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) Suppose an xv6 program executes a system call with the following C code:

```
char buffer[1];
int result = read(0, &buffer[0], 1);
```

which calls a system-call wrapper function `read`. What is true about this function? **Select all that apply.**

- ◯ as part of its execution, this function will store the system call number for read in a register or on the user stack
- ◯ this function will place the system call's arguments on the kernel stack before entering kernel mode
- ◯ this function is executed in both kernel and user mode
- ◯ this function will save all registers on the user stack before triggering an exception to enter kernel mode
- ◯ the function is located an address in the kernel's address space, so attempting to call it instead triggers an protection or page fault

2. (4 points) Consider an xv6 system running on a single core. Suppose an xv6 process A makes a `read()` system call, which requires it to wait until more input is available. While this happens, xv6 switches to another process B. Before the system call, a local variable in the user code of process A was stored in the callee-saved register `EBX`. Where will the `EBX` regsiter's value be while process B is executing?

- ◯ on the user stack of process A, because the compiler generated code to push it there before the system call
- ◯ on the kernel stack of process A, as a result of starting the system call
- ◯ on the kernel stack of process A, as a result of the context switch to process B
- ◯ on the user stack of process B, as a result of the context switch to process B
- ◯ on the kernel stack of process B, as a result of the context switch to process B
- ◯ on the kernel stack of process B, as a result of switching from kernel to user mode after the context switch
- ◯ in the processor's `EBX` register, since nothing will have happened to change `EBX`

3. (4 points) When attempting to write a shell for a POSIX-like system, a programmer discovers that after running a command like `programA > foo.txt`, any future command's output also goes to `foo.txt`. Which of the following is a likely cause of this problem?

- ◯ their shell did not wait for `programA` to finish before accepting another command
- ◯ their shell ran out of file descriptors, so opening foo.txt to run the command failed
- ◯ their shell did not `close` the file descriptor for foo.txt in the child process before executing `programA`
- ◯ `programA` called `dup2` itself, which changed the shell's standard output file descriptor
- ◯ their shell called `dup2` before forking instead of after forking

4. (7 points) Consider the following POSIX C code:

```
int fds[2];
pipe(fds);
pid_t pid = fork();
if (pid > 0) {
    dup2(fds[0], STDOUT_FILENO);
    waitpid(pid, NULL, 0);
} else {
    write(STDOUT_FILENO, "Hello!", strlen("Hello!"));
    write(fds[1], "Bye!", strlen("Bye!"));
    exit();
}
```

Recall that the write end of a `pipe` is returned in the second element of the array passed to `pipe` Assuming all `write`s, `fork`s, `dup2`s, and `waitpid`s succeed (and no partial writes occur), which of the following are possible outputs of this program? **Select all that apply.**

◯ `Hello!Bye!` ◯ `Hello!Hello!Bye!` ◯ `Bye!Hello!` ◯ `Bye!` ◯ `Hello!`
◯ `Hello!Bye!Hello!` ◯ (no output)

5. (3 points) On POSIX-like systems, the `write` and `read` system call wrapper functions are typically not used directly by programmers. Instead, it is more common for programmers to use functions like those in the C `stdio.h` header or the C++ `iostream` and `fstream` headers, which eventually call the system call wrapper functions. Which of the following are advantages or disadvantages of these higher level functions? **Select all that apply.**

   ◯ low-level functions like `read` may return before reading as much as requested from a pipe, and higher-level functions will handle this by calling the lower-level functions multiple times

   ◯ calling the higher-level functions many times is likely to be faster than calling low-level functions like `write` many times

   ◯ to avoid reading too much, when reading a line (as with `fgets` or `std::getline`), the higher-level functions must call a low-level function like `read()` once for each charcter, but a programmer using the lower-level functions directly could avoid this

6. (4 points) Which of the following is true about a scheduler that implements a round-robin scheduling policy? **Select all that apply.**

   ◯ this scheduler will minimize the number of context switches when scheduling processes that never sleep or wait for I/O

   ◯ this scheduler has higher throughput than a first-come first-served policy

   ◯ this scheduler will (approximately) evenly divide CPU time between processes that never sleep or wait for I/O

   ◯ it is possible to implement a round-robin scheduler such that selecting the next process to run and marking a process as runnable takes $O(1)$ time

7. (4 points) Suppose an operating system is trying to minimize **turnaround time** of the processes it is running. Then, which of the following are approaches its scheduler could take to help achieve this goal? **Select all that apply.**

   ◯ prefer to run processes that would start their next I/O operations sooner

   ◯ prefer to run more CPU-intensive processes before other processes

   ◯ prefer to run more processes that have been runnable less often than other processes

   ◯ when an I/O operation finishes, wait to schedule the newly runnable process until the currently running process is ready to yield the CPU

8. (4 points) A system can stop deadlocks by aborting tasks that are involved in the deadlock. However, this strategy causes additional problems, such as ____________. **Select all that apply.**

   ◯ a mechanism that can rollback the state changes it has made during the critical section is required.

