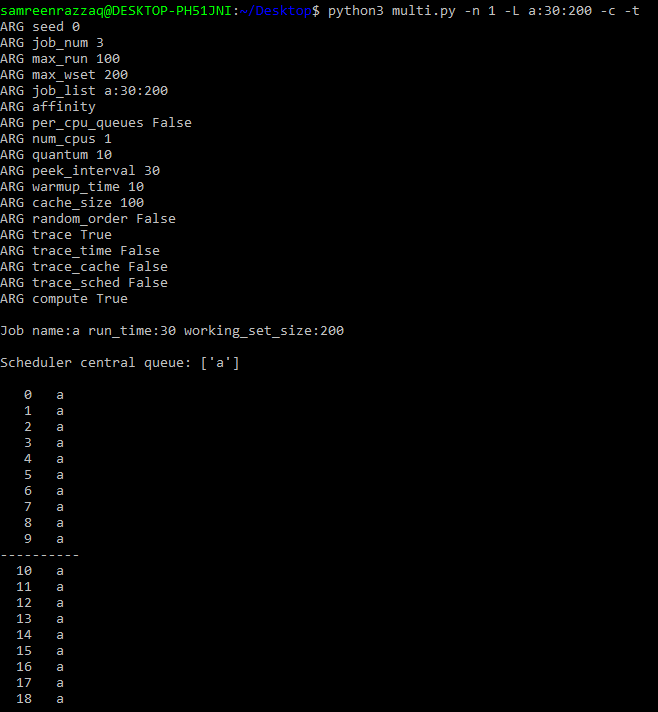
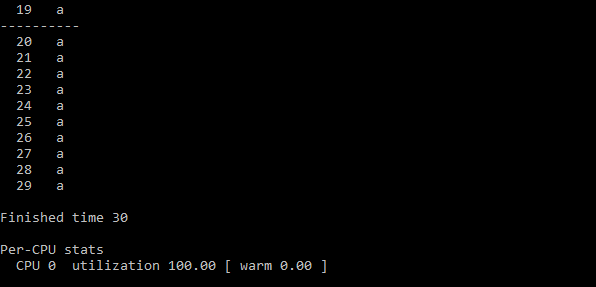


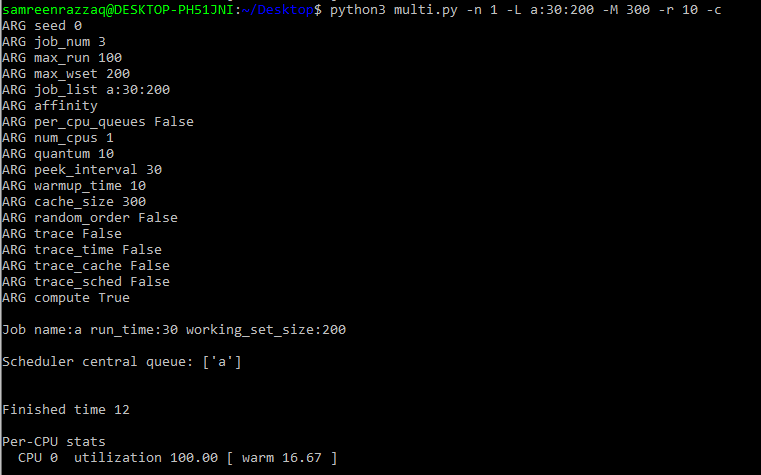
**Lab Tasks:**

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.

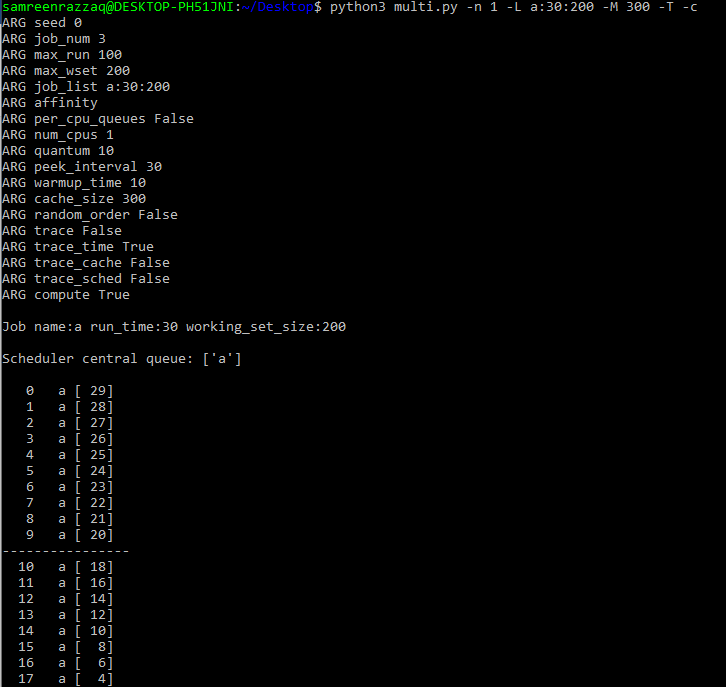


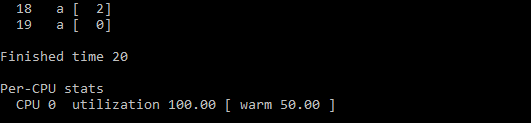
Finished time =30

nnn 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.

 By increase cache size, we see that job will run faster. You can see:

Finished time = 12

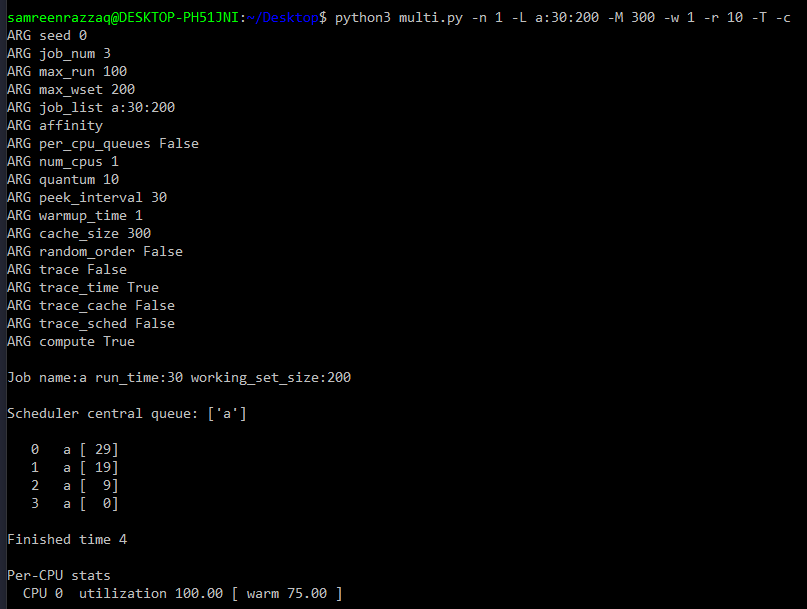
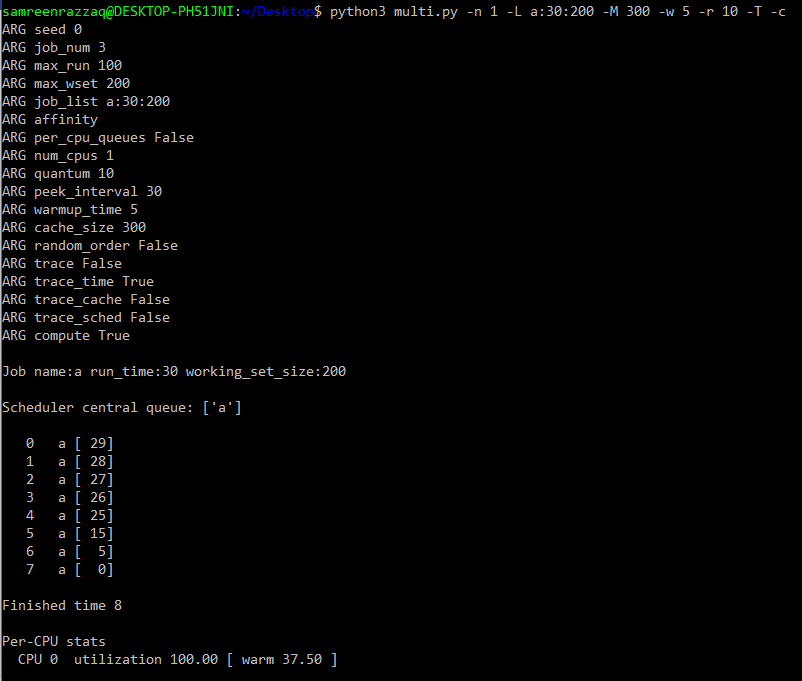
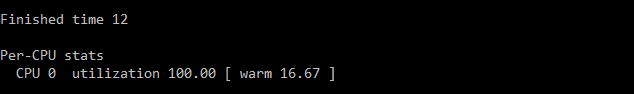
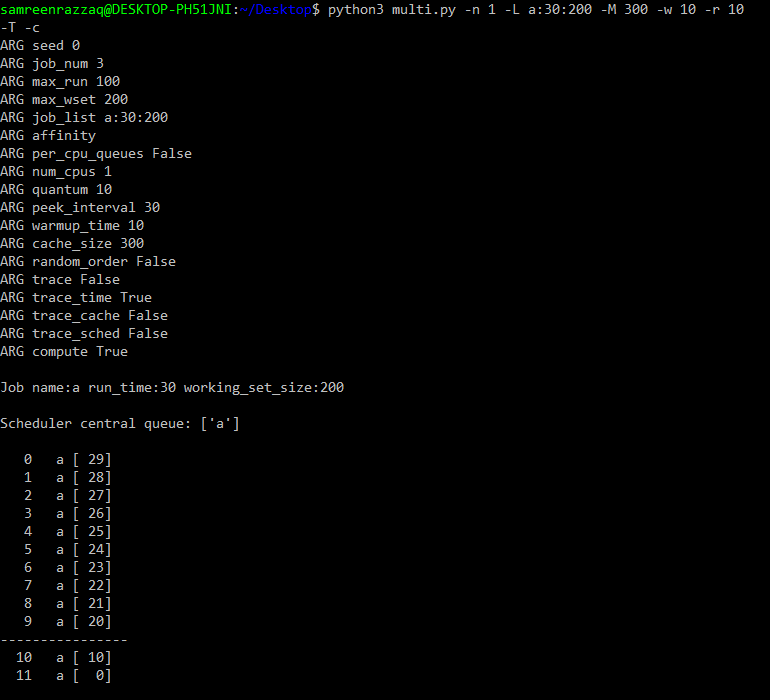
1. One cool thing about multi.py is that you can see more detail about what is going on with different tracing flags. Run the same simulation as above, but this time with time left tracing enabled (-T). This flag shows both the job that was scheduled on a CPU at each time step, as well as how much run-time that job has left after each tick has run. What do you notice about how that second column decreases?



As we know that -T is the left tracing enabler. When we use this, the tracing value decrease downward one by one value, but after running 10 instructions it decrease by 2 steps because warm rate is set 2 by default. You can change this by using -r flag.

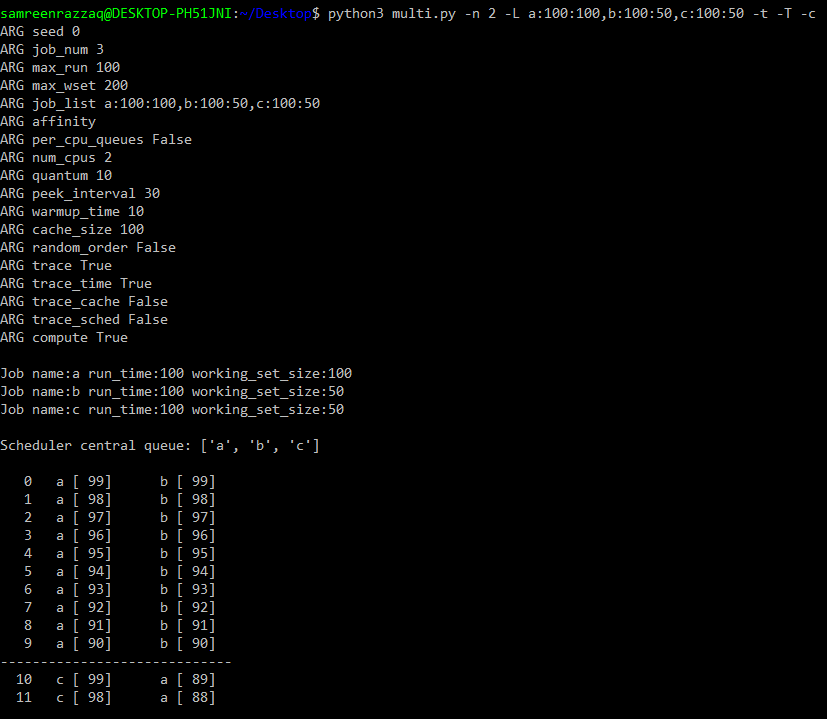
1. Now add one more bit of tracing, to show the status of each CPU cache for each job, with the -c flag. For each job, each cache will either show a blank space (if the cache is cold for that job) or a ’w’ (if the cache is warm for that job). At what point does the cache become warm for job ’a’ in this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

