# Homework Assignment 4

William Jagels

December 9, 2016

## 1.

### 1.1.

The server will not get the messages when the malicious process starts removing them from the queue. The client will think that its messages have been read, and perhaps expect a reply, but it will never get one because the malicious process intercepted the messages. This can throw the system into deadlock.

### 1.2.

The ipc_perm struct must have the permissions bits set to allow the uid/gid of the malicious process to read. For POSIX, the message queue can be set with `chown()` and `chmod()`. The permissions system works the same way as it does on regular files, as message queues in POSIX are just mountable virtual filesystems.

## 2.

1. Both can be opened by multiple processes at once
2. FIFOs require both ends to be opened before it can be used
3. Message queues have no path, just an identifier. A FIFO can exist anywhere on the filesystem, meaning it can be placed in the folder that the program is run from, eliminating issues with name conflicts.

## 3.

1. A binary semaphore can be manipulated by other processes while a mutex can only be touched by the threads of a single process.
2. Using a named seamphore, the programmer can persist state beyond the lifetime of the process, while mutexes are always in memory.
3. A binary semaphore can be safely used to make a program wait for a signal to release it, a mutex won't because the same process can't double-lock a mutex.

## 4.

### 4.1.

This is a producer-consumer example program where messages from `stdin` are passed via a shm segment to another program which dumps them onto `stdout`

### 4.2.

The semaphore is reserved before it is destroyed in order to prevent a double-destruction of that semaphore. This also protects the shared memory from a double-destruction attempt.

### 4.3.

No, because that would break safety. `shmp->cnt` will not be protected from other threads/processes when run. By reserving our semaphore before doing anything with shmp, we prevent the result from being changed by external forces.

## 5.

### 5.1.

This opens a message queue and reads messages from it ad infinitum. It sends all the received messages to `threadFunc()`, which prints the sizes of the received messages.

### 5.2.

When a message is sent to a process via the notification, the notification function is removed, so we need to re-register to continue getting messages from the message queue.

### 5.3.

No, because that would make this function MT-Unsafe. Also, we would need to know the `mq_msgsize` at compile time.

## 6.

### 6.1.

Without the call to `listen()`, the server will refuse incoming connections.

### 6.2.

Without the call to `bind()`, the server will be listening on an unbound port, resulting in the default socket handler to be used, most likely refusing the connection.

## 7.

### 7.1. To ensure proper cleanup of a thread, it must be either detached or joined with by its parent thread.

A thread can specify a cleanup handler with `pthread_cleanup_push()` which is arguably safer than trying to do it by hand. By pushing a cleanup handler, the thread does not have to be detached or joined to ensure proper cleanup, it can be canceled at any cancellation point safely.

### 7.2. Any thread in a process can cancel another thread as long as the thread being canceled has its cancel state set to `PTHREAD_CANCEL_ENABLE`.

This is true, as long as the thread hits a cancellation point or has the `PTHREAD_CANCEL_ASYNCHRONOUS` canceltype.

### 7.3. Unnamed POSIX semaphores can be used for synchronizing between both multiple threads and multiple processes.

No, you need to use a named POSIX semaphore if you want to synchronize across different processes, and the permissions also need to be set correctly if it's a different user's process.

### 7.4. The `connect()` function is only used with TCP connections.

No, it can be used with any protocol that implement `SOCK_STREAM`. `connect()` is only for protocols that require a connection, unlike a datagram protocol such as UDP.

### 7.5. If two processes attach a shared-memory segment, then that shared-memory segment can appear at different addresses in each process's virtual address-space.

This is true, it can also be at the same location because the first available location in the virtual address space will quite often be the same. The programmer can also ask for a specific location within the address space for the buffer to be placed at. It shouldn't matter where you put the buffer, because with offsets, you can access anywhere.