GUID: 2322021C

SP Assessed Exercise 2a: Report

State:

This is a functioning version of the dependency discoverer. It outputs the correct results for the test folder but not always in the right order. Be sure to recheck the output when using the bash command diff and to sort the output. The order of the output sadly doesn't always come out right but it should be correct. This program has implemented multiple threads that can be changed by altering the environment variable *CRAWLER_THREADS***.** The cpath environment variable appears to be already implemented but has not been tested so this could be a potential issue with the program. The only thing that can be said about the program is that it produces the correct output for the test folder on any number of threads between 1-8.

This implementation uses binary semaphores and a custom thread tracker class. The binary semaphores were used to lock critical areas of the code. Even though the data structures were safe for threading, the order and coordination between the threads was not, so they had to be organised so that only one check the status of the queue and pop from it at a time. And when processing the file if the thread was going to edit the queue, other threads would have to wait before they could edit it as well. The tracker class simply alerts the main thread when the slave threads have finished computing. It does this by use of a thread safe counter and a condition variable - then once this tracker is cleared, the threads are free to join the main thread.

Sequential & 1-Thread Runtimes:

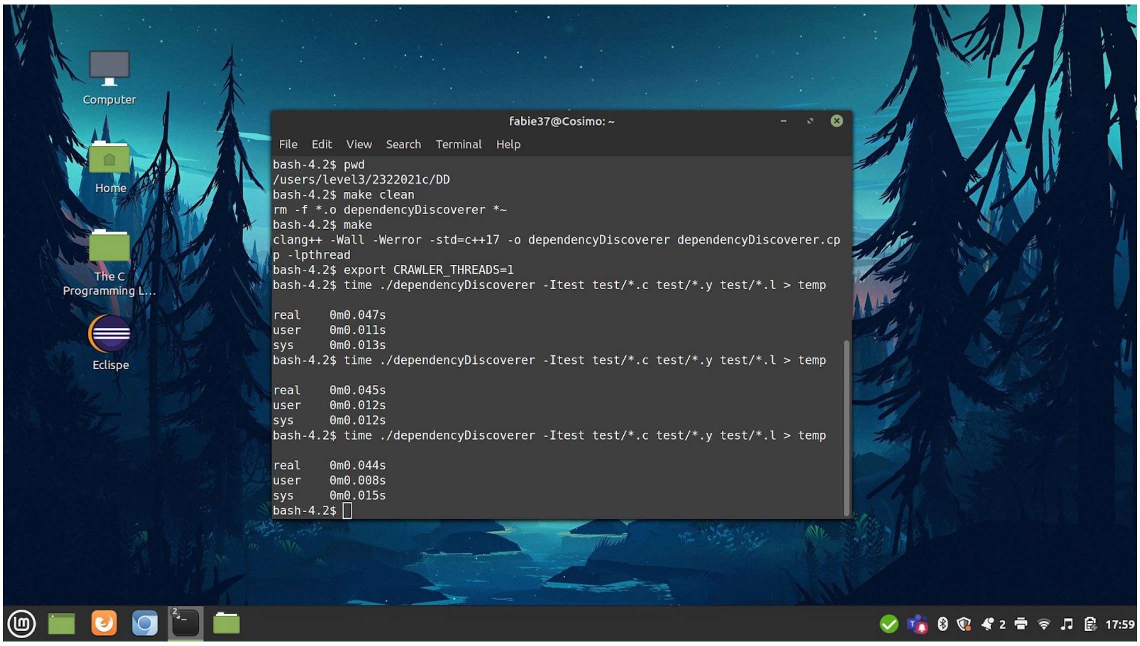
Here it is taken that the sequential file is just the crawler that was supplied in the resource folder and the 1 thread runtime is just the crawler with a single thread spawned.

