CPU Scheduling

CISC 3320 — CUNY Brooklyn College Lecture Notes

CPU Scheduling

In this chapter, we will:

1. Explain the difference between cooperative and preemptive CPU schedulers,
2. Introduce what criteria a CPU scheduler could aim at achieving,
3. Study various CPU scheduling algorithms, the ideas behind them, and their advantages/disadvantages.
4. Explain how scheduling happens when several processes are present, and
5. Discuss how CPU scheduling takes place in real-time operating systems.

CPU Scheduling

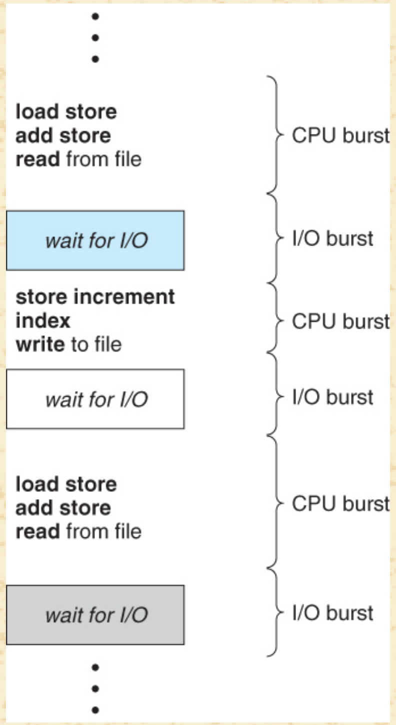
As we learned in the previous topics, when a process is running on a CPU for a while, it might block (move into waiting state) for I/O, interrupt, or other event that temporarily prevents it from running currently on the CPU.

While the process is blocking, if we keep the CPU idle (that is, doing essentially nothing,) we will be wasting precious time that would be otherwise spent on running other, ready programs.

To prevent a CPU from sitting idle, the **CPU scheduler**, which is software that is a part of the OS, is responsible for deciding how and when processes will receive the CPU to run thereon.

The major challenge of the scheduler is to make the usage of the CPU as efficient as possible (we will explain what "efficient" could mean when we speak about Scheduling Criteria) and as fair as possible. These two terms are usually some subjective priorities that the developers of the OS wish the scheduler to exercise.

CPU Scheduling

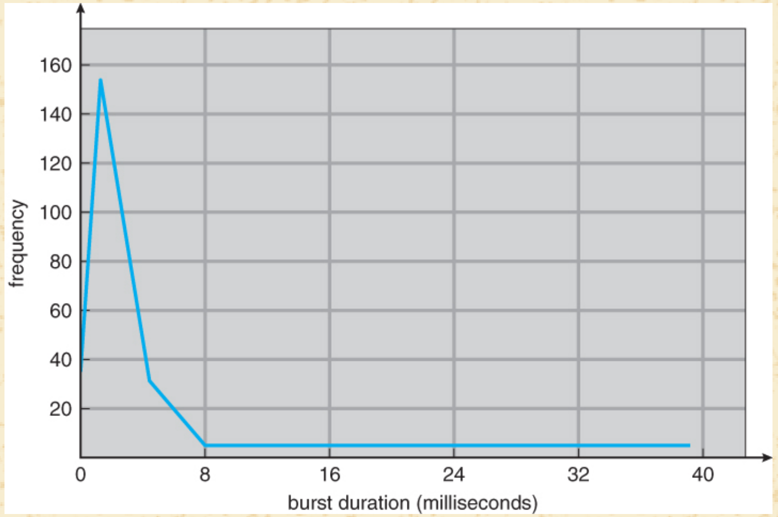


Alternating sequence of CPU and I/O bursts. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

CPU Scheduling

A **burst** is short amount of time (usually, several milliseconds) that a process spends doing some action, which is either a CPU-computation (**CPU burst**) or waiting for I/O (**I/O burst**.)

Research has found that the duration of CPU bursts is distributed as follows:



Histogram of CPU-burst durations. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

CPU Scheduling

Any questions?

