CS 162 Fall 2020 — Operating Systems and System Programming

Midterm 1

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

◯ You must choose either this option

◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

This is a **proctored, closed-book exam**. During the exam, you may **not** communicate with other people regarding the exam questions or answers in any capacity.

If there is something in a question that you believe is open to interpretation, please make a private post on Piazza asking for clarification. We will issue an announcement if we believe your question merits one.

We will overlook minor syntax errors in grading coding questions. You do not have to add necessary `#include` statements. For coding questions, the number of blank lines you see is a suggested guideline, but is not a strict minimum or maximum. There will be no limit on the length of your answer/solution.

a) Name

Peter Perfect

b) Student ID

162

c) Please read the following honor code: "I understand that this is a closed book exam. I hereby promise that the answers that I give on the following exam are exclusively my own. I understand that I am allowed to use one 8.5x11, double-sided, handwritten cheat-sheet of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or internet sources in constructing my answers." Type your full name below to acknowledge that you've read and agreed to this statement.

Peter Perfect

1. (22 points) True/False

Please **EXPLAIN** your answer in **TWO SENTENCES OR LESS** (Answers longer than this may not get credit!). Also, answers without any explanation **GET NO CREDIT!**

a) (2 points)

1 Given any two threads A and B, thread A is always able to read data from thread B's stack.

● True

○ False

2 Explain.

Threads of the same process share an address space. (It was clarified during the exam that thread A and B belong to the same process.)

b) (2 points)

1 Given any two threads A and B, thread A is always able to read data from thread B's heap.

● True

○ False

2 Explain.

Threads of the same process share an address space. (It was clarified during the exam that thread A and B belong to the same process.)

c) (2 points)

1 A pointer will point to the same (virtual) memory address before and after a successful call to `fork()`.

[✓ True]

False

2 Explain.

`fork()` creates an exact copy of the process address space.

d) (2 points)

1 A system call is a function linked into a user-mode program that executes a "transition to kernel mode" instruction, then begins to execute arbitrary code with kernel privileges.

True

[✓ False]

2 Explain.

System calls utilize a single atomic operation to both (1) make a transition from user-mode to kernel-mode and (2) begin executing one of a small number of pre-determined handlers in kernel space.

e) (2 points)

1 A system call allows a process to execute certain instructions in the kernel's code in user mode.

True

[✓ False]

2 Explain.

System calls run predefined kernel system call handlers in kernel mode: the instructions are run in kernel mode.

f) (2 points)

1 Without system calls, user programs cannot intentionally transfer control from a user program to the kernel.

True

[✓ False]

2 Explain.

User programs could intentionally trap to transfer control to the kernel via an exception. (Comment: some students put `pthread_yield()`/`yield()`, but `pthread_yield()` calls `sched_yield()`, which is a syscall in Linux)

g) (2 points)

1 Without interrupts, the kernel cannot intentionally transfer control from a user program to the kernel.

[✓ True]

False

2 Explain.

The kernel would have to wait for user programs to voluntarily transfer control to the kernel.

h) (2 points)

1 One advantage of matching every user-level thread with a kernel thread is that the resulting user-level thread can execute arbitrary instructions in kernel mode simply by using a message channel to communicate with the kernel thread.

True

[✓ False]

2 Explain.

The presence of the kernel-thread/stack means that the corresponding user-mode thread can sleep in the kernel during I/O operations without blocking other threads.

i) (2 points)

1 Every user-level thread must have a unique kernel-level thread in order to facilitate multithreading (and allow concurrency).

True

[✓ False]

2 Explain.

It is possible for user-level threads to be multiplexed at user-level by a user-level library.

j) (2 points)

1 After a thread is put on the queue for a condition variable, the thread will eventually get to run until completion.

True

[✓ False]

2 Explain.

The thread may never get woken up if other threads crash/hang. Also the condition can always be false.

k) (2 points)

1 It is possible to create a critical section without disabling interrupts and without any locks, semaphores, or condition variables.

[✓ True]

False

2 Explain.

Atomic operations like compare&swap and test&set are implemented in hardware.

l) (2 points)

1 Spinlocks–threads that spin in a loop until the lock is released–always perform worse than traditional blocking locks.

True

[✓ False]

2 Explain.

In some cases, spinlocks can waste fewer cycles than putting a thread to sleep and waking them up. For example, if the time taken to acquire a lock is shorter than the time taken to put a thread to sleep and wake it up.

m) (2 points)

1 To create a critical section in a uniprocessor, the user can just disable interrupts to avoid being preempted.

True

[✓ False]

2 Explain.

The user cannot disable interrupts: the kernel has to do so.

n) (2 points)

1 Because `cond_wait()` releases the condition variable lock in its implementation, after the thread is woken up, the thread is explicitly responsible for calling `lock_acquire`.

True

[✓ False]

2 Explain.

`cond_wait()` reacquires the lock before returning.

o) (2 points)

1 When using PintOS lists, a struct can only have one `list_elem`.

○ True

[✓ False]

2 Explain.

The `struct thread` in Pintos has two list_elems, `allelem` and `elem`.

p) (2 points)

1 In PintOS, every user program will eventually call the exit syscall (assuming that the kernel does not kill the program due to a fault).

[✓ True]

○ False

2 Explain.

When `main` returns to `_start`, it issues an exit syscall.

q) (2 points)

1 Since PintOS doesn't have a `fork` syscall, the `exec` syscall implicitly spawns a new child process.

[✓ True]

○ False

2 Explain.

In `process_execute`, a new thread is created for the child.

r) (2 points)

1 In PintOS, if `open` is called twice on the same file in the process (without anything else happening in between), the two file descriptors can be used interchangeably (similar to what would happen if `dup()` was used to generate the second file descriptor).

○ True

[✓ False]

2 Explain.

Each `open` should return a new file descriptor that must be closed independently and do not share the same position.

s) (2 points)

1 In PintOS, the kernel stack can grow dynamically to handle deeply recursive computations.

○ True

[✓ False]

2 Explain.

The kernel stack must fit on a 4KiB page and cannot grow dynamically.

t) (2 points)

1 In PintOS, both user and kernel virtual memory is per process.

○ True

[✓ False]

2 Explain.

Kernel virtual memory is global.

u) (2 points)

1 In PintOS, when accessing an unmapped user virtual address, a user thread will page fault while a kernel thread will not.

