System Calls

In the C lecture we discussed the concept of separate compilation and C "libraries." Functions like *printf()*, *strlen()* and *strncpy()* are all implemented as libraries for Linux. That is, someone, some time wrote the following code (or something like it):

```
int strlen(char *string)
{
        int i;
        i = 0;
        while(string[i] != 0) {
                i++;
        }

        return(i);
}
```

That's pretty much all *strlen()* does. As mentioned, this function is included as part of a default library called *libc.a* that is automatically linked with any C program (by default).

There are some functions that your program also gets that are not implemented as libraries. Instead, these functions (which all must manipulate data structures that are shared among all processes in the system) are implemented **directly by the operating system**. That is, they act like they are loaded as part of a default library, but in fact they are part of the resident OS. These kinds of functions are called **system calls** because they are calls to the system (the OS).

Knowing whether a Linux feature is implemented as a library or system call isn't exactly easy. As a rule, the section section of the manual (section 2) are the basic system calls. If you type

```
man 2 read
```

you will see the system call *read()* listed, but type

```
man 3 read
```

and it won't appear. However,

```
man 3 fwrite
```

shows the man page for the *fwrite()* library call.

This isn't a very satisfying or accurate way to determine whether a call is a default library call or a system call but it will get you started. Moreover, library calls like *printf()* may ultimately call system calls like *write()*. Finally, C was designed before dynamic linking of libraries was commonplace. As a result, before a program is fully loaded, the compiler may not be able to tell whether a call outside the program's address space is to a library or a system call.

Mostly you just need to read the developer documentation to determine what are the system calls that are available in any specific version of Linux (because they change from version to version). We'll discuss a few system calls that are almost assured implemented in any version of Linux as system calls and not libraries.

Getting the process identifier

All processes in Linux get a unique identifier of type *pid_t*. Printing the value of a *pid_t* turns out to a matter of some debate. Seriously. I'll go fast and loose on these examples but someone may give you grief down the road for a specific Linux or Unix version.

The system call to get the process identifier under Linux is *getpid()*. If you type

```
man getpid
```

you will see that you need to include *sys/types.h* to get the type specifier and *unistd.h* to get the right prototype for the compiler. It takes no arguments and returns a *pid_t*. Try it out using sys-call1.c:

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

/*
 * call getpid() as an example system call
 */

int main(int argc, char **argv)
{
        pid_t my_id;

        my_id = getpid();
        printf("my process id is %ld\n",(long)my_id);

         return(0);
}
```

You'll just have to trust me that *getpid()* is implemented as a system call. Only the OS can assign your process a process identifier so to get the identifier, your process must make a call to the OS to get it.

Files

One of the key abstractions that Linux provides is the file system. We'll talk in depth about how the file system works later in the class. However, from a C perspective it is important that you know how to use files (in their most basic form).

There are five system calls associated with files that you should understand as part of basic C and Linux:

- open: opens a file specified by the first argument with the "mode" specified in the second argument, and the permissions specified in the third argument. Returns an integer *>= 0* when successful and negative when not.

- close: closes the file which both tells the OS that your process will no longer access the file (useful for managing internal OS state) and also prevents the file from being accessed further in the process due to a programming error.
- read: reads a specified number of bytes (third argument) from a file (first argument) into a buffer (second argument) at "the current offset."
- write: writes a specified number of bytes (third argument), from a buffer (second argument) into a file (first argument) "at the current offset."
- lseek: changes "the current offset."

Either this sounds straight forward or it doesn't. In reality, like most things with Linux, it is neither.

First, you are probably familiar with the notion of a file being visible in the file system when you run the Linux utility *ls*. Further, that file is uniquely specified by a path name -- a string of directories, separated by the "/" character, indicating the path through a directory tree from the root to the file. Turns out that only *open()* takes a path to the file. Once the file is opened, it must be referred to in your program by its **file descriptor** which is an integer assigned by the OS when the file is opened. The exact usage will be come clear, but it is important to understand that when you want to do anything other than open a file, you don't specify the path name, but instead you specify the file descriptor number that came back from the open on the file.

Secondly, the OS keeps a "current offset" from the beginning of the file for you while the file is open on a specific file descriptor. When the file is first opened, the offset is zero. Subsequent calls to *read()* and *write()* on the file descriptor cause the offset to advance. To make the offset "back up" you either need to close the file, re-open it, and then advance the offset with read calls (which are non-destructive) or use the *lseek()* call to set the offset to a specific number.

Let's start by writing a program that takes a file name as an argument, creates the file, and writes an important and pitch message into it as a string. Consider the program code available in file-create1.c:

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

/*
 * open to create a file and write a string into it
 */
int main(int argc, char **argv)
{

        int my_file_desc;

        char file_name[4096];
        char write_buffer[4096];
        int write_length;
        char *string;
```

```
int i;
int w;

if(argc < 2) {
        printf("need to specify a file name as the first argument\n");
        exit(1);
}

/*
 * zero out the buffer for the file name
 */
for(i=0; i < sizeof(file_name); i++) {
        file_name[i] = 0;
}

/*
 * copy the argument into a local buffer
 */
strncpy(file_name,argv[1],sizeof(file_name));
file_name[sizeof(file_name)-1] = 0;

/*
 * try and open the file for creation
 */

my_file_desc = open(file_name,O_CREAT | O_WRONLY, 0600);

if(my_file_desc < 0) {
        printf("failed to open %s for creation\n",file_name);
        exit(1);
}

/*
 * file is open, write a string into it
 */
string =
 "This program brought to you by the council for better C programming";
for(i=0; i < sizeof(write_buffer); i++) {
        write_buffer[i] = 0;
}
strncpy(write_buffer,string,sizeof(write_buffer));

write_length = strlen(write_buffer);


w = write(my_file_desc,write_buffer,write_length);


if(w != write_length) {
        printf("write didn't complete, w: %d, length: %d\n",
                w,
                write_length);
        close(my_file_desc);
        exit(1);
}

close(my_file_desc);
```

```
        return(0);
}
```

First, try running it as

```
./file-create1 ./foo.txt
```

Then, look in the current directory for a file called *foo.txt*. What does it contain? Now try:

```
./file-create1 /foo.txt
```

This attempt should fail because you tried to create a file called "foo.txt" in the root directory and you don't have write permissions for that directory (hopefully).

Next, look at the line where *open()* is called in the code:

```
my_file_desc = open(file_name,O_CREAT | O_WRONLY, 0600);
```

This line says to open the file specified by the string contained in the array *file_name* when treating that string as a path in the file system. If you run it with *foo.txt* or *./foo.txt* it will use the current working directory as the place where you try and create *foo.txt*.

The second argument requires that you look at the man page for open

```
man 2 open
```

The second argument to open specifies how you want the file opened. In this case, I specified that I wanted the file created if it doesn't exist but simply opened if it did. I also specified that I wanted only to write the file. If I had tried to call *read* on *my_file_desc* at some point after the open, the read would fail.

