IE411 (OS), W22, Assignment 3: Threads and Semaphores

This lab assignment intends to familiarize you with threads and semaphores.

What you need to do:

1) Pre-lab Task (0 pts): Consult the manual pages on a Linux machine to understand the following library functions:

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`
- `int pthread_join(pthread_t thread, void **value_ptr);`
- `void pthread_exit(void *value_ptr);`
- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- `int sem_wait(sem_t *sem);`
- `int sem_post(sem_t *sem);`

2) In-lab Task (0 pts): In the lab session on March 25th, the TAs will explain how to solve part 0. Get started on part 1.

3) Post lab Task (10 pts): Complete parts 1 and 3 and submit your complete working solutions to part 0, part 1 and part 3 via classroom. This is due 31st March, 11:55 pm.

Creating threads

In pthreads, new threads are created by calling the function pthread_create. The actual definition of pthread_create is quite verbose:

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

Let's start at the beginning. pthread_create returns an int, which is used to determine if a thread was successfully created (0 for yes, non-zero for no).

- The first argument is a pointer to a pthread_t structure. This structure holds all kinds of useful information about a thread (mostly used by the threading internals). So, for each thread you create, you'll need to allocate one of these structures to store all of its information.
- The second argument is a pointer to a pthread_attr_t struct. This structure contains the attributes of the thread. This has even more information about the thread. The only really interesting thing, for us at least, is that the pthread_attr_t struct tells the system how much memory to allocate for the thread's stack. By default, this is usually 512K. This is actually quite large, and if you run with lots of threads you can run out of memory just from allocating thread stacks.

- The third argument to pthread_create is even more mysterious. Lets look at it again: void *(*start_routine)(void *). This is actually the function that the thread should start executing. This argument is an instance of what C programmers call a function pointer. The name of the argument is actually start_routine, and it is a function that returns void * and takes a void * as an argument.
- The last argument to pthread_create is a void * which is the argument to be passed to start_routine when the thread gets going. So, why a void *? Because in C, void * basically means anything. Remember you can cast pointers to void *, but you can also cast ints, chars, etc. to a void *. Which means that it is really the only way to specify a generic argument in C.

Part 0

Consider the following program (part0.c):

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

void* first(void* data) { printf("First\n"); }
void* second(void* data) { printf("Second\n"); }
void* third(void* data) { printf("Third\n"); }

int main () {
   pthread_t t1, t2, t3;

   pthread_create(&t3, NULL, third, NULL);
   pthread_create(&t2, NULL, second, NULL);
   pthread_create(&t1, NULL, first, NULL);

   /* wait for all threads */
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);
}
```

Your task is to make sure that after executing the compiled program it always outputs:

```
First
Second
Third
```

Part 1

Write a program (p1.c) that spawns two threads. The first thread outputs all even numbers up to 100, the second thread outputs all odd numbers. Use semaphores to make sure that the numbers are printed in order.

Part 2

Write a program (p2.c) that will take a list of filenames on the command line. If the list is empty, print an error message and exit. The goal is to compute how many lines are in each file using a thread for each file. Create a thread for each argument given on the command line. Each thread should open its filename. If the open succeeds, the thread should read the file and count the number of newlines. When EOF is reached, the thread should print its filename with the number of newlines in the file, and terminate the message with a newline. The thread can then exit.

Part 3

Add to a correctly functioning Part 2 the following feature (part3.c): have the threads coordinate to report a single global sum of how many newlines there are in total in all files. Your solution should use a semaphore to protect threads adding their local count to the global count. Also have the main thread wait for the global count to be complete, then have the main thread print the global count.

Do not attempt Part 3 unless Part 2 works completely and correctly.