5

Interlude: Process API

ASIDE: INTERLUDES

Interludes will cover more practical aspects of systems, including a particular focus on operating system APIs and how to use them. If you don't like practical things, you could skip these interludes. But you should like practical things, because, well, they are generally useful in real life; companies, for example, don't usually hire you for your non-practical skills.

In this interlude, we discuss process creation in UNIX systems. UNIX presents one of the most intriguing ways to create a new process with a pair of system calls: `fork()` and `exec()`. A third routine, `wait()`, can be used by a process wishing to wait for a process it has created to complete. We now present these interfaces in more detail, with a few simple examples to motivate us. And thus, our problem:

CRUX: HOW TO CREATE AND CONTROL PROCESSES

What interfaces should the OS present for process creation and control? How should these interfaces be designed to enable powerful functionality, ease of use, and high performance?

5.1 The `fork()` System Call

The `fork()` system call is used to create a new process [C63]. However, be forewarned: it is certainly the strangest routine you will ever call[1]. More specifically, you have a running program whose code looks like what you see in Figure 5.1; examine the code, or better yet, type it in and run it yourself!

[1] Well, OK, we admit that we don't know that for sure; who knows what routines you call when no one is looking? But `fork()` is pretty odd, no matter how unusual your routine-calling patterns are.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6    printf("hello (pid:%d)\n", (int) getpid());
7    int rc = fork();
8    if (rc < 0) {
9      // fork failed
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) {
13     // child (new process)
14     printf("child (pid:%d)\n", (int) getpid());
15   } else {
16     // parent goes down this path (main)
17     printf("parent of %d (pid:%d)\n",
18             rc, (int) getpid());
19   }
20   return 0;
21 }
22
```

Figure 5.1: **Calling `fork()` (`p1.c`)**

When you run this program (called `p1.c`), you'll see the following:

```
prompt> ./p1
hello (pid:29146)
parent of 29147 (pid:29146)
child (pid:29147)
prompt>
```

Let us understand what happened in more detail in `p1.c`. When it first started running, the process prints out a hello message; included in that message is its **process identifier**, also known as a **PID**. The process has a PID of 29146; in UNIX systems, the PID is used to name the process if one wants to do something with the process, such as (for example) stop it from running. So far, so good.

Now the interesting part begins. The process calls the `fork()` system call, which the OS provides as a way to create a new process. The odd part: the process that is created is an (almost) *exact copy of the calling process*. That means that to the OS, it now looks like there are two copies of the program `p1` running, and both are about to return from the `fork()` system call. The newly-created process (called the **child**, in contrast to the creating **parent**) doesn't start running at `main()`, like you might expect (note, the "hello" message only got printed out once); rather, it just comes into life as if it had called `fork()` itself.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(int argc, char *argv[]) {
7    printf("hello (pid:%d)\n", (int) getpid());
8    int rc = fork();
9    if (rc < 0) {          // fork failed; exit
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) { // child (new process)
13     printf("child (pid:%d)\n", (int) getpid());
14   } else {               // parent goes down this path
15     int rc_wait = wait(NULL);
16     printf("parent of %d (rc_wait:%d) (pid:%d)\n",
17            rc, rc_wait, (int) getpid());
18   }
19   return 0;
20 }
21
```

Figure 5.2: **Calling `fork()` And `wait()` (`p2.c`)**

You might have noticed: the child isn't an *exact* copy. Specifically, although it now has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of **fork()** is different. Specifically, while the parent receives the PID of the newly-created child, the child receives a return code of zero. This differentiation is useful, because it is simple then to write the code that handles the two different cases (as above).

