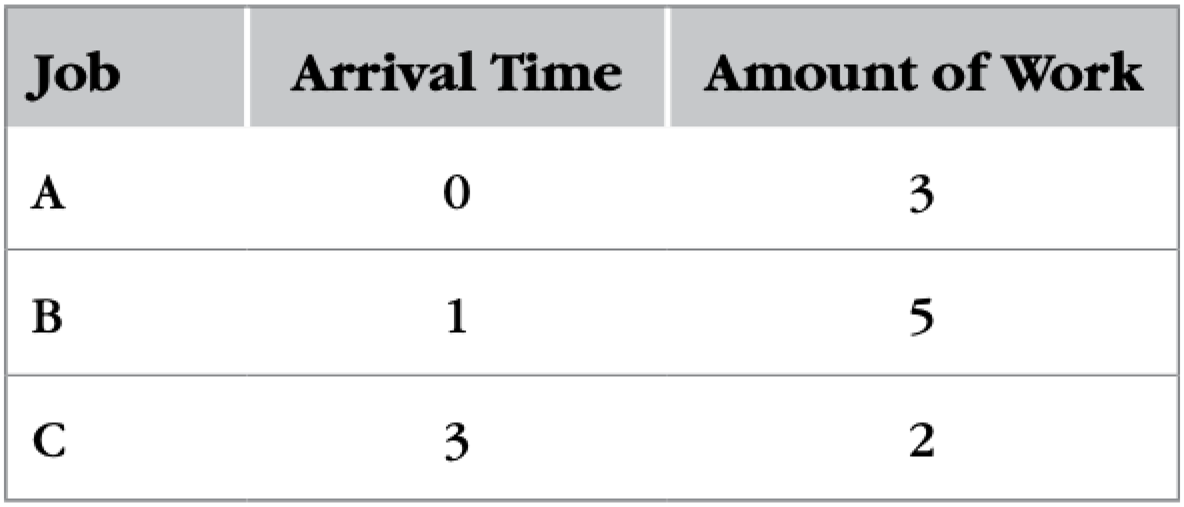
10. Scheduling Policies



Job	Arrival Time	Amount of Work
A	0	3
B	1	5
C	3	2

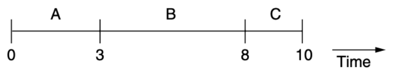
We use these processes to compare scheduling policies

We denote the cost of a context switch as “□”

<u>**FIRST-COME, FIRST-SERVED (FLFS)**</u>

- process:
  - hold a queue of threads waiting to run
  - run threads to completion
- good for batch application with pre-known tasks (payroll, auditing, etc.)

Jobs	Wait Time	Turnaround Time
A	0	5
B	4 + □	6 + □
C	5 + 2□	14 + 2□
D	13 + 3□	17 + 3□
AVERAGE	5.5 + 1.5□	10.5 + 1.5□



+ fair
+ utilization
- long wait time
- convoy effect

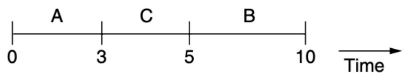
Total time of execution = 20 + 3□

* the cost of a context switch is cheap here, since no save state is necessary

SHORTEST JOB FIRST (SJF)

- process
  - hold a min-heap of waiting threads
  - run threads to completion
- this algorithm makes the assumption that runtimes are (roughly) known

Jobs	Wait Time	Turnaround Time
A	0	5
B	4 + □	6 + □
C	9 + 3□	18 + 3□
D	4 + 2□	8 + 3□
AVERAGE	4.25 + 1.5□	9.25 + 1.5□



+ improved average weight and turnaround time
+ high utilization
- unfair: starvation

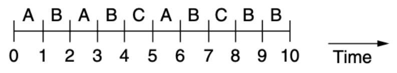
Total time of execution = 20 + 3□

* the cost of a context switch is cheap here, since no save state is necessary

ROUND-ROBIN (RR)

- FCFS, but each process is only run for a set time slice
- This algorithm necessitates preemption

Jobs	Wait Time	Turnaround Time
A	0	15 + 14□
B	□	5 + 5□
C	2□	18 + 15□
D	3□	11 + 13□
AVERAGE	4.25 + 1.5□	12.25 + 11.25□



+ fair, if new jobs are placed at the end of the queue;
  ~ this algorithm does not do that
+ shorter wait time
- lower utilization

Total time of execution = 20 + 15

* the total time to run all of the tasks has gone up

The above scheduling policies have been static, some scheduling policies are dynamic, and depend on the state of the environment of the machine

PRIORITY SCHEDULING

- Jobs are given priority, & the highest runs first
- Linux uses an inverted version with “niceness”, where nice tasks will defer to others
    - Students can increase niceness, but not lower it

```
nice.c = {
  read args
  set priority
  execvp("gcc", (char* []){"gcc", "foo.c");
}
```

Specifically, this is an example of dynamically assigned priority

There is an even more complicated version of this called a:
MULTILEVEL FEEDBACK QUEUE (MLFQ)

Goals:

i) optimize turnaround time
ii) responsive to interactive users
iii) optimize response time

Maintain many queues with distinct priority levels
Round Robin within a queue

Rules:

i) if P(A) > P(B), run A
ii) If P(A) = P(B), Round Robin
iii) new jobs enter at top queue
iv) jobs move down a queue after using time allotment
v) boost all jobs to the top queue after a set time interval

Parametrizing the time interval is tough; it is a **voodoo constant**;

Some use **decay-usage** algorithms
Some let users manipulate it with **hints**, called **advice**