Shared Memory and Semaphores

Keith Gaughan

March 22, 2003

Contents

1 What is Shared Memory?

Shared memory an IPC[1] mechanism native to UNIX. In essence, it's about two processes sharing a common segment of memory that they can both read to and write from to communicate with one another. Nothing more, nothing less. Just a chunk of memory. After you know where the shared memory segment is, it's just like other part of your process's address space, just like as if you'd just called `malloc()`.

Because it's just memory, shared memory is the fastest IPC mechanism of them all.

To use shared memory, you'll have to include the following:

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

1.1 Caveat Lector!

There's a problem with using shared memory, a rather nasty problem—race conditions.

Shared memory is, well, a *shared resource*. Without some way of letting the processes that have access to it know if it's safe to read and write to the shared memory area, you're leaving your code open to *race conditions*, and believe me, there be dragons!

This problem arises because these systems use what is known as *preemptive multitasking*, where the OS takes care of managing when and for how long each process gets to execute. In such a situation, the contents of the shared memory area can end up like stirfry very easily, and you really don't want this to happen[2].

One simple way of solving this problem is *semaphores*.

2 What are Semaphores?

Semaphores are another IPC mechanism available when developing on UNIX. They allow different processes to synchronise their access to certain resources.

In Computer Science, the most common and simplest kind of semaphore is called a *binary semaphore* because they have two states *locked* or *unlocked*. These act much like traffic lights, except there's no amber for you to mistake for *go faster*[3].

When a process wants exclusive access to a resource, shared memory being an example, they attempt to lock the semaphore associated with that resource. If the semaphore they are attempting to lock is already locked, the caller is suspended, otherwise they are granted to lock. When you've finished doing whatever you wanted to do, you unlock the resource and any processes that have attempted to lock that semaphore in the meantime are woken up again to attempt the lock again. This way only one process can have access to the resource at once[4].

In addition, semaphores can be used as a signalling mechanism. In the simple chat application appearing later in the notes, the instance of the application acting as the server locks a semaphore to suspend it until a client application unlocks the semaphore to signal that it's connected to the server and mapped the shared memory area onto its own address space.

Versatile wee yokes, aren't they!

2.1 UNIX Semaphores

UNIX uses a wierdly powerful version of semaphores called *recursive semaphores*. What that means is that the can have more than two states. For nearly all purposes, these kinds of semaphores are overkill, but

[1]InterProcess Communication

[2]Unless you're into some serious S&M and Bondage, of course

[3]I cycle—'nuff said.

[4]For those of you who've read the notes on threads, semaphores probably sound just like mutexes. Well, for the most part they are, only semaphores are a form of interprocess communication whereas mutexes are for *interthread* communication.

it is possible to get them to emulate binary semaphores using simple wrapper functions (these are supplied later).

When semaphores are allocated in UNIX, they're always allocated in chunks, and each one of these chunks has a unique ID.

3 Shared Memory Calls and Datatypes

Before we start building the chat application, it's useful to know about the functions and datatypes you'll need to know about to use shared memory.

3.1 shmget()

Allocates a shared memory segment.

```
int shmget(key_t key, int size, int flags);
```

key is the key associated with the shared memory segment you want. It's best just to pass `IPC_PRIVATE` in here.

size is the size in bytes of the shared memory segment you want allocated. Memory gets allocated in pages, so chances are you'll probably get a little more memory than you wanted.

flags indicate how you want the segment created and its access permissions. The general rule is just to use `IPC_CREAT | SHM_W | SHM_R here.`

The value returned is the id of the freshly allocated segment. Now all you have to do is get it attached to your address space, and the way about that is to call `shmat()`.

3.2 shmat()

Maps a shared memory segment onto your process's address space.

```
void* shmat(int id, const void* addr, int flags);
```

id is the id as returned by `shmget()` of the shared memory segment you wish to attach.

addr is the address you want the segment mapped at. It's best to pass `NULL`here as the system will choose a suitable unused address to attach the segment at itself. The only time you'll want to specify this yourself is when you've a specific chunk of data you want to share.

flags specifies various esoteric details about how it should be mapped. Just pass in `0` here.

What is returned is a pointer to where the segment has been mapped. You can now treat it almost like any other piece of memory you own.

