Computer Systems and Networks

Prof. Forsyth

Project 4 - Process Scheduling

Due: **July 10^{th} 2023**

1 Overview

In this project, you will implement a multiprocessor operating system simulator using a popular threading library for linux called pthreads. The framework for the multiprocessor OS simulator is nearly complete, but missing one critical component: the process scheduler! Your task is to implement the process scheduler and three different scheduling algorithms.

The simulated operating system supports only one thread per process making it similar to the systems that we discussed in Chapter 6. However, the simulator itself will use a thread to represent each of the CPUs in the simulated hardware. This means that the CPUs in the simulator will appear to operate concurrently.

Note: Make sure that multiple CPU cores are enabled in your virtual machine, otherwise you will receive incorrect results. See the TAs if you need help.

If you are using the CS 2200 VirtualBox or the CS 2200 Vagrant box, the number of cores should default to 2. In Vagrant, you can run `nproc --all` to see how many cores are available to your VM.

We have provided you with source files that constitute the framework for your simulator. You will only need to modify `answers.txt` and `student.c`. However, just because you are only modifying two files doesn't mean that you should ignore the other ones - there is helpful information in the other files. We have provided you these files:

1. `Makefile` - Working one provided for you; do not modify.
2. `os-sim.c` - Code for the operating system simulator which calls your CPU scheduler.
3. `os-sim.h` - Header file for the simulator.
4. `process.c` - Descriptions of the simulated processes.
5. `process.h` - Header file for the process data.
6. `student.c` - This file contains stub functions for your CPU scheduler.
7. `student.h` - Header file for your code to interface with the OS simulator. Also contains ready queue struct definition.

Reminder: The only files that you need to edit are `student.c` and `answers.txt`. If you edit any other files, your code may fail the autograder!

1.1 Scheduling Algorithms

For your simulator, you will implement the following three CPU scheduling algorithms:

1. **First Come, First Serve (FCFS)** - Runnable processes are kept in a ready queue. FCFS is non-preemptive; once a process begins running on a CPU, it will continue running until it either completes or blocks for I/O.
2. **Priority (PS)** - Each process has a priority associated with it. Processes with higher priority will be scheduled first. Processes with the same priority will be scheduled in the same manner as FCFS.
3. **Round-Robin** - Similar to FCFS, except preemptive. Each process is assigned a timeslice when it is scheduled. At the end of the timeslice, if the process is still running, the process is preempted, and moved to the tail of the ready queue.

1.2 Process States

In our OS simulation, there are five possible states for a process, which are listed in the `process_state_t` enum in os-sim.h:

1. `NEW` - The process is being created, and has not yet begun executing.
2. `READY` - The process is ready to execute, and is waiting to be scheduled on a CPU.

3. `RUNNING` - The process is currently executing on a CPU.
4. `WAITING` - The process has temporarily stopped executing, and is waiting on an I/O request to complete.
5. `TERMINATED` - The process has completed.

There is a field named `state` in the PCB, which must be updated with the current state of the process. The simulator will use this field to collect statistics.

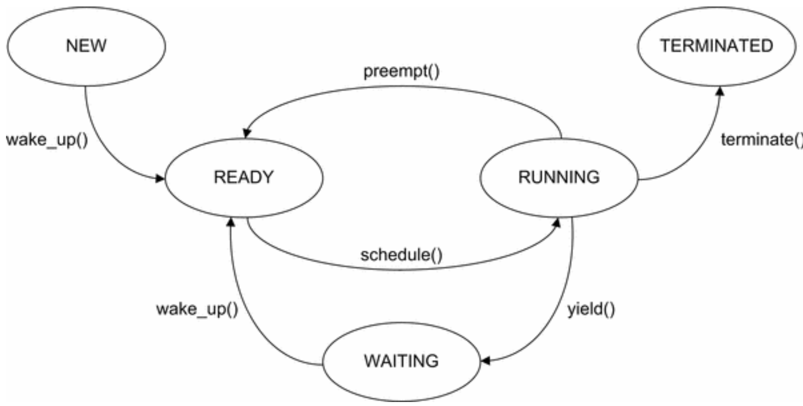

Figure 1: Process States

1.3 The Ready Queue

On most systems, there are a large number of processes, but only one or two CPUs on which to execute them. When there are more processes ready to execute than CPUs, processes must wait in the `READY` state until a CPU becomes available. To keep track of the processes waiting to execute, we keep a ready queue of the processes in the `READY` state.

