Fill out the bottom of this page with your computing ID.
Write your computing ID at the top of each page in case pages get separated.

On my honor as a student I have neither given nor received aid on this exam.

---

1. (6 points) In a typical POSIX-like operating system, which of the following operations are likely to require making a system call? **Select all that apply.**

   ◯ moving a file from one directory to another

   ◯ converting an integer to a string in preparation for outputting it

   ◯ creating a new process

   ◯ reading the updated version of a value currently only stored in another core's cache

   ◯ switching away from a program that has been running for too long

   ◯ acquiring a lock while no other thread has or is trying to acquire the lock

2. (4 points) Suppose an xv6 system is running ~~three~~ two (single-threaded) processes A and B. Process A executes a system call, and during that system call, it context switches (via the scheduler) to process B. Immediately after it is context switched to, process B executes, in kernel mode, code that forms part of the timer interrupt handling.

   Which of the following is true in this scenario? **Select all that apply.**

   ◯ even after the context switch, process A's kernel stack will most likely contain information about where the system call occurred

   ◯ the saved user stack pointer value in process A could be the same as the saved user stack pointer value in process B

   ◯ the xv6 kernel performed an I/O-like operation to trigger the timer interrupt from A's system call implementation

   ◯ since xv6 switched from process A to process B during a system call, process A's system call most likely did something to make process B runnable, such as writing to a pipe B was reading from

3. (4 points) Decreasing the length of time quanta for a round robin scheduler most likely _____. **Select all that apply.**

   ◯ increases the number of context switches that occur

   ◯ increases the amount of time between when a program becomes runnable after I/O and when it first runs

   ◯ increases the amount of storage required to implement the scheduler

   ◯ increases throughput

4. (4 points) Suppose one wanted to use a lottery scheduler to approximate SRTF (shortest remaining time first). One strategy for doing this would be to measure the amount of time a thread ran from when the thread became runnable after finishing I/O until when the thread next started I/O. Which of the following ways of setting the number of tickets each thread would have based on the running time measurements would **best** approximate SRTF?

   ◯ proportional to the inverse of the measured amount of time, so a thread with twice of the amount of time has half the number of tickets

   ◯ by ordering the threads in descending order of the amount of time, then assigning a million times as many tickets to the first thread as the second, and a million times as many to the second as the third and so on

   ◯ proportional to the measured amount of time, so a thread with twice the amount of time has twice the number of tickets

   ◯ by ordering the threads in ascending order of the amount of time, then assigning a million times as many tickets to the first thread as the second, and a million times as many to the second as the third and so on

5. Consider the following program:

```
#include <unistd.h>

char c = 'Z';
int main() {
    int pipe_fds[2];
    pipe(pipe_fds);
    pid_t pid = fork();
    if (pid == 0) {
        dup2(pipe_fds[1], STDOUT_FILENO);
    } else {
        dup2(pipe_fds[0], STDIN_FILENO);
        /* LOCATION ONE */
    }
    write(STDOUT_FILENO, "A", 1);
    read(STDIN_FILENO, &c, 1);
    write(STDERR_FILENO, &c, 1);
    return 0;
}
```

For the questions below, assume `fork()`, `pipe()`, `dup2`, etc. do not fail.

For your information:

- `int dup2(int oldfd, int newfd)` replaces file descriptor `newfd` with a copy of `oldfd`
- pipe produces an array of two file descriptors; the first is for reading, second for writing

(a) (4 points) If the code above is run with the input `XY` (followed by end-of-file), what is a possible output *to the original stdout*?

_____

(b) (4 points) If the code above is run with the input `XY` (followed by end-of-file), what is a possible output *to the original stderr*?

_____

(c) (5 points) Suppose the above program is started with only stdin, stdout, and stderr open, so it had three open file descriptors. Just before the program above returns from `main()`, how many file descriptors are open? (That is, how many file descriptor numbers refer to an open file, counting each number separately even if mutltple of those numbers refer to the same open file, like stdout and stderr often do.)

If the answer varies for some reason (for example, between parent and child), explain briefly.

