CPU Scheduling

FCFS, SJF, RR, Priority, Multilevel Queue
&
Multilevel Feedback Queue

Introduction

Multiprogramming:

- Multiple programs are loaded into memory for execution.
- A computer running more than one program at a time like running PowerPoint and Chrome.

Multiprocessing:

- Two or more CPUs (processors) within a single computer system.
- There are multiple processors available, multiple processes can be executed at a time. These multi processors share the computer bus, sometimes the clock, memory and peripheral devices also.

Introduction

Multitasking:

- Execution of multiple tasks at a time.
- Multitasking is a logical extension of multi-programming.
- **Multiprogramming** works solely on the concept of context switching whereas **multitasking** is based on time sharing alongside the concept of context switching.

Introduction

Multithreading:

- Multithreading is an extension of multitasking.
- Multithreading allows a single process to have multiple code segments (i.e., threads) running concurrently within the “context” of that process.
- For example: VLC media player, where one thread is used for opening the VLC media player, one thread for playing a particular song and another thread for adding new songs to the playlist.

Introduction

- CPU scheduling is the basis of multiprogrammed operating systems.
- By switching the CPU among processes, the operating system can make the computer more productive.
- In uni-programming system, time spent waiting for I/O is wasted and CPU is free during this time.
- In multiprogramming systems, one process can use CPU while another is waiting for I/O.
- Several processes are kept in memory at one time. When one process has to wait, OS takes the CPU away from that process and gives the CPU to another process. This pattern continues.
- Every time one process has to wait, another process can take over use of the CPU.

CPU Scheduler

- Whenever the CPU becomes idle, OS must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the CPU scheduler.
- The CPU scheduler selects from among the processes in ready queue, and allocates a CPU core to one of them.
- Ready queue is not necessarily a first-in, first-out (FIFO) queue. E.g. FIFO queue, a priority queue, a tree, or simply an unordered linked list.

Preemptive & Nonpreemptive Scheduling

- 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 state
  4) Process 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.

Preemptive & Nonpreemptive Scheduling

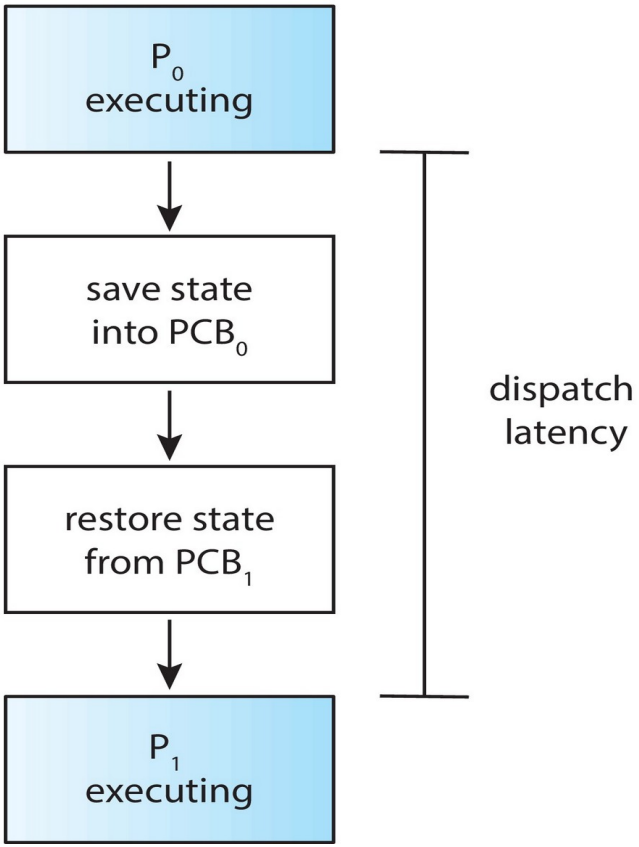
- In **preemptive scheduling**, the CPU is allocated to the processes for a limited time.
- In **Non-preemptive** scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state.
- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **nonpreemptive**.
- Otherwise, it is **preemptive**.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

Preemptive Scheduling & Race Conditions

- Preemptive scheduling can result in race conditions when **data are shared** among **several processes**.
- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run.
- The second process then tries to read the data, which are in an inconsistent state.

