GDB/Valgrind Debugging & OpenMP Debugging

CEO Tyler (GDB/Valgrind Debugging)

As ceo, I was responsible first for distributing files to my teammates. I had everyone download the tar file from the google slides and then follow the instructions below. First, I had everyone mount their home directory to their multipass ubuntu subsystem. This was done with the multipass mount $HOME <instance name> command. Then, I had my teammates start their multipass instance and then open the shell for the instance. Once they were in the shell, I had my teammates navigate to the directory where they extracted the tar file for p2-w1. There, they would have access to the gdb and omp directories for the week one project. An example of file distribution in multipass is shown below.

```
(base) Tylers-MBP-2:~ tylerburch$ multipass mount $HOME primary
 mount failed: The following errors occurred:
 "primary:/Users/tylerburch" is already mounted
(base) Tylers-MBP-2:~ tylerburch$ multipass start primary
(base) Tylers-MBP-2:~ tylerburch$ multipass shell primary
```

```
ubuntu@primary:~$ ls
 Home  icestorm-build  riscv-gnu-toolchain-rv32i  snap
ubuntu@primary:~$ cd Home/Desktop/p2-w1
ubuntu@primary:~/Home/Desktop/p2-w1$ ls
 gdb  omp
 ubuntu@primary:~/Home/Desktop/p2-w1$
```

Next, I was responsible for the initial running of the bugs binary file. To do this, I made sure that gcc, gdb, and valgrind were installed then I ran gdb on ./bugs.

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bugs...
(gdb) r
Starting program: /home/ubuntu/Home/Desktop/p2-w1/gdb/bugs
The current bug population of Earth is about: 1480000000000000000
The current bug population of Mars is about: 0
The current bug population of Venus is about: 0
The total bug population of the solar system is: 1480000000000000000
??UUUU ?UUUU UUUU UUU UU U mosquito
??UUUU ?UUUU UUUU UUU UU U butterfly
??UUUU ??UUUU p?UUUU   ?UUUU @?UUUU
                                  `UUUU
Program received signal SIGSEGV, Segmentation fault.
__strlen_avx2 () at   /sysdeps/x86_64/multiarch/strlen-avx2.S:65
```

After running, the results looked similar to this and I got a segmentation fault, or SIGSEGV signal. So, I backtraced the execution and zoomed into frame 3.

```
(gdb) bt
#0  __strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:65
#1  0x00007ffff7e49e95 in __vfprintf_internal (
    s=0x7ffff7fba6a0 <_IO_2_1_stdout_>, format=0x555555556008 "%s ",
    ap=ap@entry=0x7fffffffe210, mode_flags=mode_flags@entry=0)
    at vfprintf-internal.c:1688
#2  0x00007ffff7e32ebf in __printf (format=<optimized out>) at printf.c:33
#3  0x00005555555552de in echoohce (strs=0x7fffffffe340) at bugs.c:81
#4  0x0000555555555492 in main (argc=1, argv=0x7fffffffe498) at bugs.c:138
(gdb) f 3
#3  0x00005555555552de in echoohce (strs=0x7fffffffe340) at bugs.c:81
81          printf("%s ", *iter);
```

It was clear that something was wrong with the echoohce function and the echo function, which were supposed to be printing out components of the bug_info.sentence array. In other words, the functions iterate through a character pointer array or a string array and print out each string. This was clear from the function code, shown below.

```
Function "function" not defined.
(gdb) list echoohce
59        }
60      }
61
62      // Print a **NULL terminated** array of strings to standard out and then print the
63      // array backwards (This is broken).
64      void echoohce(char** strs) {
65        char** iter;
66        char** stop_beginning = strs - 1; // HINT: What could this possibly used for?
67
68        // FIXME: Something is wrong in this for loop. It is printing garbage characters.
(gdb) bt
```

PM Christine (GDB/Valgrind Debugging)

N/A

Dev0 Nadeen (GDB/Valgrind Debugging)

As Dev0, my role for this project was to first use GDB to debug the bugs program. One of the errors I was able to find and fix is increasing the length of the array which printed out many statements regarding the program. When running GDB, and going through the different frames, I was able to find that error. Before fixing it, the file looked like this:

```
struct Storage {
  intptr_t num_bugs_on_mars;
  const char* scary_bug;
  char* sentence[6]; // FIXME: It might be necessary to grow this array.
  const char* colorful_bug;
  intptr_t num_bugs_on_earth;
  intptr_t num_bugs_on_venus;
  char* useless_bug;
};
```

