Programming Project 2

Instructions for this assignment:

For this assignment, you will work in your assigned group of up to four students (including yourself). Submit the assignment by following the steps listed under "Submission Procedure" before the due date/time; late submissions will receive a grade of zero, but on-time submissions will receive partial credit even if incomplete.

Complete either Option 1 or Option 2 below (whichever your group finds most interesting) for full credit. (You're welcome to complete both options if you like, but no extra credit will be given!)

Option 1

Option 1 comprises two steps as follows.

Step 1

Complete "Programming Projects: Introduction to Linux Kernel Modules" Part I – "Kernel Modules Overview," and Part II – "Loading and Removing Kernel Modules," near the end of Chapter 2 (section 2.15) in the online (zyBook) version of the recommended textbook [OSC].

For this portion of the project (Steps 1 & 2), you should use an Ubuntu Linux Virtual Machine (provided by the authors of our recommended textbook) running on the VirtualBox application.
You can download the Ubuntu virtual machine from the OSC textbook's Companion Site at https://www.os-book.com/OS10/index.html .
Using the VM will allow you to run the Ubuntu operating system on your own personal computer without any system changes, as well as allowing you administrator access without the risk of modifying your host operating system, permitting access to operating system functionality.

The files `simple.c` and `Makefile` referenced in the OSC textbook may be found in the Ubuntu VM, in the folder `~/final-src-osc10e/ch2/` . (You can also download the source code as a Zip file from the OSC textbook's Companion Site using the link above.)

To begin your work on this step, open a command line window, type the following:

```
cd ~/
mkdir Project2
cd Project2
mkdir Option1Step1
cd Option1Step1
```

to create a folder for this step of the project.

You can now copy the files `simple.c` and `Makefile` files to this folder by typing:

```
cp ~/final-src-osc10e/ch2/simple.c .
cp ~/final-src-osc10e/ch2/Makefile .
```

Note that the period after the file names is required in each command.

While completing this step, in addition to creating your C program for Part II, please create screenshots of the `dmesg` output for both Parts I and II and include these screenshot files in your `Step1` folder.

Step 2

Complete Part I of "Project 3 – Linux Kernel Module for Listing Tasks" in the Programming Projects section (section 3.13) near the end of Chapter 3 of the online (zyBook) version of our required textbook [OSC].

To begin your work on this step, open a command line window, type the following:

```
cd ~/Project2
mkdir Option1Step2
cd Option1Step2
```

to create a folder for this step of the project.

While completing this step, in addition to creating your C program for the kernel module, please create screenshots of the following:

- the `dmesg` output when installing your kernel module; and
- the output of running the command `ps -el` in the command line window

and include these screenshot files in your `Option1Step2` folder.

[End of Option 1 description]

Option 2

Implement the readers/writers problem solution described in section 7.1 of the OSC textbook (in the zyBook, and Module 3 Part 2 Slides 7 through 9) using the Pthread library (`<pthread.h>`) and the Pthread semaphore library (`<semaphore.h>`). Use the program outlines in the OSC zyBook Figures 7.1.3 and 7.1.4 (Slides 8 & 9) to guide your solution.

To begin your work on this step, open a command line window, type the following:

```
cd ~/Project2
mkdir Option2
cd Option2
```

to create a folder for this step of the project. Create the file `ReadersWriters.c` in your `Option2` folder.

The following files are attached to the project description (in the Assignments content area on Blackboard) to help guide your solution:

- `Pthread.zip` – a collection of programs using Pthreads
- `ProducerConsumer.c` – another example program using Pthreads
- `PthreadAPI.pdf` – summary description of the Pthreads API

To compile a C language program using Pthreads and semaphores, type:

```
gcc -o ReaderWriters ReaderWriters.c -lpthread
```

where `ReadersWriters.c` is your C program file. Note that "`-lpthread`" contains no spaces.

To run your compiled program, type:

```
./ReaderWriters <command line arguments>
```

The last four pages of this document provide an overall outline for your program. The outline is in four parts: 1) the global shared area where you declared shared resources (semaphores and other shared variables), 2) the reader thread function where each reader thread executes; 3) the writer thread function, where each writer thread executes; and 4) the main function which sets everything up. Areas where you need to add declarations or code are in ***bold, italic, red font***.

We've included some declarations and functions to allow you to slow your threads down by causing them to sleep. The function `threadSleep()` will make your reader and writer threads sleep for a random number of nanoseconds based on a pair of parameters, **range,** and **base**. **Base** is a non-random number of nanoseconds, the minimum sleep time. **Range** is used to calculate a random number that is added to the **base** nanoseconds for the overall sleep time.

Each class of threads (readers and writers) has four sleep parameters that allow you to control how long the thread sleeps inside its critical section, and how long it sleeps outside its critical section. The complete set of **range** and **base** parameters is listed below. You can use these parameters to create reader threads that hurry to read but read a long time, readers that read quickly but not very often, and so on.

The last two parameters, **nr** and **nw**, control how many reader and writer threads will be created.