○ True

[✓ False]

2 Explain.

A kernel will still page fault when accessing an unmapped user virtual address.

v) (2 points)

1 Say two processes on the same machine `open` the same file, and that there are no race conditions or errors in doing so – that is, `open` returns a valid file descriptor to both processes. If either process makes a call to `remove` this file from the filesystem, the other process cannot `write` to its file descriptor any longer.

○ True

[✓ False]

2 Explain.

No - the file will be available to other processes that have the file open. From the `man` pages:

> If the removed name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse. If the name was the last link to a file, but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.

w) (2 points)

1 Assume `test.txt` doesn't already exist.

```
char buf[BUFSIZE];
memset(buf, 'a', 100);
int fd = open("test.txt", O_CREAT|O_RDWR);
lseek(fd, BUFSIZE, SEEK_SET);
lseek(fd, 0, SEEK_SET);
read(fd, buf, BUFSIZE);
```

Afterwards, `buf` will contain BUFSIZE `a` characters.

[✓ True]

False

2 Explain.

`read` will return 0 since the file is empty.

x) (2 points)

1 After calling `exec`, a process can read from and write to the same file descriptors that the original process had open, even though the original process's address space has been replaced.

[✓ True]

False

2 Explain.

`exec` does not destroy the process's file descriptor table.

y) (2 points)

1 Calling `pipe` on an array like `int fds[2];` creates a read and write stream that local processes can use for unidirectional IPC. Under the hood, a pipe is implemented as a buffer in user space where the read end sits at the 0th index of the pipe and the write end sits at the 1st index of the pipe.

True

[✓ False]

2 Explain.

`pipe` creates two file descriptors, and pipes are a buffer in kernel space.

z) (2 points)

1 When using sockets, the server process spins on a `while (true)` loop, awaiting new connections. When a connection arrives, the request is dequeued, the server calls `accept` and creates a new socketed connection, which returns a descriptor referencing the socket that the server process uses to service the client's future requests. Both the server and the client have the responsibility of closing their socket descriptors.

○ True

[✓ False]

2 Explain.

The server process does no such spinning. It hangs on the call to `accept`, which has already been called by the time a request gets dequeued from the request queue.

aa) (2 points)

1 Each client hardware device (with a unique IP address), can have many individual sockets connected to the same server/port combination, such that the server can differentiate between them.

[✓ True]

○ False

2 Explain.

Each connection's 5-tuple allows this.

2. (16 points) Multiple Choice

a) (3 pt) How many new processes could be created? Assume at least 2 individual calls to fork() succeed.

```
#include <sys/types.h>
#include <unistd.h>

int main() {
    for (int i = 0; i < 3; i++) {
        fork();
    }
    return 0;
}
```

- ☐ 0
- ☐ 1
- 🟥 2
- 🟥 3
- 🟥 4
- 🟥 5
- 🟥 6
- 🟥 7
- ☐ 8

Every iteration of the loop has a call to `fork()` in every running instance of this process.

Least successful case: `fork()` when `i=0` fails. `fork()` when `i=1` succeeds, and then one of the two calls to `fork()` when `i=2` succeed. There are 2 successful fork calls (creating 2 new processes) out of 4 fork calls total.

Most successful case: Every individual call to fork succeeds, so the number of processes duplicates every iteration. 1+2+4=7 out of 7 calls to `fork()` are successful, for a total of 7 new processes.

b) (3 pt) How many times could fork() be called? Assume at least 2 individual calls to fork() succeed.

```
#include <sys/types.h>
#include <unistd.h>

int main() {
    for (int i = 0; i < 3; i++) {
        fork();
    }
    return 0;
}
```

- ☐ 0
- ☐ 1
- ☐ 2
- ☐ 3
- 🟥 4
- 🟥 5
- 🟥 6
- 🟥 7
- ☐ 8

Every iteration of the loop has a call to `fork()` in every running instance of this process.

Least successful case: `fork()` when `i=0` fails. `fork()` when `i=1` succeeds, and then one of the two calls to `fork()` when `i=2` succeed. There are 2 successful fork calls (creating 2 new processes) out of 4 fork calls total.

Most successful case: Every individual call to fork succeeds, so the number of processes duplicates every iteration. 1+2+4=7 out of 7 calls to `fork()` are successful, for a total of 7 new processes.

c) (2 pt) Which of the following need to be saved and restored on a context switch between two threads in the same process?

- 🟥 Computational registers (integer and floating point)
- ☐ Page-table root pointer
- ☐ Buffered data in streams opened with fopen
- ☐ Contents of the file descriptor table
- 🟥 Stack pointer register

Each thread has its own registers and stack, so we need to restore the computation registers and the stack pointer register. Threads in the same process share an address space and file descriptor table, so we do not need to save the page-table root pointer or the fd table. Buffered data is already in memory, so we do not need to save or restore that, either.

d) (2 pt) Which of the following are valid ways for two threads in the same process to communicate?

- [✓] Reading/writing to a memory address in one thread's stack
- [✓] Reading/writing to a memory address in the process's heap
- ☐ Reading/writing to the same location in one thread's kernel's stack
- [✓] Reading/writing to an open FILE*
- [✓] Reading/writing to a pipe
- [✓] Reading/writing to sockets

Each process has its own address space and threads within the same process share that address space, though user programs may not access the kernel stack. Files, pipes, and sockets are all handled by the kernel.

e) (2 pt) Which of the following are valid ways for two threads in different processes to communicate?

- ☐ Reading/writing to a memory address in one thread's stack
- ☐ Reading/writing to a memory address in the process's heap
- ☐ Reading/writing to the same location in one thread's kernel's stack
- [✓] Reading/writing to an open FILE*
- [✓] Reading/writing to a pipe
- [✓] Reading/writing to sockets

Each process has its own address space and threads within the same process share that address space, though user programs may not access the kernel stack. Files, pipes, and sockets are all handled by the kernel.

f) (2 pt) Which of the following are atomic operations?

