CIS 5480 – Midterm Review Session

Agenda

- Review Summer 2025 Midterm
- Review any topics you have questions on (time-permitting)

Question 1

1 Files and File Descriptors [25 Points]

Take a look at the following block of code, which forks two child process, establishes a pipe between the two children, dup2s a file descriptor, and calls execvp in both of the children. After being compiled, the program is executed in the shell. **You may assume all functions are successful.**

```
1
2  int main(){
3
4      int fd_in = open("in.txt", O_RDONLY);
5      // LINE 5
6      int pipe[2];
7      pipe2(&pipe, O_CLOEXEC);
8
9      pid_t pids[2];
10     pids[0] = fork();
11
12     if(pids[0] == 0){
13         dup2(fd_in, STDIN_FILENO);
14         dup2(pipe[1], STDOUT_FILENO);
15         close(fd_in);
16         execvp(...);
17     }
18
19     close(fd_in);
20
21     pids[1] = fork();
22     if(pids[1] == 0){
23         dup2(pipe[0], STDIN_FILENO);
24         execvp(...);
25     }
26
27     waitpid(pids[1], NULL, 0);
28 }
```

(a) How many descriptors? [5 Points]

By the time we reach `line 5` (just before the pipe), **how many file descriptors are in use? Justify your answer** by listing each one by its standard name or the file it refers to.

Solution. At the start, we have three in usage, STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO. With the open calls, we add 1 more for the file `in.txt`. So, 4.

Question 1

1 Files and File Descriptors [25 Points]

Take a look at the following block of code, which forks two child process, establishes a pipe between the two children, dup2s a file descriptor, and calls execvp in both of the children. After being compiled, the program is executed in the shell. **You may assume all functions are successful.**

```
1
2  int main(){
3
4      int fd_in = open("in.txt", O_RDONLY);
5      // LINE 5
6      int pipe[2];
7      pipe2(&pipe, O_CLOEXEC);
8
9      pid_t pids[2];
10     pids[0] = fork();
11
12     if(pids[0] == 0){
13         dup2(fd_in, STDIN_FILENO);
14         dup2(pipe[1], STDOUT_FILENO);
15         close(fd_in);
16         execvp(...);
17     }
18
19     close(fd_in);
20
21     pids[1] = fork();
22     if(pids[1] == 0){
23         dup2(pipe[0], STDIN_FILENO);
24         execvp(...);
25     }
26
27     waitpid(pids[1], NULL, 0);
28 }
```

(b) How many now? [7 Points]

By the time both children have exec'd (but have not finished running), **how many *file descriptors are open in total across both children?* Do not take into account the parent. You may assume the children do not modify their own descriptors or open any files after exec'd**

Solution. Before we `fork()`, there are 3 open standard file descriptors, one open file descriptor for `in.txt`, and two file descriptors for the pipe (one for each end). That is 6 in total. In the first child, we `dup2` two file descriptors. This closes `STDIN_FILENO` and `STDOUT_FILENO`, but they are immediately reused to point to the new file descriptors. The only genuine close is the close of `fd_in`. Finally, the `exec` call closes the unused pipe ends. So, this child has 3 file descriptors open after the `exec`.

In the second child, we close `fd_in` prior to the `fork`, so it starts with 5 open file descriptors. As seen before, although `dup2` closes the file descriptor, it reuses it immediately after. Finally, the `exec` closes the unused pipe ends. So it also ends up with 3 open file descriptors after `exec`.

In total, 6 file descriptors across both processes.

Child 1 Start
0 stdin (terminal)
1 stdout (terminal)
2 stderr (terminal)
3 fd_in (in.txt)
4 pipe[0] (R) (FD_CLOEXEC set)
5 pipe[1] (W) (FD_CLOEXEC set)

Child 1 End
0 in.txt (stdin)
1 pipe[1] (W) (stdout)
2 stderr (terminal)

Child 2 Start
0 stdin (terminal)
1 stdout (terminal)
2 stderr (terminal)
4 pipe[0] (R) (FD_CLOEXEC set)
5 pipe[1] (W) (FD_CLOEXEC set)

Child 2 End
0 pipe[0] (R) (stdin)
1 stdout (terminal)
2 stderr (terminal)

