CS110 Summer 2019
Lecture 5: Introduction to Multiprocessing

Principles of Computer Systems
Stanford University, Dept. Of Computer Science
Lecturer: Roslyn Michelle Cyrus
Content adapted from material by Jerry Cain. Diagrams by Roslyn.
Announcements

- Readings and lecture slides updated on website
  - I slightly modified the last couple of lectures to address clarifications discussed in class.
  - For those of you who are unfamiliar with the virtual address space diagram that I showed in the last lecture, read Chapter 1, section 7 of the B&O textbook

- Clarification
  - Processors vs. cores: “A multi-core processor is a computer processor integrated circuit with two or more separate processing units, called cores, which each read and execute program instructions, as if the computer had several processors.” (source)

- Assignment 1 due tomorrow at noon!

- Midterm scheduled for Tuesday, July 23rd, 6 - 8 pm in Hewlett 201.
  - If you plan on taking the exam remotely (SCPD), please email me (even if you told me already).
Lecture Overview

- Multiprocessing
- The \texttt{fork} system call
Accessing Code Examples

- Today's lecture examples reside within:
  
  `/usr/class/cs110/lecture-examples/processes`

  - First `ssh` into a myth machine (`ssh yourusername@myth.stanford.edu`). When prompted for your password, it is normal for the text not to appear as you enter your password. Once logged onto a myth machine, `cd` into the above directory.

  - To get started, type:
    ```bash
    git clone /usr/class/cs110/lecture-examples cs110-lecture-examples
    ```
    at the command prompt to create a local copy of the master.

  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type `git pull`. Doing so will update your local copy to match whatever the master has become.
Processes Recap

- A **process** is an instance of a program in execution, versus a **program**, which is the code and data; a program always runs in the **context** of some process). This context has state that the program needs in order to properly run.

- Processes provide two key abstractions to an application:
  1. A private address space that provides the illusion that our program has exclusive use of memory
  2. An independent logical control flow that provides the illusion that our program has exclusive use of the processor
Multiprocessing

- Until now, we have been studying how programs interact with hardware, and now we are going to start investigating how programs interact with the operating system.

- In the CS curriculum so far, your programs have operated in a single process, meaning, basically, that one program was running your code, line-for-line. The operating system made it look like your program was the only thing running, and that was that.

- Now, we are going to move into the realm of multitasking, where you control more than one process at a time with your programs. You will tell the OS, “do these things concurrently”, and it will.
More About Processes

- When you start a program, it runs in a single process. It has a **process id** (an integer) that the OS assigns. A program can get its own process id with the `getpid` system call:

```c
// file: getpidEx.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // for getpid

int main(int argc, char *argv[]) {
    pid_t pid = getpid();
    printf("My process id: %d\n", pid);
    return 0;
}
```

$ ./getpidEx
My process id: 7526

- All programs are associated with one or more processes, and a process is scheduled to run its code by the OS.
Viewing Processes on Your Computer

Mac: Activity Monitor

Windows: Task Manager
Viewing Processes via a Linux shell

- To view processes running on a Linux machine like the myth machines, run `top`.
- You can also run `ps -A` to view a static list of all running processes.
Process Life Cycle

- Here is a simplified life cycle for each newly created process:

```
Create → Runnable → Running → Terminated
```

- Setup
- Scheduled
- Interrupted
- Canceled or completed
- Request completed
- Request made

Diagram shows transitions and states in the life cycle.
What is a shell?

- A **shell** is a command-line interpreter (CLI) that runs programs on behalf of the user.
- It is called a shell because it is a “wrapper” around the kernel.
- To find out which shell you are running:

```
myth54$ ps -p $$
```

<table>
<thead>
<tr>
<th>PID</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>28740</td>
<td>pts/8</td>
<td>00:00:00</td>
<td>bash</td>
</tr>
</tbody>
</table>

- The first `$` denotes a shell variable, and the second `$` gets the process ID of the current process, which is the shell
- Columns:
  - PID is the process ID
  - TTY is the terminal (device file) associated with the process
  - TIME is the total CPU usage
  - CMD is the command (name of the process)
- Differences between shell, terminal, etc.: read [this](#)
What is a shell? (continued)

- To view a list of login shells:

  ```
  myth54$ cat /etc/shells
  # /etc/shells: valid login shells
  /bin/sh
  /bin/dash
  /bin/bash
  /bin/rbash
  /usr/bin/screen
  /usr/bin/tmux
  /bin/zsh
  /usr/bin/zsh
  /bin/tcsh
  /usr/bin/tcsh
  /usr/bin/fish
  ```

