OMP Threads with Shared Resources

CEO Tyler (Intro)

For week 2, we continue using GDB to debug multithreaded programs. This time, we debug and analyze a variant of the dining philosophers problem.

The dining philosophers problem was a situation proposed by E.W. Dijkstra as a way to describe deadlock in parallelism. The idea starts with 5 philosophers sitting at a round table. Each philosopher has one fork to their right and a bowl of food in front of them. When a philosopher isn't eating with two forks, one in the left and right hands, the philosopher thinks. When the philosophers sit down they all want to eat, but they each need two forks to do so. The philosophers could each graph the fork to their right, but then the fork to their left will have been taken by the philosopher to the left. Thus, each philosopher will have one fork, but none will ever have a second fork, causing deadlock. The goal for this problem is to come up with a method that avoids this deadlock and allows each philosopher to avoid starvation and eventually eat from the bowl in front of them.

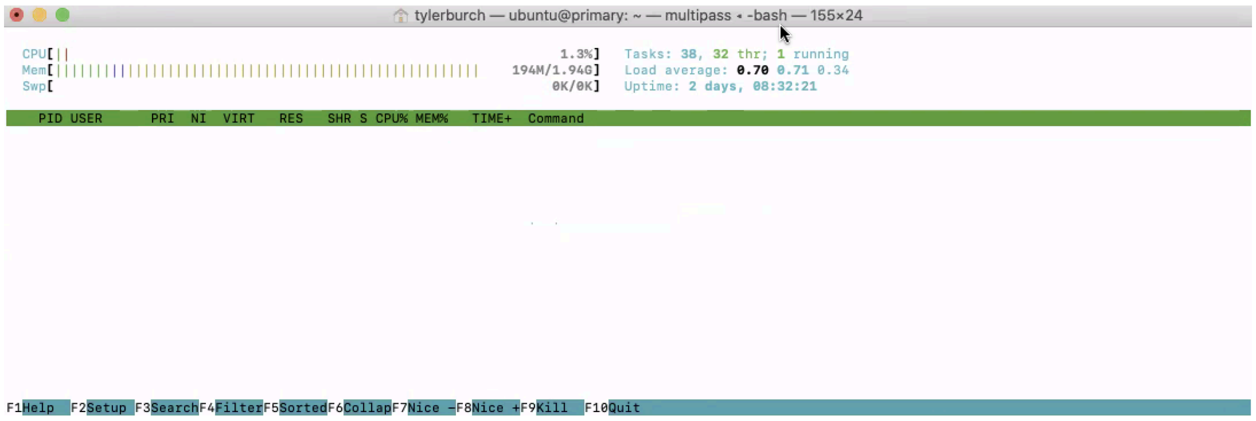
Our variant of this problem is the bodysurfers situation. Each bodysurfer starts with one fin and can attempt to access a locker where they can store their fin or take out two fins to use while bodysurfing. Our goal is to debug the bodysurfers_starter.c file and offer a solution to this situation that avoids deadlock.

First, like in week one, we need to distribute files to each of the teammates. We follow similar instructions as week one, but we download the files for week 2 and navigate to the p2-w2 folder in multipass.

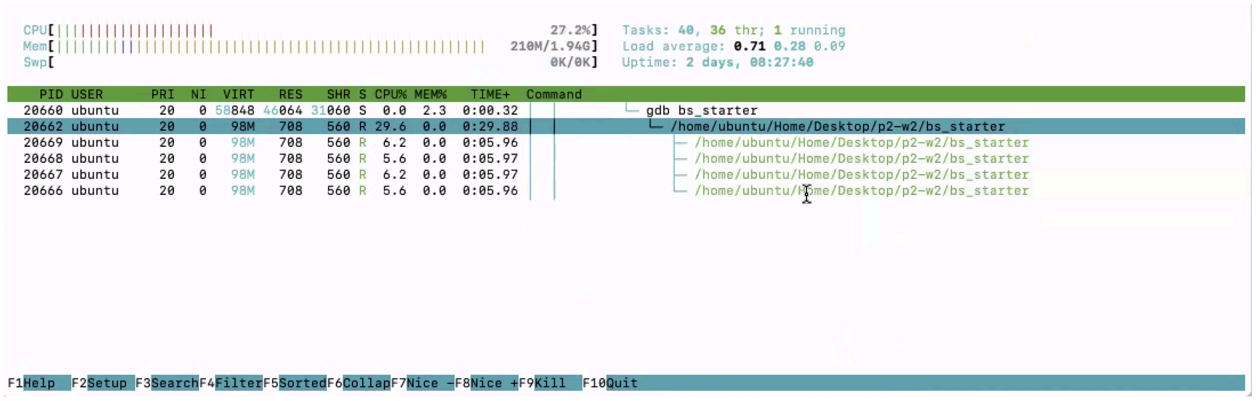
PM Christine (Plan)