The program is run by typing:

```
./ReaderWriters ricr ricb roocr roocb wicr wicb woocr woocb nr nw
```

where

- **ricr** is the range parameter for controlling how long readers sleep inside their critical sections
- **ricb** is the base number of nanoseconds a reader will sleep inside their critical sections
- **roocr** is the range parameter for controlling how long readers sleep outside their critical sections
- **roocb** is the base number of nanoseconds a reader will sleep outside their critical sections
- **wicr** is the range parameter for controlling how long writer s sleep inside their critical sections
- **wicb** is the base number of nanoseconds a writer will sleep inside their critical sections
- **woocr** is the range parameter for controlling how long writer s sleep outside their critical sections
- **woocb** is the base number of nanoseconds a writer will sleep outside their critical sections
- **nr** is the number of reader threads to create
- **nw** is the number of writer threads to create

Once you have your program working properly, **play with the parameters to perform the following experiments:**

1. Attempt to starve the writer threads; and
2. Attempt to starve the reader threads.

Remember, many fast readers can starve a few slow writers, and vice versa. Document your experiments by recording the values of the parameters that you used in each attempt, along with the values of the total number of reads and total number of writes that were performed in that attempt. **Include your experiment log in a README file for your project.**

[End of Option 2 description]

Submission Procedure

Change the name of your group's `Project2` folder to "`<YourGroupName>-P2`":

```
cd ~/
mv Project2 <YourGroupName>-P2
```

Now, use the `tar` and `gzip` commands to compress your directory containing these files into a single file called "`<YourGroupName>-P2.tar.gz`" (e.g., "`Group1-P2.tar.gz`"). For example, assuming that the current working directory is the directory that contains `Group1-P2`:

```
tar cvf Group1-P2.tar Group1-P2
gzip Group1-P2.tar
```

Alternatively, you may create a Zip file using your favorite Zip program, so long as the resulting Zip file retains the directory structure and naming scheme mandated above.

The Recorder for your group should submit the group's `.tar.gz` or `.zip` file using the Blackboard page for this assignment. **The Recorder should indicate in the Comments field for the submission all of the group members who actually contributed to the project.**

Option 2 Figures

The following figures apply to Option 2. Areas where you need to add declarations or code are in ***bold, italic, red font***.

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

// These are the globals shared by all threads, including main
#define RANGE 1000000000
#define BASE   500000000
int rICrange = RANGE;
int rICbase = BASE;
int rOOCrange = RANGE;
int rOOCbase = BASE;
int wICrange = RANGE;
int wICbase = BASE;
int wOOCrange = RANGE;
int wOOCbase = BASE;
int keepgoing = 1;
int totalReads = 0;
int totalWrites = 0;

// The global area must include semaphore declarations and
// declarations of any state variables (reader counts,
// total number of readers and writers).

// Use this function to sleep within the threads
void threadSleep(int range, int base) {
     struct timespec t;
     t.tv_sec = 0;
     t.tv_nsec = (rand() % range) + base;
     nanosleep(&t, 0);
}
```

Figure 1: The global shared area of the readers/writers program

```
void *readers(void *args) {
     int id = *((int *) args);

     threadSleep(rOOCrange, rOOCbase);
     while (keepgoing) {

          // Add code for each reader to enter the
          // reading area.
          // The totalReads variable must be
          // incremented just before entering the
          // reader area. However, you must use mutual exclusion
          // on the increment operation to prevent race conditions
          // for the increment.
          totalReads++;

          printf("Reader %d starting to read\n", id);

          threadSleep(rICrange, rICbase);

          printf("Reader %d finishing reading\n", id);

          // Add code for each reader to leave the
          // reading area.

          threadSleep(rOOCrange, rOOCbase);
     }

     printf("Reader %d quitting\n", id);
}
```

```
void *writers(void *args) {
     int id = *((int *) args);

     threadSleep(wOOCrange, wOOCbase);
     while (keepgoing) {
          // Add code for each writer to enter
          // the writing area.
          totalWrites++;

          printf("Writer %d starting to write\n", id);

          threadSleep(wICrange, wICbase);

          printf("Writer %d finishing writing\n", id);

          // Add code for each writer to leave
          // the writing area.

          threadSleep(wOOCrange, wOOCbase);
     }

     printf("Writer %d quitting\n", id);
}
```

Figure 3: The writer thread function

```
int main(int argc, char **argv) {

       int numRThreads = 0;
       int numWThreads = 0;


       if (argc == 11) {
              rICrange = atoi(argv[1]);
              rICbase = atoi(argv[2]);
              rOOCrange = atoi(argv[3]);
              rOOCbase = atoi(argv[4]);
              wICrange = atoi(argv[5]);
              wICbase = atoi(argv[6]);
              wOOCrange = atoi(argv[7]);
              wOOCbase = atoi(argv[8]);
              numRThreads = atoi(argv[9]);
              numWThreads = atoi(argv[10]);
       }
       else {
              printf("Usage: %s <reader in critical section sleep range> <reader in
critical section sleep base> \n\t <reader out of critical section sleep range> <reader out
of critical section sleep base> \n\t <writer in critical section sleep range> <writer in
critical  section sleep base> \n\t  <writer out of critical section sleep range> <writer
out of critical section sleep base> \n\t <number of readers> <number of writers>\n",
argv[0]);
              exit(-1);
       }

       // Add declarations for pthread arrays, one for reader threads and
       // one for writer threads.

       // Add declarations for arrays for reader and writer thread identities. As in the
       // dining philosopher problem, arrays of int are used.

       // Add code to initialize the binary semaphores used by the readers and writers.


       // Add a for loop to create numRThread reader threads.

       // Add a for loop to create numWThread writer threads.

       // These statements wait for the user to type a character and press
       // the Enter key. Then, keepgoing will be set to 0, which will cause
       // the reader and writer threads to quit.
       char buf[256];
       scanf("%s", &buf);
       keepgoing = 0;

       // Add two for loops using pthread_join in order to wait for the reader
       // and writer threads to quit.

       printf("Total number of reads: %d\nTotal number of writes: %d\n",
               totalReads, totalWrites);
}
```

Figure 4: The main function