Shared Memory Segments

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.

HPUX man pages

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

- ftok()
- ipcrm
- ipcs
- shmat()
- shmctl()
- shmdt()
- shmget()
