Beej's Guide to Unix IPC

Brian "Beej Jorgensen" Hall
beej@beej.us

Version 1.1.2
December 15, 2010

Contents

1. Intro

You know what's easy? **`fork()`** is easy. You can fork off new processes all day and have them deal with individual chunks of a problem in parallel. Of course, its easiest if the processes don't have to communicate with one another while they're running and can just sit there doing their own thing.

However, when you start **`fork()`**'ing processes, you immediately start to think of the neat multi-user things you could do if the processes could talk to each other easily. So you try making a global array and then **`fork()`**'ing to see if it is shared. (That is, see if both the child and parent process use the same array.) Soon, of course, you find that the child process has its own copy of the array and the parent is oblivious to whatever changes the child makes to it.

How do you get these guys to talk to one another, share data structures, and be generally amicable? This document discusses several methods of *Interprocess Communication* (IPC) that can accomplish this, some of which are better suited to certain tasks than others.

1.1. Audience

If you know C or C++ and are pretty good using a Unix environment (or other POSIXey environment that supports these system calls) these documents are for you. If you aren't that good, well, don't sweat it—you'll be able to figure it out. I make the assumption, however, that you have a fair smattering of C programming experience.

As with Beej's Guide to Network Programming Using Internet Sockets, these documents are meant to springboard the aforementioned user into the realm of IPC by delivering a concise overview of various IPC techniques. This is not the definitive set of documents that cover this subject, by any means. Like I said, it is designed to simply give you a foothold in this, the exciting world of IPC.

1.2. Platform and Compiler

The examples in this document were compiled under Linux using **gcc**. They should compile anywhere a good Unix compiler is available.

1.3. Official Homepage

This official location of this document is `http://beej.us/guide/bgipc/`.

1.4. Email Policy

I'm generally available to help out with email questions so feel free to write in, but I can't guarantee a response. I lead a pretty busy life and there are times when I just can't answer a question you have. When that's the case, I usually just delete the message. It's nothing personal; I just won't ever have the time to give the detailed answer you require.

As a rule, the more complex the question, the less likely I am to respond. If you can narrow down your question before mailing it and be sure to include any pertinent information (like platform, compiler, error messages you're getting, and anything else you think might help me troubleshoot), you're much more likely to get a response. For more pointers, read ESR's document, How To Ask Questions The Smart Way.

If you don't get a response, hack on it some more, try to find the answer, and if it's still elusive, then write me again with the information you've found and hopefully it will be enough for me to help out.

Now that I've badgered you about how to write and not write me, I'd just like to let you know that I *fully* appreciate all the praise the guide has received over the years. It's a real morale boost, and it gladdens me to hear that it is being used for good! `:-)` Thank you!

1.5. Mirroring

You are more than welcome to mirror this site, whether publicly or privately. If you publicly mirror the site and want me to link to it from the main page, drop me a line at `beej@beej.us`.

1.6. Note for Translators

If you want to translate the guide into another language, write me at `beej@beej.us` and I'll link to your translation from the main page. Feel free to add your name and contact info to the translation.

Please note the license restrictions in the Copyright and Distribution section, below.

Sorry, but due to space constraints, I cannot host the translations myself.

1.7. Copyright and Distribution

Educators are freely encouraged to recommend or supply copies of this guide to their students.

Contact beej@beej.us for more information.

2. A `fork()` Primer

"Fork", aside from being one of those words that begins to appear very strange after you've typed it repeatedly, refers to the way Unix creates new processes. This document gives a quick and dirty **`fork()`** primer, since use of that system call will pop up in other IPC documents. If you already know all about **`fork()`**, you might as well skip this document.

2.1. "Seek ye the Gorge of Eternal Peril"

`fork()` can be thought of as a ticket to power. Power can sometimes be thought of as a ticket to destruction. Therefore, you should be careful while messing with **`fork()`** on your system, especially while people are cranking their nearly-late semester projects and are ready to nuke the first organism that brings the system to a halt. It's not that you should never play with **`fork()`**, you just have to be cautious. It's kind of like sword-swallowing; if you're careful, you won't disembowel yourself.

Since you're still here, I suppose I'd better deliver the goods. Like I said, **`fork()`** is how Unix starts new processes. Basically, how it works is this: the parent process (the one that already exists) **`fork()`**'s a child process (the new one). The child process gets a *copy* of the parent's data. *Voila!* You have two processes where there was only one!

Of course, there are all kinds of gotchas you must deal with when **`fork()`**ing processes or else your sysadmin will get irate with you when you fill of the system process table and they have to punch the reset button on the machine.

First of all, you should know something of process behavior under Unix. When a process dies, it doesn't really go away completely. It's dead, so it's no longer running, but a small remnant is waiting around for the parent process to pick up. This remnant contains the return value from the child process and some other goop. So after a parent process **`fork()`**s a child process, it must **`wait()`** (or **`waitpid()`**) for that child process to exit. It is this act of **`wait()`**ing that allows all remnants of the child to vanish.

Naturally, there is an exception to the above rule: the parent can ignore the `SIGCHLD` signal (`SIGCLD` on some older systems) and then it won't have to **`wait()`**. This can be done (on systems that support it) like this:

```
 main()
{
    signal(SIGCHLD, SIG_IGN);  /* now I don't have to wait()! */
    .
    .
    fork();fork();fork();  /* Rabbits, rabbits, rabbits! */
```

Now, when a child process dies and has not been **`wait()`**ed on, it will usually show up in

a **ps** listing as "`<defunct>`". It will remain this way until the parent **`wait()`**s on it, or it is dealt with as mentioned below.

Now there is another rule you must learn: when the parent dies before it **`wait()`**s for the child (assuming it is not ignoring `SIGCHLD`), the child is reparented to the **init** process (PID 1). This is not a problem if the child is still living well and under control. However, if the child is already defunct, we're in a bit of a bind. See, the original parent can no longer **`wait()`**, since it's dead. So how does **init** know to **`wait()`** for these *zombie processes*?

The answer: it's magic! Well, on some systems, **init** periodically destroys all the defunct processes it owns. On other systems, it outright refuses to become the parent of any defunct processes, instead destroying them immediately. If you're using one of the former systems, you could easily write a loop that fills up the process table with defunct processes owned by **init**. Wouldn't that make your sysadmin happy?

Your mission: make sure your parent process either ignores `SIGHCLD`, or **`wait()`**s for all the children it **`fork()`**s. Well, you don't *always* have to do that (like if you're starting a daemon or something), but you code with caution if you're a **`fork()`** novice. Otherwise, feel free to blast off into the stratosphere.

To summerize: children become defunct until the parent **`wait()`**s, unless the parent is ignoring `SIGCHLD`. Furthermore, children (living or defunct) whose parents die without **`wait()`**ing for them (again assuming the parent is not ignoring `SIGCHLD`) become children of the **init** process, which deals with them heavy-handedly.

2.2. "I'm mentally prepared! Give me *The Button*!"

Right! Here's an example of how to use **`fork()`**:

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

int main(void)
{
    pid_t pid;
    int rv;

    switch(pid = fork()) {
    case -1:
        perror("fork");  /* something went wrong */
        exit(1);         /* parent exits */

    case 0:
        printf(" CHILD: This is the child process!\n");
        printf(" CHILD: My PID is %d\n", getpid());
        printf(" CHILD: My parent's PID is %d\n", getppid());
        printf(" CHILD: Enter my exit status (make it small): ");
        scanf(" %d", &rv);
        printf(" CHILD: I'm outta here!\n");
        exit(rv);

    default:
        printf("PARENT: This is the parent process!\n");
```

```
        printf("PARENT: My PID is %d\n", getpid());
        printf("PARENT: My child's PID is %d\n", pid);
        printf("PARENT: I'm now waiting for my child to exit()...\n");
        wait(&rv);
        printf("PARENT: My child's exit status is: %d\n", WEXITSTATUS(rv));
        printf("PARENT: I'm outta here!\n");
    }

    return 0;
}
```

There is a ton of stuff to note from this example, so we'll just start from the top, shall we?

pid_t is the generic process type. Under Unix, this is a `short`. So, I call **`fork()`** and save the return value in the *pid* variable. **`fork()`** is easy, since it can only return three things:

`0`	If it returns `0`, you are the child process. You can get the parent's PID by calling **`getppid()`**. Of course, you can get your own PID by calling **`getpid()`**.
`-1`:	If it returns `-1`, something went wrong, and no child was created. Use **`perror()`** to see what happened. You've probably filled the process table—if you turn around you'll see your sysadmin coming at you with a fireaxe.
else:	Any other value returned by **`fork()`** means that you're the parent and the value returned is the PID of your child. This is the only way to get the PID of your child, since there is no **`getcpid()`** call (obviously due to the one-to-many relationship between parents and children.)

When the child finally calls **`exit()`**, the return value passed will arrive at the parent when it **`wait()`**s. As you can see from the **`wait()`** call, there's some weirdness coming into play when we print the return value. What's this `WEXITSTATUS()` stuff, anyway? Well, that is a macro that extracts the child's actual return value from the value **`wait()`** returns. Yes, there is more information buried in that `int`. I'll let you look it up on your own.

"How," you ask, "does **`wait()`** know which process to wait for? I mean, since the parent can have multiple children, which one does **`wait()`** actually wait for?" The answer is simple, my friends: it waits for whichever one happens to exit first. If you must, you can specify exactly which child to wait for by calling **`waitpid()`** with your child's PID as an argument.

Another interesting thing to note from the above example is that both parent and child use the *rv* variable. Does this mean that it is shared between the processes? *NO!* If it was, I wouldn't have written all this IPC stuff. *Each process has its own copy of all variables.* There is a lot of other stuff that is copied, too, but you'll have to read the **man** page to see what.

A final note about the above program: I used a switch statement to handle the **`fork()`**, and that's not exactly typical. Most often you'll see an **`if`** statement there; sometimes it's as short

as:

