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 the traditional Unix's monolithic kernel operating system design (like used by Linux and xv6), which of the following typically operations occur in kernel mode? **Select all that apply.**

   - √ creating a new process when a process calls `fork` *needs to modify process table, create page table, ...*
   - √ checking whether a file exists when a process calls `open`
   - √ displaying text on the console when a process calls `printf` *actual displaying of text needs write system call*
   - √ assigning new memory to a process's address space when a process calls `malloc` *needs to modify page table*
   - ◯ comparing strings when a process calls `strcmp`
   - √ switching from one process to another

2. (4 points) In a multi-level feedback queue scheduler, programs in the highest priority queue also ____________________. **Select all that apply.**

   - √ tend to be less compute-intensive than programs at lower priorities
   - ◯ have longer timeslices than programs at lower priorities
   - ◯ will be demoted to a lower priority if they stop using the CPU before the end of their timeslice
   - √ will be demoted to a lower priority if they keep using the CPU up until the end of their timeslice

3. For **each** of the following statements, indicate whether they are true about

   - a first-come first-served scheduler (FCFS)
   - a round-robin scheduler (RR)
   - a lottery scheduler, like the one required for the scheduler assignment (lottery)
   - a shortest job first scheduler without preemption (SJF)
   - a shortest remaining time first scheduler with preemption (SRTF)
   - a multi-level feedback queue scheduler (MLFQ)
   - Linux's Completely Fair Scheduler (CFS)
   - a strict priority scheduler with preemption (prio)

   For each question, do not accounting for how users might adjust what threads they run (other than specifying scheduler parameters, like priorities) based on the scheduler in use.

   For each question below, **select all that apply**. *-1 point for each option that disagreed with key to a minimum of 0*

(a) (5 points) This kind of scheduler is subject to starvation, where some thread will **never** run, no matter how long it waits, when there are certain kinds of competing threads, even if the scheduler does not allow any one thread to run indefinitely.

◯ FCFS ◯ RR ◯ lottery √ SJF √ SRTF √ MLFQ ◯ CFS √ prio

(b) (5 points) This kind of scheduler allows the relative importance of threads to be specified by the user or system administrator and used in the scheduling decision.

◯ FCFS ◯ RR √ lottery ◯ SJF ◯ SRTF ◯ MLFQ √ CFS √ prio

(c) (5 points) This kind of scheduler is not actually possible to implement exactly on a typical operating system, because it requires knowing about the future behavior of programs.

◯ FCFS ◯ RR ◯ lottery √ SJF √ SRTF ◯ MLFQ ◯ CFS ◯ prio

4. (20 points) Consider the following incomplete C program that uses the POSIX API: (Assume all appropriate headers are included and ignore minor syntax errors.)

```
int main(void) {
    int fds[2] = {1, 1}; pid_t pid = (pid_t) -1; char c;

    __ A __; // LOCATION 1
    pid = fork();

    __ (nothing) __; // LOCATION 2
    if (pid == 0) { /* child: */

        __ D (optional) __; // LOCATION 3
        write(fds[1], "Hello!", 6);
    } else {

        __ E __; // LOCATION 4

        while (__ G __ > 0) {  // LOCATION 5
            printf("%c", c);
        }
        waitpid(pid, NULL, NULL);
    }
    return 0;
}
```

To make this program output "Hello!", one or more of the following statements need to be inserted into the blanks marked Location 1 through 5.

A. `pipe(fds)`
B. `dup2(fds[1], STDOUT_FILENO)`
C. `dup2(fds[0], STDIN_FILENO)`
D. `close(fds[0])`
E. `close(fds[1])`
F. `close(STDIN_FILENO)`
G. `read(fds[0], &c, 1)`
H. `read(fds[1], &c, 1)`
I. `write(fds[1], &c, 1)`
J. `read(STDOUT_FILENO, &c, 1)`

original question as given had a newline in the printf, but it seems that most students ignored that, assuming I wasn't being superprecise about what it meant to output Hello!.

3 point deduction for using B in the child (it has no effect; the child never outputs to STDOUT_FILENO and it suggests you might think dup2 works the other way...)

common 5 point deductions:

- *not closing fds[0] in the parent which will make the read in location 5 block forever*
- *dup2'ing to replace stdout with a pipe in the parent proces*
- *closing fds[1] before it was written to*
- *closing whatever the parent tried to read from*
- *reading from a write-only file descriptor*
- *reading from the pipe before it was ever written to*

- *writing an uninitialized value to the pipe*
- *creating the pipe after forking*

10 point deduction for never attempting to read all the characters from the pipe

full credit was also given for answers which never called pipe(fds) thus making the child write to stdout directly (STDOUT_FILENO=1), provided that they also never called dup2(fds[1], STDOUT_FILENO) (which would be dup2(1,1), which doesn't make much sense to do if you realize STDOUT_FILENO = fds[1]) and used something that would definitely return 0 or -1 in location 5. (This was not an intended answer. I did want to have some valid descriptors in fds[2], because a common problem on the shell homework was calling close on an uninitialized integer that happened to be a valid fd and I wanted to emphasize that they might not be all 0s, but I choose the wrong dummy numbers.)