Notice that if I run the program twice

```
./file-create1 foo.txt
./file-create1 foo.txt
```

I still only get one copy of the file with one string in it. That's because the *open()* sets the current offset to zero each time. When I run this program the first time, it creates *foo.txt*. Each time after that it just overwrites the file with exactly the same string.

The third argument says that I want to create the file with the permissions "600" -- or RW for owner only. See

```
man 2 chmod
```

for details on specifying permissions. It is possible to use constants for these instead of numbers but I think of them as numbers.

Finally, notice that the open returns an integer (the type of *my_file_desc* is *int*). That value is passed to *write()* to indicate which file I want to write. If the code had opened two different files, each would have a different file descriptor number. For example, as in file-create2.c:

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

/*
 * open to create a couple of files and write a strings into each
 */
int main(int argc, char **argv)
{
        int fd1;
        int fd2;
        char file_name1[4096];
        char file_name2[4096];
        char write_buffer[4096];
        int write_length;
        char *string;
        int i;
        int w;

        if(argc < 3) {
                printf("need to specify two file names as the first and second
arguments\n");
                exit(1);
        }

        /*
         * zero out the buffers for the file names
         */
        for(i=0; i < sizeof(file_name1); i++) {
                file_name1[i] = 0;
                file_name2[i] = 0;
        }

        /*
         * copy the first argument into a local buffer
         */
        strncpy(file_name1,argv[1],sizeof(file_name1));
        file_name1[sizeof(file_name1)-1] = 0;

        /*
         * and the second
         */
        strncpy(file_name2,argv[2],sizeof(file_name2));
        file_name2[sizeof(file_name2)-1] = 0;

        /*
         * try and open the first file for creation
         */

        fd1 = open(file_name1,O_CREAT | O_WRONLY, 0600);

        if(fd1 < 0) {
                printf("failed to open %s for creation\n",file_name1);
                exit(1);
        }
```

```
/*
 * and the second
 */

fd2 = open(file_name2,O_CREAT | O_WRONLY,  0600);

if(fd2 < 0) {
        printf("failed to open %s for creation\n",file_name2);
        close(fd1);
        exit(1);
}

/*
 * both files are open, write a string into first
 */
string =
  "This program brought to you by the council for better C programming";
for(i=0; i < sizeof(write_buffer); i++) {
        write_buffer[i] = 0;
}
strncpy(write_buffer,string,sizeof(write_buffer));

write_length = strlen(write_buffer);


w = write(fd1,write_buffer,write_length);


if(w != write_length) {
        printf("write didn't complete, w: %d, length: %d in %s\n",
                w,
                write_length,
                file_name1);
        close(fd1);
        close(fd2);
        exit(1);
}

/*
 * and the second
 */
string = "C programming is both nutritious and great tasting.";
for(i=0; i < sizeof(write_buffer); i++) {
        write_buffer[i] = 0;
}
strncpy(write_buffer,string,sizeof(write_buffer));

write_length = strlen(write_buffer);


w = write(fd2,write_buffer,write_length);


if(w != write_length) {
        printf("write didn't complete, w: %d, length: %d in %s\n",
                w,
```

```
                        write_length,
                        file_name2);
                close(fd1);
                close(fd2);
                exit(1);
        }

        close(fd1);
        close(fd2);

        return(0);

}
```

You should read through this example and notice that the two files are referenced with two different file descriptors.

Reading the data back

Now let's try reading the data back from a file, as in file-read1.c:

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

/*
 * open and read the contents of a file, printing them out as ascii
 * characters
 */
int main(int argc, char **argv)
{
        int my_file_desc;
        char file_name[4096];
        char read_buffer[4096];
        int i;
        int r;

        if(argc < 2) {
                printf("need to specify a file name as the first argument\n");
                exit(1);
        }

        /*
         * zero out the buffer for the file name
         */
        for(i=0; i < sizeof(file_name); i++) {
                file_name[i] = 0;
        }

        /*
         * copy the argument into a local buffer
         */
        strncpy(file_name,argv[1],sizeof(file_name));
        file_name[sizeof(file_name)-1] = 0;
```

```
        /*
         * try and open the file for reading
         */

        my_file_desc = open(file_name,O_RDONLY,0);

        if(my_file_desc < 0) {
                printf("failed to open %s for reading\n",file_name);
                exit(1);
        }

        for(i=0; i < sizeof(read_buffer); i++) {
                read_buffer[i] = 0;
        }


        r = read(my_file_desc,read_buffer,sizeof(read_buffer)-1);


        printf("file: %s contains the string: %s\n",
                file_name,
                read_buffer);

        close(my_file_desc);

        return(0);

}
```

Here the open call is a little different:

```
        my_file_desc = open(file_name,O_RDONLY,0);
```

indicating that the file is to be open for reading only. The permissions will be checked but no permissions need to be specified since the file will not be created if it it isn't present already.

Also, the read is a little different

```
        r = read(my_file_desc,read_buffer,sizeof(read_buffer)-1);
```

In the case of *read()* the system call will read bytes up to the number specified in the third argument. If there are fewer than that number of bytes, it will return the ones it read and leave the current offset at the end of the file. Another read at the end will return a zero indicating that the file is empty.

If you wanted to read the entire contents of the file, then you will need to loop until there is no more data as in file-read2.c.

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

```
/*
 * read the file in its entirety (like the cat utility)
 */
int main(int argc, char **argv)
{
        int my_file_desc;
        char file_name[4096];
        char read_buffer[4096];
        int i;
        int r;

        if(argc < 2) {
                printf("need to specify a file name as the first argument\n");
                exit(1);
        }

        /*
         * zero out the buffer for the file name
         */
        for(i=0; i < sizeof(file_name); i++) {
                file_name[i] = 0;
        }

        /*
         * copy the argument into a local buffer
         */
        strncpy(file_name,argv[1],sizeof(file_name));
        file_name[sizeof(file_name)-1] = 0;

        /*
         * try and open the file for reading
         */
        my_file_desc = open(file_name,O_RDONLY,0);
        if(my_file_desc < 0) {
                printf("failed to open %s for reading\n",file_name);
                exit(1);
        }

        for(i=0; i < sizeof(read_buffer); i++) {
                read_buffer[i] = 0;
        }

        r = read(my_file_desc,read_buffer,sizeof(read_buffer)-1);

        /*
         * read and print until EOF
         */
        while(r > 0) {
                printf("%s",read_buffer);
                for(i=0; i < sizeof(read_buffer); i++) {
                        read_buffer[i] = 0;
                }
                r = read(my_file_desc,read_buffer,sizeof(read_buffer)-1);
        }

        close(my_file_desc);
```

```
        return(0);

}
```

This is essentially what the Linux utility *cat* does although cat probably does it in a more elegant way. Try it out on a text file that is bigger than 4K bytes to assure yourself that it works correctly.

For example, try

```
./file-read2 /cs/faculty/rich/public_html/class/cs170/notes/C/index.html
```

and see if you get the text from the HTML for this web page back.