You might also have noticed: the output (of `p1.c`) is not **deterministic**. When the child process is created, there are now two active processes in the system that we care about: the parent and the child. Assuming we are running on a system with a single CPU (for simplicity), then either the child or the parent might run at that point. In our example (above), the parent did and thus printed out its message first. In other cases, the opposite might happen, as we show in this output trace:

```
prompt> ./p1
hello (pid:29146)
child (pid:29147)
parent of 29147 (pid:29146)
prompt>
```

The CPU **scheduler**, a topic we'll discuss in great detail soon, determines which process runs at a given moment in time; because the scheduler is complex, we cannot usually make strong assumptions about what

it will choose to do, and hence which process will run first. This **non-determinism**, as it turns out, leads to some interesting problems, particularly in **multi-threaded programs**; hence, we'll see a lot more non-determinism when we study **concurrency** in the second part of the book.

5.2 The `wait()` System Call

So far, we haven't done much: just created a child that prints out a message and exits. Sometimes, as it turns out, it is quite useful for a parent to wait for a child process to finish what it has been doing. This task is accomplished with the `wait()` system call (or its more complete sibling `waitpid()`); see Figure 5.2 for details.

In this example (`p2.c`), the parent process calls `wait()` to delay its execution until the child finishes executing. When the child is done, `wait()` returns to the parent.

Adding a `wait()` call to the code above makes the output deterministic. Can you see why? Go ahead, think about it.

(waiting for you to think .... and done)

Now that you have thought a bit, here is the output:

```
prompt> ./p2
hello (pid:29266)
child (pid:29267)
parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

With this code, we now know that the child will always print first. Why do we know that? Well, it might simply run first, as before, and thus print before the parent. However, if the parent does happen to run first, it will immediately call `wait()`; this system call won't return until the child has run and exited[2]. Thus, even when the parent runs first, it politely waits for the child to finish running, then `wait()` returns, and then the parent prints its message.

5.3 Finally, The `exec()` System Call

A final and important piece of the process creation API is the `exec()` system call[3]. This system call is useful when you want to run a program that is different from the calling program. For example, calling `fork()`

[2] There are a few cases where `wait()` returns before the child exits; read the man page for more details, as always. And beware of any absolute and unqualified statements this book makes, such as "the child will always print first" or "UNIX is the best thing in the world, even better than ice cream."

[3] On Linux, there are six variants of `exec()`: `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`, and `execvpe()`. Read the man pages to learn more.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char *argv[]) {
8    printf("hello (pid:%d)\n", (int) getpid());
9    int rc = fork();
10   if (rc < 0) {          // fork failed; exit
11     fprintf(stderr, "fork failed\n");
12     exit(1);
13   } else if (rc == 0) { // child (new process)
14     printf("child (pid:%d)\n", (int) getpid());
15     char *myargs[3];
16     myargs[0] = strdup("wc");   // program: "wc"
17     myargs[1] = strdup("p3.c"); // arg: input file
18     myargs[2] = NULL;           // mark end of array
19     execvp(myargs[0], myargs);  // runs word count
20     printf("this shouldn't print out");
21   } else {              // parent goes down this path
22     int rc_wait = wait(NULL);
23     printf("parent of %d (rc_wait:%d) (pid:%d)\n",
24            rc, rc_wait, (int) getpid());
25   }
26   return 0;
27 }
28
```

Figure 5.3: **Calling `fork()`, `wait()`, And `exec()` (`p3.c`)**

in `p2.c` is only useful if you want to keep running copies of the same program. However, often you want to run a *different* program; `exec()` does just that (Figure 5.3).