- ☐ `x -= 100`
- ☐ `y++`
- [✓] `test&set`
- [✓] `cond_wait`
- ☐ `process_execute` (from Project 1!)
- ☐ `pthread_mutex_init`

`test&set` is an atomic instruction implemented in hardware. `cond_wait` atomically releases a lock/acquires a lock.

g) **(2 pt)** Which of the following are true about condition variables?

- 🟥 Allows for waiting in a critical section
- ☐ Are commutative just like semaphores
- ☐ Are sometimes subjected to spurious wakeup when dealing with Hoare semantics
- 🟥 Avoids busy waiting

Condition variables avoid busy waiting by allowing for the behavior of waiting inside of a critical section. Semaphores are commutative (you can `sema_down()` before `sema_up()`), while condition variables are not (you cannot `cond_signal()` before `cond_wait()`). In Hoarse semantics, control is transferred to the awoken thread immediately, forgoing spurious wakeups where the wait condition has not been satisfied.

h) **(2 pt)** Which of the following are false about condition variables?

- ☐ Allows for waiting in a critical section
- 🟥 Are commutative just like semaphores
- 🟥 Are sometimes subjected to spurious wakeup when dealing with Hoare semantics
- ☐ Avoids busy waiting

Condition variables avoid busy waiting by allowing for the behavior of waiting inside of a critical section. Semaphores are commutative (you can `sema_down()` before `sema_up()`), while condition variables are not (you cannot `cond_signal()` before `cond_wait()`). In Hoarse semantics, control is transferred to the awoken thread immediately, forgoing spurious wakeups where the wait condition has not been satisfied.

i) **(2 pt)** When setting up the user stack in PintOS, which of the following properties must be satisfied? Select all that apply.

- 🟥 The stack starts at the very top of the user virtual address space, in the page just below the virtual address `PHYS_BASE`.
- ☐ The stack pointer, esp, must be aligned to a 16-byte boundary after pushing the return address.
- ☐ The order of words (ex: `/bin/ls -l foo bar`) must be in the same order as they were inputted on the top of the stack.
- 🟥 The value of the return address doesn't have to be any specific value.

j) **(2 pt)** The kernel in PintOS must verify pointers before accessing user memory. Which of the following must be true about a valid pointer? Select all that apply.

- 🟥 The pointer is located in mapped virtual memory.
- ☐ The pointer is located above `PHYS_BASE`.
- ☐ The pointer can be a null pointer.
- ☐ None of the Above

Null pointers are not valid. Pointers above `PHYS_BASE` aren't valid either, because user programs are not allowed to access kernel memory.

k) (2 pt) In Project 1 you had to implement the wait syscall. A common implementation was to create some shared struct with a reference count, or `ref_cnt`, that was discussed in section. What are some of the properties of this `ref_cnt`? Select all that apply.

- ☐ The `ref_cnt` can grow and shrink depending on the number of children a parent thread can have.
- 🟥 The `ref_cnt` must be synchronized with some sort of lock as different threads can concurrently read and write to it.
- 🟥 The `ref_cnt` could start at 2, to represent the starting relationship from the parent and from the child.
- ☐ None of the Above

`ref_cnt` does not track how many children a thread has. The other options are true.

l) (3 pt) Assume all calls to `read()`, `write()`, and `fork()` succeed. Suppose that `montymole.txt` was empty before running this block of code. The following code was run:

```
int main(int argc, char** argv) {
    int fd1 = open("montymole.txt", O_WRONLY);
    int fd2 = open("montymole.txt", O_WRONLY);
    write(fd1, "mole", 4);
    write(fd2, "whack", 5);
    write(fd2, "mole", 4);
    write(fd1, "mole", 4);
    write(fd1, "mole", 4);
    close(fd1);
    close(fd2);
}
```

Which of the following could be the content of "montymole.txt"?

- 🟥 `whacmolemole`
- ☐ `whacmolek`
- ☐ `molewhackmolemolemole`
- ☐ `whackmolemole`
- ☐ Nothing

In this problem, the two calls to `open()` create two open file descriptions, which have independent file offsets. Going through each write call, the file looks like: mole -> whack -> whackmole -> whacmolee -> whacmolemole

m) (3 pt) Assume all calls to `read()`, `write()`, and `fork()` succeed. Suppose that `montymole.txt` was empty before running this block of code. The following code was run:

```
void new_thread(void* arg) {
    int* fd = (int*) arg;
    write(*fd, "whackwhackmole", 14);
}
int main(int argc, char** argv) {
    int fd1 = open("montymole.txt", O_WRONLY);
    pthread monty_mole;
    pthread_create(&monty_mole, NULL, new_thread,(void*) &fd1);
    write(fd1, "mole", 4);
    close(fd1);
}
```

Which of the following could be the content of "montymole.txt"?

- [✓ `molewhackwhackmole`]
- [✓ `whackwhackmolemole`]
- ☐ `molewhackwhackwhack`
- ☐ `molewhack`
- ☐ `whackwhackmole`

In this problem, both the main thread and the created thread write to the same fd, so both threads share a file offset. Their writes will append to the file instead of overwriting the file.

n) (3 pt) Assume all calls to `read()`, `write()`, and `fork()` succeed. Suppose that `montymole.txt` was empty before running this block of code. The following code was run:

```
void fork_p(pid_t pid, int fd1) {
    if (pid == 0) {
        write(fd1, "whack", 5);
    }
    else {
        write(fd1, "mole", 4);
    }
}
int main(int argc, char** argv) {
    int fd1 = open("montymole.txt", O_WRONLY);
    fork_p(fork(), fd1);
    write(fd1, "mole", 4);
    close(fd1);
}
```

Which of the following could be the content of "montymole.txt"?

- ☐ `molekmole`
- ☐ `whacmmole`
- ☐ `whackmole`
- ☐ `molemolee`
- ☐ `whackmolemole`

In this problem, both the main process and the forked process write to the same file, so they share a file offset. Their writes will append to the file instead of overwriting the file.

o) (2 pt) Which of the following are true about files?

