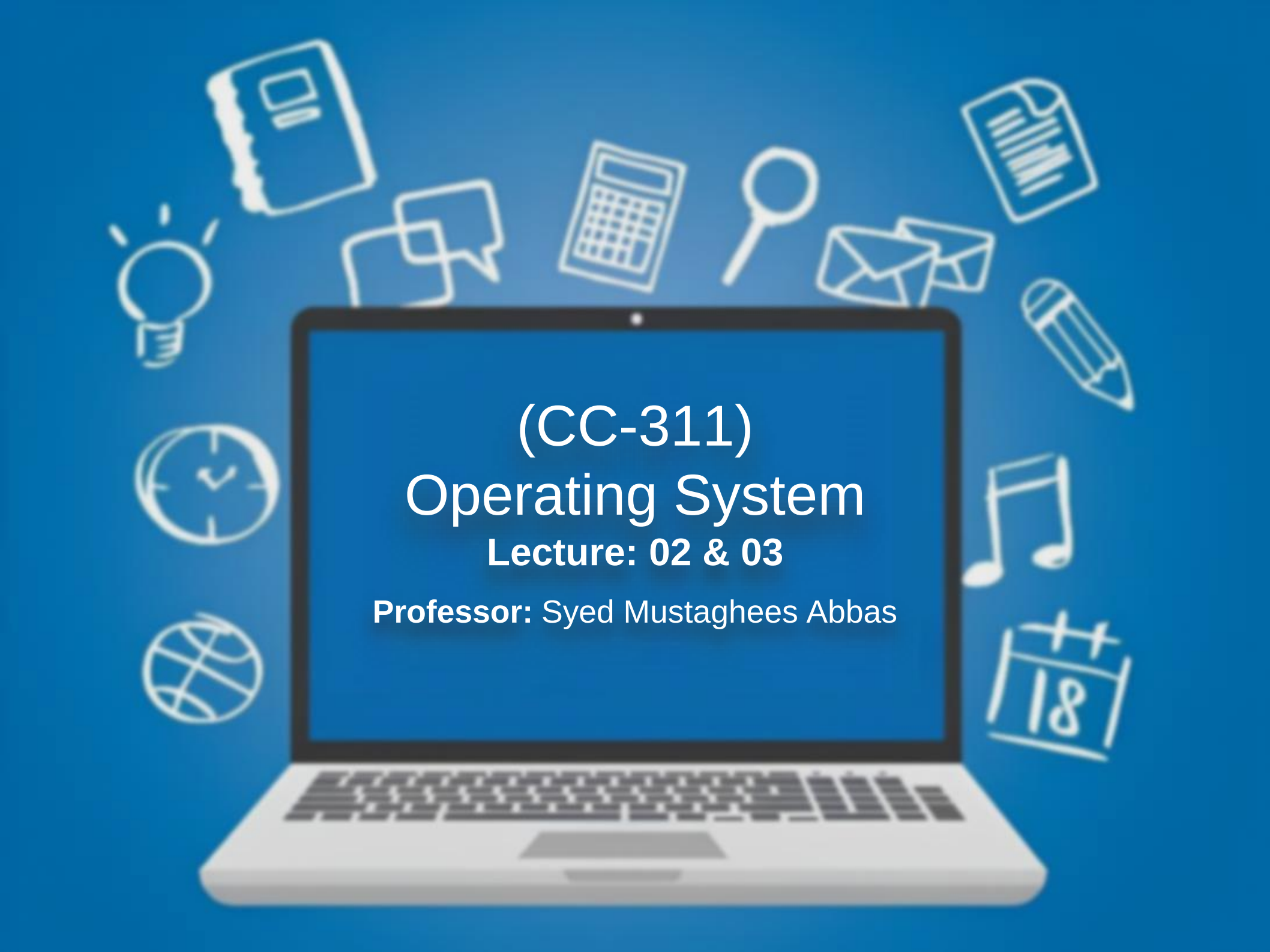


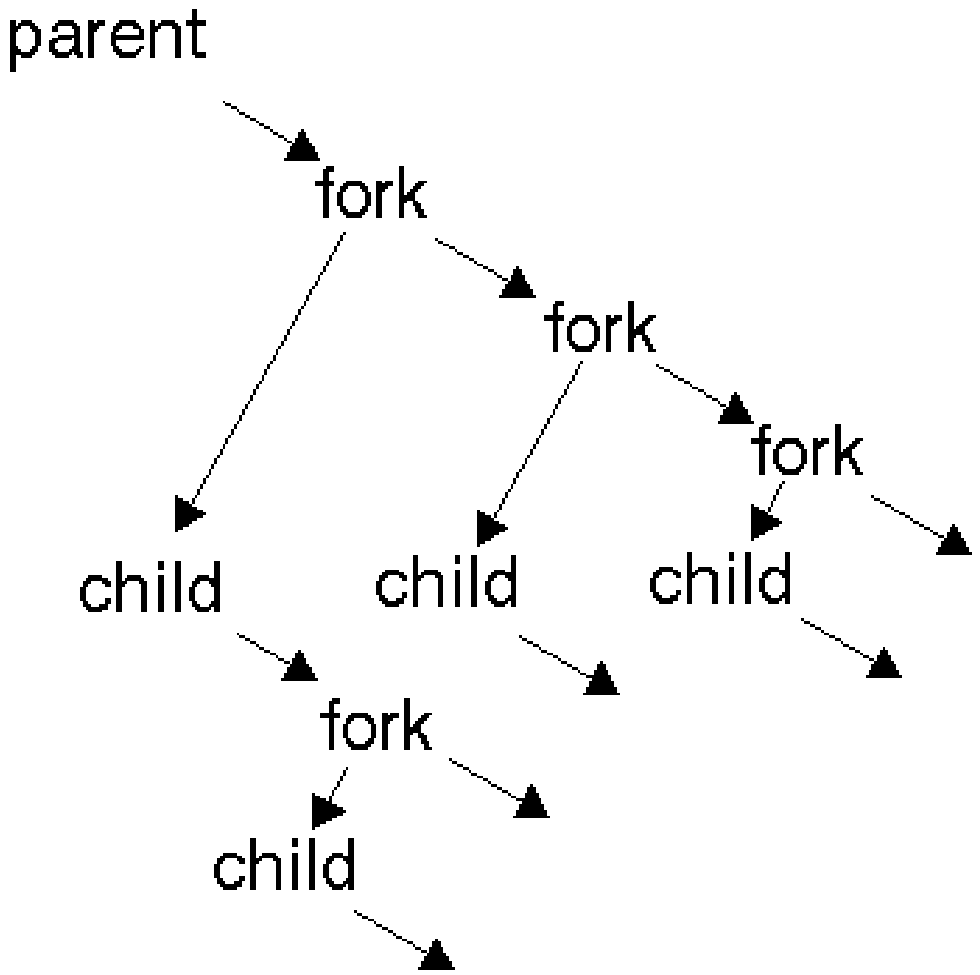
(CC-311)
Operating System
Lecture: 02 & 03

Professor: Syed Mustaghees Abbas

Operations on processes

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically. These systems must provide a mechanism for process creation and termination.
- In Modern OS different system-calls use for creation and deletion.
- **Process Creation**
  - ✓ A process may create several new processes during the course of execution. The creating process is called a **parent process** and the new processes are called the **children of that process**.
  - ✓ Each child process can further create other processes, forming a tree of processes.
  - ✓ Most operating systems identify processes according to a unique process identifier (or pid), which is typically an integer number.

Process Creation (Continued)



Process Creation (Continued)