Question 1

1 Files and File Descriptors [25 Points]

Take a look at the following block of code, which forks two child process, establishes a pipe between the two children, dup2s a file descriptor, and calls execvp in both of the children. After being compiled, the program is executed in the shell. **You may assume all functions are successful.**

```
1
2  int main(){
3
4      int fd_in = open("in.txt", O_RDONLY);
5      // LINE 5
6      int pipe[2];
7      pipe2(&pipe, O_CLOEXEC);
8
9      pid_t pids[2];
10     pids[0] = fork();
11
12     if(pids[0] == 0){
13         dup2(fd_in, STDIN_FILENO);
14         dup2(pipe[1], STDOUT_FILENO);
15         close(fd_in);
16         execvp(...);
17     }
18
19     close(fd_in);
20
21     pids[1] = fork();
22     if(pids[1] == 0){
23         dup2(pipe[0], STDIN_FILENO);
24         execvp(...);
25     }
26
27     waitpid(pids[1], NULL, 0);
28 }
```

(c) Closing Time [8 Points]

You attend office hours to review the code above and sit between two other students, Mireya and Shayla. However, you notice that, at times, the program is stalled. You want to know **the minimum number of file descriptors to close** to keep the program from hanging.

Mireya says that just before calling `waitpid`, you should close the **write end** of the pipe to prevent processes from getting stuck. **The read end doesn't matter in this case.**

Shayla argues that you should **absolutely close both the write and read ends** of the pipe before calling `waitpid`.

Who is correct? Explain your reasoning.

Solution. Mireya is right. The only call that is blocking in this case would be a `read` call. And `read` only returns once an `EOF` is received, which occurs only when all write portions of the pipe are closed. So, at a minimum, we need to close the write end of the pipe.

Question 1

1 Files and File Descriptors [25 Points]

Take a look at the following block of code, which forks two child process, establishes a pipe between the two children, dup2s a file descriptor, and calls execvp in both of the children. After being compiled, the program is executed in the shell. **You may assume all functions are successful.**

```
1
2  int main(){
3
4      int fd_in = open("in.txt", O_RDONLY);
5      // LINE 5
6      int pipe[2];
7      pipe2(&pipe, O_CLOEXEC);
8
9      pid_t pids[2];
10     pids[0] = fork();
11
12     if(pids[0] == 0){
13         dup2(fd_in, STDIN_FILENO);
14         dup2(pipe[1], STDOUT_FILENO);
15         close(fd_in);
16         execvp(...);
17     }
18
19     close(fd_in);
20
21     pids[1] = fork();
22     if(pids[1] == 0){
23         dup2(pipe[0], STDIN_FILENO);
24         execvp(...);
25     }
26
27     waitpid(pids[1], NULL, 0);
28 }
```

(d) One waitpid? [5 Points]

After applying your fix and closing all file descriptors, you see that there's *only one waitpid call*. It might look like a race condition, but assume neither child exits abnormally. **If both exec'd programs are `cat`, explain why having just one waitpid is okay here.**

To help conceptualize, think of it like running this in the shell:

```
/workspace# cat < in.txt | cat
```

Solution. The key here is knowing that, based on the functionality we saw in class, `cat` just reads from `STDIN` and writes whatever is read. However, in order for `cat` to stop reading (and thus stop writing), it has to receive an `EOF` to indicate that nothing else will come from its `STDIN`. This occurs when the first `cat` exits, as its file descriptors are closed. Thus, when the second process exits, it must be the case that the first has exited too. If the parent waits for both, it will only exit after both children have terminated.

Note: It is important that we reap all children appropriately. Still, the child that is "orphaned" is really just an orphaned zombie process, and those get reaped by the first user process created by the kernel, `init`.

Question 2

2 Time to Shreddddd [15 Points]

Take a look at the code below—this is a basic outline of a shredder implementation from a different version of the course, where **functionality within signal handlers is not allowed**.

Only the child process uses `sigaction`, but it doesn't install a real handler. When the parent receives a `SIGALRM`, it simply forwards that signal to the child to terminate it (as shown in part (b)). **You can assume the parent is always able to forward the signal when received.** However, as you'll see, there are some issues with this approach.

```
pid_t child_pid = 0;

void forwardAlarm(){
    kill(child_pid, SIGALRM);
    writeToStdout("Bwahaha ... tonight I dine on turtle soup\n");
}

void executeShell(int timeout){

    struct sigaction sa_ignore = {
        .sa_handler = SIG_IGN
    };

    command = getCommandFromInput();

    if (command != NULL) {
        child_pid = fork();
        alarm(timeout);
        ....
        if (childPid == 0) {
            sigaction(SIGALRM, sa_ignore);
            fork_exec(command); //assume this works
        }
        if (childPid != 0) {
            waitForChild(); //assume this works
        }
    }
}
```

(a) The Disposition of SIGALRM [5 Points]

When the parent receives a SIGALRM a problem occurs. The parent forwards the `SIGALRM` to the child but the child does not terminate from the signal. As a result, the `waitForChild()` call doesn't return (unless the child exits some other way, e.g. it exits normally).