- All shells are similar but come with different features
New System Call: fork

- A program may decide that it wants to run multiple processes itself. We will see many examples of why a program may want to do this as the course progresses.
- If a program wants to launch a second process, it uses the fork system call.
- fork() does exactly this:
  - It creates a new process (a child process) that starts on the next instruction after the fork call. The parent process also continues on the next instruction, as well.
  - A successful fork call returns a pid_t (an integer) to both processes (and returns -1 on error). Neither is the actual pid of the process that receives it:
    - The parent process gets a return value that is the pid of the child process.
    - The child process gets a return value of 0, indicating that it is the child.
      - The child process does, indeed, have its own pid, but it would need to call getpid itself to retrieve it.
  - All memory is identical between the parent and child, though it is not shared (it is copied).
Understanding fork

Here's a simple program that knows how to spawn new processes. It uses system calls named `fork`, `getpid`, and `getppid`. The full program can be viewed [here](#).

```c
int main(int argc, char *argv[]) {
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

- `getpid` and `getppid` return the process id of the caller and the process id of the caller's parent, respectively. Here's the output of the above program.

```
myth54$ echo $$
28740
myth54$ ./basic-fork
Greetings from process 29686! (parent 28740)
Bye-bye from process 29686! (parent 28740)
Bye-bye from process 29687! (parent 29686)
```

- `echo $$` prints the process ID of the current shell.
Understanding fork

fork is called once, but it returns twice.

- **fork** knows how to clone the calling process, synthesize a *nearly identical* copy of it (with a new process ID), and schedule the copy as if it were running all along. The child (clone) process gets a separate copy of:
  - The parent’s user-level virtual address space (code and data segments, heap, shared libraries, and user stack)
  - All open file descriptors (these copies are donated to the clone). This means the child can read and write any files that were open in the parent when it called **fork**.
- As a result, the output of our above program is the output of two processes.

```c
int main(int argc, char *argv[]) { // basic-fork.c
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

Reading: B&O’s Exceptional Control Flow chapter, section 4
Understanding fork

```c
int main(int argc, char *argv[]) { // basic-fork.c
    printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
    pid_t pid = fork();
    assert(pid >= 0);
    printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
    return 0;
}
```

- Fork calls can be diagrammed using process graphs (assert not shown):

![Diagram of fork call process](image)
Understanding fork

- Differences between parent calling `fork` and child generated by it:
  - The most obvious difference is that each gets a unique process id. That's important. Otherwise, the OS can't tell them apart.
  - Another key difference: `fork`'s return value in the two processes
    - When `fork` returns in the parent process, it returns the pid of the new child.
    - When `fork` returns in the child process, it returns 0. That isn't to say the child's pid is 0, but rather that `fork` elects to return a 0 as a way of allowing the child process to easily self-identify as the child process.
    - The return value can be used to dispatch each of the two processes in a different direction (although in this introductory example, we don't do that).
Understanding fork

myth54$ echo $$
28740
myth54$ ./basic-fork
Greetings from process 29686! (parent 28740)
Bye-bye from process 29686! (parent 28740)
Bye-bye from process 29687! (parent 29686)

● Here's why the program output makes sense:
  ○ Process IDs are generally assigned consecutively. That's why you see 29686 and 29687.
  ○ 28740 is the pid of the shell itself, and you can see that the basic-fork process—with pid 29686—is a direct child processes of the terminal. The output tells us so.
  ○ The clone of the original is assigned pid 29687, and the output is clear about the parent-child relationship between 29686 and 29687.
  ○ You can run this program multiple times to get a better understanding of the process IDs.

● Question: why is the child also able to print to stdout?
Understanding fork

- **Question:** why is the child also able to print to stdout?

  - Because the child gets its own copy of the parent’s file descriptors.
  - Note: the system could have a different open file table entry (these entries are called **file descriptions**, by the way) for the default descriptors: 0, 1, and 2 (as shown in diagram). Or, it could have one file description for fds 0, 1, and 2. It all depends on the implementation.
## Bonus: Fancy Linux Commands

- To view a list of file descriptors in the shell's descriptor table:

```
myth54$ ls /proc/28740/fd/
0 1 2 255 3 4
```

- To see which files these descriptors point to in the open file table:

```
myth54$ lsof -a -p 28740
```