Since the ready queue is accessed by multiple processors, which may add and remove processes from the ready queue, the ready queue must be protected by some form of synchronization–for this project, you will use a mutex lock that we have provided called `ready_mutex`.

1.4 Scheduling Processes

`schedule()` is the core function of the CPU scheduler. It is invoked whenever a CPU becomes available for running a process. `schedule()` must search the ready queue, select a runnable process, and call the `context_switch()` function to switch the process onto the CPU.

Note that in a multiprocessor environment, we cannot mandate that the currently running process be at the head of the ready q. There is an array (one entry for each cpu) that will hold the pointer to the PCB currently running on that cpu.

There is a special process, the idle process, which is scheduled whenever there are no processes in the `READY` state. This process simply waits for something new to be added to the ready queue and then calls `schedule()`.

1.5 CPU Scheduler Invocation

There are five events which will cause the simulator to invoke `schedule()`:

1. `yield()` - A process completes its CPU operations and yields the processor to perform an I/O request.
2. `wake_up()` - A process that previously yielded completes its I/O request, and is ready to perform CPU operations. `wake_up()` is also called when a process in the NEW state becomes runnable.

3. `preempt()` - When using a Round-Robin scheduling algorithm, a CPU-bound process may be preempted before it completes its CPU operations.
4. `terminate()` - A process exits or is killed.
5. `idle()` - Waits for a new process to be added to the ready queue (details below).

The CPU scheduler also contains one other important function: `idle()`. `idle()` contains the code that gets by the idle process. In the real world, the idle process puts the processor in a low-power mode and waits. For our OS simulation, you will use a pthread condition variable to block the thread until a process enters the ready queue.

1.6 The Simulator

We will use pthreads to simulate an operating system on a multiprocessor computer. We will use one thread per CPU and one thread as a 'supervisor' for our simulation. The supervisor thread will spawn new processes (as if a user started a process). The CPU threads will simulate the currently-running processes on each CPU, and the supervisor thread will print output.

Since the code you write will be called from multiple threads, the CPU scheduler you write must be thread-safe! This means that all data structures you use, including your ready queue, must be protected using mutexes.

The number of CPUs is specified as a command-line parameter to the simulator. For this project, you will be performing experiments with 1, 2, and 4 CPU simulations.

Also, for demonstration purposes, the simulator executes much slower than a real system would. In the real world, a CPU burst might range from one to a few hundred milliseconds, whereas in this simulator, they range from 0.2 to 2.0 seconds.