CPU Scheduling

The CPU Scheduler is invoked and must make a decision in every one of the following 4 scenarios:

1. When the running process switches to a waiting state, such as for an I/O request or a call to `wait( )`.
2. When the running process switches to a ready state, for example in response to an interrupt.
3. When a process switches from the waiting state to the ready state, such as when an I/O event completed or at a return from a call to `wait( )`.
4. When a process terminates.

In case the running process starts waiting or terminates, the Scheduler ***must*** run a different process on the CPU.

A CPU Scheduler that lets a process run on the CPU until it starts waiting or terminates is called a **non-preemptive** (also: **cooperative**) scheduler. Otherwise, if the Scheduler forcefully removes a running process from the CPU (and puts it into the ready state) and instead lets a different process run on the CPU, the Scheduler is called **preemptive**.

CPU Scheduling

Most modern operating systems use preemptive CPU Schedulers.

The **dispatcher** is another operating system software; its responsibility is to give control of the CPU to the process that the CPU Scheduler chose to run.

The dispatcher performs context switches between processes and threads, switches to user mode from kernel mode, and tells the CPU at which instruction the process or thread need to start executing.

It must work as fast as possible since it will be invoked at every context switch. The time it takes for the dispatcher to execute is called the **dispatch latency**.

CPU Scheduling

This video demonstrates the activity of the CPU Scheduler and the dispatcher:

https://www.youtube.com/watch?v=THqcAa1bbFU

Scheduling Criteria

What is defined to be the "best" order in which programs run? "Best" is a relative notion!

It depends on the goals that we are trying to achieve when running processes on a specific device. We call such goals **scheduling criteria**, some of which are:

- **CPU utilization** = how much of the time a CPU is working (how busy it is.) The ideal is 100% utilization, while, in reality, it should range from 40% to 90%.
- **Throughput** = how many processes completed in a unit of time. The greater the number of processes who got to complete, the greater the throughput is.
- **Turnaround time** = the time it takes a particular process to complete, from start to finish.
- **Waiting time** = how much time on average processes wait in the waiting queue to run on a CPU.
- **Load average** = how many processes on average are sitting in the ready queue to run on a CPU.
- **Response time** = the time it takes on average for a process to respond to a user's command.

CPU Scheduling

Any questions?

Scheduling Algorithms

We now describe several **scheduling algorithms** that the CPU Scheduler could use to decide in what order processes will run on the CPU. Here, we assume that there is one CPU.

1. **First-Come First-Serve (FCFS) Scheduling**.
    - In this algorithm, whichever process arrives first will run first.
    - The CPU Scheduler uses a queue (FIFO) data structure to implement the algorithm.
    - The FCFS algorithm is cooperative in nature since the Scheduler won't forcefully evict a process from running on the CPU, and the process will keep running there until it willingly releases the CPU.
    - **Advantage**: this is the simplest scheduling algorithm to implement.
    - **Disadvantage**: the average waiting time is long.
    - As a consequence, in interactive systems, users might need to wait for an exceptionally long time to get a response from an application that they run. Also, because many processes will wait for a single, long process to finish using the CPU, other devices, like I/O devices, will remain idle and unutilized for a long time.

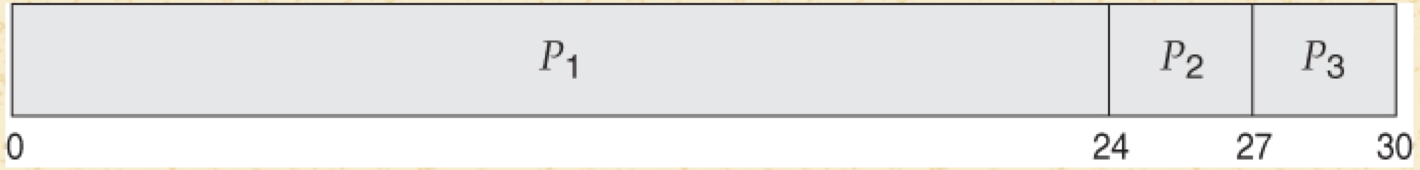
Scheduling Algorithms

Example: Suppose that 3 processes, P_1, P_2, and P_3, arrive at the ready queue:

Process	Burst Time (in milliseconds)
P_1	24
P_2	3
P_3	3

Burst Time is the time that a process will run on the CPU until it starts waiting for I/O.