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU 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.
- **Arrival Time** – Time at which the process arrives in the ready queue.
- **Completion Time –** Time at which process completes its execution.
- **Burst Time –** Time required by a process for CPU execution.

Turn Around Time = Completion Time – Arrival Time

Waiting Time = Turn Around Time – Burst Time

Objectives of Process Scheduling Algorithm

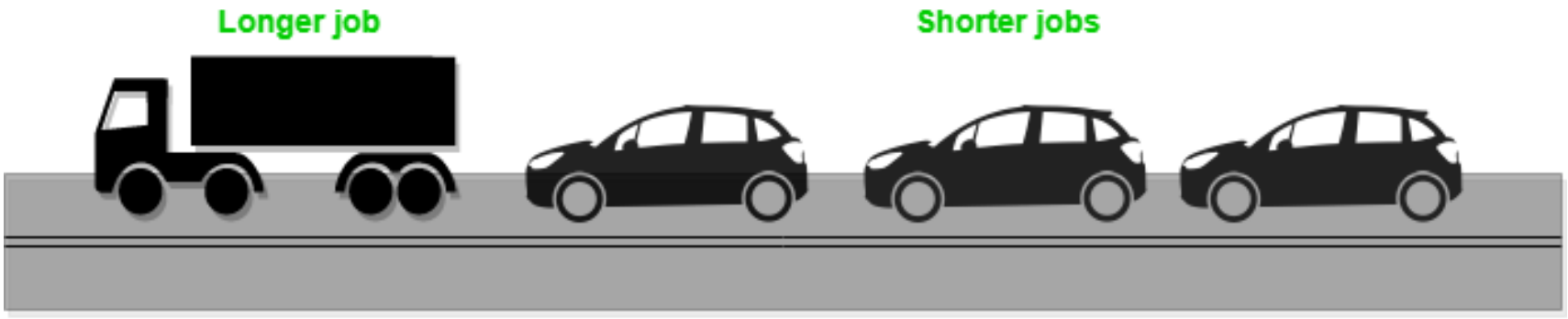
- **Max CPU utilization** (Keep CPU as busy as possible)
- **Max throughput** (Number of processes that complete their execution per time unit)
- **Min turnaround time** (Time taken by a process to finish execution)
- **Min waiting time** (Time a process waits in ready queue)
- **Min response time** (Time when a process produces first response)

First- Come, First-Served (FCFS) Scheduling

- FCFS scheduling algorithm is nonpreemptive.
- Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- A process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.

First- Come, First-Served (FCFS) Scheduling

- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.
- The average waiting time under the FCFS policy is often quite long.
- It suffers from convoy effect.
- **Convoy Effect** is phenomenon associated with the FCFS algorithm, in which the whole Operating System slows down due to few slow processes.



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

FCFS Scheduling

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

- SJF scheduling algorithm is nonpreemptive.
- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- 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.

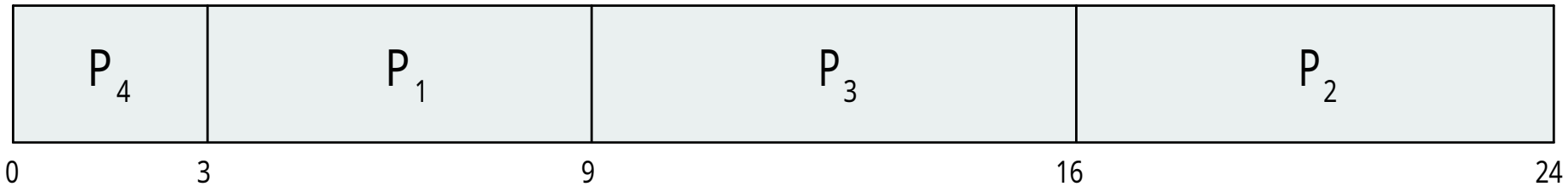
Shortest-Job-First (SJF) Scheduling

- SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst.
- Preemptive version called **shortest-remaining-time-first**.

