IPC:Semaphores

  Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed. Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter. In the computer version, a semaphore appears to be a simple integer. A process (or a thread) waits for permission to proceed by waiting for the integer to become 0. The signal if it proceeds signals that this by performing incrementing the integer by 1. When it is finished, the process changes the semaphore's value by subtracting one from it.

Semaphores let processes query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments.

Semaphores can be operated on as individual units or as elements in a set. Because System V IPC semaphores can be in a large array, they are extremely heavy weight. Much lighter weight semaphores are available in the threads library (see man semaphore and also Chapter 30.3) and POSIX semaphores (see below briefly). Threads library semaphores must be used with mapped memory . A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements.

In a similar fashion to message queues, the semaphore set must be initialized using semget(); the semaphore creator can change its ownership or permissions using semctl(); and semaphore operations are performed via the semop() function. These are now discussed below:

Initializing a Semaphore Set

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.

The semflg argument specifies the initial access permissions and creation control flags.

The following code illustrates the semget() function.

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

...

key\_t key; /\* key to pass to semget() \*/

int semflg; /\* semflg to pass tosemget() \*/

int nsems; /\* nsems to pass to semget() \*/

int semid; /\* return value from semget() \*/

...

key = ...

nsems = ...

semflg = ... ...

if ((semid = semget(key, nsems, semflg)) == -1) {

perror("semget: semget failed");

exit(1); }

else

...

Controlling Semaphores

semctl() changes permissions and other characteristics of a semaphore set. It is prototyped as follows:

int semctl(int semid, int semnum, int cmd, union semun arg);

It must be called with a valid semaphore ID, semid. The semnum value selects a semaphore within an array by its index. The cmd argument is one of the following control flags:

GETVAL

-- Return the value of a single semaphore.

SETVAL

-- Set the value of a single semaphore. In this case, arg is taken as arg.val, an int.

GETPID

-- Return the PID of the process that performed the last operation on the semaphore or array.

GETNCNT

-- Return the number of processes waiting for the value of a semaphore to increase.

GETZCNT

-- Return the number of processes waiting for the value of a particular semaphore to reach zero.

GETALL

-- Return the values for all semaphores in a set. In this case, arg is taken as arg.array, a pointer to an array of unsigned shorts (see below).

SETALL

-- Set values for all semaphores in a set. In this case, arg is taken as arg.array, a pointer to an array of unsigned shorts.

IPC\_STAT

-- Return the status information from the control structure for the semaphore set and place it in the data structure pointed to by arg.buf, a pointer to a buffer of type semid\_ds.

IPC\_SET

-- Set the effective user and group identification and permissions. In this case, arg is taken as arg.buf.

IPC\_RMID

-- Remove the specified semaphore set.

A process must have an effective user identification of owner, creator, or superuser to perform an IPC\_SET or IPC\_RMID command. Read and write permission is required as for the other control commands. The following code illustrates semctl ().

The fourth argument union semun arg is optional, depending upon the operation requested. If required it is of type union semun, which must be explicitly declared by the application program as:

union semun {

int val;

struct semid\_ds \*buf;

ushort \*array;

} arg;

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

union semun {

int val;

struct semid\_ds \*buf;

ushort \*array;

} arg;

int i;

int semnum = ....;

int cmd = GETALL; /\* get value \*/

...

i = semctl(semid, semnum, cmd, arg);

if (i == -1) {

perror("semctl: semctl failed");

exit(1);

}

else

...

Semaphore Operations

semop() performs operations on a semaphore set. It is prototyped by:

int semop(int semid, struct sembuf \*sops, size\_t nsops);

The semid argument is the semaphore ID returned by a previous semget() call. The sops argument is a pointer to an array of structures, each containing the following information about a semaphore operation:

The semaphore number

The operation to be performed

Control flags, if any.