Commands

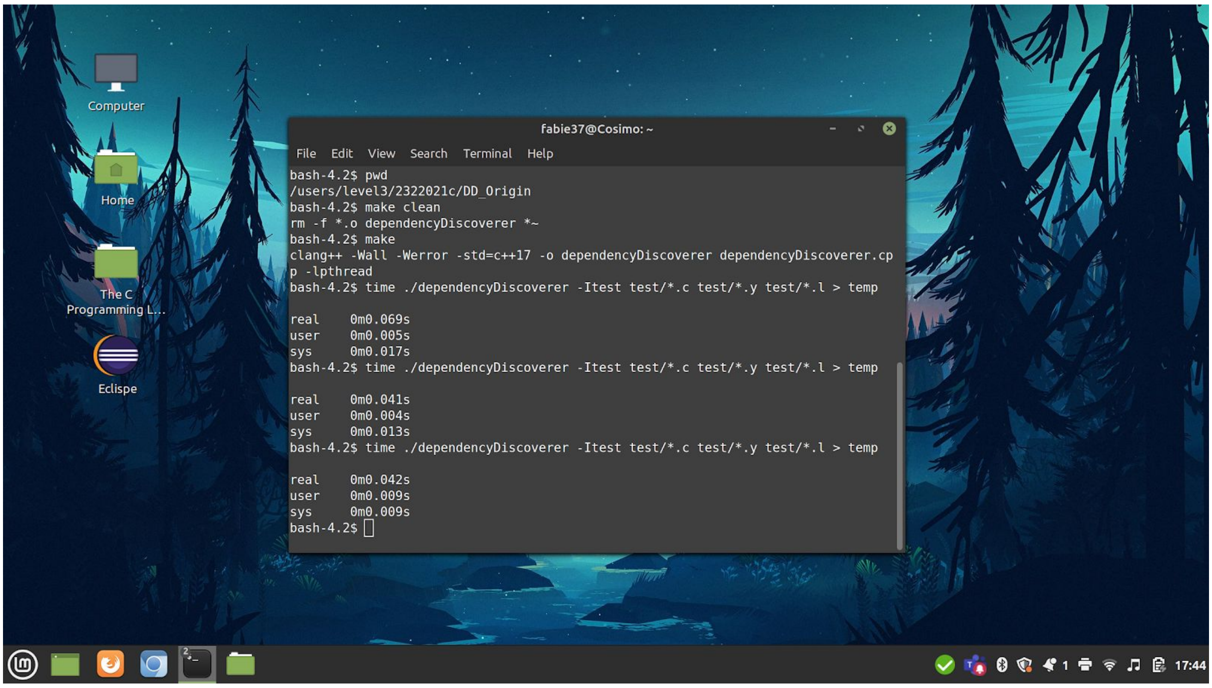
```
                            1-Thread Commands
~$ cd {dir_with_solution)
~$ export CRAWLER_THREADS=1
~$ make
~$ time ./dependencyDiscoverer -Itest test/*.c test/*.l test/*.y

                            Sequential Commands

~$ cd {dir_with_supplied_files)
~$ make
~$ time ./dependencyDiscoverer -Itest test/*.c test/*.l test/*.y
```

GUID: 2322021C

Screen Shots - 1 Thread



Screen Shots - Sequential



Multiple Runtime Analysis:

Because of the laborious task of repeatedly trying out different threads by use of the CLi, a bash script was created in order to simulate the exact same process on the school servers.

```
#!/bin/bash

threadNumb=8

echo "" > test_results.txt
for (( i=1; i <= $threadNumb; i++ ))
do
   export CRAWLER_THREADS=$i
   echo "Thread Count: $i" >> test_results.txt
   echo "Thread Count: $i"
   echo "" > $i.txt
   pwd
   for (( x=1; x<= 3; x++ ))
   do
       var1=$(  time (./dependencyDiscoverer -Itest test/*.c test/*.l
test/*.y 2>temp 1>&2 ) 2>&1)
       echo $var1 >> $i.txt
   done
   echo "$(cat $i.txt | sort -n)"
   echo "$(cat $i.txt | sort -n)" >> test_results.txt
   rm $i.txt
       rm temp
   echo
done
```