Seeking to an offset

The *lseek()* system call moves the current offset pointer by the number of bytes specified from a starting location that is taken from its third argument. For example, consider file-seek1.c.

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

/*
 * seek the file in the first argument to the location specified in the
 * second
 */
int main(int argc, char **argv)
{
        int my_file_desc;
        char file_name[4096];
        char read_buffer[4096];
        int i;
        int r;
        off_t offset;
        off_t where;

        if(argc < 3) {
                printf("need to specify a file name and an offset\n");
                exit(1);
        }

        /*
         * zero out the buffer for the file name
         */
        for(i=0; i < sizeof(file_name); i++) {
                file_name[i] = 0;
        }

        /*
         * copy the argument into a local buffer
         */
        strncpy(file_name,argv[1],sizeof(file_name));
```

```
        file_name[sizeof(file_name)-1] = 0;

        /*
         * get the offset from the second argument
         */

        offset = (off_t)atoi(argv[2]);


        /*
         * try and open the file for reading
         */
        my_file_desc = open(file_name,O_RDONLY,0);
        if(my_file_desc < 0) {
                printf("failed to open %s for reading\n",file_name);
                exit(1);
        }

        /*
         * seek to the offset specified in the second argument
         */

        where = lseek(my_file_desc,offset,SEEK_SET);


        if(where < 0) {
                printf("lseek to %d in file %s failed\n",
                        (int)offset,
                        file_name);
                close(my_file_desc);
                exit(1);
        }

        for(i=0; i < sizeof(read_buffer); i++) {
                read_buffer[i] = 0;
        }

        r = read(my_file_desc,read_buffer,sizeof(read_buffer)-1);

        /*
         * read and print until EOF
         */
        while(r > 0) {
                printf("%s",read_buffer);
                for(i=0; i < sizeof(read_buffer); i++) {
                        read_buffer[i] = 0;
                }
                r = read(my_file_desc,read_buffer,sizeof(read_buffer)-1);
        }

        close(my_file_desc);

        return(0);

}
```

The line

```
where = lseek(my_file_desc,offset,SEEK_SET);
```

after the call to *open()* but before the first call to *read()* moves the current offset to the byte offset specified in the second argument. The *SEEK_SET* parameter says to "set" the offset to the value specified (as opposed to setting it relative to the current offset). To see other options for this parameter, check out the man page.

Now try running the following:

```
./file-create1 foo.txt
./file-seek1 foo.txt 10
```

You should see something like

```
am brought to you by the council for better C programming
```

where the first 10 bytes of the line are missing from the line in the file *foo.txt*. Does this make sense as the output?

Standard In, Standard Out and Standard Error

Take a look at the code in file-fd1.c.

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

/*
 * print out some file descriptors
 */
int main(int argc, char **argv)
{
        int my_file_desc;
        char file_name[4096];
        int i;
        int r;
        char *string;

        if(argc < 2) {
                printf("need to specify a file name\n");
                exit(1);
        }

        /*
         * zero out the buffer for the file name
         */
        for(i=0; i < sizeof(file_name); i++) {
                file_name[i] = 0;
        }

        /*
         * copy the argument into a local buffer
         */
```

```
        strncpy(file_name,argv[1],sizeof(file_name));
        file_name[sizeof(file_name)-1] = 0;

        /*
         * try and open the file for reading
         */

        my_file_desc = open(file_name,O_RDONLY,0);

        if(my_file_desc < 0) {
                printf("failed to open %s for reading\n",file_name);
                exit(1);
        }

        printf("my_file_desc: %d\n",my_file_desc);

        string = "a string written to standard out\n";

        write(1,string,strlen(string));


        close(my_file_desc);

        return(0);

}
```

It should look a little alarming to you especially with respect to the *write()* call. Try running it after creating the file *foo.txt*:

```
./file-create1 foo.txt
./file-fd1 foo.txt
my_file_desc: 3
a string written to standard out
```

What happened here? First, notice that the value of file descriptor returned by the open call and stored in the integer variable *my_file_desc* is *3*. Turns out that this number is not just a random integer selected by Linux. Next, and perhaps most curiously, in the code the call to *write()* writes a string out to file descriptor that wasn't opened.

```
        string = "a string written to standard out\n";
        write(1,string,strlen(string));
```

In fact, it is just writing to file descriptor *1* without any other reference to *1* as a file descriptor.

Linux automatically "opens" three file descriptors -- *0, 1, and 2* for you when a process is launched and connects these file descriptors to the keyboard and the terminal. This feature explains the first line of output in that the smallest file descriptor that can be returned from a call to *open()* is *3* since file descriptors *0, 1, and 2* are already opened. Linux could have chosen any other integer when it chose a file descriptor for the call to *open()* in the code. For various historical reasons it will choose the smallest unused file descriptor. When the program launches, that file descriptor number will be *3*. If another call to open were made, that file descriptor would

be *4*. If *close()* is called on *my_file_desc* and then open is called again, *3* will be chosen since at that moment it would be the smallest unused file descriptor.

The three "special" file descriptors Linux opens for you at process launch are

- File descriptor *0*: connected to the keyboard and is referred to as **standard in**.
- File descriptor *1*: connected to the terminal and is referred to as **standard out**
- File descriptor *2*: also connected to the terminal and is referred to as **standard error**

The last two are separated so that the shell can send errors to the terminal even if you redirect the standard output to a file.

Thus, the call to write tells the OS to write from the string buffer the *strlen()* of the string to the standard out device which (unless you closed it) will be the terminal. Notice that the call to *write()* doesn't know whether the destination is standard out or a file -- it just writes the data to the file that the OS has associated with the file descriptor and that "file" has cleverly been impersonated by the terminal.

To see the difference between **standard out** and **standard error** try running the code in file-fd2.c

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

/*
 * print out some file descriptors
 */
int main(int argc, char **argv)
{
        char *string;
        char *string_err;

        string = "a string written to standard out\n";

        write(1,string,strlen(string));


        string_err = "a string written to standard error\n";

        write(2,string_err,strlen(string_err));


        return(0);

}
```

First, run it from the terminal and then run it, but redirect (using the shell) the output to a file:

```
./file-fd2
a string written to standard out
a string written to standard error

./file-fd2 > fd2.out
a string written to standard error
cat fd2.out
a string written to standard out
```

In the first execution, both standard out and standard error were sent to the terminal. In the second, standard out the shell opened a file and sent the output on standard out to the file. The > operator in the shell implements this file writing function. However standard error was not redirected so the string written on file descriptor *2* was sent to the terminal.

Standard in works the same way, but it needs to know when there is no more input from the keyboard. When you tun the code in file-fd3.c

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

/*
 * read from stdin and write to stdout
 */
int main(int argc, char **argv)
{
        char buffer[4096];

        memset(buffer,0,sizeof(buffer));

        read(0,buffer,sizeof(buffer));
        write(1,buffer,strlen(buffer));

        return(0);

}
```

Notice that this codes uses the function *memset()* to zero out the buffer. It reads up tot he buffer's size from standard in (the keyboard) and then writes it to the terminal. However, when you run the program, you need to type the character *ctrl-D* before the program will finish. Otherwise, it waits, blocked in the read call, waiting for another character.