If the Scheduler uses FCFS, the processes will run in the order depicted in the following **Gantt diagram**:



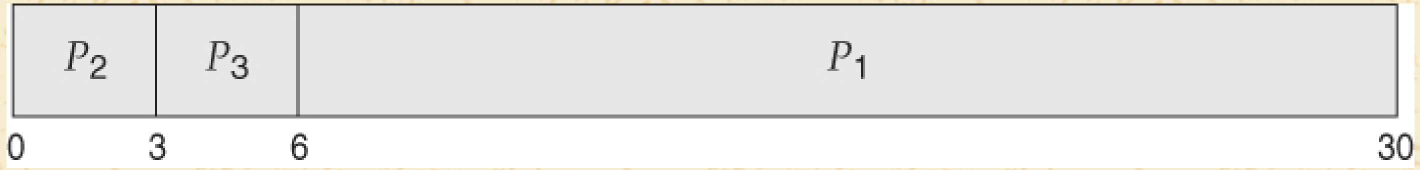
The schedule of the 3 processes under FCFS. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

Scheduling Algorithms

Notice that the long process P_1 will prevent the rest of the processes from running on the CPU for 24 milliseconds.

Specifically, the **average waiting time** in this schedule is (0 + 24 + 27) / 3 = 17 milliseconds. The **average response time** in this case is the same as the average waiting time, which is (0 + 24 + 27) / 3 = 17 ms.

If processes P_2 and P_3 would arrive just before P_1, the schedule would be much different:



The schedule of the 3 processes under FCFS. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

Above, since the smaller processes get to complete the first, the average waiting time drops to (0 + 3 + 6)/3 = 3 milliseconds. In other words, the average waiting time highly depends on when each process arrives.

Scheduling Algorithms

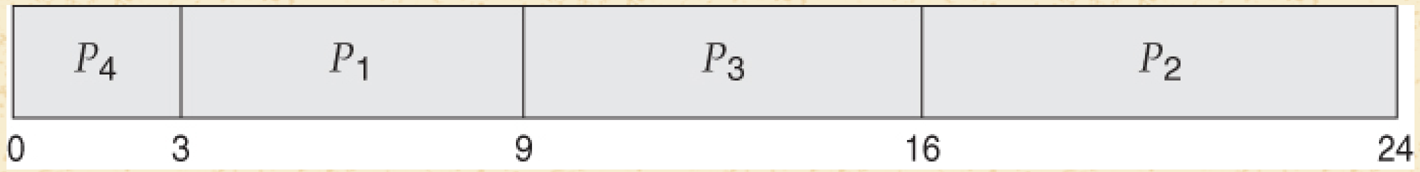
2. **Shortest-Job-First (SJF) Scheduling**.
    - In this algorithm, when several processes arrive at the same time, processes with shorter burst times will run first.
    - SJF could be either cooperative or preemptive. When the preemptive version of SJF is used, a process will be removed from the CPU when another process with a shorter burst time arrives at the ready queue.
    - **Advantage**: It will maximize the throughput and minimize the average waiting time.
    - **Disadvantage**: We don't know what the future burst time of a process will be.
    - To address this, we could do either of the following:
        1. Ask the user to indicate how long they want the process to run.
        2. Gather statistics on how long the burst time of a process of a certain type is.
        3. [Best approach:] Predict the length of the next burst, based on some historical measurement of recent burst times for this process.

Scheduling Algorithms

Example of a cooperative SJF: Suppose that 4 processes, P_1, P_2, P_3, and P_4 arrive at the ready queue:

Process	Burst Time (in milliseconds)
P_1	6
P_2	8
P_3	7
P_4	3

The processes will run in the following order:



The schedule of the 4 processes under SJF. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

Scheduling Algorithms

In the diagram above, process P_4, which has the shortest burst time of 3 ms, started running on the CPU first. Afterwards, process P_1 with the next shortest burst time of 6 ms ran, and so on. The average waiting time in this schedule is (0 + 3 + 9 + 16)/4 = 7 milliseconds, which is the same as the average response time. Note that if FCFS were used, the average waiting time would be 10.25 ms.

As we mentioned before, it is possible to prove mathematically that SJF yields the shortest possible average waiting time.