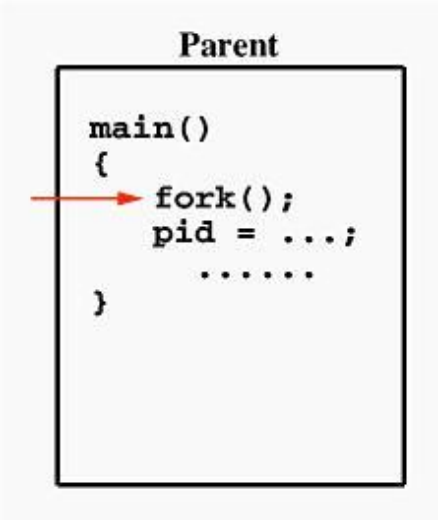
- In general, a process will need certain resources (**CPU time, memory, files,I/O devices**) to accomplish its task.
- When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the OS or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children or it may be able to share some resources (such as memory or files) among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

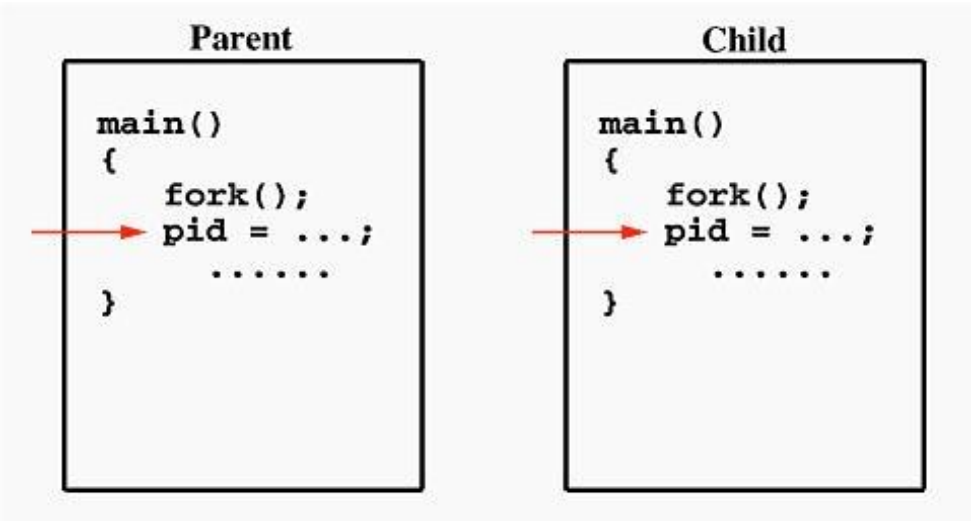
Process Creation (Continued)

- When a process creates a new process there are two possibilities exist in terms of execution and in term address space
  - ✓ **In terms of execution:**
    1. The parent continues to execute concurrently with its children.
    2. The parent waits until some or all of its children have terminated.
  - ✓ **In terms of the address space:**
    1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
    2. The child process has a new program loaded into it.

