CS111, Lecture 9
Multiprocessing System Calls

_masks strongly recommended_
Topic 2: Multiprocessing - How can our program create and interact with other programs? How does the operating system manage user programs?
CS111 Topic 2: Multiprocessing

Lecture 8: Multiprocessing Introduction

Today: Managing processes and running other programs

Lecture 10 / 11: Inter-process communication with pipes

assign3: implement your own shell!
Learning Goals

• Understand how a process is cloned and run by the OS
• Learn how to use `waitpid()` to wait for a child process to finish.
• Understand how to use `execvp()` to run a new program within a process.
• See how a shell is implemented using `fork + execvp + waitpid`
Plan For Today

• Recap: fork()
• Cloning Processes
• waitpid() and waiting for child processes
• Demo: waiting for children

```bash
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```
Plan For Today

- Recap: fork()
- Cloning Processes
- `waitpid()` and waiting for child processes
- Demo: waiting for children

```bash
cp -r /afs/ir/class/cs111/lecture-code/lect9
```
fork()

A system call that creates a new child process

• The "parent" is the process that creates the other "child" process
• From then on, both processes are running the code after the fork
• The child process is identical to the parent, except:
  • it has a new Process ID (PID)
  • for the parent, fork() returns the PID of the child; for the child, fork() returns 0
  • fork() is called once, but returns twice

```c
pid_t pidOrZero = fork();
// both parent and child run code here onwards
printf("This is printed by two processes.\n");
```
fork()

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

$ ./intro-fork
Hello from process 29686! (parent 29351)
Bye from process 29686! (parent 29351)
Bye from process 29687! (parent 29686)

$ ./intro-fork
Hello from process 29688! (parent 29351)
Bye from process 29689! (parent 29688)
Bye from process 29688! (parent 29351)

• The parent of the original process is the *shell* - the program that you run in the terminal.
• The ordering of the parent and child output is *up to the OS!*
Which of these outputs is **not** possible?

```c
// Assume parent PID 111, child PID 112
pid_t pidOrZero = fork();
printf("hello, world!\n");
printf("goodbye! (fork returned %d)\n", pidOrZero);
```

<table>
<thead>
<tr>
<th></th>
<th>A)</th>
<th></th>
<th>C)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>hello, world!</td>
<td>hello, world!</td>
<td>hello, world!</td>
</tr>
<tr>
<td></td>
<td>hello, world!</td>
<td>goodbye! (fork returned 0)</td>
<td>goodbye! (fork returned 112)</td>
</tr>
<tr>
<td></td>
<td>goodbye! (fork returned 0)</td>
<td>goodbye! (fork returned 112)</td>
<td>goodbye! (fork returned 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B)</th>
<th></th>
<th>D)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>hello, world!</td>
<td>hello, world!</td>
<td>hello, world!</td>
</tr>
<tr>
<td></td>
<td>hello, world!</td>
<td>goodbye! (fork returned 112)</td>
<td>goodbye! (fork returned 112)</td>
</tr>
<tr>
<td></td>
<td>goodbye! (fork returned 112)</td>
<td>goodbye! (fork returned 0)</td>
<td>goodbye! (fork returned 0)</td>
</tr>
<tr>
<td></td>
<td>goodbye! (fork returned 0)</td>
<td>hello, world!</td>
<td>hello, world!</td>
</tr>
</tbody>
</table>

**Respond on pollEv:** [pollev.com/cs111](http://pollev.com/cs111) or text CS111 to 22333 once to join.
Which of these outputs is *not* possible?
Plan For Today

• Recap: fork()
• Cloning Processes
• `waitpid()` and waiting for child processes
• Demo: waiting for children

```bash
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```
What happens to variables/addresses?

```c
int main(int argc, char *argv[]) {
    char str[128];
    strcpy(str, "Hello");
    printf("str's address is %p\n", str);
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // The child should modify str
        printf("I am the child. str's address is %p\n", str);
        strcpy(str, "Howdy");
        printf("I am the child and I changed str to %s. str's address is still %p\n", str, str);
    } else { // The parent should sleep and print out str
        printf("I am the parent. str's address is %p\n", str);
        printf("I am the parent, and I'm going to sleep for 2sec.\n");
        sleep(2);
        printf("I am the parent. I just woke up. str's address is %p, and its value is %s\n", str, str);
    }
    return 0;
}
```
Process Clones

How can the parent and child use the same address to store different data?