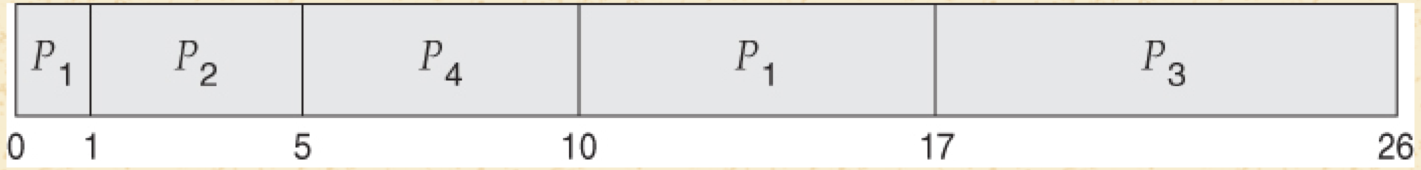
Note that those processes with the longer burst times will run the last, after the smaller processes finished running.

Scheduling Algorithms

Example of a preemptive SJF: Suppose that 4 processes, P_1, P_2, P_3, and P_4 arrive at the ready queue:

Process	Arrival Time (in milliseconds)	Burst Time (in milliseconds)
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

The processes will run in the following order:



The schedule of the 4 processes under Preemptive SJF. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

Scheduling Algorithms

In the diagram above, process P_1, which arrives first, will start running on the CPU. At millisecond 1, when process P_2 with a shorter burst time arrives, the Scheduler will forcefully remove P_1 from the CPU and let P_2 run instead. Since the next arriving processes, P_3 and P_4, have longer burst times than this of P_2, the Scheduler won't remove P_2 from the CPU. After P_2 finishes running, the Scheduler will run the remaining processes as follows: P_4 (since its burst time is 5 ms,) then P_1 (since its remaining burst time is 8 - 1 = 7 ms,) and finally P_3 (since its burst time is the longest: 9 ms.)

Because the preemptive version of the SJF algorithm relies on the remaining burst time, we often call this algorithm the **shortest remaining time first (SRTF) scheduling**.

Finally, the average waiting time in this schedule is ((0 - 0) + (10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)) / 4 = 26 / 4 = 6.5 ms. The reason we do subtraction inside the parentheses is that we subtract the arrival times from the wait times. If we use the cooperative version of SJF, the average waiting time would be 7.75 ms, and for FCFS it would be 8.75 ms.

The average resopnse time is ((0 - 0) + (1 - 1) + (5 - 3) + (17 - 2)) / 4 = 4.25 ms.

Scheduling Algorithms

Any questions?

Scheduling Algorithms

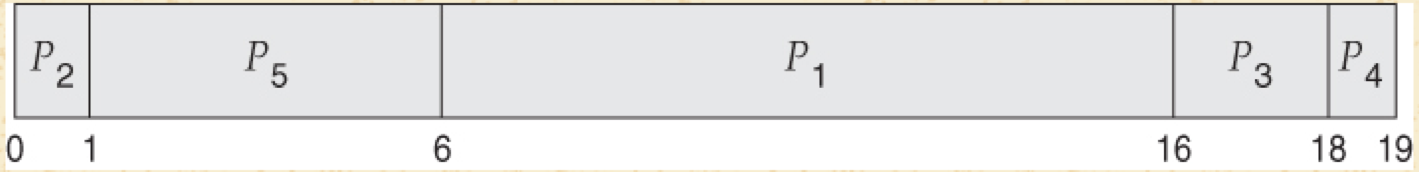
3. **Priority Scheduling**.
   - In this algorithm, every process is assigned a priority, which is usually given as a number. In the example that follows on the next slide, the smallest the number, the greater the priority is, and the earlier the process will run on the CPU.
   - Priority Scheduling is a generalization of SJF: in SJF, the priority is the inverse (1/x) of the burst time. That is, the shorter the burst time is, the greater the priority of the process is.
   - **Advantage**: Processes with the greatest priorities (those we need the most) will get to run first.
   - **Disadvantage**: A process with a small priority might never have a chance to run since new processes with greater priorities will keep arriving at the ready queue. This phenomenon is known as **indefinite blocking** or **starvation**.
   - To address this, we could apply the solution of **aging**, in which we "bump" the priority of processes based on how much they wait. That is, processes that got to wait too much time will have their priorities gradually increased, and therefore will eventually be guaranteed to run on the CPU.