Example of Shortest-Job-First

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



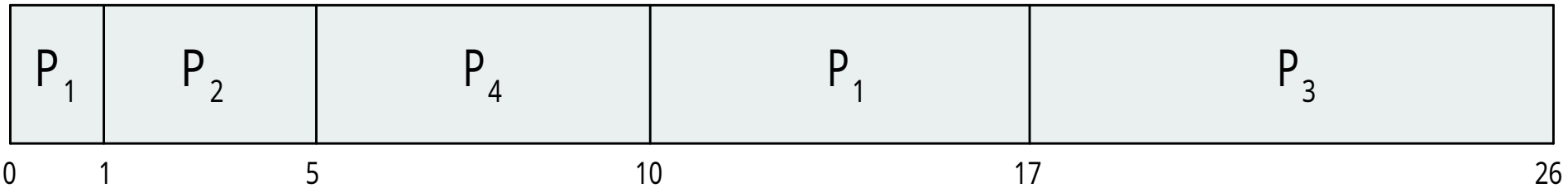
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = ((10-1)+(1-1)+(17-2)+(5-))/4 = 26/4 = 6.5

Round Robin (RR)

- The round-robin (RR) scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time, called a **time quantum** or **time slice**, is defined.
- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is **preempted** and added to the end of the ready queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

Round Robin (RR)

- 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.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

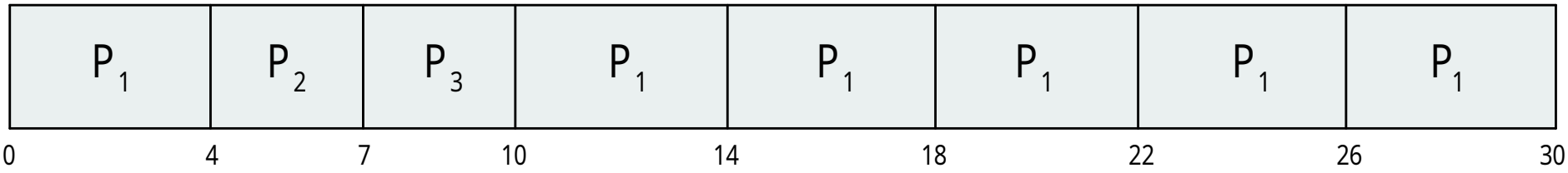
Round Robin (RR)

- The performance of the RR algorithm depends heavily on the size of the time quantum.
- At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches.

Example of RR with Time Quantum = 4

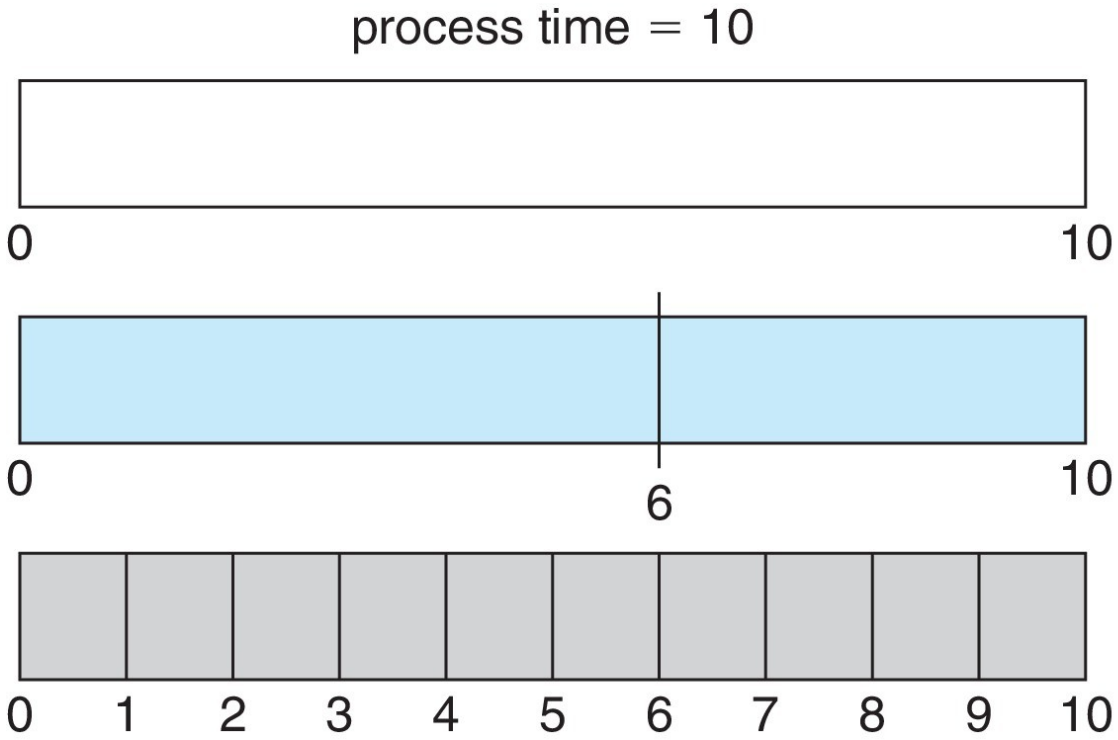
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