I ran it, types the characters in *my dog is happy!* and then *ctrl-D*.

```
./file-fd3
my dog is happy!my dog is happy!
```

The *ctrl-D* character tells the shell that the end of file has been reached which, for standard in means that no more characters will be coming from the keyboard. The program then writes the wring back out to standard out.

The shell can also redirect data from a file to standard in. Using the file from the previous example, try

```
./file-fd3 < fd2.out
```

which causes the contents fo the file *fd2.out* to be sent to standard in of *file-fd3* until the end-of-file is reached. The read then terminates the data is written to standard out.

Fork/Exec/Wait

When you log into a Linux system and you get a prompt, the program that is running is called "the shell." It is usually a BASH shell (see the man page) and its job is to create processes on your behalf. Some of those processes then run Linux utilities for you. For example, when you type

```
ls ~
```

you are telling the shell to run a program called *ls* (which is located in the /usr/bin directory) and to pass it, as its first argument, the path to your home directory. The shell expands the ~ character into a path, but *ls* is a C program that someone wrote for your Linux system. It was compiled when your Linux was built and it is placed in /usr/bin when Linux is installed. The shell looks in a few places (defined by the PATH environment variable) for programs to run when you simply give their names, like I have in this example with *ls*.

But how does the shell (or any other C program since the shell, itself, is just a program written in C and compfile for Linux) create a process and run another program that has been compiled?

Fork

The way one program runs another in Linux is a little odd. First, it makes an exact copy of itself using the *fork()* system call. When *fork()* completes, there are two copies of the same program. Then, the second copy calls *exec()* which loads the binary for the second program over itself and starts it from the beginning. I know. It gets easier once you see it.

Take a look at the code in fork-1.c:

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

/*
 * simple program calling fork
 */
int main(int argc, char **argv)
{
        pid_t child_id;
        pid_t my_id;

        my_id = getpid();
        printf("pid: %d -- I am the parent about to call fork\n",
                        (int)my_id);


        child_id = fork();
        if(child_id != 0) {
```

```
                my_id = getpid();
                printf("pid: %d -- I just forked a child with id %d\n",
                        (int)my_id,
                        (int)child_id);
        } else {
                my_id = getpid();
                printf("pid: %d -- I am the child\n",my_id);
        }


        printf("pid: %d -- I am exiting\n",my_id);
        exit(0);

}
```

The call to *fork()* creates an **exact copy of the process calling fork except for the process identifier** which is different since it is a new process. The interesting part is that this new process (typically called the *child* process) begins running at the instruction **immediately after the fork() call in the program**. That is, after the call to *fork()* completes, there are two processes running and they both execute the next line which is

```
if(child_id != 0) {
```

which is pretty typical. The call to *fork()* returns the process identifier to of the newly created process to the parent, but returns zero to the child. That way the code can have the child and parent diverge and execute different code paths. In this example, the parent and child print different messages after the child has been forked. The output I get is

```
./fork-1
pid: 40285 -- I am the parent about to call fork
pid: 40285 -- I just forked a child with id 40286
pid: 40285 -- I am exiting
pid: 40286 -- I am the child
pid: 40286 -- I am exiting
```

On each line, the code prints the process identifier of the process calling *printf()*. Thus the parent or origibal process is process *40285*. It creates a child process and Linux chooses *40286* for that process' identifier which is returned from the call to *fork()* to the parent. The parent then prints a message indicating that it has created the child and the child prints a message with its identifier. Both then print before they exit.

The parent can wait for the child to complete using the *wait()* system call. Take a look at fork-2.c:

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

/*
 * simple program calling fork and wait
 */
int main(int argc, char **argv)
{
        pid_t child_id;
        pid_t my_id;
```

```
        int child_status;
        int i;

        child_id = fork();
        if(child_id != 0) {
                my_id = getpid();
                printf("pid: %d -- I just forked a child with id %d\n",
                        (int)my_id,
                        (int)child_id);
                printf("pid: %d -- I am waiting for process %d to finish\n",
                        (int)my_id,
                        (int)child_id);

                wait(&child_status);

                printf("pid: %d -- my child has completed with status: %d\n",
                        (int)my_id,
                        child_status);
        } else {
                my_id = getpid();
                printf("pid: %d -- I am the child and I am count to 10\n",
                        (int)my_id);
                for(i=0; i < 10; i++) {
                        printf("pid: %d -- %d\n",my_id,i+1);
                }
        }

        printf("pid: %d -- I am exiting\n",my_id);
        exit(0);

}
```

Running this version yields:

```
./fork-2
pid: 74869 -- I just forked a child with id 74870
pid: 74869 -- I am waiting for process 74870 to finish
pid: 74870 -- I am the child and I am going to count to 10
pid: 74870 -- 1
pid: 74870 -- 2
pid: 74870 -- 3
pid: 74870 -- 4
pid: 74870 -- 5
pid: 74870 -- 6
pid: 74870 -- 7
pid: 74870 -- 8
pid: 74870 -- 9
pid: 74870 -- 10
pid: 74870 -- I am exiting
pid: 74869 -- my child has completed with status: 0
pid: 74869 -- I am exiting
```

Notice the process identifiers. The parent waits until the child exits and then continues after it has done so.

Notice also that the call to *wait()* takes a pointer to an integer as a parameter. The *wait()* call uses this parameter as an out parameter to deliver an integer to the parent indicating the child's exit status.

The exit status in the child is given in the argument to *exit()* which, in this example, is *0*. Most Linux utilities return an exit status of *0* when the complete successfully and a small positive integer (usually *1* or *2*) when an error has occured. If the call to exit had been

```
exit(1);
```

the parent would see the value of the integer variable *child_status* set to *1*. You should probably adopt this convention with respect to exit status (at least, as far as returning a *0* when everything went okay) because many if not most shell scripts use the exit status to determine if an error in some utility has occured.

Exec

The *fork()* clones the running process, but often what you need is to execute (as its own process) a program that is located in a file in the file system.

While the function of *fork()* has remained relatively constant over the years, *exec()* has undergone a few changes since it was first defined. The version we will discuss is *execve()* since its definition is rather clear. To be complete, you should look at the man pages for

```
man 2 execve
man 2 execl
```

The latter call will decsribe different variants of *exec()* each of which has its own peculiarities. For example, when you give the filename of the program you wish to have loaded by *exec()* you may want the search path to be the same search path as the shell uses when it ran the program that is calling *exec()*. See the second man entry for details.

However, we will constrain our remarks to *execve()* for the sakes both of simplicity and brevity.