_____

(d) (5 points) Adding which of the following lines of code where the comment "LOCATION ONE" appears would be likely to affect the program's output? **Select all that apply.**

- ◯ `dup2(pipe_fds[1], STDOUT_FILENO);`
- ◯ `close(pipe_fds[1]);`
- ◯ `dup2(pipe_fds[0], pipe_fds[1]);`
- ◯ `close(pipe_fds[0]);`
- ◯ `close(STDIN_FILENO);`

6. Consider Linux's Completely Fair Scheduler (CFS) as described in lecture, on a single core system.

   Suppose CFS is configured such that:

   - all threads have equal weight (for the purpose of calculating virtual time)
   - threads always have a 3 millisecond time quantum;
   - the virtual time for a thread that was previously not runnable is set to the maximum of its prior virtual time and 3 milliseconds less than the virtual time of the currently running thread;
   - a thread is preempted if another thread arrives whose virtual time is at least 1 millisecond less than the current thread's virtual time

   Assume context switch and other overheads are negilible unless otherwise stated.

   Suppose the following threads are running on this system:

   - thread A attempts to use the processor for 20 milliseconds to handle network input every 100 milliseconds; and
   - thread B attempts to use the processor for 20 milliseconds to handle network input every 400 milliseconds; and
   - thread C attempts to use the processor for 40 milliseconds to handle network input every 100 milliseconds; and
   - thread D attempts to use the processor continuously
   - thread E attempts to use the processor continuously

   For each the programs that handles network input, the input is received every 100 or 400 ms regardless of when the program does its computation to handle the input. (So if thread A receives input at time X, the next input will be received at time X+100 ms, regardless of whether thread A ran immediately to process the input or didn't get a chance to run at all before the next input came in.)

   If one of the programs is unable to handle one input until after the next is received, they will still need to process both inputs, performing twice as much computation (and possibly queuing up an arbitrarily large amount of computation).

(a) (8 points) Which threads will not use as much compute time as they would use if they were the only thread running on the system? **Select all that apply.**

- ◯ thread A
- ◯ thread B
- ◯ thread C
- ◯ thread D
- ◯ thread E

(b) (8 points) About what portion of the available compute time will thread D get over time?

7. (20 points) Consider a system for obtaining tickets to a popular event. When no ticket is available, a customer can ask to wait to be notified when a ticket is available, then have that ticket reserved for them to purchase. Customers are divided into two types: regular customers and VIP customers. Waiting VIP customers should get any ticket that's made available. To ensure this, regular customers must wait until there are more than enough tickets available to give one to each waiting VIP customer.

   To implement this, each customer is represented as a thread and the following methods are provided:

   - `Ticket *WaitForTicket(bool isVIP)`
   - `MakeTicketAvailable(Ticket *ticket)`

   Complete the following partial implementation of these functions. Don't worry about minor syntax errors or misnaming a pthreads function (if it's obivous what you're referring to).

   To the extent practical (without adding additional mutexes, condition variables, etc.), when completing the implementation, avoid having threads in WaitForTicket() become runnable after waiting for tickets to be available only to find out that it was still the case that no tickets were available and therefore they need to return to waiting.

```
int vip_waiters = 0; int nonvip_waiters = 0;
deque<Ticket*> available_tickets;
pthread_mutex_t lock; pthread_cond_t vip_cv, nonvip_cv;

Ticket *WaitForTicket(bool isVIP) {
    pthread_mutex_lock(&lock);
    if (isVIP) {
        vip_waiters += 1;

        while (_____________________________________________________) {
            pthread_cond_wait(&vip_cv, &lock);
        }
        vip_waiters -= 1;
    } else {
        nonvip_waiters += 1;

        while (_____________________________________________________) {
            pthread_cond_wait(&nonvip_cv, &lock);
        }
        nonvip_waiters -= 1;
    }
    Ticket *result = available_tickets.front();
    available_tickets.pop_front();
    pthread_mutex_unlock(&lock);
    return result;
}

void MakeTicketAvailable(Ticket *ticket) {
    pthread_mutex_lock(&lock);
    available_tickets.push_back(ticket);

    if (_____________________________________________________) {

        _____________________________________________________
    } else {

        _____________________________________________________
    }
    pthread_mutex_unlock(&lock);
}
```

8. Consider a video-playing site which allows users to put the videos on playlists represented as follows:

```
struct Video {
    pthread_mutex_t lock;
    int length;
    string name;
    bool deleted;
    vector<Playlist *> playlists_containing;
};
struct Playlist {
    pthread_mutex_t lock;
    vector<Video *> videos;
    int total_length;
};
```

