

## 10. Scheduling Policies

We use the following processes to compare scheduling policies:

Jobs	Arrival Time	Work
A	0	3
B	1	5
C	3	2

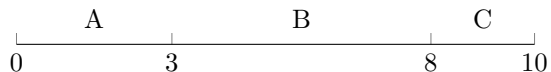
We denote the cost of a context switch as " $\delta$ ".

### FIRST-COME, FIRST-SERVED (FLFS)

The process:

1. hold a queue of threads waiting to run
2. run threads to completion

This is good for batch application with pre-known tasks (payroll, auditing, etc.)

Jobs	Wait Time	Turnaround Time	
A	0	5	
B	$4 + \delta$	$6 + \delta$	
C	$5 + 2\delta$	$14 + 2\delta$	
D	$13 + 3\delta$	$17 + 3\delta$	
AVG	$5.5 + 1.5\delta$	$10.5 + 1.5\delta$	

Properties:

- + fair
- + utilization
- long wait time
- convoy effect

Total time of execution =  $20 + 3\delta$

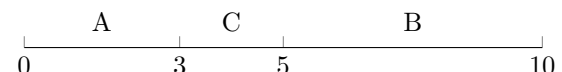
\* the cost of a context switch is cheap here, since no save state is necessary

### SHORTEST JOB FIRST (SJF)

The process:

1. hold a min-heap of waiting threads
2. run threads to completion

This algorithm makes the assumption that runtimes are (roughly) known.

Jobs	Wait Time	Turnaround Time	
A	0	5	
B	$4 + \delta$	$6 + \delta$	
C	$9 + 3\delta$	$18 + 3\delta$	
D	$4 + 2\delta$	$8 + 3\delta$	
AVG	$4.25 + 1.5\delta$	$9.25 + 1.5\delta$	

Properties:

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

Total time of execution =  $20 + 3\delta$

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

### ROUND-ROBIN (RR)

The process: FCFS, but each process is only run for a set time slice

This algorithm necessitates preemption.

Jobs	Wait Time	Turnaround Time
$A$	0	$15 + 14\delta$
$B$	$\delta$	$5 + 5\delta$
$C$	$2\delta$	$18 + 15\delta$
$D$	$3\delta$	$11 + 13\delta$
AVG	$4.25 + 1.5\delta$	$12.25 + 11.25\delta$

A

B

A

B

C

A

B

C

B

B

0

1

2

3

4

5

6

7

8

9

10

Properties

+ fair, if new jobs are placed at the end of the queue (pictured algorithm does not do that)

+ shorter wait time

- lower utilization

Total time of execution =  $20 + 15\delta$

\* 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

The process: Jobs are given priority, and 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:

1. optimize turnaround time
2. responsive to interactive users
3. optimize response time

Maintain many queues with distinct priority levels.

The process: Round Robin within a queue according to the 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 find this value using *decay-usage algorithms*. Others let users manipulate it with hints, called advice.