```
  // process exits or the pointers change.
  bug_info.sentence[0] = strdup("The");
  bug_info.sentence[1] = strdup("most");
  bug_info.sentence[2] = strdup("useless");
  bug_info.sentence[3] = strdup("bug");
  bug_info.sentence[4] = strdup("is");
  bug_info.sentence[5] = strdup("a");
  //FIXME: It may be necessary to add onto this structure.
```

To fix the error, I changed the array length to 7, and added NULL to that extra array space. Therefore, the code changed to look like this:

```
// and what is stored in it when looking at gdb and valgrind.
struct Storage {
  intptr_t num_bugs_on_mars;
  const char* scary_bug;
  char* sentence[7]; // FIXME: It might be necessary to grow this array.
  const char* colorful_bug;
  intptr_t num_bugs_on_earth;
  intptr_t num_bugs_on_venus;
  char* useless_bug;
};
```

```
  // process exits or the pointers change.
  bug_info.sentence[0] = strdup("The");
  bug_info.sentence[1] = strdup("most");
  bug_info.sentence[2] = strdup("useless");
  bug_info.sentence[3] = strdup("bug");
  bug_info.sentence[4] = strdup("is");
  bug_info.sentence[5] = strdup("a");
  bug_info.sentence[6] = NULL;
  //FIXME: It may be necessary to add onto this structure.
```

After fixing this error, and another dereferencing error which Alfie fixed, the program was successfully running and printing out the correct statements.

Dev1 Alfie (GDB/Valgrind Debugging)

Tester Ben (GDB/Valgrind Debugging)

For this portion of the project I was in charge of experimenting with valgrind and answering the following questions about my findings.

1) Run valgrind on valgrind_test using the --leak-check, --track-origins, and --show-leak-kinds flags.

a) What are these flags doing and what other flags exist?

i) --leak-check: Each individual leak will be shown in detail
ii) --track-origins: Tracks the origins of uninitialized values
iii) --show-leak-kinds: Shows all of "definite, indirect, possible, reachable" leak kinds
iv) --verbose: Gives extra information on various aspects of your program

b) What is the leak summary show?

```
ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 2/p2-w1/gdb$ valgrind ./valgrind_test
==2036== Memcheck, a memory error detector
==2036== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2036== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2036== Command: ./valgrind_test
==2036==
==2036== Conditional jump or move depends on uninitialised value(s)
==2036==    at 0x109181: main (valgrind_test.c:22)
==2036==
==2036==
==2036== HEAP SUMMARY:
==2036==     in use at exit: 1,087 bytes in 140 blocks
==2036==   total heap usage: 140 allocs, 0 frees, 1,087 bytes allocated
==2036==
==2036== LEAK SUMMARY:
==2036==    definitely lost: 552 bytes in 69 blocks
==2036==    indirectly lost: 483 bytes in 69 blocks
==2036==      possibly lost: 10 bytes in 1 blocks
==2036==    still reachable: 42 bytes in 1 blocks
==2036==         suppressed: 0 bytes in 0 blocks
==2036== Rerun with --leak-check=full to see details of leaked memory
==2036==
==2036== Use --track-origins=yes to see where uninitialised values come from
==2036== For lists of detected and suppressed errors, rerun with: -s
==2036== ERROR SUMMARY: 70 errors from 1 contexts (suppressed: 0 from 0)
```

The leak summary aboves shows the amount of bytes lost, indirectly lost, possibly lost, still reachable and suppressed and in how many blocks of memory.

c) What error(s) are thrown and where in *valgrind_test.c* are they (ignore heap summary for now)?

```
==2043== Conditional jump or move depends on uninitialised value(s)
==2043==    at 0x109181: main (valgrind_test.c:22)
```

```
int main() {
  int uninitialized_variable; // This variable is never given a value.

  for (; uninitialized_variable < 100; uninitialized_variable++) {
    void** definitely_lost = (void**) malloc(sizeof(void*)); // allocate a
                                                             // pointer on the
                                                             // heap.
```

The error indicates that there is an uninitialized variable in line 22 of the code as shown above.

2) Open up *valgrind_test.c* and fix this simple error with uninitialized_variable. Make and rerun
valgrind. What does the leak summary show now (one of the leaks should have been resolved to 0)?

a) What does each category mean (possibly lost, suppressed, etc.)?