Program using Fork()

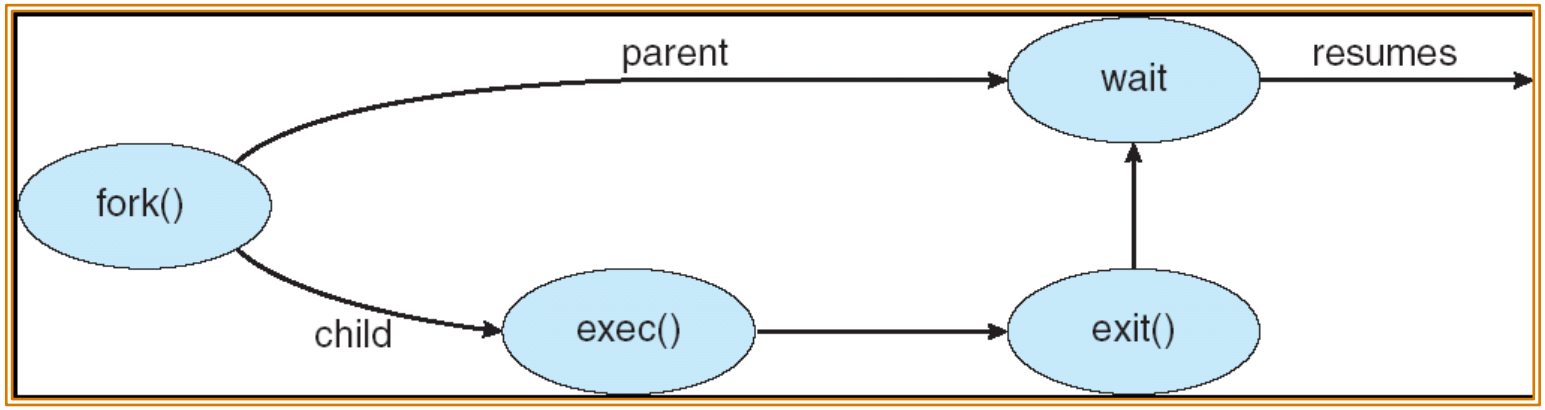
- In Linux new process is created by the fork() system call. The new process consists of a copy of the address space of the original
- Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: The return code for the fork() is zero for the new (child) process, whereas the nonzero process identifier of the child is returned to the parent.





Process Creation (Continued)

- Typically, the exec() system call is used after a fork() to replace the process's memory space with a new program.
- It loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution.
- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait () system call to move itself off the ready queue until the termination of the child.



Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- All the resources of the process: including physical and virtual memory, open files, and I/O buffers are deallocated by the operating system.
- A process can cause the termination of another process via an appropriate system call. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.

Process Termination (Continued)

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  - ✓ The task assigned to the child is no longer required.
  - ✓ The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.
- **Cascading Termination:**

  Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

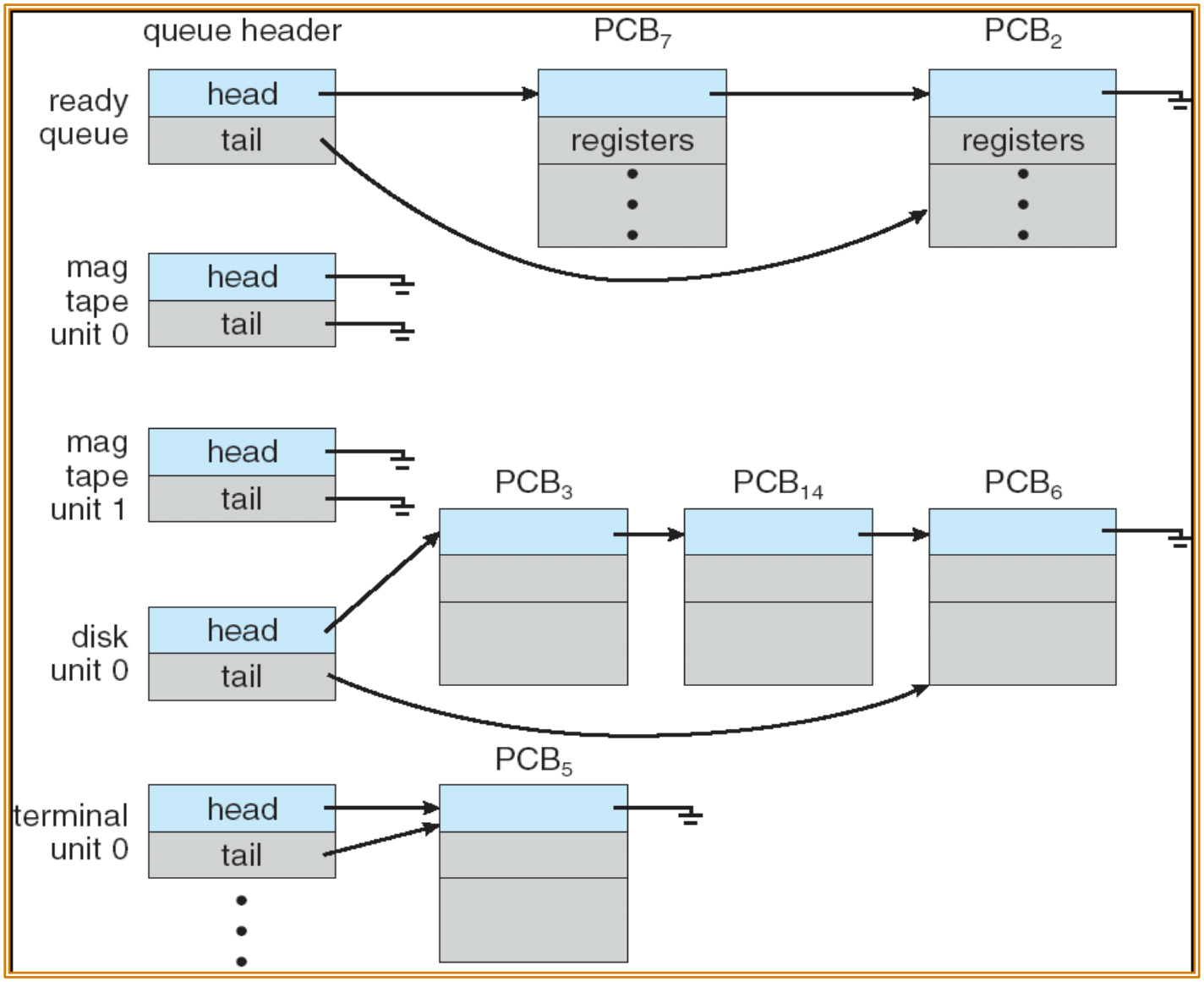
Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- For a single-processor system, there will never be more than one running process.
- If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.
- CPU scheduling is the basis of multiprogrammed operating systems
- On operating systems that support threads, it is threads—not processes—that are in fact being scheduled by the operating system. However, the terms process scheduling and thread scheduling are often used interchangeably.