Scheduling Algorithms

Example: Suppose that 5 processes, P_1, P_2, P_3, P_4, and P_5 arrive at the ready queue:

Process	Burst Time (in milliseconds)	Priority (1 = will run first)
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

The processes will run in the following order:



The schedule of the 5 processes under Priority Scheduling. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

Scheduling Algorithms

In the diagram above, process P_2, which has the greatest priority (priority 1), will be the first to run on the CPU. Then, processes with lesser priorities will get to run, until the process with the least priority, process P_4, will run.

The average waiting time in this schedule is (0 + 1 + 6 + 16 + 18) / 5 = 41 / 5 = 8.2 ms, which is the same as the average response time.

[Try at home: compute the average waiting time for the scenario on the previous slide if we were to use the FCFS, SJF, and SRJF algorithms.]

Scheduling Algorithms

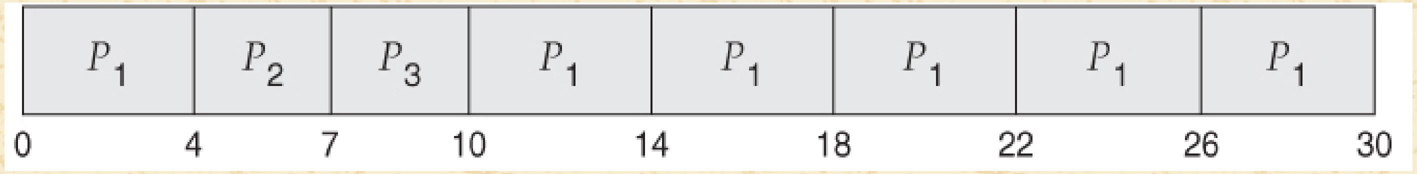
4. **Round-Robin (RR) Scheduling**.
   - In this algorithm, each process is assigned the same **time slice** (also called **time quantum**), during which the process can execute on the CPU. When the time slice passes, the Scheduler removes the process and brings a different process to run instead on the CPU.
   - When a process is removed from the CPU but some ms still remain in its burst time, the process will be moved to the end of the queue and will wait until the other processes use their time slices.
   - Processes get to run based on the time they arrive at the ready queue, so RR mostly resembles FCFS.
   - **Advantage**: The average response time decreases (processes respond faster) since all the processes equally share CPU time. Also, no starvation occurs for the same reason.
   - **Disadvantage**: Too large or too small time slices will cause issues. Too large time slices will make the algorithm behave more like FCFS. Too small time slices will increase the number of context switches and, hence, the extra time spent on performing context switches. In other words, it is difficult to select an optimal time slice.

Scheduling Algorithms

Example: Suppose that 3 processes, P_1, P_2, and P_3, arrive at the ready queue:

Process	Burst Time (in milliseconds)
P_1	24
P_2	3
P_3	3

If we set the time slice to be 4 ms, the processes will run in the order below:



The schedule of the 3 processes under RR. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

Scheduling Algorithms

In the diagram above, process P_1, which came in first, will start running on the CPU. Since P_1 can't finish running within only 4 ms of the time slice, it will be removed from the CPU in favor of the next processes in line.

A process can finish running in less than the given time slice if its burst time is less than 4 ms. This is exactly what each of processes P_2 and P_3 do.

