System V IPC (Semaphores and Shared Memory)

Specifying Keys

What about this *key* nonsense? How do we create one? Well, since the type `key_t` is actually just a `long`, you can use any number you want. But what if you hard-code the number and some other unrelated program hardcodes the same number but wants another queue? The solution is to use the `ftok()` function which generates a key from two arguments:

```
key_t ftok(const char *path, int id);
```

Ok, this is getting weird. Basically, *path* just has to be a file that this process can read. The other argument, *id* is usually just set to some arbitrary char, like 'A'. The `ftok()` function uses information about the named file (like inode number, etc.) and the *id* to generate a probably-unique *key* for your ipc. Programs that want to use the same queue must generate the same *key*, so they must pass the same parameters to `ftok()`.

Finally, it's time to make the call:

```
key = ftok("/home/ergin/somefile", 'b');
ipc_id = msgget(key, 0666 | IPC_CREAT);
```

In the above example, I set the permissions on the queue to `666` (or `rw-rw-rw-`, if that makes more sense to you). And now we have `msqid` which will be used to send and receive messages from the queue.

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!

Shared Memory

The cool thing about shared memory segments is that they are what they sound like: a segment of memory that is shared between processes. I mean, think of the potential of this! You could allocate a block a player information for a multi-player game and have each process access it at will! Fun, fun, fun.

There are, as usual, more gotchas to watch out for, but it's all pretty easy in the long run. See, you just connect to the shared memory segment, and get a pointer to the memory. You can read and write to this pointer and all changes you make will be visible to everyone else connected to the segment. There is nothing simpler. Well, there is, actually, but I was just trying to make you more comfortable.

Creating the segment and connecting

Similarly to other forms of System V IPC, a shared memory segment is created and connected to via the `shmget()` call:

```
int shmget(key_t key, size_t size, int shmflg);
```

Upon successful completion, `shmget()` returns an identifier for the shared memory segment. The *`key`* argument should be created the same was as shown in the Message Queues document, using `ftok()`. The next argument, *`size`*, is the size in bytes of the shared memory segment. Finally, the *`shmflg`* should be set to the permissions of the segment bitwise-ORd with `IPC_CREAT` if you want to create the segment, but can be `0` otherwise. (It doesn't hurt to specify `IPC_CREAT` every time--it will simply connect you if the segment already exists.)

Here's an example call that creates a 1K segment with `644` permissions (`rw-r--r--`):

```
key_t key;
int shmid;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

But how do you get a pointer to that data from the `shmid` handle? The answer is in the call `shmat()`, in the following section.

Attach me--getting a pointer to the segment

Before you can use a shared memory segment, you have to attach yourself to it using the `shmat()` call:

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

What does it all mean? Well, *`shmid`* is the shared memory ID you got from the call to `shmget()`. Next is *`shmaddr`*, which you can use to tell `shmat()` which specific address to use but you should just set it to `0` and let the OS choose the address for you. Finally, the *`shmflg`* can be set to `SHM_RDONLY` if you only want to read from it, `0` otherwise.

Here's a more complete example of how to get a pointer to a shared memory segment:

```
key_t key;
int shmid;
char *data;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);
```

And *bammo*! You have the pointer to the shared memory segment! Notice that `shmat()` returns a `void` pointer, and we're treating it, in this case, as a `char` pointer. You can treat is as anything you like, depending on what kind of data you have in there. Pointers to arrays of structures are just as acceptable as anything else.

Also, it's interesting to note that `shmat()` returns `-1` on failure. But how do you get `-1` in a `void` pointer? Just do a cast during the comparison to check for errors:

```
data = shmat(shmid, (void *)0, 0);
if (data == (char *)(-1))
    perror("shmat");
```

All you have to do now is change the data it points to normal pointer-style. There are some samples in the next section.

Reading and Writing

Lets say you have the `data` pointer from the above example. It is a `char` pointer, so we'll be reading and writing chars from it. Furthermore, for the sake of simplicity, lets say the 1K shared memory segment contains a null-terminated string.

It couldn't be easier. Since it's just a string in there, we can print it like this:

```
printf("shared contents: %s\n", data);
```

And we could store something in it as easily as this:

```
printf("Enter a string: ");
gets(data);
```

Of course, like I said earlier, you can have other data in there besides just `char`s. I'm just using them as an example. I'll just make the assumption that you're familiar enough with pointers in C that you'll be able to deal with whatever kind of data you stick in there.

Detaching from and deleting segments

When you're done with the shared memory segment, your program should detach itself from it using the `shmdt()` call:

```
int shmdt(void *shmaddr);
```

The only argument, *`shmaddr`*, is the address you got from `shmat()`. The function returns `-1` on error, `0` on success.

When you detach from the segment, it isn't destroyed. Nor is it removed when *everyone* detaches from it. You have to specifically destroy it using a call to `shmctl()`, similar to the control calls for the other System V IPC functions:

```
shmctl(shmid, IPC_RMID, NULL);
```

The above call deletes the shared memory segment, assuming no one else is attached to it. The `shmctl()` function does a lot more than this, though, and it worth looking into. (On your own, of course, since this is only an overview!)

As always, you can destroy the shared memory segment from the command line using the `ipcrm` Unix command. Also, be sure that you don't leave any usused shared memory segments sitting around wasting system resources. All the System V IPC objects you own can be viewed using the `ipcs` command.

Concurrency

What are concurrency issues? Well, since you have multiple processes modifying the shared memory segment, it is possible that certain errors could crop up when updates to the segment occur simultaneously. This *concurrent* access is almost always a problem when you have multiple writers to a shared object.

The way to get around this is to use Semaphores to lock the shared memory segment while a process is writing to it. (Sometimes the lock will encompass both a read an write to the shared memory, depending on what you're doing.)

A true discussion of concurrency is beyond the scope of this paper, and you might want to check out one of many books on the subject. I'll just leave it with this: if you start getting weird inconsistencies in your shared data when you connect two or more processes to it, you could very well have a concurrency problem.

Sample code

Now that I've primed you on all the dangers of concurrent access to a shared memory segment without using semaphores, I'll show you a demo that does just that. Since this isn't a mission-critical application, and it's unlikely that you'll be accessing the shared data at the same time as any other process, I'll just leave the semaphores out for the sake of simplicity.

This program does one of two things: if you run it with no command line parameters, it prints the contents of the shared memory segment. If you give it one command line parameter, it stores that parameter in the shared memory segment.

Here's the code for shmdemo.c:

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

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
        fprintf(stderr, "usage: shmdemo [data_to_write]\n");
        exit(1);
    }

    /* make the key: */
    if ((key = ftok("shmdemo.c", 'R')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(1);
    }

    /* attach to the segment to get a pointer to it: */
    data = shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }

    /* read or modify the segment, based on the command line: */
    if (argc == 2) {
        printf("writing to segment: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else
        printf("segment contains: \"%s\"\n", data);

    /* detach from the segment: */
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
    }

    return 0;
}
```

More commonly, a process will attach to the segment and run for a bit while other programs are changing and reading the shared segment. It's neat to watch one process update the segment and see the changes appear to other processes. Again, for simplicity, the sample code doesn't do that, but you can see how the data is shared between independent processes.

Also, there's no code in here for removing the segment--be sure to do that when you're done messing with it.