Assignment 2 – Written Answers

1. Consider two threads trying to insert a node into the list at the head. If the two threads succeed in adding an element that points to the head, we essentially have created a branch in the list where two threads have a pointer to two different heads. A linked list is not a branching data structure, so this should not be allowed. Insertion into the front of list should be guarded by a mutex. As well, if we are inserting a node, we do not want the head to dissapear if another thread deletes it. If that happens, the new inserted node will point to NULL since the head was deleted. So, when we insert, we should guard deletion of the head.

Now consider two threads trying to delete the same node. If one succeeds before the other, one will try to delete a NULL node, and we could end up with dangling pointers or a NULL pointer exception. When a node is being deleted, we should block access to that node as well as the one in front and behind it to guard against someone deleting the ‘next’ node and the ‘prev’ node.

The operations for a completely concurrent-safe linked list end up being confusing and complicated. We will need separate guard variables for each node, as well as guard variables that guard insertion and deletion. There are better data structures for concurrent use.

2. After running question 2, the file q2output.txt contains the following:

```
Process Stats
  PID NLWP
 xxxx    1
```

Where xxxx is the process ID of the question2 process. This is the output of a `ps` system command that shows the process ID as well as the Number of Light-Weight Processes. From the `ps` man page, the `NLWP` column describes the number of kernel level threads owned by the process. In the code, two threads are created. If these were in fact kernel level threads, `NLWP` would be greater than one. The OS only sees the question2 process and does not know that there are threads running within it; therefore, the created threads are user level threads.

3. Figure 1 below shows a pseudo-code semaphore implementation and Figure 2 shows a monitor implementation of the problem for question 3.

If either of the pre-processor or uploader are delayed, there is no reason for delaying the creation of files. Say the pre-processor slows for a single file. The file creator should be free to create another file, rather than have to wait for a file to be processed. If we want to create files at a rate independent of speed of the pre-processor uploader, we can have a file queue that the pre-processor checks at regular intervals for a file at the head of the queue. We can do the same thing if the uploader slows in relation to the pre-processor. We can have a queue of pre-processed files that the uploader can upload at its own pace. The uploader can check for files ready at the front of the queue, and block until a file is ready.

For both file queues, it will be necessary to have mutually exclusive access to the queues so that we do not get any race conditions from concurrent insert and remove operations. Inserting and removing from a queue is not an expensive operation so providing mutual exclusion with a mutex variable is a viable option. We also need to use our knowledge of bounded buffers to implement this.

It is necessary to deal with the issue of possibly delayed threads because the nature of threads is inherently non-determinstic. We must assume that any ordering and any amount of CPU time given to a thread is possible. In other words, it is always possible that one thread will run slower than another.

5. Python’s RLock object is known as a Reentrant Lock. RLocks differ from normal Locks in one way by the fact that RLocks are owned by a thread while Locks are not. What this means is that when an RLock is owned by a thread, another thread cannot unlock it until it owns it. As well, an RLock can be acquired multiple times by a single thread. If a single thread has multiple critical sections in it that must be guarded with mutual exclusion, RLocks allow for non-blocking calls to lock a mutex variable so that a recursive lock structure can be attained. RLocks can thus be used for recursion, whereas a normal Lock will get stuck in that situation.

```
Semaphore signal_1 = 0;
Sempahore signal_2 = 0;

// Program 1: File creator
createFile() {
      // Create an arbitrary file
      file = file.open();
      file.write_data();
      file.close();

      // Notify the pre-processor that file is created
      V(signal_1);
}

// Program 2: File pre-processor
processFile() {
      // Wait for file to be created
      P(signal_1);

      // Arbitrary file processing
      file = file.open();
      file.process_data();
      file.close();

      // Notify the uploader that file is ready to upload
      V(signal_2);
}

// Program 3: File uploader
uploadFile() {
      // Wait for a file to be processed
      P(signal_2);

      // Upload file to be uploaded
      upload_file();
}
```

(Figure 1: Q3 Semaphore Pseudo-Code Implementation – Daniel Lovegrove)

```
int done_1, done_2 = 0;
Mutex mutex_1, mutex_2;
Condvar cond_1, cond_2;

// Program 1: File creator
createFile() {
      // Create an arbitrary file
      mutex_1.lock();
      file = file.open();
      file.write_data();
      file.close();

      // Notify the pre-processor that file is created
      done_1 = 1;
      cond_1.signal()
      mutex_1.unlock();
}

// Program 2: File pre-processor
processFile() {
      // Wait for file to be created
      mutex_1.lock();
      while (done_1 == 0)
            cond_1.wait(mutex_1);

      // Arbitrary file processing
      file = file.open();
      file.process_data();
      file.close();
      done_1 = 0;
      mutex_1.unlock();

      // Notify the uploader that file is ready to upload
      mutex_2.lock();
      done_2 = 1;
      cond_2.signal();
      mutex_2.unlock();
}

// Program 3: File uploader
uploadFile() {
      // Wait for a file to be processed
      mutex_2.lock();
      while (done_2 == 0)
            cond_2.wait(mutex_2);

      // Upload file to be uploaded
      upload_file();
      done_2 = 0;
      mutex_2.unlock();
}
```

(Figure 2: Q3 Monitor Pseudo-Code Implementation – Daniel Lovegrove)