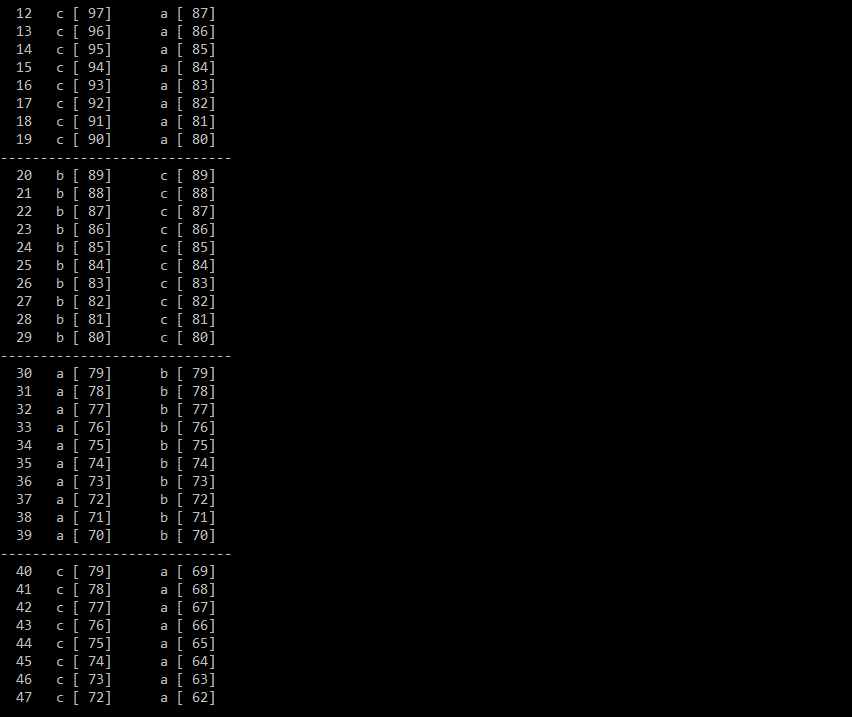
Now we change -w value (warm-up time) from lower to higher to see the results:

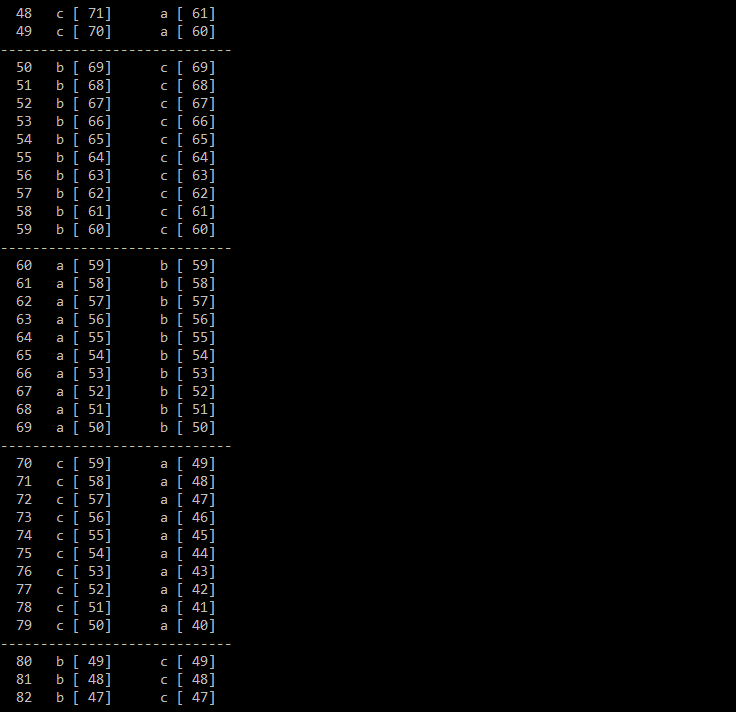
* Warm-up time = 1:
* Warm-up time = 5:
* Warm-up time = 10:

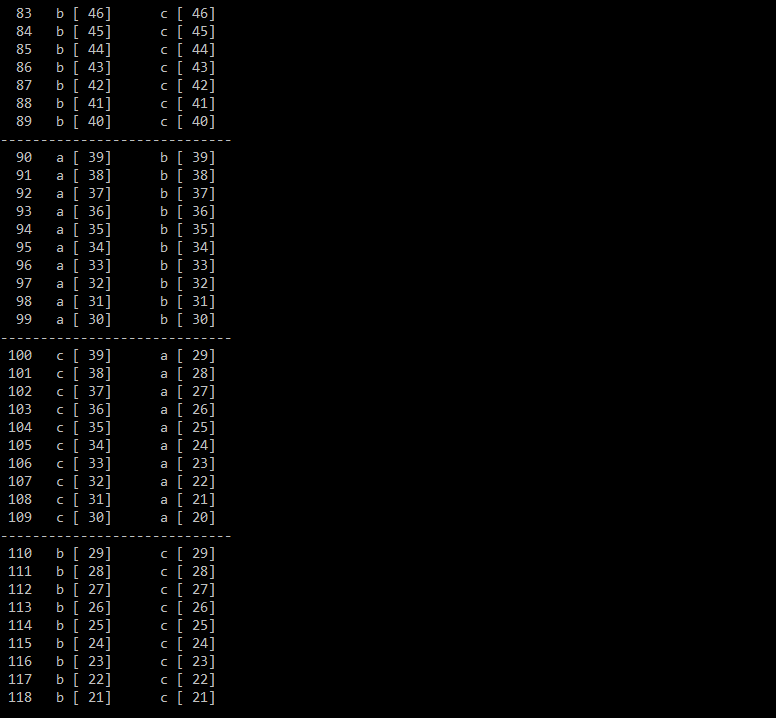
By increasing the warm-up time(-w), we see that cache’s warm value become decrease and decrease and job running time is increased.