In this example, the child process calls `execvp()` in order to run the program `wc`, which is the word counting program. In fact, it runs `wc` on the source file `p3.c`, thus telling us how many lines, words, and bytes are found in the file:

```
prompt> ./p3
hello (pid:29383)
child (pid:29384)
      29     107    1030 p3.c
parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

The `fork()` system call is strange; its partner in crime, `exec()`, is not so normal either. What it does: given the name of an executable (e.g., `wc`), and some arguments (e.g., `p3.c`), it **loads** code (and static data) from that

TIP: GETTING IT RIGHT (LAMPSON'S LAW)
As Lampson states in his well-regarded "Hints for Computer Systems Design" [L83], "**Get it right**. Neither abstraction nor simplicity is a substitute for getting it right." Sometimes, you just have to do the right thing, and when you do, it is way better than the alternatives. There are lots of ways to design APIs for process creation; however, the combination of `fork()` and `exec()` are simple and immensely powerful. Here, the UNIX designers simply got it right. And because Lampson so often "got it right", we name the law in his honor.

executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized. Then the OS simply runs that program, passing in any arguments as the `argv` of that process. Thus, it does *not* create a new process; rather, it transforms the currently running program (formerly `p3`) into a different running program (`wc`). After the `exec()` in the child, it is almost as if `p3.c` never ran; a successful call to `exec()` never returns.

5.4 Why? Motivating The API

Of course, one big question you might have: why would we build such an odd interface to what should be the simple act of creating a new process? Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell, because it lets the shell run code *after* the call to `fork()` but *before* the call to `exec()`; this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

The shell is just a user program[4]. It shows you a **prompt** and then waits for you to type something into it. You then type a command (i.e., the name of an executable program, plus any arguments) into it; in most cases, the shell then figures out where in the file system the executable resides, calls `fork()` to create a new child process to run the command, calls some variant of `exec()` to run the command, and then waits for the command to complete by calling `wait()`. When the child completes, the shell returns from `wait()` and prints out a prompt again, ready for your next command.

The separation of `fork()` and `exec()` allows the shell to do a whole bunch of useful things rather easily. For example:

```
prompt> wc p3.c > newfile.txt
```

[4] And there are lots of shells; `tcsh`, `bash`, and `zsh` to name a few. You should pick one, read its man pages, and learn more about it; all UNIX experts do.

In the example above, the output of the program `wc` is **redirected** into the output file `newfile.txt` (the greater-than sign is how said redirection is indicated). The way the shell accomplishes this task is quite simple: when the child is created, before calling `exec()`, the shell (specifically, the code executed in the child process) closes **standard output** and opens the file `newfile.txt`. By doing so, any output from the soon-to-be-running program `wc` is sent to the file instead of the screen (open file descriptors are kept open across the `exec()` call, thus enabling this behavior [SR05]).

Figure 5.4 (page 8) shows a program that does exactly this. The reason this redirection works is due to an assumption about how the operating system manages file descriptors. Specifically, UNIX systems start looking for free file descriptors at zero. In this case, STDOUT_FILENO will be the first available one and thus get assigned when `open()` is called. Subsequent writes by the child process to the standard output file descriptor, for example by routines such as `printf()`, will then be routed transparently to the newly-opened file instead of the screen.

Here is the output of running the `p4.c` program:

```
prompt> ./p4
prompt> cat p4.output
      32     109     846 p4.c
prompt>
```

You'll notice (at least) two interesting tidbits about this output. First, when `p4` is run, it looks as if nothing has happened; the shell just prints the command prompt and is immediately ready for your next command. However, that is not the case; the program `p4` did indeed call `fork()` to create a new child, and then run the `wc` program via a call to `execvp()`. You don't see any output printed to the screen because it has been redirected to the file `p4.output`. Second, you can see that when we `cat` the output file, all the expected output from running `wc` is found. Cool, right?

UNIX pipes are implemented in a similar way, but with the `pipe()` system call. In this case, the output of one process is connected to an in-kernel **pipe** (i.e., queue), and the input of another process is connected to that same pipe; thus, the output of one process seamlessly is used as input to the next, and long and useful chains of commands can be strung together. As a simple example, consider looking for a word in a file, and then counting how many times said word occurs; with pipes and the utilities `grep` and `wc`, it is easy; just type `grep -o foo file | wc -l` into the command prompt and marvel at the result.

Finally, while we just have sketched out the process API at a high level, there is a lot more detail about these calls out there to be learned and digested; we'll learn more, for example, about file descriptors when we talk about file systems in the third part of the book. For now, suffice it to say that the `fork()/exec()` combination is a powerful way to create and manipulate processes.

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

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC,
         S_IRWXU);
    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: wc
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

Figure 5.4: **All Of The Above With Redirection (`p4.c`)**

5.5 Process Control And Users

Beyond `fork()`, `exec()`, and `wait()`, there are a lot of other interfaces for interacting with processes in UNIX systems. For example, the `kill()` system call is used to send **signals** to a process, including directives to pause, die, and other useful imperatives. For convenience, in most UNIX shells, certain keystroke combinations are configured to deliver a specific signal to the currently running process; for example, control-c sends a `SIGINT` (interrupt) to the process (normally terminating it) and control-z sends a `SIGTSTP` (stop) signal thus pausing the process in mid-execution (you can resume it later with a command, e.g., the `fg` built-in command found in many shells).

The entire signals subsystem provides a rich infrastructure to deliver external events to processes, including ways to receive and process those signals within individual processes, and ways to send signals to individual processes as well as entire **process groups**. To use this form of com-

ASIDE: RTFM — READ THE MAN PAGES

Many times in this book, when referring to a particular system call or library call, we'll tell you to read the **manual pages**, or **man pages** for short. Man pages are the original form of documentation that exist on UNIX systems; realize that they were created before the thing called **the web** existed.

Spending some time reading man pages is a key step in the growth of a systems programmer; there are tons of useful tidbits hidden in those pages. Some particularly useful pages to read are the man pages for whichever shell you are using (e.g., **tcsh**, or **bash**), and certainly for any system calls your program makes (in order to see what return values and error conditions exist).

Finally, reading the man pages can save you some embarrassment. When you ask colleagues about some intricacy of `fork()`, they may simply reply: "RTFM." This is your colleagues' way of gently urging you to Read The Man pages. The F in RTFM just adds a little color to the phrase...

munication, a process should use the `signal()` system call to "catch" various signals; doing so ensures that when a particular signal is delivered to a process, it will suspend its normal execution and run a particular piece of code in response to the signal. Read elsewhere [SR05] to learn more about signals and their many intricacies.

This naturally raises the question: who can send a signal to a process, and who cannot? Generally, the systems we use can have multiple people using them at the same time; if one of these people can arbitrarily send signals such as `SIGINT` (to interrupt a process, likely terminating it), the usability and security of the system will be compromised. As a result, modern systems include a strong conception of the notion of a **user**. The user, after entering a password to establish credentials, logs in to gain access to system resources. The user may then launch one or many processes, and exercise full control over them (pause them, kill them, etc.). Users generally can only control their own processes; it is the job of the operating system to parcel out resources (such as CPU, memory, and disk) to each user (and their processes) to meet overall system goals.

5.6 Useful Tools

There are many command-line tools that are useful as well. For example, using the `ps` command allows you to see which processes are running; read the **man pages** for some useful flags to pass to `ps`. The tool `top` is also quite helpful, as it displays the processes of the system and how much CPU and other resources they are eating up. Humorously, many times when you run it, `top` claims it is the top resource hog; perhaps it is a bit of an egomaniac. The command `kill` can be used to send arbitrary