3.3 shmdt()

Detaches the given segment. You need to do this when you no longer require the shared memory segment any longer, e.g. when you program is shutting down.

```
int shmdt(const void* addr);
```

addr is the address where the shared memory segment you wish to detach is mapped.

3.4 shmctl()

Lets you frob, twiddle and generally mess with a shared memory segment. Why this wasn't broken up into more calls, I don't know...

```
int shmctl(int id, int cmd, struct shmid_ds* buf);
```

id is the segment's id as returned by `shmget()`.

cmd is the command you want to perform. There are three: `IPC_STAT`, `IPC_SET`, and `IPC_RMID`.

buf points to a buffer used by `IPC_STAT` and `IPC_SET`. When `cmd` is set to `IPC_RMID`, this should be `NULL`.

3.4.1 IPC_STAT & IPC_SET

These are for getting and setting information about the shared memory segment. If you *really* feel you need to know how to use these, type `man shmctl`.

3.4.2 IPC_RMID

This is used to mark the segment in question as destroyed. It will actually be destroyed after the last detach, i.e. when all the processes using the segment are gone. This should *always* and *only* be done after you attach the segment with `shmat()`, or be prepared for your app to go down like an Airbus[5]...

3.5 Other stuff to remember

After a `fork()`, the child inherits all the attached segments. After an `exec()` or and `exit()`, all attached segments are detached, but *not destroyed*[6]. This is why you need to use `IPC_RMID`.

3.6 Some terribly useful wrappers

Here's some wrapper functions that should make dealing with shared memory just a bit easier. We'll need these later. Now, go knock yourself out.

```
/**
 * Allocates a shared memory segment.
 *
 * @param  n  Size (in bytes) of chunk to allocate.
 * @return Id of shared memory chunk.
 */
int AllocateSharedMemory(int n)
{
    assert(n > 0); /* Idiot-proof the call. */

    return shmget(IPC_PRIVATE, n, IPC_CREAT | SHM_R | SHM_W);
}

/**
 * Maps a shared memory segment onto our address space.
 *
 * @param  id  Shared memory block to map.
 * @return Address of mapped block.
 */
```

[5]If you didn't get that joke, you're probably a bit younger than me, so ignore it.

[6]This is one of those *soggy-haddock-making-contact-with-your-head* things.

```
void* MapSharedMemory(int id)
{
    void* addr;

    assert(id != 0); /* Idiot-proof the call. */

    addr = shmat(id, NULL, 0);  /* Attach the segment...     */
    shmctl(id, IPC_RMID, NULL); /* ...and mark it destroyed. */
    return addr;
}
```

4 Semaphore Calls and Datatypes

4.1 union semun

This is a `union` the semaphore library requires. For some ungodly reason[7], it's not specified in the `sys/sem.h` header file and we're required to specify it ourselves. It's needed by `semctl()`, which we'll encounter later. For now, just dump the following in your code:

```
#if !defined(__GNU_LIBRARY__) || defined(_SEM_SEMUN_UNDEFINED)
union semun
{
    int val;                // value for SETVAL
    struct semid_ds* buf;   // buffer for IPC_STAT, IPC_SET
    unsigned short*  array; // array for GETALL, SETALL
    struct seminfo*  __buf; // buffer for IPC_INFO
};
#endif
```

4.2 struct sembuf

Defines an operation to be performed by the `semop()` call. The members you need to know about are:

sem_num is the index of the first semaphore to perform the operation on. Note that the index starts from zero.

sem_op is the operation to perform. `semop()` covers this in greater detail.

sem_flg specifies how the operation is supposed to be treated when the process exits. Usually you'll want this to be `SEM_UNDO` so it will be undone when the process exits. Passing the flag `IPC_NOWAIT` will make sure the call to `semop()` doesn't block and instead fails if it would have blocked.

4.3 semget()

Gets a semaphore set. The value returned is its id, for use with other calls.

```
int semget(key_t key, int n, int flags);
```

key is the key associated with the semaphore set you want. Don't think about it—just use `IPC_PRIVATE`.

n is the number of semaphores the set should contain.

flags specifies how how the set should be allocated. `SHM_R | SHM_W` is the best thing to pass.

[7]Because the X/OPEN committee members smoked a bit too much Jamaican Gold, I reckon.

