1 Detailed Review of Lab 3

1.1 Memory management with threads

In the previous lab, the subjects of creating and joining threads were explored. There are quite a few applications that these ideas could be applied to, but these applications are not exactly the most interesting. Most applications that do interesting things that use threading will most likely have to share variables between threads at some point. This lab will cover the subject of how to share information & variables between threads in a way that produces the desired result.

1.2 If you like it, then you shoulda put a mutex_lock on it

For the task in the previous lab, there was no reason for variables & information to be shared between multiple threads as each thread computed its own value in the result matrix without need of input from another thread. Though great, it's not a very interesting problem to solve. For more interesting problems, there is a need to share memory between threads. A trivial example of this would be to increment a shared variable between multiple threads. The program to do that is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>

void *threadCounter(void *param){
    int *args = (int *)param;
    int i;

    for(i = 0; i < 1000000; i++){
        (*args)++;
    }
}

int main(int argc, char** argv){
    pthread_t t1;
    pthread_t t2;
    int shared = 0;

    int err;

    err = pthread_create(&t1, NULL, threadCounter, (void *)&shared);
    if(err != 0){
        errno = err;
        perror("pthread_create");
        exit(1);
    }
    err = pthread_create(&t2, NULL, threadCounter, (void *)&shared);
    if(err != 0){
        errno = err;
        perror("pthread_create");
        exit(1);
```

```
    }

    err = pthread_join(t1, NULL);
    if(err != 0){
        errno = err;
        perror("pthread_join");
        exit(1);
    }
    err = pthread_join(t2, NULL);
    if(err != 0){
        errno = err;
        perror("pthread_join");
        exit(1);
    }

    printf("After both threads are done executing, `shared` = %d\n", shared);
    return 0;
}
```

Save this code snippet as `threaded_count.c`, compile it, and run the program a couple of times, observe the outputs and answer the following questions:

- what is the expected output?
- what is the calculated output?
- what caused the discrepancy between the expected and calculated values?

1.2.1 Ride into the critical section

One way to avoid the error that occurs in the threaded counter program is for the individual threads to lock the area of code where the accumulation of `shared` is performed, and unlock it once the accumulation is complete for that individual thread. This area is known as the critical section. To maximize performance, it is preferred that the critical section is as small as possible. To perform these locks, the following lines of code are needed:

```
pthread_mutex_t lock;
void *threadFunction(void *args){
.
.
.
pthread_mutex_lock(&lock);
//start of critical section
.
.
.
//end of critical section
pthread_mutex_unlock(&lock);
.
.
.
}
int main(int argc, char** argv){
.
```

```
.
.
err = pthread_mutex_init(&lock, NULL);
.
.
.
err = pthread_mutex_destroy(&lock);
return 0;
}
```

For more information regarding the init, lock, and unlock calls, consult `man 3p pthread_mutex_init`, `man 3p pthread_mutex_lock`, and `man 3p pthread_mutex_unlock` respectively. As seen above, the variable `lock` is declared as a global variable so that all the threads can get access to it. It is initialized in the main thread using `pthread_mutex_init`, and the threads use it to lock critical sections using `pthread_mutex_lock`. Once the critical section is complete, the critical section is unlocked using `pthread_mutex_unlock`. Finally, before the program exits, destroy the mutex using the `pthread_mutex_destroy` function call.

Now, add the mutex and the calls to `pthread_mutex_lock` and `pthread_mutex_unlock` to the counting thread functions in `threaded_count.c`, compile it, run it, and answer the following questions:

- Did this fix the issue with the original code?

1.3 See the threads in the streets, with not enough to do

Using mutexes to lock and unlock are great for avoiding race conditions, like what was shown in `threaded_count.c`, but it doesn't do a very good job of keeping threads from executing when we do not want them to.

Suppose that a program has one producer thread P, and two consumer threads C1 and C2. To ensure correctness of this program and to avoid duplicit computations, the critical sections of this program should be when P writes elements to the queue, and when either C1 or C2 reads from the queue. This would work fine, but there is a problem.

Suppose that the program was implemented naively, making the critical section of P, C1, and C2 are quite lengthy. During execution, P locks the mutex, produces data, writes to the queue, and releases the mutex. Then C1 gets to execute, going in to its critical section. While C1 is in its critical section, C2 gets scheduled to execute. Due to C1 still being in the critical section, C2 cannot get the lock, and thus cannot execute. A bit later, C1 finishes the execution of its critical section, and unlocks the mutex. Then P executes and goes into its critical section. While P is in its critical section, C2 gets scheduled to execute. Since P is in its critical section, C2 cannot get the lock and cannot execute. Then P finishes execution in its critical section, and unlocks the mutex. Then C1 goes next, and the cycle repeats. We see that C2 is never able to do anything, due to the fact that either P or C1 has the lock when C2 tries to get it. This situation is known as starvation. Since that is not desirable, conditional variables and semaphores should be used to avoid starvation.

1.3.1 Waiting on the conditional variable to change

Conditional variables are used to ensure that threads wait until a specific condition occurs. Using the example presented in the previous section, answer the following questions:

- What is the minimum number of conditions needed for the example to work as intended?
- What would those conditions be, and which thread(producer or consumer) should wait on that condition?

To use conditional variables, the following function calls are needed:

```
#include <pthread.h>

int test_var;
pthread_cond_t generic_condition;
pthread_mutex_t lock;

void *genericThread0(void *args){
  pthread_mutex_lock(&lock);
  //do awesome stuff
  pthread_cond_signal(&generic_condition);
  test_var = 1;
  pthread_mutex_unlock(&lock);
}

void *genericThread1(void *args){
  pthread_mutex_lock(&lock);
  while(test_var == 0){
    pthread_cond_wait(&generic_condition, &lock);
  }
  //does fun things
  pthread_mutex_unlock(&lock);
}
.
.
.
int main(int argc, char **argv){
  test_var = 0;
.
  err = pthread_mutex_init(&lock, NULL);
.
.
  err = pthread_cond_init(&generic_condition, NULL);
.
.
.
 err = pthread_cond_destroy(&generic_condition);
 return 0;
}
```

For more information regarding the cond_init, cond_wait, cond_signal(and in extension cond_broadcast), and cond_destroy please consult `man 3 pthread_cond_init`, `man 3 pthread_cond_wait`, `man 3 pthread_cond_signal`, and `man 3 pthread_cond_destroy` respectively.

As can be seen above, the variable `generic_condition` is declared as a global variable for the same reason that `lock` is declared as a global variable. `generic_condition` is then initialized in main by calling `pthread_cond_init`. `genericThread0` locks the mutex, does what it's supposed to do, then sets `test_var` to `1`, so that `genericThread1` can break out of the loop, signals the conditional variable, then unlocks the mutex.

`genericThread1` will attempt to lock the mutex, test the value of `test_var`, and call `pthread_cond_wait` to see if the conditional variable has been signaled. If not, the thread will block, and `pthread_cond_wait` will not return. However, according to the man pages, this block does not last forever, and should be re-evaluated each time `pthread_cond_wait` returns; hence the `while` loop that surrounds the call to `pthread_cond_wait`. If the conditional variable has been signaled, then `pthread_cond_wait` would return, and the thread calling it would get the mutex. The value of `test_var` would then be tested, fall through, fun things are performed, and the mutex is unlocked.

Once all is said and done, remove the conditional variable using `pthread_cond_destroy`.

To see an example of conditional variables, please take a look at `cond_example.c`. Make sure that everything pertaining to condition variables, such as how it's created, and used, is understood before compiling it. Run the compiled code, and put the output value of the program into your report.

1.3.2 Why not semaphore; you've got to declare yourself openly

Semaphores perform a similar task to conditional variables, and they are slightly easier to use. Semaphores come it two flavors, Named, and Unnamed. The differences between the two are in how they are created, and destroyed. For simplicity, the unnamed flavor of semaphores will be covered in this handout. To use semaphores, the following function calls and includes are needed:

```
#include <semaphore.h>

sem_t semaphore;
.
.
.
void *genericThread0(void *args){
 pthread_mutex_lock(&lock);
 err = sem_wait(&semaphore);
 ...
 //do awesome stuff
 err = sem_post(&semaphore);
 ...
 pthread_mutex_unlock(&lock);
}

void *genericThread1(void *args){
 err = sem_wait(&semaphore);
 pthread_mutex_lock(&lock);
 ...
 //does fun stuff
 err = sem_post(&semaphore);
```

