Computing platforms (Spring 2025)
week 2

Exercise 1

Multicore. Suppose a single application is running on a multicore system with 16 processors. If 20% of the code is inherently serial, what is the performance gain over a single processor system? What would it be if only 2% of the code would be inherently serial?

Using Amdahl's Law [1]:

$$S = \frac{1}{(1-f) + \frac{f}{N}}$$

where:

- $1-f$: Fraction of the code that is inherently serial.
- f: Fraction of the code that is parallelizable.
- $N = 16$: Number of processors.

a) **20% Serial Code** $(1-f = 0.2,\ f = 0.8)$

Substitute into the formula:

$$S = \frac{1}{(1-0.8) + \frac{0.8}{16}} = \frac{1}{0.2 + 0.05} = \frac{1}{0.25} = 4$$

Performance Gain: $S = 4$.

b) **2% Serial Code** $(1-f = 0.02,\ f = 0.98)$

Substitute into the formula:

$$S = \frac{1}{(1-0.98) + \frac{0.98}{16}} = \frac{1}{0.02 + 0.06125} = \frac{1}{0.08125} \approx 12.31$$

Performance Gain: $S \approx 12.31$.

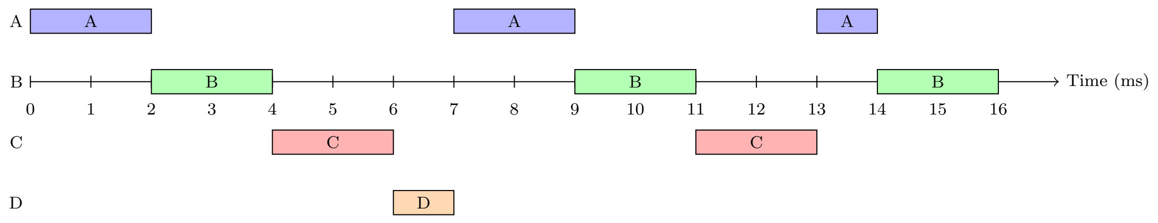
Exercise 2

We have four jobs that arrive at time 0

$$A = 5 \text{ ms}$$
$$B = 6 \text{ ms}$$
$$C = 4 \text{ ms}$$
$$D = 1 \text{ ms}$$

All are CPU-bound and we ignore context switch overhead.

(a) Round-Robin with 2 ms time slice

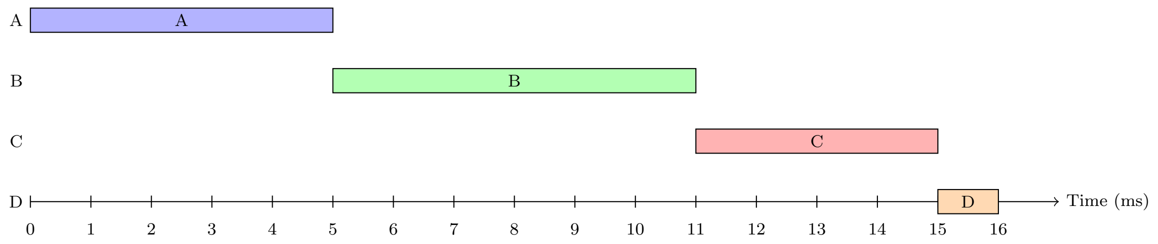


Turnaround times:

$$A \to 14$$
$$B \to 16$$
$$C \to 13$$
$$D \to 7$$

Average TAT $= \frac{14+16+13+7}{4} = 12.5\,\text{ms}$

(b) First-come-first-served



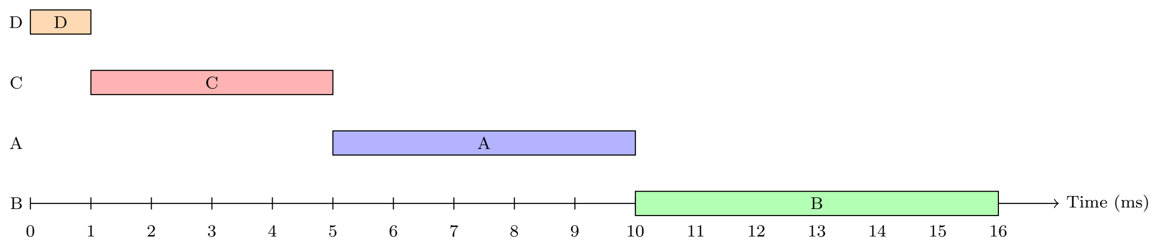
Turnaround times:

$$A \to 5$$
$$B \to 11$$
$$C \to 15$$
$$D \to 16$$

Average TAT $= \frac{5+11+15+16}{4} = 11.75\,\text{ms}$

(c) Shortest-job-first

Order by shortest runtime first: $D \to C \to A \to B$.