ASIDE: THE SUPERUSER (ROOT)
A system generally needs a user who can **administer** the system, and is not limited in the way most users are. Such a user should be able to kill an arbitrary process (e.g., if it is abusing the system in some way), even though that process was not started by this user. Such a user should also be able to run powerful commands such as `shutdown` (which, unsurprisingly, shuts down the system). In UNIX-based systems, these special abilities are given to the **superuser** (sometimes called **root**). While most users can't kill other users processes, the superuser can. Being root is much like being Spider-Man: with great power comes great responsibility [QI15]. Thus, to increase **security** (and avoid costly mistakes), it's usually better to be a regular user; if you do need to be root, tread carefully, as all of the destructive powers of the computing world are now at your fingertips.

signals to processes, as can the slightly more user friendly `killall`. Be sure to use these carefully; if you accidentally kill your window manager, the computer you are sitting in front of may become quite difficult to use.

Finally, there are many different kinds of CPU meters you can use to get a quick glance understanding of the load on your system; for example, we always keep **MenuMeters** (from Raging Menace software) running on our Macintosh toolbars, so we can see how much CPU is being utilized at any moment in time. In general, the more information about what is going on, the better.

5.7 Summary

We have introduced some of the APIs dealing with UNIX process creation: `fork()`, `exec()`, and `wait()`. However, we have just skimmed the surface. For more detail, read Stevens and Rago [SR05], of course, particularly the chapters on Process Control, Process Relationships, and Signals; there is much to extract from the wisdom therein.

While our passion for the UNIX process API remains strong, we should also note that such positivity is not uniform. For example, a recent paper by systems researchers from Microsoft, Boston University, and ETH in Switzerland details some problems with `fork()`, and advocates for other, simpler process creation APIs such as **`spawn()`** [B+19]. Read it, and the related work it refers to, to understand this different vantage point. While it's generally good to trust this book, remember too that the authors have opinions; those opinions may not (always) be as widely shared as you might think.

ASIDE: KEY PROCESS API TERMS

- Each process has a name; in most systems, that name is a number known as a **process ID** (**PID**).
- The **fork()** system call is used in UNIX systems to create a new process. The creator is called the **parent**; the newly created process is called the **child**. As sometimes occurs in real life [J16], the child process is a nearly identical copy of the parent.
- The **wait()** system call allows a parent to wait for its child to complete execution.
- The **exec()** family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.
- A UNIX **shell** commonly uses `fork()`, `wait()`, and `exec()` to launch user commands; the separation of fork and exec enables features like **input/output redirection**, **pipes**, and other cool features, all without changing anything about the programs being run.
- Process control is available in the form of **signals**, which can cause jobs to stop, continue, or even terminate.
- Which processes can be controlled by a particular person is encapsulated in the notion of a **user**; the operating system allows multiple users onto the system, and ensures users can only control their own processes.
- A **superuser** can control all processes (and indeed do many other things); this role should be assumed infrequently and with caution for security reasons.

References

[B+19] "A fork() in the road" by Andrew Baumann, Jonathan Appavoo, Orran Krieger, Timothy Roscoe. HotOS '19, Bertinoro, Italy. *A fun paper full of `fork()`ing rage. Read it to get an opposing viewpoint on the* UNIX *process API. Presented at the always lively HotOS workshop, where systems researchers go to present extreme opinions in the hopes of pushing the community in new directions.*

[C63] "A Multiprocessor System Design" by Melvin E. Conway. AFIPS '63 Fall Joint Computer Conference, New York, USA 1963. *An early paper on how to design multiprocessing systems; may be the first place the term `fork()` was used in the discussion of spawning new processes.*

[DV66] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis and Earl C. Van Horn. Communications of the ACM, Volume 9, Number 3, March 1966. *A classic paper that outlines the basics of multiprogrammed computer systems. Undoubtedly had great influence on Project MAC, Multics, and eventually* UNIX.

[J16] "They could be twins!" by Phoebe Jackson-Edwards. The Daily Mail. March 1, 2016.. *This hard-hitting piece of journalism shows a bunch of weirdly similar child/parent photos and is frankly kind of mesmerizing. Go ahead, waste two minutes of your life and check it out. But don't forget to come back here! This, in a microcosm, is the danger of surfing the web.*

[L83] "Hints for Computer Systems Design" by Butler Lampson. ACM Operating Systems Review, Volume 15:5, October 1983. *Lampson's famous hints on how to design computer systems. You should read it at some point in your life, and probably at many points in your life.*