The sembuf structure specifies a semaphore operation, as defined in <sys/sem.h>.

struct sembuf {

ushort\_t sem\_num; /\* semaphore number \*/

short sem\_op; /\* semaphore operation \*/

short sem\_flg; /\* operation flags \*/

};

The nsops argument specifies the length of the array, the maximum size of which is determined by the SEMOPM configuration option; this is the maximum number of operations allowed by a single semop() call, and is set to 10 by default. The operation to be performed is determined as follows:

A positive integer increments the semaphore value by that amount.

A negative integer decrements the semaphore value by that amount. An attempt to set a semaphore to a value less than zero fails or blocks, depending on whether IPC\_NOWAIT is in effect.

A value of zero means to wait for the semaphore value to reach zero.

There are two control flags that can be used with semop():

IPC\_NOWAIT

-- Can be set for any operations in the array. Makes the function return without changing any semaphore value if any operation for which IPC\_NOWAIT is set cannot be performed. The function fails if it tries to decrement a semaphore more than its current value, or tests a nonzero semaphore to be equal to zero.

SEM\_UNDO

-- Allows individual operations in the array to be undone when the process exits.

This function takes a pointer, sops, to an array of semaphore operation structures. Each structure in the array contains data about an operation to perform on a semaphore. Any process with read permission can test whether a semaphore has a zero value. To increment or decrement a semaphore requires write permission. When an operation fails, none of the semaphores is altered.

The process blocks (unless the IPC\_NOWAIT flag is set), and remains blocked until:

the semaphore operations can all finish, so the call succeeds,

the process receives a signal, or

the semaphore set is removed.

Only one process at a time can update a semaphore. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a semop() call, no updates are done until all operations on the array can finish successfully.

If a process with exclusive use of a semaphore terminates abnormally and fails to undo the operation or free the semaphore, the semaphore stays locked in memory in the state the process left it. To prevent this, the SEM\_UNDO control flag makes semop() allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to its previous state. If the process dies, the system applies the operations in the undo structures. This prevents an aborted process from leaving a semaphore set in an inconsistent state. If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with SEM\_UNDO in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this to restore the resource to a consistent state. When performing a semaphore operation with SEM\_UNDO in effect, you must also have it in effect for the call that will perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This ensures that, unless the process is aborted, the values applied to the undo structure are cancel to zero. When the undo structure reaches zero, it is removed.

NOTE:Using SEM\_UNDO inconsistently can lead to excessive resource consumption because allocated undo structures might not be freed until the system is rebooted.

The following code illustrates the semop() function:

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

...

int i;

int nsops; /\* number of operations to do \*/

int semid; /\* semid of semaphore set \*/

struct sembuf \*sops; /\* ptr to operations to perform \*/

...

if ((semid = semop(semid, sops, nsops)) == -1)

{

perror("semop: semop failed");

exit(1);

}

else

(void) fprintf(stderr, "semop: returned %d\n", i);

...

POSIX Semaphores: <semaphore.h>

POSIX semaphores are much lighter weight than are System V semaphores. A POSIX semaphore structure defines a single semaphore, not an array of up to twenty five semaphores. The POSIX semaphore functions are:

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.

The basic operation of these functions is essence the same as described above, except note there are more specialised functions, here. These are not discussed further here and the reader is referred to the online man pages for further details.

semaphore.c: Illustration of simple semaphore passing

/\* semaphore.c --- simple illustration of dijkstra's semaphore analogy

\*

\* We fork() a child process so that we have two processes running:

\* Each process communicates via a semaphore.

\* The respective process can only do its work (not much here)

\* When it notices that the semaphore track is free when it returns to 0

\* Each process must modify the semaphore accordingly

\*/

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

union semun {

int val;

struct semid\_ds \*buf;

ushort \*array;

};

main()

{ int i,j;

int pid;

int semid; /\* semid of semaphore set \*/

key\_t key = 1234; /\* key to pass to semget() \*/

int semflg = IPC\_CREAT | 0666; /\* semflg to pass to semget() \*/

int nsems = 1; /\* nsems to pass to semget() \*/

int nsops; /\* number of operations to do \*/

struct sembuf \*sops = (struct sembuf \*) malloc(2\*sizeof(struct sembuf));

