Computer Operating System Experiment

Laboratory 5

Synchronization

Objective:

The Pthread library offers the *pthread_mutex_t* data type, which is much like a binary semaphore and therefore somewhat of limited utility in the solution of synchronization problems. Fortunately, POSIX gives you the more general-purpose semaphore in the *sem_t* data data type. In this lab, you will learn to work with these two mechanisms for thread synchronization as you implement a solution to the bounded-buffer problem. Including:

- Learn to work with Linux and Pthread synchronization mechanisms.
- Practice the critical-section problem
- examine several classical synchronization problems

Equipment:

VirtualBox with Ubuntu Linux

Methodology:

Program and answer all the questions in this lab sheet.

1 Synchronization

Synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

2 Work with POSIX semaphore

Semaphores are very useful in process synchronization and multithreading. POSIX semaphore library in Linux systems support powerful functions. POSIX semaphore calls are much simpler than the System V semaphore calls. However, System V semaphores are more widely available, particularly on older Unix-like systems. POSIX semaphores have been available on Linux systems post version 2.6 that use glibc. Let's learn how to use it.

The basic code of a semaphore is simple as presented here. But this code cannot be written directly, as the functions require to be atomic and writing code directly would lead to a context switch without function completion and would result in a mess.

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to:

- Include semaphore.h
- Compile the code by linking with -lpthread -lrt

POSIX Named Semaphore APIs:

- To declare a semaphores

  sem_t sem;

- To lock a semaphore or wait we can use the sem_wait function:

  int sem_wait(sem_t *sem);

- To release or signal a semaphore, we use the **sem_post** function:

  int sem_post(sem_t *sem);

- A semaphore is initialised by using **sem_init**(for processes or threads) or **sem_open** (for IPC).

  sem_init(sem_t *sem, int pshared, unsigned int value);

- get the value of a semaphore. places the current value of the semaphore pointed to sem into the integer pointed to by sval.

  int sem_getvalue(sem_t *sem, int *sval);

- To destroy a semaphore

  sem_destoy(sem_t *mutex);

3 Experiments

3.1 Experiment 1: Thread Synchronization Problems

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a

critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

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

pthread_t tid[2];
int counter;
void* doSomeThing(void *arg)
{
	unsigned long i = 0;
	counter += 1;
		printf("\n Job %d started\n", counter);
	for(i=0; i<1000;i++);
		printf("\n Job %d finished\n", counter);
	return NULL;
}
int main(void)
{
	int i = 0;
	int err;
	while(count < 2)
	{
		err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
		if (err != 0)
			printf("\ncan't create thread :[%s]", strerror(err));
		i++;
	}

	pthread_join(tid[0], NULL);
	pthread_join(tid[1], NULL);
	return 0;
}
```

Figure 1 threads.c code

Requirements

1. Do you think that the counter increase correctly? If not, what is wrong? And please increase the

counter correctly using multithreads
2. if Thread 1 must run before Thread 2, how do you do?

3.2 Experiment 2: The Bounded-Buffer Problem

Assume that you have a bounded buffer with *n*, each capable of holding a single value of data type int. If this buffer is to be shared by multiple threads, you need to be careful with how you implement functionality to remove and to return a buffer to the data structure. By virtue of their scheduling, threads might interrupt each other in the middle of one of these operations leaving the data structure in an inconsistent state.

Your textbook, Operating Systems Concepts, discusses the bounded-buffer problem in the context of the classical synchronization problem of producers and consumers (Section 7.1). The solution to the problem is presented in structures that delineate the code for the two types of process as shows below.

Producer process

```
do {
   ...
   // produce an item i
   ...
   wait(empty);
   wait(mutex);
         // add item i to buffer
   signal(mutex);
   signal(full);
} while (TRUE);
```

Figure 2 producer pseudo code

Consumer process

```
do {
   ...
   // consume an item i
   ...
   wait(full);
   wait(mutex);
   // remove item i from buffer
   signal(mutex);
   signal(empty);
} while (TRUE);
```

Figure 3 consumer pseudo code

Looking at the structure of code given above, you will realize that having the producers and the consumer processes deal directly with synchronization is not ideal. Primarily, there is an issue of applying the concept of abstraction: the code would be substantially easier to manage if you had an ADT for the bounded buffer. You can define the following application programming interface (API):