Turnaround times:

$$\begin{aligned} D &\to 1 \\ C &\to 5 \\ A &\to 10 \\ B &\to 16 \end{aligned}$$

Average TAT $= \frac{1+5+10+16}{4} = 8\,\text{ms}$

Exercise 3

1) Compute the solutions for simulations with 3 jobs and random seeds of 1, 2

1) seed 1

```
$ python3 ./lottery.py -j 3 -s 1
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 1

Here is the job list, with the run time of each job:
Job 0 ( length = 1, tickets = 84 )
Job 1 ( length = 7, tickets = 25 )
Job 2 ( length = 4, tickets = 44 )


Here is the set of random numbers you will need (at most):
Random 651593
Random 788724
Random 93859
Random 28347
Random 835765
Random 432767
Random 762280
Random 2106
Random 445387
Random 721540
Random 228762
Random 945271
```

2) seed 2

```
$ python3 ./lottery.py -j 3 -s 2
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 2

Here is the job list, with the run time of each job:
Job 0 ( length = 9, tickets = 94 )
Job 1 ( length = 8, tickets = 73 )
Job 2 ( length = 6, tickets = 30 )


Here is the set of random numbers you will need (at most):
Random 605944
Random 606802
Random 581204
Random 158383
Random 430670
Random 393532
Random 723012
Random 994820
Random 949396
Random 544177
Random 444854
Random 268241
Random 35924
Random 27444
Random 464894
Random 318465
Random 380015
Random 891790
Random 525753
Random 560510
Random 236123
Random 23858
Random 325143
```

2) Now run with two specific jobs: each of length 10, but one (job 0) with 1 ticket and the other (job 1) with 100 (e.g., -l 10:1,10:100). What happens when the number of tickets is so imbalanced? Will job 0 ever run before job 1 completes? How often? In general, what does such a ticket imbalance do to the behavior of lottery scheduling?

```
$ python3 ./lottery.py -s 1 -l 10:1,10:100
ARG jlist 10:1,10:100
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 1

Here is the job list, with the run time of each job:
Job 0 ( length = 10, tickets = 1 )
Job 1 ( length = 10, tickets = 100 )


Here is the set of random numbers you will need (at most):
Random 134364
Random 847434
Random 763775
Random 255069
Random 495435
Random 449491
Random 651593
Random 788724
Random 93859
Random 28347
Random 835765
Random 432767
Random 762280
Random 2106
Random 445387
Random 721540
Random 228762
Random 945271
Random 901428
Random 30590
```

There are total of 101 tickets, 1 for job 0 and 100 for job 1. Job 0 can run before job 1 finish but it is unlikely. Job 0 has 1/101 (≈ 0.99 %) probability to run before job 1 finish. Imbalance causes job 1 'dominate' cpu and job 0 will starve.

Exercise 4

Multiprocessor scheduling.

1) The first simulation will run just one job, which has a run-time of 30, and a working-set size of 200. Run this job (called job 'a' here) on one simulated CPU as follows: ./multi.py -n 1 -L a:30:200. How long will it take to complete?

```
python3 ./multi.py -n 1 -L a:30:200 -c -t
ARG seed 0
ARG job_num 3
ARG max_run 100
ARG max_wset 200
ARG job_list a:30:200
ARG affinity
ARG per_cpu_queues False
ARG num_cpus 1
ARG quantum 10
ARG peek_interval 30
ARG warmup_time 10
ARG cache_size 100
ARG random_order False
ARG trace True
ARG trace_time False
ARG trace_cache False
ARG trace_sched False
ARG compute True

Job name:a run_time:30 working_set_size:200

Scheduler central queue: ['a']

   0   a
   1   a
   2   a
   3   a
   4   a
   5   a
   6   a
   7   a
   8   a
   9   a
----------
  10   a
  11   a
  12   a
  13   a
  14   a
  15   a
  16   a
  17   a
  18   a
  19   a
----------
  20   a
  21   a
  22   a
  23   a
  24   a
  25   a
  26   a
  27   a
  28   a
  29   a

Finished time 30

Per-CPU stats
CPU 0  utilization 100.00 [ warm 0.00 ]
```

job will complete in 30 ticks.