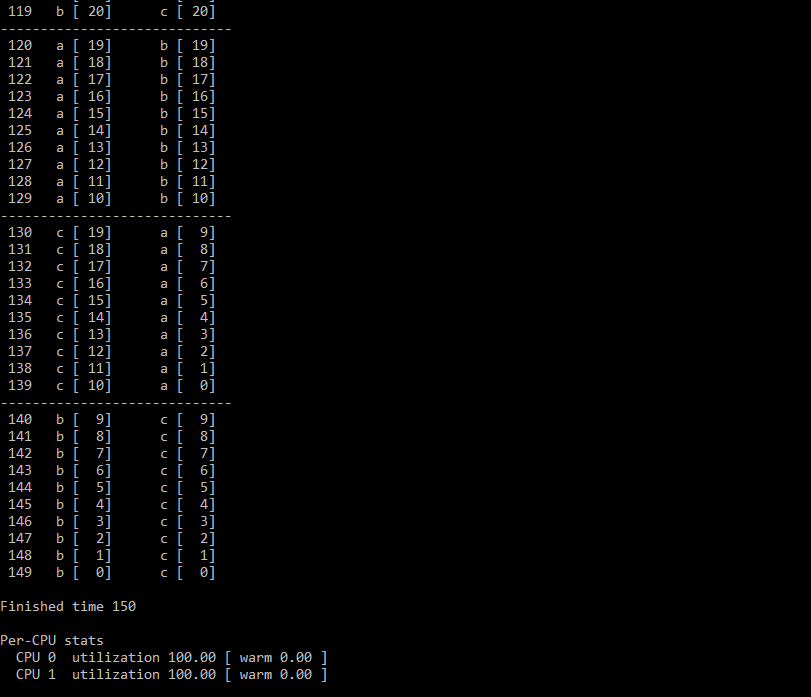
5. At this point, you should have a good idea of how the simulator works for a single job running on a single CPU. But hey, isn’t this a multi-processor CPU scheduling lab? Oh yeah! So let’s start working with multiple jobs. Specifically, let’s run the following three jobs on a two-CPU system (i.e., 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?



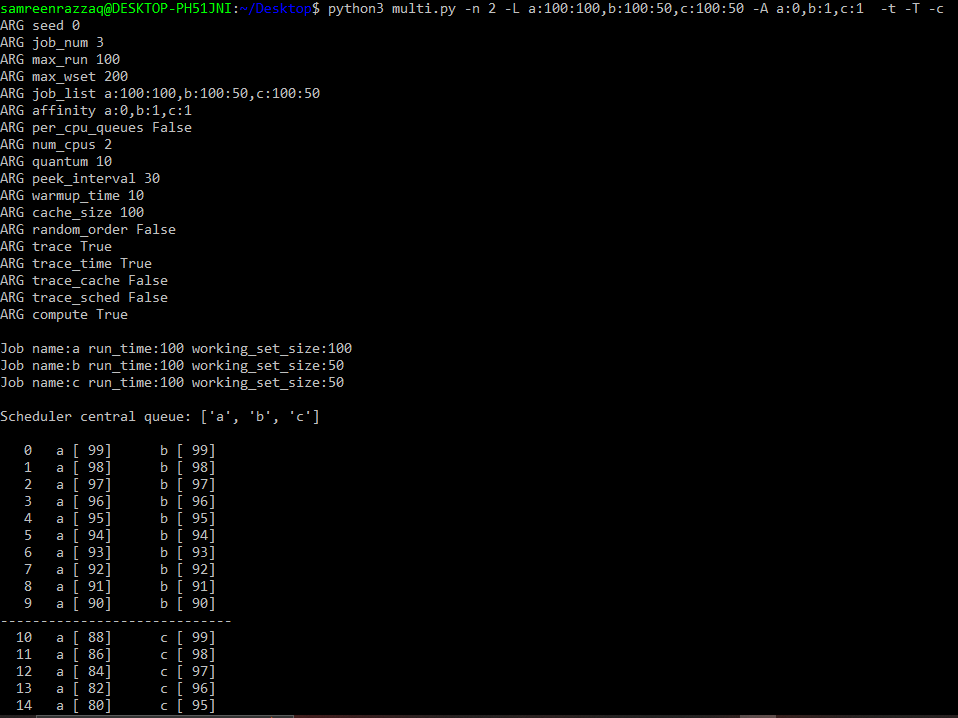
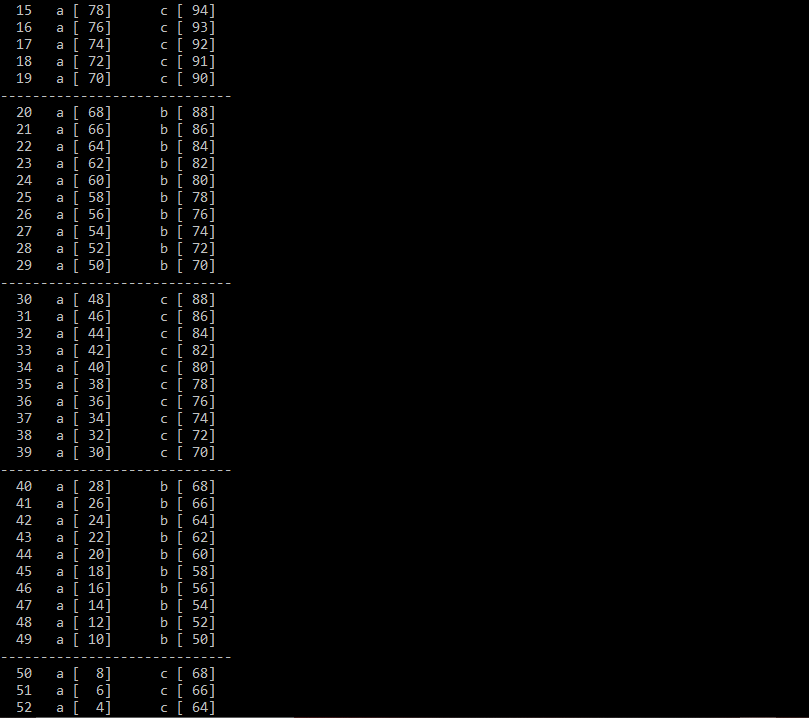