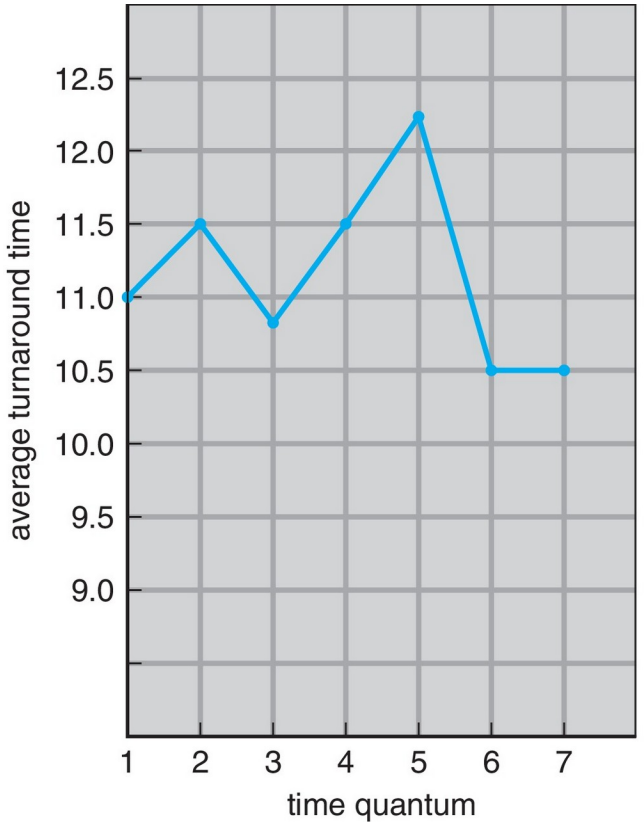
Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies with Time-Quantum

- Turnaround time also depends on the size of the time quantum. As we can see from Figure, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases.
- In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Round Robin (RR)

- The time quantum should be large compared with the contextswitch time, it should not be too large.
- A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

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).
- Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst.
- The larger the CPU burst, the lower the priority, and vice versa

Priority Scheduling

- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- However, there is no general agreement on whether 0 is the highest or lowest priority.
- Some systems use low numbers to represent low priority
- Others use low numbers for high priority.
- Priorities can be defined either **internally** or **externally**.

Priority Scheduling

- **Internally** defined priorities use some measurable quantity or quantities to compute the priority of a process.
- For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
- **External** priorities are set by criteria outside the operating system.
- For example, the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority Scheduling

- Priority scheduling can be either preemptive or nonpreemptive.
- **In preemptive:** when a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- In **nonpreemptive:** priority scheduling algorithm will simply put the new process at the head of the ready queue.

Priority Scheduling

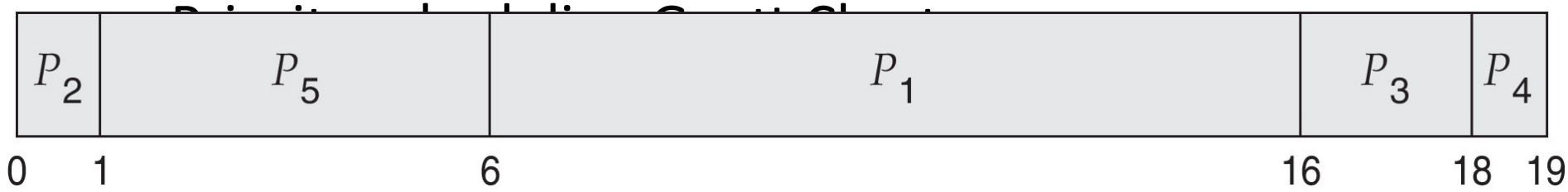
- A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**.
- A process that is ready to run but waiting for the CPU can be considered blocked.
- A priority scheduling algorithm can leave some low-priority processes waiting indefinitely.
- Problem **Starvation:**– low priority processes may never execute.
- Solution **Aging:**– as time progresses increase the priority of the process.

