**Inter Process Synchronization using Semaphore**

Pratik Mathur (G00963577)

Sarthak Sheth (G00979607)

1. **Inter Process Communication**

Inter process communication is a mechanism which allows cooperating processes to exchange information and to synchronize their action. Inter process communication is useful for creating cooperating processes. Multiple processes can have handles to the same event, mutex, semaphore, or timer object, so these objects can be used to accomplish inter process synchronization.

All processes prevent casual exchange of data. However, occasionally two processes might need to communicate with each other. One method that enables processes to communicate is called inter process synchronization. Because multiple processes can have handles to the same event or mutex object, these objects can be used to accomplish inter process synchronization.

There are different mechanisms in inter process communication. Those are 1. Pipes, 2. Signals, 3. Message queues, 4. Shared memory and 5. Semaphores. We are considering concept of semaphores in our project.

1. **Inter Process Synchronization**

Synchronization refers to the idea that multiple cooperating processes synchronize their resources.

When multiple processes of control share the same memory, we need to make sure that each process sees a consistent view of its data. If each process uses variables that other processes don't read or modify, no consistency problems exist. Similarly, if a variable is read-only, there is no consistency problem with more than one process reading its value at the same time. However, when one process can modify a variable that other processes can read or modify, we need to synchronize the processes to ensure that they don't use an invalid value when accessing the variable's memory contents.

When one process modifies a variable, other processes can potentially see inconsistencies when reading the value of the variable. On processor architectures in which the modification takes more than one memory cycle, this can happen when the memory read is interleaved between the memory write cycles. Of course, this behavior is architecture dependent, but portable programs can't make any assumptions about what type of processor architecture is being used.

1. **Semaphore:**

A semaphore is a counter used to provide access to a shared data object for multiple processes. The Single UNIX Specification includes an alternate set of semaphore interfaces in the semaphore option of its real-time extensions. We do not discuss these interfaces in this text.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened. To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel. A common form of semaphore is called a binary semaphore. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.

Each semaphore is represented by an anonymous structure containing at least the following members:

struct

{

unsigned short semval; /\* semaphore value, always >= 0 \*/

pid\_t sempid; /\* pid for last operation \*/

unsigned short semncnt; /\* # processes awaiting semval>curval \*/

unsigned short semzcnt; /\* # processes awaiting semval==0 \*/

.

.

.

};

Methods of semaphore:

1. Semget()

The first function to call is semget to obtain a semaphore ID.

#include <sys/sem.h>

int semget(key\_t key, int nsems, int flag);

1. **Semctl()**

The semctl function is the catchall for various semaphore operations.

#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd,

... /\* union semun arg \*/);

1. Semop()

The function semop atomically performs an array of operations on a semaphore set.

#include <sys/sem.h>

int semop(int semid, struct sembuf semoparray[], size\_t nops);

1. Exit()

It is a problem if a process terminates while it has resources allocated through a semaphore. Whenever we specify the SEM\_UNDO flag for a semaphore operation and we allocate resources (a sem\_op value less than 0), the kernel remembers how many resources we allocated from that particular semaphore (the absolute value of sem\_op). When the process terminates, either voluntarily or involuntarily, the kernel checks whether the process has any outstanding semaphore adjustments and, if so, applies the adjustment to the corresponding semaphore. If we set the value of a semaphore using semctl, with either the SETVAL or SETALL commands, the adjustment value for that semaphore in all processes is set to 0.

1. **Mutexes**

We can protect our data and ensure access by only one thread/process at a time by using the pthreads mutual-exclusion interfaces. A mutex is basically a lock that we set (lock) before accessing a shared resource and release (unlock) when we're done. While it is set, any other thread/ptocess that tries to set it will block until we release it. If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock. The others will see that the mutex is still locked and go back to waiting for it to become available again. In this way, only one thread will proceed at a time. If we allow one thread to access a shared resource without first acquiring a lock, then inconsistencies can occur even though the rest of our threads do acquire the lock before attempting to access the shared resource.

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,

const pthread\_mutexattr\_t \*restrict attr);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

To initialize a mutex with the default attributes, we set attr to NULL. To lock a mutex, we call pthread\_mutex\_lock. If the mutex is already locked, the calling thread will block until the mutex is unlocked. To unlock a mutex, we call pthread\_mutex\_unlock.

#include <pthread.h>

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

If a thread can't afford to block, it can use pthread\_mutex\_trylock to lock the mutex conditionally. If the mutex is unlocked at the time pthread\_mutex\_trylock is called, then pthread\_mutex\_trylock will lock the mutex without blocking and return 0. Otherwise, pthread\_mutex\_trylock will fail, returning EBUSY without locking the mutex.