/\* ptr to operations to perform \*/

/\* set up semaphore \*/

(void) fprintf(stderr, "\nsemget: Setting up seamaphore: semget(%#lx, %\

%#o)\n",key, nsems, semflg);

if ((semid = semget(key, nsems, semflg)) == -1) {

perror("semget: semget failed");

exit(1);

} else

(void) fprintf(stderr, "semget: semget succeeded: semid =\

%d\n", semid);

/\* get child process \*/

if ((pid = fork()) < 0) {

perror("fork");

exit(1);

}

if (pid == 0)

{ /\* child \*/

i = 0;

while (i < 3) {/\* allow for 3 semaphore sets \*/

nsops = 2;

/\* wait for semaphore to reach zero \*/

sops[0].sem\_num = 0; /\* We only use one track \*/

sops[0].sem\_op = 0; /\* wait for semaphore flag to become zero \*/

sops[0].sem\_flg = SEM\_UNDO; /\* take off semaphore asynchronous \*/

sops[1].sem\_num = 0;

sops[1].sem\_op = 1; /\* increment semaphore -- take control of track \*/

sops[1].sem\_flg = SEM\_UNDO | IPC\_NOWAIT; /\* take off semaphore \*/

/\* Recap the call to be made. \*/

(void) fprintf(stderr,"\nsemop:Child Calling semop(%d, &sops, %d) with:", semid, nsops);

for (j = 0; j < nsops; j++)

{

(void) fprintf(stderr, "\n\tsops[%d].sem\_num = %d, ", j, sops[j].sem\_num);

(void) fprintf(stderr, "sem\_op = %d, ", sops[j].sem\_op);

(void) fprintf(stderr, "sem\_flg = %#o\n", sops[j].sem\_flg);

}

/\* Make the semop() call and report the results. \*/

if ((j = semop(semid, sops, nsops)) == -1) {

perror("semop: semop failed");

}

else

{

(void) fprintf(stderr, "\tsemop: semop returned %d\n", j);

(void) fprintf(stderr, "\n\nChild Process Taking Control of Track: %d/3 times\n", i+1);

sleep(5); /\* DO Nothing for 5 seconds \*/

nsops = 1;

/\* wait for semaphore to reach zero \*/

sops[0].sem\_num = 0;

sops[0].sem\_op = -1; /\* Give UP COntrol of track \*/

sops[0].sem\_flg = SEM\_UNDO | IPC\_NOWAIT; /\* take off semaphore, asynchronous \*/

if ((j = semop(semid, sops, nsops)) == -1) {

perror("semop: semop failed");

}

else

(void) fprintf(stderr, "Child Process Giving up Control of Track: %d/3 times\n", i+1);

sleep(5); /\* halt process to allow parent to catch semaphor change first \*/

}

++i;

}

}

else /\* parent \*/

{ /\* pid hold id of child \*/

i = 0;

while (i < 3) { /\* allow for 3 semaphore sets \*/

nsops = 2;

/\* wait for semaphore to reach zero \*/

sops[0].sem\_num = 0;

sops[0].sem\_op = 0; /\* wait for semaphore flag to become zero \*/

sops[0].sem\_flg = SEM\_UNDO; /\* take off semaphore asynchronous \*/

sops[1].sem\_num = 0;

sops[1].sem\_op = 1; /\* increment semaphore -- take control of track \*/

sops[1].sem\_flg = SEM\_UNDO | IPC\_NOWAIT; /\* take off semaphore \*/

/\* Recap the call to be made. \*/

(void) fprintf(stderr,"\nsemop:Parent Calling semop(%d, &sops, %d) with:", semid, nsops);

for (j = 0; j < nsops; j++)

{

(void) fprintf(stderr, "\n\tsops[%d].sem\_num = %d, ", j, sops[j].sem\_num);

(void) fprintf(stderr, "sem\_op = %d, ", sops[j].sem\_op);

(void) fprintf(stderr, "sem\_flg = %#o\n", sops[j].sem\_flg);