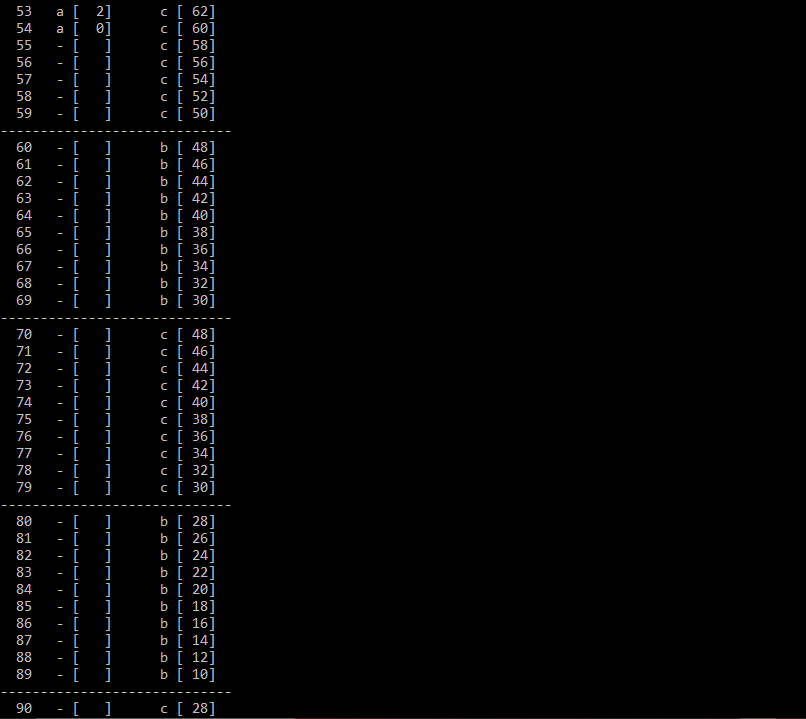


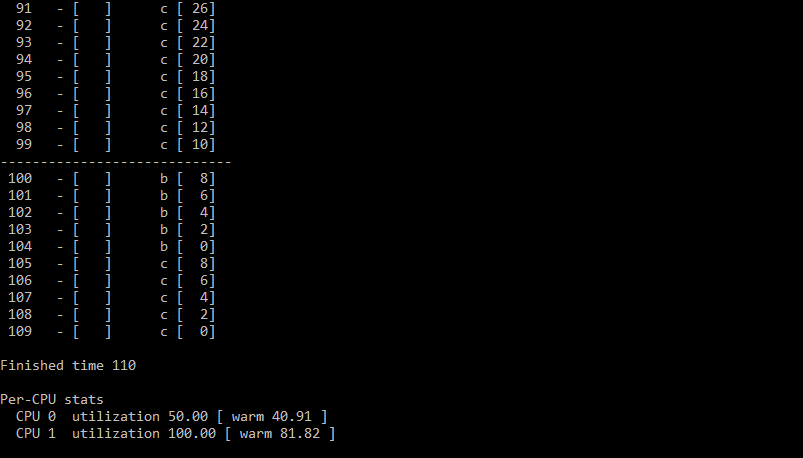


By increasing no.of jobs into three in 2 cpus then the jobs running time is 150 and cache is not warm in both cpus.

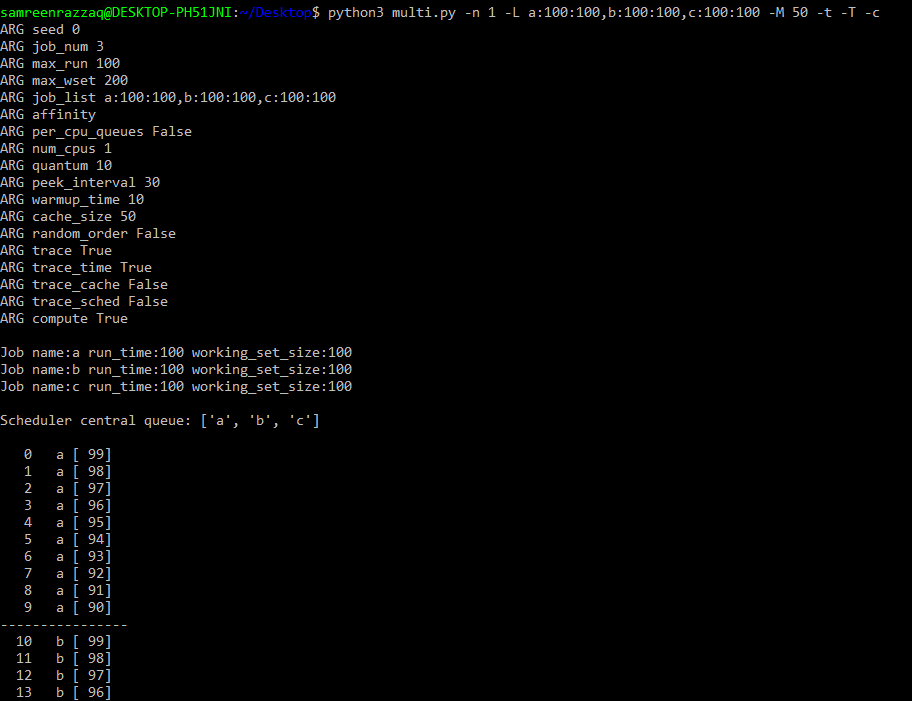
6. Now we’ll apply some explicit controls to study cache affinity. To do this, you’ll need the -A flag. This flag can be used to limit which CPUs the scheduler can place a particular job upon. In this case, let’s use it to place jobs ’b’ and ’c’ on CPU 1, while restricting ’a’ to CPU 0. This magic is 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?