```
if (!fork()) {
        printf("I'm the child!\n");
        exit(0);
    } else {
        printf("I'm the parent!\n");
        wait(NULL);
    }
```

Oh yeah—the above example also demonstrates how to **`wait()`** if you don't care what the return value of the child is: you just call it with `NULL` as the argument.

2.3. Summary

Now you know all about the mighty **`fork()`** function! It's more useful that a wet bag of worms in most computationally intensive situations, and you can amaze your friends at parties. Additionally, it can help make you more attractive to members of the opposite sex, unless you're male.

3. Signals

There is a sometimes useful method for one process to bug another: signals. Basically, one process can "raise" a signal and have it delivered to another process. The destination process's signal handler (just a function) is invoked and the process can handle it.

The devil's in the details, of course, and in actuality what you are permitted to do safely inside your signal hander is rather limited. Nevertheless, signals provide a useful service.

For example, one process might want to stop another one, and this can be done by sending the signal `SIGSTOP` to that process. To continue, the process has to receive signal `SIGCONT`. How does the process know to do this when it receives a certain signal? Well, many signals are predefined and the process has a default signal handler to deal with it.

A default handler? Yes. Take `SIGINT` for example. This is the interrupt signal that a process receives when the user hits `^C`. The default signal handler for `SIGINT` causes the process to exit! Sound familiar? Well, as you can imagine, you can override the `SIGINT` to do whatever you want (or nothing at all!) You could have your process **`printf()`** "Interrupt?! No way, Jose!" and go about its merry business.

So now you know that you can have your process respond to just about any signal in just about any way you want. Naturally, there are exceptions because otherwise it would be too easy to understand. Take the ever popular `SIGKILL`, signal #9. Have you ever typed "**kill -9 *nnnn***" to kill a runaway process? You were sending it `SIGKILL`. Now you might also remember that no process can get out of a "**kill -9**", and you would be correct. `SIGKILL` is one of the signals you **can't** add your own signal handler for. The aforementioned `SIGSTOP` is also in this category.

(Aside: you often use the Unix "**kill**" command without specifying a signal to send...so

what signal is it? The answer: `SIGTERM`. You can write your own handler for `SIGTERM` so your process won't respond to a regular "**kill**", and the user must then use "**kill -9**" to destroy the process.)

Are all the signals predefined? What if you want to send a signal that has significance that only you understand to a process? There are two signals that aren't reserved: `SIGUSR1` and `SIGUSER2`. You are free to use these for whatever you want and handle them in whatever way you choose. (For example, my cd player program might respond to `SIGUSR1` by advancing to the next track. In this way, I could control it from the command line by typing "**kill -SIGUSR1** ***nnnn***".)

3.1. Catching Signals for Fun and Profit!

As you can guess the Unix "kill" command is one way to send signals to a process. By sheer unbelievable coincidence, there is a system call called **`kill()`** which does the same thing. It takes for its argument a signal number (as defined in *`signal.h`*) and a process ID. Also, there is a library routine called **`raise()`** which can be used to raise a signal within the same process.

The burning question remains: how do you catch a speeding `SIGTERM`? You need to call **`sigaction()`** and tell it all the gritty details about which signal you want to catch and which function you want to call to handle it.

Here's the **`sigaction()`** breakdown:

```
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

The first parameter, *`sig`* is which signal to catch. This can be (probably "should" be) a symbolic name from *`signal.h`* along the lines of `SIGINT`. That's the easy bit.

The next field, *`act`* is a pointer to a `struct sigaction` which has a bunch of fields that you can fill in to control the behavior of the signal handler. (A pointer to the signal handler function itself included in the `struct`.)

Lastly *`oact`* can be `NULL`, but if not, it returns the *old* signal handler information that was in place before. This is useful if you want to restore the previous signal handler at a later time.

We'll focus on these three fields in the `struct sigaction`:

Signal	Description
`sa_handler`	The signal handler function (or `SIG_IGN` to ignore the signal)
`sa_mask`	A set of signals to block while this one is being handled
`sa_flags`	Flags to modify the behavior of the handler, or `0`

What about that `sa_mask` field? When you're handling a signal, you might want to block other signals from being delivered, and you can do this by adding them to the `sa_mask` It's a "set", which means you can do normal set operations to manipulate them: **`sigemptyset()`**, **`sigfillset()`**, **`sigaddset()`**, **`sigdelset()`**, and **`sigismember()`**. In this example, we'll just clear the set and not block any other signals.

Examples always help! Here's one that handled `SIGINT`, which can be delivered by hitting `^C`, called *`sigint.c`*:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

void sigint_handler(int sig)
{
    write(0, "Ahhh! SIGINT!\n", 14);
}

int main(void)
{
    void sigint_handler(int sig); /* prototype */
    char s[200];
    struct sigaction sa;

    sa.sa_handler = sigint_handler;
    sa.sa_flags = 0; // or SA_RESTART
    sigemptyset(&sa.sa_mask);

    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    printf("Enter a string:\n");

    if (fgets(s, sizeof s, stdin) == NULL)
        perror("fgets");
    else
        printf("You entered: %s\n", s);

    return 0;
}
```

This program has two functions: **`main()`** which sets up the signal handler (using the **`sigaction()`** call), and **`sigint_handler()`** which is the signal handler, itself.

What happens when you run it? If you are in the midst of entering a string and you hit `^C`, the call to **`gets()`** fails and sets the global variable *`errno`* to `EINTR`. Additionally, **`sigint_handler()`** is called and does its routine, so you actually see:

```
Enter a string:
the quick brown fox jum^CAhhh! SIGINT!
fgets: Interrupted system call
```

And then it exits. Hey—what kind of handler is this, if it just exits anyway?

Well, we have a couple things at play, here. First, you'll notice that the signal handler was

called, because it printed "Ahhh! SIGINT!" But then **`fgets()`** returns an error, namely `EINTR`, or "Interrupted system call". See, some system calls can be interrupted by signals, and when this happens, they return an error. You might see code like this (sometimes cited as an excusable use of `goto`):

```
restart:
    if (some_system_call() == -1) {
        if (errno == EINTR) goto restart;
        perror("some_system_call");
        exit(1);
    }
```

Instead of using `goto` like that, you might be able to set your `sa_flags` to include `SA_RESTART`. For example, if we change our `SIGINT` handler code to look like this:

```
    sa.sa_flags = SA_RESTART;
```

Then our run looks more like this:

```
Enter a string:
Hello^CAhhh! SIGINT!
Er, hello!^CAhhh! SIGINT!
This time fer sure!
You entered: This time fer sure!
```

Some system calls are interruptible, and some can be restarted. It's system dependent.

3.2. The Handler is not Omnipotent

You have to be careful when you make function calls in your signal handler. Those functions must be "async safe", so they can be called without invoking undefined behavior.

You might be curious, for instance, why my signal handler, above, called **`write()`** to output the message instead of **`printf()`**. Well, the answer is that POSIX says that **`write()`** is async-safe (so is safe to call from within the handler), while **`printf()`** is not.

The library functions and system calls that are async-safe and can be called from within your signal handlers are (breath):

`_Exit()`, **`_exit()`**, **`abort()`**, **`accept()`**, **`access()`**, **`aio_error()`**, **`aio_return()`**, **`aio_suspend()`**, **`alarm()`**, **`bind()`**, **`cfgetispeed()`**, **`cfgetospeed()`**, **`cfsetispeed()`**, **`cfsetospeed()`**, **`chdir()`**, **`chmod()`**, **`chown()`**, **`clock_gettime()`**, **`close()`**, **`connect()`**, **`creat()`**, **`dup()`**, **`dup2()`**, **`execle()`**, **`execve()`**, **`fchmod()`**, **`fchown()`**, **`fcntl()`**, **`fdatasync()`**, **`fork()`**, **`fpathconf()`**, **`fstat()`**, **`fsync()`**, **`ftruncate()`**, **`getegid()`**, **`geteuid()`**, **`getgid()`**, **`getgroups()`**, **`getpeername()`**, **`getpgrp()`**, **`getpid()`**, **`getppid()`**, **`getsockname()`**, **`getsockopt()`**, **`getuid()`**, **`kill()`**, **`link()`**, **`listen()`**, **`lseek()`**, **`lstat()`**, **`mkdir()`**, **`mkfifo()`**, **`open()`**, **`pathconf()`**, **`pause()`**, **`pipe()`**, **`poll()`**, **`posix_trace_event()`**, **`pselect()`**, **`raise()`**, **`read()`**, **`readlink()`**, **`recv()`**, **`recvfrom()`**, **`recvmsg()`**, **`rename()`**, **`rmdir()`**, **`select()`**, **`sem_post()`**, **`send()`**, **`sendmsg()`**, **`sendto()`**, **`setgid()`**, **`setpgid()`**, **`setsid()`**, **`setsockopt()`**, **`setuid()`**, **`shutdown()`**, **`sigaction()`**, **`sigaddset()`**,

sigdelset(), **sigemptyset()**, **sigfillset()**, **sigismember()**, **sleep()**, **signal()**, **sigpause()**, **sigpending()**, **sigprocmask()**, **sigqueue()**, **sigset()**, **sigsuspend()**, **sockatmark()**, **socket()**, **socketpair()**, **stat()**, **symlink()**, **sysconf()**, **tcdrain()**, **tcflow()**, **tcflush()**, **tcgetattr()**, **tcgetpgrp()**, **tcsendbreak()**, **tcsetattr()**, **tcsetpgrp()**, **time()**, **timer_getoverrun()**, **timer_gettime()**, **timer_settime()**, **times()**, **umask()**, **uname()**, **unlink()**, **utime()**, **wait()**, **waitpid()**, and **write()**.

Of course, you can call your own functions from within your signal handler (as long they don't call any non-async-safe functions.)

But wait—there's more!

You also cannot safely alter any shared (e.g. global) data, with one notable exception: variables that are declared to be of storage class and type `volatile sig_atomic_t`.

Here's an example that handles `SIGUSR1` by setting a global flag, which is then examined in the main loop to see if the handler was called. This is *sigusr.c*:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

volatile sig_atomic_t got_usr1;

void sigusr1_handler(int sig)
{
    got_usr1 = 1;
}

int main(void)
{
    struct sigaction sa;

    got_usr1 = 0;

    sa.sa_handler = sigusr1_handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);

    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    while (!got_usr1) {
        printf("PID %d: working hard...\n", getpid());
        sleep(1);
    }

    printf("Done in by SIGUSR1!\n");

    return 0;
}
```

Fire it it up in one window, and then use the **kill -USR1** in another window to kill it. The *sigusr* program conveniently prints out its process ID so you can pass it to **kill**:

```
$ sigusr
PID 5023: working hard...
PID 5023: working hard...
PID 5023: working hard...
```

Then in the other window, send it the signal `SIGUSR1`:

```
$ kill -USR1 5023
```

And the program should respond:

```
PID 5023: working hard...
PID 5023: working hard...
Done in by SIGUSR1!
```

(And the response should be immediate even if **`sleep()`** has just been called—**`sleep()`** gets interrupted by signals.)