For each line above, write the letters A through J that would be appropriate to place in each location above. **Some lines may be left blank and some letters may be left unused.**

`pipe(a)` takes an array `a` of two file descriptors and creates a pipe where element 1 of the array is the write end of the pipe and element 0 is the read end.

`dup2(from, to)` takes two file descriptors `from` and `to` and assigns the file descriptor number `to` to reference the same open file that `from` does.

You may assume `fork` and `waitpid` and `pipe` will not fail, and that `write` never writes less than requested when passed an appropriate file descriptor.

5. (8 points) Consider the following C program that uses the pthreads API:

```c
int x = 50;
void child(void *ignored_argument) {
  x = x + 10;
  printf("[child]␣%d␣", x);
  x = x + 10;
}
int main(void) {
  pthread_t child_thread;
  x = 100;
  pthread_create(&child_thread, NULL, child, NULL);
  x = x - 1;
  pthread_join(child_thread, NULL);
  printf("[main]␣%d", x);
  return 0;
}
```

Assume that `pthread_create` and `pthread_join` do not fail, that reads and writes to `x` are atomic and are executed in each thread in the order specified in the program's code (not reorderd by the compiler or processor), and that the program does not crash or fail to compile.

Which of the following outputs are possible?

child always outputs before main because of the join

110/119 can occur when child executes, then main (between pthread_create/join) executes.

109/119 can occur when main executes, then child executes

110/109 is possible from:

main	*child*
x = 100	
	start
	load x = 100
	store x = loaded + 10 = 110
load x = 110	
	print
	load x = 110
	store x = loaded + 10 = 120
store x = loaded - 1 = 109	

110/99 is similar, but with main's load of 110 replaced with an earlier load of 100.

109/120 requires the child to read 109, which can only happen after main writes either 99 or 109. But then the last write to x in the child must store 120, which means the child would need to read 110 immediately after. **Select all that apply.**

○ `[main] 110 [child] 119`

[✓ `[child] 110 [main] 109`]

○ `[child] 109 [main] 120`

○ `[child] 60 [main] 99`

[✓ `[child] 110 [main] 119`]

[✓ `[child] 110 [main] 99`]

◯ `[main] 110 [child] 109`

✓ `[child] 109 [main] 119`

6. Consider implementing a "synchronized buffer" data structure. This structure has two operations:

   - void Put(int value)
   - int Get()

   Each time a thread calls Put(), it puts a value into a shared buffer and returns *after the item has been retrieved by another thread calling Get()*. Each item that it is Put() into the buffer will be retrieved exactly once (assuming some thread eventually calls Get()).

   If a thread calls Get() and an item is not yet available, it first waits until an item is available, then returns that item.

(a) (30 points) Suppose we implement this data structure with mutexes and condition variables with the following incomplete code, where code to fill in is marked with a blank and a commented letter:

```
int current_value;
bool current_value_used = false; bool current_value_received = false;
pthread_mutex_t lock;
pthread_cond_t put_ready; pthread_cond_t put_done; pthread_cond_t get_ready;

void Put(int value) {
    pthread_mutex_lock(&lock);

    while (current_value_used) {
        pthread_cond_wait(&put_ready, &lock);
    }
    current_value = value;
    current_value_used = true;
    current_value_received = false;
    pthread_cond_signal(&get_ready);

    while ( /* A */ __ !current_value_received __) {
        pthread_cond_wait(&put_done, &lock);
    }

    /* B */ __ current_value_used = false __;

    /* C */ __ pthread_cond_signal(&put_ready) __;
    pthread_mutex_unlock(&lock);
}

int Get() {
    pthread_mutex_lock(&lock);

    while ( /* D */ __ !current_value_used || current_value_received __) {
        pthread_cond_wait(&get_ready, &lock);
    }

    int result = current_value;
    current_value_received = true;

    /* E */ __ pthread_cond_signal(&put_done) __;
    pthread_mutex_unlock(&lock);
    return result;
}
```

15 points for while loop conditions (5 for A, 10 for D); 10 points for signalling, 5 points for adjusting used/received. Almost everyone missed the corner case for D. !used is needed at least in case Get() is called just before the first call to Put() gets the lock; received is needed in case the second call to Get()

acquires the lock between when the first call to Get() signals put_done and when the first call to Put() is able to reacquire the lock at location A. **Fill in each of the blanks (A, B, C, D, E) above.** You may assume that there is code elsewhere that will correctly initialize the mutexes and condition variables.