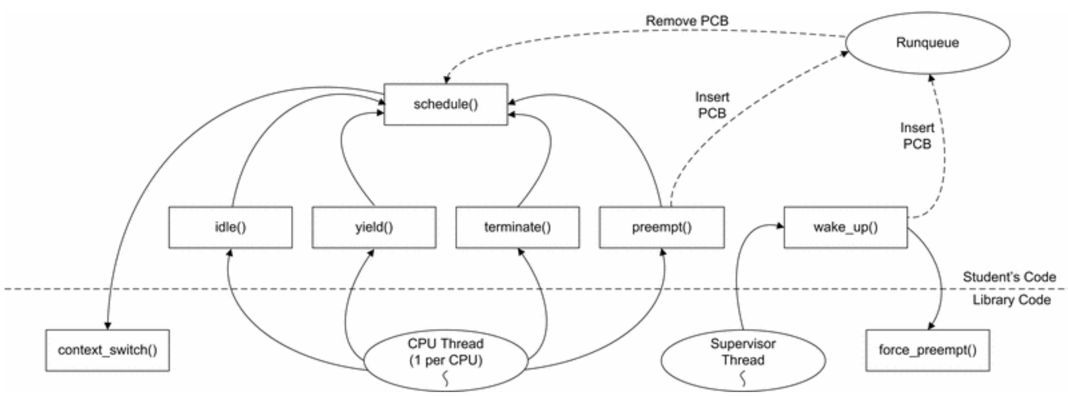

Figure 2: Simulator Function Calls

The above diagram should give you a good overview of how the system works in terms of the functions being called and PCBs moving around. Below is a second diagram that shows the entire system overview and the code that you need to write is inside of the green cloud at the bottom. All of the items outside of the green cloud are part of the simulator and will not need to be modified by you. If you would like to zoom in to get a better look, this image is included in the project files as `system-diagram.jpg`.