3.3. What about `signal()`

ANSI-C defines a function called **`signal()`** that can be used to catch signals. It's not as reliable or as full-featured as **`sigaction()`**, so use of **`signal()`**is generally discouraged.

3.4. Some signals to make you popular

Here is a list of signals you (most likely) have at your disposal:

Signal	Description
SIGABRT	Process abort signal.
SIGALRM	Alarm clock.
SIGFPE	Erroneous arithmetic operation.
SIGHUP	Hangup.
SIGILL	Illegal instruction.
SIGINT	Terminal interrupt signal.
SIGKILL	Kill (cannot be caught or ignored).
SIGPIPE	Write on a pipe with no one to read it.
SIGQUIT	Terminal quit signal.
SIGSEGV	Invalid memory reference.

SIGTERM	Termination signal.
SIGUSR1	User-defined signal 1.
SIGUSR2	User-defined signal 2.
SIGCHLD	Child process terminated or stopped.
SIGCONT	Continue executing, if stopped.
SIGSTOP	Stop executing (cannot be caught or ignored).
SIGTSTP	Terminal stop signal.
SIGTTIN	Background process attempting read.
SIGTTOU	Background process attempting write.
SIGBUS	Bus error.
SIGPOLL	Pollable event.
SIGPROF	Profiling timer expired.
SIGSYS	Bad system call.
SIGTRAP	Trace/breakpoint trap.
SIGURG	High bandwidth data is available at a socket.
SIGVTALRM	Virtual timer expired.
SIGXCPU	CPU time limit exceeded.
SIGXFSZ	File size limit exceeded.

Each signal has its own default signal handler, the behavior of which is defined in your local man pages.

3.5. What I have Glossed Over

Nearly all of it. There are tons of flags, realtime signals, mixing signals with threads, masking signals, **`longjmp()`** and signals, and more.

Of course, this is just a "getting started" guide, but in a last-ditch effort to give you more information, here is a list of man pages with more information:

Handling signals: **sigaction()** **sigwait()** **sigwaitinfo()** **sigtimedwait()** **sigsuspend()** **sigpending()**

Delivering signals: **kill()** **raise()** **sigqueue()**

Set operations: **sigemptyset()** **sigfillset()** **sigaddset()** **sigdelset()** **sigismember()**

Other: **sigprocmask()** **sigaltstack()** **siginterrupt()** **sigsetjmp()** **siglongjmp()** **signal()**

4. Pipes

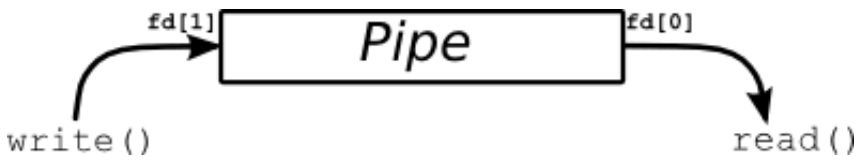
There is no form of IPC that is simpler than pipes. Implemented on every flavor of Unix, **pipe()** and **fork()** make up the functionality behind the "|" in "**ls | more**". They are marginally useful for cool things, but are a good way to learn about basic methods of IPC.

Since they're so very very easy, I shant spent much time on them. We'll just have some examples and stuff.

4.1. "These pipes are clean!"

Wait! Not so fast. I might need to define a "file descriptor" at this point. Let me put it this way: you know about "`FILE*`" from *`stdio.h`*, right? You know how you have all those nice functions like **fopen()**, **fclose()**, **fwrite()**, and so on? Well, those are actually high level functions that are implemented using *file descriptors*, which use system calls such as **open()**, **creat()**, **close()**, and **write()**. File descriptors are simply `int`s that are analogous to `FILE*`'s in *`stdio.h`*.

For example, *`stdin`* is file descriptor "0", *`stdout`* is "1", and *`stderr`* is "2". Likewise, any files you open using **fopen()** get their own file descriptor, although this detail is hidden from you. (This file descriptor can be retrived from the `FILE*` by using the **fileno()** macro from *`stdio.h`*.)



How a pipe is organized.

Basically, a call to the **pipe()** function returns a pair of file descriptors. One of these descriptors is connected to the write end of the pipe, and the other is connected to the read end. Anything can be written to the pipe, and read from the other end in the order it came in. On many systems, pipes will fill up after you write about 10K to them without reading anything out.

As a useless example, the following program creates, writes to, and reads from a pipe.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];
    char buf[30];

    if (pipe(pfds) == -1) {
        perror("pipe");
        exit(1);
    }

    printf("writing to file descriptor #%d\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf("reading from file descriptor #%d\n", pfds[0]);
    read(pfds[0], buf, 5);
    printf("read \"%s\"\n", buf);

    return 0;
}
```

As you can see, **`pipe()`** takes an array of two `int`s as an argument. Assuming no errors, it connects two file descriptors and returns them in the array. The first element of the array is the reading-end of the pipe, the second is the writing end.

4.2. `fork()` and `pipe()`—you have the power!

From the above example, it's pretty hard to see how these would even be useful. Well, since this is an IPC document, let's put a **`fork()`** in the mix and see what happens. Pretend that you are a top federal agent assigned to get a child process to send the word "test" to the parent. Not very glamorous, but no one ever said computer science would be the X-Files, Mulder.

First, we'll have the parent make a pipe. Secondly, we'll **`fork()`**. Now, the **`fork()`** man page tells us that the child will receive a copy of all the parent's file descriptors, and this includes a copy of the pipe's file descriptors. *Alors*, the child will be able to send stuff to the write-end of the pipe, and the parent will get it off the read-end. Like this:

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

int main(void)
{
    int pfds[2];
    char buf[30];

    pipe(pfds);

    if (!fork()) {
        printf(" CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
```

```
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }

    return 0;
}
```

Please note, your programs should have a lot more error checking than mine do. I leave it out on occasion to help keep things clear.

Anyway, this example is just like the previous one, except now we **`fork()`** of a new process and have it write to the pipe, while the parent reads from it. The resultant output will be something similar to the following:

```
PARENT: reading from pipe
 CHILD: writing to the pipe
 CHILD: exiting
PARENT: read "test"
```

In this case, the parent tried to read from the pipe before the child writes to it. When this happens, the parent is said to *block*, or sleep, until data arrives to be read. It seems that the parent tried to read, went to sleep, the child wrote and exited, and the parent woke up and read the data.

Hurrah!! You've just don't some interprocess communication! That was dreadfully simple, huh? I'll bet you are still thinking that there aren't many uses for **`pipe()`** and, well, you're probably right. The other forms of IPC are generally more useful and are often more exotic.

4.3. The search for Pipe as we know it

In an effort to make you think that pipes are actually reasonable beasts, I'll give you an example of using **`pipe()`** in a more familiar situation. The challenge: implement "**ls** | **wc -l**" in C.

This requires usage of a couple more functions you may never have heard of: **`exec()`** and **`dup()`**. The **`exec()`** family of functions replaces the currently running process with whichever one is passed to **`exec()`**. This is the function that we will use to run **ls** and **wc -l**. **`dup()`** takes an open file descriptor and makes a clone (a duplicate) of it. This is how we will connect the standard output of the **ls** to the standard input of **wc**. See, stdout of **ls** flows into the pipe, and the stdin of **wc** flows in from the pipe. The pipe fits right there in the middle!