Explain why this happens and suggest a possible fix.

Solution. This happens due to the disposition of the `SIGALRM` signal in the child being set to `SIG_IGN`. Although we call `execvp` and signal handlers are reset along with the signal mask, dispositions set to `SIG_IGN` persist across an `exec`. As a result, the child continues to ignore `SIGALRM` even when it is sent one from the parent.

Removing the `sigaction(SIGALRM, sa_ignore);` in the child will fix the issue.

Question 2

2 Time to Shreddddd [15 Points]

Take a look at the code below—this is a basic outline of a shredder implementation from a different version of the course, where **functionality within signal handlers is not allowed**.

Only the child process uses `sigaction`, but it doesn't install a real handler. When the parent receives a `SIGALRM`, it simply forwards that signal to the child to terminate it (as shown in part (b)). **You can assume the parent is always able to forward the signal when received.** However, as you'll see, there are some issues with this approach.

Solution. The `SA_RESTART` Flag is used to restart system calls that can be interrupted by signals. In this case, we only care about wait being restarted if *it needs to be*. Typically, the wait call can be interrupted by the handing of a SIGCHLD or if in the typical assignment, by a SIGINT or SIGTSTP. In this case `sigwait` is doing most of the heavy lifting as it returns once a signal becomes pending. Only two signals can be pending, SIGCHLD and SIGALARM.

By using sigwait here, we can control when the while loop will exit; when a SIGCHLD is received without needing to use the signal handler at all. It can not be called because the signal is already blocked. So, the `wait` call eturns immediately, as the child must be reapable upon exiting the while loop.

Note: We do not expect responses to be this verbose. Additionally, the reason we install a no-op sigchld handler is to remove its SIG_IGN disposition. Without that, those signals can not be pending. You were not required to know this.

```
    if (command != NULL) {
        child_pid = fork();
        alarm(timeout);
        ....
        if (childPid == 0) {
            sigaction(SIGALRM, sa_ignore);
            fork_exec(command); //assume this works
        }
        if (childPid != 0) {
            waitForChild(); //assume this works
        }
    }
}
```

(b) Waiting for Child [10 Points]

Consider the following possible implementation of the `waitForChild` function. You can assume all children run to completion and are never stopped or suspended. **You may refer to the `sigwait` man page for more details. Assume the issue from part (a) is fixed.**

```
void emptyHandler(int signal){ return; }

int blockSignals(){
    // <SIGALRM & SIGCHLD are blocked using sigprocmask>
}

void waitForChild(pid_t child_pid){
    blockSignals();

    //installs the emptyHandler for the SIGCHLD signal
    install_sigchld_handler();

    int sig;
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGCHLD);
    sigaddset(&set, SIGALRM);

    while( (sigwait(&set, &sig)) == 0) ){
        if(sig == SIGCHLD){
            alarm(0); //cancel alarm just in case.
            break;
        } else {
            forwardAlarm();
        }
    }
    wait(NULL);
}
```

In lecture, we mentioned the importance of using the `SA_RESTART` flag when setting up signal handlers. However, you may have noticed that it wasn't used in this implementation. **Based on the `waitForChild` function above, explain why `SA_RESTART` is not needed here.** In other words, **why is `wait` guaranteed to return immediately?**

Question 3

3 Inodes and Calculations [20 Points]

As mentioned in lecture, a lot of work in a file system involves calculating the offsets of Blocks, Inodes, and byte locations within files. Path resolution is also a key part of this process. Let's walk through a few examples together.

You may assume the following for this question.

- **Inodes Block Start:** Block 2
- **Inode size:** 64 bytes
- **Block size:** 1024 bytes
- **Block number size:** 2 bytes
- **Block pointers in inode:**
  - 12 direct
  - 1 singly indirect
  - 1 doubly indirect
  - 1 triply indirect

(a) Best Case Scenario [4 Points]

You're resolving the path: `/usr/pennstdnt/shell.txt`. In the best case, how many total blocks must be read **to find the inode number** for `shell.txt`? Assume the inode for the root directory has already been loaded. **Briefly explain your answer.**

Solution. Assuming we have already read the inode for the root block involves also assuming we've read in the entire block of inodes. In the best case scenario, we can assume that all of the inodes we need can be stored in this one block. So, all we need to do is find the **directory entries** for 3 files: `usr, pennstdnd, and shell.txt`. If we assume that these entries are in the first data blocks of dirents the inodes reference then we only need to read 3 blocks, those which contain the directory entires for those three files.