- 🟥 We can write to any file (with write permissions) using `printf`.
- ☐ If I wanted to share a file descriptor with another thread, I would have to pass it as an argument because the new thread doesn't know the file descriptor of that file.
- ☐ If I wanted to share a `FILE*` with another thread, I would have to pass it as an argument because it doesn't have access to the `FILE*`.
- 🟥 If I wanted to share a `FILE*` with a child process, I wouldn't have to do anything because the child has access to the `FILE*` as a result of the fork.
- ☐ None of the Above

Duping stdout to a file will allow `printf` to write to the file. Sharing information between threads of the same process does not necessarily need to be done through passing arguments; you can have data in a global variable, which both threads can access.

p) (2 pt) Which of the following scenarios are valid (true and/or correct)?

☐ In Thread A, I make a new file and make a new thread, Thread B. I close the file in Thread B but will still be able to write to the new file in Thread A.

[✓] If I make a new process (using `fork()`) to serve client requests in a web server, I can close both the socket and client connection in the child process.

[✓] Two processes can communicate with each other using sockets.

☐ The `fdopen()` library function takes in a file descriptor and returns a file pointer to the same file. The resulting FILE* can have more permissions than requested with the original `open()` syscall for the file descriptor.

☐ None of the Above

Thread A and Thread B are in the same process and thus share a file descriptor table, so closing the file in one thread will close it for the other. Closing the socket connection, serving the client, then closing the client connection in the child is a valid way to serve client requests (see Server Protocol v2 in lecture slides).

3. (24 points) Short Answer

a)

```
 1. Reader() {
 2. //First check self into system
 3. lock.acquire();
 4. while ((AW + WW) > 0) {
 5.     WR++;
 6.     okToRead.wait(&lock);
 7.     WR--;
 8. }
 9. AR++;
10. lock.release();
11.
12. // Perform actual read-only access
13. AccessDatabase(ReadOnly);
14.
15. // Now, check out of system
16. lock.acquire();
17. AR--;
18. if (AR == 0 && WW > 0)
19.     okToWrite.signal();
20. lock.release();
21. }
22.
23. Writer() {
24. // First check self into system
25. lock.acquire();
26. while ((AW + AR) > 0) {
27.     WW++;
28.     okToWrite.wait(&lock);
29.     WW--;
30. }
31. AW++;
32. lock.release();
33.
34. // Perform actual read/write access
35. AccessDatabase(ReadWrite);
36.
// Now, check out of system
lock.acquire();
AW--;
if (WW > 0){
    okToWrite.signal();
} else if (WR > 0) {
    okToRead.broadcast();
}
lock.release();
}
```

1 Above, we show the Readers-Writers example given in class. It used two condition variables, one for waiting readers and one for waiting writers. Suppose that all of the following requests arrive in very short order (while R1 and R2 are still executing):

`Incoming stream: R1,R2,W3,R4,W5,W6,R7,R8,R9,W10,R11,W12,W13,R14,R15,W16,R17`

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue).

`{R1, R2}, W3, W5, W6, W10, W12, W13, W16, {R4, R7, R8, R9, R11, R14, R15, R17}`

2 Above, we show the Readers-Writers example given in class. It used two condition variables, one for waiting readers and one for waiting writers. Suppose that all of the following requests arrive in very short order (while R1 and R2 are still executing):

`Incoming stream: R1,R2,R3,W4,W5,R6,W7,R8,W9,R10,W11,W12,R13,R14,R15,W16,R17`

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue).

`{R1, R2, R3}, W4, W5, W7, W9, W11, W12, W16, {R6, R8, R10, R13, R14, R15, R17}`

3 Above, we show the Readers-Writers example given in class. It used two condition variables, one for waiting readers and one for waiting writers. Suppose that all of the following requests arrive in very short order (while R1 and R2 are still executing):

`Incoming stream: R1,R2,R3,R4,W5,W6,W7,W8,R9,R10,R11,W12,R13,W14,R15,W16,R17`

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue).

`{R1, R2, R3, R4}, W5, W6, W7, W8, W12, W14, W16, {R9, R10, R11, R13, R15, R17}`

4 Above, we show the Readers-Writers example given in class. It used two condition variables, one for waiting readers and one for waiting writers. Suppose that all of the following requests arrive in very short order (while R1 and R2 are still executing):

`Incoming stream: R1,R2,R3,R4,R5,W6,W7,W8,W9,R10,R11,W12,W13,R14,W15,R16,R17`

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue).

```
{R1, R2, R3, R4, R5}, W6, W7, W8, W9, W12, W13, W15, {R10, R11, R14, R16, R17}
```

5 Above, we show the Readers-Writers example given in class. It used two condition variables, one for waiting readers and one for waiting writers. Suppose that all of the following requests arrive in very short order (while R1 and R2 are still executing):

`Incoming stream: R1,R2,R3,W4,W5,W6,W7,R8,R9,W10,R11,W12,R13,W14,R15,R16,R17`

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue).

```
{R1, R2, R3}, W4, W5, W6, W7, W10, W12, W14, {R8, R9, R11, R13, R15, R16, R17}
```

6 Above, we show the Readers-Writers example given in class. It used two condition variables, one for waiting readers and one for waiting writers. Suppose that all of the following requests arrive in very short order (while R1 and R2 are still executing):

`Incoming stream: R1,R2,W3,W4,R5,W6,W7,W8,R9,W10,R11,R12,R13,R14,R15,R16,W17`

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue).

```
{R1, R2}, W3, W4, W6, W7, W8, W10, W17, {R5, R9, R11, R12, R13, R14, R15, R16}
```

7 Above, we show the Readers-Writers example given in class. It used two condition variables, one for waiting readers and one for waiting writers. Suppose that all of the following requests arrive in very short order (while R1 and R2 are still executing):

`Incoming stream: R1,R2,R3,R4,R5,W6,W7,W8,R9,W10,W11,W12,R13,R14,R15,W16,R17`

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue).

```
{R1, R2, R3, R4, R5}, W6, W7, W8, W10, W11, W12, W16, {R9, R13, R14, R15, R17}
```

8 Explain how you got your answer.

Since this algorithm gives precedence to writes over reads, although the first few reads get to execute together and right away, the remaining reads are deferred until each write is processed (one at a time). After all writes are finished, then the reads execute as a group (unordered).