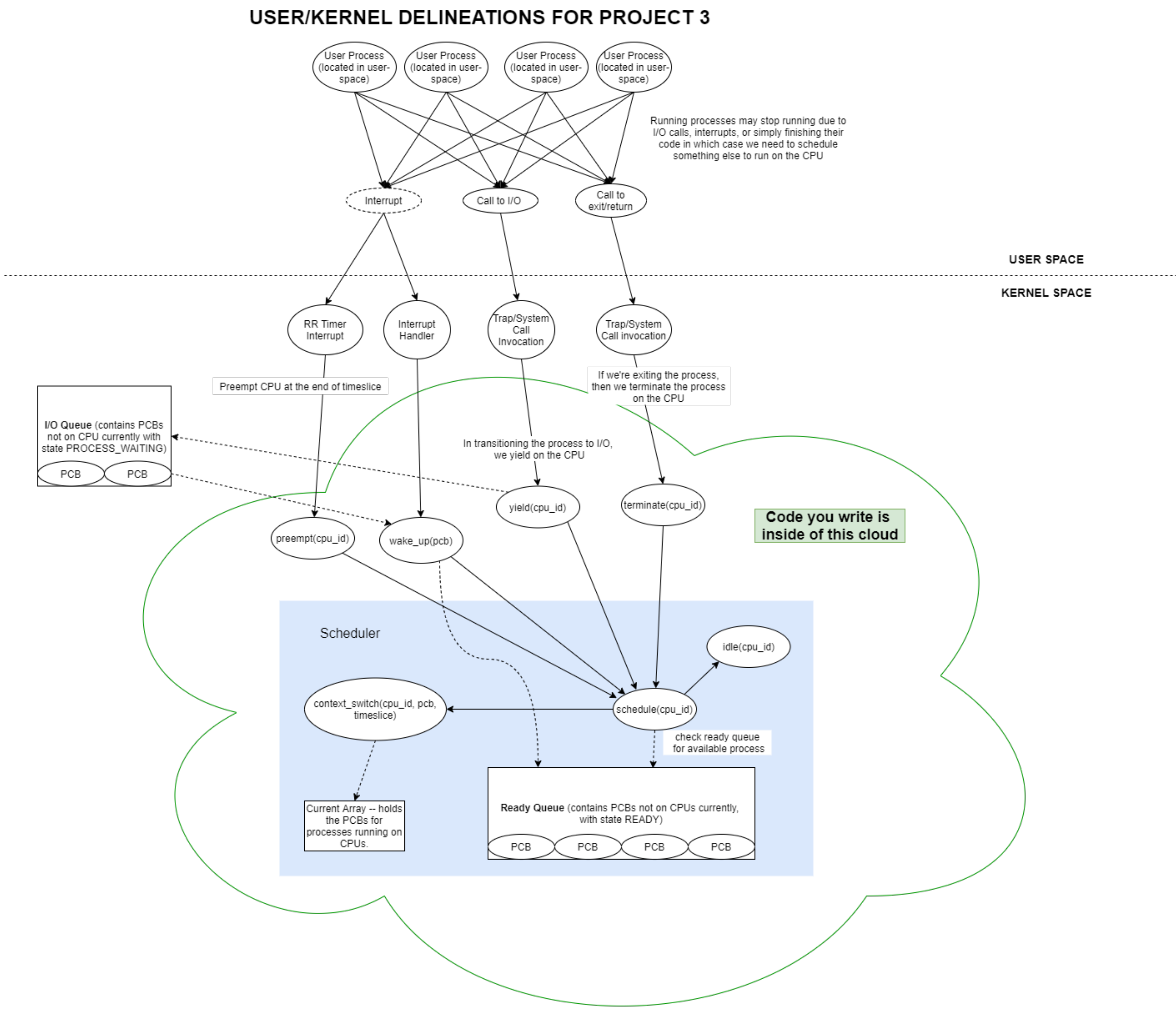

Figure 3: System overview

Compile and run the simulator with `./os-sim 2`. After a few seconds, hit Control-C to exit. You will see the output below:

```
Time  Ru Re Wa     CPU 0    CPU 1       < I/O Queue <
===== == == ==     ======== ========    =============
0.0   0  0  0      (IDLE)   (IDLE)       < <
0.1   0  0  0      (IDLE)   (IDLE)       < <
0.2   0  0  0      (IDLE)   (IDLE)       < <
0.3   0  0  0      (IDLE)   (IDLE)       < <
0.4   0  0  0      (IDLE)   (IDLE)       < <
0.5   0  0  0      (IDLE)   (IDLE)       < <
0.6   0  0  0      (IDLE)   (IDLE)       < <
0.7   0  0  0      (IDLE)   (IDLE)       < <
0.8   0  0  0      (IDLE)   (IDLE)       < <
0.9   0  0  0      (IDLE)   (IDLE)       < <
1.0   0  0  0      (IDLE)   (IDLE)       < <
......
```

Figure 4: Sample Output

The simulator generates a Gantt Chart, showing the current state of the OS at every 100ms interval. The leftmost column shows the current time, in seconds. The next three columns show the number of Running, Ready, and Waiting processes, respectively. The next two columns show the process currently running on each CPU. The rightmost column shows the processes which are currently in the I/O queue, with the head of the queue on the left and the tail of the queue on the right.

As you can see, nothing is executing. This is because we have no CPU scheduler to select processes to execute! Once you complete Problem 1 and implement a basic FIFO scheduler, you will see the processes executing on the CPUs.

2 Problem 0: The Ready Queue

Implement the helper functions `enqueue()`, `dequeue()`, and `is_empty()` in `student.c` for the queue struct provided. The struct will serve as your ready queue, and you should be using these helper functions to add and remove processes from the ready queue in the problems to follow. You can find the declarations of `queue_t`, `enqueue()`, `dequeue()`, and `is_empty()` in `student.h`.

2.1 Hints

- Your queue should be backed by a linked list with each PCB acting as a node. There is a field in the PCB, `next`, which you may use to build linked lists of PCBs.
- You should not be editing any of the fields of the PCBs inside the queue aside from `next`.
- There is an edge case for `dequeue()` which you must handle. Take a look at the documentation for the function for more details.
- When using the ready queue helper functions in the following problems, make sure to call them in a thread-safe manner. Read up on how to use mutex locks and lock/unlock your queue struct if and when you call these functions.

3 Problem 1: FCFS Scheduler

NOTE: Part B of this and each following problem requires you to put your answer down in answers.txt

Part A. Implement the CPU scheduler using the FCFS scheduling algorithm. You may do this however you like, however, we suggest the following:

- Implement the `yield()`, `wake_up()`, and `terminate()` handlers. `preempt()` is not necessary for this stage of the project. See the overview and the comments in the code for the proper behavior of these events.
- Implement `idle()`. `idle()` must wait on a condition variable that is signalled whenever a process is added to the ready queue.
- Implement `schedule()`. `schedule()` should extract the first process in the ready queue, then call `context_switch()` to select the process to execute. If there are no runnable processes, `schedule()` should call `context_switch()` with a NULL pointer as the PCB to execute the idle process.

