Question 1. [8 marks]

TRUE [✓ FALSE] All system calls will cause a thread (or process) to block.

[✓ TRUE] FALSE An interrupt will lead to a switch from user-mode to kernel-mode.

[✓ TRUE] FALSE "." and ".." are hard links to directories.

TRUE [✓ FALSE] When we create a hard link, there can be multiple inodes for the same file.

TRUE [✓ FALSE] A lock prevents a thread from context switching so that it doesn't get interrupted while executing a critical section.

[✓ TRUE] FALSE A program containing a race condition may never result in data corruption or some other incorrect behavior.

TRUE [✓ FALSE] Any solution to the mutual exclusion problem that satisfies the progress requirement will also satisfy the starvation (or bounded waiting) requirement.

TRUE [✓ FALSE] A semaphore is initialized to 0 when we plan to use it in the same way as a lock.

Question 2. [3 marks]

For each of the pseudo code lock implementations below, check **all** the true statements. (Assume the locks are correctly initialized.)

Part (a) [1 mark]

```
int lock;
void acquire(lock) {
    while (lock);
    lock = 1;
}
void release (lock) {
    lock = 0;
}
```

- ☐ It works on machines with only one CPU (single core) because it disables context switches within critical region
- ☐ It works on machines with only one CPU even when there are context switches within critical region
- ☒ It does not work on machines with only one CPU
- ☒ It does not work on machines with multiple CPUs

Part (b) [1 mark]

```
int lock;
void acquire (lock) {
    while(testandset(&lock));
}

void release (lock) {
    lock = 0;
}
```

- ☐ It works on machines with only one CPU because it disables context switches within critical region
- ☒ It works on machines with only one CPU even when there are context switches within critical region
- ☐ It does not work on machines with only one CPU
- ☐ It does not work on machines with multiple CPUs

Part (c) [1 mark]

```
int lock;
void acquire(lock) {
    disable interrupts;
}
void release(lock) {
    enable interrupts;
}
```

- ☒ It works on machines with only one CPU because it disables context switches within critical region
- ☐ It works on machines with only one CPU even when there are context switches within critical region
- ☐ It does not work on machines with only one CPU
- ☒ It does not work on machines with multiple CPUs

Question 3. [5 marks]

In class we discussed the rendezvous problem. If `thread_a` and `thread_b` are running at the same time, we want to ensure that A1 is printed before B2 and that B1 is printed before A2. Re-write the functions below using locks and condition variables to implement the synchronization. Use the smallest number of locks and condition variables possible and do not introduce any unnecessary constraints. (C-like pseudo code is fine.)

```
void *thread_a(void *) {                      void *thread_b(void *) {
    fprintf(stderr, "A1\n");                      fprintf(stderr, "B1\n");
    fprintf(stderr, "A2\n")                       fprintf(stderr, "B2\n")
}                                             }
```

```
// Declare and initialize global variables
```

Marking notes:

- Some students

```
// The two threads are the same
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t wait_a = PTHREAD_COND_INITIALIZER;

int passed = 0;

void *thread_a(void *id_ptr) {
    fprintf(stderr, "A1\n");
    pthread_mutex_lock(&mutex);
    passed ++;
    if(passed < 2) {
        pthread_cond_wait(&wait_a, &mutex);
    }
    pthread_cond_signal(&wait_a);

    pthread_mutex_unlock(&mutex);

    fprintf(stderr, "A2\n");
    return NULL;
}
```

Question 4. [8 MARKS]

Consider the following description of the contents of an ext2 file system with a block size of 4096. The numbers in parentheses indicate the size of the directory or file in bytes. (131072 == 128 KiB)

For the questions below, assume that this is the entire file system and there are no reserved inodes other than the root inode.

```
/ (4096)
|__ largefile (131072)
|__ adir (4096)
     |__ emptydir (4096)
     |__ smallfile (2048)
```

Part (a) [4 MARKS]

How many inodes are in use?	5
How many data blocks are in use?	37 (3 (dirs) 1 (small) + 32 (large) + 1 (indirect))
What is the value of the links field in the inode for `emptydir`?	2
What is the value of the links field in the inode for `adir`?	3

Part (b) [2 MARKS]

Suppose we run `mkdir /adir/emptydir/bdir`. Identify the changes in the file system considering inodes, data blocks, and bit maps in your answer. Be specific.

- New inode for bdir
- New data block for bdir
- Modify inode for emptydir (links and last modified time)
- Modify data block for emptydir
- Update bitmaps for new inode and data block

Part (c) [2 MARKS]

For the operation in **Part (b)**, in which order would you write the blocks to the disk to minimize the chance of leaving the file system in an inconsistent state if the computer were to lose power before all blocks were written? Explain your rationale.

Write the data and new inode first, then bitmaps, then data block for emptydir, inode for emptydir. Exact order for some of these doesn't really matter. The key idea is write the new data block and inode first and then the links to them. The data block for emptydir should probably be last because that completes the linking.

Question 5. [5 MARKS]

Part (a) [1 MARK]

Briefly explain what happens when a process running in user-mode executes a privileged instruction.

It will generate an interrupt, and program will fail.

Common errors: Executing a system call is not the same thing as executing a privileged instruction. System calls are not privileged instructions.

Part (b) [1 MARK]

In assignment 1, some students proposed that each `a1fs_write` call that required a new block would start a new extent. Briefly explain the main problem with this approach.

We will be more likely to run out of extents even when the file is not very big, therefore limiting the maximum file size. This approach becomes much more like the indexed approach of ext2.

Marking notes: some students were concerned about slower access times due to more seeks.

Part (c) [1 MARK]

Can a process make the transition from the ready state to the blocked state? Explain why or why not.

No. A process moves to the blocked state because it made a system call that blocks or because it received a signal that will cause it to block (e.g. SIGSTOP). In either case, it must be running before it blocks.

Part (d) [2 MARKS]

Consider the dining philosopher's problem. In class, students proposed a solution where a philosopher must acquire a global lock before checking whether both their chopsticks are free. If both chopsticks are free, then the philosopher will release the global lock, eat and then put down the chopsticks. Otherwise, the philosopher will release the global lock and try again.

Explain the major drawback to this solution.

A global lock will reduce the potential concurrency. All of the philosophers must wait to acquire the lock before they can attempt to grab their chopsticks.

Marking notes: Some students believed this would cause deadlock or livelock.

Question 6. [4 marks]

Explain how the `fork` system call is implemented. Include the steps that are taken from the point that a process calls fork until the point that control is transferred back to a user process.

- System call causes switch to kernel mode
- registers are saved
- kernel creates a new process by making a copy of the PCB
- set the return value for both the original and new process
- initiates context switch (we haven't talked about the scheduler yet)
- context for the next process is restored