9 **(3 pt)** Let us define the logical arrival order by the order in which threads first acquire the monitor lock. Suppose that we wanted the results of reads and writes to the database to be the same as if they were processed one at a time – in their logical arrival order; for example:

`Incoming stream: W1,W2,R3,R4,R5,W6,W7,R8,R9,R10`

Would be processed as: `W1,W2,{R3,R4,R5},W6,W7,{R8,R9,R10}`

regardless of the speed with which these requests arrive. Explain why the above algorithm does not satisfy this constraint.

1 Once it starts buffering items (i.e. putting requests to sleep on condition variables), the above algorithm does not keep track of the arrival order. Or, to say it another way, the above algorithm loses all ordering between buffered reads and buffered writes. Thus, it cannot guarantee that reads execute after earlier writes and before later writes (as defined by logical arrival order) unless it never buffers items (in which case it is not really doing anything at all).

2 There's only 2 wait queues–the one for readers and the one for writers–so if the requests come in too quickly, there is no way to divide requests into more than 2 groups since they are just sorted into reads and writes.

Common mistakes:

- Many just mentioned priority towards writers being the cause of why the stated algorithm doesn't process in logical arrival order, but did not elaborate more on how that doesn't satisfy the given constraint (i.e since writes are prioritized, writes will be allowed to write first even though there may be reads that arrived earlier, violating the logical arrival order). The algorithm specifically doesn't work because it doesn't keep track of arrival order.
- Some students also mentioned that `broadcast()` does not guarantee an order because it allows all readers in the queue to read, so they may not be processed in logical arrival order. However, because the question groups all readers with braces, the question is already stating that reads can be unordered. Thus, explaining that reads among groups may be unordered due to `broadcast()` was not a valid explanation for the question.

The following program is run:

b)
```
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <sys/wait.h>

void *helper(void *arg) {
    int *number = (int *) arg;
    (*number)++;
    return NULL;
}

int main(int argc, char *argv[]) {
    int data = 0;
    pthread_t thread;
    pthread_create(&thread, NULL, helper, (void *) &data);
    pthread_join(thread, NULL);
    printf("data is %d\n", data);
    data = 0;
    pid_t cpid = fork();
    if (cpid == 0) {
        helper((void *) &data);
        return 0;
    } else {
        wait(NULL);
    }
    printf("data is %d\n", data);
}
```

1 (2 pt) What does this program output? (There should be 2 lines of output.)

```
DATA_HP is 1
DATA_HP is 0
```

2 (2 pt) Explain why the two lines of output are either different or the same.

The `pthread_create()` line will increment `DATA_HP` by 1 in both the helper thread and the main thread because the code passes in a pointer to `DATA_HP`. Since the helper thread has reference to the same `DATA_HP` variable the main thread has access to, changing `DATA_HP` in the helper thread will change the main thread's `DATA_HP` as well. After the `fork()` call, the child process will increment only its own `DATA_HP` variable because processes have their own address space; changing something in the child process will not cause changes in the parent process. Thus, the parent's `DATA_HP` variable stays at 0.
Common mistakes:

- `DATA_HP` is not a global variable and it is not on the heap; it is on the stack.
- Threads do not share stacks, but they can access each other's stacks.
- It is not enough to say the main thread calls `pthread_join()` on the helper thread, so after the helper thread increments `DATA_HP`, the main thread can print 1. If a pointer to `DATA_HP` was not an argument to `pthread_create()`, even if `pthread_join()` was called, the output would not be 1. For this question, it is specifically because the helper thread has a reference to `DATA_HP`, that the main thread prints 1.

c) (2 pt) Assume that you are using a monitor for synchronizing some shared data. Explain why you might want to wrap a while loop around a `cond_wait()` statement even in a system with **Hoare** scheduling.

A couple of reasons could work: 1. Your signal statement is more general than the wait condition. Example: in readers/writers you signal/broadcast any thread (such as a reader OR writer) but the wait condition must make sure that a write is allowed to proceed. 2. There is the possibility for the condition variable to wake up spuriously so that you must check the condition again to be sure that you can proceed.
Common mistake: While external events like interrupts can cause context switch after the waiting thread wakes up, it still has the lock for the conditional variable. Thus, other threads won't be able to modify it.

d) (2 pt) Explain why **Hoare** scheduling for a monitor might result in less efficient use of the processor cache than **Mesa** scheduling.

Because the signal semantics in Hoare scheduling require more context switching than Mesa scheduling (a signal forces two context switches – to the signaled thread and then back to the signaler). This extra context switching can cause more cache misses.
Common mistake: Just stating there is overhead from more context switching does not explain how processor cache is affected.

e) (2 pt) Assume that you are using a monitor for synchronizing some shared data. Explain why you need a while loop around a check for a condition that puts a thread to sleep on a condition variable when using **Mesa** scheduling.

You need to use a while loop to recheck the condition with Mesa scheduling because a signal statement merely places the signaled thread on the ready queue. By the time that this thread gets to run, another thread may intervene and change the condition again.

f) (2 pt) Assume that you are using a monitor for synchronizing some shared data. Explain why you usually do not need a while loop around a check for a condition that puts a thread to sleep on a condition variable when using **Hoare** scheduling.

In Hoare scheduling, the signaling thread yields the processor and lock to the signaled thread. As a result, under normal usage, the signaled thread can immediately run since no intervening threads could have altered the conditions between when the signaler signals and the signalee runs.

g) (2 pt) Explain in 1-2 *short* sentences when we would **NOT** need to incorporate reference counts in structs that are shared between processes in PintOS.

Structs that are stored on the stack or in static memory don't need reference counts, since we don't need to free them. Take the load status struct for example. This struct only needs to persist across calls `process_execute -> thread_create -> start_process` and is safely stored on the stack since even if one of the threads that are sharing it exit early, the other won't be stuck with garbage memory.

h) (2 pt) John (M) and Frank decide the implementation of threads and processes in PintOS is too trivial. They are now interested in adding the ability to create multithreaded processes. Recommend to them whether they should base their implementation on calling functions from the `pthread` library (i.e. `pthread_create`) or take an alternative approach. Defend your recommendation in no more than 3 sentences.