3.1 Hints

- Be sure to update the `state` field of the PCB. The library will read this field to generate the Running, Ready, and Waiting columns, and to generate the statistics at the end of the simulation.
- Four of the five entry points into the scheduler (`idle()`, `yield()`, `terminate()`, and `preempt()`) should cause a new process to be scheduled on the CPU. In your handlers, be sure to call `schedule()`, which will select a runnable process, and then call `context_switch()`. When these four functions return, the library will simulate the execution of the process selected by `context_switch()`.
- `context_switch()` takes a timeslice parameter, which is used for preemptive scheduling algorithms. Since FCFS is non-preemptive, use -1 for this parameter to give the process an infinite timeslice.
- Make sure to use the helper functions in a thread-safe manner when adding and removing processes from the ready queue!
- The `current[]` array should be used to keep track of the process currently executing on each CPU. Since this array is accessed by multiple CPU threads, it must be protected by a mutex. `current_mutex` has been provided for you.

Part B. Run your OS simulation with 1, 2, and 4 CPUs. Compare the total execution time of each. Is there a linear relationship between the number of CPUs and total execution time? Why or why not? Keep in mind that the execution time refers to the simulated execution time.

4 Problem 2: Priority

Part A. Add priority scheduling to your code. The only thing you should change from the FCFS scheduler should be your `enqueue()` and `dequeue()` functions.

Modify `main()` to accept the -p parameter to select `PS` as the `scheduler_algorithm`. The default FCFS scheduler should continue to work. (Hint: create a global variable to identify the type of your scheduling algorithm).

The scheduler should use the `priority` field of the PCB to prioritize processes that have a higher priority (**NOTE: Lower number means higher priority**).

For priority scheduling, the `current[]` array should be used to keep track of the process currently executing on each CPU.

Since this array is accessed by multiple CPU threads, it must be protected by a mutex. `current_mutex` has been provided for you.

Part B. Priority schedulers can sometimes lead to starvation among processes with lower priority. What is a way that operating systems can mitigate starvation in a priority scheduler?

5 Problem 3: Round-Robin Scheduler

Part A. Add Round-Robin scheduling functionality to your code. You should modify main() to add a command line option, -r, which selects the Round-Robin scheduling algorithm, and accepts a parameter, the length of the timeslice. For this project, timeslices are measured in tenths of seconds. E.g.:

`./os-sim <# CPUs> -r 5`

should run a Round-Robin scheduler with timeslices of 500 ms. While:

`./os-sim <# of CPUs>`

should continue to run a FCFS scheduler. You should also make sure preempt is implemented in this section of the project.

To specify a timeslice when scheduling a process, use the timeslice parameter of `context_switch()`. The simulator will simulate a timer interrupt to preempt the process and call your `preempt()` handler if the process executes on the CPU for the length of the timeslice without terminating or yielding for I/O.

Part B. Run your Round-Robin scheduler with timeslices of 800ms, 600ms, 400ms, and 200ms. Use only one CPU for your tests. Compare the statistics at the end of the simulation. Is there a relationship between the total waiting time and timeslice length? If so, what is it? In contrast, in a real OS, the shortest timeslice possible is usually not the best choice. Why not?

6 Problem 4: The Priority Inversion Problem (Short Answer)

Assume we have three processes P1, P2, and P3; P1 has high priority, P2 has medium priority, and P3 has low priority.

Assume P1 and P3 work with/access a shared resource S, *and that* P2 *does not need or use* S. There exists a locking mechanism so that for any process P which is currently using S, no other process can use S unless P gives up the lock.

Assume that, currently, P3 has control of S. If P1 tries to get access to S, P1 must wait until P3 (a lower priority process) gives up S (and releases the locking mechanism).

