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. (5 points) In xv6, system call arguments are placed on the stack before executing the special instruction to trigger a system call exception. Although the placement of system call arguments matches the 32-bit x86 calling convention used with in the Linux kernel, xv6 has utility functions like `argptr()` that are used from system call handlers (like `sys_write`) to retrieve the arguments for system calls. These functions are helpful because _____. **Select all that apply.**

   - √ the kernel needs to handle the case where the stack on which the arguments should be stored is corrupted or otherwise invalid
   - √ the arguments are on a different stack than the one being used when the system call handler is called
   - √ the system call handler function is not called directly, so the arguments are not in the location where the C compiler would expect to retrieve them
   - ◯ a context switch might occur between when the system call is triggered and when the system call arguments are read, resulting in the arguments being moved on the stack
   - ◯ the kernel needs to change the page table base pointer before it can access system call arguments

2. (10 points) Suppose an xv6-like operating system is running two processes A and B. Initially, process A is active. Then, it makes a system call that requires it to wait for keyboard input. While it is waiting for keyboard input, process B runs. Process B reads some input data from a file, then performs an intensive calculation until a keypress (which A was waiting for) occurs. Then process A begins running again. Identify each user-to-kernel mode switch that occurs during this process:

   Solution: system call from A; system call for reading input from B; interrupt for keypress

3. (5 points) Suppose one wants to use a single counting semaphore to limit to 16 the number of temporary files threads place in a directory. (When the limit is reached, a thread waits for a temporary file to be deleted.) What should the initial value of this semaphore be?

   16

4. For each program below, identify which outputs are possible. You may assume that none of the POSIX functions (`fork`, `waitpid`, `pthread_join`, etc.) fail. Ignore any minor syntax errors.

(a) (8 points)
```
int a = 1;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
void output(int *ptr) {
    pthread_mutex_lock(&lock);
    printf("%d %d ", a, *ptr);
    pthread_mutex_unlock(&lock);
}

void *thread_func(void *raw_ptr) {
    int b = *((int*) raw_ptr);
    pthread_mutex_lock(&lock);
    a *= 3; b *= 3;
    pthread_mutex_unlock(&lock);
    output(&b);
    return NULL;
}

int main(void) {
    int b = 2;
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, &b);
    output(&b);
    pthread_join(thread, NULL);
    return 0;
}
```

Which outputs are possible? *-1 for each disagreeing to a minimum of 0* **Select all that apply.**

◯ 1 2 ◯ 1 2 1 2 ◯ 1 2 3 2
√ 1 2 3 6
◯ 1 6 3 2
◯ 3 6 ◯ 3 6 1 2 ◯ 3 6 1 6 √ 3 6 3 2 ◯ 3 6 3 6

(b) (8 points) *exam as printed had pid < 0 instead of pid == 0*

```
char global = 'A';
int main(void) {
    char local = '1';
    int pipe_fds[2];
    pipe(pipe_fds);
    pid_t pid = fork();
    if (pid == 0) {
        read(pipe_fds[0], &global, 1);
        write(STDOUT_FILENO, &global, 1);
        write(STDOUT_FILENO, &local, 1);
    } else {
        local = '2';
        write(pipe_fds[1], &local, 1);
        write(STDOUT_FILENO, &global, 1);
        waitpid(pid, NULL, 0);
    }
}
```

Which of the following are possible outputs of this program? *-1 point for each disagreeing, except counting the duplciated answer only once* **Select all that apply.**

◯ 11A ◯ 112 √ 21A √ 2A1 ◯ 211 √ 21A ◯ A12A12 √ A21

5. Suppose a system is running three threads A, B, and C and scheduling them so that they share a single core. Assume context switch overheads are negligible, and during the time period of interest the threads run as follows:

   At time 4, thread A will finish an I/O operation. Then, after it performs 7 time units of computation, it terminates.

   At time 5, thread B will finish an I/O operation. Then, after it performs 2 time units of computation, then it performs an I/O operation that completes 5 time units after it starts. Then, it performs 2 time units of computation, then it terminates.

   At time 0, thread C will finish an I/O operation. Then, after it performs 100 time units of computation, it terminates.