Anyway, here is the code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];

    pipe(pfds);
```

```
    if (!fork()) {
        close(1);       /* close normal stdout */
        dup(pfds[1]);   /* make stdout same as pfds[1] */
        close(pfds[0]); /* we don't need this */
        execlp("ls", "ls", NULL);
    } else {
        close(0);       /* close normal stdin */
        dup(pfds[0]);   /* make stdin same as pfds[0] */
        close(pfds[1]); /* we don't need this */
        execlp("wc", "wc", "-l", NULL);
    }

    return 0;
}
```

I'm going to make another note about the **`close()`**/**`dup()`** combination since it's pretty weird. **`close(1)`** frees up file descriptor 1 (standard output). **`dup(pfds[1])`** makes a copy of the write-end of the pipe in the first available file descriptor, which is "1", since we just closed that. In this way, anything that **ls** writes to standard output (file descriptor 1) will instead go to *`pfds[1]`* (the write end of the pipe). The **wc** section of code works the same way, except in reverse.

4.4. Summary

There aren't many of these for such a simple topic. In fact, there are nearly just about none. Probably the best use for pipes is the one you're most accustomed to: sending the standard output of one command to the standard input of another. For other uses, it's pretty limiting and there are often other IPC techniques that work better.

5. FIFOs

A FIFO ("First In, First Out", pronounced "Fy-Foh") is sometimes known as a *named pipe*. That is, it's like a pipe, except that it has a name! In this case, the name is that of a file that multiple processes can **`open()`** and read and write to.

This latter aspect of FIFOs is designed to let them get around one of the shortcomings of normal pipes: you can't grab one end of a normal pipe that was created by an unrelated process. See, if I run two individual copies of a program, they can both call **`pipe()`** all they want and still not be able to speak to one another. (This is because you must **`pipe()`**, then **`fork()`** to get a child process that can communicate to the parent via the pipe.) With FIFOs, though, each unrelated process can simply **`open()`** the pipe and transfer data through it.

5.1. A New FIFO is Born

Since the FIFO is actually a file on disk, you have to do some fancy-schmancy stuff to create it. It's not that hard. You just have to call **`mknod()`** with the proper arguments. Here is a **`mknod()`** call that creates a FIFO:

```
mknod("myfifo", S_IFIFO | 0644 , 0);
```

In the above example, the FIFO file will be called "*myfifo*". The second argument is the creation mode, which is used to tell **mknod()** to make a FIFO (the S_IFIFO part of the OR) and sets access permissions to that file (octal 644, or rw-r--r--) which can also be set by ORing together macros from *sys/stat.h*. This permission is just like the one you'd set using the **chmod** command. Finally, a device number is passed. This is ignored when creating a FIFO, so you can put anything you want in there.

(An aside: a FIFO can also be created from the command line using the Unix **mknod** command.)

5.2. Producers and Consumers

Once the FIFO has been created, a process can start up and open it for reading or writing using the standard **open()** system call.

Since the process is easier to understand once you get some code in your belly, I'll present here two programs which will send data through a FIFO. One is *speak.c* which sends data through the FIFO, and the other is called *tick.c*, as it sucks data out of the FIFO.

Here is *speak.c*:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "american_maid"

int main(void)
{
    char s[300];
    int num, fd;

    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("waiting for readers...\n");
    fd = open(FIFO_NAME, O_WRONLY);
    printf("got a reader--type some stuff\n");

    while (gets(s), !feof(stdin)) {
        if ((num = write(fd, s, strlen(s))) == -1)
            perror("write");
        else
            printf("speak: wrote %d bytes\n", num);
    }

    return 0;
}
```

What **speak** does is create the FIFO, then try to **open()** it. Now, what will happen is that the **open()** call will *block* until some other process opens the other end of the pipe for

reading. (There is a way around this—see O_NDELAY, below.) That process is *tick.c*, shown here:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "american_maid"

int main(void)
{
    char s[300];
    int num, fd;

    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("waiting for writers...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("got a writer\n");

    do {
        if ((num = read(fd, s, 300)) == -1)
            perror("read");
        else {
            s[num] = '\0';
            printf("tick: read %d bytes: \"%s\"\n", num, s);
        }
    } while (num > 0);

    return 0;
}
```

Like *speak.c*, **tick** will block on the **`open()`** if there is no one writing to the FIFO. As soon as someone opens the FIFO for writing, **tick** will spring to life.

Try it! Start **speak** and it will block until you start **tick** in another window. (Conversely, if you start **tick**, it will block until you start **speak** in another window.) Type away in the **speak** window and **tick** will suck it all up.

Now, break out of **speak**. Notice what happens: the **`read()`** in **tick** returns 0, signifying EOF. In this way, the reader can tell when all writers have closed their connection to the FIFO. "What?" you ask "There can be multiple writers to the same pipe?" Sure! That can be very useful, you know. Perhaps I'll show you later in the document how this can be exploited.

But for now, lets finish this topic by seeing what happens when you break out of **tick** while **speak** is running. "Broken Pipe"! What does this mean? Well, what has happened is that when all readers for a FIFO close and the writer is still open, the writer will receiver the signal SIGPIPE the next time it tries to **`write()`**. The default signal handler for this signal prints "Broken Pipe" and exits. Of course, you can handle this more gracefully by catching SIGPIPE through the **`signal()`** call.

Finally, what happens if you have multiple readers? Well, strange things happen.

Sometimes one of the readers get everything. Sometimes it alternates between readers. Why do you want to have multiple readers, anyway?

5.3. `O_NDELAY`! I'm UNSTOPPABLE!

Earlier, I mentioned that you could get around the blocking **`open()`** call if there was no corresponding reader or writer. The way to do this is to call **`open()`** with the `O_NDELAY` flag set in the mode argument:

```
fd = open(FIFO_NAME, O_RDONLY | O_NDELAY);
```

This will cause **`open()`** to return `-1` if there are no processes that have the file open for reading.

Likewise, you can open the reader process using the `O_NDELAY` flag, but this has a different effect: all attempts to **`read()`** from the pipe will simply return `0` bytes read if there is no data in the pipe. (That is, the **`read()`** will no longer block until there is some data in the pipe.) Note that you can no longer tell if **`read()`** is returning `0` because there is no data in the pipe, or because the writer has exited. This is the price of power, but my suggestion is to try to stick with blocking whenever possible.

5.4. Concluding Notes

Having the name of the pipe right there on disk sure makes it easier, doesn't it? Unrelated processes can communicate via pipes! (This is an ability you will find yourself wishing for if you use normal pipes for too long.) Still, though, the functionality of pipes might not be quite what you need for your applications. Message queues might be more your speed, if your system supports them.

6. File Locking

File locking provides a very simple yet incredibly useful mechanism for coordinating file accesses. Before I begin to lay out the details, let me fill you in on some file locking secrets:

There are two types of locking mechanisms: mandatory and advisory. Mandatory systems will actually prevent **`read()`**s and **`write()`**s to file. Several Unix systems support them. Nevertheless, I'm going to ignore them throughout this document, preferring instead to talk solely about advisory locks. With an advisory lock system, processes can still read and write from a file while it's locked. Useless? Not quite, since there is a way for a process to check for the existence of a lock before a read or write. See, it's a kind of *cooperative* locking system. This is easily sufficient for almost all cases where file locking is necessary.

Since that's out of the way, whenever I refer to a lock from now on in this document, I'm referring to advisory locks. So there.

Now, let me break down the concept of a lock a little bit more. There are two types of (advisory!) locks: read locks and write locks (also referred to as shared locks and exclusive

locks, respectively.) The way read locks work is that they don't interfere with other read locks. For instance, multiple processes can have a file locked for reading at the same. However, when a process has an write lock on a file, no other process can activate either a read or write lock until it is relinquished. One easy way to think of this is that there can be multiple readers simultaneously, but there can only be one writer at a time.

One last thing before beginning: there are many ways to lock files in Unix systems. System V likes **`lockf()`**, which, personally, I think sucks. Better systems support **`flock()`** which offers better control over the lock, but still lacks in certain ways. For portability and for completeness, I'll be talking about how to lock files using **`fcntl()`**. I encourage you, though, to use one of the higher-level **`flock()`**-style functions if it suits your needs, but I want to portably demonstrate the full range of power you have at your fingertips. (If your System V Unix doesn't support the POSIX-y **`fcntl()`**, you'll have to reconcile the following information with your **`lockf()`** man page.)

6.1. Setting a lock

The **`fcntl()`** function does just about everything on the planet, but we'll just use it for file locking. Setting the lock consists of filling out a `struct flock` (declared in *`fcntl.h`*) that describes the type of lock needed, **`open()`**ing the file with the matching mode, and calling **`fcntl()`** with the proper arguments, *comme ça*:

```
struct flock fl;
int fd;

fl.l_type   = F_WRLCK;  /* F_RDLCK, F_WRLCK, F_UNLCK    */
fl.l_whence = SEEK_SET; /* SEEK_SET, SEEK_CUR, SEEK_END */
fl.l_start  = 0;        /* Offset from l_whence         */
fl.l_len    = 0;        /* length, 0 = to EOF           */
fl.l_pid    = getpid(); /* our PID                      */

fd = open("filename", O_WRONLY);

fcntl(fd, F_SETLKW, &fl);  /* F_GETLK, F_SETLK, F_SETLKW */
```

What just happened? Let's start with the `struct flock` since the fields in it are used to *describe* the locking action taking place. Here are some field definitions:

`l_type`	This is where you signify the type of lock you want to set. It's either `F_RDLCK`, `F_WRLCK`, or `F_UNLCK` if you want to set a read lock, write lock, or clear the lock, respectively.
`l_whence`	This field determines where the *`l_start`* field starts from (it's like an offset for the offset). It can be either `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, for beginning of file, current file position, or end of file.
`l_start`	This is the starting offset in bytes of the lock, relative to *`l_whence`*.
`l_len`	This is the length of the lock region in bytes (which starts from *`l_start`* which is relative to *`l_whence`*.

l_pid	The process ID of the process dealing with the lock. Use **getpid()** to get this.

In our example, we told it make a lock of type F_WRLCK (a write lock), starting relative to SEEK_SET (the beginning of the file), offset 0, length 0 (a zero value means "lock to end-of-file), with the PID set to **getpid()**.

The next step is to **open()** the file, since **flock()** needs a file descriptor of the file that's being locked. Note that when you open the file, you need to open it in the same *mode* as you have specified in the lock, as shown in the table, below. If you open the file in the wrong mode for a given lock type, **fcntl()** will return -1 and *errno* will be set to EBADF.

l_type	**mode**
F_RDLCK	O_RDONLY or O_RDWR
F_WRLCK	O_WRONLY or O_RDWR

Finally, the call to **fcntl()** actually sets, clears, or gets the lock. See, the second argument (the *cmd*) to **fcntl()** tells it what to do with the data passed to it in the struct flock. The following list summarizes what each **fcntl()** *cmd* does:

F_SETLKW	This argument tells **fcntl()** to attempt to obtain the lock requested in the struct flock structure. If the lock cannot be obtained (since someone else has it locked already), **fcntl()** will wait (block) until the lock has cleared, then will set it itself. This is a very useful command. I use it all the time.
F_SETLK	This function is almost identical to F_SETLKW. The only difference is that this one will not wait if it cannot obtain a lock. It will return immediately with -1. This function can be used to clear a lock by setting the *l_type* field in the struct flock to F_UNLCK.
F_GETLK	If you want to only check to see if there is a lock, but don't want to set one, you can use this command. It looks through all the file locks until it finds one that conflicts with the lock you specified in the struct flock. It then copies the conflicting lock's information into the struct and returns it to you. If it can't find a conflicting lock, **fcntl()** returns the struct as you passed it, except it sets the *l_type* field to F_UNLCK.

In our above example, we call **fcntl()** with F_SETLKW as the argument, so it blocks until it can set the lock, then sets it and continues.

6.2. Clearing a lock

Whew! After all the locking stuff up there, it's time for something easy: unlocking! Actually, this is a piece of cake in comparison. I'll just reuse that first example and add the code to unlock it at the end:

```
struct flock fl;
int fd;

fl.l_type   = F_WRLCK;  /* F_RDLCK, F_WRLCK, F_UNLCK    */
fl.l_whence = SEEK_SET; /* SEEK_SET, SEEK_CUR, SEEK_END */
fl.l_start  = 0;        /* Offset from l_whence         */
fl.l_len    = 0;        /* length, 0 = to EOF           */
fl.l_pid    = getpid(); /* our PID                      */

fd = open("filename", O_WRONLY);  /* get the file descriptor */
fcntl(fd, F_SETLKW, &fl);  /* set the lock, waiting if necessary */
.
.
.
fl.l_type   = F_UNLCK;  /* tell it to unlock the region */
fcntl(fd, F_SETLK, &fl); /* set the region to unlocked */
```

Now, I left the old locking code in there for high contrast, but you can tell that I just changed the *`l_type`* field to `F_UNLCK` (leaving the others completely unchanged!) and called **`fcntl()`** with `F_SETLK` as the command. Easy!

6.3. A demo program

Here, I will include a demo program, *`lockdemo.c`*, that waits for the user to hit return, then locks its own source, waits for another return, then unlocks it. By running this program in two (or more) windows, you can see how programs interact while waiting for locks.

Basically, usage is this: if you run **lockdemo** with no command line arguments, it tries to grab a write lock (`F_WRLCK`) on its source (*`lockdemo.c`*). If you start it with any command line arguments at all, it tries to get a read lock (`F_RDLCK`) on it.

Here's the source:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
                    /* l_type   l_whence  l_start  l_len  l_pid   */
    struct flock fl = {F_WRLCK, SEEK_SET,   0,      0,     0 };
    int fd;

    fl.l_pid = getpid();

    if (argc > 1)
        fl.l_type = F_RDLCK;

    if ((fd = open("lockdemo.c", O_RDWR)) == -1) {
        perror("open");
```

```
        exit(1);
    }

    printf("Press <RETURN> to try to get lock: ");
    getchar();
    printf("Trying to get lock...");

    if (fcntl(fd, F_SETLKW, &fl) == -1) {
        perror("fcntl");
        exit(1);
    }

    printf("got lock\n");
    printf("Press <RETURN> to release lock: ");
    getchar();

    fl.l_type = F_UNLCK;  /* set to unlock same region */

    if (fcntl(fd, F_SETLK, &fl) == -1) {
        perror("fcntl");
        exit(1);
    }

    printf("Unlocked.\n");

    close(fd);

    return 0;
}
```

Compile that puppy up and start messing with it in a couple windows. Notice that when one **lockdemo** has a read lock, other instances of the program can get their own read locks with no problem. It's only when a write lock is obtained that other processes can't get a lock of any kind.

Another thing to notice is that you can't get a write lock if there are any read locks on the same region of the file. The process waiting to get the write lock will wait until all the read locks are cleared. One upshot of this is that you can keep piling on read locks (because a read lock doesn't stop other processes from getting read locks) and any processes waiting for a write lock will sit there and starve. There isn't a rule anywhere that keeps you from adding more read locks if there is a process waiting for a write lock. You must be careful.

Practically, though, you will probably mostly be using write locks to guarantee exclusive access to a file for a short amount of time while it's being updated; that is the most common use of locks as far as I've seen. And I've seen them all...well, I've seen one...a small one...a picture—well, I've heard about them.

6.4. Summary

Locks rule. Sometimes, though, you might need more control over your processes in a producer-consumer situation. For this reason, if no other, you should see the document on System V semaphores if your system supports such a beast. They provide a more extensive and at least equally function equivalent to file locks.

7. Message Queues

Those people who brought us System V have seen fit to include some IPC goodies that have been implemented on various platforms (including Linux, of course.) This document describes the usage and functionality of the extremely groovy System V Message Queues!

As usual, I want to spew some overview at you before getting into the nitty-gritty. A message queue works kind of like a FIFO, but supports some additional functionality. Generally, see, messages are taken off the queue in the order they are put on. Specifically, however, there are ways to pull certain messages out of the queue before they reach the front. It's like cutting in line. (Incidentally, don't try to cut in line while visiting the Great America amusement park in Silicon Valley, as you can be arrested for it. They take cutting *very* seriously down there.)

In terms of usage, a process can create a new message queue, or it can connect to an existing one. In this, the latter, way two processes can exchange information through the same message queue. Score.

One more thing about System V IPC: when you create a message queue, it doesn't go away until you destroy it. All the processes that have ever used it can quit, but the queue will still exist. A good practice is to use the **ipcs** command to check if any of your unused message queues are just floating around out there. You can destroy them with the **ipcrm** command, which is preferable to getting a visit from the sysadmin telling you that you've grabbed every available message queue on the system.

7.1. Where's my queue?

Let's get something going! First of all, you want to connect to a queue, or create it if it doesn't exist. The call to accomplish this is the **`msgget()`** system call:

```
int msgget(key_t key, int msgflg);
```

`msgget()` returns the message queue ID on success, or `-1` on failure (and it sets *`errno`*, of course.)

The arguments are a little weird, but can be understood with a little brow-beating. The first, *`key`* is a system-wide unique identifier describing the queue you want to connect to (or create). Every other process that wants to connect to this queue will have to use the same *`key`*.

The other argument, *`msgflg`* tells **`msgget()`** what to do with queue in question. To create a queue, this field must be set equal to `IPC_CREAT` bit-wise OR'd with the permissions for this queue. (The queue permissions are the same as standard file permissions—queues take on the user-id and group-id of the program that created them.)

A sample call is given in the following section.

7.2. "Are you the Key Master?"

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 **`msgget()`**. 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:

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

key = ftok("/home/beej/somefile", 'b');
msqid = 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.

7.3. Sending to the queue

Once you've connected to the message queue using **`msgget()`**, you are ready to send and receive messages. First, the sending:

Each message is made up of two parts, which are defined in the template structure `struct msgbuf`, as defined in *`sys/msg.h`*:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

The field *`mtype`* is used later when retrieving messages from the queue, and can be set to any positive number. *`mtext`* is the data this will be added to the queue.

"What?! You can only put one byte arrays onto a message queue?! Worthless!!" Well, not exactly. You can use any structure you want to put messages on the queue, as long as the first element is a long. For instance, we could use this structure to store all kinds of goodies:

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    struct pirate_info {
        char name[30];
        char ship_type;
        int notoriety;
        int cruelty;
        int booty_value;
    } info;
};
```

Ok, so how do we pass this information to a message queue? The answer is simple, my friends: just use **`msgsnd()`**:

```
int msgsnd(int msqid, const void *msgp,
           size_t msgsz, int msgflg);
```

`msqid` is the message queue identifier returned by **`msgget()`**. The pointer *`msgp`* is a pointer to the data you want to put on the queue. *`msgsz`* is the size in bytes of the data to add to the queue (not counting the size of the *`mtype`* member). Finally, *`msgflg`* allows you to set some optional flag parameters, which we'll ignore for now by setting it to `0`.

When to get the size of the data to send, just subtract the **`sizeof(long)`** (the *`mtype`*) from the **`sizeof()`** the whole message buffer structure:

```
struct cheese_msgbuf {
    long mtype;
    char name[20];
    int type;
    float yumminess;
};

/* calculate the size of the data to send: */

int size = sizeof(struct cheese_msgbuf) - sizeof(long);
```

(Or if the payload is a simple `char[]`, you can use the length of the data as the message size.)

And here is a code snippet that shows one of our pirate structures being added to the message queue:

```
#include <sys/msg.h>
#include <stddef.h>

key_t key;
int msqid;
struct pirate_msgbuf pmb = {2, { "L'Olonais", 'S', 80, 10, 12035 } };

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

/* stick him on the queue */
msgsnd(msqid, &pmb, sizeof(struct pirate_msgbuf) - sizeof(long), 0);
```

Aside from remembering to error-check the return values from all these functions, this is all there is to it. Oh, yeah: note that I arbitrarily set the *`mtype`* field to `2` up there. That'll be important in the next section.

7.4. Receiving from the queue

Now that we have the dreaded pirate Francis L'Olonais stuck in our message queue, how do we get him out? As you can imagine, there is a counterpart to **`msgsnd()`**: it is **`msgrcv()`**. How imaginative.

A call to **`msgrcv()`** that would do it looks something like this:

```
#include <sys/msg.h>
#include <stddef.h>

key_t key;
int msqid;
struct pirate_msgbuf pmb; /* where L'Olonais is to be kept */