Dev0 Nadeen (Implementation)

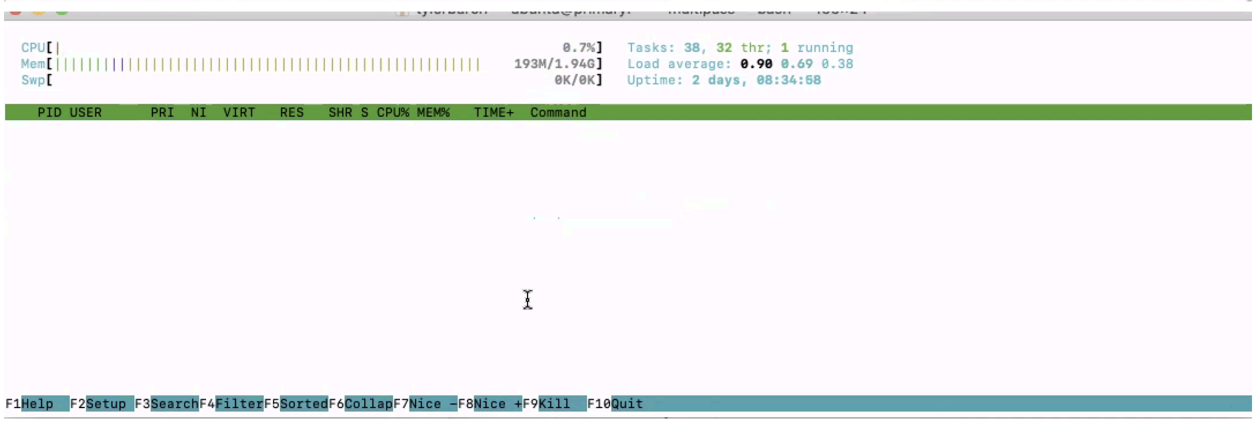
For week 2, aside from helping my teammates debug the code for the surfer's philosophers program, I was responsible for monitoring HTOP to see what the threads looked like before we ran the program, during, and after. I was also responsible for monitoring HTOP and how the threads looked different without the use of GDB. I used the same features I discussed in my part earlier above to filter out the running processes on my machine and view them in a tree form. Below is how my HTOP looked before during and after running the program (with the use of GDB) :



```
tylerburch — ubuntu@primary: ~ — multipass ◂ -bash — 155×24

  CPU[||                                                   1.3%]   Tasks: 38, 32 thr; 1 running
  Mem[||||||||||||||||||||||||||||||||||||||||||||||||| 194M/1.94G]   Load average: 0.70 0.71 0.34
  Swp[                                                    0K/0K]   Uptime: 2 days, 08:32:21

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command

F1Help  F2Setup F3SearchF4FilterF5SortedF6CollapF7Nice -F8Nice +F9Kill  F10Quit
```

