Worked with: Tong, Zhen Zhang, Jiahao Yan

Questions:

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.**

```
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 False
ARG trace_time False
ARG trace_cache False
ARG trace_sched False
ARG compute False

Job name:a run_time:30 working_set_size:200

Scheduler central queue: ['a']
```

Answer:

There is only one CPU, and its cache size is 100, but our job cache is 200, which is larger than the CPU cache. For this reason, the CPU cache never has the chance to 'warm' up and use the warm-up rate to speed up. And this is the only available job in the queue; thus, it will take 30 units of time to complete.

2. **Now increase the cache size 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.**

```
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 False

Job name:a run_time:30 working_set_size:200

Scheduler central queue: ['a']
```

Answer:
With the -M 300 flag, we increase the CPU cache size to 300, and we keep other parameters unchanged. Now the CPU cache has the chance into the 'warm' mode. But we also need to pay attention to warmup_time is 10. In other words, for 30 runtime jobs, we use the first 10 units of time to warm up, rest 20 runtimes of the job 'a' was in cache 'warm' mode. The default warm rate is 2 times faster. The total runtime is 10+ 20 / 2 = 10(cold mode) + 10(warm mode) = 20 units of time. The job will use 2 times faster in warm mode, and the CPU will process two runtimes of the job 'a' in one unit of time.

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 scheduled on a CPU at each time step and how much run-time that job has left after each tick has run. What do you notice about how that second column decreases?

Answer:
From time 0 to time 9, the job is in code mode, and the CPU use 1 unit of time to process 1 runtime of the job 'a'. From time 10 to time 19, the CPU use 1 unit of time to process 2 runtimes of the job 'a'.

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?

Answer:
At time 9 the cache is warm for job 'a'. When we change the warmup time with the -w flag earlier for example, -w 8, then the cache will enter warm mode earlier for job 'a', and then the process 'a' will also complete earlier, in this case at time 18. If we increase the warmup time to -w 12, then the cache will enter warm mode later, and the total runtime will also increase. Thus, the job 'a' will be complete at time 21.

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?