Priority Scheduling

- Another option is to combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the **same priority** using round-robin scheduling.
- The same priority processes will execute in round-robin order until they complete.

Example of 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

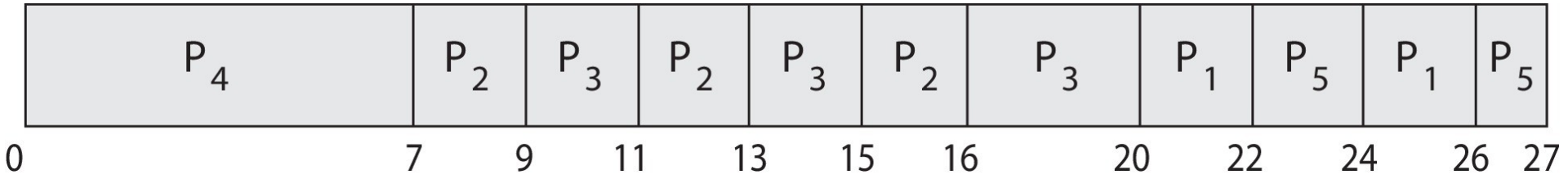


- Average waiting time = 8.2

Priority Scheduling with Round-Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority.
- Processes with the same priority run round-robin
- Gantt Chart with time quantum = 2

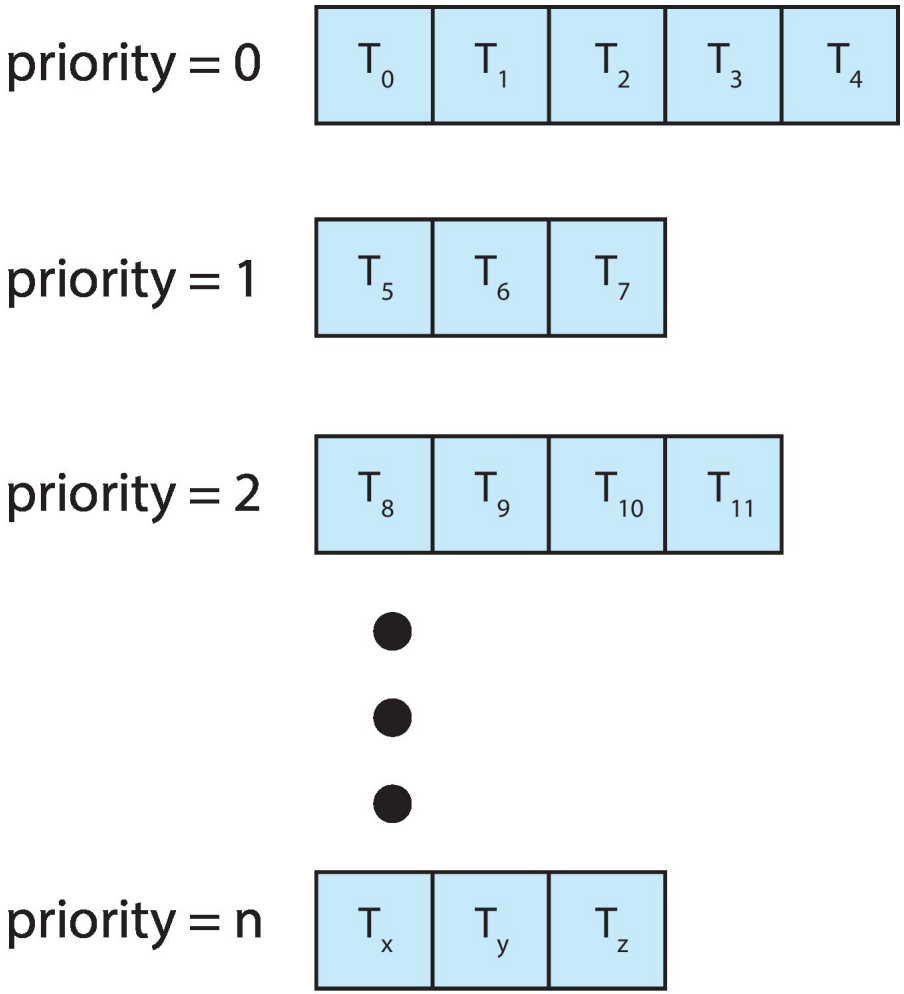


Multilevel Queue

- With both priority and round-robin scheduling, all processes may be placed in a single queue
- The scheduler then selects the process with the highest priority to run.
- Depending on how the queues are managed, an O(n) search may be necessary to determine the highest-priority process.
- In practice, it is often easier to have separate queues for each distinct priority.
- Priority scheduling simply schedules the process in the highest-priority queue.

