Operating System Lab

Lab #7

Process Synchronization I

Goal:

The lab is to teach you how to use pthread and protected the shared variable.

Preparation:

Prepare the working environment

- Check out the repo from Github classroom
  - Use `git clone {repo URL}`

- Rebuild Docker image and run the container
  - In a Windows Powershell or MacOS/Liunx terminal, change the current folder to the "lab" folder (the one you check out).
    `cd {lab_folder}`
  - For **Mac** or **Linux**
    - You can execute the shell script "clean_start.sh" if you want to remove old image and create and run a new one.
      `./clean_start.sh`
    - Otherwise, you can just run the following "docker run" command to run a container.
  - For **Windows**
    - You need the following docker commands to run the container from the existing image, *ubuntu_os*.

```
docker run -itd -p8888:8888 --rm --name ubuntu-os-con -v"$PWD/share":/home/app ubuntu_os
```

    - If you do not have the image *ubuntu_os*, you need to rebuild the image

```
docker build -t ubuntu_os .
```

  - To enter the "Bash Shell" of the Ubuntu container for compiling and testing codes

```
docker exec -it ubuntu-os-con /bin/bash
```

- Use VSCode or any text editors for development
  - In the host OS (Windows or MacOS), use terminal to change to the host's lab folder "{check out repo folder}/share" or VSCode to open that host's lab folder. Files you created or modified in the "{check out repo folder}/share" will be shown in docker container's "/home/app" folder.
  - Copy required files from "{check out repo folder}/share" to "{check out repo folder}/answer" and check in for submission.

Part I: Thread

1. Review or Learn Thread Basic Methods

In this section, you will see how to do the following with POSIX threads (pthreads):

- Create threads
- Exit from a thread correctly
- Wait for a thread to finish and interpret what it returns (join it)
- Tell a thread you will not wait for it (detach it)

Some basic pthreads methods:

- **pthread_create()** - fork() + exec*() for threads
  - *int pthread_create(pthread_t* thread,*
    *const pthread_attr_t * attr,*
    *void * (*start_routine)(void *),*
    *void *arg);*
  - Arguments:
    - **thread** - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
    - **attr** - Set to NULL if default thread attributes are used
    - **void * (*start_routine)** - pointer to the function to be threaded. Function has a single argument: pointer to void.
    - ***arg** - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.
- **pthread_self()** - get thread id
- **pthread_exit()** - exit() for threads
  - *void pthread_exit(void *retval);*
  - Arguments:
    - **retval** – return value of thread
- **pthread_join()** - wait() for threads. Use to wait for the termination of a thread.
  - *int pthread_join(pthread_t thread, void **retval);*
  - Arguments:
    - **thread –** thread id of the read for which the current thread waits.
    - **retval -** pointer to the location where the exit status of the thread mentioned in thread is stored.
- **pthread_detach()** - tell a thread you won't join it (wait for it)
- …

To use these commands you must include **pthreads.h**.
Also in Linux, to compile a program using pthreads. you need to compile with -pthread to links against the pthread library and enable necessary pthread features in some headers:

```
gcc -pthread example.c -o example
```

An example of using pthreads is like the following:

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

void *print_message_function( void *ptr );

void main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n", iret1);
     printf("Thread 2 returns: %d\n", iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s from thread id %ld\n", message, pthread_self());
}
```

Figure 1: example of using pthreads

To-do (Part1):

1. You learn how the pthread works by reading and typing, write the code in Figure 1 and name the program "*example1.c*", compile it with -lpthread in the gcc and then use "script" command to record your execution results in a file called "*example1_results*".
2. Write another program called "*n_thread_print.c*" that can take a command line input N (N<6) and then create N threads to print message like the following execution. (Note: bold font is input value)
   Hint: you might need to use snprint() and malloc() to create a message.

```
How many threads?
5
Thread 4 say hello from thread id 140278148118272
Thread 0 say hello from thread id 140278181689088
```

```
Thread 3 say hello from thread id 140278156510976
Thread 2 say hello from thread id 140278164903680
Thread 1 say hello from thread id 140278173296384
```

Part II: Shared Variable

1. Threads on Shared Variables

One of the advantages for using threads is that they can share information via global variables. The following program (see Figure2) shows one way to use a shared variable sum.

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

// Compile with -pthread
int sum = 0; //shared

void *countgold(void *param) {
    int i;
    for (i = 0; i < 10000000; i++) {
        sum += 1;
    }
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

    //Wait for both threads to finish:
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("ARRRRG sum is %d\n", sum);
    return 0;
}
```

Figure 2: Increment a shared variable example.

