**Owen Galvin, CSCI E-88 Homework02**

**NOTE: Looks like I used “assign01” instead of “assign02” as the prefix for majority of solutions files, doesn’t make any sense to update the file names after the fact given the number of screenshots**.

**Problem 1a:**

**write a simple program in your choice of language that does:**

Gray table cell below has the contents of my assign01\_p1a.py:

* + **creates specified number of threads - based on the input parameter**
  + **starts each of those threads**
* that number of threads is passed in on the command-line via the -t/--thread-count variable. A loop in the do\_threads() function creates and starts new threads, counting from 1 up to the number passed into the program
* **each thread does:**
  + **sleep for 10-15 ms**
* also configurable on the command line and defaulting to 10 milliseconds if no value passed in
  + **do some CPU-intensive work - like parsing a RegExp [your choice]**
* my choice was a configurable summing of 3-member combinations of a consecutive pool of integers. The only real reason for making it configurable was during creation of the script – it was easier if I could quickly run through loops at certain times.
  + **do above 2 things forever**
* the while loop in do\_stuff() is designed to run indefinitely if no value is passed in as the debug argument in the script, else do a loop of only 100. At least once during testing I forgot to set the debug and was later wondering why my laptop was vrooming (initial development tested on a local linux VM), so I know it should work.

**Assign01\_p1a.py**

|  |
| --- |
| **import** argparse **from** threading **import** Thread **from** time **import** sleep **import** itertools  **def** do\_stuff(thread\_num, sleep\_for\_ms=10, combo\_iter=300, is\_debug=**False**):  i = 1  **while** i <= 100 **or not** is\_debug:  sleep(sleep\_for\_ms/1000)  sum(sum(i) **for** i **in** itertools.combinations(range(combo\_iter), 3))  **if** is\_debug:  print(**'thread: {} iter: {} combo iter range: {}'**.format(thread\_num, i, combo\_iter))  i += 1  **def** do\_threads(thread\_count, sleep\_milliseconds, combo\_iterations, is\_debug):  **for** i **in** range(1, thread\_count + 1):  new\_thread = Thread(target=do\_stuff, args=(i, sleep\_milliseconds, combo\_iterations, is\_debug))  new\_thread.start()  print(**'thread {} started'**.format(i))   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  parser = argparse.ArgumentParser(description=**'Run selected number of threads'**)  parser.add\_argument(**"-t"**, **"--thread-count"**, type=int, default=1)  parser.add\_argument(**"-m"**, **"--sleep-milliseconds"**, type=int, default=10)  parser.add\_argument(**"-c"**, **"--combo-iterations"**, type=int, default=300)  parser.add\_argument(**"-d"**, **"--debug"**, type=bool, default=**False**)  args = parser.parse\_args()  print(**'BEGIN TEST, thread count: {}, milliseconds to sleep for: {}, combo iterations: {}, debug: {}'**.format(  args.thread\_count, args.sleep\_milliseconds, args.combo\_iterations, args.debug))   do\_threads(args.thread\_count, args.sleep\_milliseconds, args.combo\_iterations, args.debug) |

Not sure if any further details on the code itself are necessary but a typical use pattern might involve running a command like below:

python3 assign01\_p1a.py -t 2 -m 10 -c 300

* which will start up the script with 2 threads, waiting/sleeping for 10 milliseconds in between each calculation. Passing in 300 for the combo-iterations means that (depending on a number of different variables) the calculation should take at least a second.
* do\_threads() creates and starts the expected number of threads. I saw use of join() in the lab but since the primary use case is to run the threads indefinitely I wasn’t sure how the script would benefit.
* do\_stuff() is the bit that does the cpu-intensive calculation, sleeping for desired number of milliseconds before each calc. Normally there is nothing output during the loops but for development and debug purposes there is a print() block that will be executed if --debug argument was activated.
* **start your app and let it run**
* **using Unix tools like ‘top’, ‘ps’ , 'htop' and Java specific ones like ‘jps’ - try to figure out how your threads are mapped to available CPUs - include the results into your solution document (screenshots and explanation)**

The first thing to note here is my environment, which is a Windows 10 laptop hosting, for this first question, a 64-bit CentOS virtual machine, via Virtual Box. (This was easiest way to get started and since problem 2 was going to involve AWS I didn’t feel the need to also do the early steps in AWS.) The laptop itself has 4 physical cores, with hyperthreading another 4 virtual cores. For this first test I’ve assigned the VM 2 CPU cores in the VM’s settings:

|  |
| --- |
|  |

I’ll start the script off telling it to use only one thread,

python3 assign01\_p1a.py -t 1 -m 10 -c 300

|  |
| --- |
|  |

Certainly not much to see in above screenshot (note: this is not in any way a cloudera quickstart VM, I had a user name = cloudera in a vm for a different class and went with it again for no good reason).

Moving on to **htop**, which I had already installed on this VM but I updated display to include more relevant columns. Additionally, I’ve applied a Filter so only the desired python-related processes/threads will be displayed.

|  |
| --- |
|  |

There is more action here but still nothing too exciting. There is the parent process (PID 24780) along with its single child thread (PID 24781) and both share the same PPID (Parent Process ID) and PGRP (Process Group ID). The CPU column indicates which core is being used and I actually see the number jumping around, with the parent process living on CPU 1 but the thread jumping between CPU 1 and 2.

As expected pstree agrees that there is only the single parent/child thread combo:

pstree -p 9051

|  |
| --- |
|  |

Next try a watch/ps command to track the thread, updating the display every 500 milliseconds.

watch -n .5 "ps -eL -o pid,psr,comm | grep python3 | grep -v grep"

|  |
| --- |
|  |

Behavior mirrors that of htop, where obviously this display is zero-based in terms of CPU numbering but otherwise the parent process remains on the first CPU (0 above) but the thread flips back and forth between the two CPU cores.

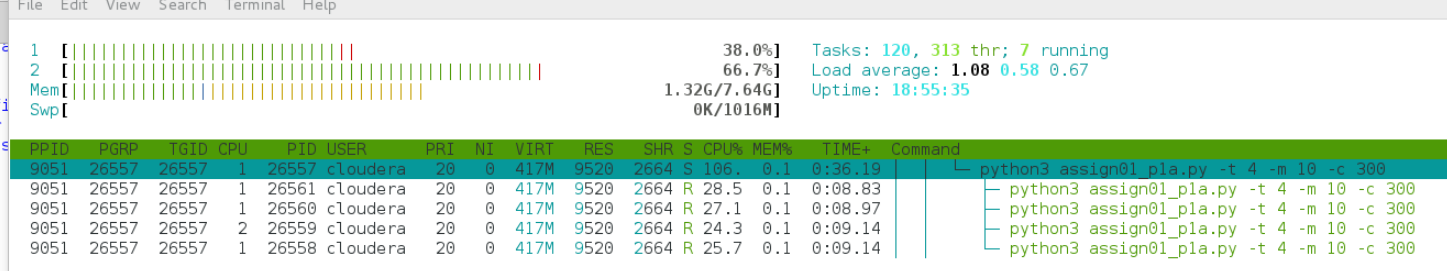
* **change the input parameter (number of threads to start) - see how it affects the CPU usage. Include your results into your solution document  (screenshots and explanation)**

Run same series of steps from above but now telling it to generate 4 threads

python3 assign01\_p1a.py -t 4 -m 10 -c 300

|  |
| --- |
|  |

htop shows similar pattern to before – one parent process with now-multiple threads sharing the same PPID and PGRP but each, including that parent process, with individual PIDs. The value reported for TIME appears to be an aggregation of the time of the individual threads. Again, the parent process begins on one CPU (1) while the four threads bounce around between the two available CPU cores.



pstree on the parent process displays the four child threads but, at least with default arguments, not much other useful info

pstree -p 26557

|  |
| --- |
|  |

Finally, my watch/ps -eL combo also reveals a similar pattern to htop. Parent process on the one CPU and then the four threads bouncing around both/either cores.

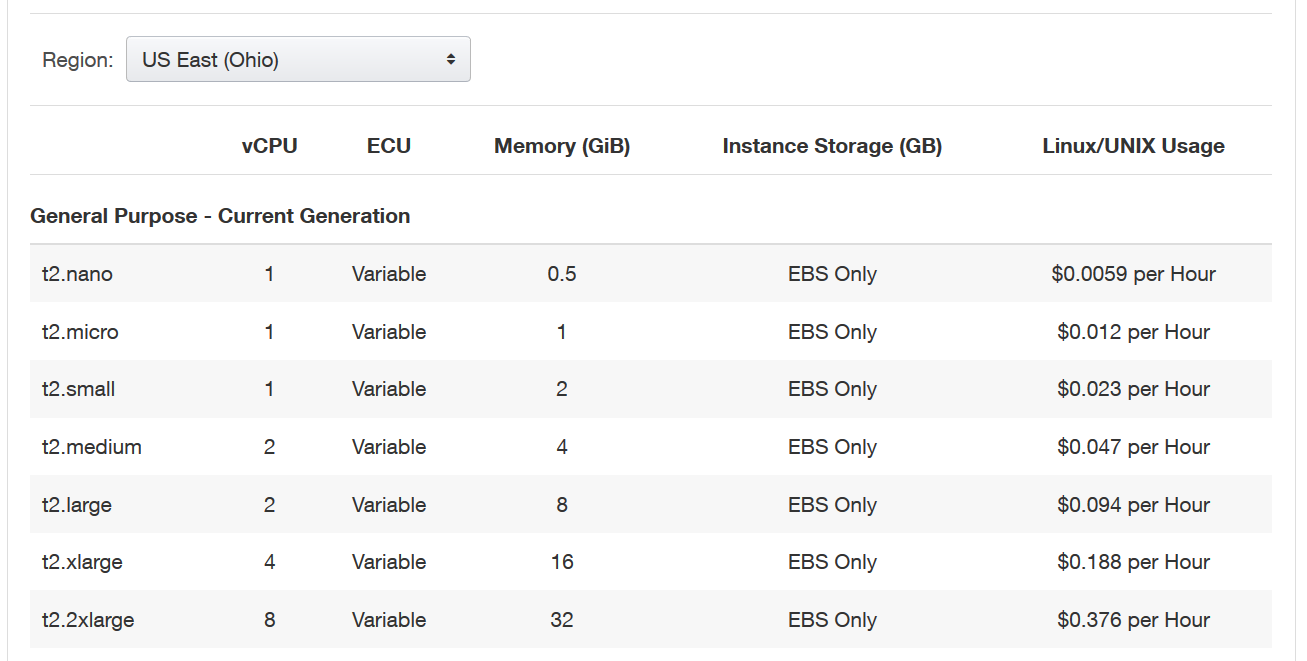
watch -n .5 "ps -eL -o pid,psr,comm | grep python3 | grep -v grep"

|  |
| --- |
|  |

**Problem 2a: [points: 20]**

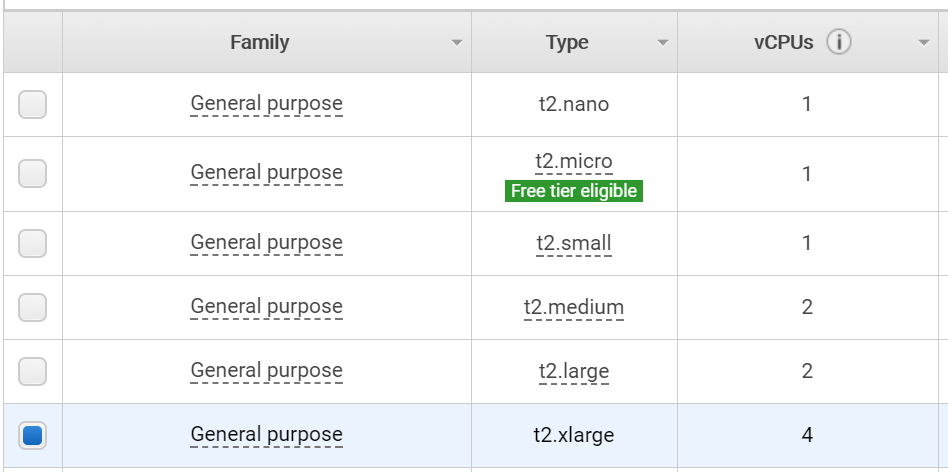
* **Create two EC2 instances in AWS , or two VMs, with different hardware profiles:**

First select the EC2 instances to use in these tests, below screenshot from <https://aws.amazon.com/ec2/pricing/on-demand/> along with comparison to other types on that page indicate **t2.xlarge** (4 vCPU) and **t2.2xlarge** (8vCPU) will likely be most efficient choices.