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

/* get him off the queue! */
msgrcv(msqid, &pmb, sizeof(struct pirate_msgbuf) - sizeof(long), 2, 0);
```

There is something new to note in the **`msgrcv()`** call: the `2`! What does it mean? Here's the synopsis of the call:

```
int msgrcv(int msqid, void *msgp, size_t msgsz,
           long msgtyp, int msgflg);
```

The `2` we specified in the call is the requested *`msgtyp`*. Recall that we set the *`mtype`* arbitrarily to `2` in the **`msgsnd()`** section of this document, so that will be the one that is retrieved from the queue.

Actually, the behavior of **`msgrcv()`** can be modified drastically by choosing a *`msgtyp`* that is positive, negative, or zero:

`msgtyp`	**Effect on `msgrcv()`**
Zero	Retrieve the next message on the queue, regardless of its *`mtype`*.
Positive	Get the next message with an *`mtype`* *equal to* the specified *`msgtyp`*.
Negative	Retrieve the first message on the queue whose *`mtype`* field is less than or equal to the absolute value of the *`msgtyp`* argument.

So, what will often be the case is that you'll simply want the next message on the queue, no matter what *`mtype`* it is. As such, you'd set the *`msgtyp`* parameter to `0`.

7.5. Destroying a message queue

There comes a time when you have to destroy a message queue. Like I said before, they will stick around until you explicitly remove them; it is important that you do this so you don't waste system resources. Ok, so you've been using this message queue all day, and it's getting old. You want to obliterate it. There are two ways:

1. Use the Unix command **ipcs** to get a list of defined message queues, then use the command **ipcrm** to delete the queue.
2. Write a program to do it for you.

Often, the latter choice is the most appropriate, since you might want your program to clean up the queue at some time or another. To do this requires the introduction of another

function: **`msgctl()`**.

The synopsis of **`msgctl()`** is:

```
int msgctl(int msqid, int cmd,
           struct msqid_ds *buf);
```

Of course, *`msqid`* is the queue identifier obtained from **`msgget()`**. The important argument is *`cmd`* which tells **`msgctl()`** how to behave. It can be a variety of things, but we're only going to talk about `IPC_RMID`, which is used to remove the message queue. The *`buf`* argument can be set to `NULL` for the purposes of `IPC_RMID`.

Say that we have the queue we created above to hold the pirates. You can destroy that queue by issuing the following call:

```
#include <sys/msg.h>
.
.
msgctl(msqid, IPC_RMID, NULL);
```

And the message queue is no more.

7.6. Sample programs, anyone?

For the sake of completeness, I'll include a brace of programs that will communicate using message queues. The first, *`kirk.c`* adds messages to the message queue, and *`spock.c`* retrieves them.

Here is the source for kirk.c:

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

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

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

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }
```

```
    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */

    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
        int len = strlen(buf.mtext);

        /* ditch newline at end, if it exists */
        if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

        if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
            perror("msgsnd");
    }

    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }

    return 0;
}
```

The way **kirk** works is that it allows you to enter lines of text. Each line is bundled into a message and added to the message queue. The message queue is then read by **spock**.

Here is the source for spock.c:

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

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {  /* same key as kirk.c */
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }

    printf("spock: ready to receive messages, captain.\n");

    for(;;) { /* Spock never quits! */
        if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {
            perror("msgrcv");
            exit(1);
        }
        printf("spock: \"%s\"\n", buf.mtext);
    }
```

```
    return 0;
}
```

Notice that **spock**, in the call to **`msgget()`**, doesn't include the `IPC_CREAT` option. We've left it up to **kirk** to create the message queue, and **spock** will return an error if he hasn't done so.

Notice what happens when you're running both in separate windows and you kill one or the other. Also try running two copies of **kirk** or two copies of **spock** to get an idea of what happens when you have two readers or two writers. Another interesting demonstration is to run **kirk**, enter a bunch of messages, then run **spock** and see it retrieve all the messages in one swoop. Just messing around with these toy programs will help you gain an understanding of what is really going on.

7.7. Summary

There is more to message queues than this short tutorial can present. Be sure to look in the man pages to see what else you can do, especially in the area of **`msgctl()`**. Also, there are more options you can pass to other functions to control how **`msgsnd()`** and **`msgrcv()`** handle if the queue is full or empty, respectively.

8. 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.)

8.1. 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 section.)

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**!)

Wait! Warning! *¡Advertencia! ¡No pongas las manos en la tolva!* (That's the only Spanish I learned while working at Pizza Hut in 1990. It was printed on the dough roller.) Look here:

When you first create some semaphores, they're all uninitialized; it takes another call to mark them as free (namely to **semop()** or **semctl()**—see the following sections.) What does this mean? Well, it means that creation of a semaphore is not *atomic* (in other words, it's not 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.

One way to get around this difficulty is by having a single init process that creates and initializes the semaphore long before the main processes begin to run. The main processes just access it, but never create nor destroy it.

Stevens refers to this problem as the semaphore's "fatal flaw". He solves it by creating the semaphore set with the `IPC_EXCL` flag. If process 1 creates it first, process 2 will return an error on the call (with *errno* set to `EEXIST`.) At that point, process 2 will have to wait until the semaphore is initialized by process 1. How can it tell? Turns out, it can repeatedly call **semctl()** with the `IPC_STAT` flag, and look at the *sem_otime* member of the returned `struct semid_ds` structure. If that's non-zero, it means process 1 has performed an operation on the semaphore with **semop()**, presumably to initialize it.

For an example of this, see the demonstration program *semdemo.c*, below, in which I generally reimplement Stevens' code.

In the meantime, let's hop to the next section and take a look at how to initialize our freshly-minted semaphores.

8.2. Controlling your semaphores with `semctl()`

Once you have created your semaphore sets, you have to initialize them to a positive value to show that the resource is available to use. The function **`semctl()`** allows you to do atomic value changes to individual semaphores or complete sets of semaphores.

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

`semid` is the semaphore set id that you get from your call to **`semget()`**, earlier. *`semnum`* is the ID of the semaphore that you wish to manipulate the value of. *`cmd`* is what you wish to do with the semaphore in question. The last "argument", "*`arg`*", if required, needs to be a `union semun`, which will be defined by you in your code to be one of these:

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

The various fields in the `union semun` are used depending on the value of the *`cmd`* parameter to **`setctl()`** (a partial list follows—see your local man page for more):

`cmd`	**Effect**
`SETVAL`	Set the value of the specified semaphore to the value in the *`val`* member of the passed-in `union semun`.
`GETVAL`	Return the value of the given semaphore.
`SETALL`	Set the values of all the semaphores in the set to the values in the array pointed to by the *`array`* member of the passed-in `union semun`. The *`semnum`* parameter to **`semctl()`** isn't used.
`GETALL`	Gets the values of all the semaphores in the set and stores them in the array pointed to by the *`array`* member of the passed-in `union semun`. The *`semnum`* parameter to **`semctl()`** isn't used.
`IPC_RMID`	Remove the specified semaphore set from the system. The *`semnum`* parameter is ignored.
`IPC_STAT`	Load status information about the semaphore set into the `struct semid_ds` structure pointed to by the *`buf`* member of the `union semun`.

For the curious, here are the contents of the `struct semid_ds` that is used in the `union semun`:

```
struct semid_ds {
    struct ipc_perm sem_perm;  /* Ownership and permissions
    time_t          sem_otime; /* Last semop time */
    time_t          sem_ctime; /* Last change time */
    unsigned short  sem_nsems; /* No. of semaphores in set */
};
```

We'll use that *`sem_otime`* member later on when we write our **`initsem()`** in the sample code, below.

8.3. `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`:

```
/* Warning!  Members might not be in this order! */

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**
Negative	Allocate resources. Block the calling process until the value of the semaphore is greater than or equal to the absolute value of *`sem_op`*. (That is, wait until enough resources have been freed by other processes for this one to allocate.) Then add (effectively subtract, since it's negative) the value of *`sem_op`* to the semaphore's value.
Positive	Release resources. The value of *`sem_op`* is added to the semaphore's value.
Zero	This process will wait until the semaphore in question reaches 0.

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 sembufs` 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 sembufs` 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.

8.4. 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 *cmd* set to `IPC_RMID`.

Basically, you want to call **semctl()** and 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 parameter *semnum* has no meaning in the `IPC_RMID` context and can just be set to zero.

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

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

Easy peasy.

8.5. Sample programs

There are two of them. The first, *semdemo.c*, creates the semaphore if necessary, and performs some pretend file locking on it in a demo very much like that in the File Locking document. The second program, *semrm.c* is used to destroy the semaphore (again, **ipcrm** could be used to accomplish this.)