```
#include "buffer.h"
/* the buffer */
Buffer_item buffer[BUFFER SIZE];

/**
 * add item into the buffer
 * @param item pointer to add an item into buffer
 */

int insert_item(buffer_item item) {
      /* insert item into buffer return 0 if successful, otherwise
      return -1 indicating an error condition */
}

/**
 * Remove item from the buffer
 * @param item pointer to remove an item form buffer
 */

int remove_item(buffer_item *item) {
      /* remove an object from buffer placing it in item return 0 if
      successful, otherwise return -1 indicating an error condition */
}
```

Figure 4 buffuer.h

The buffer will be manipulated with two functions, insert item() and remove item() , which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in Figure 5.

The main() function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the main() function will sleep for a period of time and, upon awakening, will terminate the application. The main() function will be passed three parameters on the command line:

- The number of producer threads
- The number of consumer threads

- Time to sleep before terminating

```
#include "buffer.h"
int main(int argc, char *argv[]) {
1. Get command line arguments argv[1],argv[2],argv[3] */
2. Initialize buffer */
3. Create producer thread(s) */
4. Create consumer thread(s) */
5. Sleep */
6. Exit */
}
```

Figure 5 code structure

The Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the rand() function, which produces random integers between 0 and RAND MAX . The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer.

As noted earlier, you can solve this problem using either Pthreads. In the following sections, we supply more information on each of these choices.

Pthreads Thread Creation and Synchronization

Creating threads using the Pthreads API. Coverage of mutex locks and semaphores using Pthreads is provided. Refer to those sections for specific instructions on Pthreads thread creation and synchronization.

Requirements

1) The running command should be the following.

Prod_com [num_prod] [num_cons] [sleep_time]

where:

- [num_prod] is the number of producer threads
- [num_cons] is the number of consumer threads
- [sleep_time] is the number of seconds to sleep before the process terminates

2) Modify your makefile to compile it.
3) Compile and run the resulting program.

3.3 Experiment 3: Dining Philosophers (optional if you are still available)

The standard statement of Dining Philosophers is a resource allocation problem. As in the version for pre-lab problem, it consists of five philosophers sitting around a table and cycling through the states thinking, getting hungry, and eating.

Philosophers need chopsticks in order to eat. Between each pair of philosophers, there lies one single chopstick. When a philosopher wants to eat, he must acquire two chopsticks; that is, s/he must try to get exclusive access to the chopstick on the left and also to the one on the right. Effectively, this means that no two neighboring philosophers can ever eat at the same time. A complicating factor is that when a philosopher wants to eat, he must request each chopstick separately, that is, one at a time.

For this problem, you will create a program called philosophers.c, which implements your solution to the dining philosopher's problem. In this program, you will use the mutex and semaphore constructs, which have used before. Each chopstick must be modeled by a mutex and each philosopher by a thread.

The Philosopher threads request chopsticks by doing two successive locks: one for the chopstick on the left and one for chopstick on the right. Once a Philosopher passes through the wait on a mutex, it is assumed to have picked up the corresponding chopstick. Since a Philosopher must pick up both the left and the right chopsticks in order to be able to eat, you can think of the "eating" section as the critical section of the Philosopher 's code.

Your implementation must follow the guidelines below, which are consistent with the solution below:

```
semaphore chopstick[5];
do {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
        . . .
        /* eat for a while */
        . . .
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
        . . .
        /* think for a while */
        . . .
} while (true);
```

- You must have an array chopsticks[] of five mutexes, which is visible to all Philosopher threads

and correctly initialized.

- Modify your Philosopher thread so that it works with the chopsticks[] array of mutexes.
- Each Philosopher thread will make calls to pthread_mutex_lock and pthread_mutex_unlock to simulate picking up and putting down chopsticks. You should use the following scheme for numbering the chopsticks: the chopstick to the left of Philosopher i is numbered i, while the chopstick to the right is numbered (i+1)%5 (remember that Philosopher 0 is to the "right" of Philosopher 4).

Questions:

1) Do you think that this solution is good? If not, what is wrong?
2) Can you modify this solution to meet the requirements? And use semaphores to implement it.