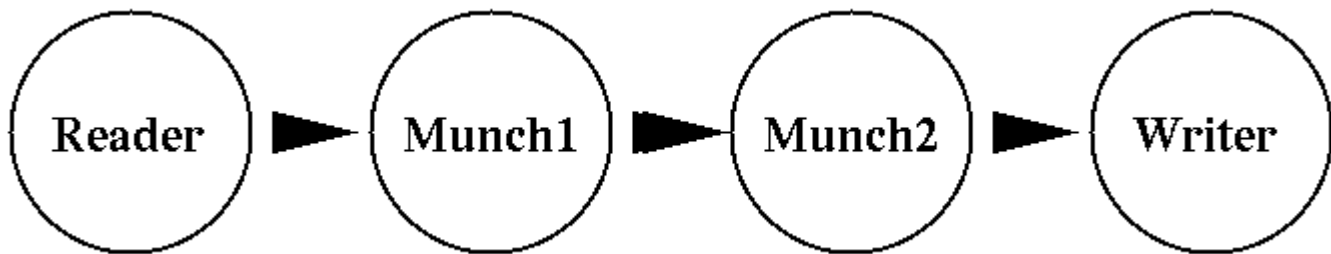


Programming Assignment #2
(Due Tuesday, October 13, at 5pm)

A [list of hints and helps for this assignment](#) is now available. It will be updated as I get more questions from the class.

Shared Memory Producer/Consumer Program

The goal of this assignment is to get experience in writing a program that actually runs in parallel on Linux using threads (pthreads) and synchronization. You will write a program with four threads, structured like:



- The Reader thread will read from standard input, one line at a time. Reader will take each line of the input and pass it to thread Munch1 through a queue of character strings.
- Munch1 will scan the line and replace each space character (not tabs or newlines) with an asterisk ("*") character. It will then pass the line to thread Munch2 through another queue of character strings.
- Munch2 will scan the line and convert all lower case letters to upper case (e.g., convert "a" to "A"). It will then pass the line to thread Writer through yet another queue of character strings.
- Writer will write the line to standard output.

Memory Allocation of Strings

Each time that the Reader thread is going to read a string from standard input, it will first `malloc` memory for that string. The pointer to that string will then be passed down the queues between the threads until it finally arrives at the Writer thread. After Writer writes out the string pointed to by the pointer, it will then `free` the memory that holds the string.

The Queue Module

One of the most important parts of your code is going to be the module (.c file) that implements synchronized queue. Since your program is reading in strings of characters and passing them from thread to thread, your queue will be a queue of pointers to strings (`char *`).

Queues are dynamically created. So, for this assignment, you will need one between each pair of threads (so you'll need three queues).

Queues will be represented by a structure of type `Queue`.

The main external functions for this module should be:

```
Queue *CreateStringQueue(int size)
```

Dynamically allocate a new Queue structure and initialize it with an array of character points of length size. That means you'll malloc the queue structure and then malloc the char ** array pointed to from that structure. Also remember to any state and synchronization variables used in this structure.

The function returns a pointer to the new queue structure.

For testing purposes, create your Queue's with a size of 10.

```
void EnqueueString(Queue *q, char *string)
```

This function places the pointer to the string at the end of queue q. If the queue is full, then this function blocks until there is space available.

```
char * DequeueString(Queue *q)
```

This function removes a pointer to a string from the beginning of queue q. If the queue is empty, then this function blocks until there is a string placed into the queue. This function returns the pointer that was removed from the queue.

```
void PrintQueueStats(Queue *q)
```

This function prints the statistics for this queue (see the next section for details).

Keeping Queue Statistics

Each Queue structure will also contain some extra fields for statistics-keeping. These fields will include:

enqueueCount

A count of the number of strings enqueued on this queue.

dequeueCount

A count of the number of strings dequeued on this queue. We would expect that when the program exits, the two count values are equal.

enqueueTime

The amount of elapsed (wallclock) time that a thread spent trying to do an enqueue. You will record the time at the start of the EnqueueString function and again at the end of the function, then take the difference.

dequeueTime

The amount of elapsed (wallclock) time that a thread spent trying to do an dequeue. You will record the time at the start of the DequeueString function and again at the end of the function, then take the difference.