2) Now increase the cache size so as to make the job's working set (size=200) fit into the cache (which, by default, is size=100); for example, run ./multi.py -n 1 -L a:30:200 -M 300. Can you predict how fast the job will run once it fits in cache?

```
$ python3 ./multi.py -n 1 -L a:30:200 -M 300 -c
ARG seed 0
ARG job_num 3
ARG max_run 100
ARG max_wset 200
ARG job_list a:30:200
ARG affinity
ARG per_cpu_queues False
ARG num_cpus 1
ARG quantum 10
ARG peek_interval 30
ARG warmup_time 10
ARG cache_size 300
ARG random_order False
ARG trace False
ARG trace_time False
ARG trace_cache False
ARG trace_sched False
ARG compute True

Job name:a run_time:30 working_set_size:200

Scheduler central queue: ['a']


Finished time 20

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 50.00 ]
```

Now job finish in 20 ticks.

3) Run the same simulation as above, but this time with time left tracing enabled (-T). This flag shows both the job that was scheduled on a CPU at each time step, as well as how much run-time that job has left after each tick has run. What do you notice about how that second column decreases?

```
$ python3 ./multi.py -n 1 -L a:30:200 -M 300 -c -T
ARG seed 0
ARG job_num 3
ARG max_run 100
ARG max_wset 200
ARG job_list a:30:200
ARG affinity
ARG per_cpu_queues False
ARG num_cpus 1
ARG quantum 10
ARG peek_interval 30
ARG warmup_time 10
ARG cache_size 300
ARG random_order False
ARG trace False
ARG trace_time True
ARG trace_cache False
ARG trace_sched False
ARG compute True

Job name:a run_time:30 working_set_size:200

Scheduler central queue: ['a']

   0   a [ 29]
   1   a [ 28]
   2   a [ 27]
   3   a [ 26]
   4   a [ 25]
   5   a [ 24]
   6   a [ 23]
   7   a [ 22]
   8   a [ 21]
   9   a [ 20]
----------------
  10   a [ 18]
  11   a [ 16]
  12   a [ 14]
  13   a [ 12]
  14   a [ 10]
  15   a [  8]
  16   a [  6]
  17   a [  4]
  18   a [  2]
  19   a [  0]

Finished time 20

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 50.00 ]
```

Once cache warmup is finished, decrease of job time left doubles.

4) Now add one more bit of tracing, to show the status of each CPU cache for each job, with the -C flag. For each job, each cache will either show a blank space (if the cache is cold for that job) or a 'w' (if the cache is warm for that job). At what point does the cache become warm for job 'a' in this simple example? What happens as you change the warmup time parameter (-w) to lower or higher values than the default?

```
$ python3 ./multi.py -n 1 -L a:30:200 -M 300 -c -T -C
ARG seed 0
ARG job_num 3
ARG max_run 100
ARG max_wset 200
ARG job_list a:30:200
ARG affinity
ARG per_cpu_queues False
ARG num_cpus 1
ARG quantum 10
ARG peek_interval 30
ARG warmup_time 10
ARG cache_size 300
ARG random_order False
ARG trace False
ARG trace_time True
ARG trace_cache True
ARG trace_sched False
ARG compute True

Job name:a run_time:30 working_set_size:200

Scheduler central queue: ['a']

   0   a [ 29] cache[ ]
   1   a [ 28] cache[ ]
   2   a [ 27] cache[ ]
   3   a [ 26] cache[ ]
   4   a [ 25] cache[ ]
   5   a [ 24] cache[ ]
   6   a [ 23] cache[ ]
   7   a [ 22] cache[ ]
   8   a [ 21] cache[ ]
   9   a [ 20] cache[w]
-------------------------
  10   a [ 18] cache[w]
  11   a [ 16] cache[w]
  12   a [ 14] cache[w]
  13   a [ 12] cache[w]
  14   a [ 10] cache[w]
  15   a [  8] cache[w]
  16   a [  6] cache[w]
  17   a [  4] cache[w]
  18   a [  2] cache[w]
  19   a [  0] cache[w]

Finished time 20

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 50.00 ]
```

When cache warmup has happened ticks remaining for job start to decrease faster, shorter warmup time will reduce total ticks faster making total time for job shorter.