Consider the code contained in fork-3.c:

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

/*
 * run a program using fork and execve
 */
int main(int argc, char **argv, char **envp)
{
        pid_t child_id;
        pid_t my_id;
        int child_status;
        int i;
        char file_name[4096];
        int err;
```

```
        if(argc < 2) {
                printf("must specify file name as first argument\n");
                exit(1);
        }

        memset(file_name,0,sizeof(file_name));
        strncpy(file_name,argv[1],sizeof(file_name));

        child_id = fork();
        if(child_id != 0) {
                my_id = getpid();
                printf("pid: %d -- I forked pid: %d for: %s\n",
                        my_id,
                        child_id,
                        file_name);
                wait(&child_status);
                printf("pid: %d -- %s has completed with status: %d\n",
                        (int)my_id,
                        file_name,
                        child_status);
        } else {
                my_id = getpid();
                printf("pid: %d -- I am the child and I am going to exec %s\n",
                        (int)my_id,
                        file_name);

                err = execve(file_name,&(argv[1]),envp);
                /*
                 * not reached if execve is successful
                 */
                printf("pid: %d -- execve of %s failed with error %d\n",

                        (int)my_id,
                        file_name,
                        err);

        }

        printf("pid: %d -- I am exiting\n",my_id);
        exit(0);

}
```

You should read this code carefully as it contains a few subtleties. First, notice that it forks a child and then the child calls *execve()* to run a program. The string *file_name* is used to tell *execve()* what program to run. Also, the parent program passes *&(argv[1])* as the second argument to *execve()* and the environment pointer for the main program. Finally, notice that the error message is printed immediately after the call to *execve()* without testing to see if the value is non-zero.

Let's take these observations one at a time.

The first argument to *execve()* really needs to be a path to the file in the file system that contains an executable program. Thus, it will need to contain something like */bin/hostname* or *./fork-1* -- a complete path name to the binary.

Next, *execve()* takes a list of arguments that it will pass (as the *char *argv[]* parameter) to the *main()* function of the program that is being run. In this example, the first argument passed to *fork-3* will be the path name to the binary to execute and the remaining arguments that need to be pass to it to do its job.

So, for example, to get *fork-3* to run */bin/ls* on the */tmp* directory you would execute

```
./fork-3 /bin/ls /tmp
```

and */tmp* will need to be passed to */bin/ls* through the call to *execve()*. Further, like argv in the parent, the *argv[0]* passed to the program that is being run must be the path name of the binary. Think about this for a minute. It will make sense after a bit.

Try a few examples:

```
./fork-3 /bin/date
pid: 75976 -- I forked pid: 75977 for: /bin/date
pid: 75977 -- I am the child and I am going to exec /bin/date
Wed Jan 14 15:01:23 PST 2015
pid: 75976 -- /bin/date has completed with status: 0
pid: 75976 -- I am exiting
```

Take a look at the process identifiers. Notice that the parent forks the child and waits, the child calls *execve()* and when */bin/date* has completed, the parent unblocks from its call to *wait()* getting back the exit status generated by */bin/date*.

Notice also that the message immediately after the call to *execve()* in the child is **not** printed. That's because if the call to *execve()* is successful, the child is **completely overwritten** by the new code and, thus, stops executing its own code. If the call to *execve()* fails, however, then the child continues to execute. For example,

```
./fork-3 date
pid: 76043 -- I forked pid: 76044 for: date
pid: 76044 -- I am the child and I am going to exec date
pid: 76044 -- execve of date failed with error -1
pid: 76044 -- I am exiting
pid: 76043 -- date has completed with status: 0
pid: 76043 -- I am exiting
```

Here, the path name to */bin/date* is not fully specified. The system call *execve()* does not use the *$PATH* environment variable to determine what to run. The simply specifying *date* causes *execve()* to fail because it can't find *date* in the file system. Try it with */usr/date*

```
./fork-3 /usr/date
pid: 76096 -- I forked pid: 76097 for: /usr/date
pid: 76097 -- I am the child and I am going to exec /usr/date
pid: 76097 -- execve of /usr/date failed with error -1
```

```
pid: 76097 -- I am exiting
pid: 76096 -- /usr/date has completed with status: 0
pid: 76096 -- I am exiting
```

and it fails the same way because there is no file called *date* in */usr*.

Now try it with */bin/ls*

```
./fork-3 /bin/ls -al fork-3.c
pid: 76113 -- I forked pid: 76114 for: /bin/ls
pid: 76114 -- I am the child and I am going to exec /bin/ls
-rw-r--r--  1 rich  staff  1118 Jan 14 14:52 fork-3.c
pid: 76113 -- /bin/ls has completed with status: 0
pid: 76113 -- I am exiting
```

Compare that output to

```
/bin/ls -al fork-3.c
-rw-r--r--  1 rich  staff  1118 Jan 14 14:52 fork-3.c
```

Nifty. In fact, *fork-3* is doing what */bin/bash* does when you tell the shell to run */bin/ls* -- it is more or less the same logic.

Oh-oh. Buckle up.

```
MossPiglet% ./fork-3 /bin/bash
pid: 76193 -- I forked pid: 76194 for: /bin/bash
pid: 76194 -- I am the child and I am going to exec /bin/bash
MossPiglet%
```

What happened here? It started running bash and bash is blocked waiting for my input. Notice that the parent is still waiting for the child to exit. The child called *execve()* on */bin/bash* and bash ran and is waiting for my input. If I type *exit* or *ctrl-D* that bash will exit, the parent will wake up, and *fork-3* will exit.

```
MossPiglet% ./fork-3 /bin/bash
pid: 76241 -- I forked pid: 76242 for: /bin/bash
pid: 76242 -- I am the child and I am going to exec /bin/bash
MossPiglet% exit
exit
pid: 76241 -- /bin/bash has completed with status: 0
pid: 76241 -- I am exiting
```

Why did this work? Keep in mind that the child overwrote itself with the code from the executable binary located in the file */bin/bash* but it **inherits everything else from the parent**. In particular, it gets the same open file descriptors and they remain open. What open file descriptors does bash need? Standard in, standard out, and standard error. Thus, bash gets the same standard in, standard out, and standard error that *fork-3* had and, thus, it works just fine as its own shell. Only when you call *exit* the *fork-3* process is waiting and it wakes up to get the exit status. Try it with a different status:

```
./fork-3 /bin/bash
pid: 76401 -- I forked pid: 76402 for: /bin/bash
```

```
pid: 76402 -- I am the child and I am going to exec /bin/bash
MossPiglet% exit 1
exit
pid: 76401 -- /bin/bash has completed with status: 256
pid: 76401 -- I am exiting
```

Whoops. Why did *256* come back as a status instead of *1*?

Turns out *wait()*encodes the exit status in the status integer. The correct way to manage it is depcited in fork-4.c:

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

#include < sys/wait.h >


/*
 * run a program using fork and execve
 * pring the status correctly
 */
int main(int argc, char **argv, char **envp)
{
        pid_t child_id;
        pid_t my_id;
        int child_status;
        int i;
        char file_name[4096];
        int err;

        if(argc < 2) {
                printf("must specify file name as first argument\n");
                exit(1);
        }

        memset(file_name,0,sizeof(file_name));
        strncpy(file_name,argv[1],sizeof(file_name));

        child_id = fork();
        if(child_id != 0) {
                my_id = getpid();
                printf("pid: %d -- I forked pid: %d for: %s\n",
                        my_id,
                        child_id,
                        file_name);
                wait(&child_status);

                if(WIFEXITED(child_status)) {
                        printf("pid: %d -- %s has completed with status: %d\n",
                                (int)my_id,
                                file_name,
                                WEXITSTATUS(child_status));
                }
```

```
        } else {
                my_id = getpid();
                printf("pid: %d -- I am the child and I am going to exec %s\n",
                        (int)my_id,
                        file_name);
                err = execve(file_name,&(argv[1]),envp);
                /*
                 * not reached if execve is successful
                 */
                printf("pid: %d -- execve of %s failed with error %d\n",
                        (int)my_id,
                        file_name,
                        err);

        }

        printf("pid: %d -- I am exiting\n",my_id);
        exit(0);

}
```

Turns out that the parent will wake up on conditions other than the child exiting (like the child gets a signal). You can test to see why the child woke up with and if it exited gets its one byte of status using the *WIFEXITED()* and *WEXITSTATUS()* macros respectively. They com in the header file *sys/wait.h*. Consult the man pages for further details.

Night of the Living Dead

This exit status processing can cause a bit of an issue. In fact, if the parent doesn't call *wait()* and the child exits, Linux creates a **zombie process** which waits around for the parent **only so that it can report its exit status**. If the parent never calls *wait()*, the zombie never truly dies. It can't do anything else, but Linux won't clean up its process state until the parent calls *wait()*.

Or until the parent dies. When the parent of a child dies, the child is *usually* **adopted** by a special Linux process called *init*. Every running Linux system has an *init* process. After the system boots, the job of *init* is just to call *wait()* so that orphaned children can report their exit status and avoid becoming zombies, or cease to be zombies.

For example, a parent that creates a child and then goes into an infinite loop before calling *wait()* creates a zombie when that child exits. If you kill the parent with a *ctrl-C*, the zombie child will immediately be adopted by *init* which is calling *wait()*. The exit status will be reported to *init* (which discards it) and the zombie child finally dies completely. The *init* process goes back to waiting for other orphaned children.

Seriously.

Like with many of the original Unix semantics, this behavior (which dates to early versions of Unix) can be modified in the current implementation of Linux. The default behavior is for *init* to adopt all parentless children. However using the **prctl()** system call with the **PR_SET_CHILD_SUBREAPER** argument you can set a process to be available as an adopter of parentless children. When called all descendents of the process will inherit this process as their "reaper" should a parent die. In the event of a parent termination, the nearest

anscestor in the process tree that is a subreaper for the descendents becomes the reaper. However if no subreapers are in the process tree, then *init* defaults to being th reaper.

Um. Yeah.

Pipes

Pipes are a way for two processes to communicate. When you create a pipe, you give the *pipe()* system call an array of two integers that it uses as an out parameter. It allocates two file descriptors (one for reading and the other for writing). Any data written to the write file descriptor will be available for reading on the read file descriptor. If the read and and write ends are in separate processes, the transfers the data from the writer process to the reader process.

Because they are represented using file descriptors, the same *read()* and *write()* calls that work for files also work for pipes. You can't call *lseek()* on them and you used the *pipe()* call to open then instead of calling *open()* but once they are set up, they behave like files from a usage perspective.

In pipe-1.c, the code opens a pipe, writes a string to the "write end", and then reads the "read end" to get what ever is in the pipe (up to the size of the read buffer).

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

/*
 * simple program to create and use a pipe
 */
int main(int argc, char **argv)
{
        int pipe_desc[2];
        int err;
        char *string;
        char read_buffer[4096];


        err = pipe(pipe_desc);

        if(err < 0) {
                printf("error creating pipe\n");
                exit(1);
        }

        string = "a string";

        printf("writing %s to pipe_desc[1] which is %d\n",
                        string,pipe_desc[1]);

        write(pipe_desc[1],string,strlen(string));

        memset(read_buffer,0,sizeof(read_buffer));
```

```
        printf("attempting to read pipe_desc[0] which is %d\n",pipe_desc[0]);
        read(pipe_desc[0], read_buffer, sizeof(read_buffer));

        printf("read %s from pipe_desc[0]\n",read_buffer);

        close(pipe_desc[0]);
        close(pipe_desc[1]);

        return(0);

}
```

The code simply opens a pipe, writes a string into it using the *write()* system call and then reads the string back from the pipe.

There are a couple of subtlties here, though. First, while the write end "knows" how much data it will write, the read end doesn't. In this example, I made the read buffer 4096 bytes in length. If the writer had written more than 4096 bytes, the reader would have only read the first 4096 **assuming that the pipe is big enough to hold 4096 bytes**. The semantics are that the pipe has some capacity (which is not known) and if the writing program exceeds that capacity it blocks under a reader has drained the pipe there by freeing some capacity.

In other words, a pipe is the Linux system call implementation of the bounded buffer problem we discussed in the lecture on condition variables. Notice that in this example, the process could "deadlock" itself by writing more into the pipe than the pipe can hold.

Pipes are really designed to be used between processes so having the writing process block until a reading process drains some of the data from the pipe is necessary.

Similarly, the reader of read end of the pipe will block if there is no data to read in the pipe. However, if and when the write end is closed, the read end will unblock and the read will read no data. This interaction between reading a pipe, blocking, and closing a pipe will become clearer as we look at pipes between processes. The easiest way to think of it, though, is to realize that the reader of a pipe waiting for data would wait forever if the writer died without a way to inform the reader that the write end will never produce more data.

Usually a pipe is established between processes that wish to communicate rather than within a single process (as in the previous example). In pipe-2.c

```
#include < unistd.h >
#include < stdlib.h >
#include < stdio.h >
#include < string.h >
#include < fcntl.h >
#include < sys/wait.h >

/*
 * simple program creating a pipe between two processes
 */
int main(int argc, char **argv)
{
        pid_t child_id;
        pid_t my_id;
```

```
int pipe_desc[2];
char *string;
char read_buffer[4096];
int child_status;
int err;

/*
 * create the pipe
 */
err = pipe(pipe_desc);
if(err < 0) {
        printf("error creating pipe\n");
        exit(1);
}

/*
 * then fork
 */
child_id = fork();
if(child_id != 0) {
        /*
         * parent will be the writer
         * doesn't need the read end
         */
        my_id = getpid();
        close(pipe_desc[0]);
        /*
         * send the child a string
         */
        string = "a string made by the parent\n";
        printf("pid: %d -- writing %s to pipe_desc[1]\n",
                (int)my_id,
                string);

        write(pipe_desc[1],string,strlen(string));

