**CSCI 4727 Homework 4**

**Due: November 10**

The goal of this assignment is for you to understand how threaded programs work using shared memory and synchronization primitives. Below is some information about the pthreads library (including pthread mutex variables) and the associated semaphore library.

To use the pthread library, #include <pthread.h> and link with the flag –lpthread. To use the semaphore library, #include <semaphore.h>

pthread\_t An internal pthread struct, whose internals can safely be ignored, used as a thread identifier. Declare one for each thread to be created.

pthread\_create(pthread\_t \* p, NULL, void \*function, void \*argument) Creates a thread and dispatches it to run function with the given argument. The function ***must*** have the signature void \*function(void \*parameter). If you need to start that function with an actual argument, consider creating a struct variable, filling it with data, and passing it by address but cast to void \*.

pthread\_join(pthread\_t p, NULL) Waits for the indicated thread to complete.

pthread\_exit(void \*status) Kills the thread making this call.

pthread\_kill(pthread p, void\*\* status) Kills the indicated thread.

Example:

// in parent:

pthread\_t p;

if ( pthread\_create(&p, NULL, fn, NULL) < 0 ) { // Use fn( )as the startup function for p

// some error code here

}

pthread\_join(p, NULL); // Wait for p to finish

// in thread:

void\* fn(void){

// some thread code goes here

pthread\_exit(0);

}

Another example (passing an argument):

// in parent:

int i;

int\* v;

for (i=0; i<5; i++) { // Spawning 5 identical children

v = new int; // Get it from the heap

\*v = i; // Don’t send i

if ( pthread\_create(&p, NULL, fn, (void\*)v) < 0 ) {

// some error code

}

}

To pass a struct, do something similar, and cast its address to (void\*).

pthread\_mutex\_t A mutex (atomic lock) variable struct, whose contents can safely be ignored. Declare one for each mutex needed. A mutex should be statically initialized by setting it equal to PTHREAD\_MUTEX\_INITIALIZER.

pthread\_mutex\_lock(&pthread\_mutex\_t) Locks the mutex.

pthread\_mutex\_unlock(&pthread\_mutex\_t) Unlocks the mutex.

Example:

// in parent:

pthread\_mutex\_t mylock = PTHREAD\_MUTEX\_INITIALIZER; // Set to unlocked

// in thread code:

pthread\_mutex\_lock(&mylock);

// ... some critical region code goes inside here ...

pthread\_mutex\_unlock(&mylock);

The semaphore library is associated with pthreads, and it provides counting semaphores.

sem\_t An internal semaphore struct whose internals can safely be ignored. Declare one for each semaphore needed.

sem\_init(sem\_t \*, 0, unsigned int count) Initializes the indicated semaphore to count (note: ignore the second parameter for our purposes.) That is, set count to the number of shared resources, or the number of threads that can safely share one resource.

sem\_wait(sem\_t \*s) If the semaphore is 0, blocks the thread; otherwise decrements the semaphore’s value.

sem\_post(sem\_t \*s) This is what is usually called “signal”. Increments the semaphore’s value; if the old value was zero, wakes up one blocked thread (if any where actually blocked).

Example:

sem\_t mysem; // Make this a global variable

sem\_init(&mysem, 0, 1); // Since count = 1, this acts like a mutex, but count could be any positive int

sem\_wait(&mysem);

// ... some critical region code goes inside here ...

sem\_post(&mysem);

Re-write the solution for Homework 3 using threads instead of processes and shared memory instead of pipes. The parent should create ***seven threads*** – one for each stage, one to read the input file – note that this was not a separate child process in Homework3, but now is a separate thread - and it should initialize the “stations” field of each product record, and one to write the output file – also different from Homework 3. Instead of pipes, declare an ***array of queues*** of product\_record’s, one per stage, plus an output queue. When a product record needs to be sent on to another stage, it should be enqueued to the appropriate queue. When the last stage finishes with a record, it enqueues the widget on the output queue. The writer thread removes product records from this queue for writing to the output file. Make sure these queues are ***global, shared data***. Do not have stages busy-wait on empty queues – use ***counting semaphores*** for efficient waiting. Use ***mutexes*** to protect all access to shared memory. Use the all-done product\_record to signal the threads to quit, just like in Homework 3.

For the queues, use the queue data structure from the STL; use #include <queue>. You declare one like this:

queue<product\_record> myqueue; // Note: new not needed

and that the operations on it are

myqueue.push(rec); // Pushes a copy of rec

and

rec=myqueue.front( ); // Returns a copy of rec

myqueue.pop( ); // Actually removes rec

The latter takes two steps since pop() does not actually remove the thing at the front of the queue. Again, note that ***queue operations are not atomic*** (not thread-safe, to use another term) so they are critical regions – so protect each one with its own mutex.

Be ready to demonstrate your work in lab on the 10th and drop your code files in the drop box.