Hacettepe University
Department of Computer Science and Engineering

BIL342 Programming Laboratory
Experiment III

Subject:	Concurrent Programming
Advisors:	Prof. Dr. Ali SAATÇİ R.A. Kerem ERZURUMLU
Submission Date:	15/03/2007
Deadline:	02/04/2007
Programming Language:	ANSI C

AIM

The aim of this experiment is to give students the ability to cope with some of the UNIX interprocess communication and synchronisation mechanisms. Within this context, during this experiment you will learn how to use UNIX shared memory segments and UNIX semaphores.

PROBLEM DESCRIPTION

The experiment requires you to use a shared memory segment to hold a simple matrix. The matrix is a two dimensional array of numbers, maintained such, that the last number of each row is the sum of the other numbers in that row; and the last number of each column is the sum of the other numbers in that column so the total dimension of the matrix is (n+1) x (n+1). Take n=10 for the experiment (Figure-1).

The matrix will be accessed by two processes: an **`updater`** and a **`checker`** that you have to design and implement.

The **`updater`** will periodically change a randomly selected cell of the matrix to a random value and update the row and column totals accordingly. The **`checker`** will periodically print the matrix out and check that the row and column totals are correct. If not it will print an error message.

Both processes have to access concurrently the matrix held on a shared memory segment. Therefore they both have critical sections. Without mutually excluded execution of critical sections, the **`checker`** will, in fact find, from time to time wrongly totaled rows and columns.

In order to implement mutual exclusion among critical sections, you need to consider what

granularity of locking to provide. The simplest approach is to use a single semaphore, so that each time you enter a critical section you lock the entire matrix. However, for this experiment, we ask you to lock only the row and column corresponding to a given cell. For instance, if the **`updater`** is changing the cell at the intersection of row 2 and column 3, it needs to lock that row and column only; there is no reason why the **`checker`** cannot check some other row or column at that time.

In order to lock the individual rows and columns of the matrix you are asked to use two arrays of semaphores; one for the rows, one for the columns. Figure-1 shows the schema along with a possible locking scenario.

Figure-1. Locking matrix rows and columns

Using the information given, design and implement with C the two programs: the **`updater`** and the **`checker`**. The **`updater`** has to be created as a child of the **`checker`**.

First implement them with full synchronisation (correct implementation) and be sure that the **`checker`** never produces error messages. Next, comment out lines where synchronisation is coded (erroneous implementation) and observe the frequency of error messages produced by the **`checker`**. Comment on the results obtained by referring to your knowledge of process management.

Shared Memory

Shared Memory concept

In a multi-user system, at any time, there is a number of active processes sharing the physical memory of the machine. The memory management system ensures that no process is able to

access memory that belongs to another process. Processes have to stay within their own address space. This is an essential feature of any multi-user operating system. The system must prevent a malicious (or merely errant) process from interfering with the operation of other processes, or, worse still, from overwriting parts of the operating system's address space.

There are times, however, when shared memory can be a very effective form of communication among a group of cooperating processes. Consequently, most versions of UNIX provide a mechanism whereby a group of processes can explicitly elect to share a piece of memory. This is done by the shared memory segment mechanism of UNIX.

Shared Memory Operations

A process can create a shared memory segment by specifying a size and allocating to it a unique numeric KEY. This is done by **`shmget()`** system call.

```
int id = shmget(key_t KEY, sizeof(struct shared segment),IPC_CREAT | 0666);
```

`shmget()` returns a handle **`(int id)`**.

Some other independent processes wishing to use that shared memory segment should know its KEY and use **`shmget()`** in the following form, to get a handle for an existing segment:

```
int id = shmget(key_t KEY, sizeof(struct shared segment), NULL);
```

The created shared memory segment should be attached to the address space of a process before being accessed. This is done by:

```
ptr = (struct shared memory *)shmat(int id, NULL, NULL);
```

`shmat()` return a pointer to the shared memory segment structure.

Please learn more about **`shmget()`** and **`shmat()`** by using the **`man pages.`**

You should not think of a shared memory segment as a piece of memory belonging to the address space of a "master" process. Much like a file, a shared memory segment has an existence quite independent of any one process. The segment remains present in the system even if it is not attached to any process. Shared memory segments have other attributes in common with files such as access permissions and modification timestamps. Much like files, child processes inherit handles of shared memory segments created by their parent; so they do not need to call **`shmget()`** to get a handle.

UNIX Semaphores

Shared memory mechanism in UNIX has no inherent tools to control the concurrent access to the shared segment. Therefore, other means of synchronisation should be used while accessing shared segments concurrently.

UNIX semaphores may be used for that purpose.

UNIX Semaphore Operations

UNIX semaphores are rather complex. With a single call you can create a whole array of

semaphores instead of only one. For this, UNIX provides a system call named **semget()**.

A process can create an array of semaphores by specifying the dimension of the array (number of semaphores-**nsems**) and allocating to it a unique numeric KEY.

```
int id = semget(key_t KEY, int nsems, IPC_CREAT | 0666);
```

semget() returns a handle **(int id)**.

Some other independent processes, wishing to use that array of semaphores for synchronisation purposes, should know its KEY and use **semget()** in the following form, to get a handle for an existing array:

```
int id = semget(key_t KEY, int nsems, NULL);
```

Child processes (in contrast to independent processes) inherit handles of semaphore arrays created by their parent; so they do not need to call **semget()** to obtain a handle.

The UNIX semaphores are multivalued (counting), not binary. The semaphore's value indicates the number of units (i.e. records) of the associated resource (i.e. buffer) that are currently available. Used as binary semaphore, a value of 0 indicates that the corresponding resource is locked, free otherwise (value#0)

You can specify a whole array of operations on a whole array of semaphores. This is done by **semop()** system call.

Here are two example functions to lock and unlock a specified binary semaphore in a specified set of semaphores:

```
/* lock semaphore n from set specified by id */
void lock(int id, int n)
    {
    struct sembuf sop; /* build a 1-element array */
    sop.sem_num = n;   /* of semaphore operations */
    sop.sem_op = -1;   /* -1 locks one "unit"     */
    sop.sem_flg = 0;
    semop(id, &sop, 1);
```

```
    }
/* unlock semaphore n from set specified by sem */
void unlock(int id, int n)
    {
    struct sembuf sop; /* build a 1-element array */
    sop.sem_num = n;   /* of semaphore operations */
    sop.sem_op = 1;    /* +1 unlocks one “unit”   */
    sop.sem_flg = 0;
    semop(id, &sop, 1);
    }
```

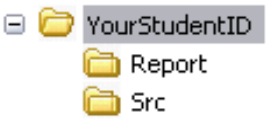
Header files to include in your program:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
```

NOTES

1. You are asked to follow announcements made to Courses.Bil342 newsgroup which is located at nntp://news.cs.hacettepe.edu.tr.
2. A copy of this paper can be found at

   ftp://ftp.cs.hacettepe.edu.tr/pub/dersler/Bil3XX/Bil342/06-07/3
3. Your report and program must be submitted at the same time.
4. You are asked to give both soft and hard copy of your reports. Valid Soft-Copy formats are HTML and PDF.
5. You should submit your work and report on a floppy with the following structure:



6. E-mail submissions are not accepted.
7. Late submissions will not be accepted.
8. At the submission your experiments will be checked for common errors. Incomplete work may not be graded. Do not leave everything to the last minute.
9. Please send additional questions to Courses.Bil342 newsgroup.

Good Luck