4.4 semop()

Performs a semaphore operation (i.e. incrementing, decrementing, etc.) on the selected members of a semaphore set. This is one of those ones that should really be a bunch of seperate calls.

```
int semop(int id, struct sembuf* op, unsigned n);
```

id is the semaphore set's id.

op is the operation to perform.

n is the number of semaphores to affect. You'll nearly always be passing in a value of `1` here.

`struct sembuf`'s `sem_op` field is important. It specifies what you want to do to the semaphore, be it incrementing, decrementing, or toasting over an open fire[8].

- A non-zero value will be added to the semaphore's value. Note that this means negative values indicate subtraction.
- A value of zero will make the operation block until the semaphore value becomes zero.

4.5 semctl()

A bit like `shmctl()`, but for semaphores. Again, ridiculously overcomplicated.

```
int semctl(int id, int iSem, int cmd, union semun arg);
```

id is the semaphore set id.

iSem is the semaphore you want to twiddle. Only valid with some of the commands.

cmd is the command you want to perform.

arg is used for fiddling with semaphore values. With everything but `cmd` set to `SETALL`, just pass `NULL`.

There are two values for `cmd` worth looking at: `SETALL` and `IPC_RMID`. For details on the others, type `man semctl`.

4.5.1 IPC_RMID

Automatically removes the semaphore set and awakens any processes waiting for a semaphore in the set to unlock. This should be done by the last process using the semaphores. `semctl()` will return `-1` if something goes horribly wrong with this.

4.5.2 SETALL

Initialises all the semaphores in the set with a given value. Best demonstrated with some sample code.

```
void SetAllSemaphores(int id, short* vals)
{
    union semun arg;

    assert(vals != NULL);

    arg.array = vals;
    semctl(id, 0, SETALL, arg);
}
```

The `array` field of `union semun` is used to specify an array of shorts holding the values you want each one of the semaphores in the set to have.

[8] Or maybe not. . .

4.6 Yet more terribly useful wrappers

These wrappers around the UNIX semaphores implement plain old binary semaphores, exactly the kind that are actually useful. We'll be using them later.

```
/**
 * Creates a new semaphore set.
 *
 * @param  n     Number of semaphores in set.
 * @param  vals  Default values to start off with.
 * @return Id of semaphore set.
 */
int CreateSemaphoreSet(int n, short* vals)
{
    union semun arg;
    int id;

    assert(n > 0);        /* You need at least one! */
    assert(vals != NULL); /* And they need initialising! */

    id = semget(IPC_PRIVATE, n, SHM_R | SHM_W);

    arg.array = vals;
    semctl(id, 0, SETALL, arg);

    return id;
}

/**
 * Frees up the given semaphore set.
 *
 * @param  id  Id of the semaphore group.
 */
void DeleteSemaphoreSet(int id)
{
    if (semctl(id, 0, IPC_RMID, NULL) == -1)
    {
        perror("Error releasing semaphore!");
        exit(EXIT_FAILURE);
    }
}

/**
 * Locks a semaphore within a semaphore set.
 *
 * @param  id  Semaphore set it belongs to.
 * @param  i   Actual semaphore to lock.
 *
 * @note If it’s already locked, you’re put to sleep.
 */
void LockSemaphore(int id, int i)
{
    struct sembuf sb;

    sb.sem_num = i;
    sb.sem_op = -1;
    sb.sem_flg = SEM_UNDO;
    semop(id, &sb, 1);
}

/**
 * Unlocks a semaphore within a semaphore set.
 *
 * @param  id  Semaphore set it belongs to.
 * @param  i   Actual semaphore to unlock.
 */
void UnlockSemaphore(int id, int i)
{
    struct sembuf sb;

    sb.sem_num = i;
    sb.sem_op = 1;
    sb.sem_flg = SEM_UNDO;
    semop(id, &sb, 1);
}
```

5 The obligatory crappy example: a chat system

In a vain attempt to come up with a simple and yet vaguely practical example, I'm introducing *CrapChat*, a rather crappy chat system. Why is it crap? Well, the two people chatting need to be on the same machine for a start and, so as to show semaphores working, each person needs to take turns talking. It's not replacing *IRC*, *ICQ*, *talk*[9] (type `man talk` some time) or any other instant messenging service any time soon. Something like a version of *Naughts and Crosses* would be better, but that's out of question unfortunately.