}

/\* Make the semop() call and report the results. \*/

if ((j = semop(semid, sops, nsops)) == -1) {

perror("semop: semop failed");

}

else

{

(void) fprintf(stderr, "semop: semop returned %d\n", j);

(void) fprintf(stderr, "Parent Process Taking Control of Track: %d/3 times\n", i+1);

sleep(5); /\* Do nothing for 5 seconds \*/

nsops = 1;

/\* wait for semaphore to reach zero \*/

sops[0].sem\_num = 0;

sops[0].sem\_op = -1; /\* Give UP COntrol of track \*/

sops[0].sem\_flg = SEM\_UNDO | IPC\_NOWAIT; /\* take off semaphore, asynchronous \*/

if ((j = semop(semid, sops, nsops)) == -1) {

perror("semop: semop failed");

}

else

(void) fprintf(stderr, "Parent Process Giving up Control of Track: %d/3 times\n", i+1);

sleep(5); /\* halt process to allow child to catch semaphor change first \*/

}

++i;

}

}

}

The key elements of this program are as follows:

After a semaphore is created with as simple key 1234, two prcesses are forked.

Each process (parent and child) essentially performs the same operations:

Each process accesses the same semaphore track ( sops[].sem\_num = 0).

Each process waits for the track to become free and then attempts to take control of track

This is achieved by setting appropriate sops[].sem\_op values in the array.

Once the process has control it sleeps for 5 seconds (in reality some processing would take place in place of this simple illustration)

The process then gives up control of the track sops[1].sem\_op = -1

an additional sleep operation is then performed to ensure that the other process has time to access the semaphore before a subsequent (same process) semaphore read.

Note: There is no synchronisation here in this simple example an we have no control over how the OS will schedule the processes.

Some further example semaphore programs

The following suite of programs can be used to investigate interactively a variety of semaphore ideas (see exercises below).

The semaphore must be initialised with the semget.c program. The effects of controlling the semaphore queue and sending and receiving semaphore can be investigated with semctl.c and semop.c respectively.

semget.c: Illustrate the semget() function

/\*

\* semget.c: Illustrate the semget() function.

\*

\* This is a simple exerciser of the semget() function. It prompts

\* for the arguments, makes the call, and reports the results.

\*/

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

extern void exit();

extern void perror();

main()

{

key\_t key; /\* key to pass to semget() \*/

int semflg; /\* semflg to pass to semget() \*/

int nsems; /\* nsems to pass to semget() \*/

int semid; /\* return value from semget() \*/

(void) fprintf(stderr,

"All numeric input must follow C conventions:\n");

(void) fprintf(stderr,

"\t0x... is interpreted as hexadecimal,\n");

(void) fprintf(stderr, "\t0... is interpreted as octal,\n");

(void) fprintf(stderr, "\totherwise, decimal.\n");

(void) fprintf(stderr, "IPC\_PRIVATE == %#lx\n", IPC\_PRIVATE);

(void) fprintf(stderr, "Enter key: ");

(void) scanf("%li", &key);

(void) fprintf(stderr, "Enter nsems value: ");

(void) scanf("%i", &nsems);

(void) fprintf(stderr, "\nExpected flags for semflg are:\n");

(void) fprintf(stderr, "\tIPC\_EXCL = \t%#8.8o\n", IPC\_EXCL);

(void) fprintf(stderr, "\tIPC\_CREAT = \t%#8.8o\n",

IPC\_CREAT);

(void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);

(void) fprintf(stderr, "\towner alter = \t%#8.8o\n", 0200);

(void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);

(void) fprintf(stderr, "\tgroup alter = \t%#8.8o\n", 020);

(void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);

(void) fprintf(stderr, "\tother alter = \t%#8.8o\n", 02);

(void) fprintf(stderr, "Enter semflg value: ");

(void) scanf("%i", &semflg);

(void) fprintf(stderr, "\nsemget: Calling semget(%#lx, %

%#o)\n",key, nsems, semflg);

if ((semid = semget(key, nsems, semflg)) == -1) {

perror("semget: semget failed");

exit(1);

} else {

(void) fprintf(stderr, "semget: semget succeeded: semid =

%d\n",

semid);

exit(0);

}

}

semctl.c: Illustrate the semctl() function

/\*

\* semctl.c: Illustrate the semctl() function.