```
 ...
 pthread_mutex_unlock(&lock);
}

int main(int argc, char **argv){
 .
 .
 .
 err = sem_init(&semaphore, 0, 1);
 .
 .
 .
 err = sem_destroy(&semaphore);
 ...
 return 0;
}
```

For more information on the init, wait, post, and destroy functions, consult `man 3 sem_init`, `man 3 sem_wait`, `man 3 sem_post`, and `man 3 sem_destroy` respectively.

Note that the calls to`pthread_mutex_lock` and `pthread_mutex_unlock` do not necessarily have to be where they are shown in the above code, i.e., the mutex lock does not have to occur before the semaphore wait, and the mutex unlock doesn't have to occur after the semaphore post.

For similar reasons to `lock` and `generic_condition`, `semaphore` is declared as a global variable. It is initialized in main using `sem_init` with the value for *pshared* set to `0`(meaning the semaphore is only shared between the threads of the current process), and it's initial value will be `1`(last argument to `sem_init`).

`genericThread0` decrements `semaphore` by one using `sem_wait`. If `genericThread1` attempts to decrement `semaphore` when it got to its call to `sem_wait` before `genericThread0` incremented the semaphore by calling `sem_post`, then `genericThread1` will block right at its `sem_wait` line. This is because decrementing a semaphore past `0` will not evaluate. So if a semaphore is already at a value of `0`, decrementing it with `sem_wait` will cause the thread calling `sem_wait` to wait until the value of the semaphore is incremented to a value greater than `0`.

Once everything is completed, destroy the semaphore by calling `sem_destroy` on `semaphore`.

For an example of semaphores being used, take a look at `sem_example.c`. Please make sure that everything in the program `sem_example.c` pertaining to semaphores is understood before compiling it. Run the program, and note the order in which the buffer is read/written to. Run the program multiple times, and again not the order in which the buffer is read/written to. Do they look different? Why do you think that is the case?

2 Tasks for this lab

2.1 Print Server

This week's lab is to write a multi-threaded print server program. The server will take in jobs from stdin and send them to printer drivers that will print them out. The printers are configured using the `config.rc` file. Printers are arranged into groups where each group contains one or more printer. When a job is sent it must include a `PRINTER` tag which will select the group that that job can be printed to. Any printer in that group is able to print the job.

2.1.1 What you are given

Please look through and understand the code in the `src` directory. The code you are given will spawn a consumer thread for each printer and one producer thread that will read print jobs from stdin. It is your job to implement the pushing and popping to the `job_queue` in the producer and consumer threads.

2.1.2 Makefile

Everything you have done for this class so far were simple one file programs. This time the project is larger and being split into multiple files. The supplied makefile will compile all of the source files and link them together to create a executable file. You should look at the makefile, but you do not have to make any changes to it for this lab. This make file can do the following:

```
# compile the code
$ make
# remove all binaries and object files
$ make clean
# generate the documentation
$ make doc
```

2.2 Input file format

The files to be printed will be supplied through `FILE` flag. The files must be in a PostScript format, a standard ASCII document format.

2.3 Accepted flags

The print server gets its print jobs from stdin by default. The jobs are given as a series of optional and required tags. The accepted tags are

```
- NEW: (required) start a new print job
- FILE: (required) the .ps file to print
- NAME: (required) the name of the print job
- DESCRIPTION: (optiona) a discription of the print job
- PRINTER: (required) the name of the printer group to print the job
- PRINT: (required) send the print job to the printer
```

2.4 What you should do

First read all of the code and understand what it currently does and read the rest of this document. Go through the code and fix all of the `warning` tags and implement the functionality of the program. The provided test script only tests a very basic case; you should write a better test script that will test all of the features of the code. You should also take in a `-l` flag from the command line that sets a log file output. In that file you should have the consumer threads print detailed information about each print job including it's arrival time, finish time, and elapsed time.

To ensure that the printers are created before running `main`, navigate to the `printer` directory, and run the following commands

```
# make the drivers directory
$ mkdir drivers
# compile the virtual printer
$ make
# to run the virtual printer; replace X with a positive number that matches with
what's in the `config.rc` file
$ ./virt-printer -n printerX
```

The `drivers` directory only needs to be run once, along with the call to compile the printer. For each printer in the rc file, the command to run the printer should be called. This will guarantee that files are printed correctly, and that no errors occur when starting up `main`.

You should write a summary answering all questions in this lab write up and detailing how you solved this problem. If you made any changes to the code explain what you changed and why. Talk about any issues you ran into and how you overcame them.

2.5 What to submit

Submit the following via BlackBoard:

- Any code you have written.
  - Your code should be well commented with doxygen style comment (see http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html)
  - Your code must be backwards compatible with the supplied API
- The test script you wrote to test and any additional files it takes (such as additional .ps files).
- Your lab summary
- A README explaining how to use your test script

2.6 Additional resources

The man pages are going to be your best friend for this lab. In addition to the man pages, the following sites will also be helpful:

- https://computing.llnl.gov/tutorials/pthreads/

- https://www.gnu.org/software/libc/manual/pdf/libc.pdf
- https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc.pdf

2.7 Extra credit

There are a couple of ways to get extra credit for this lab. One is to support additional printer drivers. For example you could use ghostwriter to take in a Postscript file and output an ascii text file (`ps2ascii`). Another way to get extra credit is to support additional **useful** command line arguments. Please document any additional features in your lab report so the TAs know to grade them. You can also use the libresuse to better profile the system and memory usage of this program and look for ways to improve it.

3 License