2. Protecting Shared Variables: mutex

It is important to synchronize access to resources between threads. You need to worry about access to global variables now as well. The POSIX threads API offers many methods to synchronize access to resources. In this section, we focus on the mutex. A mutex is like a single binary semaphore with a couple small differences:

- 0 means the mutex is available, 1 means it is locked - this is the opposite of how semaphores work
- Only the thread that locks a mutex can unlock it whereas any process or thread can release a semaphore

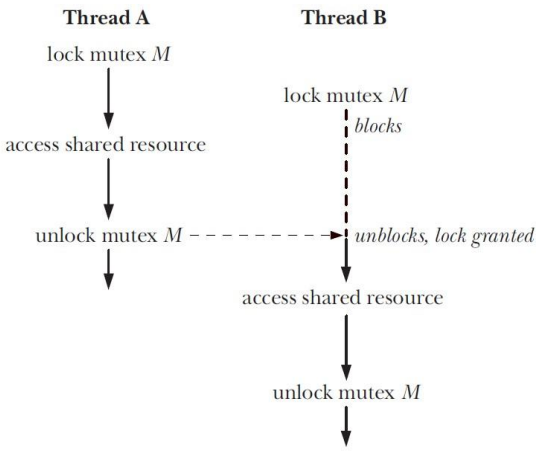

Figure 3: Mutex between two threads

The pthreads mutex methods:

- **pthread_mutex_init()** - used to create a new mutex
  - *int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);*
  - Arguments:
    - **mutex**: refers to a real variable of type pthread_mutex_t
    - **attr**: can refer to a custom attrubutes for the new mutex
      - Providing a NULL pointer results in a mutex with default attributes. This is good enough for now.
- **pthread_mutex_lock() -** blocks until the mutex is available
  - *int pthread_mutex_lock(pthread_mutex_t *mutex);*
- **pthread_mutex_trylock() -** used to get and lock access to serial (one at a time) resources
  - *int pthread_mutex_trylock(pthread_mutex_t *mutex);*
  - Arguments:
    - **mutex**: refers to a valid mutex created by pthread_mutex_init
- **pthread_mutex_unlock()** - used to unlock access to serial (one at a time) resources. Only the thread that locked a mutex may unlock it.
  - *int pthread_mutex_unlock(pthread_mutex_t *mutex);*
  - Arguments:
    - **mutex**: refers to a valid mutex created by pthread_mutex_init
- **pthread_mutex_destroy()** - used to free up unneeded mutexes, a locked mutex will not be destroyed - the attempt will fail.
  - *int pthread_mutex_destroy(pthread_mutex_t *mutex);*
  - Arguments:
    - **mutex**: refers to a valid mutex created by pthread_mutex_init

A mutex is a variable of the type pthread mutex t. Before it can be used, a mutex must always be initialized. For a statically allocated mutex, we can do this by assigning it the value

PTHREAD_MUTEX_INITIALIZER, as in the following example:

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
```

After initialization, a mutex is unlocked. To lock and unlock a mutex, we use the pthread_mutex_lock()

and pthread_mutex_unlock() functions as follow:

```
pthread_mutex_lock( &mutex1 );
```

```
pthread_mutex_unlock( &mutex1 );
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t thread_ids[3];
int counter = 0;
pthread_mutex_t lock;

void* doJob(void* arg)  {
    pthread_mutex_lock(&lock);
    counter++;
    printf("Job %d has started\n", counter);
    sleep(2);
    printf("Job %d has finished\n", counter);
    pthread_mutex_unlock(&lock);
}

int main(void) {
    int error1 = pthread_create(&(thread_ids[0]), NULL, &doJob, NULL);
    int error2 = pthread_create(&(thread_ids[1]), NULL, &doJob, NULL);
    int error3 = pthread_create(&(thread_ids[2]), NULL, &doJob, NULL);
    if (error1 != 0 || error1 !=0 || error2 !=0) { //Not good but okay for demo
        printf("\nThread can't be created");
    }

    pthread_join(thread_ids[0], NULL);
    pthread_join(thread_ids[1], NULL);
    pthread_join(thread_ids[2], NULL);

    return 0;
}
```

Figure 4: Mutex between two threads

To-do (Part 2):

The following questions require a text file called "*answer.md*" for putting your answers.

1. Write the code in Figure 2, name "*example2.c*" compile and link it with -lpthread and answer the following questions.
   After running the code, please answer the following questions.
   (a) What is expected to be the output for sum value? (Put your answer under the part2-1-a: in the "answer.md")
   (b) What is the major problem regarding the code? (Put your answer under the part2-1-b: in the "answer.md")
2. Create a program called "*example3.c*" by following codes in the Figure 4.
   - Use script command to record and save your execution results in the file "*ex3_has_mutex*".
   - Then, take out the mutex lock. Use script command to save your execution results in the file "*ex3_no_mutex*". This result should have no mutex lock.
   - Describe what you find by comparing these two results and put your answer in the answer.md file under the item to-do-part2-2:)
3. Define a mutex variable. and modify the code (*example2.c*) in Figure 2, called "*fix_example2.c*", to protect the shared variable. Run your code and verify your code is working correctly.

Reminder:

<u>This lab has many files in the submission. Therefore, we list them in the following in the *"answer"* folder.</u>

- A text file for your written answer:
  - *answer.md*
- Programs:
  - *example1.c*
  - *example2.c*
  - *n_thread_print.c*
  - *example3.c*
  - *fix_example2.c*
- Script recording results:
  - *example1_results*
  - *ex3_has_mutex*
  - *ex3_no_mutex*