2 OS Concepts [6 points]

A (user-mode) program can invoke OS code only via a well-defined system call interface, but this same program can invoke any instruction of library code (e.g., it could jump to any arbitrary instruction of the `printf` function in the `libc` library). However, library code is often shared across programs (similar to programs sharing the same OS code). Answer the following questions.

Part A [3 points]

Explain why sharing of library code does not affect the correctness of other programs.

The shared library code is read only (similar to text region) and hence cannot be changed. The data and stack regions are modifiable but not shared across programs and hence a program can only affect itself.

Part B [3 points]

List **three** reasons why OS code cannot be shared with programs in the same way as library code.

- Some OS code needs to execute privileged instructions and hence will not run in user mode.
- The OS code may manipulate data that can be globally shared across programs (e.g., the scheduler run queue). Since libraries share code but not data, the library code would not be able to access this shared data.
- If the OS code is run in user mode, and it could be modified then the security guarantees provided by the OS could be compromised.

3 OS Concepts [5 points]

System calls ensure that OS code starts executing at well-defined entry points. However, the OS must perform additional operations to ensure that user programs cannot compromise (affect the correct behavior of) the OS. List **five** such operations that the OS should perform for the `mkdir` system call. The prototype for the system call is shown below.

```
// Create a new directory called "name" (e.g., "/tmp/ece344f")
// Set the permissions of the directory to "mode"
// (e.g., readable and writable by owner,
// not readable or writable by others)
mkdir(char *name, int mode)
```

- Check that name points to a string that lies within the address space of the process
- Check that the directory (e.g., /tmp/) exists
- Check that the filename (e.g., ece344f) does not already exist
- Check that the directory is writable by the user
- Check that the mode flags are within range

Question 10. OS/161 Design: User-level Synchronization [14 MARKS]

Assume that OS/161 has been extended to support a simple form of user-level shared memory that allows a user process to share parts of its address space with its children. The details of the shared memory support are not important, but processes sharing memory will need a way to coordinate their accesses to shared data. Your job is to design a user-level lock package that processes can use for this purpose.

Part (a) [2 MARKS] One option is to build the `lock_acquire` and `lock_release` functions entirely at user-level but this will require busy waiting in the `lock_acquire` function if the lock is already held. **Explain** why some form of busy waiting is needed in this case, and **briefly explain** why busy waiting is undesirable in a uniprocessor system like the one that OS/161 runs on.

A process has to busy wait in lock acquire because we have no way of suspending the process on a wait queue (or wait channel) from user-level. Busy waiting is bad in a uniprocessor because only one process can be executing at any time, and if a waiter is spinning, there is no way for the lock holder to make progress and release the lock.

A second option is to use system calls to request OS services to assist with user-level lock operations. Four system calls are needed, as listed in the box below:

```
int lock_init(int *lock_id);      /* returns 0 on success, and sets lock_id for
                                   * use in subsequent operations, or returns -1
                                   * on failure with errno set to a suitable value.
                                   */

int lock_acquire(int lock_id);    /* returns 0 when the lock specified by lock_id
                                   * has been acquired by the calling process, or
                                   * -1 on failure with errno set appropriately
                                   * (the lock is not acquired in case of failure).
                                   */

int lock_release(int lock_id);    /* releases the lock specified by lock_id and
                                   * returns 0 on success or -1 on failure with
                                   * errno set appropriately.
                                   */

int lock_destroy(int lock_id);    /* destroys the lock specified by lock_id and
                                   * cleans up any state maintained by the OS,
                                   * returns 0 on success or -1 on failure.
                                   */
```


This question continues on the following page

For each of the following questions, pseudo-code may help to present your answer more clearly, but a point-form English description is also acceptable. You may invent new error codes for this question, as long as they describe errors that nmight reasonably occur.

Part (b) [3 MARKS] Describe the operating system data structure(s) that you would use to keep track of lock ids. Make sure you describe (i) what is stored in the data structure, (ii) how it is accessed, and (iii) where this data structure is located.

You can use an array, similar to the open file table that was used to track le descriptors. The array stores pointers to "struct lock"s, i.e. the existing OS161 lock data structure. The lock_id is the index into the array. We can have either have a global table, which is accessed through a global variable and is shared by all processes, or a per-process table which is accessed through the "struct thread". [Per-process means unrelated threads can't interfere with the locking for a group of parent/child processes, but we have to make sure parent/child share the table after fork.]

Part (c) [3 MARKS] Outline an implementation of the system call handler for the `lock_init` call. (i.e., `sys_lock_init`). Identify one specific error that might reasonably be returned by `lock_init`.

1. Lock lock table (either global or current process's table, which might be shared with children)
2. Search lock table for an unused entry, e. Return ENOLOCKS if all entries in lock table are in use.
3. Create OS161 kernel lock. Return ENOMEM if create fails (or return error code returned by lock create).
4. Set pointer to new lock, e.g. lock table[e] = new lock
5. Unlock lock table
6. Return e as result of system call (e.g. *retval = e)

NOTE: Creating a lock can be omitted if the table is initialized by creating all the possible locks up front, but that is a terrible waste of memory and time.