```
[ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 2/p2-w1/gdb$ valgrind ./valgrind_test
==2060== Memcheck, a memory error detector
==2060== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2060== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2060== Command: ./valgrind_test
==2060==
==2060==
==2060== HEAP SUMMARY:
==2060==     in use at exit: 1,552 bytes in 202 blocks
==2060==   total heap usage: 202 allocs, 0 frees, 1,552 bytes allocated
==2060==
==2060== LEAK SUMMARY:
==2060==    definitely lost: 800 bytes in 100 blocks
==2060==    indirectly lost: 700 bytes in 100 blocks
==2060==      possibly lost: 10 bytes in 1 blocks
==2060==    still reachable: 42 bytes in 1 blocks
==2060==         suppressed: 0 bytes in 0 blocks
==2060== Rerun with --leak-check=full to see details of leaked memory
==2060==
==2060== For lists of detected and suppressed errors, rerun with: -s
==2060== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

I initialized the uninitialized variable to 0, but the leak summary is still saying there are leaks in the program. However, the Error Summary at the bottom of the visual says that there are 0 errors from 0 contexts.

Definitely lost means that no pointer to the block can be found. The block is classified as "lost", because the programmer could not possibly have freed it at program exit, since no pointer to it exists. This is likely a symptom of having lost the pointer at some earlier point in the program.

Indirectly lost means that the block is lost, not because there are no pointers to it, but rather because all the blocks that point to it are themselves lost.

Possibly lost means that a chain of one or more pointers to the block has been found, but at least one of the pointers is an interior-pointer.

Still reachable means that a start-pointer or chain of start-pointers to the block is found. Since the block is still pointed at, the programmer could, at least in principle, have freed it before program exit.

Suppressed errors are system specific, such as known problems with your libraries.

3) As the heap summary clearly depicts, we still have a few memory leaks. Read the given comments to understand the test program better. Determine all existing leaks and hypothesize the reasons behind the apparent leaks in *valgrind_test.c* (*it may not even be a problem...*).

a) One example of a memory leak lies with still_reachable (what should you always do with malloc?)
    i) You need to deallocate memory using the free method to reduce the wastage of memory. The memory allocated using functions malloc() and calloc() is not deallocated on their own..

```
possibly_lost = malloc(10);
```

Possibly lost: There is a pointer pointing to the middle of the allocated block but nothing points to the front of the block.

```
still_reachable = malloc(42);
```

Still reachable: The value is never freed but is pointed to in the global scope.

```
*definitely_lost = (void*) malloc(7);
```

Definitely lost: The definitely_lost variable gets out of scope because it is not pointing to anything on the heap, making it impossible for us to free it.

Indirectly lost: The pointer pointed to by definitely_lost is indirectly lost since we were only able to reach the pointer through definitely_lost.

4) Propose a few solutions to fix the memory leaks with your rationale.

Possibly lost: Assign a pointer that points to the front of the block.

Still reachable: Free the value using free()

Definitely lost/Indirectly lost: Point the definitely_lost variable to something in the heap so it doesn't fall out of scope.

CEO Tyler (Intro) [OpenMP Debugging]

In the OpenMP debugging section of p2-w1, our goal was to use gdb to debug a program that makes use of the OpenMP multithreading api. Using gdb on multithreaded programs is very useful because you can observe different threads and the local variables within those threads, a task that would be very difficult with print statements.

First, I handled the distribution of the files to my teammates by instructing them to navigate to the omp folder in multipass. This folder was installed when we distributed the p2-w1 tar file in the beginning.

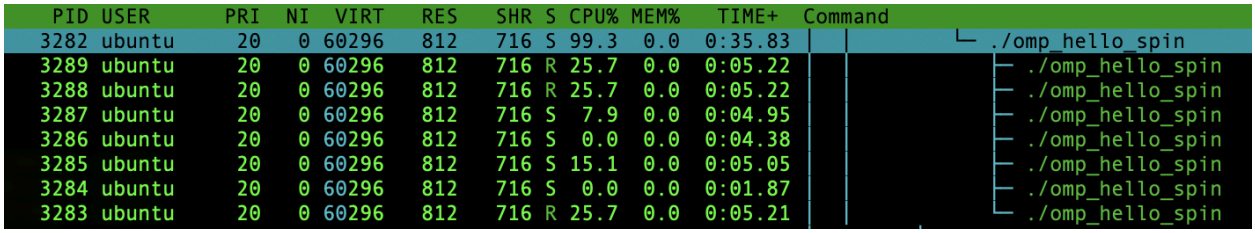
Starting with the files, I ran the omp_getEnvInfo program to see the environment variable values. Although we were supposed to see multiple threads running, the environment info stated that the max number of threads permitted by omp was 1. Thus, we needed to change this number to 8 in order to debug the omp_hello_spin files. To do this, we changed the OMP_NUM_THREADS omp variable to 8. This changed the max number of threads permitted by OpenMP to 8.