\*

\* This is a simple exerciser of the semctl() function. It lets you

\* perform one control operation on one semaphore set. It gives up

\* immediately if any control operation fails, so be careful not

to

\* set permissions to preclude read permission; you won't be able

to

\* reset the permissions with this code if you do.

\*/

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <time.h>

struct semid\_ds semid\_ds;

static void do\_semctl();

static void do\_stat();

extern char \*malloc();

extern void exit();

extern void perror();

char warning\_message[] = "If you remove read permission\

for yourself, this program will fail frequently!";

main()

{

union semun arg; /\* union to pass to semctl() \*/

int cmd, /\* command to give to semctl() \*/

i, /\* work area \*/

semid, /\* semid to pass to semctl() \*/

semnum; /\* semnum to pass to semctl() \*/

(void) fprintf(stderr,

"All numeric input must follow C conventions:\n");

(void) fprintf(stderr,

"\t0x... is interpreted as hexadecimal,\n");

(void) fprintf(stderr, "\t0... is interpreted as octal,\n");

(void) fprintf(stderr, "\totherwise, decimal.\n");

(void) fprintf(stderr, "Enter semid value: ");

(void) scanf("%i", &semid);

(void) fprintf(stderr, "Valid semctl cmd values are:\n");

(void) fprintf(stderr, "\tGETALL = %d\n", GETALL);

(void) fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);

(void) fprintf(stderr, "\tGETPID = %d\n", GETPID);

(void) fprintf(stderr, "\tGETVAL = %d\n", GETVAL);

(void) fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);

(void) fprintf(stderr, "\tIPC\_RMID = %d\n", IPC\_RMID);

(void) fprintf(stderr, "\tIPC\_SET = %d\n", IPC\_SET);

(void) fprintf(stderr, "\tIPC\_STAT = %d\n", IPC\_STAT);

(void) fprintf(stderr, "\tSETALL = %d\n", SETALL);

(void) fprintf(stderr, "\tSETVAL = %d\n", SETVAL);

(void) fprintf(stderr, "\nEnter cmd: ");

(void) scanf("%i", &cmd);

/\* Do some setup operations needed by multiple commands. \*/

switch (cmd) {

case GETVAL:

case SETVAL:

case GETNCNT:

case GETZCNT:

/\* Get the semaphore number for these commands. \*/

(void) fprintf(stderr, "\nEnter semnum value: ");

(void) scanf("%i", &semnum);

break;

case GETALL:

case SETALL:

/\* Allocate a buffer for the semaphore values. \*/

(void) fprintf(stderr,

"Get number of semaphores in the set.\n");

arg.buf = &semid\_ds;

do\_semctl(semid, 0, IPC\_STAT, arg);

if (arg.array =

(ushort \*)malloc((unsigned)

(semid\_ds.sem\_nsems \* sizeof(ushort)))) {

/\* Break out if you got what you needed. \*/

break;

}

(void) fprintf(stderr,

"semctl: unable to allocate space for %d values\n",

semid\_ds.sem\_nsems);

exit(2);

}

/\* Get the rest of the arguments needed for the specified

command. \*/

switch (cmd) {

case SETVAL:

/\* Set value of one semaphore. \*/

(void) fprintf(stderr, "\nEnter semaphore value: ");

(void) scanf("%i", &arg.val);

do\_semctl(semid, semnum, SETVAL, arg);

/\* Fall through to verify the result. \*/

(void) fprintf(stderr,

"Do semctl GETVAL command to verify results.\n");

case GETVAL:

/\* Get value of one semaphore. \*/

arg.val = 0;

do\_semctl(semid, semnum, GETVAL, arg);

break;

case GETPID:

/\* Get PID of last process to successfully complete a

semctl(SETVAL), semctl(SETALL), or semop() on the

semaphore. \*/

arg.val = 0;

do\_semctl(semid, 0, GETPID, arg);

break;

case GETNCNT:

/\* Get number of processes waiting for semaphore value to

increase. \*/

arg.val = 0;

do\_semctl(semid, semnum, GETNCNT, arg);

break;

case GETZCNT:

/\* Get number of processes waiting for semaphore value to

become zero. \*/

arg.val = 0;

do\_semctl(semid, semnum, GETZCNT, arg);

break;

case SETALL:

/\* Set the values of all semaphores in the set. \*/

(void) fprintf(stderr,

"There are %d semaphores in the set.\n",

semid\_ds.sem\_nsems);

(void) fprintf(stderr, "Enter semaphore values:\n");

for (i = 0; i < semid\_ds.sem\_nsems; i++) {

(void) fprintf(stderr, "Semaphore %d: ", i);

(void) scanf("%hi", &arg.array[i]);

}

do\_semctl(semid, 0, SETALL, arg);

/\* Fall through to verify the results. \*/

(void) fprintf(stderr,

"Do semctl GETALL command to verify results.\n");

case GETALL:

/\* Get and print the values of all semaphores in the

set.\*/

do\_semctl(semid, 0, GETALL, arg);

(void) fprintf(stderr,

"The values of the %d semaphores are:\n",

semid\_ds.sem\_nsems);

for (i = 0; i < semid\_ds.sem\_nsems; i++)

(void) fprintf(stderr, "%d ", arg.array[i]);

(void) fprintf(stderr, "\n");

break;

case IPC\_SET:

/\* Modify mode and/or ownership. \*/

arg.buf = &semid\_ds;

do\_semctl(semid, 0, IPC\_STAT, arg);

(void) fprintf(stderr, "Status before IPC\_SET:\n");

do\_stat();

(void) fprintf(stderr, "Enter sem\_perm.uid value: ");

(void) scanf("%hi", &semid\_ds.sem\_perm.uid);

(void) fprintf(stderr, "Enter sem\_perm.gid value: ");

(void) scanf("%hi", &semid\_ds.sem\_perm.gid);

(void) fprintf(stderr, "%s\n", warning\_message);

(void) fprintf(stderr, "Enter sem\_perm.mode value: ");

(void) scanf("%hi", &semid\_ds.sem\_perm.mode);

do\_semctl(semid, 0, IPC\_SET, arg);

/\* Fall through to verify changes. \*/

(void) fprintf(stderr, "Status after IPC\_SET:\n");

case IPC\_STAT:

/\* Get and print current status. \*/

arg.buf = &semid\_ds;

do\_semctl(semid, 0, IPC\_STAT, arg);

do\_stat();

break;

case IPC\_RMID:

/\* Remove the semaphore set. \*/

arg.val = 0;

do\_semctl(semid, 0, IPC\_RMID, arg);

break;

default:

/\* Pass unknown command to semctl. \*/

arg.val = 0;

do\_semctl(semid, 0, cmd, arg);

break;

}

exit(0);

}

/\*

\* Print indication of arguments being passed to semctl(), call

\* semctl(), and report the results. If semctl() fails, do not

\* return; this example doesn't deal with errors, it just reports

\* them.

\*/

static void

do\_semctl(semid, semnum, cmd, arg)

union semun arg;

int cmd,

semid,

semnum;

{

register int i; /\* work area \*/

void) fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d,

",

semid, semnum, cmd);

switch (cmd) {

case GETALL:

(void) fprintf(stderr, "arg.array = %#x)\n",

arg.array);

break;

case IPC\_STAT:

case IPC\_SET:

(void) fprintf(stderr, "arg.buf = %#x)\n", arg.buf);

break;

case SETALL:

(void) fprintf(stderr, "arg.array = [", arg.buf);

for (i = 0;i < semid\_ds.sem\_nsems;) {

(void) fprintf(stderr, "%d", arg.array[i++]);

if (i < semid\_ds.sem\_nsems)

(void) fprintf(stderr, ", ");

}

(void) fprintf(stderr, "])\n");

break;

case SETVAL:

default:

(void) fprintf(stderr, "arg.val = %d)\n", arg.val);

break;

}

i = semctl(semid, semnum, cmd, arg);

if (i == -1) {

perror("semctl: semctl failed");

exit(1);

}

(void) fprintf(stderr, "semctl: semctl returned %d\n", i);

return;

}

/\*

\* Display contents of commonly used pieces of the status

structure.

