Programming Project 2 Com S 352 Spring 2013
Released: Monday, April 1, 2013
Due: 11:59pm, Friday, April 26, 2013

Objective

In this project, a C (or C++) library is to be developed to implement the semaphore data type based on the unnamed pipe mechanism, and test code is developed to solve a special version of reader-writer problem by using the library.

Note: process/thread synchronization mechanisms learned in Chapter 6, i.e., hardware instruction (e.g., test-and-set, swap) based solutions, mutex, semaphore, etc., are not allowed to be used in this project. Unnamed pipes should be used instead.

Preliminary: example implementing mutually exclusive lock using unnamed pipe

The following C code shows one method to implement mutually exclusive lock by using unnamed pipe. Understand the implementation is helpful to the implementation of semaphore by using unnamed pipe.

```
#include <unistd.h>
#include <sys/ipc.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <pthread.h>

void lock_release (int fd[2]) /*called when releasing a lock */
{
	int pid;
	pid=fork();
	if(pid<0){
		perror("fail to implement unlock\n");
		exit(1);
	}
	if(pid==0){
		write(fd[1],"ok",10);
		exit(0);
	}
}

void lock_acquire(int fd[2]) /*called when acquiring a lock*/
{
	char buf[10];
	read(fd[0],buf,10);
}

void lock_init(int fd[2]) /*called when initializing a lock*/
{
	if(pipe(fd)!=0){
		perror("fail to initialize pipe\n");
```

```
            exit(1);
      }
      lock_release(fd);
}

/*following is the test code*/
int count=0;
int lock[2];

void *th_code (void *x) /*code executed by each thread*/
{
      while(1){
            lock_acquire(lock);
            printf("thread %ld: enter CS: count=%d\n", pthread_self(), count);
            count++;
            sleep(2);
            count--;
            printf("thread %ld: exit CS: count=%d\n", pthread_self(), count);
            lock_release(lock);
            sleep(1);
      }
}

void main()
{
      pthread_t th[4];
      int i;
      lock_init (lock);
      for(i=0;i<4;i++) pthread_create(&th[i],NULL,th_code,NULL);
      for(i=0;i<4;i++) pthread_join(th[i],NULL);
}
```

The new semaphore library: Interface

The semaphore library that you will develop should define a data structure called pipe_sem_t in a header file pipe_sem.h
.
The library should provide the following APIs:

• void pipe_sem_init(pipe_sem_t *sem, int value): to initialize a semaphore and set its initial value.

• void pipe_sem_wait(pipe_sem_t *sem): to perform a wait operation on the semaphore.

• void pipe_sem_signal(pipe_sem_t *sem): to perform a signal operation on the semaphore.

The code that implements the above APIs should be put into a C (or C++) source file pipe_sem.c (or pipe_sem.cpp). Your implementation of the library could use the afore-described pipe-based lock implementation and/or the unnamed pipe mechanism directly.

Test code: Required

A program, named rw_test.c (or rw_test.cpp), is required to be developed to implement the following “writer priority version” of the reader-writer problem:

• Reader and writer processes share an integer variable
• Readers read the variable without changing its content
• Writers may change the variable
• Concurrent reads by several readers are allowed, but only one writer can write at a time
• Readers and writers exclude each other
• Arriving writers have priority over waiting readers; a waiting or arriving reader can access the resource only when there is no writer waiting in the system

In the program, each reader or writer process is implemented as a thread (can be implemented as a Pthread thread). Once started, the thread first requests to access the resource. When a thread is allowed to access the resource (i.e., enter its CS), it first outputs a message:

Reader thread <id> enters CS

or

Writer thread <id> enters CS

Then, it calls

sleep(1);

to simulate the time elapse for its access. Finally, it outputs:

Reader thread <id> is exiting CS

or

Writer thread <id> is exiting CS. Then the thread ends.

The user interface of the rw_test.c (or rw_test.cpp) program is as follows. The command for launching the program should be:

> rw_test <number-of-arriving threads> <a sequence of 0 and 1 separated by a blank-space> <thread arrival interval>

For example, to simulate the scenario that 5 threads arriving (i.e., being created) according to the following timings: reader-1 at time 0 second, reader-2 at time 1 second, writer-1 at time 2 second, writer-2 at time 3 second and reader-3 at time 4 second, the command will be:

> rw_test 5 0 0 1 1 0 1

Here, the first argument “5” is the total number of reader/writer threads, and the following 5-element sequence “0 0 1 1 0” specifies the arriving order of readers (denoted by “0”) and writers (denoted by “1”). The last argument “1” specifies the time elapse between two consecutive arrivals in the unit of second.

Other Instructions

• Your code should compile successfully in pyrite. Otherwise, you may receive no points.

• Write comments in your code to help debugging, testing and grading.

• Well test your code before submission.

• Your all source code (including header files and C/C++ source files) and a make-file (or a script file) should be tarred or zipped into a single compressed file. After being uncompressed, the make-file or a script file can be run by the TA to compile your code.

• Your code should be submitted to Blackboard system by deadline. If submitted late, the same late policy as weekly assignments will be applied.

• Name your submitted tarred/zipped file as OSProj2_InitialOfYourFirstName_YourLastName.tar/zip

• Start as early as possible!