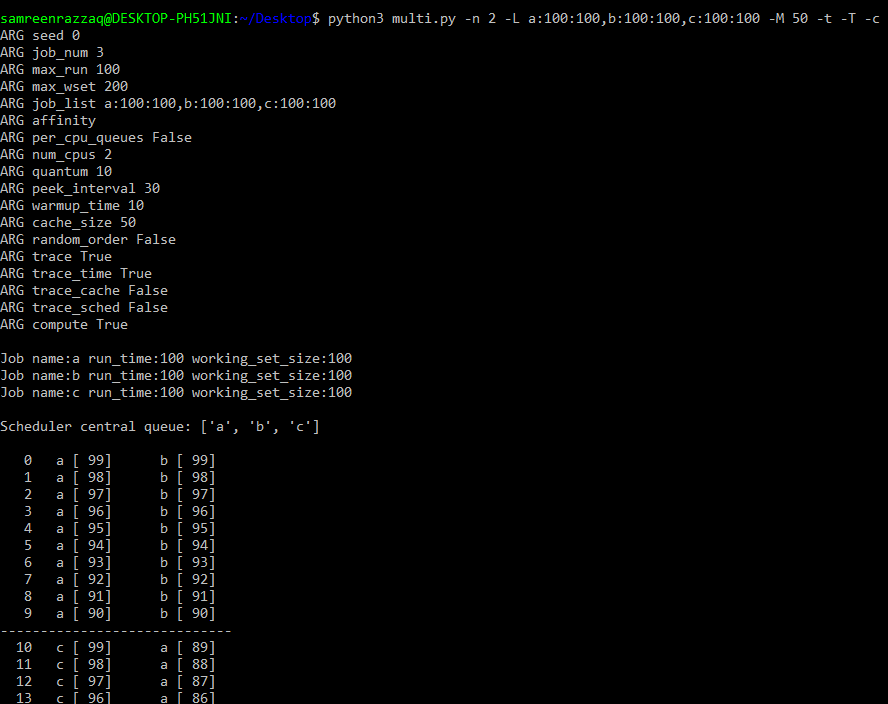
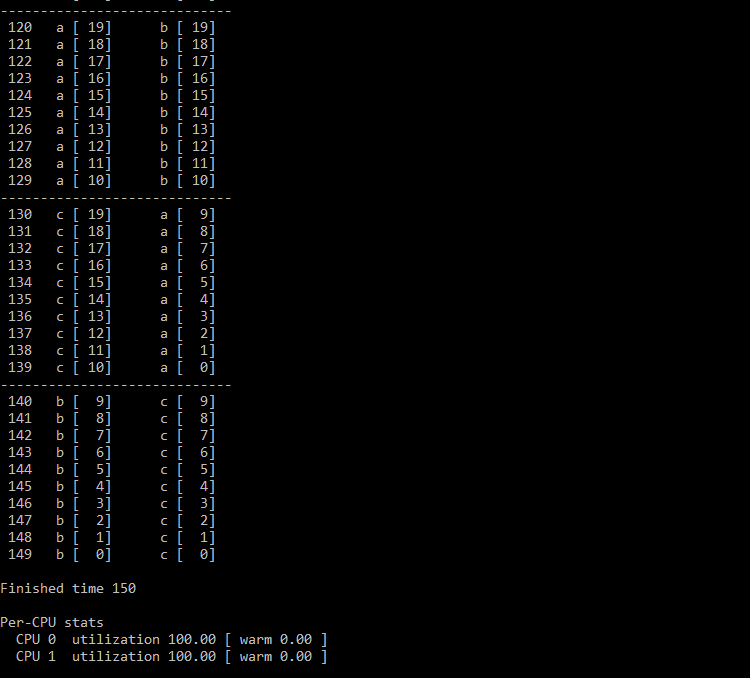


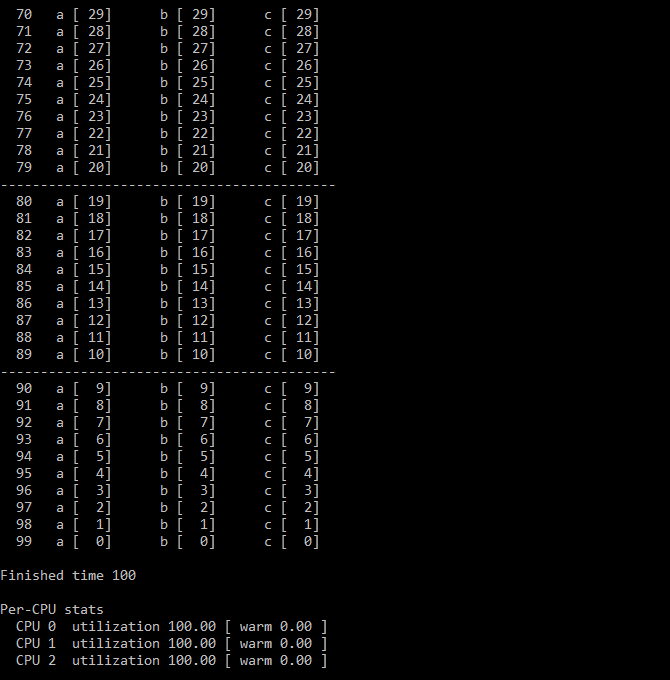
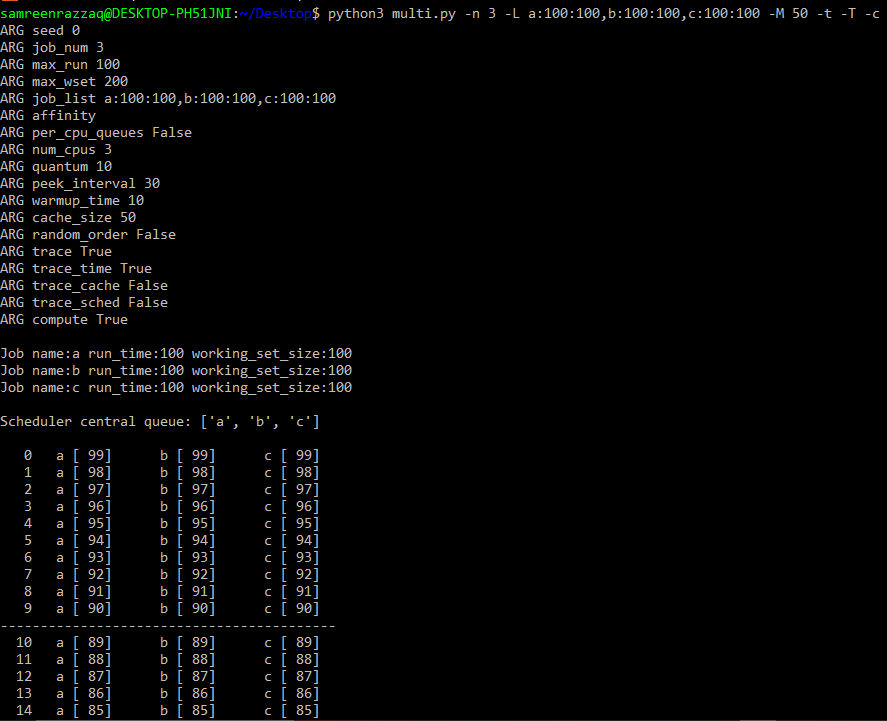
In this question, we assign ‘a’ job to cpu 0 and ‘b,c’ jobs are assigned to cpu 1. We assume that work load in cpu 0 is less so cache’s warm time is also less and in cpu 1, work load is more so that cache’s warm-up time is more in this. But as you know we use 2 cpu’s here as above but there job run fastly as compare to above.