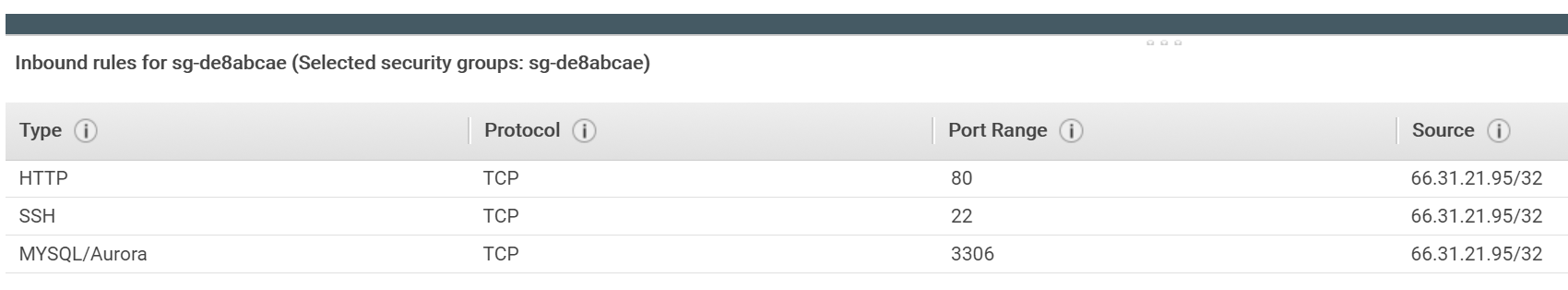


* **1st VM: small number of CPUs (4)**

I presume we can shorten display of some steps here as they were more fully captured in HW1, begin by choosing CentOS 7 (x86\_64) - with Updates HVM from the AMI Marketplace. Choose t2.xlarge as discussed above,

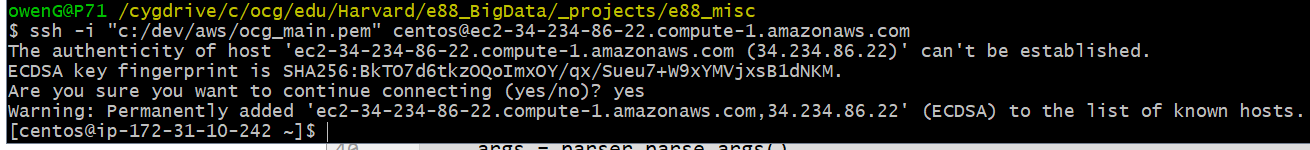


Click Next through most config screens, accepting the default. On Security Group page, make sure to select the group I set up for HW0, so I can ssh right in.



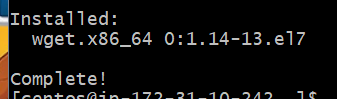
After Launching, ssh in, using standard key

ssh -i "c:/dev/aws/ocg\_main.pem" [centos@ec2-34-234-86-22.compute-1.amazonaws.com](mailto:centos@ec2-34-234-86-22.compute-1.amazonaws.com)



* Install wget

sudo yum install wget

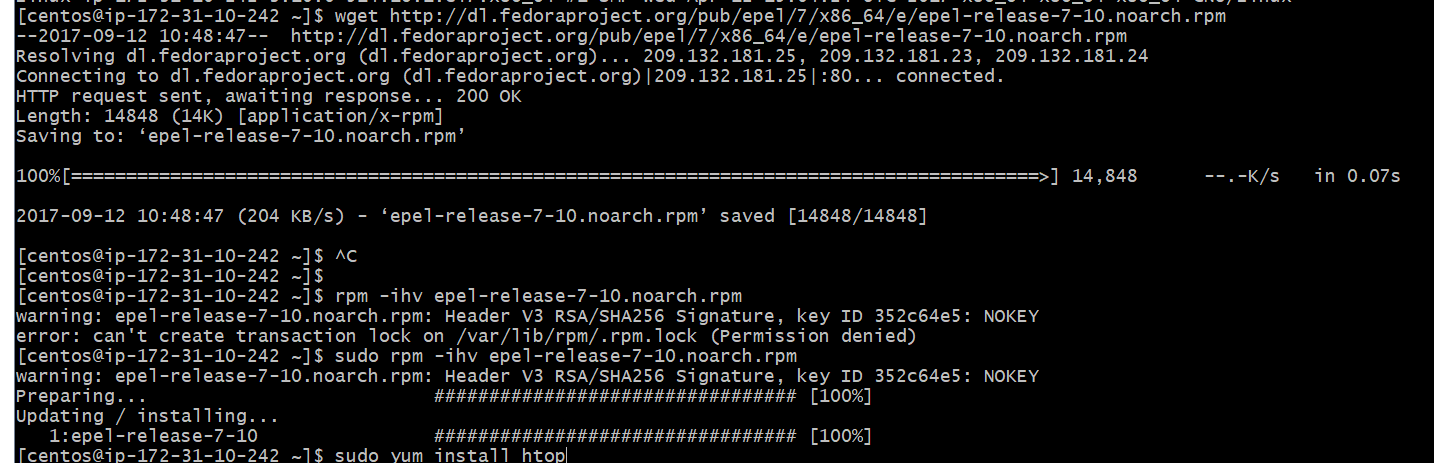


* Install htop, not a one-step process, need to get repo etc. updated first

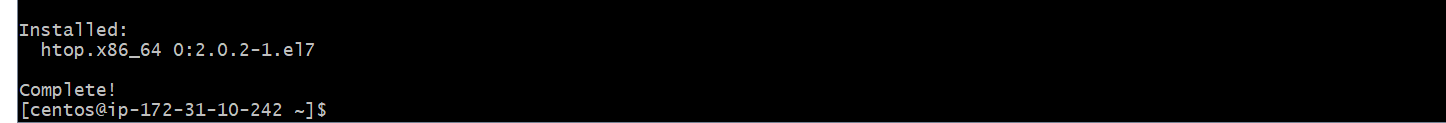
wget <http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-10.noarch.rpm>

rpm -ihv epel-release-7-10.noarch.rpm

sudo yum install htop

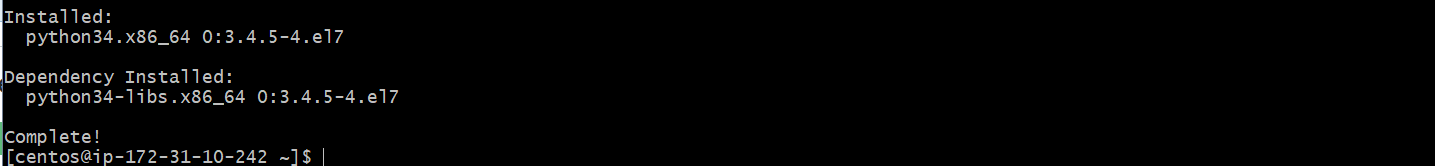


…



* Since my python is written in 3.x, install python 3.4 instead of retrofitting my code to default python 2.7

sudo yum install python34



For sake of expediency I simply do a clipboard transfer to get the code up to my ec2, instead of doing an scp as in Assignment1, saved me a few moments

vi assign01\_p1a.py

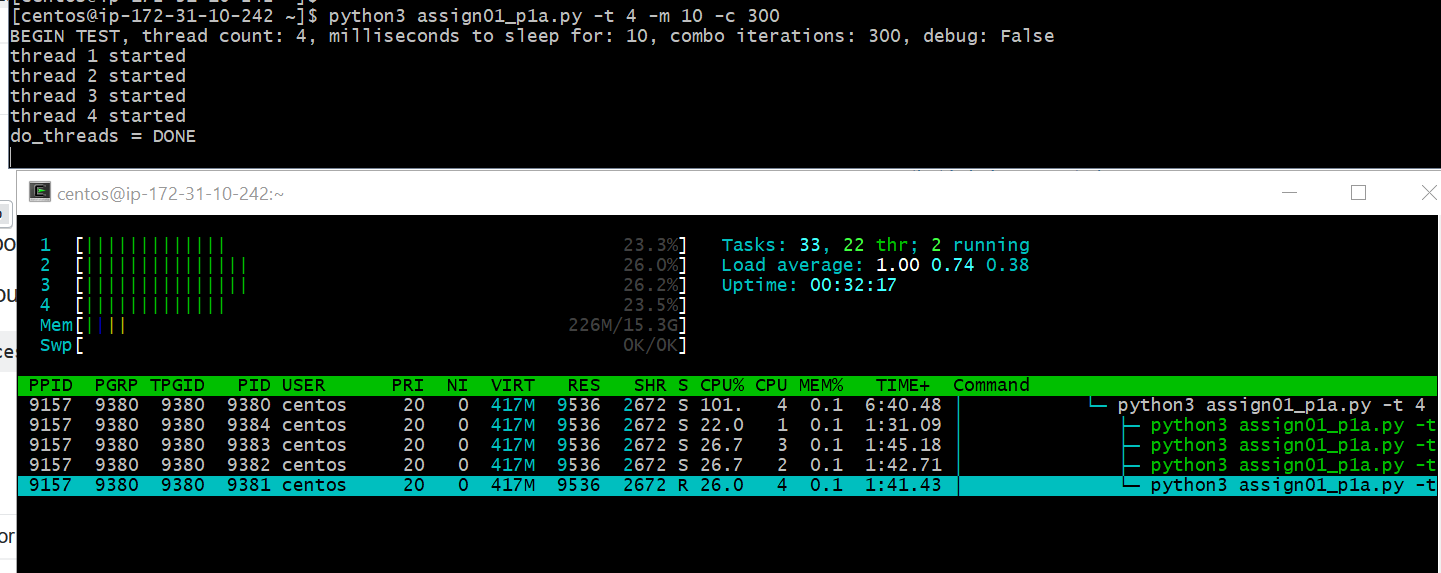


* **Run the program from Problem 1a on both VMs, with the same variations of number of threads as in Problem 1a; monitor CPU and I/O utilization**

(I assume the I/O mention above is only for the bonus question)

Run the script and monitor htop at the same time, running script with 4 threads

python3 assign01\_p1a.py -t 4 -m 10 -c 300

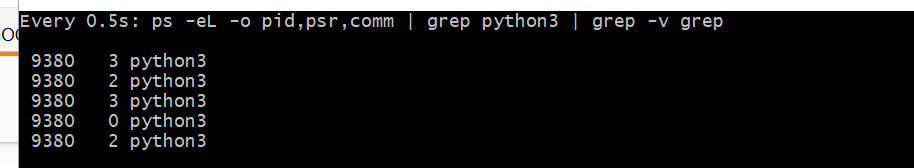


htop has been configured to add some of the useful columns, including PROCESSOR, which appears as CPU in above, column toward the middle. Observe the parent process remaining on CPU = 4 and then the 4 child processes.

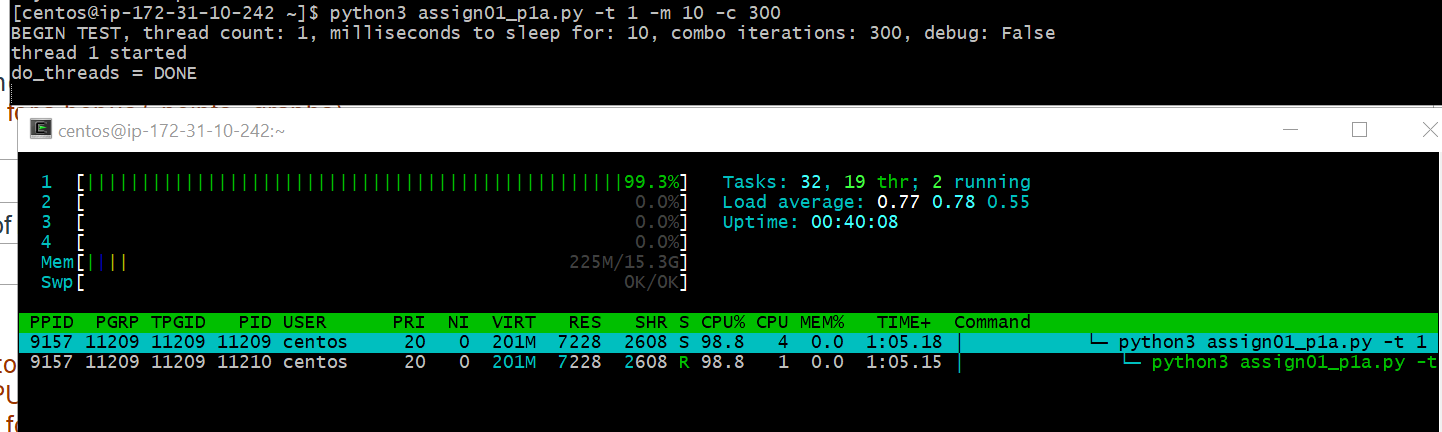
pstree not pre-installed and a simple yum install fails, didn’t reveal much in my linux VM anyway, move on to wath/ps

watch -n .5 "ps -eL -o pid,psr,comm | grep python3 | grep -v grep"

Similar results as on the linux VM, and via htop, though again the CPU core number is zero based. Parent process stays on “3”, child threads bounce around CPUs.