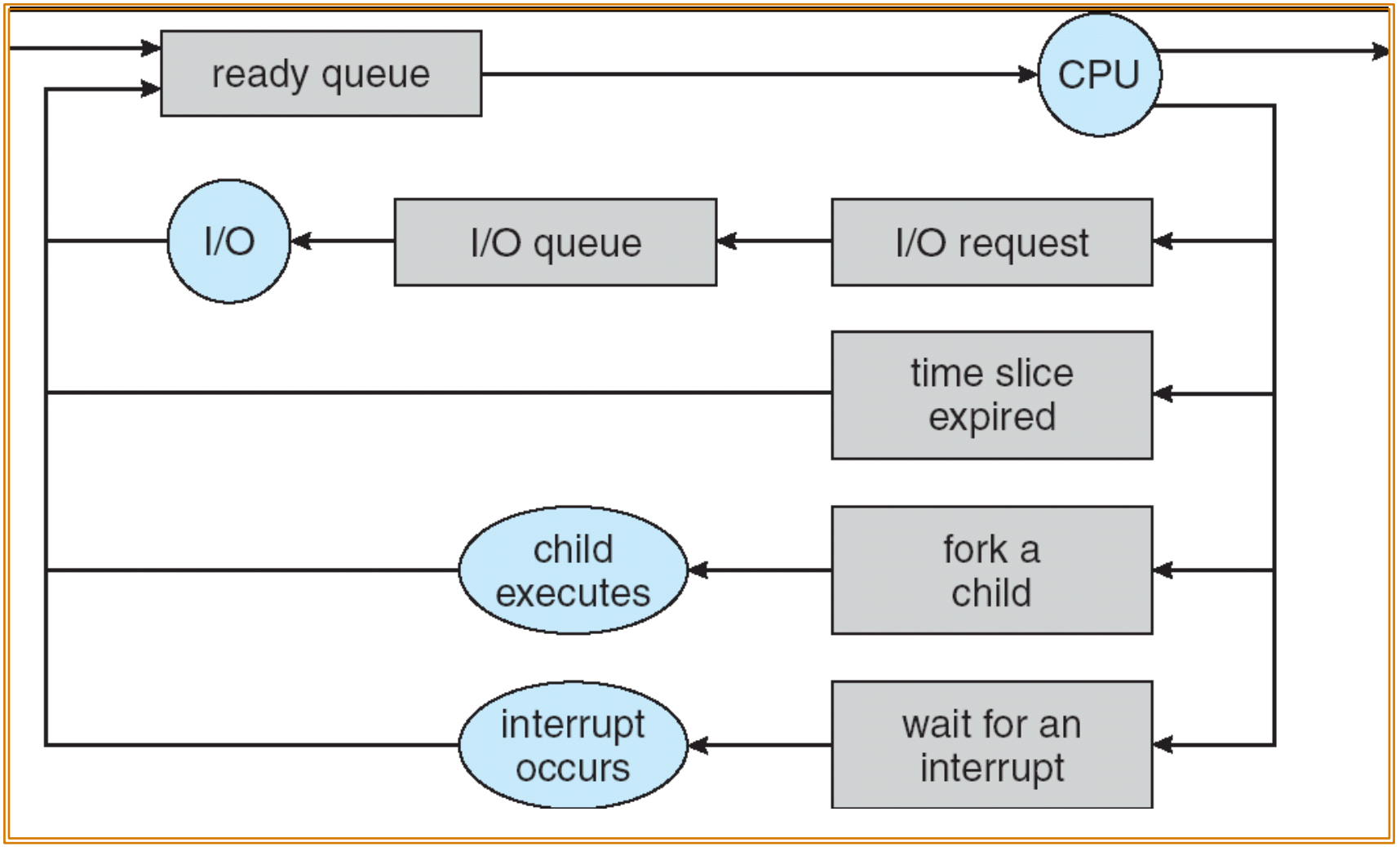
Process Scheduling Queues

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.
- This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- Suppose the process while execution makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process.
- The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own **device queue**.

Ready Queue And Various I/O Device Queues



Queuing Diagram



A common representation for a discussion of process scheduling is a queuing diagram

Process Scheduling Queues

- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue waiting to be selected for execution.
- Once the process is allocated the CPU and is executing, one of several events could occur
- The process could issue an I/O request and then be placed in an I/O queue.