        /*
         * close the pipe to let the read end know we are
         * done
         */
        close(pipe_desc[1]);
        /*
         * wait for the child to exit
         */
        wait(&child_status);
} else {
        /*
         * child reads the read end
         */
        my_id = getpid();
        /*
         * doesn't need the write end
         */
        close(pipe_desc[1]);
        memset(read_buffer,0,sizeof(read_buffer));

        read(pipe_desc[0],read_buffer,sizeof(read_buffer));
```

```
                printf("pid: %d -- received %s from parent\n",
                                (int)my_id,
                                read_buffer);
                close(pipe_desc[0]);
        }

        printf("pid: %d -- I am exiting\n",my_id);
        exit(0);

}
```

Notice that the code creates a pipe before it calls *fork()*. As a result, both the parent and child process have the pipe open on the same set of file descriptors. It is good programming practice to close the pipe descriptors in a process that are not being used. Thus, the parent closes the read end (because it writes in this example) and the child closes the write end (because it reads). The parent closes the pipe after the write and then waits for the child to exit.

To see the close of write end trigger the reading end to exit, take a look at the code in pipe-3.c

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

/*
 * simple program creating a pipe between two processes
 */
int main(int argc, char **argv)
{
        pid_t child_id;
        pid_t my_id;
        int pipe_desc[2];
        char *string;
        char read_buffer[4096];
        int err;

        /*
         * create the pipe
         */
        err = pipe(pipe_desc);
        if(err < 0) {
                printf("error creating pipe\n");
                exit(1);
        }

        /*
         * then fork
         */
        child_id = fork();
        if(child_id != 0) {
                /*
                 * parent will be the writer
                 * doesn't need the read end
                 */
```

```
                my_id = getpid();
                close(pipe_desc[0]);
                /*
                 * send the child a string
                 */
                string = "a string made by the parent\n";
                printf("pid: %d -- writing %s to pipe_desc[1]\n",
                        (int)my_id,
                        string);
                write(pipe_desc[1],string,strlen(string));
                /*
                 * and another string
                 */
                string = "and another string";
                printf("pid: %d -- writing %s to pipe_desc[1]\n",
                        (int)my_id,
                        string);
                write(pipe_desc[1],string,strlen(string));

                /*
                 * fall off the end to close
                 */

        } else {
                /*
                 * child will read until pipe closes
                 * close the write end
                 */
                my_id = getpid();
                close(pipe_desc[1]);
                memset(read_buffer,0,sizeof(read_buffer));

                while(read(pipe_desc[0],read_buffer,sizeof(read_buffer))) {
                        printf("pid: %d -- received %s from parent\n",
                                (int)my_id,
                                read_buffer);
                        memset(read_buffer,0,sizeof(read_buffer));
                }

                printf("pid: %d -- child detects write end closed\n",
                        (int)my_id);
                close(pipe_desc[0]);
        }

        printf("pid: %d -- I am exiting\n",my_id);
        exit(0);

}
```

Here, the child reads the pipe in a while loop looking for the read to complete with a value of zero (indicating end-of-file). When the parent closes the write end, the read end will deliver EOF **after** it has delivered what ever data is in the pipe. That is, the close and exit of the parent does not prevent the data in transit from being delivered.

The output

```
./pipe-3
```

```
pid: 98839 -- writing a string made by the parent
 to pipe_desc[1]
pid: 98839 -- writing and another string to pipe_desc[1]
pid: 98839 -- I am exiting
pid: 98840 -- received a string made by the parent
and another string from parent
pid: 98840 -- child detects write end closed
pid: 98840 -- I am exiting
```

also shows the parent writing both strings and exiting before the child runs. The child gets both strings (but in one read call -- not two) then detects the close of the write end as a zero rerturned from read (causing it to fall out of the while loop).

Putting It All Together

At this point, you have enough to understand how to write a program that creates arbitrary chains of pipes that communicate via processes that use Standard In and Standard Out.

Consider the code in my-cat.c which is a program that implements functionality similar to the Linux *cat* utility:

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

/*
 * simple program implementing cat
 */
int main(int argc, char **argv)
{
        pid_t my_id;
        char buffer[4096];

        my_id = getpid();
        fprintf(stderr,"pid: %d -- I am my-cat and I have started\n",my_id);

        memset(buffer,0,sizeof(buffer));

        while(read(0,buffer,sizeof(buffer)) > 0) {
                fprintf(stderr,"pid: %d read some data\n",getpid());
                buffer[4095] = 0; /* safety first */
                write(1,buffer,strlen(buffer));
                fprintf(stderr,"pid: %d wrote some data\n",getpid());
                memset(buffer,0,sizeof(buffer));
        }

        fprintf(stderr,"pid: %d -- I am my-cat and I am exiting\n",my_id);

        exit(0);

}
```

It simply reads data from Standard In (that it assumes is string data) and echos it to Standard Out. It also prints a message when it starts and when it ends. If you run it from the command line

```
./my-cat
pid: 33672 -- I am my-cat and I have started
I love the OS class more than any other class!
pid: 33672 read some data
I love the OS class more than any other class!
pid: 33672 wrote some data
It is really great to know how to program in C and to understand Linux!
pid: 33672 read some data
It is really great to know how to program in C and to understand Linux!
pid: 33672 wrote some data
It is really great to know how to program in C and to understand Linux!
pid: 33672 read some data
It is really great to know how to program in C and to understand Linux!
pid: 33672 wrote some data
pid: 33672 -- I am my-cat and I am exiting
```

it will echo each line you type, one at a time, until you type *ctrl-D*.

Wait.

One line at a time? Why? That is, why doesn't it wait until I type all three lines and a *ctrl-D* before it echos the buffer?

The answer is that the newline character on the input line causes the call to *read()* to complete and return. It still what you want. Try putting the three different sentences in a .txt file and running *./my-cat < my.txt*. It should just print the lines out in the order they appear.

Also notice that the messages *my-cat* prints always go to the terminal because I print them to Standard Error. I use Standard Error in this for reasons that will become clear in a moment. However, notice that *my-cat* prints its pid every time it reads something or writes something.

To create a chain of *my-cat* process, each of which reads from the one before it in the chain and write to the one after it in the chain, you need to

- create a pipe to serve as the communication channel between processes
- in the up-stream process make make stdout write the write end of the pipe
- in the down stream process make stdin read in the read end of the pipe
- repeat

You also need to make sure that the first process in the chain gets Standard In from the original process (i.e the terminal) and the last process in the chain gets the Standard Out from the original process.

To change file descriptors you need to use the system calls *dup()* or *dup2()*. I prefer *dup2()* because it is more explicit. These calls duplicate an open file descriptor. The *dup()* call will choose the lowest numbered file descriptor that is unopened as the target and *dup2()* lets you specify which file descriptor you want to use as the target.

For example

```
int fd = open("./my-file.txt",O_RDWR);
close(0);
dup(fd);
```

will set the Standard In file descriptor to be the same file descriptor as *fd*. Thus when your process calls *read(0,...)* it will not be reading from the terminal any longer but from the file. The sequence

```
int fd = open("./my-file.txt",O_RDWR);
close(0);
dup2(fd,0);
```

does the same. If you leave out the call to *close()* for the target, it will close it for you.

Thus *dup()* and *dup2()* are ways that you can control file descriptors in general. More specifically, you can use them to change the Standard In and Standard Out of a process to be the read and write ends of a pipe.

In pipe-4.c the code does, more or less what the shell does when you use the | symbol create chains of programs separated by pipes.