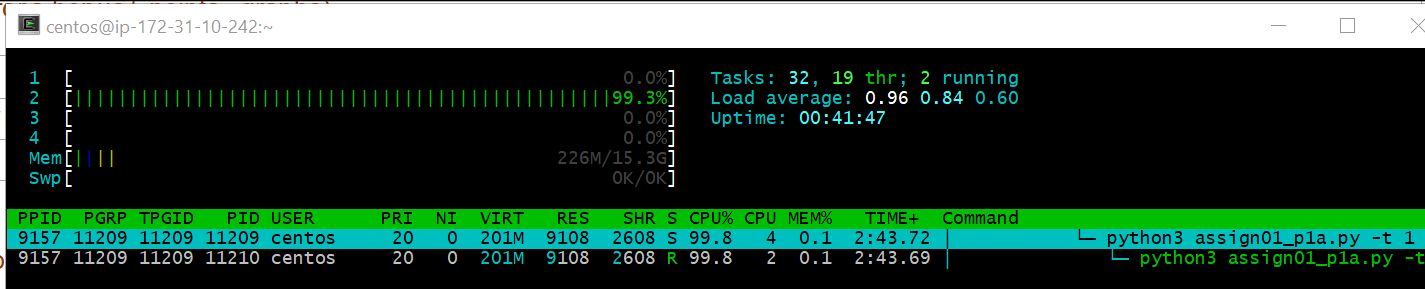


Now run the script again, this time bouncing down to only a single thread



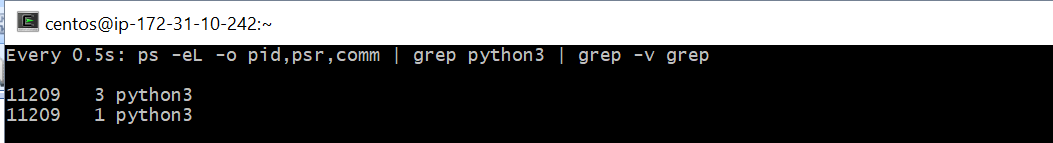
Here we see the single thread appears to be tied to CPU, which is being pegged at 99.3% utilization, instead of the more-or-less even spread that the four threads were doing, where at any given time each CPU was running around 25% utilization. But they were NOT running at ~100%, i.e. each core is not being fully utilized in parallel, instead the 1-core-fully-engaged amount of “work” is spread amongst a set of cores. It would make sense this is the python GIL in action – CPU bound work is not running in parallel across multiple cores.

But there actually is CPU switching going on, check back a minute later



And there is one CPU at 99.3% but it is a different number, 2 this time.

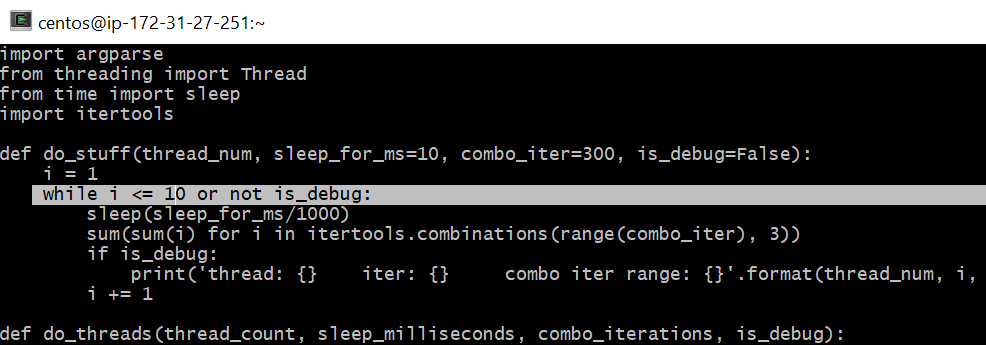
Move on to the watch on ps and not much to see there, the parent process remains on the one CPU, displayed as 3 below, the child thread occasionally moves between cpu cores.

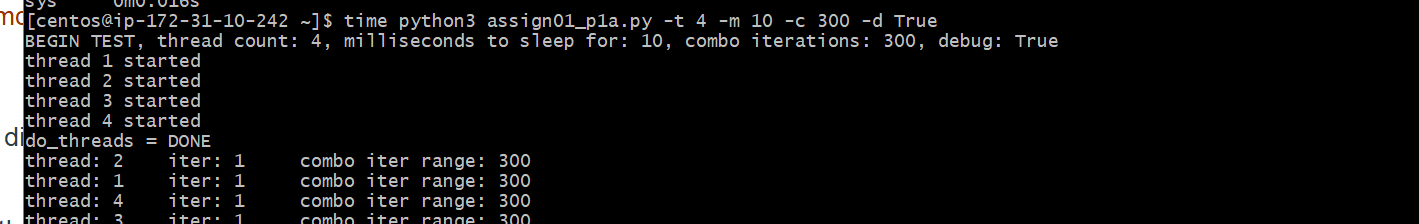


The main takeaway I see here is that the single-threaded application, on a multicore system (or presumably on a single-core system) will monopolize one of the CPU cores, though which core can change depending on whatever black-box calculations the OS uses to make such choices. With multiple threads each core will do relatively less work per-core but the work done will presumably be equivalent.

Homework assignment only mentions the never-ending cpu-intensive loop, so even though the lab discussed use of linux’ **time** command that doesn’t really make sense here, especially as there is no single task to be accomplished that could be set up by spreading across threads. Run a test anyway while my ec2 is up and still in its first hour of life.

4 threads, debug mode, with time **AND** the iteration count reduced from 100 to 10 (hard-coded in python file), so it completes in reasonable time.

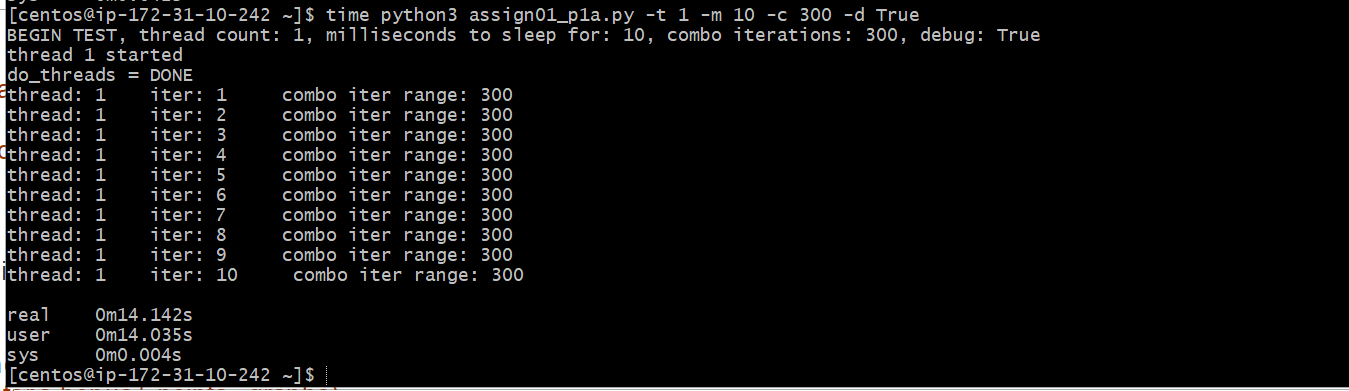


time python3 assign01\_p1a.py -t 4 -m 10 -c 300 -d True

4 threads: results = 4 sets of 10 iterations in 46 seconds



Now with only a single thread:



1 set of 10 iterations in 14 seconds… 4 \* 14 = 56…so same “amount” of calculation horsepower, but it would have all run on a single core (at any given moment, earlier tests show it may switch cores).

* **2nd VM: high number of CPUs (8)**

Repeat the same steps from above, this time beginning with a CentOS 7 (x86\_64) - with Updates HVM with size = **t2.2xlarge** (8vCPU). Again, assign it my e88 security group

|  |
| --- |
|  |

After Launching, ssh in, using standard key

ssh -i "c:/dev/aws/ocg\_main.pem" centos@ec2-54-198-100-55.compute-1.amazonaws.com

* **Run the program from Problem 1a on both VMs, with the same variations of number of threads as in Problem 1a; monitor CPU and I/O utilization**

Run through same setup steps, skipping the screenshots this go-round, which would be the same except for a different ip-address-prompt.

* Install wget

sudo yum install wget

* Install htop, of course not a one-step process, need to get repo etc. updated first

wget <http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-10.noarch.rpm>

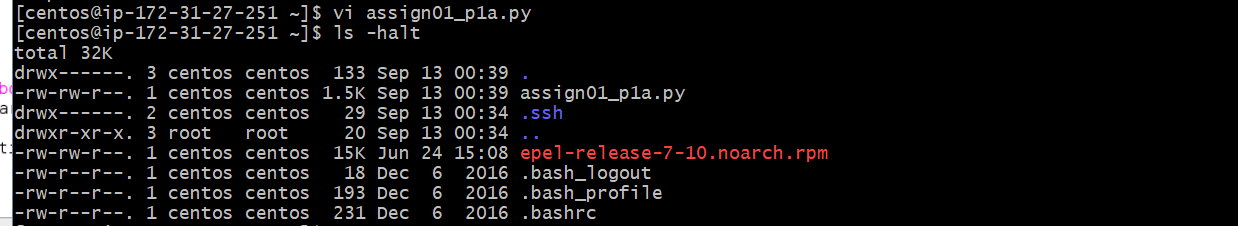
sudo rpm -ihv epel-release-7-10.noarch.rpm

sudo yum install htop

* Since my python is written in 3.x, install python 3.4 instead of updating my code

sudo yum install python34

Clipboard-copy the file = assign01\_p1a.py up to my home directory on the instance.