Process Scheduling Queues

- The process could create a new subprocess and wait for the subprocess's termination.The parent process goes into waiting state.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

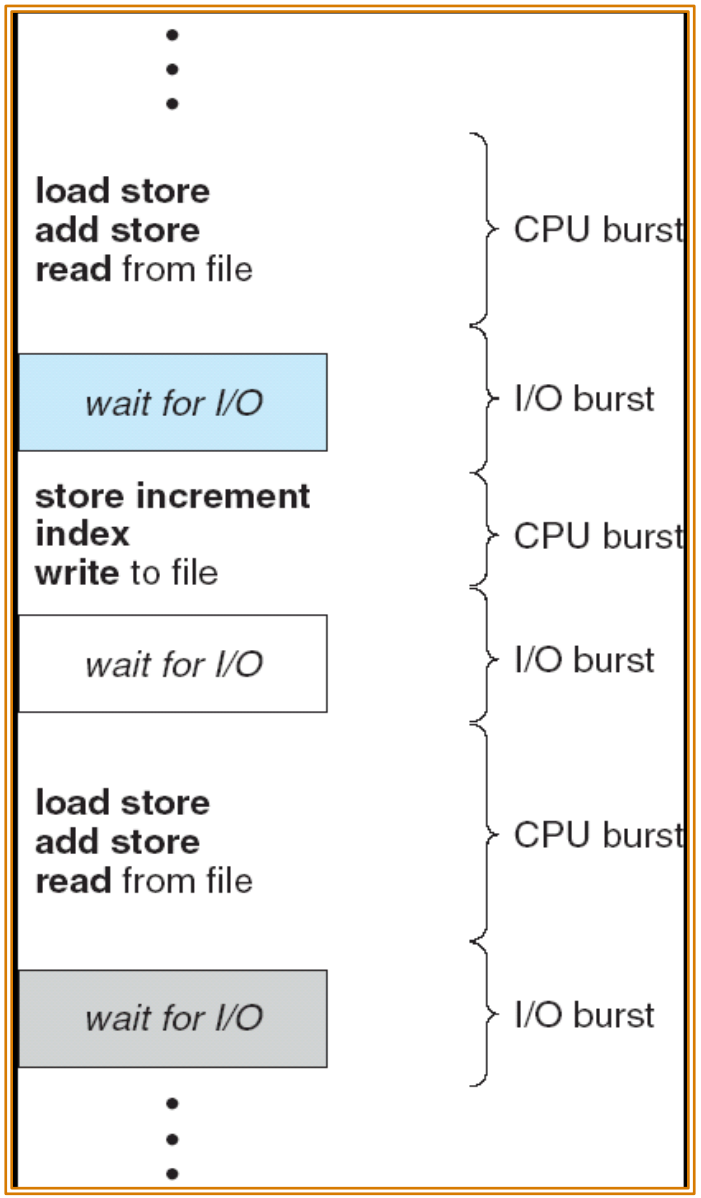
Schedulers

- A process migrates among the various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler from these queues in some fashion.
- A **long-term scheduler** is typical of a batch system. It runs infrequently, (such as when one process ends selecting one more to be loaded in from disk in its place), and can afford to take the time to implement intelligent and advanced scheduling algorithms.
- The **short-term scheduler**, or CPU Scheduler, runs very frequently, on the order of 100 milliseconds, and must very quickly swap one process out of the CPU and swap in another one.

CPU-I/O Burst Cycle

- Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst.That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- The durations of CPU bursts have been measured extensively
- Although they vary greatly from process to process and from computer to computer, generally we have a large number of short CPU bursts and a small number of long CPU bursts.

Alternating Sequence of CPU And I/O Bursts



Process Scheduling

- In general, most processes can be described as either I/O bound or CPU bound.
- An **I/O-bound** process is one that spends more of its time doing I/O than it spends doing computations.
- A **CPU-bound** process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- It is important that the long-term scheduler make a careful selection. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU- bound and I/O-bound processes.

Schedulers

- On some systems, the long-term scheduler may be absent or minimal.
- For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.
- Time-sharing systems, may introduce an additional, intermediate level of scheduling known as **medium-term scheduler**.
- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory and thus reduce the degree of multiprogramming.

Schedulers

- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.
- The process is swapped out, and is later swapped in, by the medium-term scheduler.
- Swapping may be necessary to improve the process mix (IO bound vs CPU bound) or available memory is exhausted and new process needs to be run requiring memory to be freed up.

Schedulers

- 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
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.
- When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **non-preemptive** or cooperative; otherwise, it is **preemptive**.

Preemptive and non-preemptive scheduling?

- Processes have priorities and at times it is necessary to run a certain process that has a higher priority before another process although it is running. Therefore, the running process is interrupted for some time and resumed later when the high priority process has finished its execution. This is called preemptive scheduling. Precisely

- ***Preemptive scheduling***: The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.
- ***Non-Preemptive scheduling***: When a process enters the state of running, the process continues to run until it finishes its service time.