   ◯ all resources must be acquired at once and in a consistent order

   ◯ livelock can occur in place of the deadlock

   ◯ a new scheduling algorithm is required to allow preemption

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

   - 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, **select all that apply** and do not account for how users might adjust what threads they run (other than specifying scheduler parameters, like priorities) based on the scheduler in use.

(a) (4 points) This kind of scheduler adjusts when it runs a thread next based the thread's behavior the last time it ran.

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

(b) (4 points) Across a typical desktop workload and, for schedulers which require configuration of paramteres like ticket counts, appropriate configuration of those parameters, this kind of scheduler is likely to achieve lower mean turnaround times than a first-come, first-served scheduler.

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

(c) (4 points) Programs running under this scheduler can affect the scheduler's decision by *performing additional I/O* to end up getting much more compute time than they would otherwise be able to.

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

10. (14 points) Consider the following C++ code:

```
pthread_mutex_t lk;
int global = 0;
void *thread_function(void *arg) {
    int *ptr = (int*) arg;
    pthread_mutex_lock(&lk);
    global = global + *ptr;
    cout << *ptr << "/" << global << " ";
    pthread_mutex_unlock(&lk);
    return NULL;
}

int main() {
    pthread_t threads[2];
    int i;
    pthread_mutex_init(&lk, NULL);
    pthread_mutex_lock(&lk);
    global = 100;
    for (i = 0; i < 2; i = i + 1) {
        pthread_create(&threads[i], NULL, thread_function, (void*) &i);
    }
    pthread_mutex_unlock(&lk);
    pthread_join(&threads[0], NULL);
    pthread_mutex_lock(&lk);
    std::cout << "[in main " << global << "] ";
    pthread_mutex_unlock(&lk);
    pthread_join(&threads[1], NULL);
    return 0;
}
```

Assume all the pthreads calls above do not fail and ignore any minor syntax errors. Which of the following are possible outputs? **Select all that apply.**

- ◯ `[in main 100] 0/0 1/1`
- ◯ `[in main 100] 2/102 2/104`
- ◯ `[in main 100] 0/100 1/101`
- ◯ `0/0 [in main 100] 1/1`
- ◯ `2/102 [in main 102] 2/104`
- ◯ `0/100 [in main 100] 1/101`
- ◯ `0/100 1/101 [in main 101]`
- ◯ `0/100 1/101 [in main 100]`
- ◯ `2/102 2/104 [in main 104]`
- ◯ (a segmentation fault)
- ◯ (it can hang before outputting anything)

11. (12 points) Consider a multiprocessor system that uses a cache coherency protocol where

- each processor has its own cache,
- processors are connected via a single shared bus,
- each cache block in a processor's cache is either in an Invalid (block not stored), Shared, or Modified state, and
- when possible, processors will request that a cached value be invalidated instead of updating it
- each cache block can hold 4 words

Suppose the following operations occur in the following order, starting from caches with no values cached. Assume address X_0 is in the first word of a cache block X, X_1 is in the second word of the cache block, and so on. Identify the state of the cache block in each processor after edch step below occurs, assuming all reads and writes are 1 word. The first one is done for you.

1. processor A reads from address X_0
   Processor A state: **Shared** Processor B state: **Invalid**
2. processor A reads from address X_2
   Processor A state: ____________ Processor B state: ____________
3. processor B reads from address X_1
   Processor A state: ____________ Processor B state: ____________
4. processor B writes to address X_2
   Processor A state: ____________ Processor B state: ____________
5. processor A reads from address X_0
   Processor A state: ____________ Processor B state: ____________

12. (20 points) Suppose we want to implement a variant of a reader/writer lock where instead of allowing any number of readers *or* exactly one writer to acquire the lock, we allow up to K readers *or* exactly one writer. When K readers are using the lock and a new reader asks the lock, it must wait until one reader leaves.