\*/

static void

do\_stat()

{

(void) fprintf(stderr, "sem\_perm.uid = %d\n",

semid\_ds.sem\_perm.uid);

(void) fprintf(stderr, "sem\_perm.gid = %d\n",

semid\_ds.sem\_perm.gid);

(void) fprintf(stderr, "sem\_perm.cuid = %d\n",

semid\_ds.sem\_perm.cuid);

(void) fprintf(stderr, "sem\_perm.cgid = %d\n",

semid\_ds.sem\_perm.cgid);

(void) fprintf(stderr, "sem\_perm.mode = %#o, ",

semid\_ds.sem\_perm.mode);

(void) fprintf(stderr, "access permissions = %#o\n",

semid\_ds.sem\_perm.mode & 0777);

(void) fprintf(stderr, "sem\_nsems = %d\n",

semid\_ds.sem\_nsems);

(void) fprintf(stderr, "sem\_otime = %s", semid\_ds.sem\_otime ?

ctime(&semid\_ds.sem\_otime) : "Not Set\n");

(void) fprintf(stderr, "sem\_ctime = %s",

ctime(&semid\_ds.sem\_ctime));

}

semop() Sample Program to Illustrate semop()

/\*

\* semop.c: Illustrate the semop() function.

\*

\* This is a simple exerciser of the semop() function. It lets you

\* to set up arguments for semop() and make the call. It then

reports

\* the results repeatedly on one semaphore set. You must have read

\* permission on the semaphore set or this exerciser will fail.

(It

\* needs read permission to get the number of semaphores in the set

\* and to report the values before and after calls to semop().)

\*/

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

static int ask();

extern void exit();

extern void free();

extern char \*malloc();

extern void perror();

static struct semid\_ds semid\_ds; /\* status of semaphore set \*/

static char error\_mesg1[] = "semop: Can't allocate space for %d\

semaphore values. Giving up.\n";

static char error\_mesg2[] = "semop: Can't allocate space for %d\

sembuf structures. Giving up.\n";

main()

{

register int i; /\* work area \*/

int nsops; /\* number of operations to do \*/

int semid; /\* semid of semaphore set \*/

struct sembuf \*sops; /\* ptr to operations to perform \*/

(void) fprintf(stderr,

"All numeric input must follow C conventions:\n");

(void) fprintf(stderr,

"\t0x... is interpreted as hexadecimal,\n");

(void) fprintf(stderr, "\t0... is interpreted as octal,\n");

(void) fprintf(stderr, "\totherwise, decimal.\n");

/\* Loop until the invoker doesn't want to do anymore. \*/

while (nsops = ask(&semid, &sops)) {

/\* Initialize the array of operations to be performed.\*/

for (i = 0; i < nsops; i++) {

(void) fprintf(stderr,

"\nEnter values for operation %d of %d.\n",

i + 1, nsops);

(void) fprintf(stderr,

"sem\_num(valid values are 0 <= sem\_num < %d): ",

semid\_ds.sem\_nsems);

(void) scanf("%hi", &sops[i].sem\_num);

(void) fprintf(stderr, "sem\_op: ");

(void) scanf("%hi", &sops[i].sem\_op);

(void) fprintf(stderr,

"Expected flags in sem\_flg are:\n");

(void) fprintf(stderr, "\tIPC\_NOWAIT =\t%#6.6o\n",

IPC\_NOWAIT);

(void) fprintf(stderr, "\tSEM\_UNDO =\t%#6.6o\n",

SEM\_UNDO);

