CSD204, Operating Systems, Spring 2015
Shiv Nadar University
Instructor: Shashi Prabh

Lab 9 Extra, Ungraded: Sleeping TA Problem
SGG, Chapter 5

Due: Mar. 31, 2015

1 The Sleeping Teaching Assistant

A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TAs office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time.
Using POSIX threads, mutex locks, and semaphores, implement a solution that coordinates the activities of the TA and the students. Details for this assignment are provided below.

The Students and the TA
Using Pthreads, begin by creating `n` students. Each will run as a separate thread. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in a chair in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA is sleeping, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn. If no students are present, the TA may return to napping.
Perhaps the best option for simulating students programming-as well as the TA providing help to a student-is to have the appropriate threads sleep for a random period of time.

POSIX Synchronization Coverage of POSIX mutex locks and semaphores from Section 5.9.4 of your textbook.
The Pthreads API is available for programmers at the user level and is not part of any particular kernel. This API provides mutex locks, condition variables, and readwrite locks for thread synchronization.
Mutex locks represent the fundamental synchronization technique used with Pthreads. A mutex lock is used to protect critical sections of code-that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section. Pthreads uses the `pthread_mutex_t` data type for mutex locks. A mutex is created with the `pthread_mutex_init()` function. The first parameter is a pointer to the mutex. By passing NULL as a second parameter, we initialize the mutex to its default attributes. This is illustrated below:

```
#include <pthread.h>
pthread mutex t mutex;
/* create the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

The mutex is acquired and released with the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. If the mutex lock is unavailable when `pthread_mutex_lock()` is invoked, the calling thread is blocked until the owner invokes `pthread_mutex_unlock()`. The following code illustrates protecting a critical section with mutex locks:

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);
/* critical section */
/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a non-zero error code. Condition variables and read-write locks behave similarly to the way they are described in Sections 5.8 and 5.7.2, respectively. Many systems that implement Pthreads also provide semaphores, although semaphores are not part of the Pthreads standard and instead belong to the POSIX SEM extension. POSIX specifies two types of semaphores - named and unnamed. The fundamental distinction between the two is that a named semaphore has an actual name in the file system and can be shared by multiple unrelated processes. Unnamed semaphores can be used only by threads belonging to the same process. In this section, we describe unnamed semaphores. The code below illustrates the `sem_init()` function for creating and initializing an unnamed semaphore:

```
#include <semaphore.h>
sem_t sem;
/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

The `sem_init()` function is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphores initial value

In this example, by passing the flag 0, we are indicating that this semaphore can be shared only by threads belonging to the process that created the semaphore.
A non-zero value would allow other processes to access the semaphore as well. In addition, we initialize the semaphore to the value 1.
In Section 5.6, we described the classical `wait()` and `signal()` semaphore operations. Pthreads names these operations `sem_wait()` and `sem_post()`, respectively. The following code sample illustrates protecting a critical section using the semaphore created above:

```
/* acquire the semaphore */
sem_wait(&sem);
/* critical section */
/* release the semaphore */
sem_post(&sem);
```

Just like mutex locks, all semaphore functions return 0 when successful, and non-zero when an error condition occurs. There are other extensions to the Pthreads API including spinlocks but it is important to note that not all extensions are considered portable from one implementation to another.

2 Instructions

- Implement the solution to Sleeping Teaching Assistant Problem as outlined above using POSIX API in C. Name your solution source code as sleepingta.c
- Write Makefile to compile your code. Your makefile should have target named clean to delete object files and binaries. Use makefile macros to name your compiler and path to your source files.