Topic 08: CPU Scheduling

Reading: 7.1, 7.4
Next Reading: None

- Process (or thread or CPU) scheduling is a subset of the general problem of *resource* scheduling. Resources are the things operated on by processes, and range from CPU time to disk space to network bandwidth.

- Resources fall into two classes:

  - *Preemptible*: Can take the resource away, use it for something else, then give it back later. Examples: CPU, I/O device

  - *Non-preemptible*: once given, it can't be reused until process gives it back. Examples are file space, terminal, printer, and maybe memory.

  - This distinction is a little arbitrary, since anything is preemptible if it can be saved and restored. It generally measures the difficulty of preemption.

- OS makes two related kinds of decisions about resources:

  - *Allocation*: who gets what. Given a set of requests for resources, which processes should be given which resources in order to make the most efficient use of the resources? Implication is that resources aren't easily preemptible.

  - *Scheduling*: how long can they keep it. When more resources are requested than can be granted immediately, in which order should they be serviced? Examples are CPU scheduling (one CPU, many processes), memory scheduling in virtual memory systems. Implication is that resource is preemptible.

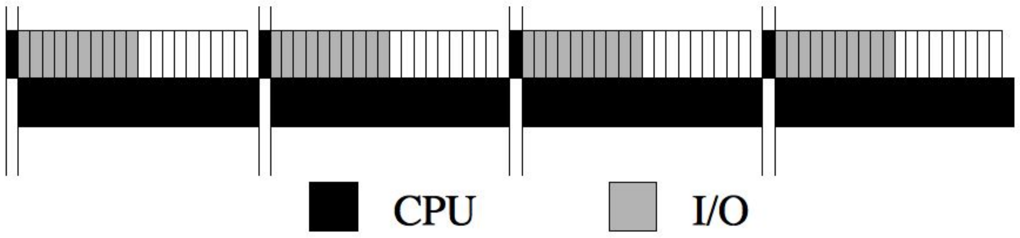
- Resource #1: the processor.
  - Processes may be in any one of three general scheduling states:
    - *Running*.
    - *Ready*. That is, waiting for CPU time. Scheduler and dispatcher determine transitions between this and running state.
    - *Blocked*. Process cannot be run because it is waiting for some other event: input from user, disk I/O, another process to do something, etc.

- Possible goals for scheduling disciplines:
  - Keep users happy.
  - Efficient use of resources. (keep CPU and disks busy).
  - Maximize *throughput*. Throughput is a measure of work, e.g. jobs per hour. (maximize number of jobs per hour)
  - Minimize overhead. (context switches)

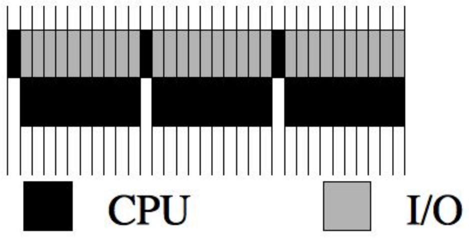
- Minimize response time (*latency*). Response time is the time to complete a job.
- Distribute cycles equitably. What does this mean?
- Meet all deadlines (real-time). We won't talk too much about this.

- **F**irst-**I**n-**F**irst-**O**ut (**FIFO**). Also called **F**irst-**C**ome-**F**irst-**S**erved (**FCFS**)
  - Run until blocked or completed. When a blocked process becomes ready put it at the end of the run queue.
  - "Run until completed" results in uniprogramming.
  - Problem: one process can monopolize CPU.
- Solution: limit the maximum amount of time that a process can run without a context switch. This time is called a *time slice*, or a *time quantum*.
  - Use the clock interrupt to enforce the time slice.
- **Round Robin**: run process for one time slice, then move to back of the queue. Each process gets equal share of the CPU. Most systems use some variant of this. What happens if the time slice isn't chosen carefully?
  - Too long: one process can monopolize the CPU, interactive behavior suffers.

- ○ Too small: too much time wasted in context switches.
- ○ For example, consider two processes, one doing 1 ms computation followed by 10 ms I/O, the other doing all computation. Suppose we use 20 ms time slice and round-robin scheduling: I/O process runs at 11/21 speed, I/O devices are only utilized 10/21 of time.



- ○ Suppose we use 1 ms time slice: then the compute-bound process gets interrupted 9 times unnecessarily before the I/O-bound process is runnable.



- ○ Want quanta of different sizes.

- ● Originally, Unix had 1-second time slices. Too long. Needs to be short enough so that interactive processes respond quickly to the user, but no shorter. 50-100 ms is reasonable.