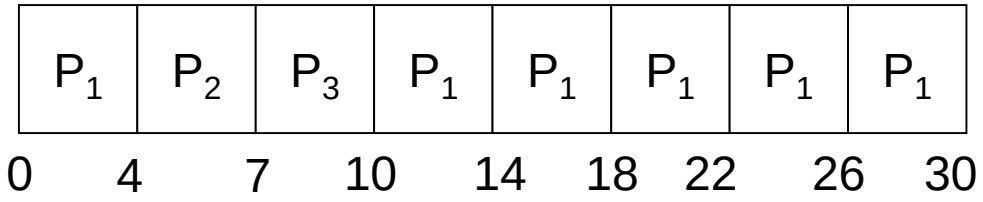
- Each process gets **a small unit of CPU time (*time quantum*), usually 10-100 milliseconds**. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the 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 $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high



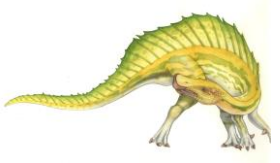
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*

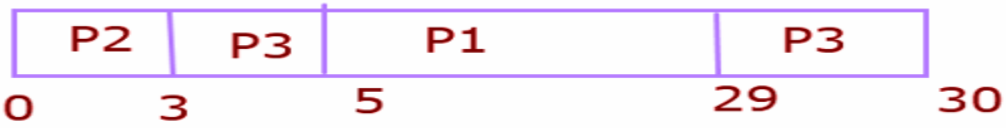


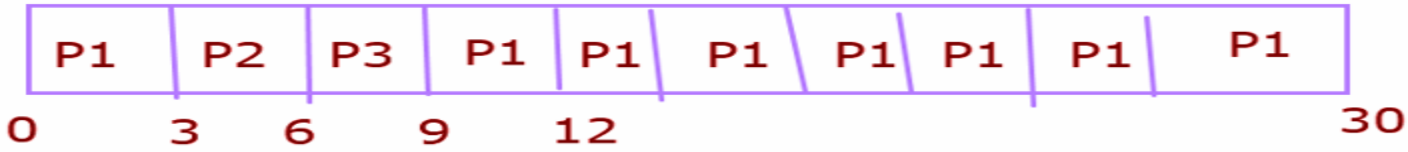
Example of RR with Time Quantum = 4

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

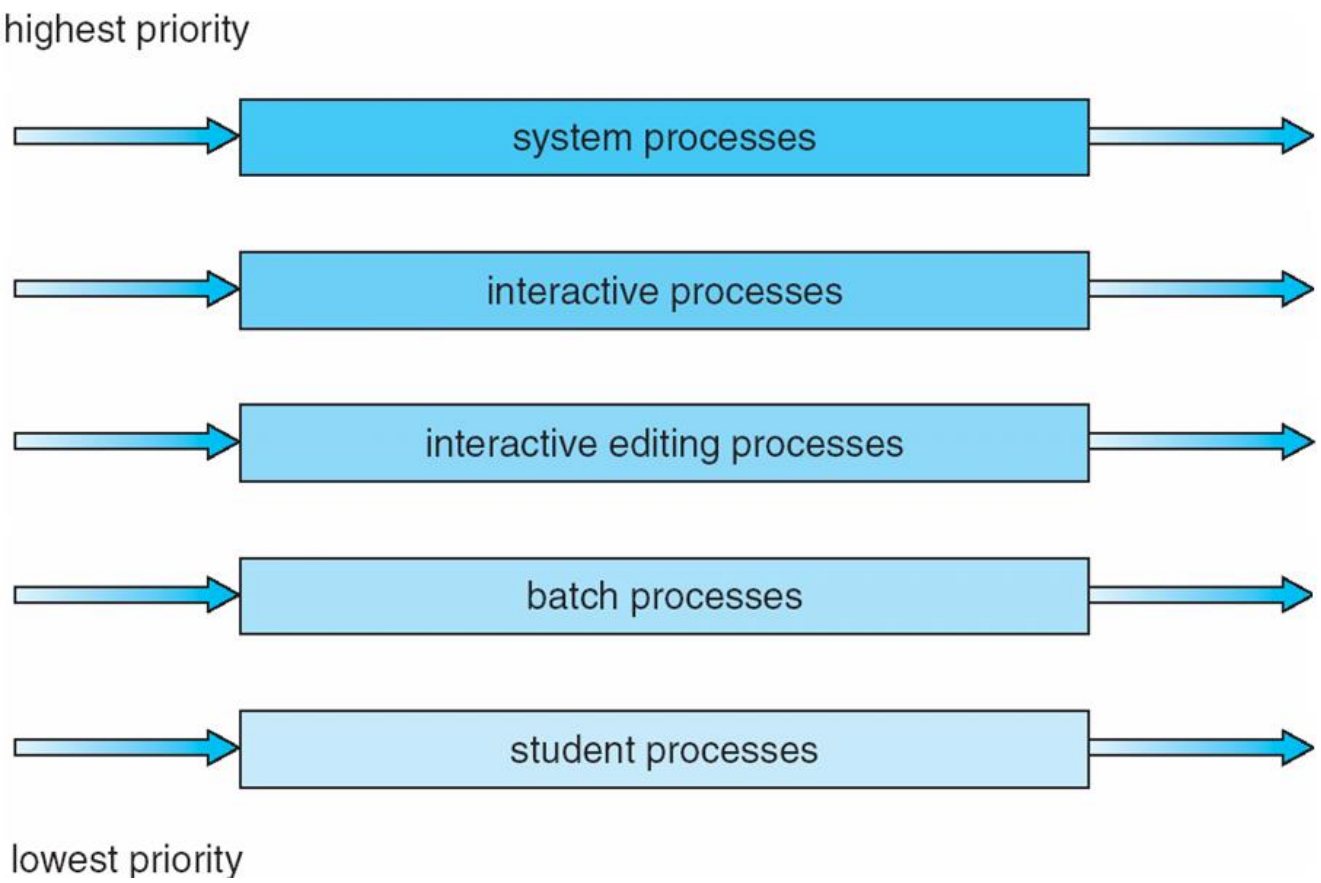




Multilevel Queue

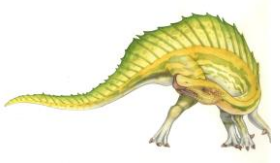
- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then 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