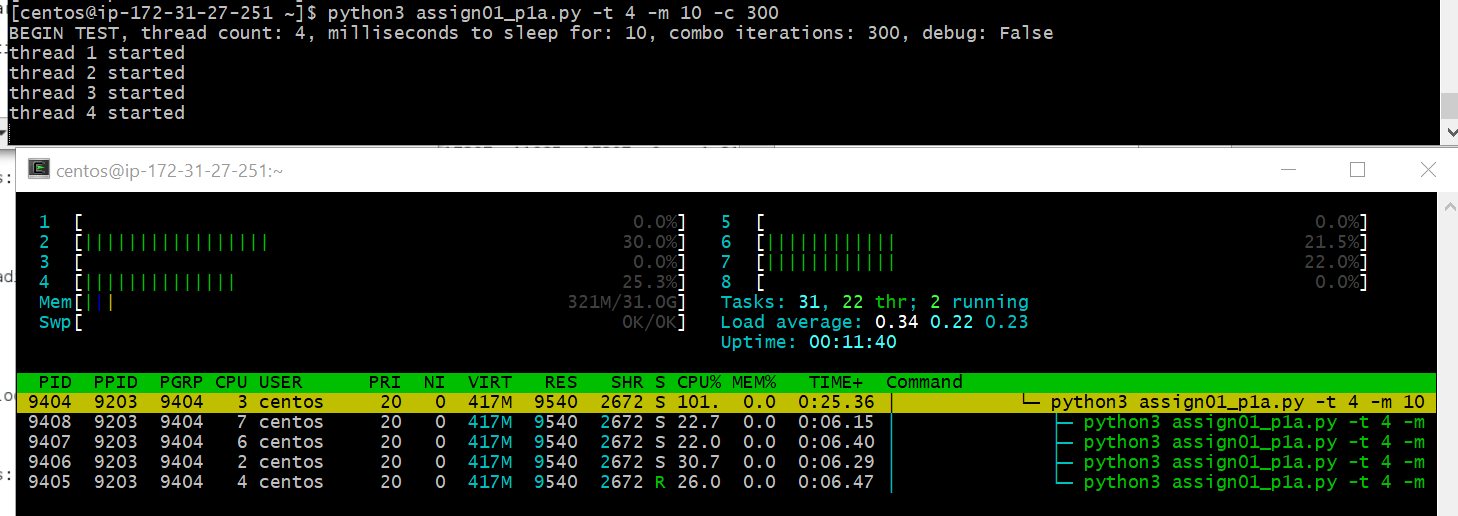


* **Run the program from Problem 1a on both VMs, with the same variations of number of threads as in Problem 1a; monitor CPU and I/O utilization**

Run the script and monitor htop at the same time, running script with 4 threads:

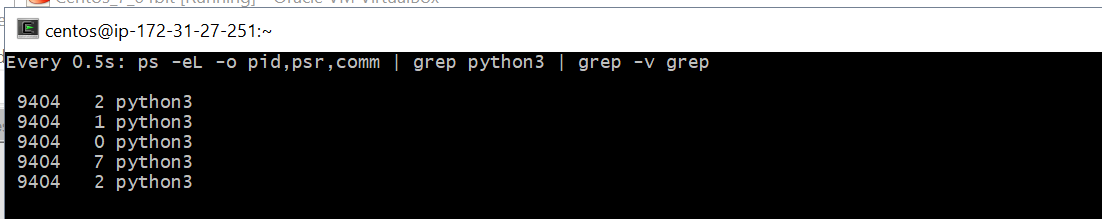
python3 assign01\_p1a.py -t 4 -m 10 -c 300

htop has been configured to add some of the useful columns, including PROCESSOR, which appears as CPU in above, column toward the middle. Filter the output to python3 processes only. Observe the parent process remaining on CPU = 3 and then the 4 child processes do some minimal jumping around… on the 4-core ec2 had notice threads moving to different cores more often. Still happens here but doesn’t seem as common. And yet… after a few minutes of things running begin to see threads running on different cores more often. Was there any kind of warmup associated with instanciating a new ec2 instance? Or a general warmup on the process?



use watch in combination with PS move on to monitor which CPUs threads are running on

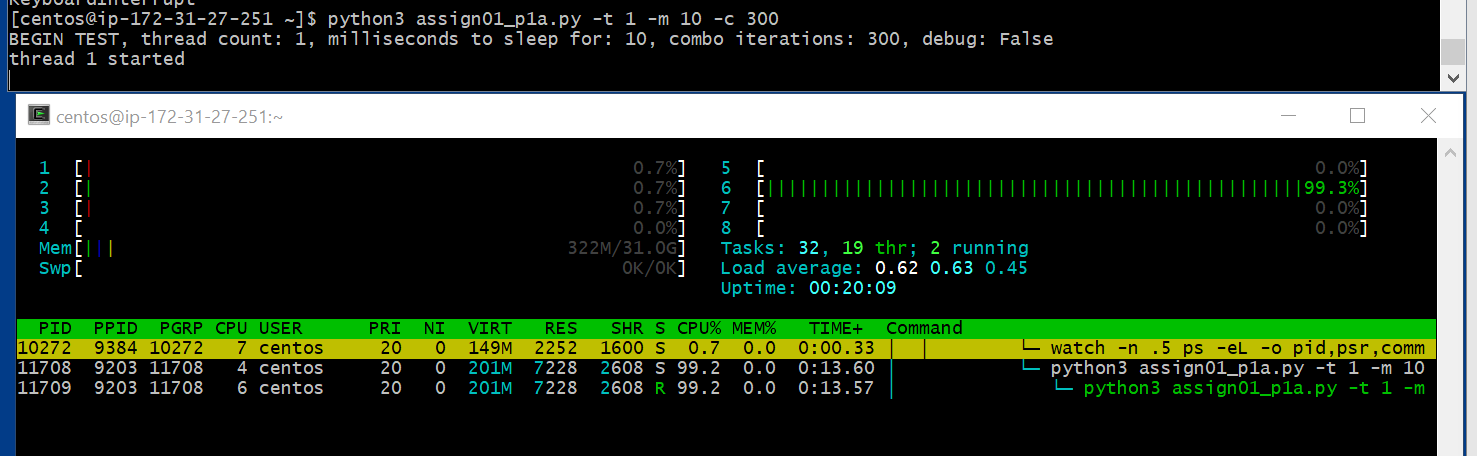
watch -n .5 "ps -eL -o pid,psr,comm | grep python3 | grep -v grep"



Results confirm what was seen more recently with htop, where there is quite a bit of CPU switching each time the watch updates (every 500 milliseconds.)

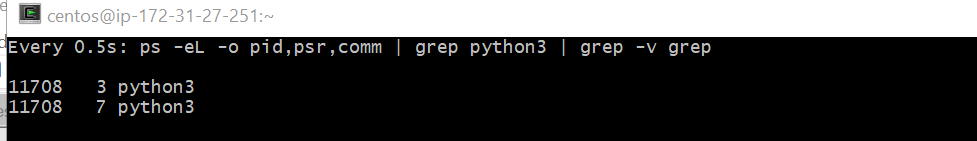
Run the script again, this time bouncing down to only a single thread.

python3 assign01\_p1a.py -t 1 -m 10 -c 300



Results are similar to those of the 4-core experiment. Single thread is bound to a core and runs at near capacity for that CPU. With the 4-core, did see the OS switch thread to a different core occasionally, haven’t seen that here yet…. ok after ~5 minutes it switched to a different CPU. Als,o it seems that at some point the parent thread did indeed switch to a different CPU, only noticed by comparing to screenshot above.

Run watch/ps to see if anything exciting happens

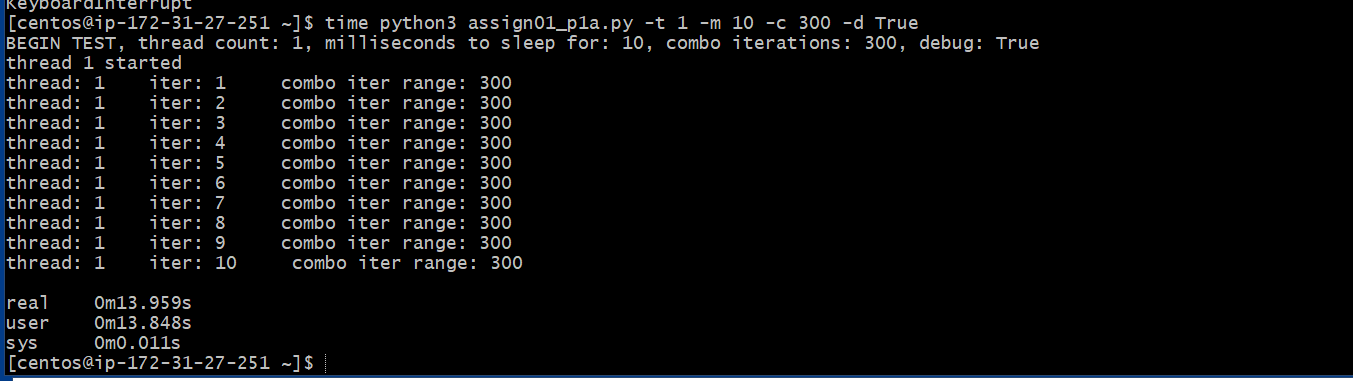


nope. Agrees with htop though once zero-based values accounted for, as expected.

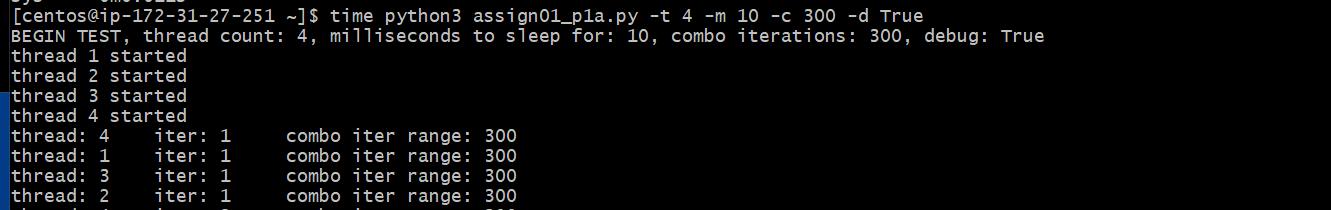
Repeat the time testing, with the script in debug mode:

4 threads, debug mode, with time **AND** the iteration count reduced from 100 to 10, so it completes in reasonable time.

time python3 assign01\_p1a.py -t 1 -m 10 -c 300 -d True



finishes in 14 seconds, repeat with 1 thread

…



4 threads: results = 4 sets of 10 iterations in 58 seconds. Each comparable block has perhaps taken longer in this scenario, as 14 \* 4 = 56 seconds… but only very slightly.

So similar results to that of 4-core system, where we don’t automagically see the four threads running in parallel at 100%. Instead 4x as much work takes 4x as much time to execute. Perhaps with there were other fundamentally different types of needs, e.g. I/O bound actions in some of the threads, then there would be effective speedup with multiple threads -> multiple cores.

**Problem 3: [points: 30]**

**write a multi-threaded program to generate "log" files as following:**

* **specify application parameters: number of distinct userIds, list of URLs , number of "events" per URL per user; number of threads in the app**
* **number of log lines should be: [number of distinct userIds] X [number of URLs] X [number of "events" per URL per user]**
* **create specified number of threads - "writers"**
* **each writer thread should generate one output file with logs in the format (one line == one event)**
  + **<timestamp> <url> <userId>**
* **name of the file should be:**
  + **<threadID>\_events.txt**

While the source file will be included in submission, I’ll paste the whole of it below and then go through each function.

**assign01\_p3.py**

|  |
| --- |
| **import** argparse **from** threading **import** Thread **from** random **import** randint **import** collections **import** os, shutil  **def** prepare\_output\_dir(output\_dir):  **if** os.path.exists(output\_dir):  **if** os.path.isfile(output\_dir):  os.remove(output\_dir)  **else**:  shutil.rmtree(output\_dir)  wait\_seconds = 10  waited = 0  **while** os.path.exists(output\_dir) **and** waited < wait\_seconds:  **import** time  time.sleep(1)  waited += 1  os.mkdir(output\_dir)  *# make a deque with all the (random over a static day) timestamps that will be needed for given set of inputs* **def** get\_all\_timestamps(count):  *#'YYYY-MM-DD HH:MM:SS'* **return** collections.deque(**'2017-09-12 {hour:02}:{minute:02}:{second:02}'**.format(  hour=randint(0,23), minute=randint(0,59), second=randint(0,59)) **for** i **in** range(count))  *# generate a list of user-ids in format of u##, zero-filled from the left as appropriate to total count* **def** get\_userids(count):  **return** [**'u'** + str.zfill(**'{}'**.format(i), len(str(count))) **for** i **in** range(1, count+1)]  **def** generate\_log(thread\_number, output\_dir, data, is\_debug=**False**):  *# import itertools  # sum(sum(i) for i in itertools.combinations(range(300), 3))* log\_path = os.path.join(output\_dir, **'{}\_events.txt'**.format(thread\_number))  **with** open(log\_path, **'w'**) **as** f:  lines = **'\n'**.join(data).strip()  f.write(lines)  **if** is\_debug:  print(**'-- wrote: {}'**.format(log\_path), flush=**True**)  **def** do\_threads(num\_userids, urls, event\_count, thread\_count, total\_num\_events, is\_debug):  output\_dir\_path = **'output'** prepare\_output\_dir(output\_dir\_path)   ts = get\_all\_timestamps(total\_num\_events)  uids = get\_userids(num\_userids)  visits = []  **for** uid **in** uids:  **for** url **in** urls:  **for** i **in** range(event\_count):  visits.append(**'{}\t{}\t{}'**.format(ts.pop(),url, uid,))   *#sort by the random timestamp, for better simulation* visits = sorted(visits) *#, key=lambda line: line.split()[-1])* **if** is\_debug:  **for** visit **in** visits:  print(visit)   log\_threads = []  **for** i **in** range(1, thread\_count + 1):  new\_thread = Thread(target=generate\_log, args=(i, output\_dir\_path, visits, is\_debug))  print(**'thread {} starting'**.format(i), flush=**True**)  log\_threads.append(new\_thread)  new\_thread.start()   *#don't return until all the threads have completed* [t.join() **for** t **in** log\_threads]   **return** output\_dir\_path   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  parser = argparse.ArgumentParser(description=**'Run selected number of threads'**)  parser.add\_argument(**'-n'**, **'--num-userids'**, type=int, default=5)  parser.add\_argument(**'-u'**, **'--urls'**, type=str, default=**''**, help=**'comma separated string of urls'**)  parser.add\_argument(**'-e'**, **'--event-count'**, type=int, default=5)  parser.add\_argument(**'-t'**, **'--thread-count'**, type=int, default=1)  parser.add\_argument(**'-d'**, **'--debug'**, type=bool, default=**False**)  args = parser.parse\_args()  urls = [u **for** u **in** args.urls.split(**','**) **if** u]  url\_count = len(urls)  print(**'BEGIN TEST, userid count: {}, number of urls: {}, number of events: {}, number of threads: {}, debug: {}'**.format(  args.num\_userids, url\_count, args.event\_count, args.thread\_count, args.debug))  total\_event\_count = args.num\_userids \* url\_count \* args.event\_count  print(**'TOTAL EVENT COUNT: {}'**.format(total\_event\_count))   output\_path = do\_threads(args.num\_userids, urls, args.event\_count, args.thread\_count, total\_event\_count, args.debug)   print(**'\nDONE, contents of {}\n - {}'**.format(output\_path, **'\n - '**.join(os.listdir(output\_path)))) |