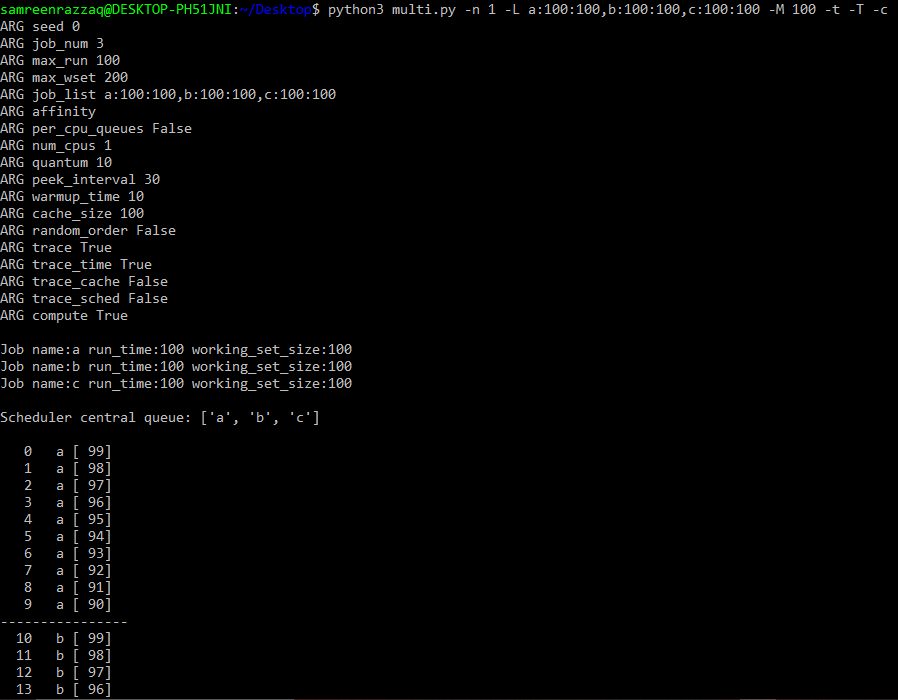
7. One interesting aspect of caching multiprocessors is the opportunity for better-than-expected speed up of jobs when using multiple CPUs (and their caches) as compared to running jobs on a single processor. Specifically, when you run on N CPUs, sometimes you can speed up by more than a factor of N, 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.

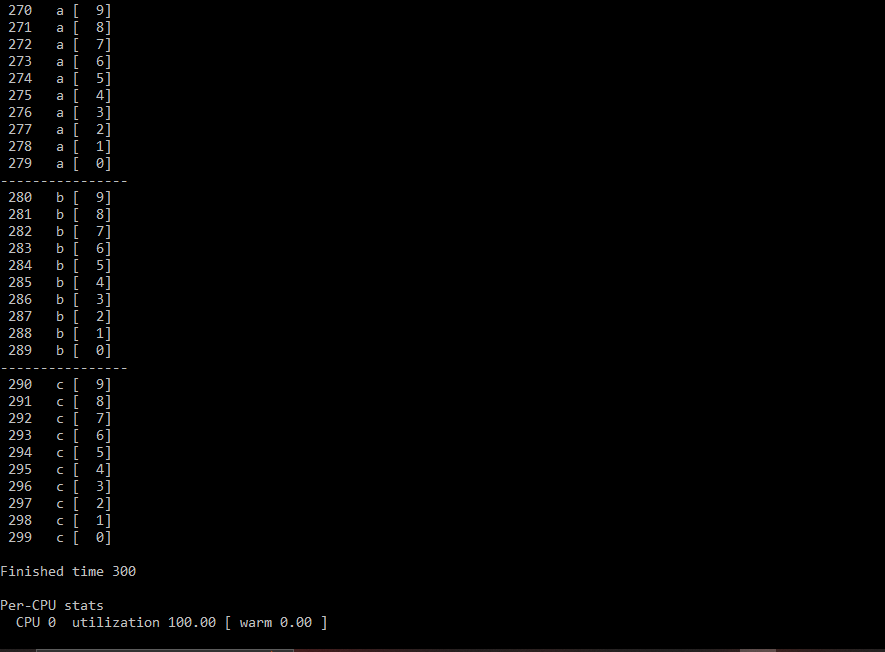
* Firstly we use Cache size = 50 (with 1,2,3 cpus):