Question 3

3 Inodes and Calculations [20 Points]

As mentioned in lecture, a lot of work in a file system involves calculating the offsets of Blocks, Inodes, and byte locations within files. Path resolution is also a key part of this process. Let's walk through a few examples together.

You may assume the following for this question.

- **Inodes Block Start:** Block 2
- **Inode size:** 64 bytes
- **Block size:** 1024 bytes
- **Block number size:** 2 bytes
- **Block pointers in inode:**
  - 12 direct
  - 1 singly indirect
  - 1 doubly indirect
  - 1 triply indirect

(b) Worst Case Scenario [6 Points]

You're resolving the same path: `/usr/pennstdnt/shell.txt`. In the worst case, how many blocks must be read **to find the inode number** for `shell.txt`? Again, assume the root inode is already in memory. You may leave your answer as an expression. **Briefly explain your answer.**

Solution. In the worse case scenario, each of the inodes whos entries we need are in separate blocks. These are the inodes we need: `usr` and `pennstdn`. So, we are at 2 blocks.

Next, we must to find the **directory entries** for 3 files: `usr, pennstdnd, and shell.txt`. In the worst case scenario, those entries are in the last data blocks of dirents the inodes reference. The maximum amount each can reference is the follwowing:

$$max = 12 + \frac{1024}{2} + \frac{1024}{2} \times \frac{1024}{2} + \frac{1024}{2} \times \frac{1024}{2} \times \frac{1024}{2}$$

So the overall total is

$$3 * max + 2$$

Question 3

3 Inodes and Calculations [20 Points]

As mentioned in lecture, a lot of work in a file system involves calculating the offsets of Blocks, Inodes, and byte locations within files. Path resolution is also a key part of this process. Let's walk through a few examples together.

You may assume the following for this question.

- **Inodes Block Start:** Block 2
- **Inode size:** 64 bytes
- **Block size:** 1024 bytes
- **Block number size:** 2 bytes
- **Block pointers in inode:**
  - 12 direct
  - 1 singly indirect
  - 1 doubly indirect
  - 1 triply indirect

(c) Locating the Inode [6 Points]

You've found that the inode number for `text.txt` is 6723. Which block contains this inode, and what is the index (or offset) of the inode within that block? Show your work.

Solution. Following the calculations used in class, we can do the following.

$$\text{inodes_per_block} = \frac{1024}{64}$$

$$\text{block_num} = 2 + \left\lfloor \frac{6723 - 1}{\text{inodes_per_block}} \right\rfloor$$

$$\text{inode_rel_index} = (6723 - 1)\%\text{inodes_per_block}$$

Question 3

3 Inodes and Calculations [20 Points]

As mentioned in lecture, a lot of work in a file system involves calculating the offsets of Blocks, Inodes, and byte locations within files. Path resolution is also a key part of this process. Let's walk through a few examples together.

You may assume the following for this question.

- **Inodes Block Start:** Block 2
- **Inode size:** 64 bytes
- **Block size:** 1024 bytes
- **Block number size:** 2 bytes
- **Block pointers in inode:**
  - 12 direct
  - 1 singly indirect
  - 1 doubly indirect
  - 1 triply indirect

(d) Locating the Byte [4 Points]

Using the inode from the previous part, determine **which data block of the file** (i.e., the `i`th block containing the file's actual contents) is needed to access `byte 102345`.

For example, `byte 999` is in the **0th data block of the file**, and `byte 12288` is in the **11th data block of the file** (which is the last direct block).

In addition to the block number within the file, also determine the **path required to locate that block via the inode** — whether it is accessed through a **direct pointer**, or via a **singly, doubly, or triply indirect pointer**.

You may leave your answers as expressions if convenient. **Show your work.**

Solution. This is a bit more involved than you would expect, but the math plays out nicely. We will not need to navigate the doubly or triply indirect blocks. We do have to calculate the range of block numbers referenced. First, we need to know which i^{th} block we need. Bytes 0 - 1023 are in block 0 and so forth. So byte, 102345 is in the following block:

$$\left\lfloor \frac{102345 - 1}{1024} \right\rfloor = 99$$

Now, it is a question of where this 99^{th} block is. Blocks 0 - 11 are used by the 12 direct block numbers. However, which blocks are referenced by the first singly indirect block? Well, we know that a singly direct block number references 512 blocks. These are blocks 12 - 523. So, block 99 must fall somewhere within this inode.