It provides the following operations, with pseudocode implementations shown:

```
AppendPlaylistToPlaylist(destination, source) {
    Lock destination;
    Lock source;
    for each video in source->videos {
        Append video to destination->videos;
    }
    add source->total_length to destination->total_length;
    Unlock source;
    Unlock destination;
}
AppendVideoToPlaylist(video, playlist) {
    Lock playlist;
    edit playlist->videos;
    Lock video;
    edit video->playlists_containing;
    add video->length to playlist->total_length;
    Unlock video;
    Unlock playlist;
}
DeleteVideo(video) {
    Lock video;
    set video deleted;
    for each playlist in video->playlists_containing {
        Lock playlist;
        subtract video->length from playlist->total_length;
        Unlock playlist;
    }
    Unlock video;
}
```

(a) (8 points) Give an example of a deadlock that can occur by calling `DeleteVideo` and any function of your your choice above (including but not limited to DeleteVideo) at the same time from two different threads. Describe the deadlock by identifying the two parallel calls and identifying what locks are being waited on.

(b) (10 points) Describe a way to adjust the pseudocode above to prevent the deadlock you described in the previous question from occuring. (Your solution must still ensure that only one thread accesses the (non-lock) fields of a particular Video or Playlist at a time and should preserve the normal functionality of the functions above.)

9. Suppose we want to use counting semaphores to implement the following pair of functions:

   - Finish() — marks a task as finished; called exactly once
   - WaitForFinish() — waits for the task to be finished; returns immediately if Finish has already been called

   Consider the following implementation:

```
sem_t mutex;
sem_t gate;
int waiting = 0;
bool finished = false;

void Finish() {
    sem_wait(&mutex);
    finished = true;
    while (waiting > 0) {
        sem_post(&gate);
        waiting -= 1;
    }
    sem_post(&mutex);
}

void WaitForFinish() {
    sem_wait(&mutex);
    if (!finished) {
        waiting += 1;
        sem_post(&mutex);
        sem_wait(&gate);
    } else {
        sem_post(&mutex);
    }
}
```

(a) What should the initial value of the two semaphores be?

i. (3 points) mutex

_____

ii. (3 points) gate

_____

(b) (5 points) Which of the following changes, each done alone and with the correct choice of any parameters (like new semaphore values), would not change the functionality of the above code? **Select all that apply.**

- ◯ `post`'ing on `mutex` in Finish just before the while loop instead of after it
- ◯ replacing every wait on `mutex` with a `post` and every post on `mutex` with a `wait` while choosing an appropriate, new initial value for `mutex`
- ◯ changing the while loop to a `do { ... } while (...)` loop with the same condition
- ◯ `wait`'ing on `gate` in WaitForFinish before the post to `mutex` instead of after it
- ◯ replacing uses of `mutex` with a memory barrier before reading and after writing `waiting` (assuming reads/writes to `waiting` are atomic)

10. Consider a system that uses virtual memory, with page tables structured the way they are in x86-based xv6:

    - virtual and physical addresses are 32 bits
    - there are two levels of page tables
    - each page table at each level has a 1024 (2^{10}) entries
    - each page table entry is 4 bytes
    - pages are 2^{12} bytes

    For the questions below, you may leave your answers as unsimplified arithmetic expressions.

    In the questions below, a 'first-level page table' is what the page table base pointer supplied to the processor points to. A 'second-level page table' is what a page table entry in the first-level page table points to.

(a) Suppose a process running on this system can access 5MB ($5 \cdot 2^{20}$ bytes) of memory without page faults occuring.

i. (5 points) In order to do this, what is the minimum number of valid entries required in the process's second-level page tables?

ii. (5 points) In order to do this, what is the minimum number of valid entries required in the first-level page table?

(b) (5 points) Suppose a process running on this system has the following memory layout:

- `0x0010 0000`–`0x001F FFFF` (1MB): code (read-only)
- `0x0030 0000`–`0x004F FFFF` (2MB): data (read/write)
- `0x7FFF E000`–`0x7FFF FFFF` (8KB): stack (read/write)

How much space **in kilobytes** is allocated for this process's page tables (both first and second-level)? Assume all pages listed above are accessible without page faults and no other memory is accessible (so there are no valid page table entries other than the ones corresponding to the memory regions listed above, not even ones reserved for the OS).

(For this question, a kilobyte is 2^{10} bytes and a megabyte is 2^{20} bytes.)