Individual functions, followed by the initiating python:

**def** prepare\_output\_dir(output\_dir):  
 **if** os.path.exists(output\_dir):  
 **if** os.path.isfile(output\_dir):  
 os.remove(output\_dir)  
 **else**:  
 shutil.rmtree(output\_dir)  
 wait\_seconds = 10  
 waited = 0  
 **while** os.path.exists(output\_dir) **and** waited < wait\_seconds:  
 **import** time  
 time.sleep(1)  
 waited += 1  
 os.mkdir(output\_dir)

First step is to prepare a simple function to help during development. Removes the output dir if it exists and creates a new one to ensure clean set of new log files.

*# make a deque with all the (random over a static day) timestamps that will be needed for given set of inputs***def** get\_all\_timestamps(count):  
 *#'YYYY-MM-DD HH:MM:SS'* **return** collections.deque(**'2017-09-12 {hour:02}:{minute:02}:{second:02}'**.format(  
 hour=randint(0,23), minute=randint(0,59), second=randint(0,59)) **for** i **in** range(count))

Here I generate all the needed timestamp values, based off of an arbitrary hard-coded day in time and then otherwise random hour/seconds in that day. All the timestamp strings are placed onto a queue-type object so that they can be popped off on an as needed basis. Timestamp format wasn’t specified so I went with what I’m most familiar with in RDBMS systems I’ve worked in, simplest to not include any timezone information.

*# generate a list of user-ids in format of u##, zero-filled from the left as appropriate to total count***def** get\_userids(count):  
 **return** [**'u'** + str.zfill(**'{}'**.format(i), len(str(count))) **for** i **in** range(1, count+1)]

Generate and return a list of string userids, in format of ‘u#’ if 9 or fewer user ids needed, ‘u0#’ if 99 or fewer needed, etc.

**def** generate\_log(thread\_number, output\_dir, data, is\_debug=**False**):  
log\_path = os.path.join(output\_dir, **'{}\_events.txt'**.format(thread\_number))  
 **with** open(log\_path, **'w'**) **as** f:  
 lines = **'\n'**.join(data).strip()  
 f.write(lines)  
 **if** is\_debug:  
 print(**'-- wrote: {}'**.format(log\_path), flush=**True**)

generate\_log() is what each thread will call to create the actual log files. The text to write comes in as a list of strings, insert a hard return in between each line and write to file. If debug flag passed in output to console a line indicating a given file was written.

**def** do\_threads(num\_userids, urls, event\_count, thread\_count, total\_num\_events, is\_debug):  
 output\_dir\_path = **'output'** prepare\_output\_dir(output\_dir\_path)  
  
 ts = get\_all\_timestamps(total\_num\_events)  
 uids = get\_userids(num\_userids)  
 visits = []  
 **for** uid **in** uids:  
 **for** url **in** urls:  
 **for** i **in** range(event\_count):  
 visits.append(**'{}\t{}\t{}'**.format(ts.pop(),url, uid,))  
  
 *#sort by the random timestamp, for better simulation* visits = sorted(visits) *#, key=lambda line: line.split()[-1])* **if** is\_debug:  
 **for** visit **in** visits:  
 print(visit)  
  
 log\_threads = []  
 **for** i **in** range(1, thread\_count + 1):  
 new\_thread = Thread(target=generate\_log, args=(i, output\_dir\_path, visits, is\_debug))  
 print(**'thread {} starting'**.format(i), flush=**True**)  
 log\_threads.append(new\_thread)  
 new\_thread.start()  
  
 *#don't return until all the threads have completed* [t.join() **for** t **in** log\_threads]  
  
 **return** output\_dir\_path

Here is where most stuff happens, with a number of arguments determining the details. First thing is a hardcoded relative output folder = “output”; could easily have added as an additional argument to the script but for the assignments purpose this should be fine. This directory is then “prepped” as discussed earlier. Then in successive calls, populate collections with timestamps, and userids respectively. The visits variable will hold the list of log line strings created by the loop that follows, which is pretty clear I believe. Using the deque object allows me to skip a fourth inner loop, instead can simple .pop() each timestamp.

The list of log lines is sorted by timesatmp, more for aesthetic purposes than anything. Since timestamp is the first value in log line and uses the YYYY-MM-DD HH:MM:SS format a call to sorted() is all that is needed. (if in debug mode, print each log line to console.)

Next, since I wanted to print a ‘DONE’ marker once all threads have completed, create an empty list object = log\_threads to which each thread can be appended. Then do a loop to create a new thread up to the number specified via thread\_count arg. Each thread is created with intent of calling generate\_log(), passing in appropriate parameters. Append to log\_threads list and start each thread.

After all threads are spawned, to a .join() on each of the created threads so that the main do\_threads() function doesn’t return until all threads have actually completed their tasks, at which point the directory in which the logs were generated is returned.

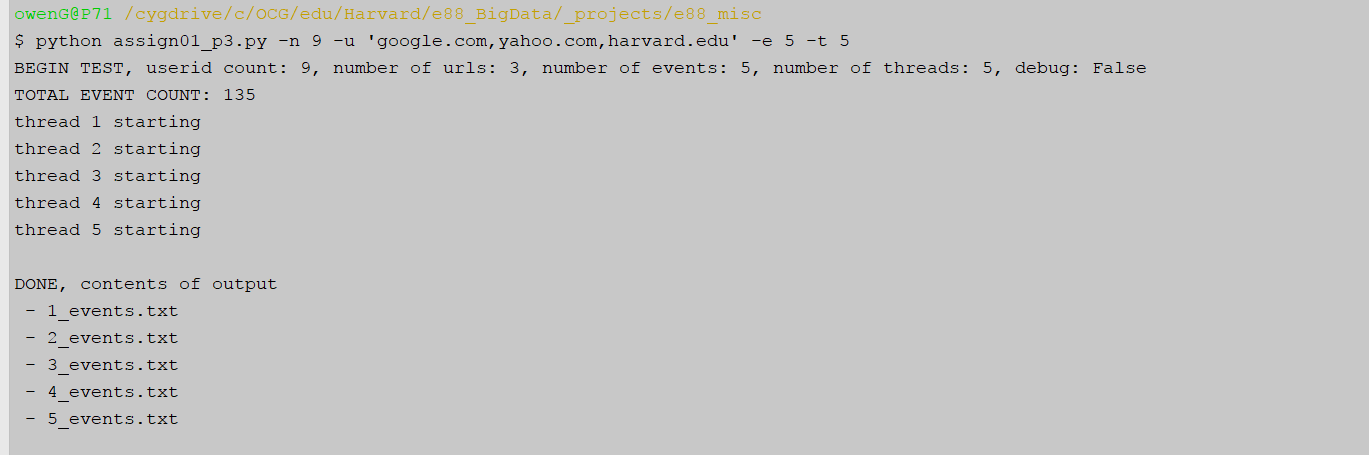
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 parser = argparse.ArgumentParser(description=**'Run selected number of threads'**)  
 parser.add\_argument(**'-n'**, **'--num-userids'**, type=int, default=5)  
 parser.add\_argument(**'-u'**, **'--urls'**, type=str, default=**''**, help=**'comma separated string of urls'**)  
 parser.add\_argument(**'-e'**, **'--event-count'**, type=int, default=5)  
 parser.add\_argument(**'-t'**, **'--thread-count'**, type=int, default=1)  
 parser.add\_argument(**'-d'**, **'--debug'**, type=bool, default=**False**)  
 args = parser.parse\_args()  
 urls = [u **for** u **in** args.urls.split(**','**) **if** u]  
 url\_count = len(urls)  
 print(**'BEGIN TEST, userid count: {}, number of urls: {}, number of events: {}, number of threads: {}, debug: {}'**.format(  
 args.num\_userids, url\_count, args.event\_count, args.thread\_count, args.debug))  
 total\_event\_count = args.num\_userids \* url\_count \* args.event\_count  
 print(**'TOTAL EVENT COUNT: {}'**.format(total\_event\_count))  
  
 output\_path = do\_threads(args.num\_userids, urls, args.event\_count, args.thread\_count, total\_event\_count, args.debug)  
  
 print(**'\nDONE, contents of {}\n - {}'**.format(output\_path, **'\n - '**.join(os.listdir(output\_path))))

And, finally, we have the code that actually gets run when the script itself is called. The expected arguments are pretty self-explanatory, given the specifications of this assignment. The only additional one is ‘debug’, which generally allows for additional print statements and similar to help with both debugging and initial script development. Calculate the url\_count now so it can be displayed in the BEGIN output and be used for the total event count (which will seed the number of timestamps to create). Call do\_threads() to do the actual work of the assignment, where once all the threads have completed the output directory will be returned. The last line prints a directory listing of that output dir, both serving as an indicator the script has completed and also a final sanity check that the log files that were generated match expectations.

On to some screenshots of program execution:

* 9 user ids, 3 urls, 5 events per-user per-url, 5 threads

python assign01\_p3.py -n 9 -u 'google.com,yahoo.com,harvard.edu' -e 5 -t 5

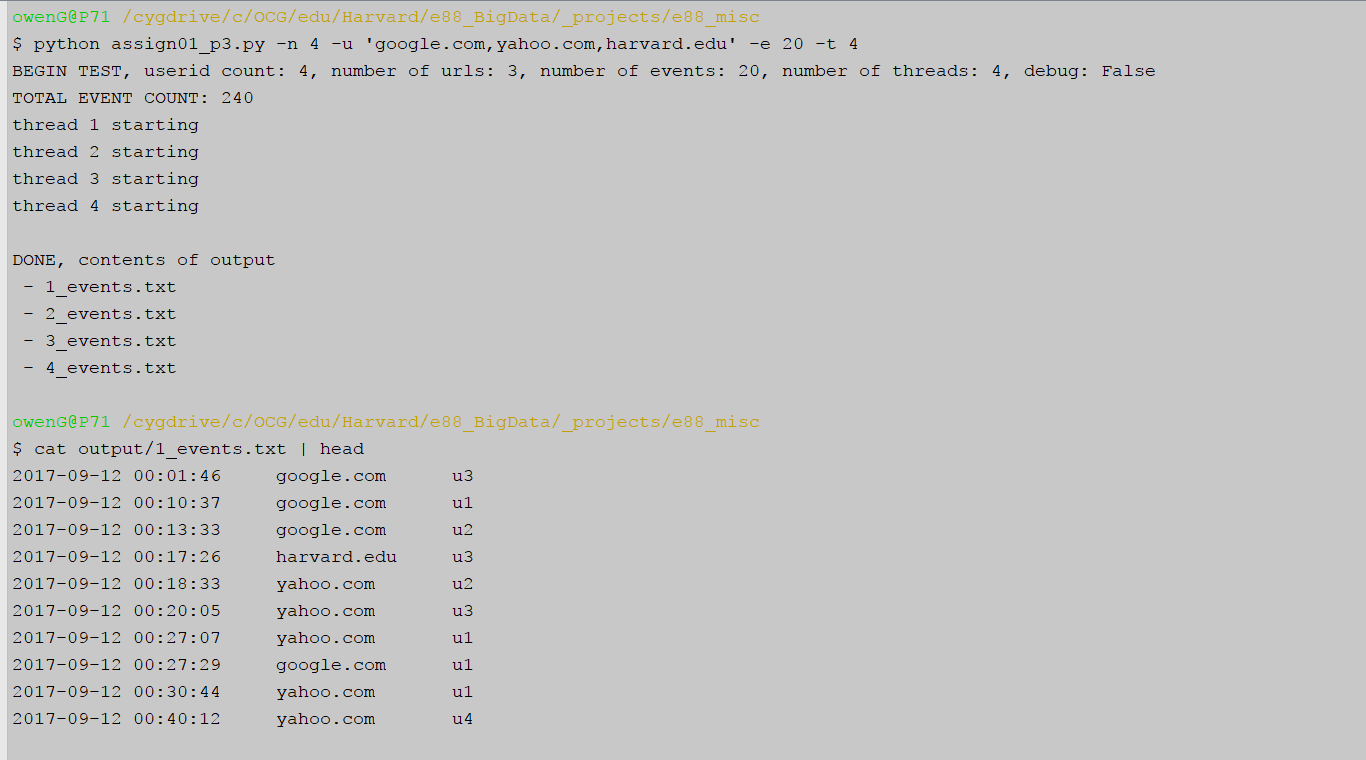


* 4 user ids, three urls, 20 events per-user per-url, 4 threads & then do a head on one of the files