This is enough to receive credit for the problem. You did not need to find the relative index/offset of the block number within this block.

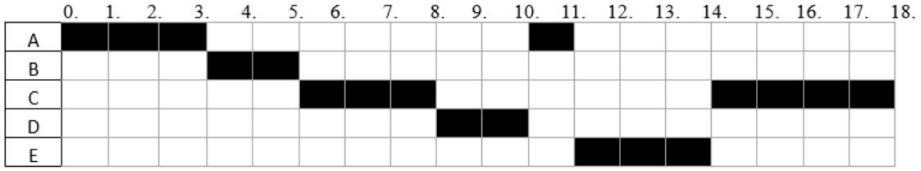
Question 4

4 I want to schedule more sleep time [26 Points]

In the class, we scheduled the following processes:

Process	Arrival Time	Duration
A	0	4
B	1	2
C	2	7
D	3	2
E	4	3

Scheduling this with a quantum of 3 with round robin algorithm results in 6 context switches. See the diagram below:



In this scenario, we made the following assumptions:

- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time quantum finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.
- If a process would not finish before the time quantum is up, and there are no other processes to schedule, then the process runs for another time quantum without context switching.

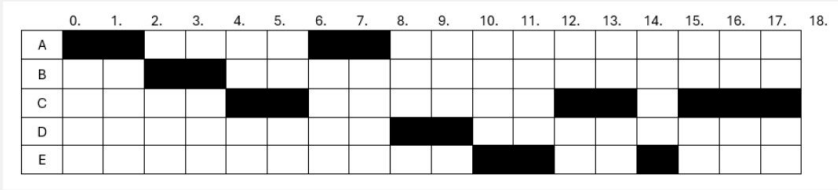
Let's make slight modifications to this scenario and see its impacts on the scheduler.

(a) Let's Schedule It [16]

Schedule the same tasks that we ran using round robin, **but with a time quantum of two**. We specifically want to see the time each task finishes. **Make sure to justify your answer by showing work.**

Solution.

If we were to draw out how we scheduled it, it would like like this:



This would have 8 total context switch with the following finishing times:

- A finishing at time 8
- B finishing at time 4
- C finishing at time 18
- D finishing at time 10
- E finishing at time 15

For more clarity, here is how the ready queue would change over time:

- Time 0: []
- Time 1: [B]
- Time 2: [C, A]
- Time 3: [C, A, D]
- Time 4 & 5: [A, D, E]
- Time 6 & 7: [D, E, C]
- Time 8 & 9: [E, C]
- Time 10 & 11: [C]
- Time 12 & 13: [E]
- Time 14: [C]
- The Rest: []

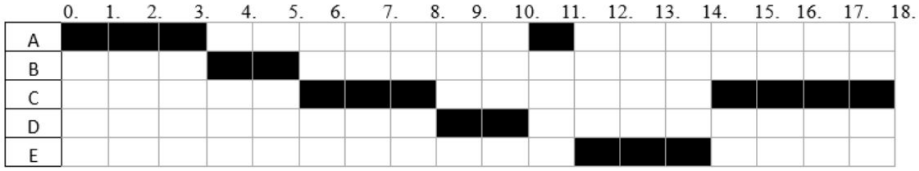
Question 4

4 I want to schedule more sleep time [26 Points]

In the class, we scheduled the following processes:

Process	Arrival Time	Duration
A	0	4
B	1	2
C	2	7
D	3	2
E	4	3

Scheduling this with a quantum of 3 with round robin algorithm results in 6 context switches. See the diagram below:



In this scenario, we made the following assumptions:

- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time quantum finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.
- If a process would not finish before the time quantum is up, and there are no other processes to schedule, then the process runs for another time quantum without context switching.

Let's make slight modifications to this scenario and see its impacts on the scheduler.

No More Assumptions

In our problem, we made several unrealistic assumptions about the nature of the scheduler and the processes it runs (For example: "All processes do not block for I/O or any resource" and "Context switching and running the Scheduler are instantaneous".) Real schedulers would have to consider this in their design.

Let's explore one advantage and one disadvantage in choosing a shorter time quantum instead of a longer time quantum for a real scheduler.