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>PID</th>
<th>USER</th>
<th>FD</th>
<th>TYPE</th>
<th>DEVICE</th>
<th>SIZE/OFF</th>
<th>NODE</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>bash</td>
<td>28740</td>
<td>rcyrus</td>
<td>0u</td>
<td>CHR</td>
<td>136,8</td>
<td>0t0 11</td>
<td>/dev/pts/8</td>
<td></td>
</tr>
<tr>
<td>bash</td>
<td>28740</td>
<td>rcyrus</td>
<td>1u</td>
<td>CHR</td>
<td>136,8</td>
<td>0t0 11</td>
<td>/dev/pts/8</td>
<td></td>
</tr>
<tr>
<td>bash</td>
<td>28740</td>
<td>rcyrus</td>
<td>2u</td>
<td>CHR</td>
<td>136,8</td>
<td>0t0 11</td>
<td>/dev/pts/8</td>
<td></td>
</tr>
<tr>
<td>bash</td>
<td>28740</td>
<td>rcyrus</td>
<td>3r</td>
<td>CHR</td>
<td>1,9</td>
<td>0t0 11</td>
<td>/dev/urandom</td>
<td></td>
</tr>
<tr>
<td>bash</td>
<td>28740</td>
<td>rcyrus</td>
<td>4u</td>
<td>IPv4</td>
<td>1589382</td>
<td>0t0 11</td>
<td>TCP myth54.stanford.edu:56312-&gt;ldap2.stanford.edu:ldap (ESTABLISHED)</td>
<td></td>
</tr>
<tr>
<td>bash</td>
<td>28740</td>
<td>rcyrus</td>
<td>255u</td>
<td>CHR</td>
<td>136,8</td>
<td>0t0 11</td>
<td>/dev/pts/8</td>
<td></td>
</tr>
</tbody>
</table>

- `/dev/pts/8` is the terminal, which you can verify with the `tty` command:

```
myth54$ tty
/dev/pts/8
```

- Thus file descriptors 0, 1, 2, and 255 are connected to the terminal for reading and writing. 255 is like a backup file descriptor that bash uses for terminal access.
Bonuses: Fancy Linux Commands

- Checking the open file table entries for the `basic-for` program:

```
myth54$ lsof /dev/pts/8

myth54$ ./basic-fork &
[1] 17769
Greetings from process 17769! (parent 28740)
Bye-bye from process 17769! (parent 28740)
Bye-bye from process 17770! (parent 17769)
```

```
myth54$ lsof /dev/pts/8

```

Reading: B&O's Exceptional Control Flow chapter, section 4
Understanding fork

- The parent and child are separate processes that run concurrently.
  - The instructions in their logical control flows are arbitrarily interleaved by the kernel, so as programmers, we cannot make assumptions about the order in which the instructions will run in different processes. The output is nondeterministic.
- There is no default sharing of data between the two processes, though the parent process can wait (more on this later) for child processes to complete.
- You can use shared memory to communicate between processes, but this must be explicitly set up before making fork calls.
Understanding fork

- Another example (code here):

```c
int main() // fork-ints.c
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) { // Child
        printf("child: x=%d
", ++x);
        return 0;
    }

    // Parent
    printf("parent: x=%d
", --x);
    return 0;
}
```

What will the values of x be in each process?

```
myth54$ ./fork-ints
parent: x=0
child: x=2
```

- Since the parent and child are separate processes, they each have their own private address spaces, so any changes to one of the processes will not be reflected in the memory of the other process.
Understanding fork

- A trickier example (code [here](#)):

```c
int main(int argc, char *argv[]) {
    int x = 1;
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) { // in child
        printf("x=%d\n", ++x);
    }
    printf("x=%d\n", --x);
    return 0;
}
```

- The catch is that the child actually will execute both print statements since it doesn’t return/exit in the `if` statement! Also remember the possible nondeterministic orderings here. Think about a topological sort of the process graph.

**What will the values of x be in each process?**

**Answer:** In the child: x=2, then x=1. In the parent: x=0.

<table>
<thead>
<tr>
<th>myth54$ ./forkprob0</th>
<th>myth54$ ./forkprob0</th>
<th>myth54$ ./forkprob0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=0</td>
<td>x=2</td>
<td>x=2</td>
</tr>
<tr>
<td>x=2</td>
<td>x=1</td>
<td>x=0</td>
</tr>
<tr>
<td>x=1</td>
<td>x=0</td>
<td>x=1</td>
</tr>
</tbody>
</table>
Understanding fork

● Yet another example: a tree of fork calls (the full program can be viewed [here](#)):

```c
static const char const *kTrail = "abcd";

int main(int argc, char *argv[]) {
    size_t trailLength = strlen(kTrail);

    for (size_t i = 0; i < trailLength; i++) {
        printf("%c
", kTrail[i]);
        pid_t pid = fork();
        assert(pid >= 0);
    }

    return 0;
}
```

● While you rarely have reason to use `fork` this way, it's instructive to trace through a short program where spawned processes themselves call `fork`.