(a) *(as printed on exam, B in second row was listed as 'not runnable' instead of '(waiting for I/O)'* Suppose a system schedules these threads as follows:

time	thread A	thread B	thread C
0–4	(waiting for I/O)	(waiting for I/O)	running
4–5	running	(waiting for I/O)	(not running but runnable)
5–6	running	(not running but runnable)	(not running but runnable)
6–8	(not running but runnable)	running	(not running but runnable)
8–13	running	(waiting for I/O)	(not running but runnable)
13–15	(terminated)	running	(not running but runnable)
15–111	(terminated)	(terminated)	running

i. (2 points) How many context switches occur in the schedule above? (Ignore any context switch at time 0 or time 111.)

5

ii. (6 points) Recall that turnaround time is the time from when a thread becomes runnable (for example, due to I/O finising) till when it ceases to be runnable (for example, due to starting an I/O operation). What is the sum of the turnaround times experienced by each thread for each of their CPU burst in the schedule above?

Thread A	Thread B	Thread C
9 time units	3+2=5 time units	111 time units

(b) (10 points) Write a schedule for the above threads that minimizes their **total** turnaround time. You do not need to include information about when threads are not running (everything in parenthesis in the schedule above). The first line is done for you.

original version of the key had an incorrect annotations in parens: marking A as runnable rather than terminated, and marking C as terminated but then running again

time	thread A	thread B	thread C
0–4	(waiting for I/O)	(waiting for I/O)	running
4–5	running	(waiting for I/O)	(not running but runable)
5–7	(not running but runnable)	running	(not running but runnable)
7–12	running	(waiting for I/O)	(not running but runnable)
12–13	running	(not running but runnable)	(not running but runnable)
13–15	(terminated)	running	(not running but runnable)
15–111	(terminated)	(terminated)	running

6. Consider a multi-level feedback queue scheduler with three priority levels:

   - priority 2 (highest), which runs threads with a 10 ms timeslice;
   - priority 1, which runs threads with a 20 ms timeslice; and
   - priority 0 (lowest), which runs threads with a 100 ms timeslice

   When a thread uses its entire timeslice and remains runnable, it is moved to a lower priority level (if possible). When a thread does not use entire timeslice (and was not preempted), it is moved to a higher priority level (if possible). Initially threads start out at the highest priority level. Within each priority level, threads are scheduled in a round-robin manner.

   Suppose a system with this scheduler is running three thread A, B, and C:

   - A alternates between requiring 15 ms of computation and an I/O operation that takes 40 ms
   - B alternates between requiring 5 ms of computation and an I/O operation that takes 10 ms
   - C alternates between requiring 450 ms of computation and an I/O operation that takes 1 ms

   Assume context switch overheads and time spent by the OS processing I/O requests are negligible. All questions below assume that the threads have been running for a very long time (so, for example, the priority level a newly created thread starts is unlikely to affect your answers).

(a) (4 points) After these threads have been running for a long time, what priority levels will thread C run at? (For example, if you think it will end up alternating between priority 2 and priority 0, write "0 and 2".)

0 and 1 (half credit for just 0)

(b) After these threads have been running for a long time, suppose thread B stops to perform I/O while threads A and C are runnable.

i. (5 points) What is the **minimum** time from when thread B's I/O completes until when it will start its next I/O operation?

5 ms

ii. (5 points) What is the **maximum** time from when thread B's I/O completes until when it will start its next I/O operation?

Because A was runnable when B stopped to perform its I/O, A will have either started another I/O by the time B runs or will have been demoted to priority 1. In either case, B will be able to run immediately when it returns from its I/O.

We gave most of the credit for 15ms, based on B ***returning*** *from its I/O when A and B were runnable, which was our original intention when writing the question, but not the question we actually wrote.*

A multi-level feedback queue scheduler should immediately start running a higher priority thread when it becomes runnable while a lower priority thread is running. This was hinted at the text above with "(and was not preempted)". Byeond that, it wouldn't make sense to do otherwise given the goal of the scheduler: if we let something at priority 0 use its entire 100 ms timeslice, then failing to interrupt that 100 ms timeslice for something with a short CPU burst will dramatically fail to achieve the goal of having a low turnaround time. It's also generally the most sensible thing to do for a priority scheduler in general, since otherwise the priorities won't actually be very effective.

Some students interpreted the choice about whether to run something at a higher priority immediately, or to wait for the current time slice to end, as a choice about being preemptive or not. The scheduler is preemptive regardless — because it enforces a maximum time slice, it must be preempting programs in order to do that.