\* this config matches that used for file that gets used in problem 4

python assign01\_p3.py -n 4 -u 'google.com,yahoo.com,harvard.edu' -e 20 -t 4

cat output/1\_events.txt | head



**Problem 4: [points: 60]**

* **implement Example from Lecture 2 (getting unique click and visitor counts): one app multiple threads**
* **decide how to provide separate sets of input files for each thread**

***Lecture 2, Example re-cap: getting unique click and visitor counts***

***--Input: files with events (one line is one event == one user click on a URL) - generated in Problem 3***

***-- write a program , with multiple threads; each thread reads in and processes one input file***

***-- your program has to keep the shared state in memory***

***-- after processing all data files, the program should  output results to the following queries:***

***Query 1: get count of unique URLs***

***Query 2: get count of unique visitors per URL***

***Query 3: get count of unique (by userId) clicks per URL***

Going to follow same pattern as with problem 3, though this one has a few more inline code comments because it would have been too onerous to talk about the map/reduce steps otherwise.

**assign01\_p4.py (full)**

|  |
| --- |
| **import** argparse **import** collections **import** pprint **import** os **from** threading **import** Thread **from** queue **import** Queue **from** random **import** randint   **def** prep\_input\_files(master\_log\_file, input\_folder, is\_debug):  num\_new\_files = 4   inputs = collections.defaultdict(list)  **with** open(master\_log\_file, **'r'**) **as** f:  **for** line **in** f.readlines():  *# each line will be randomly assigned to one of the files that will be written  # each file will wind up with similar number of lines but rarely the same number (unless source file has many lines)* file\_num = randint(1, num\_new\_files )  inputs[file\_num].append(line)  **if** is\_debug:  pprint.pprint(inputs)   *# for each group of clicks, write them to a new file* **for** k,v **in** inputs.items():  **with** open(**'{}/{}\_input.log'**.format(input\_folder, k), **'w'**) **as** f:  f.writelines(v)   **return** os.listdir(input\_folder)  **def** parse\_log(q, file\_path, is\_debug):  **with** open(file\_path, **'r'**) **as** f:  single\_map = collections.defaultdict(dict)  **for** line **in** f.readlines():  *# source log files are tab separated* timestamp, url, user\_id = line.strip().split(**'\t'**)  **if** url **in** single\_map **and** user\_id **in** single\_map[url]:  *# this url already registered a hit for this uid, increment the count* single\_map[url][user\_id] = single\_map[url][user\_id] + 1  **else**:  *# either a brand new url or a brand new user to the existing url, register as a new hit* single\_map[url].update({user\_id: 1})   **if** is\_debug:  print(**'----'**, file\_path)  pprint.pprint(single\_map)  q.put(single\_map)  **def** update\_dict(d, timestamp, url, user\_id):  **if** url **in** d:  **if** user\_id **in** d[url]:  d[url][user\_id] = d[url][user\_id] + 1  **else**:  d[url].update({user\_id: 1})  **else**:  new\_user = {user\_id: 1}  d[url].update(new\_user)  *#****TODO: handle scenario where number of available threads <> number of files* def** process\_files(input\_folder, is\_debug):  log\_threads=[]  q = Queue()  **for** i, input\_file **in** enumerate(os.listdir(input\_folder)):  new\_thread = Thread(target=parse\_log, args=(q, os.path.join(input\_folder, input\_file), is\_debug))  print(**'thread {} starting'**.format(i+1), flush=**True**)  log\_threads.append(new\_thread)  new\_thread.start()   *# .join() on all the threads so the queue won't be returned until it is fully populated* [t.join() **for** t **in** log\_threads]   **return** q  **def** reduce\_maps(q, is\_debug):  *# there will be at least one map in the queue, get the first one and update it in place if there are more* base\_map = q.get()  **if** is\_debug:  print(**'\*\*\* first dict'**)  pprint.pprint(base\_map)   **while not** q.empty():  new\_map = q.get()  **if** is\_debug:  print(**'\*\*\* next dict'**)  pprint.pprint(new\_map)  *# url is the key for top level dict, iterate through the urls at top level* **for** url, user\_dict **in** new\_map.items():  **if** url **in** base\_map:  *# in this toy example all the urls in additional maps will be in that base\_map, this will be expected path* **for** uid, count **in** user\_dict.items():  *# iterate through the user\_id: count map for the current url in "new" map* **if** uid **in** base\_map[url]:  *# this uid already appeared for this url, increment count* base\_map[url][uid] = base\_map[url][uid] + count  **else**:  *# new uid for current url, transfer the count from current map to base\_map* base\_map[url][uid] = count  **else**:  *# should a new url be encountered, add it to the base map* base\_map[url] = user\_dict   **if** is\_debug:  print(**'\n\*\*\* REDUCED dict'**)  pprint.pprint(base\_map)   **return** base\_map   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  parser = argparse.ArgumentParser(description=**'** **Use map of maps strategy to map/reduce source log data'**)  parser.add\_argument(**'-s'**, **'--source-file'**, type=str, help=**'leave empty to skip creation of new input files'**)  parser.add\_argument(**'-l'**, **'--log-directory'**, type=str, default=**'input'**,  help=**'folder in which results of processing source file will be saved'**)  parser.add\_argument(**"-d"**, **"--debug"**, type=bool, default=**False**)  args = parser.parse\_args()   **if** args.source\_file:  written = prep\_input\_files(args.source\_file, args.log\_directory, args.debug)  print(**'DONE, log directory contents: \n - {}'**.format(**'\n - '**.join(written)))   q = process\_files(args.log\_directory, args.debug)  reduced = reduce\_maps(q, args.debug)  print(**'Query 1: get count of unique URLs: {}'**.format(len(reduced)))  query\_2 = [**'{}: {}'**.format(url, len(user)) **for** url, user **in** reduced.items()]  print(**'Query 2: get count of unique visitors per URL: \n - {}'**.format(**'\n - '**.join(query\_2)))  print( **'Query 3: get count of unique (by userId) clicks per URL: '**)  **for** url,uids **in** reduced.items():  print(**' - {}'**.format(url))  **for** uid, count **in** uids.items():  print(**' - {}: {}'**.format(uid, count)) |

The first function relates to the bit about providing a set of input files for each thread and is designed to be run on-demand only, i.e. via the correct command line argument new files may be generated but it would be even more common to run the script agains a set of previously generated files. Either way the main inputs to this function are a “master” log file = one of the files created in problem 4. The log file against which I did most runs had 240 events – the arguments to the problem 4 python had been for 4 userids \* 3 urls \* 20 click events. I initially split the master log file into 4 pieces based on userid but of course that made the mapping process rather artificial as one “mapper” only handled a single userid. I switched instead to using random number generator such that each file would wind up with ~25% of the source lines. With the relatively small numbers present here I saw new files that were being generated with lines that were well over/under 25% but that was fine and potentially more representative of a real-world scenario.

|  |
| --- |
| **def** prep\_input\_files(master\_log\_file, input\_folder, is\_debug):  num\_new\_files = 4   inputs = collections.defaultdict(list)  **with** open(master\_log\_file, **'r'**) **as** f:  **for** line **in** f.readlines():  *# each line will be randomly assigned to one of the files that will be written  # each file will wind up with similar number of lines but rarely the same number (unless source file has many lines)* file\_num = randint(1, num\_new\_files )  inputs[file\_num].append(line)  **if** is\_debug:  pprint.pprint(inputs)   *# for each group of clicks, write them to a new file* **for** k,v **in** inputs.items():  **with** open(**'{}/{}\_input.log'**.format(input\_folder, k), **'w'**) **as** f:  f.writelines(v)   **return** os.listdir(input\_folder) |

The first function that tackles the main purpose of problem 4 is the log parser, which starts by making a defaultdict that itself uses a dict as the default object for its value. Iterate over each line in the source file and pull out the timestamp/url/usre\_id by splitting on a tab character. Then as per the inline comments, either add any brand new url/user\_id, or if already encountered, increment its count by 1. Once the file is processed the final map (dictionary of dictionary) is placed onto the Queue object that was passed in.

|  |
| --- |
| **def** parse\_log(q, file\_path, is\_debug):  **with** open(file\_path, **'r'**) **as** f:  single\_map = collections.defaultdict(dict)  **for** line **in** f.readlines():  *# source log files are tab separated* timestamp, url, user\_id = line.strip().split(**'\t'**)  **if** url **in** single\_map **and** user\_id **in** single\_map[url]:  *# this url already registered a hit for this uid, increment the count* single\_map[url][user\_id] = single\_map[url][user\_id] + 1  **else**:  *# either a brand new url or a brand new user to the existing url, register as a new hit* single\_map[url].update({user\_id: 1})   **if** is\_debug:  print(**'----'**, file\_path)  pprint.pprint(single\_map)  q.put(single\_map) |

The main job of process\_files() is to instantiate the threads that will run the multiple calls to parse\_log. The mapping between source files and threads in current code is quite simple = create a new thread for each file in the input\_folder directory. The magic to coordinating the results is an instance of queue.Queue – that single object gets passed to each thread’s function and in turn each of those spawned functions put their “maps” onto that single Queue – doesn’t matter what order that happens in. Once all the threads have completed the populated Queue is returned.

|  |
| --- |
| *#****TODO: handle scenario where number of available threads <> number of files* def** process\_files(input\_folder, is\_debug):  log\_threads=[]  q = Queue()  **for** i, input\_file **in** enumerate(os.listdir(input\_folder)):  new\_thread = Thread(target=parse\_log, args=(q, os.path.join(input\_folder, input\_file), is\_debug))  print(**'thread {} starting'**.format(i+1), flush=**True**)  log\_threads.append(new\_thread)  new\_thread.start()   *# .join() on all the threads so the queue won't be returned until it is fully populated* [t.join() **for** t **in** log\_threads]   **return** q |

Below is where the “reduce” actions take place. The first map (dict of dicts) is pulled off of the Queue and, if that were to be the only item on the Queue it would wind up being returned. Otherwise, each successive map is popped off the queue and processed in a manner similar to how the maps were created in the first place. The inline comments explain the details. At the end, the first map on the queue will have been updated with the contents of each successive map.

|  |
| --- |
| **def** reduce\_maps(q, is\_debug):  *# there will be at least one map in the queue, get the first one and update it in place if there are more* base\_map = q.get()  **if** is\_debug:  print(**'\*\*\* first dict'**)  pprint.pprint(base\_map)   **while not** q.empty():  new\_map = q.get()  **if** is\_debug:  print(**'\*\*\* next dict'**)  pprint.pprint(new\_map)  *# url is the key for top level dict, iterate through the urls at top level* **for** url, user\_dict **in** new\_map.items():  **if** url **in** base\_map:  *# in this toy example all the urls in additional maps will be in that base\_map, this will be expected path* **for** uid, count **in** user\_dict.items():  *# iterate through the user\_id: count map for the current url in "new" map* **if** uid **in** base\_map[url]:  *# this uid already appeared for this url, increment count* base\_map[url][uid] = base\_map[url][uid] + count  **else**:  *# new uid for current url, transfer the count from current map to base\_map* base\_map[url][uid] = count  **else**:  *# should a new url be encountered, add it to the base map* base\_map[url] = user\_dict   **if** is\_debug:  print(**'\n\*\*\* REDUCED dict'**)  pprint.pprint(base\_map)   **return** base\_map |

The final piece is the initiating code

* ‘-s’ if the single source log file has to first be partitioned into separate files
* ‘-l’ is for the log directory, the output directory into which aggregated file will be written
* ‘-d’ to include additional print statements to show object contents etc.