Exercise 5

Multiprocessor scheduling (cont'd). We continue from previous exercise, that is, use the simulator multi.py as in previous exercise, and answer the homework questions 5-7 from OSTEP Chapter 10.

5) Let's run the following three jobs on a two-CPU system (i.e., type ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50) Can you predict how long this will take, given a round-robin centralized scheduler? Use -c to see if you were right, and then dive down into details with -t.

```
$ python3 ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -c
ARG seed 0
ARG job_num 3
ARG max_run 100
ARG max_wset 200
ARG job_list a:100:100,b:100:50,c:100:50
ARG affinity
ARG per_cpu_queues False
ARG num_cpus 2
ARG quantum 10
ARG peek_interval 30
ARG warmup_time 10
ARG cache_size 100
ARG random_order False
ARG trace False
ARG trace_time False
ARG trace_cache False
ARG trace_sched False
ARG compute True

Job name:a run_time:100 working_set_size:100
Job name:b run_time:100 working_set_size:50
Job name:c run_time:100 working_set_size:50

Scheduler central queue: ['a', 'b', 'c']


Finished time 150

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 0.00 ]
  CPU 1  utilization 100.00 [ warm 0.00 ]
```

Jobs finish with 150 ticks. Dummy round robin scheduling without considering job cpu history cause losing caches often.

6) Now we'll apply some explicit controls to study cache affinity, as described in the chapter. To do this, you'll need the -A flag. This flag can be used to limit which CPUs the scheduler can place a particular job upon. Can you predict how fast this version will run? Why does it do better? Will other combinations of 'a', 'b', and 'c' onto the two processors run faster or slower?

```
$ python3 ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -A a:0,b:1,c:1 -c
ARG seed 0
ARG job_num 3
ARG max_run 100
ARG max_wset 200
ARG job_list a:100:100,b:100:50,c:100:50
ARG affinity a:0,b:1,c:1
ARG per_cpu_queues False
ARG num_cpus 2
ARG quantum 10
ARG peek_interval 30
ARG warmup_time 10
ARG cache_size 100
ARG random_order False
ARG trace False
ARG trace_time False
ARG trace_cache False
ARG trace_sched False
ARG compute True

Job name:a run_time:100 working_set_size:100
Job name:b run_time:100 working_set_size:50
Job name:c run_time:100 working_set_size:50

Scheduler central queue: ['a', 'b', 'c']


Finished time 110

Per-CPU stats
  CPU 0  utilization 50.00 [ warm 40.91 ]
  CPU 1  utilization 100.00 [ warm 81.82 ]
```

Jobs finish now at 110 (instead of 150) because caches are not lost.

7) One interesting aspect of caching multiprocessors is the opportunity for better-than-expected speed up of jobs when using multiple CPUs (and their caches) as compared to running jobs on a single processor. Specifically, when you run on N CPUs, sometimes you can speed up by more than a factor of N, a situation entitled super-linear speedup. What do you notice about performance as the number of CPUs scales? Use -c to confirm your guesses, and other tracing flags to dive even deeper.

```
$ python3 ./multi.py -n 3 -L a:100:100,b:100:100,c:100:100 -M 100  -c
ARG seed 0
ARG job_num 3
ARG max_run 100
ARG max_wset 200
ARG job_list a:100:100,b:100:100,c:100:100
ARG affinity
ARG per_cpu_queues False
ARG num_cpus 3
ARG quantum 10
ARG peek_interval 30
ARG warmup_time 10
ARG cache_size 100
ARG random_order False
ARG trace False
ARG trace_time False
ARG trace_cache False
ARG trace_sched False
ARG compute True

Job name:a run_time:100 working_set_size:100
Job name:b run_time:100 working_set_size:100
Job name:c run_time:100 working_set_size:100

Scheduler central queue: ['a', 'b', 'c']


Finished time 55

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 81.82 ]
  CPU 1  utilization 100.00 [ warm 81.82 ]
  CPU 2  utilization 100.00 [ warm 81.82 ]
```

When CPU # equal to job # and cache size is big enough to fit working set caches are never lost hence caches after warmup operate optimally.

References

[1] William Stallings. *Operating Systems: Internals and Design Principles.* Pearson, 9th edition, 2018.