If P2 becomes runnable in this time interval, during the time that P1 is waiting on P3 to give up S, P2 will preempt P3, as P2 has higher priority than P3 and doesn't require the usage/access to S. As a result, P3 is unable to give up access to S (because the preemption done by P2 didn't affect P3's holding of the lock).

We face an interesting problem: we observe starvation of a high priority process P1 by lower priority processes P2 and P3. This is known as a *priority inversion problem.*

Assume we have a scheduler that does have preemption (and we cannot use non-preemptive schedulers). Assume also that we can decrease or increase the priority of a process *during its runtime on the CPU.* Given these constraints, how might we ensure that P1 is able to execute before P2 ?

7 The Gradescope Environment

You will be submitting files to Gradescope, where they will be tested on/in a VM setup that runs through Gradescope. The specifications of this VM are that it runs `Ubuntu 20.04 LTS (64-bit)` and `gcc 9.3.0`, and so we expect that your files can run in such a setup. This means that *when you are running your project locally*, you will want to ensure you are using a VM/some setup that runs `Ubuntu 20.04 LTS (64-bit)` and `gcc 9.3.0`; this way, you can ensure that what occurs locally is what will occur when you submit to Gradescope.

8 Deliverables

NOTE: Each problem (excluding Problem 0 and Problem 4) has two parts (labeled A and B). The first is the actual implementation, and the second is a question linked to the scheduling algorithm you are implementing. Make sure you complete both.

You need to upload `student.c`, and `answers.txt` to Gradescope, and an autograder will run to check if your scheduler is working. The autograder might take a couple of minutes to run.

Keep your answers detailed enough to cover the question, including support from simulator results if appropriate. Don't write a book; but if you're not sure about an answer, err on the side of giving us too much information.

9 How to Run / Debug Your Code – Debugging deadlocks and synchronization errors

9.1 Running

We have provided a Makefile that will run gcc for you. To compile your code with no optimizations (which you should do while developing, it will make debugging easier) and test with the fcfs algorithm and one CPU, run:

```
$ make debug
$./os-sim 1
```

To run the other algorithms, run with the flags you implemented for round robin and priority. Remember that round robin requires you to enter a time slice.

In case you encounter difficulties with Project 4 and are uncertain about the direction to take, various resources are available to assist you.

9.2 GDB

Let us investigate how to debug deadlocks through a basic example:

```c
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t lock1;
pthread_mutex_t lock2;

void *thread1(void *data)
{
    pthread_mutex_lock(&lock1);
    sleep(1);
    pthread_mutex_lock(&lock2);
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
}

void *thread2(void *data)
{
    pthread_mutex_lock(&lock2);
    sleep(1);
    pthread_mutex_lock(&lock1);
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
}

int main()
{
    pthread_t t1, t2;
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
}
```

Following the execution and compilation of the code, it appears to become unresponsive. To investigate the root cause of this issue, it is recommended to utilize the GNU Debugger (gdb) to identify the problem:

```
(gdb) r
Starting program: /mnt/c/Users/kevin/OneDrive/Desktop/GT/CS 2200/TA/deadlock-demo/deadlock
BFD: /usr/lib/debug/.build-id/45/87364908de169dec62ffa538170118c1c3a078.debug: unable to initialize decompress status for se
ction .debug_aranges
BFD: /usr/lib/debug/.build-id/45/87364908de169dec62ffa538170118c1c3a078.debug: unable to initialize decompress status for se
ction .debug_aranges
warning: File "/usr/lib/debug/.build-id/45/87364908de169dec62ffa538170118c1c3a078.debug" has no build-id, file skipped
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
BFD: /usr/lib/debug/.build-id/18/78e6b475720c7c51969e69ab2d276fae6d1dee.debug: unable to initialize decompress status for se
ction .debug_aranges
BFD: /usr/lib/debug/.build-id/18/78e6b475720c7c51969e69ab2d276fae6d1dee.debug: unable to initialize decompress status for se
ction .debug_aranges
warning: File "/usr/lib/debug/.build-id/18/78e6b475720c7c51969e69ab2d276fae6d1dee.debug" has no build-id, file skipped
[New Thread 0x7ffff7da8700 (LWP 426)]
[New Thread 0x7ffff75a7700 (LWP 427)]
```