5ms

(c) (4 points) Each of C's 450 ms computation periods will take **approximately** _____ ms to complete (from the time C becomes runnable until it starts its short IO operation). Choose the most

reasonable estimate below. *Note that C is always at a lower priority then A or B, so it can only run when both are not running.*

If we assume A and B always ran their CPU burst immediately after finishing their I/O, then A would run $15/(15+40) = 27\%$ *of the time and B would run* $5/(5+10) = 33\%$ *of the time. This would give around 40% of the time C, turning 450 ms of computation into about 1125 ms.*

Now, we know that sometimes A or B might need to wait for each other.

If B delays A, the worst case is that A has to spend 5 ms waiting for B's CPU burst to run each time its interrupted. Since A's CPU burst is only 15ms, this can happen at most twice. This would result in A running for about $15/(10+15+40) = 23\%$ *of the time. Since B's I/O is only 10 ms, some scenario like this could potentailly happen repeatedly*

If A delays B, the worst case is that B has to wait 10ms for A to run at priority 2 before it gets downgraded. But when this happens, the next two times B runs it can't be interrupted by A, because A will be in the middle of its 40ms I/O. This would result in B runnning for about $(5+5+5)/(5+10+10+5+10+5+10) = 27\%$ *of the time.*

Both B and A delaying each other together would give around an upper bound of around 50% of the CPU time for C, causing 450ms of computation to take 900 ms.

- ◯ 455 ms
- ◯ 500 ms
- ◯ 700 ms *also accepted*
- √ 1100 ms
- ◯ 1500 ms
- ◯ 4500 ms

7. (30 points) Suppose in order to avoid overloading the network, a web browser limits the number of simulatenous downloads it makes to web servers as follows:

   - it makes at most one download at a time of a video files
   - it makes at most five downloads at a time in total

   To implement this, the web browser performs downloading from multiple threads. Before starting a download, a thread calls a `StartVideoDownload` or `StartNonVideoDownload` function. These functions is responsible for waiting to ensure that the limits are respected **without consuming a large amount of compute time**. After finishing a download, a thread calls `EndVideoDownload` or `EndNonVideoDownload` function.

   In order to ensure that videos get downloaded promptly, the browser also **gives priority to threads waiting to perform a video download over threads waiting to perform a non-video download**.

   On the next page, **complete an implementation of these functions using monitors**. Whenever practical, have threads wait without using the compute time in your implementation. (Please ignore minor syntax errors and missing error checking.)

   The implementation includes a utility function called `EndAny`, which is only used internally by `EndVideoDownload` and `EndNonVideoDownload`.

   12, 12 and 6 points per function

   For Start functions, 1 point for while, 4 points for each of two parts of while conditions; 3 points for wait

   for end function, 2 points for if condition; 2 points for each of two signals; 1 point total deduction if ever broadcasting instead of signalling

   code on next page is example; several equivalent variations are fine

```
int num_nonvideo = 0; int num_video = 0; int num_pending_video = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t video_ready_cv = PTHREAD_COND_INITIALIZER;
pthread_cond_t nonvideo_ready_cv = PTHREAD_COND_INITIALIZER;

void StartVideoDownload() {
    pthread_mutex_lock(&lock);
    num_pending_video += 1;
    while (num_video == 1 || (num_video + num_non_video) == 5) {
        pthread_cond_wait(&video_ready_cv, &lock);
    }
    num_pending_video -= 1;
    num_video += 1;
    pthread_mutex_unlock(&lock);  // original exam as printed use lock() here by accident
}

void StartNonVideoDownload() {
    pthread_mutex_lock(&lock);
    /* our original key had, for the first
       part of the condition below:
         (num_pending_video > 0)
       [and we gave credit for this].
       This does give priority to video downloads but at the cost of not running
       non-video downloads more often than necessary, so it's not really a good choice.
       [added to key 2019-10-28]*/
    while ((num_pending_video > 0 && num_non_video >= 4)|| (num_video + num_non_video) == 5)
        pthread_cond_wait(&nonvideo_ready_cv, &lock);
    }
    num_nonvideo += 1;
    pthread_mutex_unlock(&lock);
}

// utility function
void EndAny() {
    if (num_pending_video > 0 && num_video != 1) {
        pthread_cond_signal(&video_ready_cv);
    } else {
        pthread_cond_signal(&nonvideo_ready_cv);
    }
}

void EndVideoDownload() {
    pthread_mutex_lock(&lock);
    num_video -= 1;
    EndAny();
    pthread_mutex_unlock(&lock);
}

void EndNonVideoDownload() {
    pthread_mutex_lock(&lock);
    num_nonvideo -= 1;
    EndAny();
    pthread_mutex_unlock(&lock);
}
```