    In addition, in this reader/writer lock, writers have priority over readers.

    Fill in the blanks in the following partial implementation to complete it. Ignore any minor syntax errors, and assume all mutxes and condition variables are appropriately initialized. Avoid implementations that very frequently wake up threads when they cannot actually make progress immediately.

```
const int K = ...; // maximum number of readers
int readers, writers, waiting_readers, waiting_writers;
pthread_mutex_t lock;
pthread_cond_t reader_cv, writer_cv;
void ReadLock() {
    pthread_mutex_lock(&lock);
    waiting_readers += 1;

    ____________ (_____________________________________________________________) {
        pthread_cond_wait(&reader_cv, &lock);
    }
    waiting_readers -= 1;
    readers += 1;
    pthread_mutex_unlock(&lock);
}
void WriteLock() {
    pthread_mutex_lock(&lock);
    waiting_writers += 1;

    ______________ (___________________________________________________________) {
        pthread_cond_wait(&writer_cv, &lock);
    }
    waiting_writers -= 1;
    writers += 1;
    pthread_mutex_unlock(&lock);
}
void ReadUnlock() {
    pthread_mutex_lock(&lock);
    readers -= 1;

    if (_____________________________________________________________________) {
        pthread_cond_signal(&writer_cv);
    } else {

        ______________________________________________________________________
    }
    pthread_mutex_unlock(&lock);
}
void WriteUnlock() {
    pthread_mutex_lock(&lock);
    writers -= 1;

    if (waiting_writers > 0) {
        pthread_cond_signal(&writer_cv);
    } else {
        pthread_cond_broadcast(&reader_cv);
    }
    pthread_mutex_unlock(&lock);
}
```

13. (15 points) Suppose we want to implement a pair of functions, `SetFlag` and `WaitForFlag`. `SetFlag` sets a boolean flag which is initially false to true and makes threasd waiting for the flag to become true stop waiting. `WaitForFlag` does nothing if the flag is already true and otherwise waits until it is set by `SetFlag`. (Recall that `post` is the up operation and `wait` is the down operation.)

Fill in the blanks (marked A, B, and C) in the following implementation using counting semaphores: (Ignore minor syntax errors, missing error handling, and assume semaphores are initialized elsewhere as specified in the comments. You may choose not to use all of the blanks.)

```
bool flag = false;
int num_waiting = 0;
sem_t mutex /* initialized with count 1 */;
sem_t gate /* initialized with count  0 */;

void SetFlag() {
    sem_wait(&mutex);
    flag = true;
    while (num_waiting > 0) {

        __________________________________________   /* A */
        num_waiting -= 1;
    }
    sem_post(&mutex);
}

void WaitForFlag() {
    sem_wait(&mutex);
    if (flag == false) {
        num_waiting += 1;

        __________________________________________ /* B */

        __________________________________________ /* C */
    } else {
        sem_post(&mutex);
    }
}
```

14. Consider a system with virtual memory and (like xv6)

- 4096 byte (2^{12} byte) pages
- 4-byte page table entries
- two level page tables
- 2^{10} entries for the page tables at each level

Suppose the first-level page table entry corresponding to virtual byte address `0x12345`:

- has its present (valid) bit set
- has its writeable bit unset
- has its user-mode accessible bit set
- contains physical page number `0x55`

and the second-level page table entry corresponding to this address:

- has its present (valid) bit set
- has its writable bit unset
- has its user-mode-accessible bit set
- contains physical page number `0x44`

(a) (5 points) When a program performs a read from virtual byte address `0x12345`, the read will return the value at what physical byte address?

____

(b) (5 points) What is the physical byte address of the second-level page table entry for virtual byte address `0x12345`?

____

(c) (4 points) When a program performs a write to virtual byte address `0x12345`, this will cause a fault (a kind of exception). In response to the fault, suppoes the operating system wants to copy a page data around address `0x12345`, update the page table entries to point to this new copy, then continues execution of the program, retrying the failed memory access. The physical page number of the copy should be placed in ____________. **Select all that apply.**

- ◯ the page table entry in the second-level page table that follows the entry for `0x12345`
- ◯ the second-level page table entry for virtual addresss `0x12345`
- ◯ the first-level page table entry for vitrual address `0x12345`
- ◯ the first page table entry in the second-level page table corresponding to virtual address `0x12345`