The idea is to run run *semdemo.c* in a few windows and see how all the processes

interact. When you're done, 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 *semdemo.c*, including a function named **initsem()** that gets around the semaphore race conditions, Stevens-style:

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

#define MAX_RETRIES 10

union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};

/*
** initsem() -- more-than-inspired by W. Richard Stevens' UNIX Network
** Programming 2nd edition, volume 2, lockvsem.c, page 295.
*/
int initsem(key_t key, int nsems)  /* key from ftok() */
{
    int i;
    union semun arg;
    struct semid_ds buf;
    struct sembuf sb;
    int semid;

    semid = semget(key, nsems, IPC_CREAT | IPC_EXCL | 0666);

    if (semid >= 0) { /* we got it first */
        sb.sem_op = 1; sb.sem_flg = 0;
        arg.val = 1;

        printf("press return\n"); getchar();

        for(sb.sem_num = 0; sb.sem_num < nsems; sb.sem_num++) {
            /* do a semop() to "free" the semaphores. */
            /* this sets the sem_otime field, as needed below. */
            if (semop(semid, &sb, 1) == -1) {
                int e = errno;
                semctl(semid, 0, IPC_RMID); /* clean up */
                errno = e;
                return -1; /* error, check errno */
            }
        }

    } else if (errno == EEXIST) { /* someone else got it first */
        int ready = 0;

        semid = semget(key, nsems, 0); /* get the id */
        if (semid < 0) return semid; /* error, check errno */

        /* wait for other process to initialize the semaphore: */
        arg.buf = &buf;
        for(i = 0; i < MAX_RETRIES && !ready; i++) {
            semctl(semid, nsems-1, IPC_STAT, arg);
```

```
            if (arg.buf->sem_otime != 0) {
                ready = 1;
            } else {
                sleep(1);
            }
        }
        if (!ready) {
            errno = ETIME;
            return -1;
        }
    } else {
        return semid; /* error, check errno */
    }

    return semid;
}

int main(void)
{
    key_t key;
    int semid;
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = -1;  /* set to allocate resource */
    sb.sem_flg = SEM_UNDO;

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

    /* grab the semaphore set created by seminit.c: */
    if ((semid = initsem(key, 1)) == -1) {
        perror("initsem");
        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* for removing the semaphore when you're done:

```
#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!

8.6. Summary

I might have 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.

9. 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.

9.1. 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.

9.2. 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.

9.3. 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.

9.4. 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.

9.5. 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 the Wikipedia article on the matter. 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.

9.6. 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.

10. Memory Mapped Files

There comes a time when you want to read and write to and from files so that the information is shared between processes. Think of it this way: two processes both open the same file and both read and write from it, thus sharing the information. The problem is, sometimes it's a pain to do all those **`fseek()`**s and stuff to get around. Wouldn't it be easier if you could just map a section of the file to memory, and get a pointer to it? Then you could simply use pointer arithmetic to get (and set) data in the file.

Well, this is exactly what a memory mapped file is. And it's really easy to use, too. A few simple calls, mixed with a few simple rules, and you're mapping like a mad-person.

10.1. Mapmaker

Before mapping a file to memory, you need to get a file descriptor for it by using the **`open()`** system call:

```
int fd;

fd = open("mapdemofile", O_RDWR);
```

In this example, we've opened the file for read/write access. You can open it in whatever mode you want, but it has to match the mode specified in the *`prot`* parameter to the **`mmap()`** call, below.

To memory map a file, you use the **`mmap()`** system call, which is defined as follows:

```
void *mmap(void *addr, size_t len, int prot,
           int flags, int fildes, off_t off);
```

What a slew of parameters! Here they are, one at a time:

`addr` This is the address we want the file mapped into. The best way to use this is to set it to `(caddr_t)0` and let the OS choose it for you. If you tell it to use an address the OS doesn't like (for instance, if it's not a multiple of the virtual memory page size), it'll give you an error.

`len` This parameter is the length of the data we want to map into memory. This can be any length you want. (Aside: if *`len`* not a multiple of the virtual memory page size, you will get a blocksize that is rounded up to that size. The extra bytes will be 0, and any changes you make to them will not modify the file.)

`prot` The "protection" argument allows you to specify what kind of access this process has to the memory mapped region. This can be a bitwise-ORd mixture of the following values: `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`, for read, write, and execute permissions, respectively. The value specified here must be equivalent to the mode specified in the **`open()`** system call that is used to get the file descriptor.

`flags` There are just miscellaneous flags that can be set for the system call. You'll want to set it to `MAP_SHARED` if you're planning to share your changes to the file with other processes, or `MAP_PRIVATE` otherwise. If you set it to the latter, your process will get a copy of the mapped region, so any changes you make to it will not be reflected in the original file—thus, other processes will not be able to see them. We won't talk about `MAP_PRIVATE` here at all, since it doesn't have much to do with IPC.

fildes	This is where you put that file descriptor you opened earlier.
off	This is the offset in the file that you want to start mapping from. A restriction: this *must* be a multiple of the virtual memory page size. This page size can be obtained with a call to **getpagesize()**.

As for return values, as you might have guessed, **mmap()** returns -1 on error, and sets *errno*. Otherwise, it returns a pointer to the start of the mapped data.

Anyway, without any further ado, we'll do a short demo that maps the second "page" of a file into memory. First we'll **open()** it to get the file descriptor, then we'll use **getpagesize()** to get the size of a virtual memory page and use this value for both the *len* and the *off*. In this way, we'll start mapping at the second page, and map for one page's length. (On my Linux box, the page size is 4K.)

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>

int fd, pagesize;
char *data;

fd = open("foo", O_RDONLY);
pagesize = getpagesize();
data = mmap((caddr_t)0, pagesize, PROT_READ, MAP_SHARED, fd,
pagesize);
```

Once this code stretch has run, you can access the first byte of the mapped section of file using *data[0]*. Notice there's a lot of type conversion going on here. For instance, **mmap()** returns caddr_t, but we treat it as a char*. Well, the fact is that caddr_t usually is defined to be a char*, so everything's fine.

Also notice that we've mapped the file PROT_READ so we have read-only access. Any attempt to write to the data (data[0] = 'B', for example) will cause a segmentation violation. Open the file O_RDWR with *prot* set to PROT_READ|PROT_WRITE if you want read-write access to the data.

10.2. Unmapping the file

There is, of course, a **munmap()** function to un-memory map a file:

```
int munmap(caddr_t addr, size_t len);
```

This simply unmaps the region pointed to by *addr* (returned from **mmap()**) with length *len* (same as the *len* passed to **mmap()**). **munmap()** returns -1 on error and sets the errno variable.

Once you've unmapped a file, any attempts to access the data through the old pointer will result in a segmentation fault. You have been warned!

A final note: the file will automatically unmap if your program exits, of course.

10.3. Concurrency, again?!

If you have multiple processes manipulating the data in the same file concurrently, you could be in for troubles. You might have to lock the file or use semaphores to regulate access to the file while a process messes with it. Look at the Shared Memory document for a (very little bit) more concurrency information.

10.4. A simple sample

Well, it's code time again. I've got here a demo program that maps its own source to memory and prints the byte that's found at whatever offset you specify on the command line.

The program restricts the offsets you can specify to the range 0 through the file length. The file length is obtained through a call to **`stat()`** which you might not have seen before. It returns a structure full of file info, one field of which is the size in bytes. Easy enough.

Here is the source for *`mmapdemo.c`*:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    int fd, offset;
    char *data;
    struct stat sbuf;

    if (argc != 2) {
        fprintf(stderr, "usage: mmapdemo offset\n");
        exit(1);
    }

    if ((fd = open("mmapdemo.c", O_RDONLY)) == -1) {
        perror("open");
        exit(1);
    }

    if (stat("mmapdemo.c", &sbuf) == -1) {
        perror("stat");
        exit(1);
    }

    offset = atoi(argv[1]);
    if (offset < 0 || offset > sbuf.st_size-1) {
        fprintf(stderr, "mmapdemo: offset must be in the range 0-%d\n", \
                                                            sbuf.st_size-1);
        exit(1);
    }

    data = mmap((caddr_t)0, sbuf.st_size, PROT_READ, MAP_SHARED, fd, 0)) \
                                                        == (caddr_t)(-1)) {
    if (data == (caddr_t)(-1)) {
        perror("mmap");
```

```
        exit(1);
    }

    printf("byte at offset %d is '%c'\n", offset, data[offset]);

    return 0;
}
```

That's all there is to it. Compile that sucker up and run it with some command line like:

```
$ mmapdemo 30
byte at offset 30 is 'e'
```

I'll leave it up to you to write some really cool programs using this system call.

10.5. Summary

Memory mapped files can be very useful, especially on systems that don't support shared memory segments. In fact, the two are very similar in most respects. (Memory mapped files are committed to disk, too, so this could even be an advantage, yes?) With file locking or semaphores, data in a memory mapped file can easily be shared between multiple processes.

11. Unix Sockets

Remember FIFOs? Remember how they can only send data in one direction, just like a Pipes? Wouldn't it be grand if you could send data in both directions like you can with a socket?

Well, hope no longer, because the answer is here: Unix Domain Sockets! In case you're still wondering what a socket is, well, it's a two-way communications pipe, which can be used to communicate in a wide variety of *domains*. One of the most common domains sockets communicate over is the Internet, but we won't discuss that here. We will, however, be talking about sockets in the Unix domain; that is, sockets that can be used between processes on the same Unix system.

Unix sockets use many of the same function calls that Internet sockets do, and I won't be describing all of the calls I use in detail within this document. If the description of a certain call is too vague (or if you just want to learn more about Internet sockets anyway), I arbitrarily suggest *Beej's Guide to Network Programming using Internet Sockets*. I know the author personally.

11.1. Overview

Like I said before, Unix sockets are just like two-way FIFOs. However, all data communication will be taking place through the sockets interface, instead of through the file interface. Although Unix sockets are a special file in the file system (just like FIFOs), you won't be using **`open()`** and **`read()`**—you'll be using **`socket()`**, **`bind()`**, **`recv()`**, etc.

When programming with sockets, you'll usually create server and client programs. The

server will sit listening for incoming connections from clients and handle them. This is very similar to the situation that exists with Internet sockets, but with some fine differences.

For instance, when describing which Unix socket you want to use (that is, the path to the special file that is the socket), you use a `struct sockaddr_un`, which has the following fields:

```
struct sockaddr_un {
    unsigned short sun_family;  /* AF_UNIX */
    char sun_path[108];
}
```

This is the structure you will be passing to the **`bind()`** function, which associates a socket descriptor (a file descriptor) with a certain file (the name for which is in the *`sun_path`* field).

11.2. What to do to be a Server

Without going into too much detail, I'll outline the steps a server program usually has to go through to do it's thing. While I'm at it, I'll be trying to implement an "echo server" which just echos back everything it gets on the socket.

Here are the server steps:

1. **Call `socket()`:** A call to **`socket()`** with the proper arguments creates the Unix socket:

   ```
   unsigned int s, s2;
   struct sockaddr_un local, remote;
   int len;

   s = socket(AF_UNIX, SOCK_STREAM, 0);
   ```

   The second argument, `SOCK_STREAM`, tells **`socket()`** to create a stream socket. Yes, datagram sockets (`SOCK_DGRAM`) are supported in the Unix domain, but I'm only going to cover stream sockets here. For the curious, see Beej's Guide to Network Programming for a good description of unconnected datagram sockets that applies perfectly well to Unix sockets. The only thing that changes is that you're now using a `struct sockaddr_`**`un`** instead of a `struct sockaddr_`**`in`**.

   One more note: all these calls return `-1` on error and set the global variable *`errno`* to reflect whatever went wrong. Be sure to do your error checking.

2. **Call `bind()`:** You got a socket descriptor from the call to **`socket()`**, now you want to bind that to an address in the Unix domain. (That address, as I said before, is a special file on disk.)

   ```
   local.sun_family = AF_UNIX;  /* local is declared before socket() ^ */
   strcpy(local.sun_path, "/home/beej/mysocket");
   unlink(local.sun_path);
   len = strlen(local.sun_path) + sizeof(local.sun_family);
   bind(s, (struct sockaddr *)&local, len);
   ```

This associates the socket descriptor "*s*" with the Unix socket address "`/home/beej/mysocket`". Notice that we called **`unlink()`** before **`bind()`** to remove the socket if it already exists. You will get an `EINVAL` error if the file is already there.

3. **Call `listen()`:** This instructs the socket to listen for incoming connections from client programs:

```
listen(s, 5);
```

   The second argument, `5`, is the number of incoming connections that can be queued before you call **`accept()`**, below. If there are this many connections waiting to be accepted, additional clients will generate the error `ECONNREFUSED`.

4. **Call `accept()`:** This will accept a connection from a client. This function returns *another socket descriptor*! The old descriptor is still listening for new connections, but this new one is connected to the client:

```
len = sizeof(struct sockaddr_un);
s2 = accept(s, &remote, &len);
```

   When **`accept()`** returns, the *`remote`* variable will be filled with the remote side's `struct sockaddr_un`, and *`len`* will be set to its length. The descriptor *`s2`* is connected to the client, and is ready for **`send()`** and **`recv()`**, as described in the Network Programming Guide.

5. **Handle the connection and loop back to `accept()`:** Usually you'll want to communicate to the client here (we'll just echo back everything it sends us), close the connection, then **`accept()`** a new one.

```
while (len = recv(s2, &buf, 100, 0), len > 0)
    send(s2, &buf, len, 0);

/* loop back to accept() from here */
```

6. **Close the connection:** You can close the connection either by calling `close()`, or by calling **`shutdown()`**.

With all that said, here is some source for an echoing server, *`echos.c`*. All it does is wait for a connection on a Unix socket (named, in this case, "echo_socket").

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

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, s2, t, len;
    struct sockaddr_un local, remote;
```

```
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    local.sun_family = AF_UNIX;
    strcpy(local.sun_path, SOCK_PATH);
    unlink(local.sun_path);
    len = strlen(local.sun_path) + sizeof(local.sun_family);
    if (bind(s, (struct sockaddr *)&local, len) == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(s, 5) == -1) {
        perror("listen");
        exit(1);
    }

    for(;;) {
        int done, n;
        printf("Waiting for a connection...\n");
        t = sizeof(remote);
        if ((s2 = accept(s, (struct sockaddr *)&remote, &t)) == -1) {
            perror("accept");
            exit(1);
        }

        printf("Connected.\n");

        done = 0;
        do {
            n = recv(s2, str, 100, 0);
            if (n <= 0) {
                if (n < 0) perror("recv");
                done = 1;
            }

            if (!done)
                if (send(s2, str, n, 0) < 0) {
                    perror("send");
                    done = 1;
                }
        } while (!done);

        close(s2);
    }

    return 0;
}
```

As you can see, all the aforementioned steps are included in this program: call **`socket()`**, call **`bind()`**, call **`listen()`**, call **`accept()`**, and do some network **`send()`**s and **`recv()`**s.

11.3. What to do to be a client

There needs to be a program to talk to the above server, right? Except with the client, it's a lot easier because you don't have to do any pesky **`listen()`**ing or **`accept()`**ing. Here are the steps:

1. Call **`socket()`** to get a Unix domain socket to communicate through.
2. Set up a `struct sockaddr_un` with the remote address (where the server is listening) and call **`connect()`** with that as an argument
3. Assuming no errors, you're connected to the remote side! Use **`send()`** and **`recv()`** to your heart's content!

How about code to talk to the echo server, above? No sweat, friends, here is *`echoc.c`*:

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

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, t, len;
    struct sockaddr_un remote;
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    printf("Trying to connect...\n");

    remote.sun_family = AF_UNIX;
    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.sun_family);
    if (connect(s, (struct sockaddr *)&remote, len) == -1) {
        perror("connect");
        exit(1);
    }

    printf("Connected.\n");

    while(printf("> "), fgets(str, 100, stdin), !feof(stdin)) {
        if (send(s, str, strlen(str), 0) == -1) {
            perror("send");
            exit(1);
        }

        if ((t=recv(s, str, 100, 0)) > 0) {
            str[t] = '\0';
            printf("echo> %s", str);
        } else {
            if (t < 0) perror("recv");
            else printf("Server closed connection\n");
            exit(1);
        }
    }

    close(s);

    return 0;
}
```

In the client code, of course you'll notice that there are only a few system calls used to set

things up: **`socket()`** and **`connect()`**. Since the client isn't going to be **`accept()`**ing any incoming connections, there's no need for it to **`listen()`**. Of course, the client still uses **`send()`** and **`recv()`** for transferring data. That about sums it up.

11.4. `socketpair()`—quick full-duplex pipes

What if you wanted a **`pipe()`**, but you wanted to use a single pipe to send and recieve data from *both sides*? Since pipes are unidirectional (with exceptions in SYSV), you can't do it! There is a solution, though: use a Unix domain socket, since they can handle bi-directional data.

What a pain, though! Setting up all that code with **`listen()`** and **`connect()`** and all that just to pass data both ways! But guess what! You don't have to!

That's right, there's a beauty of a system call known as **`socketpair()`** this is nice enough to return to you a pair of *already connected sockets*! No extra work is needed on your part; you can immediately use these socket descriptors for interprocess communication.

For instance, lets set up two processes. The first sends a `char` to the second, and the second changes the character to uppercase and returns it. Here is some simple code to do just that, called *`spair.c`* (with no error checking for clarity):

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

int main(void)
{
    int sv[2]; /* the pair of socket descriptors */
    char buf; /* for data exchange between processes */

    if (socketpair(AF_UNIX, SOCK_STREAM, 0, sv) == -1) {
        perror("socketpair");
        exit(1);
    }

    if (!fork()) {  /* child */
        read(sv[1], &buf, 1);
        printf("child: read '%c'\n", buf);
        buf = toupper(buf);  /* make it uppercase */
        write(sv[1], &buf, 1);
        printf("child: sent '%c'\n", buf);

    } else { /* parent */
        write(sv[0], "b", 1);
        printf("parent: sent 'b'\n");
        read(sv[0], &buf, 1);
        printf("parent: read '%c'\n", buf);
        wait(NULL); /* wait for child to die */
    }

    return 0;
}
```

Sure, it's an expensive way to change a character to uppercase, but it's the fact that you have simple communication going on here that really matters.

One more thing to notice is that **`socketpair()`** takes both a domain (`AF_UNIX`) and socket type (`SOCK_STREAM`). These can be any legal values at all, depending on which routines in the kernel you want to handle your code, and whether you want stream or datagram sockets. I chose `AF_UNIX` sockets because this is a Unix sockets document and they're a bit faster than `AF_INET` sockets, I hear.

Finally, you might be curious as to why I'm using **`write()`** and **`read()`** instead of **`send()`** and **`recv()`**. Well, in short, I was being lazy. See, by using these system calls, I don't have to enter the *`flags`* argument that **`send()`** and **`recv()`** use, and I always set it to zero anyway. Of course, socket descriptors are just file descriptors like any other, so they respond just fine to many file manipulation system calls.

12. More IPC Resources

12.1. Books

Here are some books that describe some of the procedures I've discussed in this guide, as well as Unix details in specific:

Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 0131411551, 0130810819.

Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 0201433079.

Bach, Maurice J. *The Design of the UNIX Operating System*. New Jersey: Prentice-Hall, 1986. ISBN 0132017997.

12.2. Other online documentation

UNIX Network Programming Volume 2 home page—includes source code from Stevens' superfine book

The Linux Programmer's Guide—in-depth section on IPC

UNIX System Calls and Subroutines using C—contains modest IPC information

The Linux Kernel—how the Linux kernel implements IPC

12.3. Linux man pages

There are Linux manual pages. If you run another flavor of Unix, please look at your own man pages, as these might not work on your system.

accept(), **bind()**, **connect()**, **dup()**, **exec()**, **exit()**, **fcntl()**, **fileno()**, **fork()**, **ftok()**, **getpagesize()**, **ipcrm**, **ipcs**, **kill**, **kill()**, **listen()**, **lockf()**, **lseek()** (for the *l_whence* field in `struct flock`), **mknod**, **mknod()**, **mmap()**, **msgctl()**, **msgget()**, **msgsnd()**, **munmap()**, **open()**, **pipe()**, **ps**, **raise()**, **read()**, **recv()**, **semctl()**, **semget()**, **semop()**, **send()**, **shmat()**, **shmctl()**, **shmdt()**, **shmget()**, **sigaction()**, **signal()**, signals, **sigpending()**, **sigprocmask()**, sigsetops, **sigsuspend()**, **socket()**, **socketpair()**, **stat()**, **wait()**, **waitpid()**, **write()**.