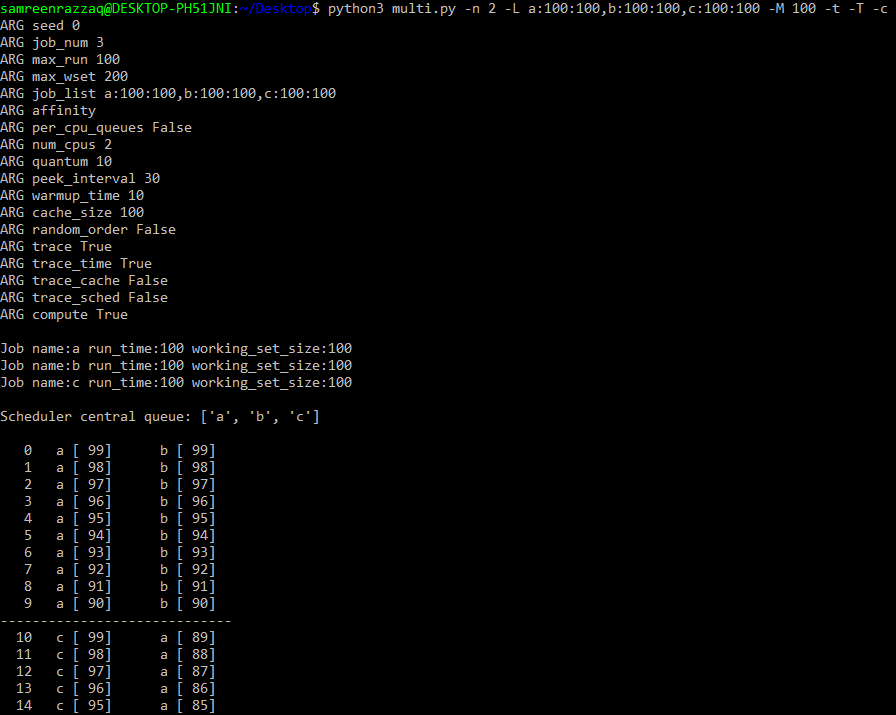


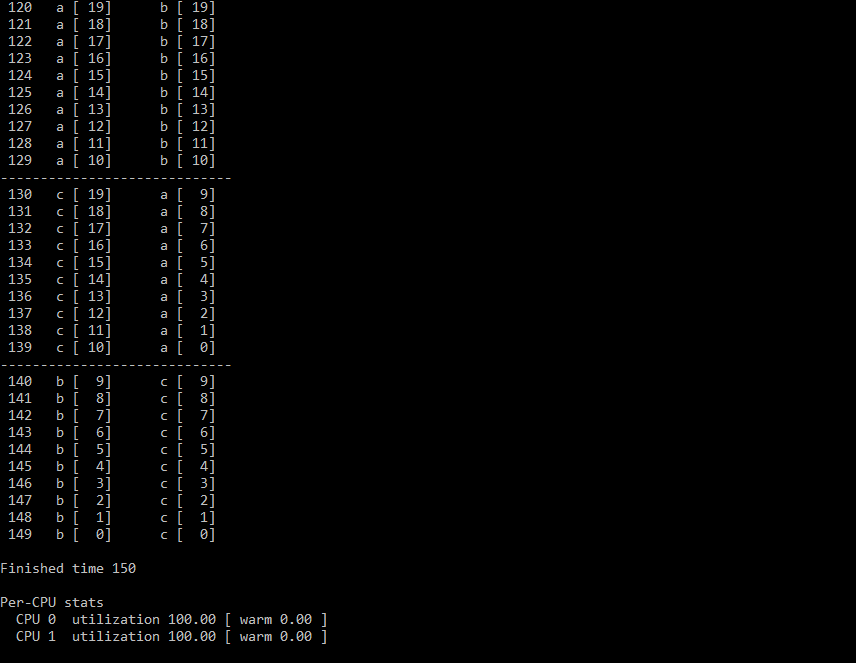


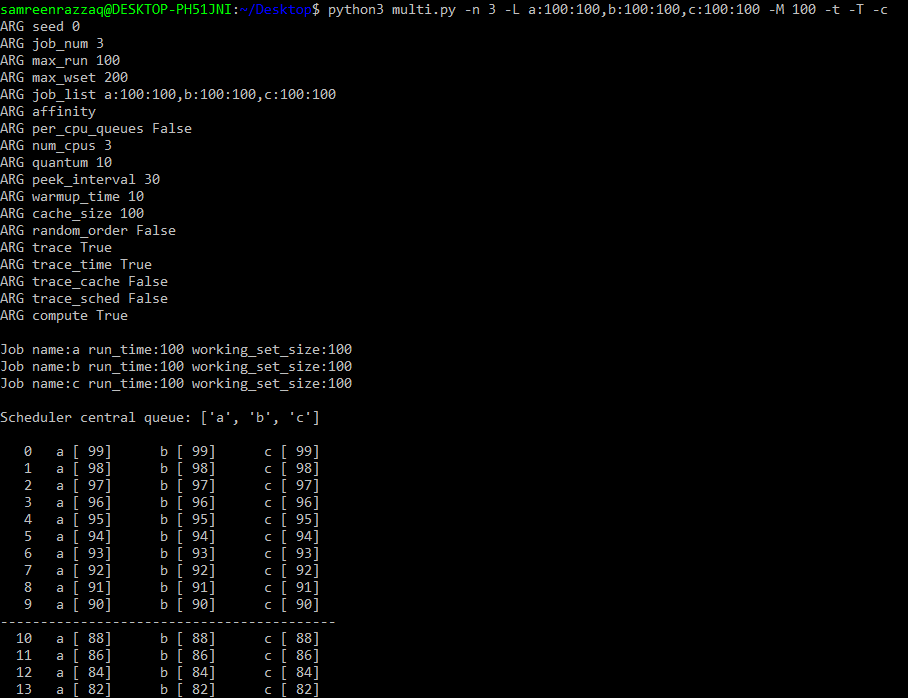
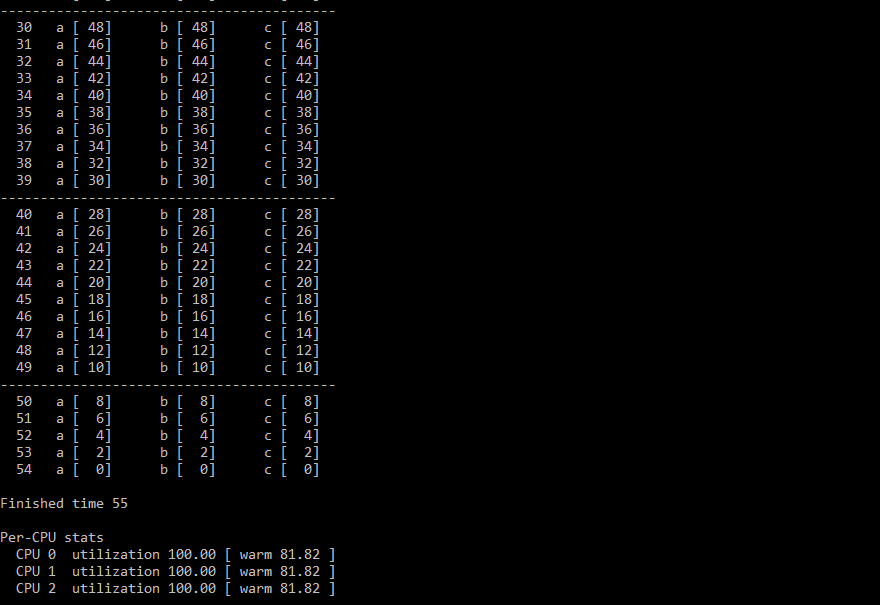


* Secondly we use Cache size = 100 (with 1,2,3 cpus):







**Results:**

* When we use Cache size = 50:

1. Cpu 1 (-n 1): Job running time is 300.
2. Cpu 2 (-n 2): Job running time is 150.
3. Cpu 3 (-n 3): Job running time is 100.

In this, when we increase no.of cpus then job running time decrease and decrease linearly (super linear speed-up) and there cache is not warm in any step.

* When we use Cache size = 100:

1. Cpu 1 (-n 1): Job running time is 300.
2. Cpu 2 (-n 2): Job running time is 150.
3. Cpu 3 (-n 3): Job running time is 55.

In (c), job running time very less than others (Jobs run fastly). That’s why cache is warm-up (warm-up time is 81.82 in all 3 cpus).

8. 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!

In all above questions, we use different sizes of cpus, cache size, use -t flag (trace), -T flag (trace time-left), different lists of jobs and assume results on job running time, warm-up time etc… (all screenshots are attached with results in this document).