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

Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – Number 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. It is the sum of the periods spent waiting in the ready queue.
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced

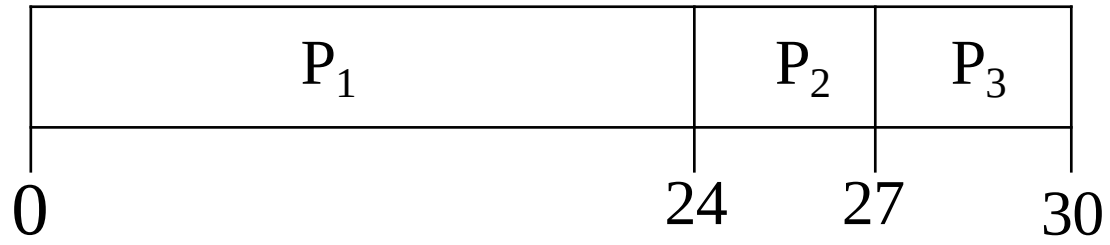
Optimization Criteria

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

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1= 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

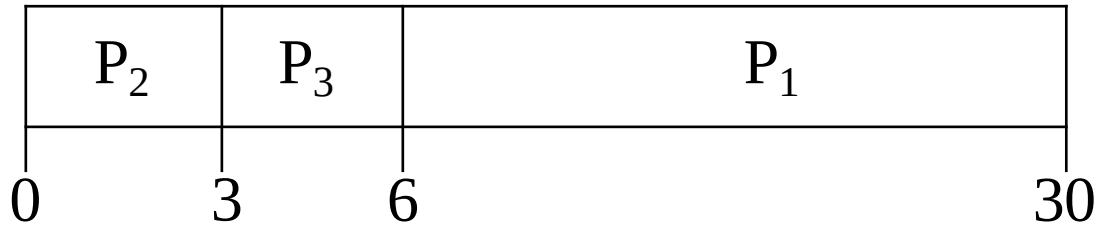
Can we somehow reduce the average wait time?

FCFS Scheduling (Continued)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- ***Convoy effect*** short process behind long process

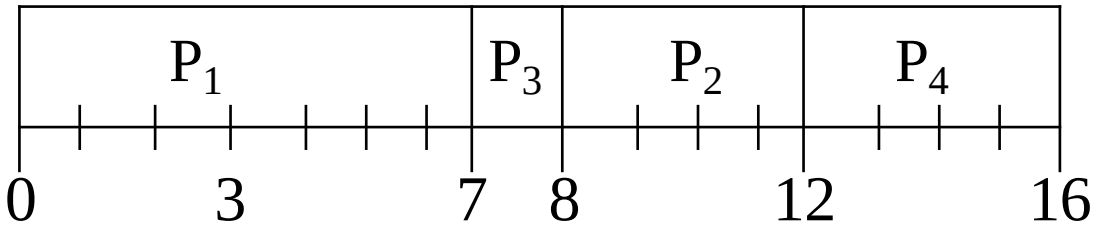
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - ✓ **Non-preemptive:** once CPU given to the process it cannot be preempted until completes its CPU burst
  - ✓ **Preemptive:** if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the **Shortest-Remaining-Time-First (SRTF)**
- SJF is optimal: gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

SJF (non–preemptive)

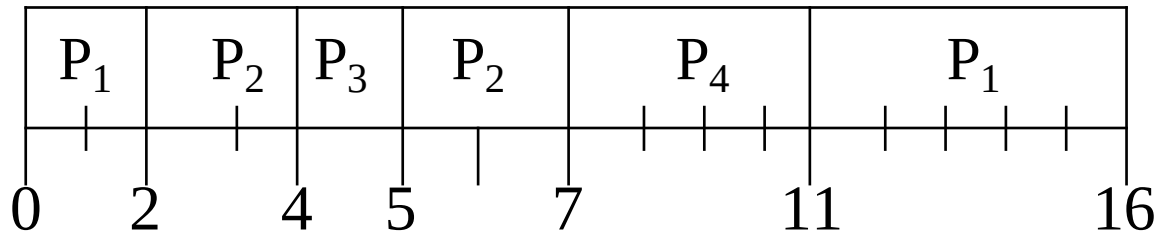


- Waiting times $P_1 = 0$, $P_2 = 6$, $P_3 = 3$, $P_4 = 7$
- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

SJF (preemptive)



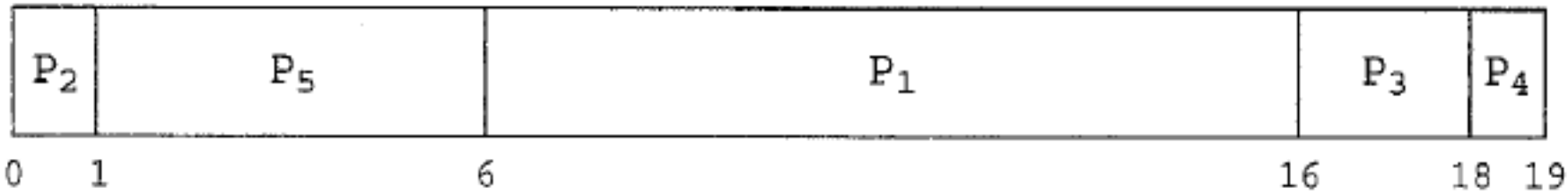
- Waiting times $P_1 = 9$, $P_2 = 1$, $P_3 = 0$, $P_4 = 2$
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

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

Example of Non-Preemptive Priority scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



- Waiting times P_1=6, P_2 =0, P_3 =16, P_4 =18, P_5 =1
- Average waiting time = (6 + 0 + 16 +18+1)/5 = 8.2

Example of Preemptive Priority scheduling

Process	Burst time	Priority	Arrival time
P_1	10	3	3
P_2	1	1	4
P_3	2	4	2
P_4	1	5	0
P_5	5	2	1