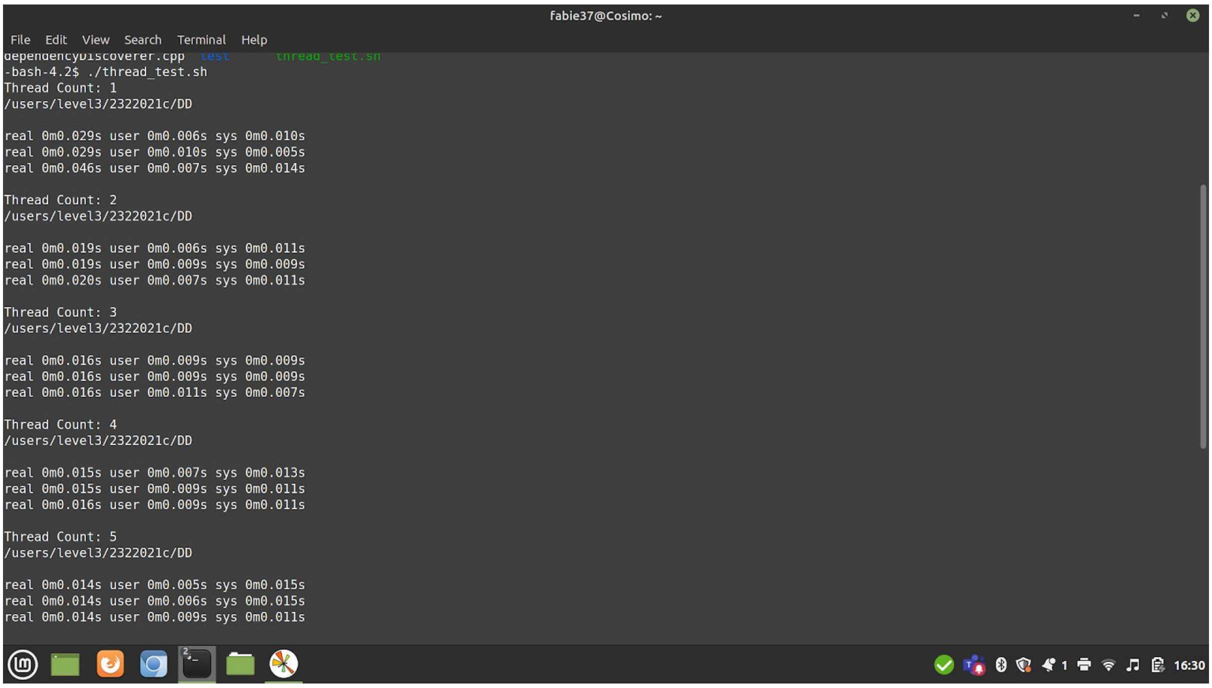
The script was saved onto a file called “*thread_test.sh*” and was placed into the same directory as the dependecyDiscoverer.cpp file, it’s runnable binary and the test folder.