8. Consider the following C++ code that forms part of a course registration system that keeps track of the students and courses in memory. (`set` is a container class provided by the standard library which stores a set of values using something similar to a balanced binary tree.) *exam as printed did used both `course` and `from_course` to refer to the first arg of TransferStudents, was missing the insert into enrolled() and was missing ampersands on some lock calls*

```
struct Course {
    pthread_mutex_t lock; string name; set<Student *> enrolled;
};
struct Student {
    pthread_mutex_t lock; string name; set<Course *> courses;
};

void RemoveStudentFromCourse(Student *student, Course *course) {
    pthread_mutex_lock(&student->lock);
    pthread_mutex_lock(&course->lock);
    course->enrolled.erase(student);
    student->courses.erase(course);
    pthread_mutex_unlock(&course->lock);
    pthread_mutex_unlock(&student->lock);
}

void TransferStudents(Course *from_course, Course *to_course) {
    pthread_mutex_lock(&from_course->lock);
    /* special syntax for "foreach Student* called 'student' in course->enrolled" */
    for (Student *student : from_course->enrolled) {
        pthread_mutex_lock(&student->lock);
        pthread_mutex_lock(&to_course->lock);
        student->courses.erase(from_course);
        to_course->enrolled.insert(student);
        student->courses.insert(to_course);
        pthread_mutex_unlock(&to_course->lock);
        pthread_mutex_unlock(&student->lock);
    }
    from_course->enrolled.clear();
    pthread_mutex_unlock(&from_course->lock);
}
```

If `RemoveStudentFromCourse` and `TransferStudents` are called in parallel from two different threads, then deadlock can occur.

(a) (10 points) Give an example of the arguments to such parallel calls and identify which locks or other resources each thread could be waiting for when the deadlock occurs.

Solution: multiple possible answers; one example: RemoveStudentFromCourse(A, X), TransferStudents(X, Y) where X->enrolled initially contains A. RemoveStudenFromCourse will be waiting on X's lock, TransferStudents will be waiting on A's lock.

(b) (5 points) Propose a change to `RemoveStudentFromCourse` and/or `TransferStudents` that will fix this deadlock. Avoid harming the functionality and efficiency of the functions with your change.

Solution: multiple possible answers (and depends on previous question); one example: have RemoveStudentFromCourse always lock course before locking the student

9. On 32-bit x86 as used by xv6, virtual memory uses 4096 byte (2^{12} byte) pages, with two-level page tables. In these two-level page tables, tables at each level have 1024 entries, each of which is 4 bytes and contains a valid (also known as 'present') bit, a physical page number, and various permission bits.

   Suppose a process is running with a page table base register (CR3) value of `0x10000` (which is a physical byte address, not a page number).

(a) Suppose when the process accesses memory at virtual address `0x00880230`, the corresponding first-level page table entry for the process is marked as valid and contains the physical page number 0x13 and the corresponding second-level page table entry is also marked as valid and contains physical page number 0x15.

   i. (5 points) What is the physical address of the second-level page table entry? (The address at which is stored, not any address it represents.)

   0x13200 (half-credit for 0x13080)

   ii. (5 points) When the process writes to virtual address `0x00880230`, what physical address be written to?

   0x15230

(b) (6 points) Which of the following are true about page tables on 32-bit x86? **Select all that apply.**

- ◯ each physical page number appears in at most one page table entry in any particular process's page tables
- ◯ an `add` instruction that sets %esp (the stack pointer) to a virtual address without a valid page table entry triggers an exception
- √ attempting to write to a virtual memory address whose page table entry is marked as non-writeable triggers an exception
- ◯ if a first-level page table entry is valid, then all the second-level page table entries that are accessed using that first-level page table entry are also valid
- ◯ if a first-level page table entry for a virtual page is marked invalid, then the second-level page table entry for the page can override that
- √ when an exception is triggered by attempting to access an invalid page table entry, returning from the exception normally will retry to the failed memory access