What kind of output do you expect?
Understanding fork

- Two samples runs are shown on the right.
- Reasonably obvious: A single a is printed by the soon-to-be-great-great-granddaddy process.
- Less obvious: The first child and the parent each return from `fork` and continue running in mirror processes, each with their own copy of the global "abcd" string, and each advancing to the `i++` line within a loop that promotes a 0 to 1. It's hopefully clear now that two b's will be printed, and this pattern will continue until the end.
- Key questions to answer:
  - Why aren't the two b's always consecutive?
  - How many c's get printed?
  - How many d's get printed?
  - Why is there a shell prompt in the middle of the output of the second run on the right?
Understanding fork

```
myth60$ ./fork-puzzle
a
b
```

```
myth60$ ./fork-puzzle
a
b
```

```
myth60$ d
```

```
myth60$ d
```

Reading: B&O's Exceptional Control Flow chapter, section 4
Waiting for Children to Finish

- Notice in the fork puzzle example that it was possible for the d's to be printed after the prompt. This is because the parent finishes before the child, but the child still has access to stdout and continues printing its data.

- Synchronization between parent and child can be done by using the system call `waitpid`. It can be used to temporarily block a process until a child process terminates or stops.

\[ \text{pid_t waitpid(pid_t pid, int *status, int options);} \]

- The first argument specifies the wait set, which for the moment is just the ID of the child process that needs to complete before `waitpid` can return.

- The second argument supplies the address of an integer where termination information can be placed (or we can pass in `NULL` if we don't care for the information).

- The third argument is a collection of bitwise-or'ed flags we'll study later. For the time being, we'll just go with 0 as the required parameter value, which means that `waitpid` should only return when a process in the supplied wait set exits.

- The return value is the pid of the child that exited, or -1 if `waitpid` was called and there were no child processes in the supplied wait set.
Waiting For Children to Finish

Consider the following program, which is more representative of how `fork` really gets used in practice (full program, with error checking, is right here):

```c
int main(int argc, char *argv[]) {
    printf("Before.\n");
    pid_t pid = fork();
    printf("After.\n");
    if (pid == 0) {
        printf("I am the child, and the parent will wait up for me.\n");
        return 110; // contrived exit status
    } else {
        int status;
        waitpid(pid, &status, 0);
        if (WIFEXITED(status)) {
            printf("Child exited with status %d.\n", WEXITSTATUS(status));
        } else {
            printf("Child terminated abnormally.\n");
        }
        return 0;
    }
}
```

- The parent process correctly waits for the child to complete using `waitpid`.
- The parent lifts child exit information out of the `waitpid` call, and uses the `WIFEXITED` macro to examine some high-order bits of its argument to confirm the process exited normally, and it uses the `WEXITSTATUS` macro to extract the lower eight bits of its argument to produce the child return value (which is 110 as expected).
- The `waitpid` call also donates child process-oriented resources back to the system.
Waiting For Children to Finish

The output is likely what is shown on the left below every single time the program is executed, since the parent likely continues running without halting when it calls fork (since all fork does is set up new data structures for a new process, returns, and then carries on). However, it is theoretically possible to get the output on the right if the child runs first:

```bash
myth60$ ./separate
Before.
After.
After.
I am the child, and the parent will wait up for me.
Child exited with status 110.
```

```bash
myth60$ ./separate
Before.
After.
After.
I'm the child, and the parent will wait up for me.
Child exited with status 110.
```
You might be asking yourself, *How do I debug two processes at once?* This is a very good question! `gdb` has built-in support for debugging multiple processes, as follows:

- **set detach-on-fork off**
  - This tells `gdb` to capture any fork'd processes, though it pauses them upon the fork.

- **info inferiors**
  - This lists the processes that `gdb` has captured.

- **inferior X**
  - Switch to a different process to debug it.

- **detach inferior X**
  - Tell `gdb` to stop watching the process, and continue it.

You can see an entire debugging session on the basic-fork program [right here.](#)
The Point Of It All

So, now we know how to create processes... but why would we do that in the first place? There are two major reasons:

- performance (ability to use multiple CPUs) and
- security (isolation of possibly sensitive components of an application).

Next time, we’ll talk about a third reason: starting executables from disk.

Reading: B&O’s Exceptional Control Flow chapter, section 4
End of Lecture 5