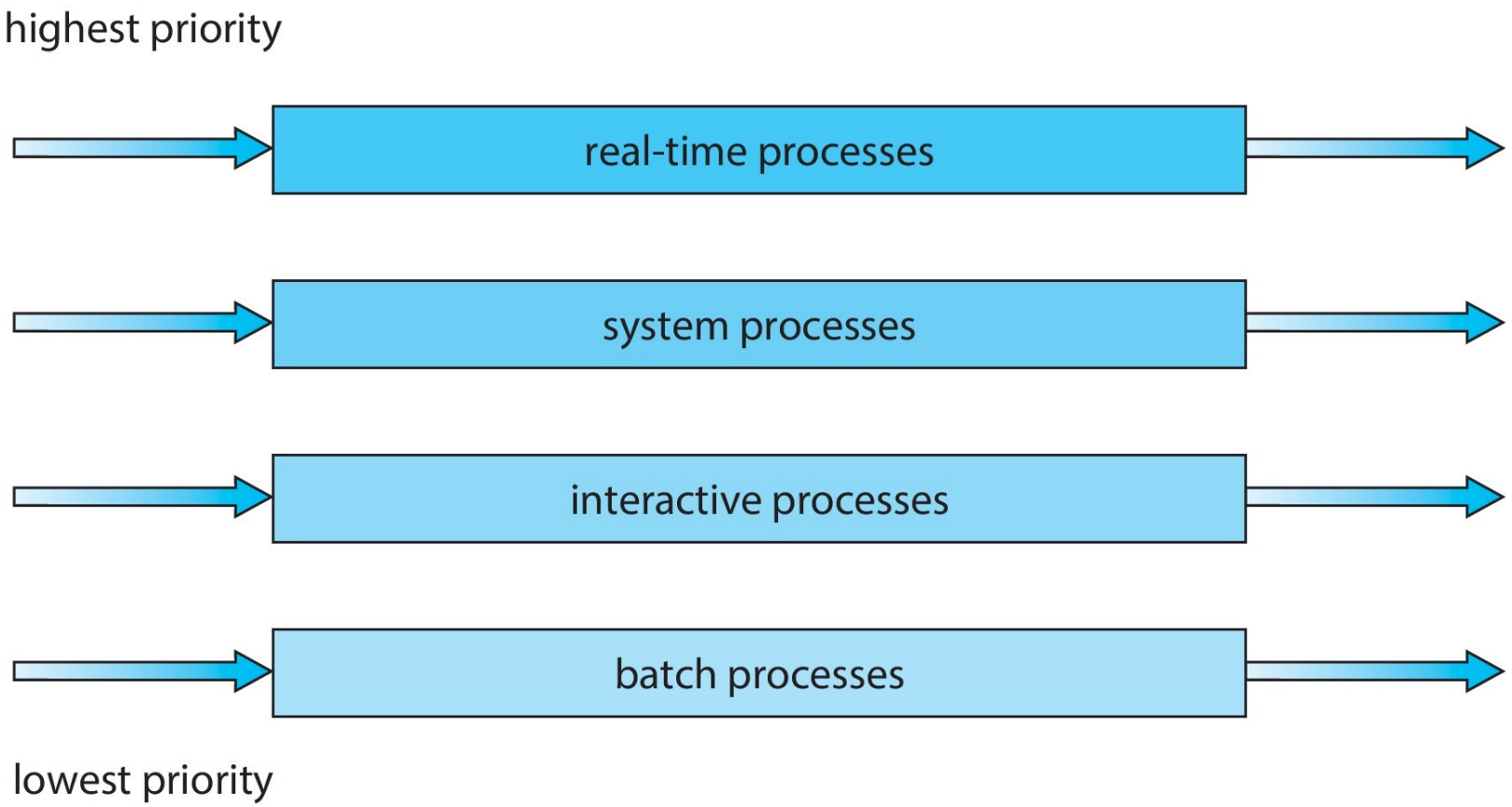
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Multilevel Queue