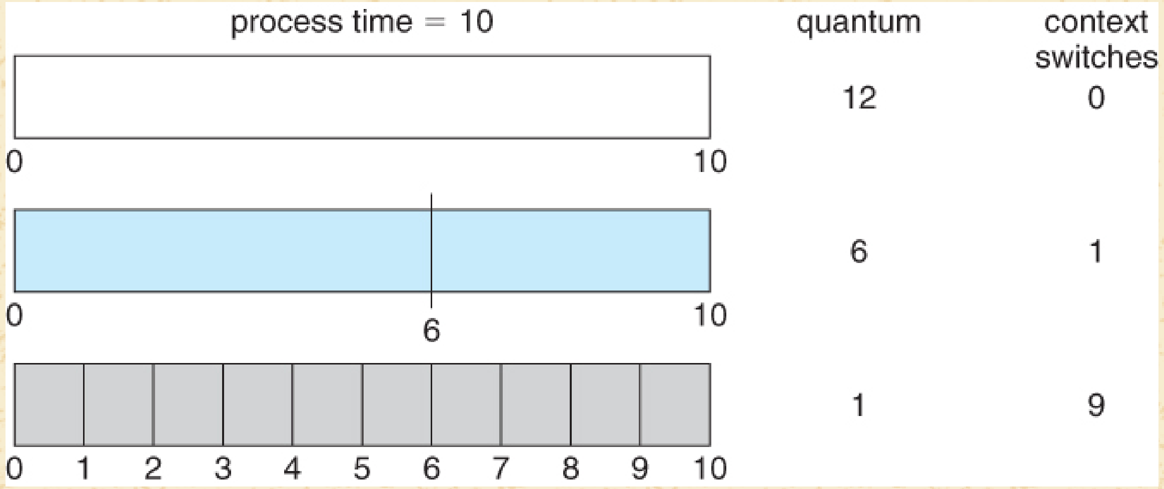
The average waiting time in this schedule is (0 + (10 - 4) + 4 + 7) / 3 = 17 / 3 = 5.66 ms. We count (10 - 4) since we need to find how much time process P_1 waited to get onto the CPU for the 2nd time, too.

The response time in this schedule is (0 + 4 + 7) / 3 = 11 / 3 = 3.66 ms.

[Remember that we define the response time to be the time it takes the process to get the CPU for the first time minus the arrival time. Since all 3 processes arrived at the same time (0th millisecond,) we don't subtract anything.]

Scheduling Algorithms

Here is a diagram that illustrates that decreasing the time slice will lead to many more context switches:



Time slice vs. number of context switches. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

To have fewer context switches, it is advisable to select the time slice so that 80 percent of the CPU bursts are shorter than the time slice.

Scheduling Algorithms

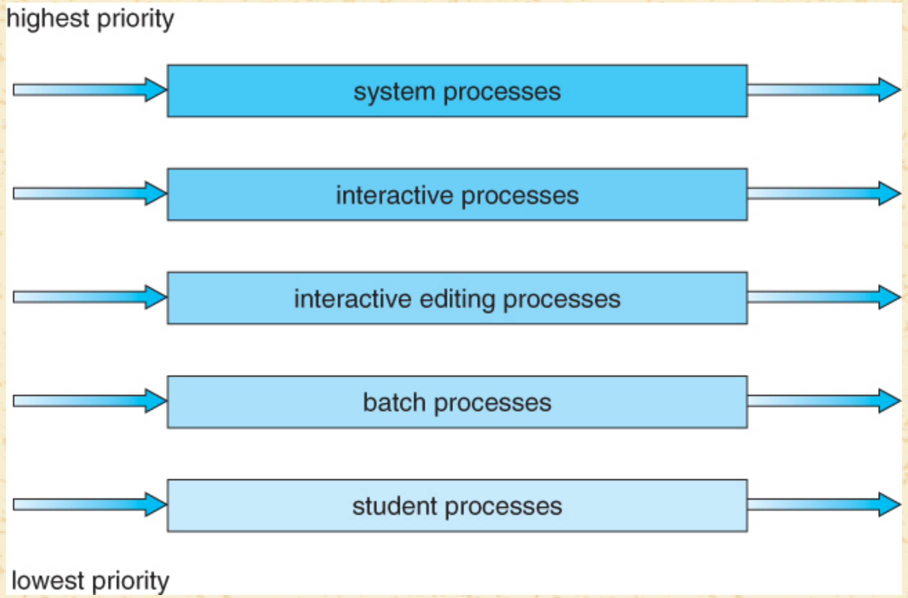
Any questions?