(b) Suppose we implement this synchronized buffer data structure with semaphores with the following code: *The code below doesn't do this, because we don't wait for the Get to finish at the end of a Put. Another semaphore would be needed to fix this. However, it does implement a simpler shared buffer data structure, and there's only one sensible value for the semaphores below that doesn't result in Put values being ignored entirely or Get returning an uninitialized value. I think most people interpreted the question as 'what semaphore value makes any sense' anyways, so we graded it that way, but if you were confused because you were looking for put to wait after setting a value and can explain how that affected your answer, please tell me (Prof. Reiss) and I may make some adjustments.*

```
int current_value;
sem_t put_gate;
sem_t get_gate;

void Put(int value) {
    sem_wait(&put_gate); // wait = down or P operation
    current_value = value;
    sem_post(&get_gate); // post = up or V operation
}

int Get() {
    sem_wait(&get_gate);
    int result = current_value;
    sem_post(&put_gate);
    return result;
}
```

What should the initial values of the counting semaphores `put_gate` and `get_gate` be?

i. (5 points) `put_gate`:

1

ii. (5 points) `get_gate`:

0

7. (7 points) Suppose a multiprocessor system uses a cache coherency mechanism where:

- each processor has its own cache
- memory and these caches are all connected via a shared bus
- each block in the caches are Invalid or Modified (value is dirty and held by exactly one processor's cache) or Shared (value is not dirty and potentilly held by another processor's cache)

Which of the following operations require a procesor to send a message on the shared bus? **Select all that apply.**

- ◯ writing to a value which it currently has cached in the Modified state
- ✓ reading from a value which it currently does not have cached
- ◯ completing an atomic exchange operation on a value it currently has cached in the Modified state
- ✓ completing an atomic exchange operation on a value it currently has cached in the Shared state
- ✓ writing to a value which it currently does not have cached
- ✓ writing to a value which it currently has cached in the Shared state
- ◯ reading from a value which it currently has cached in the Shared state

8. Consider the following C++ code for a flight reservation system. The `makeReservation` function simulatenously tries to simulatenously reserve a seat (decrementing from a count of available seats) on several flights and returns true if it is successful; otherwise, if not enough seats were available, it returns false. Unfortunately, it has few bugs.

```
struct Flight { string name; int remainingSeats; pthread_mutex_t lock; };
bool makeReservation(vector<Flight*> flights) {
    for (int i = 0; i < flights.size(); ++i) {
        pthread_mutex_lock(&flights[i]->lock);
        if (flights[i]->remainingSeats == 0) {
            pthread_mutex_unlock(&flights[i]->lock);
            return false;
        }
    }
    for (int i = 0; i < flights.size(); ++i) {
        flights[i]->remainingSeats -= 1;
        pthread_mutex_unlock(&flights[i]->lock);
    }
    return true;
}
```

(a) (10 points) When one of the flights does not have any seats left, the above `makeReservation` function can result in the program hanging later on — even if the program only has one thread. Explain **briefly** (at most one sentence) or give an example of how this could happen.

> **Solution:** makeReservation won't unlock the lock for the first flight if the first has enough seats but the second does not (so the program will hang if it ever tries to lock the first flight again)

(b) (8 points) Even when all flights have enough seats left, the above `makeReservation` function can result in the program hanging if it is called in a particular way from multiple thread. Describe two simulatenous calls that would cause this. You may use shorthand like `makeReservation([A, B, C])` to represent a call to makeReservation with a vector of flights A, B, and C.

> **Solution:** makeReservation([A, B]) from one thread and makeReservation([B, A]) from another.

(c) (8 points) What is a way the hang described in the previous question (8(b)) be corrected? Explain briefly (at most one sentence).

> **Solution:** acquire locks in order such as by Flight* address (some other solutions were also accepted)

9. Consider a system which, like x86, which has:

- 32-bit virtual addresses
- a 2-level page table
- 20-bit virtual page numbers
- 4096 (2^{12}) byte pages
- 4-byte page table entries
- 4096 byte page tables at each level

On this system, the first 10 bits of a virtual page numberare used to look up the page table entry in the first level page table and the second 10 bits of are used to look up entries in the second level page table.

(a) (6 points) If the first-level page table entry, located at physical byte address `0x8010`, for virtual address `0x1020123` specifies that the second-level page table is located at physical byte address `0x4000`, then at what physical byte address is the second-level page table entry for this virtual address?

0x4080 (half-credit for 0x4020)

(b) (4 points) Suppose the second-level page table entry for virtual address `0x4444` is marked as not present (also known as 'invalid'). Suppose the OS wants to make reads and writes to this address work in spite of this. What could the OS do from its fault handler when a page fault occurs from the program trying to write to address `0x4448` as part of a pushq (push 8 bytes on the stack) instruction? Suppose that the OS determines that writes to this address should go to physical memory address `0x8448`.

- ◯ update physical address `0x8448` location from the kernel, then return from the fault handler
- ◯ update the program's saved stack pointer to contain `0x8450`, then return from the fault handler
- [✓ update the second-level page table entry to reference the page containing `0x8448`, then return from the fault handler]
- ◯ update the second-level page table entry to reference the page containing `0x8448`, then restart the faulting program from the beginning of main