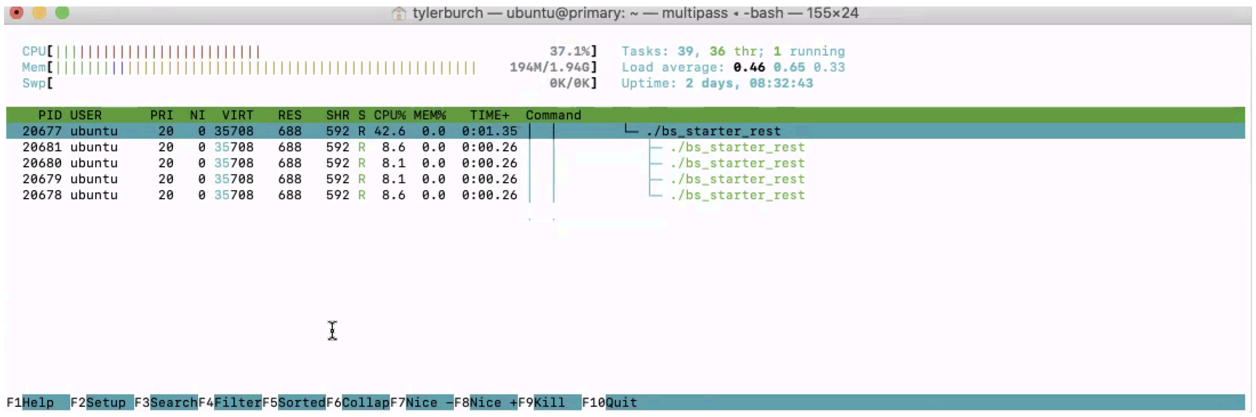


```
  CPU[||||||||||||||||||||                                27.2%]   Tasks: 40, 36 thr; 1 running
  Mem[||||||||||||||||||||||||||||||||||||||||||||||||||| 210M/1.94G]   Load average: 0.71 0.28 0.09
  Swp[                                                    0K/0K]   Uptime: 2 days, 08:27:40

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  20660 ubuntu      20   0 58848 46064 31060 S  0.0  2.3  0:00.32            └─ gdb bs_starter
  20662 ubuntu      20   0   98M   708   560 R 29.6  0.0  0:29.88               └─ /home/ubuntu/Home/Desktop/p2-w2/bs_starter
  20669 ubuntu      20   0   98M   708   560 R  6.2  0.0  0:05.96                  ├─ /home/ubuntu/Home/Desktop/p2-w2/bs_starter
  20668 ubuntu      20   0   98M   708   560 R  5.6  0.0  0:05.97                  ├─ /home/ubuntu/Home/Desktop/p2-w2/bs_starter
  20667 ubuntu      20   0   98M   708   560 R  6.2  0.0  0:05.97                  ├─ /home/ubuntu/Home/Desktop/p2-w2/bs_starter
  20666 ubuntu      20   0   98M   708   560 R  5.6  0.0  0:05.96                  └─ /home/ubuntu/Home/Desktop/p2-w2/bs_starter

F1Help  F2Setup F3SearchF4FilterF5SortedF6CollapF7Nice -F8Nice +F9Kill  F10Quit
```



```
  CPU[|                                                    0.7%]   Tasks: 38, 32 thr; 1 running
  Mem[||||||||||||||||||||||||||||||||||||||||||||||||| 193M/1.94G]   Load average: 0.90 0.69 0.38
  Swp[                                                    0K/0K]   Uptime: 2 days, 08:34:58

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command

F1Help  F2Setup F3SearchF4FilterF5SortedF6CollapF7Nice -F8Nice +F9Kill  F10Quit
```

It is obvious that before and after the program is finished running, the threads are not visible anymore on HTOP. The reason for that would be that the processes are not in the running state. Therefore, they are only visible while the threads are running.
Below is how HTOP looked without the use of GDB:



```
tylerburch — ubuntu@primary: ~ — multipass ◂ -bash — 155×24

CPU[||||||||||||||||||||||||||                 37.1%]   Tasks: 39, 36 thr; 1 running
Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||  194M/1.94G]   Load average: 0.46 0.65 0.33
Swp[                                             0K/0K]   Uptime: 2 days, 08:32:43

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
20677 ubuntu     20   0 35708   688   592 R 42.6  0.0  0:01.35 │  │       └─ ./bs_starter_rest
20681 ubuntu     20   0 35708   688   592 R  8.6  0.0  0:00.26 │  │          ├─ ./bs_starter_rest
20680 ubuntu     20   0 35708   688   592 R  8.1  0.0  0:00.26 │  │          ├─ ./bs_starter_rest
20679 ubuntu     20   0 35708   688   592 R  8.1  0.0  0:00.26 │  │          ├─ ./bs_starter_rest
20678 ubuntu     20   0 35708   688   592 R  8.6  0.0  0:00.26 │  │          └─ ./bs_starter_rest

F1Help  F2Setup F3SearchF4FilterF5SortedF6CollapF7Nice -F8Nice +F9Kill  F10Quit
```