Scheduling Algorithms

5. **Multilevel Queue Scheduling**.
   - If the processes that run on a device can be grouped into clear categories, we could have a different scheduling algorithm for each group.
   - Every group of processes will be dedicated its own queue (hence the name "Multilevel Queue" algorithm.)
   - The Scheduler will also need to schedule the activity *between* the queues (that is, the processes of which queue run first.)
   - In this algorithm, a process cannot move from one queue to another: it is fixed to a particular queue.
   - **Advantage**: Scheduling could be more granulated: we will 'match' the best algorithm in each group's queue based on the traits of the processes in that group.
   - **Disadvantage**: It might be challenging to decide to which group a process will belong. Also, the implementation of this algorithm could be complicated since we need to both schedule processes within a queue *and* between queues.

Scheduling Algorithms

Example: Each one of the following groups of processes uses its own algorithm. We use Priority Scheduling for between-queue scheduling.



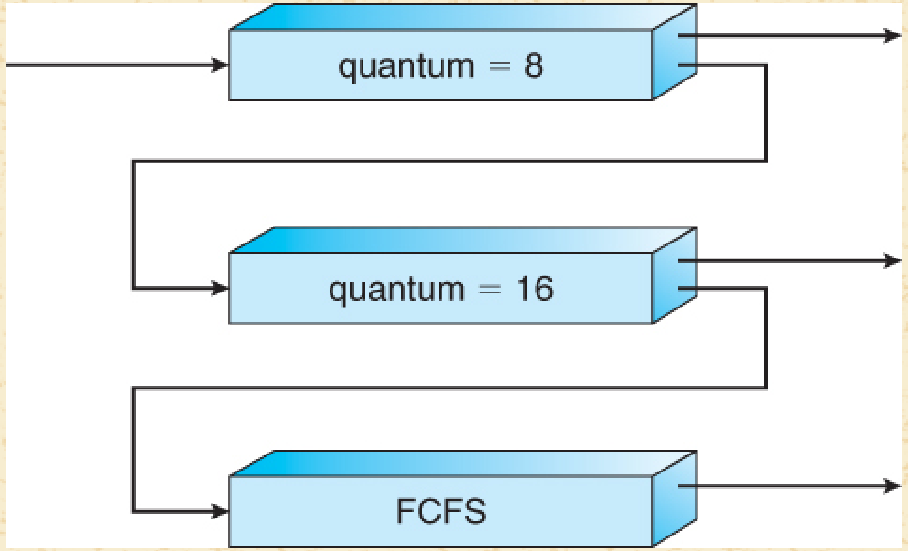
Multilevel queue scheduling. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

Scheduling Algorithms

6. **Multilevel Feedback Queue Scheduling**.
   - This is the same as Multilevel Queue Scheduling, except that a process can move from one queue to another.
   - A few reasons that we could decide to move a process to another queue are:
     1. During the process's execution, it might change its nature, such as being more CPU-bound or more I/O-bound, so moving it to another queue will make its execution (and this of other processes) more efficient.
     2. If priorities are used for scheduling, we might move a process to a queue whose priority is larger if the priority of the process increases due to aging.
   - **Advantage**: This algorithm is very flexible and lets us change process scheduling during the execution of the process and not only when it starts running (as we do in the Multilevel Queue Scheduling algorithm.)
   - **Disadvantage**: It is very difficult to implement this algorithm due to its complexity.

Scheduling Algorithms

Example: This diagram shows how a process could move from one queue to another. Below, "quantum = 8" stands for the Round Robin algorithm with a time slice of 8 ms.



Multilevel feedback queue scheduling. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

Multiple-Processor Scheduling

Scheduling becomes more difficult when several CPUs are installed since the OS needs to divide work among the CPUs to keep each one busy.

In general, there might exist different kinds of CPUs on the same computer (heterogeneous), or all the CPUs could be of the same type (homogenous). To keep things simple for the sake of this discussion, we assume that the CPUs are homogenous.

An operating system could exhibit **asymmetric multiprocessing**, in which one 'boss' CPU controls the other CPUs, which is pretty simple to implement.

Otherwise, it could use **symmetric multiprocessing (SMP)**, in which each processor schedules its own jobs. Most moderns operating systems use symmetric multiprocessing.

Multiple-Processor Scheduling

Processors contain cache memory, which speeds up repeated accesses to the same memory locations.

If the operating system were to switch a process from one CPU to another, this would require the entire cache to be copied over.