- Prioritization based upon process type



Multilevel Queue

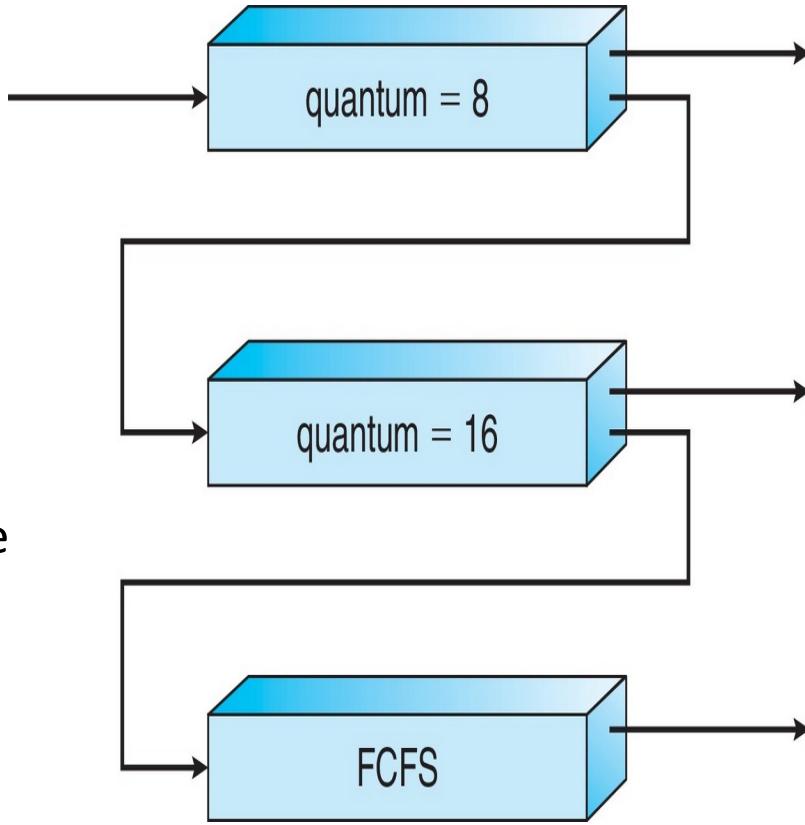
- In addition, each queue may have its own scheduling algorithm.
- For example, the foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.
- In addition, there must be scheduling among the queues.
- Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the **CPU time**, which it can then schedule among its various processes.
- For example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

Multilevel Feedback Queue

- Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
- For example, processes do not move from one queue to the other, since processes do not change their foreground or background nature.
- The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues.
- The idea is to separate processes 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.

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 process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Multilevel Feedback Queue

- 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.
- Aging can be implemented using multilevel feedback queue

System calls

- A process can retrieve and change its nice value with the nice function.
- `nice()`, a process can affect only its own nice value; it can't affect the nice value of any other process.

```
int nice(int incr);
```

- A process' nice value is a non-negative number.
- The `incr` argument is added to the nice value of the calling process.
- If `incr` is too large, the system silently reduces it to the maximum legal value.
- Similarly, if `incr` is too small, the system silently increases it to the minimum legal value.

System calls

- Calling the `nice()` function has no effect on the priority of processes or threads with policy `SCHED_FIFO` or `SCHED_RR`.
- Upon successful completion, nice() returns the new nice value minus {NZERO}. Otherwise, -1 is returned.
- The nice() function will fail if: The `incr` argument is negative and the calling process does not have appropriate privileges.

System calls

```
#include <sys/resource.h>
int getpriority(int which, id_t who);
```

- The `getpriority` function can be used to get the nice value for a process.
- The which argument can take on one of three values:
  - `PRIO_PROCESS` to indicate a process,
  - `PRIO_PGRP` to indicate a process group, and
  - `PRIO_USER` to indicate a user ID.
- The `which` argument controls how the `who` argument is interpreted and the `who` argument selects the process or processes of interest.
- If the `who` argument is 0, then it indicates the calling process, process group, or user

System calls

```
#include <sys/resource.h>
int setpriority(int which, id_t who, int value);
```

- The value is added to NZERO and this becomes the new nice value.