```
#include < unistd.h >
#include < stdlib.h >
#include < stdio.h >
#include < string.h >
#include < fcntl.h >
#include < sys/wait.h >

/*
 * simple program creating a pipe between a set of processes
 * that read and write stdin and stdout
 * first child get stdin of parent
 * last child gets stdout of parent
 */
int main(int argc, char **argv, char **envp)
{
        pid_t child_id;
        int child_status;
        int pipe_desc[2];
        char read_buffer[4096];
        int err;
        char file_name[4096];
        int proc_count;
        int i;
        int last_stdout;

        if(argc < 3) {
                printf("usage: pipe-4 executable-file number-of-procs\n");
                exit(1);
        }

        /*
         * get the arguments
         */
```

```
memset(file_name,0,sizeof(file_name));
strncpy(file_name,argv[1],sizeof(file_name));
proc_count = atoi(argv[2]);

/*
 * save the parent stdout for the last child
 */

last_stdout = dup(1);


/*
 * create pipes, fork and exec the processes
 */
for(i=0; i < proc_count; i++) {
        /*
         * create the pipe
         */
        err = pipe(pipe_desc);
        if(err < 0) {
                printf("error creating pipe\n");
                exit(1);
        }

        /*
         * then fork
         */
        child_id = fork();
        if(child_id == 0) {
                /*
                 * child closes standard out
                 */

                close(1);
                /*
                 * child dups the write end of the pipe
                 * if it is not the last child, otherwise
                 * it dups the last_stdout it got from the parent
                 * the closed stdout will be chosen as the
                 * target
                 */
                if(i < (proc_count - 1)) {
                        dup2(pipe_desc[1],1);
                } else {
                        dup2(last_stdout,1);
                        close(pipe_desc[1]);
                }

                /*
                 * child runs the program
                 */
                err = execve(file_name,&argv[1],envp);
                printf("parent error: %s didn't exec\n",
                        file_name);
                exit(1);
        } else {
```

```
                        /*
                         * parent closes standard in
                         */
                        close(0);
                        /*
                         * parent dups the read end of the pipe for the
                         * next child
                         */
                        if(i < (proc_count - 1)) {
                                dup2(pipe_desc[0],0);
                        }
                        close(pipe_desc[0]);
                        close(pipe_desc[1]);

                }
        }

        close(1);
        dup2(last_stdout,1);
        /*
         * parent now waits for the children to exit
         */
        for(i=0; i < proc_count; i++) {
                child_id = wait(&child_status);
        }


        exit(0);

}
```

This code is worth understanding. It takes the name of a file as its first argument and the number of processes to create as its second argument. The processes must read and write Standard In and Standard Out (like my-cat.c.

Try running it:

```
./pipe-4 ./my-cat 2
pid: 34122 -- I am my-cat and I have started
pid: 34123 -- I am my-cat and I have started
I love OS
pid: 34122 read some data
pid: 34122 wrote some data
pid: 34123 read some data
I love OS
pid: 34123 wrote some data
pid: 34122 -- I am my-cat and I am exiting
pid: 34123 -- I am my-cat and I am exiting
```

The code forked two children (pid: 34122 and pid: 34123). When you type an important sentence to the terminal, 34122 wakes up, reads the message from Standard In (file descriptor 0) and writes it to Stabdard Out (file descriptor 1). The parent has been careful to leave file descriptor alone on its first fork so that the first child gets the terminal as Standard IN. Then 34123 wakes up, reads its Standard In, and writes it to its Standard Out. In this case, however, since it is the

end of the chain, the parent has arranged that its Standard Out is the terminal. Thus you see the sentence a second time when 34123 writes it. Try it with three processes

```
./pipe-4 ./my-cat 3
pid: 34201 -- I am my-cat and I have started
pid: 34202 -- I am my-cat and I have started
pid: 34203 -- I am my-cat and I have started
I love OS
pid: 34201 read some data
pid: 34201 wrote some data
pid: 34202 read some data
pid: 34202 wrote some data
pid: 34203 read some data
I love OS
pid: 34203 wrote some data
pid: 34201 -- I am my-cat and I am exiting
pid: 34202 -- I am my-cat and I am exiting
pid: 34203 -- I am my-cat and I am exiting
```

and with four

```
./pipe-4 ./my-cat 4
pid: 34225 -- I am my-cat and I have started
pid: 34226 -- I am my-cat and I have started
pid: 34228 -- I am my-cat and I have started
pid: 34227 -- I am my-cat and I have started
I love OS
pid: 34225 read some data
pid: 34225 wrote some data
pid: 34226 read some data
pid: 34226 wrote some data
pid: 34227 read some data
pid: 34227 wrote some data
pid: 34228 read some data
I love OS
pid: 34228 wrote some data
pid: 34225 -- I am my-cat and I am exiting
pid: 34226 -- I am my-cat and I am exiting
pid: 34227 -- I am my-cat and I am exiting
pid: 34228 -- I am my-cat and I am exiting
```

Each time, the first process in the set of children that has been forked reads the string from Standard In and passes it to the next process in the chain. The parent gave the first child its own Standard In which was the terminal so that first child reads the terminal. Eventually, the last process in the chain writes it to Standard Out which has been set to be the same Standard Out the parent had when it ran.

The logic takes a little while to understand. In a loop, the parent creates a pipe and forks a child. The child, before it execs the program specified as the second argument, dups the write end of the pipe to Standard Out so that after the exec, the program that is running will be writing the write end of the pipe when it writes to its Standard Out.

The parent, then, must dup the read end of the pipe into its own Standard In so that when it forks the next time, the child will receive the read end of the pipe the parent created on the last iteration. Thus the child before it in the loop will be writing the write end and the next child will be reading the read end.

The logic must also detect when the last child in the list has been forked and, instead of duping the write end of the pipe, it must dup the original Standard Out for the parent (which is duped at the start of the parent as *last_stdout*).

Lastly, because *my-cat* is also trying to print a status message, if it uses *printf()* Linux will try to use Standard Out. It actually works, but has a problem as the processes exit because Standard Out isn't being closed on process exit. To keep things "straight" I've used Standard Error (which none of the code change with *dup()* so that *my-cat* can send messages to the screen without using Standard Out.

That's it for the basic system calls in Linux. There are many others, but these are the ones you need to do file I/O, create and descroy processes and allow processes to communicate between themselves.