(b) Pros [5 Points]

In general (not specific to this scenario), what is one advantage to choosing a shorter time quantum instead of a longer time quantum for a real scheduler? **Justify your answer.**

Solution. There are a couple that one could argue, as long as it is justified correctly:

- It is Good for Interactive tasks since interactive tasks would have to wait less for non-interactive tasks to run before ti is the interactive tasks turn again
- If we have a bunch of short processes and some long processes we get some better throughput on the short processes since they won't have to wait as long for the long processes to run before the short processes can run again
- Less starvation / CPU deprivation in general since tasks should generally have to wait less before it is their turn again.
- Less initial waiting time since tasks should generally have to wait less before it is their turn again.

(c) Cons [5 Points]

In general (not specific to this scenario), what is one disadvantage to choosing a shorter time quantum instead of a longer time quantum for a real scheduler? **Justify your answer.**

Solution. The main con is that the context switching time would be higher. We would spend more time running the scheduler and switching between tasks instead of actually running the tasks themselves. There may be other cons, but this is the one we expect to see by far the most.

5 Threads of Sanity [12 Points]

Consider the following code that utilizes threads:

```
//global
bool flag = false;

void* thrd_fn1(void* arg) {

  if (flag) {
    printf("foo\n");
    flag = false;
  } else {
    printf("bar\n");
    flag = true;
  }

  return NULL;
}

void* thrd_fn2(void* arg) {

  flag = true;

  return NULL;
}

int main() {
  pthread_t thd1;
  pthread_t thd2;

  pthread_create(&thd1, NULL, thrd_fn1, NULL);
  pthread_create(&thd2, NULL, thrd_fn2, NULL);

  pthread_join(thd1, NULL);
  pthread_join(thd2, NULL);

  if (flag) {
    printf("flag is true\n");
  } else {
    printf("flag is false\n");
  }
}
```

(a) False Flag [5 Points]

Is it possible for main to print "flag is false"? If possible, describe the steps necessary in the programs execution to achieve this. If not possible, briefly justify why. **We are not looking for a formal proof nor is the output of thread 1 relevant.**

Solution. Yes, this is entirely possible. What would happen is that after main creates thread 1 and thread 2, it waits for both to finish. Then thread 2 could run first and set the global flag to true. Afterwards thread 1 can run to completion, print "foo", and set flag to false. At this point main can resume from its calls to pthread_join since both threads have finished. Main will then see that flag is false, and print "flag is false".

Assume that the code compiles and that none of the functions fail. Additionally, there are no hardware or compiler optimizations that occur. Using only what we have learned so far about threads in class, answer the questions on the next page.

5 Threads of Sanity [12 Points]

Consider the following code that utilizes threads:

```
//global
bool flag = false;

void* thrd_fn1(void* arg) {

  if (flag) {
    printf("foo\n");
    flag = false;
  } else {
    printf("bar\n");
    flag = true;
  }

  return NULL;
}

void* thrd_fn2(void* arg) {

  flag = true;

  return NULL;
}

int main() {
  pthread_t thd1;
  pthread_t thd2;

  pthread_create(&thd1, NULL, thrd_fn1, NULL);
  pthread_create(&thd2, NULL, thrd_fn2, NULL);

  pthread_join(thd1, NULL);
  pthread_join(thd2, NULL);

  if (flag) {
    printf("flag is true\n");
  } else {
    printf("flag is false\n");
  }
}
```

(b) Foo True [7 Points]

Is it possible for thread1 to print "foo" and main to print "flag is true"? If possible, describe the order of thread execution to achieve this. If not possible, briefly justify why. **Again, we are not looking for a formal proof.**

Solution. No it is not. For foo to be printed, then thread1 would have to see that the flag is true. For the flag to be true, it must be set to true by thread2 before hand since flag starts as false. Thread 1 will then set the flag to false before terminating. At this point thread 1 and thread 2 are finished so there is no further modification of the global flag. Main will then have to print "flag is false". So if "foo" is printed by thread 1, main must print "flag is false" and so the output in the question is not possible.

Note that it is not possible for main to print anything until both threads have terminated since main calls pthread_join on both threads before printing. Some may think that main can print "flag is false" after thread1 has printed but before thread1 has set the flag to false, but this is not possible because of the pthread_joins.

Assume that the code compiles and that none of the functions fail. Additionally, there are no hardware or compiler optimizations that occur. Using only what we have learned so far about threads in class, answer the questions on the next page.

Any Questions?