Name : Shreyas Vijaykumar Sakri

PRN : 2019BTEIT00075

Write a program to illustrate the semaphore concept. Use fork so that 2 process running simultaneously and communicate via semaphore.

Objectives:

1. To learn about IPC through semaphore. 2. Use of system call and IPC mechanism to write effective application programs.

Theory: A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Working of semaphore : In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit. • If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented. • Otherwise, the thread will be blocked until a permit can be acquired. • When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented. • If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

The function semget() initializes or gains access to a semaphore. It is prototyped by:

int semget(key\_t key, int nsems, int semflg); When the call succeeds, it returns the semaphore ID (semid).

The key argument is a access value associated with the semaphore ID.

The nsems argument specifies the number of elements in a semaphore array. The call fails when nsems is greater than the number of elements in an existing array; when the correct count is not known, supplying 0 for this argument ensures that it will succeed.

POSIX Semaphores: <semaphore.h>

sem\_open() -- Connects to, and optionally creates, a named semaphore

sem\_init() -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

sem\_close() -- Ends the connection to an open semaphore.

sem\_unlink() -- Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.

sem\_destroy() -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

sem\_getvalue() -- Copies the value of the semaphore into the specified integer.

sem\_wait(), sem\_trywait() -- Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process.

sem\_post() -- Increments the count of the semaphore.

Data Dictionary:

Number Variable/function Data Type Use

1 KEY Long int External identifier for the program

2 id int Number by which semaphore is known within a program

3 argument Union semun

To pass arguments to the semctl function

4 retval int Store return value of semop function

Program 1: Initialization of Semaphore

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <stdio.h>

/\* The semaphore key is an arbitrary long integer which serves as an external identifier by which the semaphore is known to any program that wishes to use it. \*/

#define KEY (1492)

void main()

{

int id; /\* Number by which the semaphore is known within a program \*/

158

/\* The next thing is an argument to the semctl() function. Semctl() does various things to the semaphore depending on which arguments are passed. We will use it to make sure that the value of the semaphore is initially 0. \*/

union semun {

int val;

struct semid\_ds \*buf;

ushort \* array;

} argument;

argument.val = 0;

/\* Create the semaphore with external key KEY if it doesn't already exists. Give permissions to the world. \*/

id = semget(KEY, 1, 0666 | IPC\_CREAT);

/\* Always check system returns. \*/

if(id < 0)

{

fprintf(stderr, "Unable to obtain semaphore.\n");

exit(0);

}

/\* What we actually get is an array of semaphores. The second argument to semget() was the array dimension - in our case 1. \*/

/\* Set the value of the number 0 semaphore in semaphore array # id to the value 0. \*/

if( semctl(id, 0, SETVAL, argument) < 0)

{

fprintf( stderr, "Cannot set semaphore value.\n");

}

else

{

fprintf(stderr, "Semaphore %d initialized.\n", KEY);

}

}

Program 2: Semaphore A

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#define KEY (1492)

/\* This is the external name by which the semaphore is known to any program that wishes to access it. \*/

void main()

{

int id; /\* Internal identifier of the semaphore. \*/

struct sembuf operations[1];

/\* An "array" of one operation to perform on the semaphore. \*/

int retval; /\* Return value from semop() \*/

/\* Get the index for the semaphore with external name KEY. \*/

id = semget(KEY, 1, 0666);

if(id < 0)

/\* Semaphore does not exist. \*/

{

fprintf(stderr, "Program sema cannot find semaphore, exiting.\n");

exit(0);

}

/\* Do a semaphore V-operation. \*/

printf("Program sema about to do a V-operation. \n");

/\* Set up the sembuf structure. \*/

/\* Which semaphore in the semaphore array : \*/

operations[0].sem\_num = 0;

/\* Which operation? Add 1 to semaphore value : \*/

160

operations[0].sem\_op = 1;

/\* Set the flag so we will wait : \*/

operations[0].sem\_flg = 0;

/\* So do the operation! \*/

retval = semop(id, operations, 1);

if(retval == 0)

{

printf("Successful V-operation by program sema.\n");

}

else

{

printf("sema: V-operation did not succeed.\n");

perror("REASON");

}

} Program 3: Semaphore B

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#define KEY (1492)

/\* This is the external name by which the semaphore is known to any program that wishes to access it. \*/

void main()

{

int id; /\* Internal identifier of the semaphore. \*/

struct sembuf operations[1];

/\* An "array" of one operation to perform on the semaphore. \*/

161

int retval; /\* Return value from semop() \*/

/\* Get the index for the semaphore with external name KEY. \*/

id = semget(KEY, 1, 0666);

if(id < 0)

/\* Semaphore does not exist. \*/

{

fprintf(stderr, "Program semb cannot find semaphore, exiting.\n");

exit(0);

}

/\* Do a semaphore P-operation. \*/

printf("Program semb about to do a P-operation. \n");

printf("Process id is %d\n", getpid());

/\* Set up the sembuf structure. \*/

/\* Which semaphore in the semaphore array : \*/

operations[0].sem\_num = 0;

/\* Which operation? Subtract 1 from semaphore value : \*/

operations[0].sem\_op = -1;

/\* Set the flag so we will wait : \*/

operations[0].sem\_flg = 0;

/\* So do the operation! \*/

retval = semop(id, operations, 1);

if(retval == 0)

{

printf("Successful P-operation by program semb.\n");

printf("Process id is %d\n", getpid());

}

else

{

162

printf("semb: P-operation did not succeed.\n");

}

}

Output:

Conclusion:

Use of semaphore for two different processes where one has to wait for other to release is studied using IPC Semaphore

References:

Dave’s Programming in C Tutorials