1. **Project Code**

#include <semaphore.h>

#include <stdio.h>

#include <errno.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <sys/mman.h>

int main()

{

int i, j, fd;

int \*ptr;

sem\_t mutex;

/\* open a file to use in a memory mapping \*/

fd = open("/dev/zero", O\_RDWR|O\_CREAT, S\_IRWXO);

/\* create a shared memory map with the open file for the data structure that will be shared between processes \*/

ptr=mmap(NULL, sizeof(int), PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0);

close(fd);

if(sem\_init(&mutex,1,1) <0)

{

perror("semaphore initialization");

exit(0);

}

/\* fork a new process \*/

if (fork() == 0)

{

/\* the child will run this section of code \*/

For (j=0; j<2; j++)

{

/\* have the child "wait" for the semaphore \*/

printf("Child PID(%d): waiting...\n", getpid());

sem\_wait(&mutex);

/\* the child decremented the semaphore \*/

printf("Child PID(%d): decrement semaphore.\n", getpid());

}

/\* exit the child process \*/

printf("Child PID(%d): exiting...\n", getpid());

exit(0);

}

/\* The parent will run this section of code \*/

/\* give the child a chance to start running \*/

sleep(1);

for (i=0;i<2;i++)

{

/\* increment (post) the semaphore \*/

printf("Parent PID(%d): posting semaphore.\n", getpid());

sem\_post(&mutex);

/\* wait a second \*/

sleep(1);

}

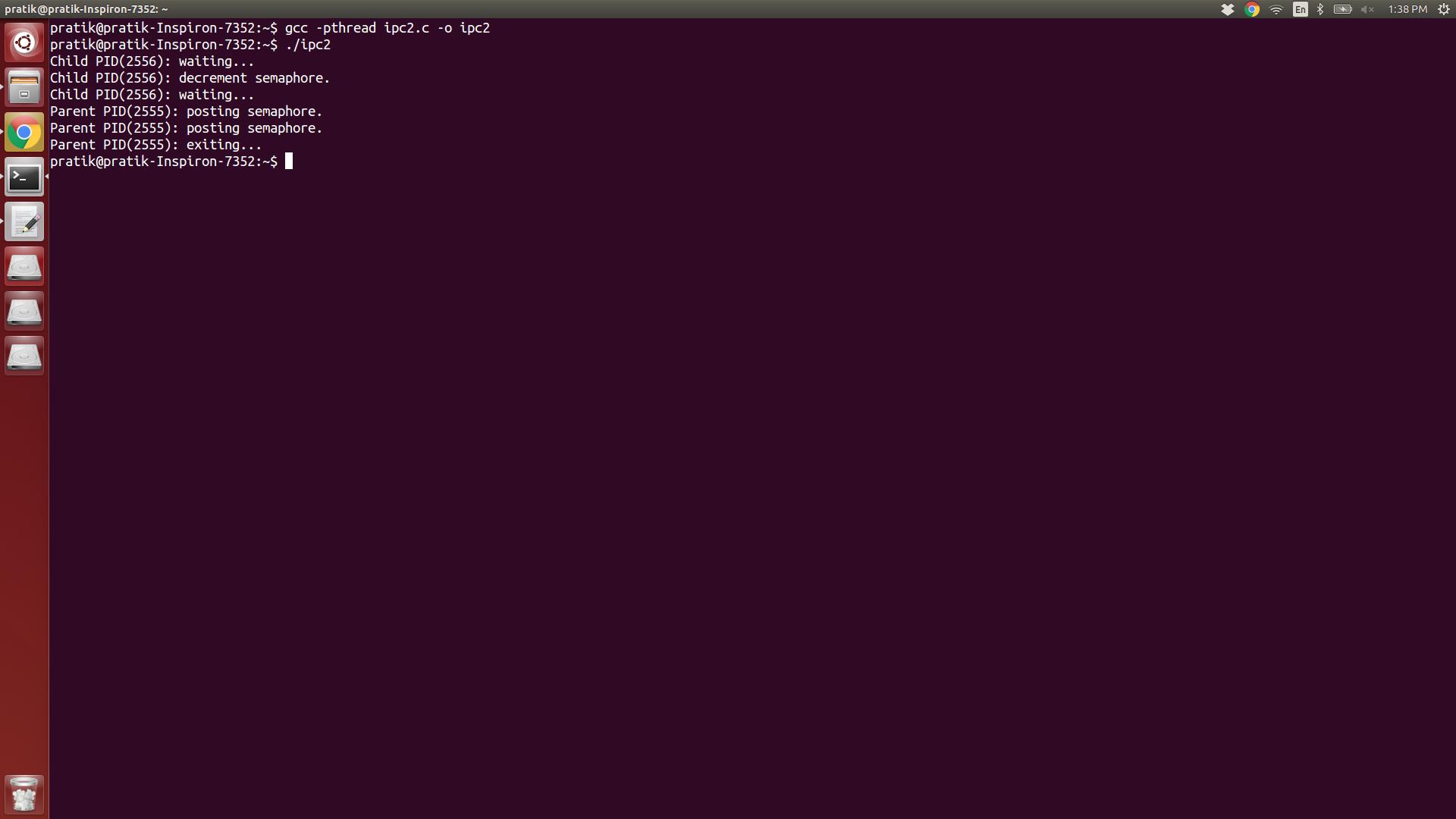
/\* exit the parent process \*/

printf("Parent PID(%d): exiting...\n", getpid());

return(0);

}

1. **Output:**

****

1. **Bibliography**
2. W. Richard Stevens and Stephen A. Rago, “Advanced Programming in the UNIX environment”, Second edition.
3. Threading Programming examples - https://www.cs.cf.ac.uk/Dave/C/node32.html
4. Multithreading in C - <http://softpixel.com/~cwright/programming/threads/threads.c.php>