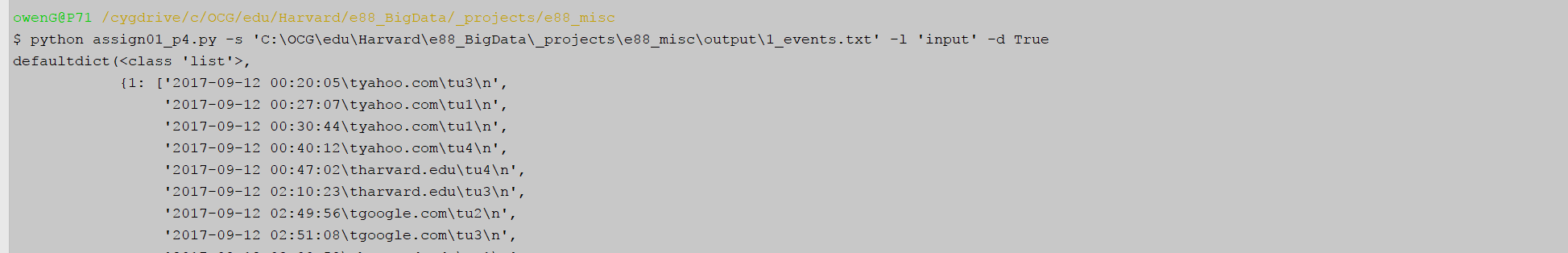
If the ‘-s’ is passed, prep\_input\_files() is called to create the multiple source files. Otherwise it goes directly into getting a queue.Queue object populated with the multiple per-file maps. That Queue is passed into reduce\_maps(), which combines the multiple maps into one final map-of-maps that contains all the data.

Query 1 is answered simply by getting the length of the parent map, the number of keys. Query 2 involves iterating through each of the child maps and summing up the number of uids per parent map (the url). Query 3 is basically the contents of the map of maps, recursively travel through those and print out contents.

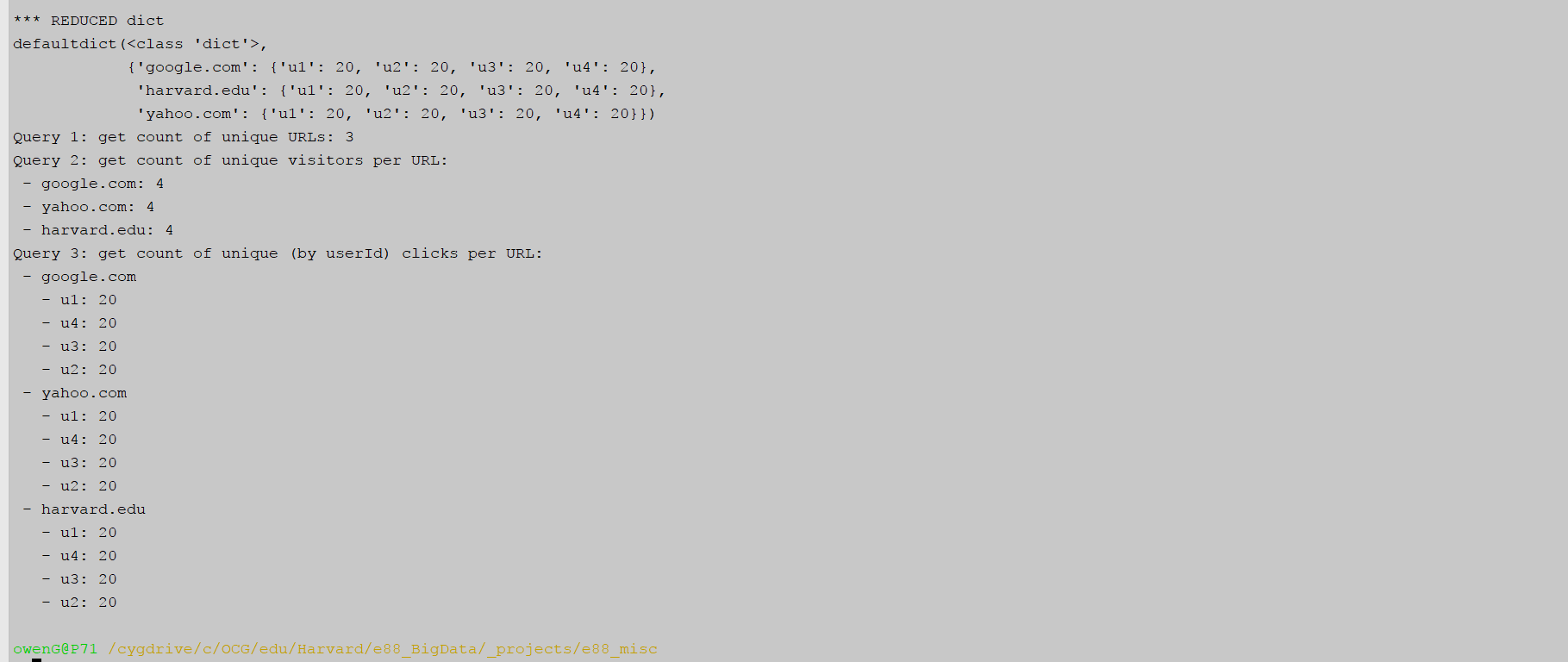
|  |
| --- |
| **if** \_\_name\_\_ == **'\_\_main\_\_'**:  parser = argparse.ArgumentParser(description=**'** **Use map of maps strategy to map/reduce source log data'**)  parser.add\_argument(**'-s'**, **'--source-file'**, type=str, help=**'leave empty to skip creation of new input files'**)  parser.add\_argument(**'-l'**, **'--log-directory'**, type=str, default=**'input'**,  help=**'folder in which results of processing source file will be saved'**)  parser.add\_argument(**"-d"**, **"--debug"**, type=bool, default=**False**)  args = parser.parse\_args()   **if** args.source\_file:  written = prep\_input\_files(args.source\_file, args.log\_directory, args.debug)  print(**'DONE, log directory contents: \n - {}'**.format(**'\n - '**.join(written)))  q = process\_files(args.log\_directory, args.debug)  reduced = reduce\_maps(q, args.debug)  print(**'Query 1: get count of unique URLs: {}'**.format(len(reduced)))  query\_2 = [**'{}: {}'**.format(url, len(user)) **for** url, user **in** reduced.items()]  print(**'Query 2: get count of unique visitors per URL: \n - {}'**.format(**'\n - '**.join(query\_2)))  print(**'Query 3: get count of unique (by userId) clicks per URL: '**)  **for** url,uids **in** reduced.items():  print(**' - {}'**.format(url))  **for** uid, count **in** uids.items():  print(**' - {}: {}'**.format(uid, count)) |

Now to run the script, which will include generating the four partitioned log files as a first step. Then as discussed above the map-reduce-like actions bring those back into a single map-of-maps entity.

python assign01\_p4.py -s 'C:\OCG\edu\Harvard\e88\_BigData\\_projects\e88\_misc\output\1\_events.txt' -l 'input' -d True

(plan to attach below output as p4\_debug\_output.txt)

…



As would be expected, each of the three urls wound up with exactly four userids, with 20 events each, which matches the parameters used for generating the source file, in problem 3:

python assign01\_p3.py -n 4 -u 'google.com,yahoo.com,harvard.edu' -e 20 -t 4

**Handling of parsing errors is defined**

I took a stab at the bonus part of this question but it isn’t part of the submitted python file since I had basically finished the script and didn’t want to perhaps cause damage by doing a full implementation of error logging.

|  |
| --- |
| **def** parse\_log(q, file\_path, is\_debug):  **with** open(file\_path, **'r'**) **as** f:  single\_map = collections.defaultdict(dict)  **for** i,line **in** enumerate(f.readlines()):  *# source log files are tab separated* **try**:  timestamp, url, user\_id = line.strip().split(**'\t'**)  **except** ValueError **as** e:  print(**'ERROR PARSING LINE {} in FILE {}: {}, problem line:\n\t{}'**.format(i+1, file\_path, e, line))  **continue  if** url **in** single\_map **and** user\_id **in** single\_map[url]:  *# this url already registered a hit for this uid, increment the count* single\_map[url][user\_id] = single\_map[url][user\_id] + 1  **else**:  *# either a brand new url or a brand new user to the existing url, register as a new hit* single\_map[url].update({user\_id: 1}) |

Inserted a try/except to catch any ValueErrors that might crop up due to failed parsing of a line. If error, print it and then call continue so that the loop continues on to the next line. Printing directly to output strikes me as a partial fix only though and a more robust solution would involve passing in another Queue object, dedicated solely to error lines. Toward end of program, if parsing errors then print to file, provide a warning and display path to the errror file.

**Problem 7: [points: 30]**

**setup a 3[or 5]-node AWS EMR cluster (alternatively, you could build a 3[5]-node cluster on VMWare VMs - using Cloudera QuickStart VMs or installing Hadoop manually, but it is much harder to do)**

Starting in EMR Services section, click Create Cluster

|  |
| --- |
|  |

Select below options and create the cluster

|  |
| --- |
|  |

* Get the latest EMR Release = 5.8.0
* Base core Hadoop services are the default and only need minimal components for this assignment
* choose m3.xlarge, 3 nodes as possibly the cheapest of the standard options (had tried creating a cluster with m1.medium since it was the cheapest available and we weren’t actually executing any jobs with this assignment, but cluster creation failed with message saying that instance type actually wasn’t available in selected availability zone)

After waiting several minutes cluster becomes available.

|  |
| --- |
|  |

* **identify your master and slave nodes - include screenshots that show that**

The above screenshot indicates which of the three EC2’s is the master node:

**Master public DNS:** ec2-54-165-42-20.compute-1.amazonaws.com

Click on View cluster details in above screenshot and follow the first link for the security group for the master node, seen below.

|  |
| --- |
|  |

In security group listing select that sg-ca4bf6b9 security group and on the Inbound tab, click Edit and add one more Rule to the long list of auto-generated rules, selecting to only allow access from my current IP address:

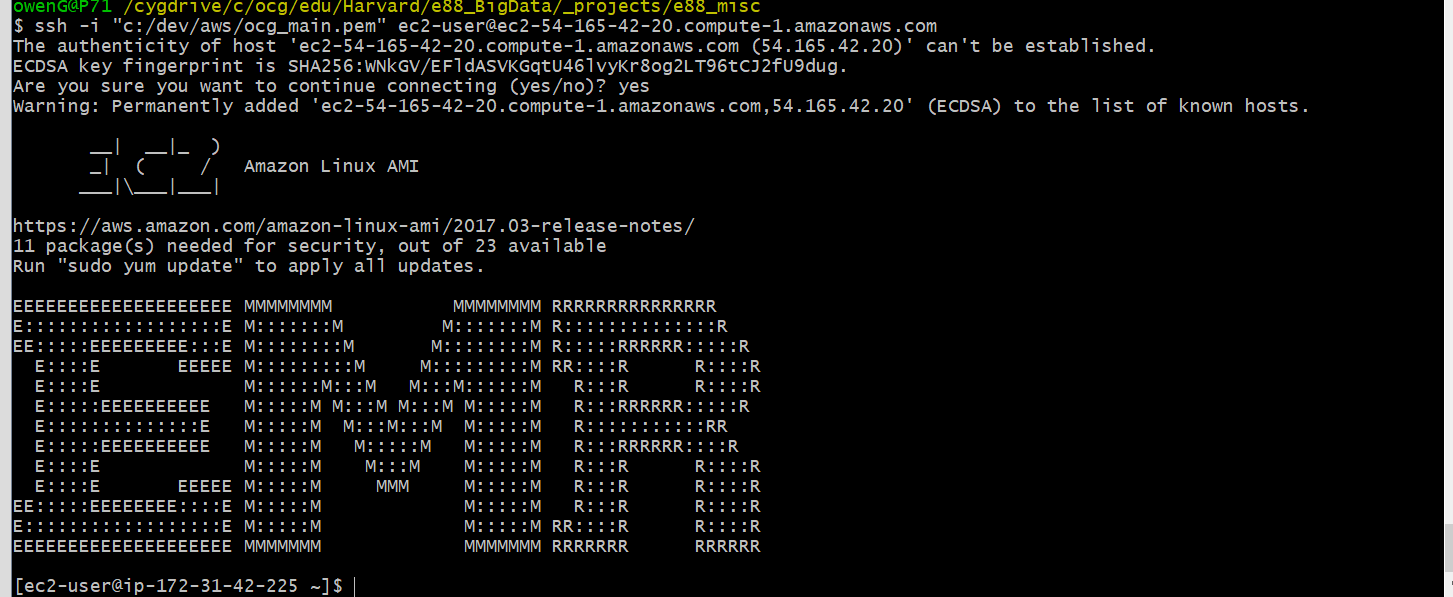
|  |
| --- |
|  |

Then select the security group with Group Name = “ElasticMapReduce-slave” and do the exact same thing:

|  |
| --- |
|  |

First ssh into the master

ssh -i "c:/dev/aws/ocg\_main.pem" ec2-user@ec2-54-165-42-20.compute-1.amazonaws.com