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 False

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']
```

Notice this time, we have two CPU, but our time slice is ten, and the cache warm-up time is also ten. In other words, once the cache starts to enter the warm mode, the round-robin police will force the CPUs to switch jobs. Let's focus on CPU 0, at time 0 the job 'a' is running on the CPU 0. Then at time 9, the cache just warm up for job 'a', but the round robin activate, and force cpu 0 switch to job 'c' . Since the maximum cache size for cpu 0 is 100, and work size for job 'a' is 100 too. Then, CPU 0 must move all the cache data back to memory, and use up 50 units cache size store job 'c'. At time 19, the cache warm up for job 'c', and round robin activate. Then the CPU switch to job 'b'. The CPU 1's cache use the last 50 units of cache for job 'b'. At time 29, the CPU 1's cache warm up for both job 'b' and 'c', but round robin switch job to 'a' again. Then it move all it's cache information back to the memory. This is one cycle. Same story for the CPU 1. This results in none of the jobs will enter the warm mode. Since the total run time is 100 + 100 + 100 = 300, and we have two cores, each core executes one instruction in one unit of time. Thus, all three jobs will be finished at 300 / 2(cores) = 150. In the -c flag, we notice each core executes one instruction each time, and they never enter warm mode, which the CPU core will execute two instructions each time.

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?

Answer:
Job 'a' can only run-on CPU 0, and jobs 'b' and 'c' and only run on CPU 1. Notice all cache size is big enough to support all three jobs. Thus, each job may have a chance to enter warm mode. That said, let's first analyze CPU 0: Job a will arrive at CPU 0 in time 0. For the first 10 units of time is cache is in cold mode. At time 9, the round-robin police start effect, but for CPU 1, we set its affinity to 'a' only so that it won't switch to other jobs; then, at time 11, the cache enters the warm mode. At time 54, the job 'a' is complete, and CPU 0 is in idle status. Next, let's analyze CPU 1. At time 0, the job 'b' and 'c' enter the queue. Notice both jobs' working size is 50, and CPU 1's cache can hold 100. Thus, the CPU 1 keep both job 'a' and 'c' in warm mode. Then CPU 1 will start to execute job 'b'. At time 9, the cache warms up, the round robin police activate. Now it is job 'c' time, and it use last 50 units of cache store information for job 'c'. At time 19 both jobs are warm up. Which means the CPU 1 keep switch job 'b' and 'c' but they are running in warm mode entirely. Their rest of run time should be (90+90) / 2 = 90, and their total runtime is 90 + 10 + 10 = 110. Thus, at time 109 both job 'b' and 'c' is complete. And we know job 'a' was complete by the CPU 1 at time 54. Finally, the all the jobs complete at time 109. Notice is 109 < 150. The reason is doing better is because both CPU's cache on running on the warm mode most of the time compared to previously one, that they never have a chance enter warm mode.
I believe this is the fastest combination, because has 100 work size and maxim size of cache for both CPU are 100 as well. If we move the job 'a' during the executions, the job 'a' will overwrite other cpu's cache, and other two job's will overwrite the cpu 0 's cache.

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.

Notice each -M 50 flag we limited each CPU's cache to 50, and each job's work size is 100. In other words, none of the jobs will run int warm mode. When we use -n 1 flag (1 CPU), the runtime is 300. Once we increase the number of CPU, the total runtime also decreases because we can distribute the jobs to different CPU, the total run decrease to 150 with -n 2 flag, and 100 with -n 3, which make sense we divide total runtime in half, and one thirds. When we increase CPUs' cache to 100, the first two -n 1, and -n 2 won't change because, the round robin cause all the job’s never enter warm mode. But with -n 3 (three CPU), the total run decrease to 55. Because each CPU only working on when job, the each job after first 10 units of time, it will enter the warm mode, run in double execution speed.

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?

With -p flag, the job ‘a’ and ‘c’ will enter cpu 0 ‘s queue, and job ‘b’ will enter cpu 1’s queue. The round robin won’t switch job into different queue. And only once cpu is in idle status will ‘steal’ job from other CPU’s queue. At time 54, the CPU 1 will finish the job b, and in time 60, the CPU 1 ‘steal’ job a from the CPU 0’s queue. And then they both run in parallel. At time 69 the job ‘a’ will also enter warm mode until they finished at time 99. CPU 0 will be complete at time 94, and the CPU 1 will be complete at time 99.

With -P 0, we turn off the peak mode, then in time 54, the CPU 1 will never steal job a from the CPU 0’s queue. In that case all jobs complete at time 199, which longer with enable the ‘steal’ job.

With -p 5, we peek other CPU’s queue every 5 units of time when the current CPU is idle, in this way the all job complete time reduce to time 90.

With -p 10, the peek time set to 10, which means for an idle CPU it will wait 10 units of time then start ‘steal’ job from other CPU’s queue. Notice for this time all jobs complete at time 99. This time is higher because we wait too long to start to ‘steal’ other queue’s job.

For this simulation the ‘steal’ operation have to wait one time-slice which is 10 units of time.

9. Finally, feel free to just generate random workloads and see if you can predict their performance on different numbers of processors, cache sizes, and scheduling options. If you do this, you’ll soon be a multi-processor scheduling master, which is a pretty awesome thing to be. Good luck!

Command: ./multi.py -s 1
ARG seed 1
ARG job_num 3

ARG max_run 100
ARG max_wset 200
ARG job_list
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 False

Job name:0 run_time:10 working_set_size:160
Job name:1 run_time:70 working_set_size:50
Job name:2 run_time:40 working_set_size:80

Scheduler central queue: ['0', '1', '2']

Answer:
All jobs enter one queue with two CPU. CPU 0 run job ‘0’ at time 0, at time 9 job ‘0’ done, at time 2 cpu 0 workingon job ‘2’, at time 19 the cache is warm, and it will be complete at time 34. For CPU 1, at time 0 it work on job 1, at time 9, job 1 warm up, and at time 39, all job are done.

ARG seed 3
ARG job_num 2
ARG max_run 100
ARG max_wset 200
ARG job_list
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 False

```
ARG trace_time False
ARG trace_cache False
ARG trace_sched False
ARG compute False

Job name:0 run_time:20 working_set_size:100
Job name:1 run_time:30 working_set_size:120

Scheduler central queue: ['0', '1']
```

Notice this time we have one cpu and it will keep switch between two jobs, not until first job finish, it won’t have chance enter warm mode. At time 29 the job 0 is finished, only job 1 left but job one has work size of 120 which large than the CPU’s cache, and then cache won’t run in warm mode. Thus, at time 20 + 30 = 50 all jobs complete .