As anticipated, the program continues to remain unresponsive when run within the gdb environment. To interrupt the program, press Ctrl + c. To analyze the various threads associated with the program, utilize the ”info threads” command within gdb, which provides detailed information regarding each active thread:

```
Thread 1 "deadlock" received signal SIGINT, Interrupt.
__pthread_clockjoin_ex (threadid=140737351681792, thread_return=0x0, clockid=<optimized out>, abstime=<optimized out>,
    block=<optimized out>) at pthread_join_common.c:145
145     pthread_join_common.c: No such file or directory.
(gdb) info threads
  Id   Target Id         Frame
* 1    Thread 0x7ffff7da9740 (LWP 422) "deadlock" __pthread_clockjoin_ex (threadid=140737351681792, thread_return=0x0,
    clockid=<optimized out>, abstime=<optimized out>, block=<optimized out>) at pthread_join_common.c:145
  2    Thread 0x7ffff7da8700 (LWP 426) "deadlock" __lll_lock_wait (futex=futex@entry=0x555555558080 <lock2>, private=0)
    at lowlevellock.c:52
  3    Thread 0x7ffff75a7700 (LWP 427) "deadlock" __lll_lock_wait (futex=futex@entry=0x555555558040 <lock1>, private=0)
    at lowlevellock.c:52
```

Upon examining the running threads using the ”info threads” command within gdb, we can observe that threads 2 and 3, which were created within the main() function, are currently situated within the __lll_lock_wait function. To obtain the backtrace of these threads, we can use the ”thread apply all” command along with the ”backtrace” command, which can be abbreviated as ”t a a bt”.

```
(gdb) t a a bt

Thread 3 (Thread 0x7ffff75a7700 (LWP 427)):
#0  __lll_lock_wait (futex=futex@entry=0x555555558040 <lock1>, private=0) at lowlevellock.c:52
#1  0x00007ffff7fa90a3 in __GI___pthread_mutex_lock (mutex=0x555555558040 <lock1>) at ../nptl/pthread_mutex_lock.
#2  0x0000555555555288 in thread2 (data=0x0) at deadlock.c:20
#3  0x00007ffff7fa6609 in start_thread (arg=<optimized out>) at pthread_create.c:477
#4  0x00007ffff7ecb133 in clone () from /lib/x86_64-linux-gnu/libc.so.6

Thread 2 (Thread 0x7ffff7da8700 (LWP 426)):
#0  __lll_lock_wait (futex=futex@entry=0x555555558080 <lock2>, private=0) at lowlevellock.c:52
#1  0x00007ffff7fa90a3 in __GI___pthread_mutex_lock (mutex=0x555555558080 <lock2>) at ../nptl/pthread_mutex_lock.
#2  0x000055555555523b in thread1 (data=0x0) at deadlock.c:11
#3  0x00007ffff7fa6609 in start_thread (arg=<optimized out>) at pthread_create.c:477
#4  0x00007ffff7ecb133 in clone () from /lib/x86_64-linux-gnu/libc.so.6

Thread 1 (Thread 0x7ffff7da9740 (LWP 422)):
#0  __pthread_clockjoin_ex (threadid=140737351681792, thread_return=0x0, clockid=<optimized out>, abstime=<optimi
    block=<optimized out>) at pthread_join_common.c:145
#1  0x000055555555532b in main () at deadlock.c:32
```

The backtrace command confirms that threads 2 and 3 are indeed stuck at the pthread_mutex_lock function. To gain a more in-depth understanding of the specific thread’s state, we can utilize the gdb command ”thread