Synchronization and Communication

The threads will communicate through shared memory using pthreads synchronization. For this program, the synchronization should be mainly contained in the when constructing queue module. You have two choices from

which to pick

1. **Semaphores:** You declare semaphores with the type `sem_t`, and operate on these semaphores with `sem_init`, `sem_wait`, and `sem_post`.

Using semaphores with the pthreads library is described in OSTEP Chapter 31. You'll want to refer to the Linux manual pages as well.

2. **Monitor synchronization:** Since you're programming in C, you **don't** have the compiler's help to automatically lock on entry to a monitor and unlock on exit. So you have to declare the monitor lock as type `pthread_mutex_t` and declare your condition variables as type `pthread_cond_t`.

You lock and unlock the monitor lock with `pthread_mutex_lock` and `pthread_mutex_unlock`.

You signal and wait on condition variables with `pthread_cond_signal` and `pthread_cond_wait`.

(**Don't** mistakenly try to use the `lock_t` type with condition variables.)

Using mutexs and condition variables with the pthreads library is described in OSTEP Chapters 27 and 30. You'll want to refer to the Linux manual pages as well.

Pick only **one of the above two choices** and use it consistently throughout your program. Do **not** mix them.

You cannot use busy waiting (spin locks) or somehow try to read the value of a semaphore, mutex, or condition variable.

Compiling Your Program

You will use `-Wall` and `-Wextra`.

On the link `gcc` command, you'll also need to specify the inclusion of the pthreads library by including the `-pthread` option.

Your executable file should be named **prodcom**.

Checking Your Code

Your code will need to scan clean of warnings from the `gcc -Wall` and `-Wextra` options **and** from the Clang Static Analyzer. Here is [more explanation on how to check the warnings and run Clang](#).

Good programmers in the real world check this *every time* that they compile their code, and then fix these warnings as they occur. Your makefile should include the commands to run Clang and view the results.

A Few More Program Details

1. Your main program will first create the three queues.
2. Your main program will then create four pthread threads. Your main thread will create these new threads by calling `pthread_create`, and then wait for these threads to finish by calling `pthread_join`. Each thread will call `pthread_exit` when it is done with all its work.
3. Thread Reader will read in each input line from `stdin`. You must check that the input line does not exceed the size of your buffer. If it does exceed the length, then you will reject that line by (1) printing out an error message to `stderr` and (2) throw away (flush to end of line) any remaining characters on that line. Note that the TA's will be testing your programs with input that contains extremely long lines.

4. See the manual page entry for the function "index" to making writing Munch1 easier.
5. See the manual page entries for "islower" and "toupper" to making writing Munch2 easier.
6. When there are no more strings to process, thread Writer will also print the number of strings processed to stdout.
7. Threads should terminate when there is no more input (end of file). Proper termination can be a bit tricky. Think about this one carefully.

Do **not** use a global variable (or multiple global variables) to signal completion of the input for the various threads. There are all sorts of bad things that can result from this technique.

8. Before the main program exits, it should print out statistics to stderr for each queue using the PrintQueueStats function.

Deliverables

You can work individually or in groups of two; in either case you will turn in a single copy of your program, clearly labeled with the name and logins of all authors.

You will turn in your programs, including all .c and .h files and your makefile. Also include a README file which describes a little bit about what you did for this project.

Note that you must run your programs on the Linux systems provided by the CS Department.

You should:

1. Submit only one copy of your code if you are working in pairs.
2. **Make sure for each partner you have the names, NetIDs and CompSci login names are in comments at the top of each source file.**
3. Each of you should create a file called partner.txt in **both** (if working in pair) of your proj2 directories, where you should have one or two lines that have the names, NetIDs and CompSci logins you and your partner (if applicable).

Handing in Your Assignment

Your CS537 handin directory is `~cs537-1/handin/your_login` where *your_login* is your CS login. Inside of that directory, you need to create a proj2 subdirectory.

Copying your files to this directory is accomplished with the cp program, as follows:

```
shell% cp *.ch makefile README ~cs537-1/handin/your_login/proj2
```

You can hand files in multiple times, and later submissions will overwrite your previous ones. To check that your files have been handed in properly, you should list `~cs537-1/handin/your_login/proj2` and make sure that your files are there.

Last modified: Fri Oct 2 11:57:47 CDT 2020 by [bart](#)