This question continues on the following page

Part (d) [3 MARKS] Outline an implementation of the system call handler for the `lock_acquire` call (i.e. `sys_lock_acquire`). Identify one specific error that might reasonably be returned by `lock_acquire`.

```
if ( lock_id < MAX_LOCK_ID && lock_table[lockid] != NULL) {
    lock_acquire(locktable[lockid]);
} else {
    return EINVAL; /* invented error code like EBADLOCK also ok */
}
```

Part (e) [3 MARKS] Outline an implementation of the system call handler for the `lock_release` call (i.e. `sys_lock_release`). Identify one specific error that might reasonably be returned by `lock_release` but would not be returned by `lock_acquire`.

```
if (lock_id < MAX_LOCK_ID && lock_table[lock_id] != NULL) {
   if (lock_do_i_hold(lock_table[lock_id])) {
      lock_release(lock_table[lock_id]);
   } else {
      return EINVAL; /* ENOTOWNER okay */
   } else {
   return EINVAL; /* EBADLOCK okay */
}
```

[Use the space below for rough work.]

Question 4. Hardware and OS Design [8 MARKS]

You plan to start a company that will design a simple, power-efficient processor for next generation smartphones. Your design must allow the mobile operating system to isolate processes from each other and protect itself from malicious processes. To do so, you are convinced that the simplest processor design requires only one privileged instruction. Convince your funding partner that your design will work.

Part (a) [2 MARKS] Describe exactly what operation the privileged instruction will perform in your processor.

The privileged instruction should be able to setup the MMU registers so that the OS can ensure memory protection.

There should be no IO instructions and all IO and interrupt access should be memory mapped.

Part (b) [4 MARKS] Explain how the OS will prevent the following:

1. A process directly modifies kernel data:

The OS uses the MMU to ensure that kernel memory is not mapped to user space, so the process cannot access kernel memory.

2. A process directly executes arbitrary kernel code:

The OS uses the MMU to ensure that kernel memory is not mapped to user space, so the process cannot access kernel memory.

3. A process writes directly to a network device:

IO access is memory mapped. This memory range lies in kernel memory.

4. A process disables interrupts:

Interrupt access is memory mapped. This memory range lies in kernel memory.

Part (c) [2 MARKS] Describe one way by which a malicious process may still be able to compromise the OS.

OS has bugs or vulnerabilities. A process can issue a system call with bad/crafted arguments, that cause the OS to crash, reveal information to user space, etc.

Question 2. OS Design [10 MARKS]

Part (a) [3 MARKS] For the `write` system call shown below, list 3 unique checks that the OS must perform to ensure that an application does not crash or corrupt the OS.

```
int write(int fd, const void *buf, size_t count);
```

Check 1	fd is valid (fd points to a valid index in the file descriptor table).
Check 2	buf[0] to buf[count-1] lies in the address space of process.
Check 3	various other checks: file is not writeable/fd is not open for writing, writing to a file beyond the largest supported file size, etc.

Part (b) [3 MARKS] The timer interrupt is a key mechanism used by the OS. Usually, it waits some amount of time (say 10 milliseconds) and then interrupts the CPU. Suppose, the interrupt is not based on time but rather based on the number of TLB misses the CPU encounters. Once a certain number of TLB misses take place, the CPU is interrupted and the OS gets control (e.g., the page fault frequency replacement algorithm might be implemented this way). Will this design affect the timer interrupt and its usefulness?

1. The timer interrupt will no longer be time based, so programs depending on timing would be affected.
2. The timer interrupt may not fire if a program has not TLB misses, so the OS will never get control. This is a much more serious problem.

Part (c) [4 MARKS] In a Unix file system, the `/A/B/C/D.txt` file uses one block. Assume that each directory in this file system uses one block as well. To read the file `D.txt`, list the maximum and the minimum number of disk reads for different types of blocks (e.g., inode, data, etc.). In both cases, assume that the buffer cache is initially empty.

Maximum	
Block Type	# of reads
super block	1
inode block	5
directory block	4
data block	1

Minimum	
Block Type	# of reads
super block	1
inode block	1
directory block	4
data block	1

When all the five inodes are located in the same block, only one inode block read is required.

Question 3. OS Design [10 MARKS]

In this question, you will add signals to the threads library that you implemented in the OS labs.

Recall that Unix signals allow the OS to notify a process (or thread) of some event of interest. For instance, the kernel delivers the SIGTERM signal to a process when a human user types `Ctrl-C`. If a process doesn't handle this signal, the process is killed by the kernel. Alternatively, the process can choose to ignore the signal, or perform some action (e.g., print "I am unkillable") upon receiving this signal.

A process registers a signal handler function that is invoked when a signal is to be delivered to the process. The signal can be delivered either by the kernel (e.g., when a process accesses bad memory), or at the request of another process (e.g., bash shell sends `Ctrl-C` to child process). Note that signal delivery interrupts the process. For example, in your labs, the SIGALRM signal was used to implement preemptive threading.

Given this information, describe how you would implement signal support in your threads library. In particular, your library should allow a thread to signal another thread. Make sure to describe the minimal, but complete, set of changes, including to data structures, interfaces, etc., that you would make to your current threads implementation. Write your answer as a set of bullet points, preferably with a short caption for each point. Be as precise as possible. Hint: think about the entire lifetime of the signaling mechanism.

1. Add an interface function (e.g., thread_register_signal) that allows registering (and deregistering) a signal function.

2. Add an interface function (e.g., thread_signal) to signal another thread.

3. Your implementation of thread_exit already implements exiting another thread. You can simply reuse that implementation for implementing signals (this is what we were looking for in your answer). A call to thread_signal should be very similar to thread_exit, except that it should set a "signaled bit" (rather than the "exited bit") in the target thread.

4. When a signaled thread is run the next time and the signal function has been registered then the signal function should be run. (If the signal function is not registered, the signaled thread should exit.) The signaled bit should be unset.

5. When the signal function returns, the original code of the thread should be run again. To detect function return, a signal stub (similar to the thread_stub function) should be used to run the signal function.