Anything but using the `pthread` library, which in some sense is just an API for creating and managing user threads. Implementing threads in PintOS means actually seeing all the gory details of an internal API (`struct thread`, `thread.c`, etc.), not using an existing frontend. In addition, the `pthread` library does not directly support creating or managing kernel threads (obviously, since it isn't a kernel implementation).
Consider what `pthread_create` would do when run in PintOS userspace. Conceptually, it would end up making a syscall whose handler's implementation would look similar to the `exec` flow: `process_execute -> thread_create -> start_process` (implementation details omitted because `pthread` does different things on different distributions). Of course, you would need a new implementation of PintOS threads and processes for the syscall to work, but surely if something needs to syscall into the kernel to implement a kernel feature, it is not a valid solution.

i) (2 pt) In 3-4 sentences, explain how you would implement a `fork()` syscall in PintOS. Specifically, discuss how you would implement two different return values from the same system call.

Create a `fork` callback in `syscall_handler` in `userprog/syscall.c` that clones the current thread's address space. This could be implemented by making a new helper called `thread_copy` or something of the like with very similar structure as `thread_create`, including running the new kernel thread on `start_process`. But, the important part is that within the `start_process` function, we can directly modify the `if_` variable's value for `eax` to be whatever we want: 0 in the child and some other value in the parent, which can be implemented by adding variables to `struct thread`, adding additional arguments to `process_execute`, etc.

j) (2 pt) In section 1, we reviewed how the syscall handler protects the kernel from corrupt or malicious user code. Here was the solution:

```
1. The syscall number is used to index into a vector mapping number of fixed syscall handlers
-- this prevents the user from getting the kernel to run user code in kernel mode.
2. The handler copies the user arguments from the user registers or stack onto the kernel
stack -- this protects the kernel from malicious code on the user stack from evading
checks.
3. All of the arguments are validated before the syscall is executed -- this protects the
kernel from errors in the user arguments like invalid or null pointers.
4. Results from the syscall are copied back into user memory so the caller has access to
them.
```

Discuss why steps 2 and 3 can or cannot be switched in PintOS in no more than 3 sentences.

They cannot be switched. Otherwise, we run the risk of allowing user programs to panic the kernel, which is unacceptable. (Linux handles this by returning an `-EFAULT` and PintOS by (like was seen in proj0) throwing a page fault with rights violation.)
Clarification: step 3 does not include execution of the syscall.

k) (2 pt) Describe two non-trivial differences between a function that reads in an entire file into a buffer using `fread` vs. `read`.

1 `read` needs to be called in a loop
2 `fread` is buffered

l) (2 pt) Neil is working on HW2 and is implementing the pipes portion of the homework. As part of his implementation, he has decided that the standard output of a parent process will be the standard input to its child process. Neil first calls `fork()`, then the child process calls `pipe()` to facilitate IPC with the parent process. Is this a good idea or not? Why?

No, this isn't a good idea. This is due to the fact that `fork()` will copy over file descriptors and the address space of a process so both the parent and child process will have access to the pipes. This is from the underlying concept that `fork()` copies over file descriptors as well.

m) (2 pt) Suppose Taj was coding a multithreaded echo server (similar to the one in Section 4). He is making a new thread to serve a new client request, and thinks it would be a good idea to pass in the `server_socket` and the `client_socket` to the new thread. Then, once the thread is done resolving the request, the thread would close the `server_socket` as well as the `client_socket`. Is this a good idea? Explain your reasoning.

This isn't a good idea, because sockets are shared across threads (just like file descriptors). Closing the server socket will mean we can no longer accept new client connections in our main thread.

n) (2 pt) When would it be better to use threads over processes to serve client requests?

- When performance is important, the cost and overhead of launching a thread is much less than that of a process. So, when there is a potential for a high rate of incoming client requests, this could be an important distinction. Overhead from switching between threads versus processes is also different.
- Additionally, in the case where multiple threads are needed, it is easier for threads to communicate between each other (in comparison to processes) because threads have access to each others' stacks and share the heap and global variables.

o) (2 pt) When would it be better to use processes over threads to serve client requests?

It would be better when security/isolation is more important than the increased overhead from launching new processes. One example use case could be a small number of client requests that creates a virtual environment for the user to run arbitrary code.

p) (2 pt) How does a web server keep communication with various simultaneous clients separate (so that responses do not get interleaved)?

Unique 5-tuple of source/dest, ip/port, and protocol, specifically the source ip and port.

4. (22 points) Synchronization Primitives (Or Not?)

Recall from lecture that disabling interrupts is a valid way to gain mutual exclusion to the processor and hence all data in RAM on a uniprocessor system. However, disabling interrupts around critical sections comes with some downsides.

a) **(2 pt)** Explain two downsides of disabling interrupts around critical sections on a uniprocessor.

1 Process holding lock may not release for a long time, effectively halting the machine.
2 Can deadlock if program holds the lock and waits for some I/O, which requires an interrupt.
3 Does not maintain the "acquired" state of the lock across a context switch (`yield()`)

b) To solve this problem, we use disabling interrupts sparingly and leverage the ability to put threads to sleep while they wait for access to the critical section. Given functions intr_disable() and intr_enable(), which disable interrupts and enable interrupts, respectively, implement a semaphore that (a) does not busy wait on sema_down operations and (b) does not keep interrupts disabled for a large number of operations. You are also allowed to use a PintOS list in your solution as well as the following functions:

- `thread_current()`: gives a reference to the current thread. Assume this thread has an "elem" list element that can be put onto PintOS lists
- `thread_block()`: blocks the current thread and chooses a new thread to run
- `thread_block_and_intr_enable()`: blocks the current thread, chooses a new thread to run, and then enables interrupts
- `thread_unblock(thread)`: unblocks the given thread and puts it on the ready-queue

The solution for these questions requires no more than 6 lines each. You may find the PintOS function signatures at the end of the exam useful.