The difference between using GDB and not using it, is seen here. We lose the first starter thread named "gdb_bs_starter". However the threads remain running in a parallel manner with one of them starting them off as a primary thread.

Dev1 Alfie (Implementation)

Tester Ben (Results)

As the tester, it was my job to experiment with the different wait times in the code and how that would affect the time it would take for all the threads to run to completion. I had to record the results from everyone's machines.

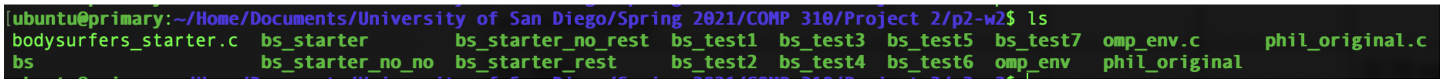
The experiment:

- with/without wait (sleep) after locker check
- with/without wait after surfing
- with/without wait while surfing

As you can see I have to experiment with the wait times after the locker check, after surfing, and while surfing. In total there are 8 different combinations and the specifics of each combination is listed in the 'Test' column of the table below. I performed these experiments by simply commenting out the wait times in the 'bodysurfers_starter.c' file in respect to each particular combination and compiling that file into an executable (a total of 8 executables were generated). I had the entire team, including myself run each executable on our own personal machines and list the elapsed times of each test.

Note: Again, I was unable to acquire the times from Christine's or Alfie's machines due them having trouble with Multipass.

Executables for each test:



```
ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 2/p2-w2$ ls
bodysurfers_starter.c  bs_starter         bs_starter_no_rest  bs_test1  bs_test3  bs_test5  bs_test7  omp_env.c       phil_original.c
bs                     bs_starter_no_no   bs_starter_rest     bs_test2  bs_test4  bs_test6  omp_env   phil_original
```

Elapsed time per team member:

Test	Tyler	Christine	Nadeen	Alfie	Ben
bs wait after lock check wait after surfing wait while surfing		N/A	54.12 seconds	N/A	56.35 seconds
bs_test1 **no** wait after lock check wait after surfing wait while surfing		N/A	820.11 seconds	N/A	851.24 seconds
bs_test2 wait after lock check **no** wait after surfing wait while surfing		N/A	46.7 seconds	N/A	47.99 seconds
bs_test3 wait after lock check wait after surfing **no** wait while surfing		N/A	50.1 seconds	N/A	52.59 seconds
bs_test4 **no** wait after lock check **no** wait after surfing wait while surfing		N/A	33.9 seconds	N/A	34.78 seconds
bs_test5 **no** wait after lock check wait after surfing **no** wait while surfing		N/A	22.8 seconds	N/A	23.95 seconds
bs_test6 wait after lock check **no** wait after surfing **no** wait while surfing		N/A	20.1 seconds	N/A	21.35 seconds
bs_test7 **no** wait after lock check **no** wait after surfing **no** wait while surfing		N/A	0.04 seconds	N/A	0.04 seconds

CEO Tyler (Conclusion)

Overall, as a team we were able to utilize GDB again to debug a multithreaded program. This time, we learned how to debug a program that was stuck in a deadlocked loop by using control c. Then we analyzed the multiple threads in a similar way as in week 1 by using the t, bt, and f commands. We were also able to analyze execution times to find the best method for executing the bodysurfing philosophers problem.