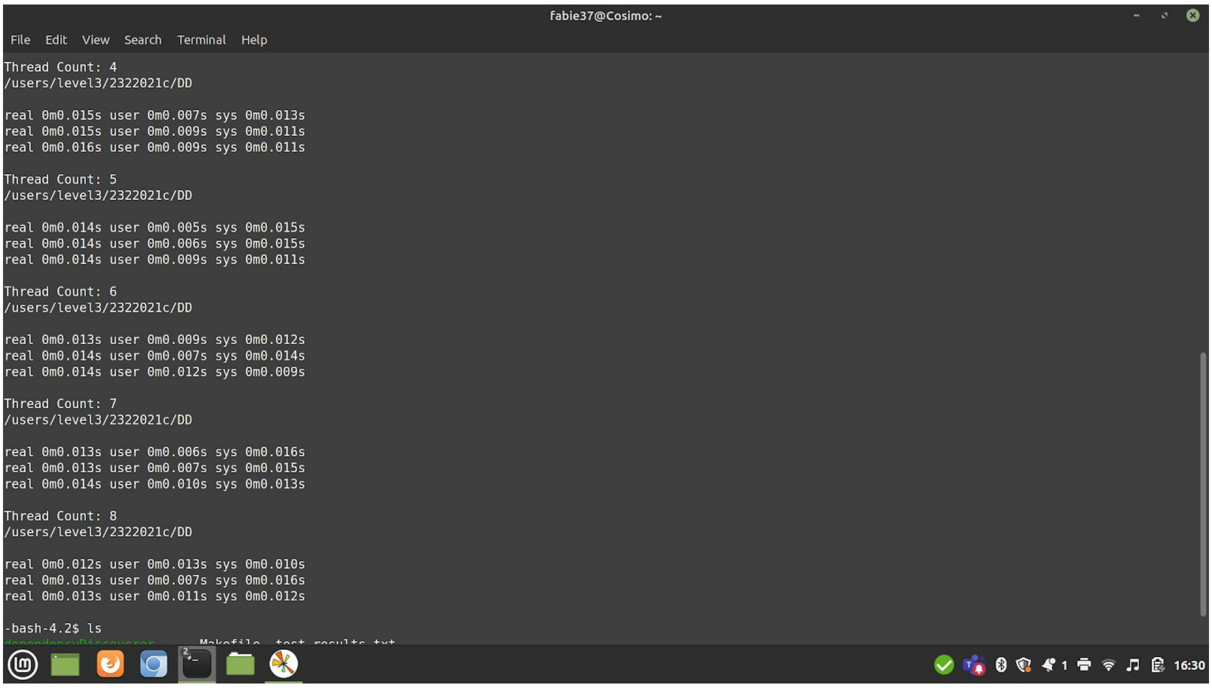
This script was run on the school servers alongside a python script to generate the results in the form of a line graph.

The script also helped identify potential deadlock situations by easily being able to execute multiple versions of the code program. An issue was identified in the process function which was solved by locking critical areas of code so that only one thread had access to it at a time. This was done by implementing a binary semaphore.

GUID: 2322021C

Screenshots - Multiple Threads





GUID: 2322021C

Results:

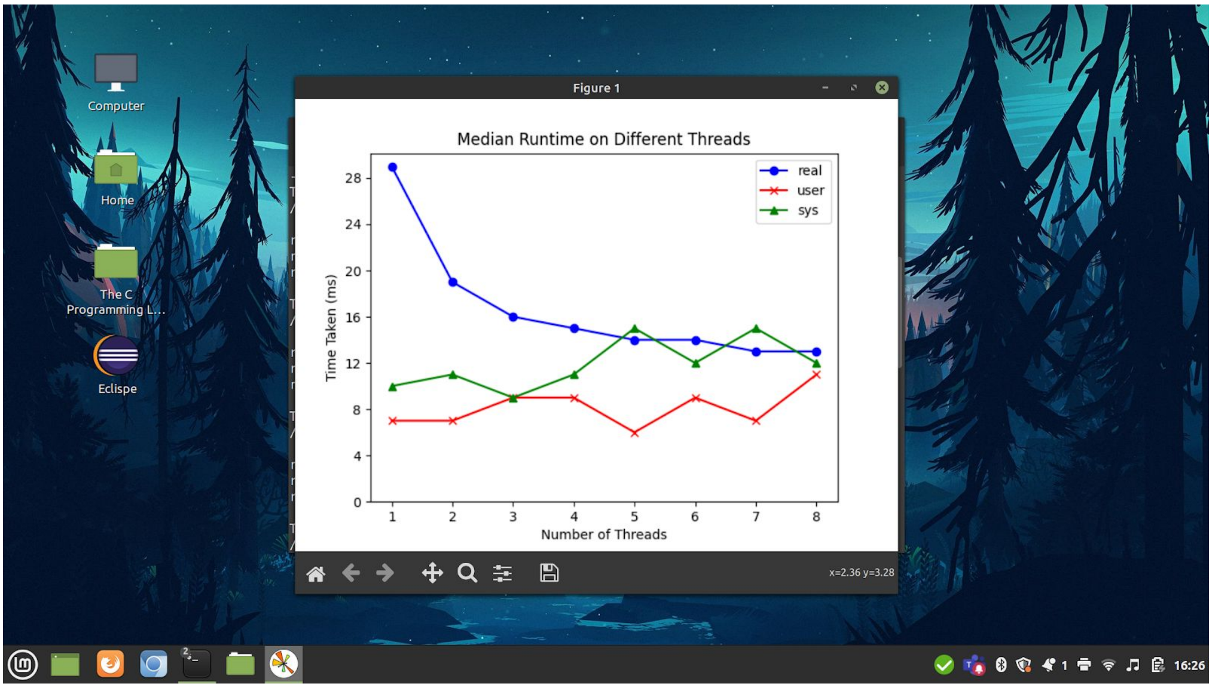
CRAWLER_THREADS	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7	Thread 8
	Elapsed Time Real (s)	Elapsed Time Real (s)	Elapsed Time Real (s)	Elapsed Time Real (s)	Elapsed Time Real (s)	Elapsed Time Real (s)	Elapsed Time Real (s)	Elapsed Time Real (s)
Exe 1	0.029	0.019	0.016	0.015	0.014	0.013	0.013	0.012
Exe 2	0.029	0.019	0.016	0.015	0.014	0.014	0.013	0.013
Exe 3	0.046	0.020	0.016	0.016	0.014	0.014	0.014	0.013
Median	0.029	0.019	0.016	0.015	0.014	0.014	0.013	0.013

CRAWLER_THREADS	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7	Thread 8
	Elapsed Time User (s)	Elapsed Time User (s)	Elapsed Time User (s)	Elapsed Time User (s)	Elapsed Time User (s)	Elapsed Time User (s)	Elapsed Time User (s)	Elapsed Time User (s)
Exe 1	0.006	0.006	0.009	0.007	0.005	0.009	0.006	0.013
Exe 2	0.010	0.009	0.009	0.009	0.006	0.007	0.007	0.007
Exe 3	0.007	0.007	0.011	0.009	0.009	0.012	0.010	0.011
Median	0.007	0.007	0.009	0.009	0.006	0.009	0.007	0.011

CRAWLER_THREADS	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7	Thread 8
	Elapsed Time Sys (s)	Elapsed Time Sys (s)	Elapsed Time Sys (s)	Elapsed Time Sys (s)	Elapsed Time Sys (s)	Elapsed Time Sys (s)	Elapsed Time Sys (s)	Elapsed Time Sys (s)
Exe 1	0.010	0.011	0.009	0.013	0.015	0.012	0.016	0.010
Exe 2	0.005	0.009	0.009	0.011	0.015	0.014	0.015	0.016
Exe 3	0.014	0.011	0.007	0.011	0.011	0.009	0.013	0.012
Median	0.010	0.011	0.09	0.011	0.015	0.012	0.015	0.012

GUID: 2322021C

Analysis:



As we can see from the above graph - the median time for the program to terminate if measured by a stop-watch (real time) decreased in a negative, logarithmic fashion! This suggests that increasing the thread count decreases the total execution time - to a point. After 5 threads or so, the returns in speed are minimal, slowly leveling out. There seems to be a caveat though. The sys time slowly increases with slight variations as we increase the thread count. This suggests that more time is spent by the CPU in the kernel to process the application than if it was run on the main thread. An even slower increase was with the user time although it draws a similar correlation to the sys time. What this implies is that as we increase the thread count - the total time for the code to finish decreases to a point before diminishing returns as the total time spent by the CPU to execute the process increases slightly.