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 *data on disk requires I/O*

   ◯ converting an integer to a string in preparation for outputting it *computation, no need for kernel to be involved. Actual output will require syscall, but this isn't that step.*

   √ creating a new process *modifying kernel data structures; affecting scheduler*

   ◯ reading the updated version of a value currently only stored in another core's cache *done by processors and caches without OS getting involved*

   ◯ switching away from a program that has been running for too long *done by timer interrupt, not triggered by program explicitly*

   ◯ acquiring a lock while no other thread has or is trying to acquire the lock *in typical lock implementation uses a spinlock (or similar) with atomic operations that don't require the OS getting involved; if a process needs to wait, then the OS needs to get involved because the scheduler is invoked*

2. (4 points) Suppose an xv6 system is running ~~three~~ two*was wrong on exam* (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 OS needs to save information to return back to where the system call was made; xv6 does this as part of the saved registers*

   √ the saved user stack pointer value in process A could be the same as the saved user stack pointer value in process B *different address spaces, so they could have the same address but point to different values*

   ◯ the xv6 kernel performed an I/O-like operation to trigger the timer interrupt from A's system call implementation *the timer interrupt must have been arranged for before A's system call was running*

   ◯ 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 *unlikely since process B was in a timer interrupt*

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 *each time quanta expiration is a context switch*

   ◯ increases the amount of time between when a program becomes runnable after I/O and when it first runs *decreases it instead; it has to, at worst, wait for each other program to run once; if N programs = N time quanta, which gets shorter with shorter time quanta*

   ◯ increases the amount of storage required to implement the scheduler *unaffected*

   ◯ increases throughput *more context switches*

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 *want strict priority of, e.g., 1ms CPU burst over 1.5ms CPU burst*

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*? *parent writes A to original stdout only*

A

(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*? *child reads X from stdin (Y never read) and writes to stderr; parent reads A from pipe and writes to stderr*

XA or AX

(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.

5

(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);` *stops child from outputting to stdout*
- ◯ `close(pipe_fds[1]);` *this fd never used after this point*
- ◯ `dup2(pipe_fds[0], pipe_fds[1]);` *this fd never used after this point*
- ◯ `close(pipe_fds[0]);` *this fd never used after this point*
- √ `close(STDIN_FILENO);` *breaks read from stdin in child*

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).

   - The resetting of virtual time won't change the amount of time programs use over the long term — it will only affect their short term usage. If, for example, C doesn't manage to do all its 40 ms of work before the next network input, then it won't have it's virtual time reset (to the max of prior time and current program's time - 3ms) — instead it will still be runnable wiht its original virtual time. Because of this, either the programs will be running all the time or they will finish the computation they wanted to do by the time the next network input comes. Because virtual time is reset, its computation will not run as quickly after the network input as it would otherwise, but it will still complete before the next network input.
   - If no adjustments to virtual time are made and all programs are always runnable, CFS will divide up the compute time evenly over the long term, giving each program 1/5th (20%) of the time.
   - Some programs won't be able to use all of their 20% share. Their time will be divided up evenly among all the programs which can use it (to the extent they can use it).
   - So, to figure out how much compute time each program uses, we can sort the programs by how much CPU time they would use alone:
     - B: 5%
     - A: 20%
     - C: 40%
     - D: 100%
     - E: 100%

     then we can go through in order and allocate time to fill up the lowest-demand program evenly. So first, we give each program 5% of the time to satisfy B
     - B: 5% → 5% done; 0% left

- A: 20% → 5% done; 15% left
- C: 40% → 5% done; 35% left
- D: 100% → 5% done; 95% left
- E: 100% → 5% done; 95% left

and then 15% to the remaining programs to fill A:

- B: 5% → 5% done; 0% left
- A: 20% → 20% done; 0% left
- C: 40% → 20% done; 20% left
- D: 100% → 20% done; 80% left
- E: 100% → 20% done; 80% left

then we've allocated all but 15% of the compute time, so we divide the remaining 15% evenly among the remaining programs (C, D, E):

- B: 5% → 5% done; 0% left
- A: 20% → 20% done; 0% left
- C: 40% → 25% done; 15% left
- D: 100% → 25% done; 75% left
- E: 100% → 25% done; 75% left

and we see that programs C, D, and E don't get the full portion of the compute time they are trying to use.

(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? *-0.5 point total if just inverted; otherwise, -2 points per wrong, minimum 0* **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?

Solution: 25%. 20% taken by A, 5% by B, leaving 75% split between C, D, and E, for 25% a piece. All would naturally use more than 25%, so D gets 25% and not more.

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.

   - 5 points for checking ticket count in while loop
   - 5 points for checking vip_waiters in while loop
   - 5 points for condition on if statement
   - 5 points for signalling
   - -2 points if only checking `vip\_waiters > 0` instead of comparing to ticket count (one deduction across whole question, regardless of whether done in while loop, if statement, or both)
   - -1 point for mixing up $\geq$ and $>$ or $\leq$ and $<$ (each)
   - -2 point for inverting a condition or mixing up AND and OR
   - -2 point for broadcast instead of signal
   - -2 point (in addition to deduction for getting conditions wrong) for trying to use condition variable as boolean value

```
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 (available_tickets.size() == 0) {
            pthread_cond_wait(&vip_cv, &lock);
        }
        vip_waiters -= 1;
    } else {
        nonvip_waiters += 1;
        while (vip_waiters >= available_tickets.size()) {
            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 (vip_waiters >= available_tickets.size()) {
        pthread_cond_signal(&vip_cv);
    } else {
        pthread_cond_signal(&nonvip_cv);
    }
    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. *4 points for identifying calls that cause deadlock; 2 points per lock correctly i'd; -2 overall for swapping which function was waiting on which lock*

> **Solution:** DeleteVideo(video) and AppendVideoToPlaylist(video, playlist) where the playlist already contains the video once. DeleteVideo: waiting on playlist lock; AppendVideToPlaylist: waiting on video lock

(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.)

> **Solution:** Option 1: AppendVideoToPlaylist should lock video first instead of later; Option 2: DeleteVideo should stash `video->length` in a temporary and unlock video before editing playlists

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

1

ii. (3 points) gate

0

(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 *final value of gate is too high if no waiters, but not actually a correctness issue*
- ◯ `wait`'ing on `gate` in WaitForFinish before the post to `mutex` instead of after it *deadlock*
- ◯ replacing uses of `mutex` with a memory barrier before reading and after writing `waiting` (assuming reads/writes to `waiting` are atomic) *doesn't ensure that the += doesn't have a lost write*

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?

Solution: $5 \cdot 2^{20}/2^{12} = 5 \cdot 256 = 1280$

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

Solution: 2

(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.)

Solution: 4 times (1 (first-level) + 1 (second-level for VPNs 0x00000-0x003FF) + 1 (second-level for VPNs 0x00400-0x007FF) + 1 (second-level including 0x7FFF)) = 4 times 4 = 16 KB