- Waiting times P_1=4, P_2 =0, P_3 =15, P_4 =0, P_5 =1
- Average waiting time = (4 + 0 + 15 + 0 + 1)/5 = 4

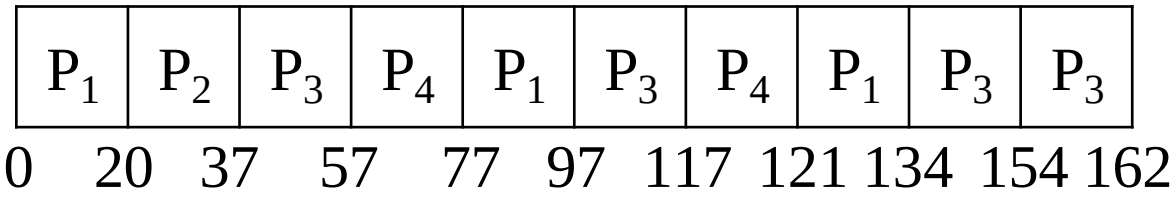
Round Robin (RR)

- 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.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

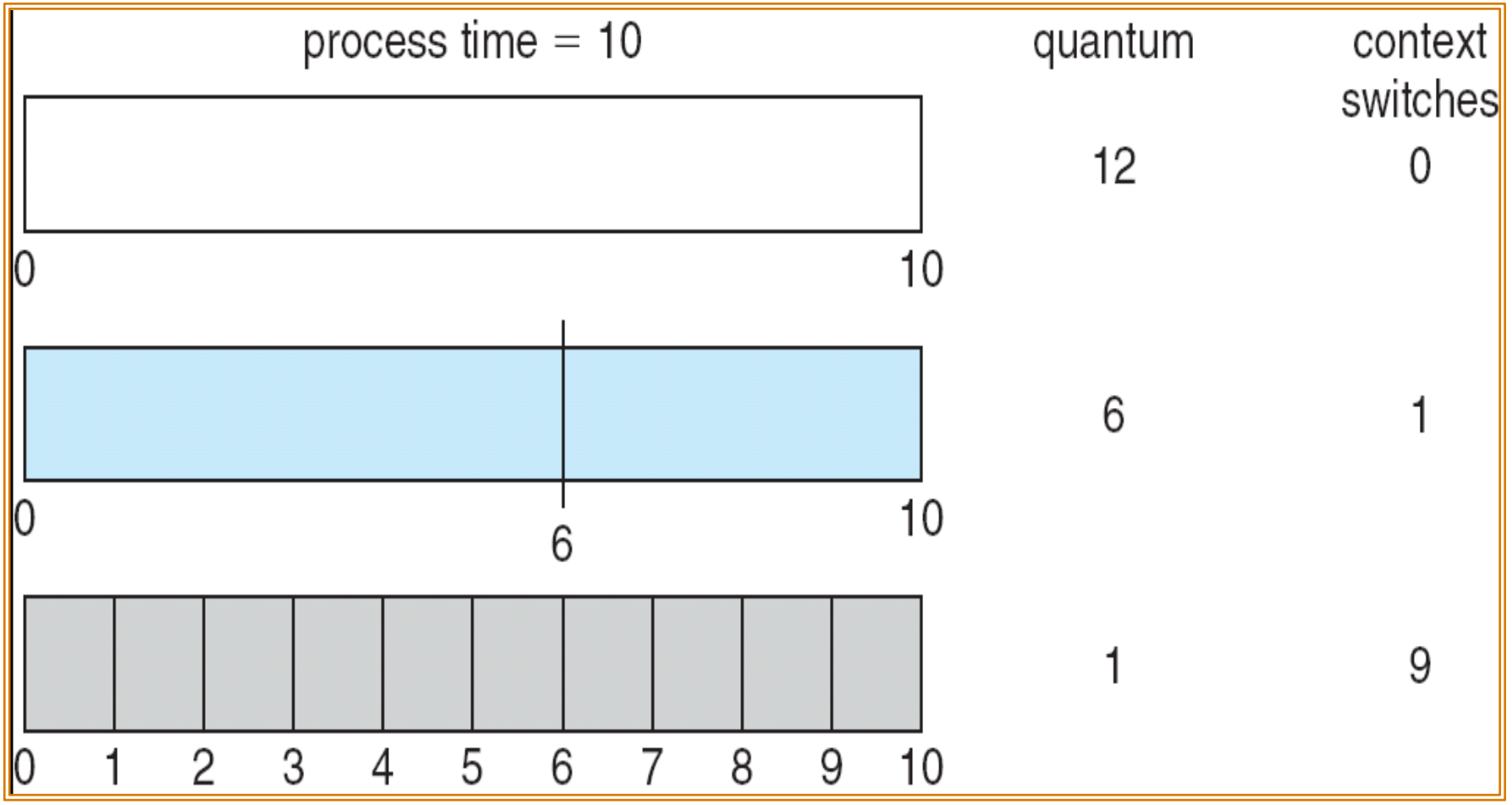


- Waiting times P_1 = 81, P_2 = 20, P_3 = 94, P_4 = 97
- Average waiting time = (81 + 20 + 94 +97)/4 = 73

Round Robin (RR)

- Performance of the RR algorithm depends heavily on the size of the time quantum.
- If the time **quantum is extremely large**, the RR policy is the same as the FCFS policy
- If the **time quantum is extremely small** (say, 1 millisecond), the RR approach is called processor sharing and creates the appearance that each of n processes has its own processor running at 1/n the speed of the real processor. (context switching overhead)

Time Quantum and Context Switch Time



Multilevel Queue Scheduling

- Multilevel queue is used when processes can be **classified into groups** based on some characteristic like response time, priority etc like
  - ✓ foreground (interactive)
  - ✓ background (batch)
- Ready queue is partitioned into separate queues and each queue has its own scheduling algorithm
  - ✓ foreground – RR
  - ✓ background – FCFS
- Processes are permanently assigned to one queue, generally based on some property of the process, such as process priority, or process type.