Easiest way to identify emr slave nodes is to go to my running ec2 instances and locate the two that are NOT the master nodes

|  |
| --- |
|  |

(could also go to Cluster details, select Hardware)

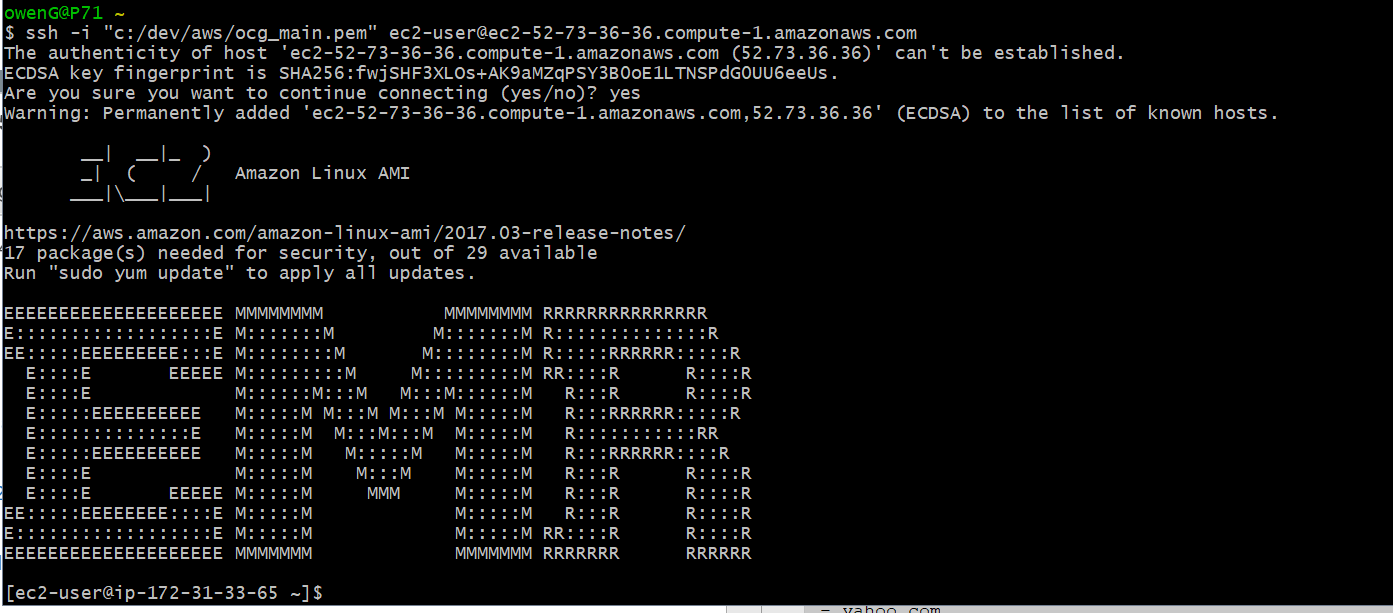
|  |
| --- |
|  |

(click on bottom link in above, the ID hyperlink for **CORE** and view the slave details)

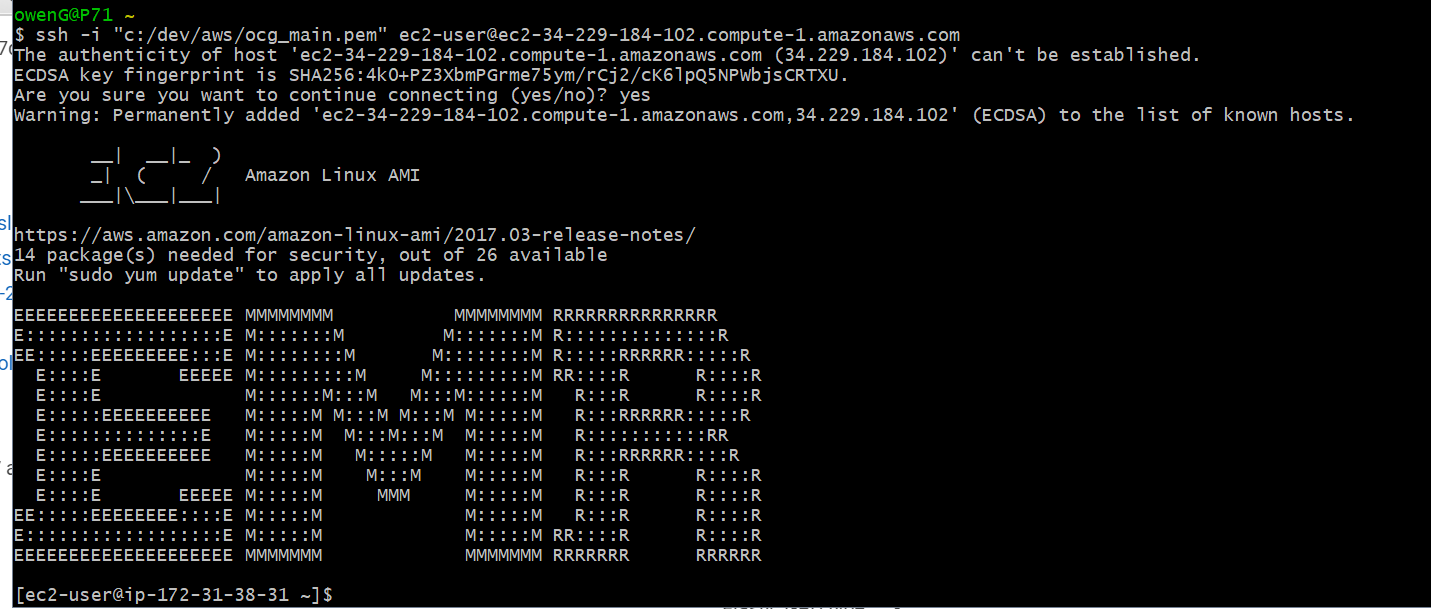
|  |
| --- |
|  |

ssh to the first slave and then the second (had initially thought ssh’ng to these was part of the assignment, no real reason to remove them from the .doc afterwards)

ssh -i "c:/dev/aws/ocg\_main.pem" ec2-user@ec2-52-73-36-36.compute-1.amazonaws.com



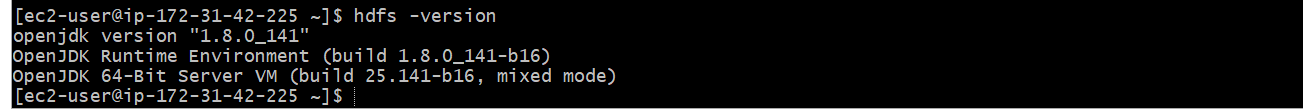
ssh -i "c:/dev/aws/ocg\_main.pem" ec2-user@ec2-34-229-184-102.compute-1.amazonaws.com



**ssh to your master node and verify version of HDFS by issuing a command like: "hdfs -version"**

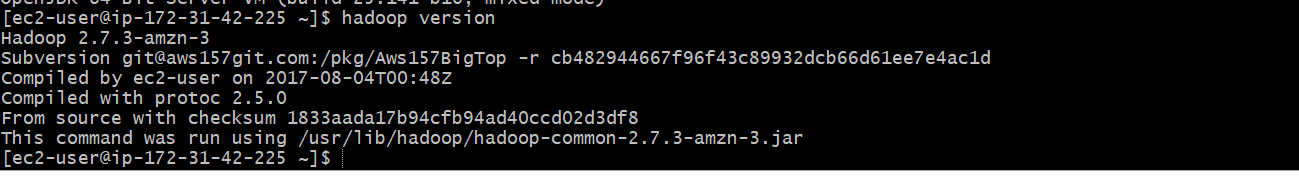
While still in ssh to master node above, check the version of hdfs

hdfs -version



Get Hadoop version for kicks

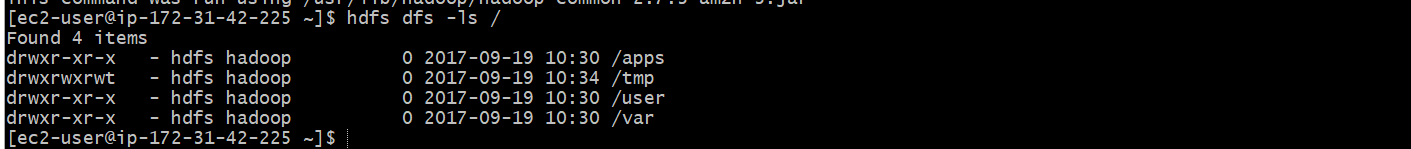
hadoop version



* **demonstrate basic file operations in HDFS (create, get, delete a file) - using HDFS commands documentation for reference**

Back to Master node, see what we have by default via an ls command

hdfs dfs -ls /



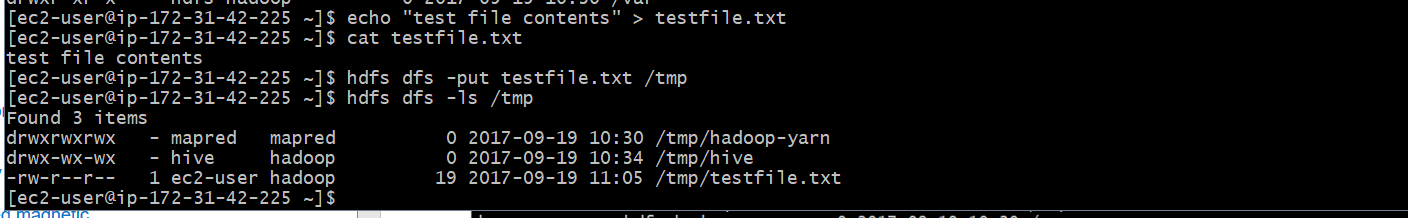
Create a local test file and insert some simple text, copy that into the /tmp directory in hdfs and then use ls again to confirm it made it in:

echo "test file contents" > testfile.txt

cat testfile.txt

hdfs dfs -put testfile.txt /tmp

hdfs dfs -ls /tmp

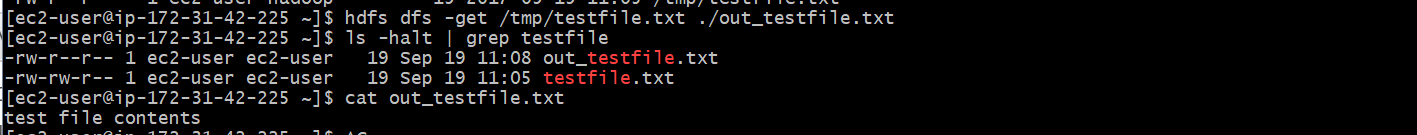


Now get that same file out again, only as a file named out\_testfile.txt, confirm success and then output contents to double-confirm:

hdfs dfs -get /tmp/testfile.txt ./out\_testfile.txt

ls -halt | grep testfile

cat out\_testfile.txt



Use the rm command to delete that file from hdfs, note confirmation from hdfs but do another ls in /tmp of hdfs to confirm.

hdfs dfs -rm /tmp/testfile.txt

hdfs dfs -ls /tmp