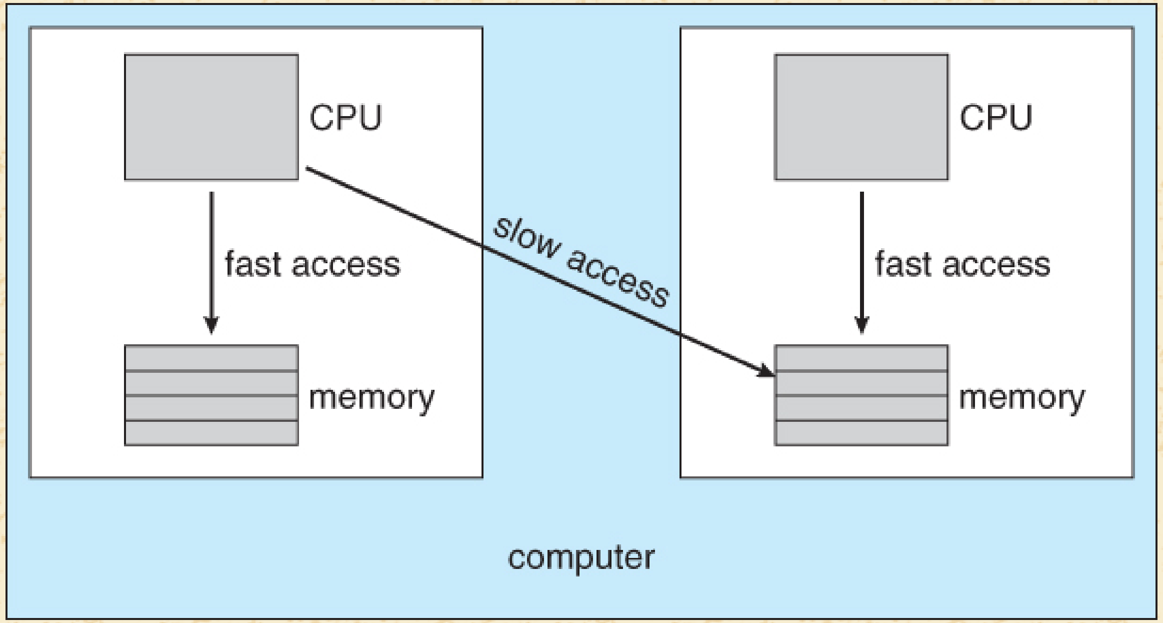
As such, operating system use **processor affinity**, which is some form of binding a process to a particular CPU.

Soft affinity occurs when the system attempts to keep processes on the same processor but makes no guarantees.

Hard affinity is a guarantee that a process won't switch to other CPUs. Linux uses hard affinity (there's a C function that does so: `sched_setaffinity()`.)

Multiple-Processor Scheduling

When every CPU has it own cache memory, we could let a CPU access its own cache and the cache of other CPUs, but at different speeds (**Non-Uniform Memory Access** = NUMA):



NUMA and CPU scheduling. Taken from Bell, John T. "CPU Scheduling." University of Illinois, Chicago.

Real-Time CPU Scheduling

Recall that a Real-Time OS is one whose processes need to complete within defined time limits.

There are 2 types of Real-Time systems:

1. **Soft real-time systems**: in such OSs, performance will suffer if tasks don't get completed in time. Example: gaming console.
2. **Hard real-time systems**: these OSs will totally fail their purpose if tasks don't get completed in time. Example: medical or military equipment.

Real-time systems require pre-emptive priority-based scheduling. Whenever a task must run, the OS will generate an interrupt that will cause the currently-running process to be removed from the CPU in favor of the timed process.

Hard real-time systems must provide 100% guarantees that a task's scheduling needs can be met. One approach is to use an **admission-control algorithm**, in which each task must specify its needs at the time it attempts to launch.

Real-Time CPU Scheduling

Any questions?

CPU Scheduling: Sources

Bell, John T. "CPU Scheduling." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/6_CPU_Scheduling.html

Chapter 5: The Process (pp. 96-98).
Wienand, Ian (2019). Computer Science from the Bottom Up. This work is licensed under the Creative Commons Attribution-ShareAlike3.0 Unported License. URL: https://www.bottomupcs.com/csbu.pdf