IE 411 (Winter 2020-21), Lab 4: Threads, semaphores and CVs

This objective of this lab is to familiarize you with threads, semaphores and condition variables.

To get checked off on this lab:

- Complete task 1 during the lab session and demonstrate your solution to a TA.
- Turn in your solution for tasks 2 and 3 via classroom. This is due 20th March, 11:59 pm. Late submissions will not be accepted.

You may refer to 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 pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);`
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- `int sem_wait(sem_t *sem);`
- `int sem_post(sem_t *sem);`

Task #0 (will be discussed by TA)

Consider the code below.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM 50

void *deposit(void *amount);
double balance=10;

int main(){

 pthread_t tids[NUM];
 double amount=10;
 int i=0;
 int* status;

 printf("Begin thread creation.\n");
 for (i=0;i<NUM;i++){
   pthread_create(&tids[i], NULL, deposit, &amount);
 }
 printf("Begin thread joining.\n");
 for (i=0; i<NUM; i++){

   /* Notice: NOT &tids[i] */
   pthread_join(tids[i], (void**) &status);    /* LINE A */
   printf("Joined %d, with status = %d\n", i, *status);

   /* Why free()? Discuss */
   free(status);
 }
 printf("final balance = %lf\n", balance);
 return 0;
}

void *deposit(void *amount){
  double am= *((double *) amount);  /* LINE B */
  /* Can't return address of a local variable */
  /* Therefore malloc()ing on the heap. */
  int *retstatus=(int *)malloc(sizeof(int));

  double local=0;
  local=balance;
  local+=am;
  balance=local;
  *retstatus=balance;
  pthread_exit(retstatus);
}
```

Review Questions:

- How is the thread returning a value back to the main thread?
- Explain what is happening on LINE B.
- Why is deposit using `malloc`?
- If we run the above code multiple times (try at least 10000 times), does the balance print out to be the same? If not, what are we witnessing?

Task #1 (to be completed within the lab session)

Consider the following program (order.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
```

Your solution should use semaphores.

Task #2 (solution to be turned in via classroom)

Write a program, in a file `f-test.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.

When you have finished the above part, 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.

Task #3 (solution to be turned in via classroom)

Source code for the program is available in the lab directory, in the file `p-test.c`. The program comments explain what it does.

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

/****
 * The program takes an integer command-line argument, N.  The program creates
 * a thread to print consecutive odd numbers from 1 to N.  The main thread
 * prints consecutive even integers from 2 to N.
 *
 * The main thread and created thread run asynchronously. This means that
 * there is a race condition between the threads, such that standard output of
 * odd and even numbers is unpredictably interspersed.
 *
 */

/**
 * Print consecutive odd integers from 1 to arg.  The signature of this
 * function is that required for a thread start routine.
 */

void* print_odds(void* arg) {
   int i;
   int n = *((int*) arg);

   for (i = 1; i <= n; i += 2)
   {
      printf("%d0, i);
   }

   return NULL;
}
```

```
/*
 * Print consecutive odd and even integers, up to n.  A created thread prints
 * the odds, the main thread prints the evens.
 */
void print_numbers(int n)
{
    int i;              /* loop index */
    pthread_t tid;      /* id for thread to be created */
    int *arg;           /* argument sent to thread start routine */

    /*
     * Allocate storage for the start routine argument, which must be void*.
     */
    if ((arg = malloc(sizeof(int))) == NULL) {
       perror("malloc");
       exit(-1);
    }

    /* Initialize the argument. */
    *arg = n;

    /*
     * Create the thread, which means its start routine begins concurrent
     * execution.
     */
    if (pthread_create(&tid, NULL, print_odds, arg)) {
       perror("pthread_create");
       exit(-1);
    }

    /*
     * Back in the main thread, print out even integers, concurrently with the
     * odd printing in the thread.
     */
    for (i = 2; i <= n; i += 2) {
       printf("%d0, i);
    }

    /* Wait for the thread to terminate. */
    if (pthread_join(tid, NULL)) {
       perror("pthread_join");
       exit(-1);
    }
}
```

```
/* Get the single command-line argument and call print_numbers with it. */
int main(int argc, char *argv[]) {

    int n;              /* integer value of command-line arg */
    char* end;          /* for use with strtol */

    /* Make sure there is exactly one command-line arg. */
    if (argc != 2) {
       fprintf(stderr, "usage: %s <number>0, argv[0]);
       exit(1);
    }

    /* Convert and validate arg as an integer. */
    n = strtol(argv[1], &end, 10);
    if (*end != ' ') {
       fprintf(stderr, "%s - %s is not a valid integer0, argv[0], argv[1]);
       exit(1);
    }

    /* Do the deed. */
    print_numbers(n);

    return EXIT_SUCCESS;
}
```

Starting with the `p-test` program above, write a program named `m-test` that uses condition variables to guarantee that the threads will take turns printing each integer. The program begins by having the main thread print `0`. Then the created thread prints `1`. Printing continues with each thread printing only a single number each turn.

Your solution must work without any sort of busy wait. The following threading functions would be useful.

Function	**Description**
`pthread_mutex_init`	create a mutex
`pthread_mutex_lock`	lock a mutex
`pthread_cond_wait`	wait on a locked mutex
`pthread_mutex_unlock`	unlock a mutex
`pthread_cond_signal`	wake up a waiting thread