[QI15] "With Great Power Comes Great Responsibility" by The Quote Investigator. Available: `https://quoteinvestigator.com/2015/07/23/great-power`. *The quote investigator concludes that the earliest mention of this concept is 1793, in a collection of decrees made at the French National Convention. The specific quote: "Ils doivent envisager qu'une grande responsabilité est la suite inséparable d'un grand pouvoir", which roughly translates to "They must consider that great responsibility follows inseparably from great power." Only in 1962 did the following words appear in Spider-Man: "...with great power there must also come–great responsibility!" So it looks like the French Revolution gets credit for this one, not Stan Lee. Sorry, Stan.*

[SR05] "Advanced Programming in the UNIX Environment" by W. Richard Stevens, Stephen A. Rago. Addison-Wesley, 2005. *All nuances and subtleties of using* UNIX *APIs are found herein. Buy this book! Read it! And most importantly,* ***live it.***

Homework (Simulation)

This simulation homework focuses on `fork.py`, a simple process creation simulator that shows how processes are related in a single "familial" tree. Read the relevant README for details about how to run the simulator.

Questions

1. Run `./fork.py -s 10` and see which actions are taken. Can you predict what the process tree looks like at each step? Use the `-c` flag to check your answers. Try some different random seeds (`-s`) or add more actions (`-a`) to get the hang of it.

2. One control the simulator gives you is the `fork_percentage`, controlled by the `-f` flag. The higher it is, the more likely the next action is a fork; the lower it is, the more likely the action is an exit. Run the simulator with a large number of actions (e.g., `-a 100`) and vary the `fork_percentage` from 0.1 to 0.9. What do you think the resulting final process trees will look like as the percentage changes? Check your answer with `-c`.

3. Now, switch the output by using the `-t` flag (e.g., run `./fork.py -t`). Given a set of process trees, can you tell which actions were taken?

4. One interesting thing to note is what happens when a child exits; what happens to its children in the process tree? To study this, let's create a specific example: `./fork.py -A a+b,b+c,c+d,c+e,c-`. This example has process 'a' create 'b', which in turn creates 'c', which then creates 'd' and 'e'. However, then, 'c' exits. What do you think the process tree should like after the exit? What if you use the `-R` flag? Learn more about what happens to orphaned processes on your own to add more context.

5. One last flag to explore is the `-F` flag, which skips intermediate steps and only asks to fill in the final process tree. Run `./fork.py -F` and see if you can write down the final tree by looking at the series of actions generated. Use different random seeds to try this a few times.

6. Finally, use both `-t` and `-F` together. This shows the final process tree, but then asks you to fill in the actions that took place. By looking at the tree, can you determine the exact actions that took place? In which cases can you tell? In which can't you tell? Try some different random seeds to delve into this question.

ASIDE: CODING HOMEWORKS

Coding homeworks are small exercises where you write code to run on a real machine to get some experience with some basic operating system APIs. After all, you are (probably) a computer scientist, and therefore should like to code, right? If you don't, there is always CS theory, but that's pretty hard. Of course, to truly become an expert, you have to spend more than a little time hacking away at the machine; indeed, find every excuse you can to write some code and see how it works. Spend the time, and become the wise master you know you can be.

Homework (Code)

In this homework, you are to gain some familiarity with the process management APIs about which you just read. Don't worry – it's even more fun than it sounds! You'll in general be much better off if you find as much time as you can to write some code, so why not start now?

Questions

1. Write a program that calls `fork()`. Before calling `fork()`, have the main process access a variable (e.g., `x`) and set its value to something (e.g., `100`). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of `x`?

2. Write a program that opens a file (with the `open()` system call) and then calls `fork()` to create a new process. Can both the child and parent access the file descriptor returned by `open()`? What happens when they are writing to the file concurrently, i.e., at the same time?

3. Write another program using `fork()`. The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this *without* calling **wait()** in the parent?

4. Write a program that calls `fork()` and then calls some form of `exec()` to run the program `/bin/ls`. See if you can try all of the variants of `exec()`, including (on Linux) `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()`, and `execvpe()`. Why do you think there are so many variants of the same basic call?

5. Now write a program that uses `wait()` to wait for the child process to finish in the parent. What does `wait()` return? What happens if you use `wait()` in the child?

6. Write a slight modification of the previous program, this time using `waitpid()` instead of `wait()`. When would `waitpid()` be useful?
7. Write a program that creates a child process, and then in the child closes standard output (`STDOUT_FILENO`). What happens if the child calls `printf()` to print some output after closing the descriptor?
8. Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the `pipe()` system call.