Semaphores

Remember file locking? Well, semaphores can be thought of as really generic advisory locking mechanisms. You can use them to control access to files, shared memory, and, well, just about anything you want. The basic functionality of a semaphore is that you can either set it, check it, or wait until it clears then set it ("test-n-set"). No matter how complex the stuff that follows gets, remember those three operations.

This document will provide an overview of semaphore functionality, and will end with a program that uses semaphores to control access to a file. (This task, admittedly, could easily be handled with file locking, but it makes a good example since it's easier to wrap your head around than, say, shared memory.)

Grabbing some semaphores

With System V IPC, you don't grab single semaphores; you grab *sets* of semaphores. You can, of course, grab a semaphore set that only has one semaphore in it, but the point is you can have a whole slew of semaphores just by creating a single semaphore set.

How do you create the semaphore set? It's done with a call to `semget()`, which returns the semaphore id (hereafter referred to as the *`semid`*):

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

What's the *`key`*? It's a unique identifier that is used by different processes to identify this semaphore set. (This *`key`* will be generated using `ftok()`, described in the Message Queues document.)

The next argument, *`nsems`*, is (you guessed it!) the number of semaphores in this semaphore set. The exact number is system dependent, but it's probably between 500 and 2000. If you're needing more (greedy wretch!), just get another semaphore set.

Finally, there's the *`semflg`* argument. This tells `semget()` what the permissions should be on the new semaphore set, whether you're creating a new set or just want to connect to an existing one, and other things that you can look up. For creating a new set, you can bit-wise or the access permissions with `IPC_CREAT`.

Here's an example call that generates the *`key`* with `ftok()` and creates a 10 semaphore set, with 666 (`rw-rw-rw-`) permissions:

```
#include <sys/ipc.h>
#include <sys/sem.h>

key_t key;
int semid;

key = ftok("/home/beej/somefile", 'E');
semid = semget(key, 10, 0666 | IPC_CREAT);
```

Congrats! You've created a new semaphore set! After running the program you can check it out with the `ipcs` command. (Don't forget to remove it when you're done with it with `ipcrm`!)

`semop()`: Atomic power!

All operations that set, get, or test-n-set a semaphore use the `semop()` system call. This system call is general purpose, and its functionality is dictated by a structure that is passed to it, `struct sembuf`:

```
struct sembuf {
    ushort sem_num;
    short sem_op;
```

```
    short sem_flg;
};
```

Of course, `sem_num` is the number of the semaphore in the set that you want to manipulate. Then, `sem_op` is what you want to do with that semaphore. This takes on different meanings, depending on whether `sem_op` is positive, negative, or zero, as shown in the following table:

`sem_op`	What happens
Positive	The value of `sem_op` is added to the semaphore's value. This is how a program uses a semaphore to mark a resource as allocated.
Negative	If the absolute value of `sem_op` is greater than the value of the semaphore, the calling process will block until the value of the semaphore reaches that of the absolute value of `sem_op`. Finally, the absolute value of `sem_op` will be subtracted from the semaphore's value. This is how a process releases a resource guarded by the semaphore.
Zero	This process will wait until the semaphore in question reaches 0.

Table 1. `sem_op` values and their effects.

So, basically, what you do is load up a `struct sembuf` with whatever values you want, then call `semop()`, like this:

```
int semop(int semid ,struct sembuf *sops, unsigned int nsops);
```

The *`semid`* argument is the number obtained from the call to `semget()`. Next is *`sops`*, which is a pointer to the `struct sembuf` that you filled with your semaphore commands. If you want, though, you can make an array of `struct sembuf`s in order to do a whole bunch of semaphore operations at the same time. The way `semop()` knows that you're doing this is the *`nsop`* argument, which tells how many `struct sembuf`s you're sending it. If you only have one, well, put `1` as this argument.

One field in the `struct sembuf` that I haven't mentioned is the `sem_flg` field which allows the program to specify flags the further modify the effects of the `semop()` call.

One of these flags is `IPC_NOWAIT` which, as the name suggests, causes the call to `semop()` to return with error `EAGAIN` if it encounters a situation where it would normally block. This is good for situations where you might want to "poll" to see if you can allocate a resource.

Another very useful flag is the `SEM_UNDO` flag. This causes `semop()` to record, in a way, the change made to the semaphore. When the program exits, the kernel will automatically undo all changes that were marked with the `SEM_UNDO` flag. Of course, your program should do its best to deallocate any resources it marks using the semaphore, but sometimes this isn't possible when your program gets a SIGKILL or some other awful crash happens.

Destroying a semaphore

There are two ways to get rid of a semaphore: one is to use the Unix command `ipcrm`. The other is through a call to `semctl()` with the proper arguments.

Now, I'm trying to compile this code under both Linux and HPUX, but I've found the the system calls differ. Linux passes a `union semun` to `semctl()`, but HPUX just uses a variable argument list in its place. I'll try to keep code clear for both, but I'll favor the Linux-style, since it is what is described in Steven's Unix Network Programming book.