1 **(1 pt)**

```
typedef struct semaphore {
    ----------
    ----------
} sema_t;
```

```
unsigned int value;
struct list waiters;
```

2 **(1 pt)**

```
void sema_init(sema_t *sema, int value) {
    ----------
    ----------
}
```

```
sema->value = value;
list_init(&sema->waiters);
```

3 (2 pt)

```
void sema_down(sema_t *sema) {
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
}
```

```
intr_disable();
while (sema->value == 0) {
    list_push_back(sema->waiters, &thread_current->elem);
    thread_block_and_intr_enable();
}
sema->value--;
intr_enable();
```

4 (2 pt)

```
void sema_up(sema_t *sema) {
    ----------
    ----------
    ----------
    ----------
    ----------
}
```

```
intr_disable();
sema->value++;
if (!list_empty(&sema->waiters))
    thread_unblock(list_entry(list_pop_front(&sema->waiters), struct thread, elem));
intr_enable();
```

Semaphores are a powerful synchronization primitive. They can be used to implement both locks and condition variables. See the implementation of locks below:

c)
```
typedef struct lock {
    sema_t sema;
} lock_t;

void lock_init(lock_t *lock) {
    sema_init(&lock->sema, 1);
}

void lock_acquire(lock_t *lock) {
    sema_down(&lock->sema);
}

void lock_release(lock_t *lock) {
    sema_up(&lock->sema);
}
```

Semaphores can also be used to implement condition variables (and therefore monitors)! Implement condition variables via semaphores below. For simplicity, we allow you to assume that semaphores can be added to PintOS lists.

1 (1 pt)

```
typedef struct condition_variable {
    ----------
} cv_t;
```

```
struct list waiters;
```

2 (1 pt)

```
void cond_init(cv_t *cv) {
    ----------
}
```

```
list_init(&cv->waiters);
```

3 (2 pt)

```
void cond_wait(cv_t *cv, lock_t *lock) {
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
}
```

```
sema_t sema; // Signaling semaphore
sema_init(&sema, 0);
list_push_back(&cv->waiters, &sema->elem);
lock_release(lock);
sema_down(&sema);
lock_acquire(lock);
```

4 (2 pt)

```
void cond_signal(cv_t *cv, lock_t *lock) {
    ----------
    ----------
}
```

```
if (!list_empty(&cv->waiters))
    sema_up(&list_entry(list_pop_front(&cv->waiters), sema_t, elem));
```

5 (2 pt)

```
void cond_broadcast(cv_t *cv, lock_t *lock) {
    ----------
    ----------
}
```

```
while (!list_empty(&cv->waiters))
    cond_signal(cv, lock);
```

We have now implemented all the synchronization primitives from enabling and disabling interrupts, where interrupts are not disabled for too long and the processor does not busy wait! You did it! You now have the ability to write synchronized data structures that are thread-safe!

Let us now forget everything we just did, and implement a data structure without synchronization primitives. We will implement a thread-safe queue via a circular array buffer without any synchronization primitives at all. For simplicity, assume that the following operations never occur:

d)

- Enqueueing when the queue is already full
- Dequeuing when the queue is empty

Consider the single-threaded implementation of the queue via a circular array buffer:

```
# define MAX_SIZE 128

typedef struct circular_array_buffer {
    int FRONT;
    int BACK;
    int values[MAX_SIZE];
} queue_t; // U r a queue_t \pi

void queue_init(queue_t *queue) {
    queue->FRONT = queue->BACK = 0;
    for (int i = 0; i < MAX_SIZE; i++)
        queue->values = 0;
}

void queue_push(queue_t *queue, int value) {
    queue->values[queue->FRONT++ % MAX_SIZE] = value;
}

int queue_pop(queue_t *queue) {
    return queue->values[queue->BACK++ % MAX_SIZE]
}
```

If multiple threads access this data structure concurrently, there will (clearly) be race conditions. We can solve these race conditions by locking accesses to the queue and granting mutual exclusion. In this case, in the presence of contention, one or more threads would sleep while another thread is in the critical section. Instead, we implement a non-blocking, thread-safe version of the queue without synchronization primitives.

Armed with only the `compare&swap` and `compare&swap2` atomic operations (defined below) implement the `queue_push` and `queue_pop` operations to be thread-safe.

```
// Typical compare&swap operation
compare&swap(&addr, reg1, reg2)
    if (M[addr] == reg1)
        M[addr] = reg2
        return success
    else
        return failure

// Special compare&swap operation that allows two memory updates
compare&swap2(&addr1, reg1, reg2, &addr2, reg3)
    if (M[addr1] == reg1)
        M[addr1] = reg2
        M[addr2] = reg3
        return success
    else
        return failure
```

1 (3 pt)

```
void queue_push(queue_t *queue, int value) {
    ----------
    ----------
    ----------
    ----------
    ----------
}
```

```
// If the queue front hasn't changed while we were working,
// perform updates, otherwise try again
do {
    int front = queue->front;
    int *front_ptr = &queue->values[front];
    int new_front = (front + 1) % MAX_SIZE;
} while (!compare&swap2(&queue->front, front, new_front, new_front_pointer, value));
```

2 (3 pt)

```
int queue_pop(queue_t *queue) {
    ----------
    ----------
    ----------
    ----------
    ----------
}
```

```
// If the queue front hasn't changed while we were working,
// perform updates, otherwise try again
int value;
do {
    int back = queue->back;
    value = queue->values[back];
    int new_back = (back + 1) % MAX_SIZE;
} while (!compare&swap(&queue->back, back, new_back));
return value;
```

5. (16 points) PintOS Pipes

In this question you will be implementing support for a highly-simplified version of pipes within PintOS. Specifically, your pipes implementation will be simplified in the following ways:

- Each process has **exactly** one pipe.
- A process may only write to a pipe belonging to one of its direct children, using the `write_pipe` syscall.
- A process may only read from its own pipe, using the `read_pipe` syscall.
- Attempting to read from an empty pipe will result in the process being blocked until data is available (this is also true of normal Linux pipes). However, writing to a full pipe **should not** block - for the sake of keeping this question short, we are assuming a process will never write more data to a pipe than the pipe has capacity for.
- Pipes are not given file descriptors (hence the dedicated syscalls), and as such none of the conditions surrounding what happens if a write/read descriptor for a pipe is still open apply to this problem.