PM Christine (Plan) [OpenMP Debugging]

N/A

Dev0 Nadeen (Implementation) [OpenMP Debugging]

Another role I had for week 1 of this project was to use OpenMp and monitor HTOP when the threads for the omp_hello_spin file were running. To do so, I used some HTOP features to make it easier for me to trace these threads without having to sort through other threads running on my machine. Therefore, I filtered the processes available on HTOP by typing in Ubuntu. This limited all the threads available to only show me the ubuntu omp threads I wanted to see. Then, I used the tree functionality to see the threads in a tree sorted way. This helps in seeing how the threads relate to one another, and how one process acts as primary starter to the others. Below is how my HTOP looked after using filter and tree.



Now, it is clear to see that we have 8 threads running when our program is running throup omp. We can see that one thread started running before the others, but then all other 8 threads are running parallel to each other.

Dev1 Alfie (Implementation) [OpenMP Debugging]

Tester Ben (Results) [OpenMP Debugging]

As the tester, it was my responsibility to compare and record CPU times per thread to expected before debugging and after debugging. Here are the results from everyone's machines.

Note: I was unable to obtain screenshots of Christine's or Alfie's terminals as they were both having trouble using Multipass.

<u>CPU times before debugging:</u>

Tyler:

```
[ubuntu@primary:~/Home/Desktop/p2-w1/omp$ ./omp_hello_spin
 Thread 0, wait count 1809318008
 Thread 1, wait count  489950964
 Thread 2, wait count  626793282
 Thread 3, wait count 1144695449
 Thread 4, wait count  766968600
 Thread 5, wait count   22738458
 Thread 6, wait count  209328532
 Thread 7, wait count 1425293942
 Hello World from thread = 3.  I waited  489950964 counts
 Hello World from thread = 2.  I waited  489950964 counts
 Hello World from thread = 0.  I waited  489950964 counts
 Hello World from thread = 5.  I waited  489950964 counts
 Number of threads = 8
 Hello World from thread = 7.  I waited  489950964 counts
 Hello World from thread = 4.  I waited  489950964 counts
 Hello World from thread = 1.  I waited  489950964 counts
 Hello World from thread = 6.  I waited  489950964 counts
```

```
CPU[||||||||||||||||||||||100.0%]   Tasks: 39, 41 thr; 1 running
Mem[|||||||||||||||||189M/1.94G]   Load average: 4.03 1.71 0.64
Swp[                      0K/0K]   Uptime: 18:05:34

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
14058 ubuntu     20   0 60296   744   648 R 100.  0.0  0:10.25
14065 ubuntu     20   0 60296   744   648 R 12.6  0.0  0:01.28
14064 ubuntu     20   0 60296   744   648 R 12.6  0.0  0:01.28
14063 ubuntu     20   0 60296   744   648 R 12.6  0.0  0:01.28
14062 ubuntu     20   0 60296   744   648 R 12.6  0.0  0:01.28
14061 ubuntu     20   0 60296   744   648 R 12.6  0.0  0:01.28
14060 ubuntu     20   0 60296   744   648 R 12.6  0.0  0:01.28
14059 ubuntu     20   0 60296   744   648 R 11.9  0.0  0:01.27
```

Christine: N/A

Nadeen:

Alfie: N/A

Ben:

```
ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 2/p2-w1/omp$ ./omp_hello_spin
Thread 0, wait count  634253933
Thread 1, wait count 1281375015
Thread 2, wait count 1981443288
Thread 3, wait count  162635592
Thread 4, wait count 1192351502
Thread 5, wait count   15493649
Thread 6, wait count 1844997260
Thread 7, wait count 1990470608
Hello World from thread = 1.  I waited 1281375015 counts
Hello World from thread = 7.  I waited 1281375015 counts
Hello World from thread = 2.  I waited 1281375015 counts
Hello World from thread = 3.  I waited 1281375015 counts
Hello World from thread = 5.  I waited 1281375015 counts
Hello World from thread = 6.  I waited 1281375015 counts
Hello World from thread = 4.  I waited 1281375015 counts
Hello World from thread = 0.  I waited 1281375015 counts
Number of threads = 8
```

```
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 2345 ubuntu     20   0 60296   752   656 R 99.4  0.0  0:33.59 ./omp_hello_spin
 2347 ubuntu     20   0 60296   752   656 R 12.9  0.0  0:04.19 ./omp_hello_spin
 2346 ubuntu     20   0 60296   752   656 R 12.9  0.0  0:04.19 ./omp_hello_spin
 2351 ubuntu     20   0 60296   752   656 R 12.9  0.0  0:04.20 ./omp_hello_spin
 2348 ubuntu     20   0 60296   752   656 R 12.3  0.0  0:04.19 ./omp_hello_spin
 2352 ubuntu     20   0 60296   752   656 R 12.3  0.0  0:04.19 ./omp_hello_spin
 2350 ubuntu     20   0 60296   752   656 R 12.3  0.0  0:04.19 ./omp_hello_spin
 2349 ubuntu     20   0 60296   752   656 R 12.3  0.0  0:04.19 ./omp_hello_spin
```

From the screenshots listed above we can see that the wait counts of each thread are the same per everyone's machine. This correlates to the output we see on HTOP where the CPU time for each thread is the exact same. This was expected before debugging.

<u>CPU times after debugging:</u>

Tyler:

```
ubuntu@primary:~/Home/Desktop/p2-w1/omp$ ./omp_hello_spin
Thread 0, wait count  528165657
Thread 1, wait count  695308150
Thread 2, wait count 1555881297
Thread 3, wait count  809688505
Thread 4, wait count  740989633
Thread 5, wait count  452041231
Thread 6, wait count   53122379
Thread 7, wait count 1152347556
Hello World from thread = 5.  I waited  695308150 counts
Hello World from thread = 4.  I waited  695308150 counts
Hello World from thread = 7.  I waited  695308150 counts
Hello World from thread = 2.  I waited  695308150 counts
Hello World from thread = 0.  I waited  695308150 counts
Number of threads = 8
Hello World from thread = 3.  I waited  695308150 counts
Hello World from thread = 6.  I waited  695308150 counts
Hello World from thread = 1.  I waited  695308150 counts
```

```
  CPU[||||||||||||||||||||||||100.0%]   Tasks: 39, 41 thr; 1 running
  Mem[|||||||||||||||||||186M/1.94G]   Load average: 1.79 0.40 0.20
  Swp[                      0K/0K]   Uptime: 18:34:11

    PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  14073 ubuntu     20   0 60296   880   784 R 100.  0.0  0:14.02
  14080 ubuntu     20   0 60296   880   784 R 11.8  0.0  0:01.74
  14079 ubuntu     20   0 60296   880   784 R 12.5  0.0  0:01.75
  14078 ubuntu     20   0 60296   880   784 R 12.5  0.0  0:01.75
  14077 ubuntu     20   0 60296   880   784 R 12.5  0.0  0:01.75
  14076 ubuntu     20   0 60296   880   784 R 12.5  0.0  0:01.75
  14075 ubuntu     20   0 60296   880   784 R 12.5  0.0  0:01.75
  14074 ubuntu     20   0 60296   880   784 R 12.5  0.0  0:01.75
```

Christine: N/A

Nadeen:

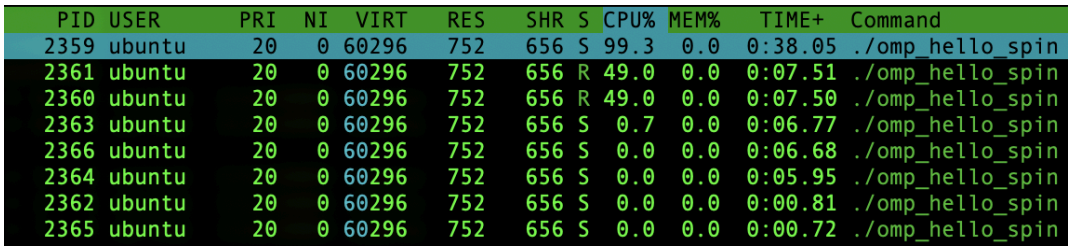
```
[ubuntu@p2w1:/Users/nabdulkarim/p2-w1/omp$ ./omp_hello_spin
 Thread 0, wait count  689626244
 Thread 1, wait count   93764143
 Thread 2, wait count 1938276282
 Thread 3, wait count 1510222503
 Thread 4, wait count  716753330
 Thread 5, wait count  196187572
 Thread 6, wait count  515415379
 Thread 7, wait count   92700681
 Hello World from thread = 7.  I waited   92700681 counts
 Hello World from thread = 1.  I waited   93764143 counts
 Hello World from thread = 5.  I waited  196187572 counts
 Hello World from thread = 6.  I waited  515415379 counts
 Hello World from thread = 0.  I waited  689626244 counts
 Number of threads = 8
 Hello World from thread = 4.  I waited  716753330 counts
 Hello World from thread = 3.  I waited 1510222503 counts
 Hello World from thread = 2.  I waited 1938276282 counts
```

```
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 9248 ubuntu     20   0 60296   804   708 R 99.4  0.1  0:08.87 ./omp_hello_spin
 9250 ubuntu     20   0 60296   804   708 R 12.7  0.1  0:01.11 ./omp_hello_spin
 9251 ubuntu     20   0 60296   804   708 R 12.7  0.1  0:01.11 ./omp_hello_spin
 9249 ubuntu     20   0 60296   804   708 R 12.0  0.1  0:01.10 ./omp_hello_spin
 9253 ubuntu     20   0 60296   804   708 R 12.0  0.1  0:01.10 ./omp_hello_spin
 9252 ubuntu     20   0 60296   804   708 R 12.0  0.1  0:01.10 ./omp_hello_spin
 9254 ubuntu     20   0 60296   804   708 R 12.0  0.1  0:01.10 ./omp_hello_spin
 9255 ubuntu     20   0 60296   804   708 R 12.0  0.1  0:01.10 ./omp_hello_spin
```

Alfie: N/A

Ben:

```
ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 2/p2-w1/omp$ ./omp_hello_spin
Thread 0, wait count  420994081
Thread 1, wait count 1859359415
Thread 2, wait count 1857741207
Thread 3, wait count  153099705
Thread 4, wait count 1482100259
Thread 5, wait count 1288371448
Thread 6, wait count  141470806
Thread 7, wait count 1485088464
Hello World from thread = 6.  I waited  141470806 counts
Hello World from thread = 3.  I waited  153099705 counts
Hello World from thread = 0.  I waited  420994081 counts
Number of threads = 8
Hello World from thread = 5.  I waited 1288371448 counts
Hello World from thread = 7.  I waited 1485088464 counts
Hello World from thread = 4.  I waited 1482100259 counts
Hello World from thread = 2.  I waited 1857741207 counts
Hello World from thread = 1.  I waited 1859359415 counts
```



PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
2359	ubuntu	20	0	60296	752	656	S	99.3	0.0	0:38.05	./omp_hello_spin
2361	ubuntu	20	0	60296	752	656	R	49.0	0.0	0:07.51	./omp_hello_spin
2360	ubuntu	20	0	60296	752	656	R	49.0	0.0	0:07.50	./omp_hello_spin
2363	ubuntu	20	0	60296	752	656	S	0.7	0.0	0:06.77	./omp_hello_spin
2366	ubuntu	20	0	60296	752	656	S	0.0	0.0	0:06.68	./omp_hello_spin
2364	ubuntu	20	0	60296	752	656	S	0.0	0.0	0:05.95	./omp_hello_spin
2362	ubuntu	20	0	60296	752	656	S	0.0	0.0	0:00.81	./omp_hello_spin
2365	ubuntu	20	0	60296	752	656	S	0.0	0.0	0:00.72	./omp_hello_spin

After debugging we can see that the wait counts for each thread on everyone's respective machines is different after tweaking the 'omp_hello_spin.c' file Again this correlates with output as the CPU time for each PID (thread) is different as it should be. We can even match each specific PID number to the thread it correlates to. I will use mine for demonstration.

We can see that 'thread 6' in my screenshot has a wait count of 141470806, the smallest wait count of the threads, which is why it finishes first. If we then observe HTOP we can see that PID 2365 has the smallest CPU time of 0.72 seconds. We can therefore deduce that PID 2364 correlates to 'thread 6'. Let's now look at 'thread 3' which was the second to smallest wait count of 153099705. Again, following the same process we did for 'thread 6', we see that PID 2362 had the second to smallest CPU time of 0.81, confirming that PID 2362 is 'thread 3'. We can follow this process with all the other threads to see how CPU time correlates to the wait count of each thread when we run the program.

CEO Tyler (Conclusion) [OpenMP Debugging]

As a result of this project, we were able to learn the value in utilizing GDB and understand when and how to use its commands to quickly debug simple programs and complex multithreaded programs. We also learned how to install and use valgrind to see memory leaks and to correct memory management errors.