All that's missing from this are the semaphore and shared memory wrapper functions defined earlier. Enjoy!

```
/*                           vim:set ts=4 sw=4 noai sr sta et cin:
 * crapchat.c
 * by Keith Gaughan <kmgaughan@eircom.net>
 *
 * A really crappy chat application demoing Shared Memory and
 * Semaphores.
 *
 * Copyright (c) Keith Gaughan, 2003.
 * This software is free; you can redistribute it and/or modify it
 * under the terms of the Design Science License (DSL). If you
 * didn't recieve a copy of the DSL with this software, one can be
 * obtained at <http://www.dsl.org/copyleft/dsl.txt>.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <syslog.h>
#include <sys/types.h>
```

[9]Betcha didn't know that UNIX has had Instant Messenging for decades now!

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

// Declarations for wrapper functions...
int AllocateSharedMemory(int n);
void* MapSharedMemory(int id);
int CreateSemaphoreSet(int n, short* vals);
void DeleteSemaphoreSet(int id);
void LockSemaphore(int id, int i);
void UnlockSemaphore(int id, int i);

// The various semaphores used in the program.
enum
{
    SEM_USER_1, // Indicates it’s the first person’s turn.
    SEM_USER_2  // Indicates it’s the second person’s turn.
};

int main(int argc, char* argv[])
{
    int idShMem;   // Shared memory handle.
    int idSem;     // Semaphore set handle.
    char* buf;     // Shared memory buffer address.

    short vals[2]; // Values for initialising the semaphores.
    int   mySem;   // Semaphore indicating our user.
    int   yourSem; // Semaphore indicating the other user.

    puts("Welcome to CrapChat! Type ‘\\quit’ to exit.\n");

    // Get shared memory segment id off the command line.
    if (argc < 2)
    {
        // No segment id passed in, so we’ve got to create it.
        idShMem = AllocateSharedMemory(BUFSIZ);
        buf     = (char*) MapSharedMemory(idShMem);

        // We want each of the users to be blocked straight off
        // when they try to lock the shared memory area. When the
        // second user starts up, they’ll unlock the first so that
        // they can type. That’s what the zeros are for.
        vals[SEM_USER_1] = 0;
        vals[SEM_USER_2] = 0;
        idSem = CreateSemaphoreSet(2, vals);

        // Save the semaphore id in our shared memory so the other
        // user can get it.
        *((int*) buf) = idSem;

        // Record which semaphores we need to wait one and signal.
        mySem   = SEM_USER_1;
        yourSem = SEM_USER_2;
        // Write out the shared memory segment id so the other who
        // wants to chat with us can know.
        printf("You’re user one. Shared memory id is: %d\n",
               idShMem);
        puts("Waiting for user two...");
    }
    else
    {
        // We’ve a value! That means we’re the second user.
        idShMem = atoi(argv[1]);
        buf     = (char*) MapSharedMemory(isShMem);

        // Get the semaphore set id from shared memory.
        idSem = *((int*) buf);

        // Record which semaphores we need to wait one and signal.
        mySem   = SEM_USER_2;
        yourSem = SEM_USER_1;

        // Put an empty string in the shared memory.
        sprintf(buf, "");

        // Unlock the other user to signal they can talk.
        puts("You’re user two. Signalling to user one...");
        UnlockSemaphore(idSem, yourSem);
    }

    for (;;)
    {
        // Wait until it’s my turn to talk.
        LockSemaphore(idSem, semMe);

        // Did the other user exit?
        if (strcmp(buf, "\\quit\n") == 0)
            break;

        // Print the reply, if any.
        if (strlen(buf) > 0)
            printf("Reply: %s", buf);

        // Get your response.
        printf("> ");
        fgets(buf, BUFSIZ, stdin);

        // Hand over to the other user.
        UnlockSemaphore(idSem, semYou);

        // Do you want to exit?
        if (strcmp(buf, "\\quit\n") == 0)
            break;
    }

    // First user has to deallocate the semaphores.
    if (mySem == SEM_USER_1)
```

```
        DeleteSemaphoreSet(idSem);
}
```

6 I’m outta here!

If you’ve any questions or comments, you’ll always be able to get me at <kmgaughan@eircom.net>, though <kgaughan@mcom.cit.ie> should work too. The latest version of this document should be downloadable from either my website or my weblog.