(void) fprintf(stderr, "sem\_flg: ");

(void) scanf("%hi", &sops[i].sem\_flg);

}

/\* Recap the call to be made. \*/

(void) fprintf(stderr,

"\nsemop: Calling semop(%d, &sops, %d) with:",

semid, nsops);

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

{

(void) fprintf(stderr, "\nsops[%d].sem\_num = %d, ", i,

sops[i].sem\_num);

(void) fprintf(stderr, "sem\_op = %d, ", sops[i].sem\_op);

(void) fprintf(stderr, "sem\_flg = %#o\n",

sops[i].sem\_flg);

}

/\* Make the semop() call and report the results. \*/

if ((i = semop(semid, sops, nsops)) == -1) {

perror("semop: semop failed");

} else {

(void) fprintf(stderr, "semop: semop returned %d\n", i);

}

}

}

/\*

\* Ask if user wants to continue.

\*

\* On the first call:

\* Get the semid to be processed and supply it to the caller.

\* On each call:

\* 1. Print current semaphore values.

\* 2. Ask user how many operations are to be performed on the next

\* call to semop. Allocate an array of sembuf structures

\* sufficient for the job and set caller-supplied pointer to

that

\* array. (The array is reused on subsequent calls if it is big

\* enough. If it isn't, it is freed and a larger array is

\* allocated.)

\*/

static

ask(semidp, sopsp)

int \*semidp; /\* pointer to semid (used only the first time) \*/

struct sembuf \*\*sopsp;

{

static union semun arg; /\* argument to semctl \*/

int i; /\* work area \*/

static int nsops = 0; /\* size of currently allocated

sembuf array \*/

static int semid = -1; /\* semid supplied by user \*/

static struct sembuf \*sops; /\* pointer to allocated array \*/

if (semid < 0) {

/\* First call; get semid from user and the current state of

the semaphore set. \*/

(void) fprintf(stderr,

"Enter semid of the semaphore set you want to use: ");

(void) scanf("%i", &semid);

\*semidp = semid;

arg.buf = &semid\_ds;

if (semctl(semid, 0, IPC\_STAT, arg) == -1) {

perror("semop: semctl(IPC\_STAT) failed");

/\* Note that if semctl fails, semid\_ds remains filled

with zeros, so later test for number of semaphores will

be zero. \*/

(void) fprintf(stderr,

"Before and after values are not printed.\n");

} else {

if ((arg.array = (ushort \*)malloc(

(unsigned)(sizeof(ushort) \* semid\_ds.sem\_nsems)))

== NULL) {

(void) fprintf(stderr, error\_mesg1,

semid\_ds.sem\_nsems);

exit(1);

}

}

}

/\* Print current semaphore values. \*/

if (semid\_ds.sem\_nsems) {

(void) fprintf(stderr,

"There are %d semaphores in the set.\n",

semid\_ds.sem\_nsems);

if (semctl(semid, 0, GETALL, arg) == -1) {

perror("semop: semctl(GETALL) failed");

} else {

(void) fprintf(stderr, "Current semaphore values are:");

for (i = 0; i < semid\_ds.sem\_nsems;

(void) fprintf(stderr, " %d", arg.array[i++]));

(void) fprintf(stderr, "\n");

}

}

/\* Find out how many operations are going to be done in the

next

call and allocate enough space to do it. \*/

(void) fprintf(stderr,

"How many semaphore operations do you want %s\n",

"on the next call to semop()?");

(void) fprintf(stderr, "Enter 0 or control-D to quit: ");

i = 0;

if (scanf("%i", &i) == EOF || i == 0)

exit(0);

if (i > nsops) {

if (nsops)

free((char \*)sops);

nsops = i;

if ((sops = (struct sembuf \*)malloc((unsigned)(nsops \*

sizeof(struct sembuf)))) == NULL) {

(void) fprintf(stderr, error\_mesg2, nsops);

exit(2);

}

}

\*sopsp = sops;

return (i);

}