a) Part 1: Maintaining pipe state

Fill in the blanks below with whatever additional state you need to implement support for pipes. (Hint: some of this should be quite similar to what you did in project 1 in order to implement the `wait` syscall)

1 (2 pt) Inside `threads/thread.h`:

```
#define PIPE_BUFFER_SIZE 256

/*
 * Assume the pipe struct is properly reference counted
 * and therefore properly freed, in a manner similar to project 1
 * and the example shown with "Shared Data" in Section 2.
 *
 * Also assume that adding a child's pipe to its parent's
 * child_pipes list is taken care of for you.
 */
struct pipe {
    /* Reference count handling fields (not shown) */
    tid_t tid;
    uint8_t buffer[PIPE_BUFFER_SIZE];
    struct list_elem elem; /* List element for child_pipes */
    ----------
    ----------
    ----------
    ----------
};
```

```
Semaphore solution:


struct semaphore read_sema;
struct lock lock;
size_t size;


Monitor solution:


struct condition read_cv;
struct lock lock;
size_t size;
```

2 (1 pt) Inside `threads/thread.h`:

```
struct thread {
    tid_t tid;
    struct list child_pipes;
    /* Fields irrelevant to this problem (not shown) */
    ----------
};
```

```
struct pipe* pipe;
```

3 (3 pt) Inside `userprog/process.c`:

Fill in the below blanks with code that initializes the state that you added above.

```
static void start_process(/* Omitted */) {
    struct thread* cur = thread_current();
    /* Initialize interrupt frame, load executable, initialize
     * other state in thread struct (not shown).
     */
    ----------
    ----------
    ----------
    ----------
}
```

```
Semaphore solution:


cur->pipe = malloc(sizeof *cur->pipe);
cur->pipe->size = 0;
sema_init(&cur->pipe->read_sema, 0);
lock_init(&cur->pipe->lock);


Monitor solution:


cur->pipe = malloc(sizeof *cur->pipe);
cur->pipe->size = 0;
cond_init(&cur->pipe->read_cv);
lock_init(&cur->pipe->lock);
```

Part 2: Implementing the `read_pipe` and `write_pipe` syscalls

Assume that below functions are called when the `read_pipe` and `write_pipe` syscalls are made, respectively. **The below functions should work even if the `read_pipe` and `write_pipe` syscalls are made multiple times during a thread's lifetime.**

Inside `userprog/syscall.c`:

b)
```
/*
 * Kills the current thread if pointer is not a valid pointer
 * within the user address space
 */
static void verify(void* pointer) {
    /* Code not shown */
}

/*
 * Returns the pipe belonging to child_tid if child_tid the tid
 * of a child of the current process, and returns NULL otherwise
 */
static struct pipe* helper(tid_t child_tid) {
    /* Code not shown */
}
```

1 (6 pt)

```
/*
 * If the pipe is empty, this should block until data is available.
 * Otherwise it should read up to `length` bytes from the current thread's
 * pipe. If there are fewer than `length` bytes available, but more than zero,
 * read them all, but do not wait for more data to become available
 *
 * For example if there are 20 bytes of data currently in the pipe, and
 * length is given as 50, this function should read in the 20 bytes from the pipe
 * and then return (i.e. do not wait around for 50 bytes to become available)
 *
 * Should return the number of bytes read
 */
static int sys_read_pipe(uint8_t* buffer, unsigned length) {
    verify(buffer);
    verify(buffer + length - 1);
    struct thread* cur = thread_current();
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
}
```

```
Semaphore solution:


struct pipe* pipe = cur->pipe;
lock_acquire(&pipe->lock);
while (pipe->size == 0) { // Needs to be while, not if
    lock_release(&pipe->lock);
    sema_down(&pipe->read_sema);
    lock_acquire(&pipe->lock);
}
unsigned read_length = length > pipe->size ? pipe->size : length;
memcpy(buffer, pipe>buffer, read_length);
pipe->size -= read_length;
memmove(pipe->buffer, pipe->buffer + read_length, pipe->size);
lock_release(&pipe->lock);
return read_length;


Monitor solution:


struct pipe* pipe = cur->pipe;
lock_acquire(&pipe->lock);
```

```
while (pipe->size == 0) { // Needs to be while, not if
    cond_wait(&pipe->read_cv, &pipe->lock);
}
unsigned read_length = length > pipe->size ? pipe->size : length;
memcpy(buffer, pipe>buffer, read_length);
pipe->size -= read_length;
memmove(pipe->buffer, pipe->buffer + read_length, pipe->size);
lock_release(&pipe->lock);
return read_length;
```

2 (4 pt)

```
/*
 * Writes `length` bytes from `buffer` to thread `child_tid`'s pipe
 * Assume that there is always enough space in the pipe's buffer
 * to write `length` bytes without overflowing it.
 */
static int sys_write_pipe(tid_t child_tid, uint8_t* buffer, unsigned length) {
    verify(buffer);
    verify(buffer + length - 1);
    struct thread* cur = thread_current();
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    ----------
    return 0;
}
```

```
Semaphore solution:


struct pipe* = find_pipe_for_child(child_tid);
if (pipe == NULL)
    thread_exit();
lock_acquire(&pipe->lock);
memcpy(pipe->buffer + pipe->size, buffer, length);
pipe->size += length;
sema_up(&pipe->read_sema);
lock_release(&pipe->lock);


Monitor solution:


struct pipe* = find_pipe_for_child(child_tid);
if (pipe == NULL)
    thread_exit();
lock_acquire(&pipe->lock);
memcpy(pipe->buffer + pipe->size, buffer, length);
pipe->size += length;
cond_signal(&pipe->read_cv, &pipe->lock);
lock_release(&pipe->lock);
```

6. Reference Sheet

```
/******************************* Threads *******************************/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
const pthread_mutexattr_t *restrict attr);
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);

/****************************** Processes *******************************/
pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);

/*************************** High-Level I/O ****************************/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *stream);

/******************************** Sockets *******************************/
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, structure sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);

/**************************** Low-Level I/O *****************************/
int open(const char *pathname, int flags);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);

/******************************* PintOS ********************************/
void list_init(struct list *list);
struct list_elem *list_head(struct list *list)
struct list_elem *list_tail(struct list *list)
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);
void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
```