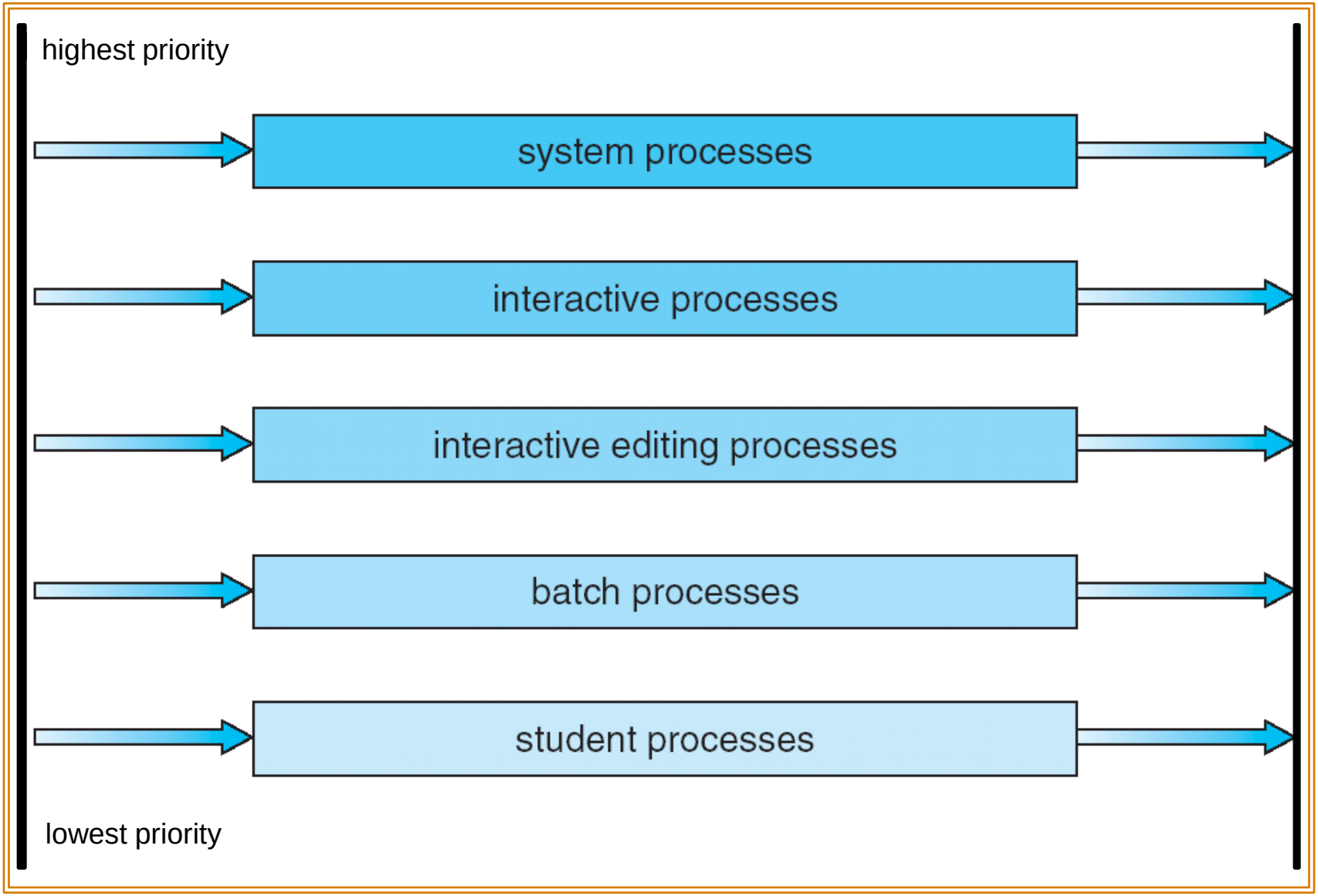
Multilevel Queue (Continued)

- Scheduling must be done between the queues. Two options
  - ✓ **Fixed priority preemptive scheduling;** Each queue has absolute priority over lower-priority queues. 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
  - ✓ Since processes cannot change their foreground or background nature (move from one queue to another), this scheme is inflexible.

Multilevel Queue Scheduling



Multilevel Feedback Queue Scheduling

- Multilevel feedback-queue scheduling algorithm allows a process to **move between queues**.
- Processes are scheduled according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- In addition, a process that waits too long in a lower- priority queue may be moved to a higher-priority queue. This form of **aging** prevents starvation.

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**: The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty.
  - ✓ A process entering the ready queue is put in queue 0 and given a time quantum of 8 milliseconds.
  - ✓ If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds.
  - ✓ If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

Multilevel Feedback Queue (Continued)

- Hence highest priority is given to any process with a CPU burst of 8 milliseconds or less as such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.
- Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes.
- Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

Multilevel Feedback Queues