Here is the Linux-style `union semun`, along with the `semctl()` call that will destroy the semaphore:

```
union semun {
    int val;              /* used for SETVAL only */
    struct semid_ds *buf; /* for IPC_STAT and IPC_SET */
    ushort *array;        /* used for GETALL and SETALL */
};
```

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

Notice that `union semun` just provides a way to pass either an `int`, a `struct semid_ds`, or a pointer to a `ushort`. It is this flexibility that the HPUX version of `semctl()` achieves with a variable argument list:

```
int semctl(int semid, int semnum, int cmd, ... /*arg*/);
```

In HPUX, instead of passing in a `union semun`, you just pass whatever value it asks for (`int` or otherwise). Check the man page for more information about your specific system. However, the code from here on out is Linux-style.

Where were we? Oh yeah--destroying a semaphore. Basically, you want to set *`semid`* to the semaphore ID you want to axe. The *`cmd`* should be set to `IPC_RMID`, which tells `semctl()` to remove this semaphore set. The two parameters *`semnum`* and *`arg`* have no meaning in the `IPC_RMID` context and can be set to anything.

Here's an example call to torch a semaphore set:

```
union semun dummy;
int semid;
.
.
semid = semget(...);
.
.
semctl(semid, 0, IPC_RMID, dummy);
```

Easy peasy.

Caveat

When you first create some semaphores, they're all initialized to zero. This is bad, since that means they're all marked as allocated; it takes another call (either to `semop()` or `semctl()` to mark them as free. What does this mean? Well, it means that creation of a semaphore is not *atomic* (in other words, a one-step process). If two processes are trying to create, initialize, and use a semaphore at the same time, a race condition might develop.

I get around this problem in the sample code by having a single process that creates and initializes the semaphore. The main process just accesses it, but never creates or destroys it.

Just be on the lookout. Stevens refers to this as the semaphore's "fatal flaw".

Sample programs

There are three of them, all of which will compile under Linux (and HPUX with modification). The first, `seminit.c`, creates and initializes the semaphore. The second, `semdemo.c`, performs some pretend file locking using the semaphore, in a demo very much like that in the File Locking document. Finally, `semrm.c` is used to destroy the semaphore (again, `ipcrm` could be used to accomplish this.)

The idea is to run `seminit.c` to create the semaphore. Try using `ipcs` from the command line to verify that it exists. Them run `semdemo.c` in a couple of windows and see how they interact. Finally, use `semrm.c` to remove the semaphore. You could also try removing the semaphore while running `semdemo.c` just to see what kinds of errors are generated.

Here's `seminit.c` (run this first!):

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/sem.h>

int main(void)
{
    key_t key;
    int semid;
    union semun arg;

    if ((key = ftok("semdemo.c", 'J')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* create a semaphore set with 1 semaphore: */
    if ((semid = semget(key, 1, 0666 | IPC_CREAT)) == -1) {
        perror("semget");
        exit(1);
    }

    /* initialize semaphore #0 to 1: */
    arg.val = 1;
    if (semctl(semid, 0, SETVAL, arg) == -1) {
        perror("semctl");
        exit(1);
    }

    return 0;
}
```

Here's semdemo.c:

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

int main(void)
{
    key_t key;
    int semid;
    struct sembuf sb = {0, -1, 0};  /* set to allocate resource */

    if ((key = ftok("semdemo.c", 'J')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* grab the semaphore set created by seminit.c: */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

    printf("Press return to lock: ");
    getchar();
    printf("Trying to lock...\n");

    if (semop(semid, &sb, 1) == -1) {
        perror("semop");
        exit(1);
    }

    printf("Locked.\n");
    printf("Press return to unlock: ");
    getchar();
```

```
        sb.sem_op = 1; /* free resource */
        if (semop(semid, &sb, 1) == -1) {
            perror("semop");
            exit(1);
        }

        printf("Unlocked\n");

        return 0;
    }
```

Here's semrm.c:

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

    int main(void)
    {
        key_t key;
        int semid;
        union semun arg;

        if ((key = ftok("semdemo.c", 'J')) == -1) {
            perror("ftok");
            exit(1);
        }

        /* grab the semaphore set created by seminit.c: */
        if ((semid = semget(key, 1, 0)) == -1) {
            perror("semget");
            exit(1);
        }

        /* remove it: */
        if (semctl(semid, 0, IPC_RMID, arg) == -1) {
            perror("semctl");
            exit(1);
        }

        return 0;
    }
```

Isn't that fun! I'm sure you'll give up Quake just to play with this semaphore stuff all day long!

Conclusions

Hmmm. I think I've understated the usefulness of semaphores. I assure you, they're very very very useful in a concurrency situation. They're often faster than regular file locks, too. Also, you can use them on other things that aren't files, such as Shared Memory Segments! In fact, it is sometimes hard to live without them, quite frankly.

Whenever you have multiple processes running through a critical section of code, man, you need semaphores. You have zillions of them--you might as well use 'em.

HPUX man pages

If you don't run HPUX, be sure to check your local man pages!

- ipcrm
- ipcs

- semctl()
- semget()
- semop()