[thread number]" to switch to a particular thread and examine its current state.

```
(gdb) t 3
[Switching to thread 3 (Thread 0x7ffff75a7700 (LWP 427))]
#0  __lll_lock_wait (futex=futex@entry=0x555555558040 <lock1>, private=0) at lowlevellock.c:52
52      lowlevellock.c: No such file or directory.
(gdb) bt
#0  __lll_lock_wait (futex=futex@entry=0x555555558040 <lock1>, private=0) at lowlevellock.c:52
#1  0x00007ffff7fa90a3 in __GI___pthread_mutex_lock (mutex=0x555555558040 <lock1>) at ../nptl/pthread_mutex_lock.
#2  0x0000555555555288 in thread2 (data=0x0) at deadlock.c:20
#3  0x00007ffff7fa6609 in start_thread (arg=<optimized out>) at pthread_create.c:477
#4  0x00007ffff7ecb133 in clone () from /lib/x86_64-linux-gnu/libc.so.6
(gdb) f 2
#2  0x0000555555555288 in thread2 (data=0x0) at deadlock.c:20
20    _     pthread_mutex_lock(&lock1);
```

By switching to thread 3 within gdb, we can identify the precise line of code where it has become deadlocked. Once we have identified the problematic line, we can utilize gdb's features, such as printing values or switching stack frames, to investigate further and gain a better understanding of the issue at hand. Read the gdb thread documentation here for more information.

9.3 Valgrind (Helgrind or DRD)

What about issues when accessing a shared resource? Valgrind has some handy tools to help detect such problems. You may use your 2110 docker or just look up "valgrind installation" online for download instructions. Let's modify the code from the previous section:

```c
#include <pthread.h>

pthread_mutex_t lock;
int shared;

void *thread1(void *data)
{
    pthread_mutex_lock(&lock);
    shared = 1;
    pthread_mutex_unlock(&lock);
}

void *thread2(void *data)
{
    shared = 2;
}

int main()
{
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
}
```

Lets run Helgrind with the command 'valgrind tool=helgrind <program>'. This is the result:

```
 Lock at 0x10C040 was first observed
   at 0x4843D9D: pthread_mutex_init (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind-amd64-linux.so)
   by 0x109262: main (deadlock.c:21)
 Address 0x10c040 is 0 bytes inside data symbol "lock"

Possible data race during write of size 4 at 0x10C068 by thread #3
Locks held: none
   at 0x10922A: thread2 (deadlock.c:15)
   by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind-amd64-linux.so)
   by 0x4860608: start_thread (pthread_create.c:477)
   by 0x499A132: clone (clone.S:95)

This conflicts with a previous write of size 4 by thread #2
Locks held: 1, at address 0x10C040
   at 0x109205: thread1 (deadlock.c:9)
   by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind-amd64-linux.so)
   by 0x4860608: start_thread (pthread_create.c:477)
   by 0x499A132: clone (clone.S:95)
 Address 0x10c068 is 0 bytes inside data symbol "shared"
```

Upon executing Valgrind's tool Helgrind, we can observe that it has successfully identified an issue within the program where thread2 is accessing a shared variable without acquiring a corresponding lock. Additionally, DRD (another tool within Valgrind) also provides comparable output, albeit with fewer error lines. It is essential to rectify these issues to ensure proper synchronization and avoid potential data race conditions. To compare, here is the same program run with DRD ('valgrind tool=drd <program>'):

```
Thread 3:
Conflicting store by thread 3 at 0x0010c068 size 4
   at 0x10922A: thread2 (deadlock.c:15)
   by 0x48424BA: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_drd-amd64-linux.so)
   by 0x486C608: start_thread (pthread_create.c:477)
   by 0x49A6132: clone (clone.S:95)
Allocation context: BSS section of /mnt/c/Users/kevin/OneDrive/Desktop/GT/CS 2200/TA/deadlock-demo/deadlock
Other segment start (thread 2)
   (thread finished, call stack no longer available)
Other segment end (thread 2)
   (thread finished, call stack no longer available)
```

Valgrind and DRD are also able to debug other types of synchronization errors. You can read the documentation about Helgrind here and DRD here.

Credit to this video from an old class I took.