Project 4: Synchronization

CS 411

For this project, you'll implement *variations* of two of the classic synchronization problems: Dining Philosophers and The Sleeping Barber. You'll use Pthreads mutex and condition variables in your implementation of the Dining Philosophers problem and POSIX unnamed semaphores in your implementation of the Sleeping Barber problem. Your solutions will be thread-based.

You don't need your virtual machines for this project — you can use phoenix. You'll be using a new repository on GitHub to submit this project and to collaborate with your partner. See Canvas for the link to use to create your team and to create your repository. Remember to create a new directory for your working files and the local copy of your repo. As a refresher, here's the basic sequence of git commands:

```
mkdir Project4
cd Project4
git init   # DON'T DO THIS IN YOUR HOME DIRECTORY!!!!!!
git remote add origin YOUR_GITHUB_REPO_URL
# add files
git add .
git commit -m "Initial commit."
git push -u origin master # Only for the very first push to the remote
git push # All pushes after the very first push
git clone YOUR_GITHUB_REPO_URL
git pull
```

References

The following are all available on, or linked to from, the course web site, or in the textbook:

1. Dining Philosophers Solution Using Monitors, Section 5.8.2 of the textbook. Note that this is a monitor-based solution, so it will require some modification — the addition of mutual exclusion for critical sections and the use of while loops with condition variables, rather than if statements.

2. *The Sleeping Barber Problem.* Includes a semaphore-based solution.

3. Pthreads Synchronization, Section 5.9.4 of the textbook for introductory material on Pthreads mutex and condition variables, and POSIX unnamed semaphores.

4. Project 1 and Project 2 in Chapter 5, on pp. 251–253, for useful background information.

5. `semaphore.c` — A concise illustration of POSIX semaphore functions in Linux.

6. `producerConsumer.c` — An illustration of Pthreads operations in Linux, including thread creation, parameter passing, join, and mutex and condition variable operations.

7. `helloThreaded.c` — An illustration of how to manage a fair number of threads using a `pthread_t` array.

Code Structure for an Actor

This illustrates the basic structure of a philosopher, barber, customer, etc. thread that requests or performs some service at various intervals. Because this is an infinite loop, you'll need to terminate your programs by typing `Ctrl-c`.

```
/* Remember to inlude the Pthreads library when compiling:
 *
 *    gcc -o foo foo.c -lpthread
 */

/* Include files */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <assert.h>

/* Function prototypes */
void delay(unsigned int *seed, unsigned int min, unsigned int max);
void *actor1(void * ptr);

/* Sleep a random amount of time between min and max seconds. */
void delay(unsigned int *seed, unsigned int min, unsigned int max)
{
   /* assert() terminates the program if its condition is false. */
   assert(min <= max);
   sleep((unsigned int) (min + (rand_r(seed) % (max - min + 1))));
}

void *actor1(void * ptr)
{
   long tid = (long) ptr;
   /* Each thread needs its own private seed for the pseudo-random
    * number generator.
    */
   unsigned int seed = tid;

   while (1)
   {
      /* Simulate action 1 by sleeping. */
      printf("Action 1\n");
      delay(&seed, 3, 8);

      /* Some synchronization here. */
```

```
      /* Simulate action 2 by sleeping. */
      printf("Action 2\n");
      delay(&seed, 0, 10);

      /* More synchronization here. */

      /* etc. */
   }
}

int main()
{
   pthread_t actor;

   pthread_create(&actor, NULL, actor1, (void *) 1l);
   pthread_join(actor, NULL);
   return 0;
}
```

I/O Hints

In order to un-buffer I/O, put the following at the beginning of `main()`:

```
/* Use unbuffered output for output logging purposes. */
setvbuf(stdout, (char *)NULL, _IONBF, 0);
setvbuf(stderr, (char *)NULL, _IONBF, 0);
```

You may find the following useful for timestamping your output:

```
/* Necessary include files */
#include <time.h>
#include <sys/time.h>
#include <string.h>

/* Function prototype */
void timestamp(char *msg);

/* Function */
void timestamp(char *msg)
{
   char buf[128];
   struct timeval tv;
   struct tm ltv;

   gettimeofday(&tv, NULL);
   localtime_r(&tv.tv_sec, &ltv);
   sprintf(buf, "%02d:%02d:%05.2f   ", ltv.tm_hour, ltv.tm_min,
           (double) (ltv.tm_sec + tv.tv_usec / 1.0E6));
```

```
   strncat(buf, msg, 128 - strlen(buf) - 1);
   printf("%s", buf);
}

/* Usage example */
foo(int tid)
{
   char msg[128];

   sprintf(msg, "Cust %d wants haircut\n", tid);
   timestamp(msg);
}
```

Dining Philosophers

Implement a mutex and condition variable-based solution to Dining Philosophers, based upon the monitor-based pseudo-code given in Section 5.8.2 of the textbook. Using a `turn` variable, ensure that starvation is avoided in your implementation, without unduly causing the other philosophers to go hungry. (I.e., if it's philosopher 3's turn and philosopher 3 is thinking, philosophers 2 and 4 shouldn't be prevented from eating.) Once the philosopher whose turn it is finally eats, it's the next philosopher's turn, where "next" is used in a circular way.

Because of the real possibility of spurious signal operations, ensure that your condition variable wait operations are "guarded" by while statements.

The Sleeping Barber

Implement a POSIX unnamed semaphores-based solution to The Sleeping Barber, using the pseudo-code in the *The Sleeping Barber Problem* reference document as a starting point. The town in which the barber cuts hair has one Very Important Person (VIP). When the VIP visits the barber, he gets preferential service — he doesn't preempt a customer receiving a haircut at the time he arrives, nor does he cause a fifth waiting customer to relinquish his seat, but he will always be the next customer to receive a haircut. The barber thread should contain logic and `printf()` statements to indicate when the barber has no customers and will therefore go to sleep. Finally, use a semaphore, controlled by the barber, to explicitly force a rendezvous between the barber and the customer/VIP receiving a haircut.

Assume that a customer's haircut takes one second, but that the VIP's haircut takes two seconds. The VIP needs a new haircut five seconds after the previous haircut. Ordinary customer i needs a new haircut $i + 2$ seconds after the previous visit to the barbershop.

To demonstrate your logic, run your program with two ordinary customers, demonstrating that the barber sleeps occasionally. Then, run your program with 10 ordinary customers, demonstrating that sometimes customers leave without getting a haircut, that the VIP receives priority treatment, and that the barber doesn't sleep when customers are waiting.

Project 4 Turn-in

By the project deadline, email to me the GitHub https URL of your project repository. Remember to document assistance you received from others. This can be done in the email you send me your

repo URL, in your README.md file, or in comments at the very top of your source code files. This project will be assessed as follows:

- Dining Philosophers: 40%
- The Sleeping Barber: 40%
- Readability: 10%
- Verifiability: 10% (Your source code should contain a sufficient number of informative `printf()` calls so that I can understand what's happening as your program runs.)