1. **To start things off, let's learn how to use the simulator to study how to build an effective multi-processor scheduler. 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? Turn on the -c flag to see a final answer, and the -t flag to see a tick-by-tick trace of the job and how it is scheduled.**

Predicted time: 30

```
Job name:a run_time:30 working_set_size:200


Scheduler central queue: ['a']



Finished time 30


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

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? (hint: remember the key parameter of the warm rate, which is set by the -r flag) Check your answer by running with the solve flag (-c) enabled.**

Predicted time: 20 (10 for warming and 20/2 = 10 of the rest time)

```
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 ]
```

3. **One cool thing about multi.py is that you can see more detail about what is going on with different tracing flags. 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?**

```
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]
```

Після пройденого «часу нагрівання» процесор став обробляти процес удвічі швидше.

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?**

```
Job name:a run_time:30 working_set_size:200


Scheduler central queue: ['a']


   0   a cache[ ]
   1   a cache[ ]
   2   a cache[ ]
   3   a cache[ ]
   4   a cache[ ]
   5   a cache[ ]
   6   a cache[ ]
   7   a cache[ ]
   8   a cache[ ]
   9   a cache[w]
--------------------
  10   a cache[w]
  11   a cache[w]
  12   a cache[w]
  13   a cache[w]
  14   a cache[w]
  15   a cache[w]
  16   a cache[w]
  17   a cache[w]
  18   a cache[w]
  19   a cache[w]
```

Кеш став теплим для процесу ‘a’ на десятому такті годинника. Якщо зменшити час нагріву – то процес виконається швидше, і виконуватиметься довше в протилежному випадку.

5. **At this point, you should have a good idea of how the simulator works for a single job running on a single CPU. But hey, isn't this a multi-processor CPU scheduling chapter? Oh yeah! So let's start working with multiple jobs. Specifically, 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 to see a step-by-step and then -C to see whether caches got warmed effectively for these jobs. What do you notice?**

Predicted time: 55 for a, 85 for b(wrong, because I take wrong scheduling strategy)

```
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 ]
```

Розігрів кешу не здійснив потрібного ефекту на виконання процесів. Це через те, що час розігріву та квантовий час співпадають, через що кожен раз як якийсь процес був розігрітий, то на даний процесор потрапляв інший процес. Так як ми маємо процес 'a', який потребує все місце кешу процесорів, то у висновку він стирав дані про інші два процеси, аналогічно вони стирали дані про перший процес, що нівелювало цей механізм.

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. In this case, let's use it to place jobs 'b' and 'c' on CPU 1, while restricting 'a' to CPU 0. This magic is accomplished by typing this ./multi.py -n 2 -L a:100:100,b:100:50, c:100:50 -A a:0,b:1,c:1 ; don't forget to turn on various tracing options to see what is really happening! 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?**

Predicted time: 110 (55 for a, 105 for b, 110 for c) (my previous strategy was similar with this, but I put 1 process in CPU1 after 'a' was done)

```
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 ]
```

Цей випадок виконався краще, адже тепер використовується розігрітий кеш, відповідно час першого процесу: 10 на розігрів + 90/2 = 45 = 55, для другого та третього 105 та 110 відповідно. Інші комбінації працюватимуть повільніше, адже перший процес потребує весь кеш, в той час як інші два лише половину, тому в інших комбінаціях на одному з процесорів відбуватиметься аналогічна ситуація до попередньої.

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. To experiment with this, use the job description here (-L a:100:100,b:100:100,c:100:100) with a small cache (-M 50) to create three jobs. Run this on systems with 1, 2, and 3 CPUs (-n 1, -n 2, -n 3). Now, do the same, but with a larger per-CPU cache of size 100. 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.**

```
ARG cache_size 50

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 300

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

```
ARG cache_size 50

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 150
```

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

```
ARG cache_size 50

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 100

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 0.00 ]
  CPU 1  utilization 100.00 [ warm 0.00 ]
  CPU 2  utilization 100.00 [ warm 0.00 ]
```

```
ARG cache_size 100

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 300

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

```
ARG cache_size 100

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 150

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

```
ARG cache_size 100

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 ]
```

Зі збільшенням кількості процесорів прямо пропорційно зменшується час виконання процесів, адже вони рівномірно розподіляються між процесами. Збільшення кешу до розміру, який потребують процеси, вплинув лише на випадок з 3 процесорами, адже для меншої кількості існує аналогічна проблема, що була в попередніх прикладах, а в цьому випадку за відсутності

черги кожен процес просто виконується у конкретному процесорів, через що має вплив розігріву кешу й виконується швидше.

8. **One other aspect of the simulator worth studying is the per-CPU scheduling option, the -p flag. Run with two CPUs again, and this three job configuration (-L a:100:100,b:100:50,c:100:50). How does this option do, as opposed to the hand-controlled affinity limits you put in place above? How does performance change as you alter the 'peek interval' (-P) to lower or higher values? How does this per-CPU approach work as the number of CPUs scales?**

Параметр –p створює черги для кожного процесора та розподіляє процеси по ним (як?). Відрізняється від ручних налаштувань тим, що процеси можуть переміщатися між процесорами в ході роботи. Якщо поставити піковий інтервал менше квантового часу, то процес 'c' перекине на другий процесор. Так як він потребує 50 кешу як і 'b', то вони обидва виконуються на 2 процесорі надалі (якщо я правильно зрозумів механізм роботи пікового інтервалу, зі спостережень випливає таке).