• Each program thinks it is given all memory addresses to use
• The operating system maps these *virtual* addresses to *physical* addresses
• When a process forks, its virtual address space stays the same
• The operating system will map the child's virtual addresses to different physical addresses than for the parent

$ ./fork-copy
str's address is 0x7fffc8cfa9990
I am the parent. str's address is 0x7fffc8cfa9990
I am the parent, and I'm going to sleep for 2sec.
I am the child. str's address is 0x7fffc8cfa9990
I am the child and I changed str to Howdy. str's address is still 0x7fffc8cfa9990
I am the parent. I just woke up. str's address is 0x7fffc8cfa9990, and its value is Hello
Isn't it expensive to make copies of all memory when forking?

- The operating system only *lazily* makes copies.
- It will have them share physical addresses until one of them changes its memory contents to be different than the other.
- This is called *copy on write* (only make copies when they are written to).
Plan For Today

• **Recap**: `fork()`
• Cloning Processes
• **`waitpid()`** and waiting for child processes
• **Demo**: waiting for children

```bash
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```
It would be nice if there was a function we could call that would "stall" our program until the child is finished.
A system call that a parent can call to wait for its child to exit:

`pid_t waitpid(pid_t pid, int *status, int options);`

- **pid**: the PID of the child to wait on (we'll see other options later)
- **status**: where to put info about the child's termination (or NULL)
- **options**: optional flags to customize behavior (always 0 for now)
- the function returns when the specified **child process** exits
- the return value is the PID of the child that exited, or -1 on error (e.g. no child to wait on)
- If the child process has already exited, this returns immediately - otherwise, it blocks
```c
// waitpid.c
int main(int argc, char *argv[]) {
    printf("Before.\n");
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        sleep(2);
        printf("I (the child) slept and the parent waited for me.\n");
    } else {
        pid_t result = waitpid(pidOrZero, NULL, 0);
        printf("I (the parent) finished waiting for the child. This always prints last.\n");
    }
    return 0;
}
```

Before.
I (the child) slept and the parent waited for me.
I (the parent) finished waiting for the child. This always prints last.
I'm the child, and the parent will wait up for me.
Child exited with status 111.
... int status;
int result = waitpid(pid, &status, 0);
if (WIFEXITED(status)) {
    printf("Child exited with status %d.\n", WEXITSTATUS(status));
} else {
    printf("Child terminated abnormally.\n");
}
...

Provided macros (see man page for full list) let us extract info from the status.

- **WIFEXITED** – check if child terminated normally
- **WEXITSTATUS** – get exit status of child

This output will be the same every time! The parent will always wait for the child to finish before continuing.
Another benefit of `waitpid`: it cleans up the state of the terminated child process

- A process that finished but hasn’t yet been waited on by its parent is called a zombie 🧟.

- Zombies take up system resources (until they are ultimately cleaned up later by the OS). Therefore, a parent process should always wait on its children processes.

- If a child is still running, `waitpid` in the parent will block until the child finishes, and then clean it up. If a child process is a zombie, `waitpid` will return immediately and clean it up.

- Child processes whose parent process terminates without waiting on them get the `init` process (PID 1) as their parent.
Make sure to clean up after your zombie children.
(wait, what?)
Plan For Today

- Recap: fork()
- Cloning Processes
- `waitpid()` and waiting for child processes
- Demo: waiting for children

```bash
cp -r /afs/ir/class/cs111/lecture-code/lect9 .
```
Waiting for Children

**Problem:** if we have multiple children and want to wait on all of them, in what order do we wait on them to finish?

*Ideally we could say “wait until one of my children finishes”.*

- A parent can pass `-1` as the PID to `waitpid` to wait on any of its children.
- **Key Idea:** the children may terminate in any order!
- If `waitpid` returns `-1` and sets `errno` to `ECHILD`, this means there are no more children.

Let’s see a demo!

```bash
> reap-as-they-exit.c
```
Recap

• Recap: fork()
• Cloning Processes
• `waitpid()` and waiting for child processes
• Demo: waiting for children

Lecture 9 takeaway:
processes can be run by the OS in any order. `waitpid` lets a parent process wait for a child process to finish.

Next time: making our own shell, and how to have multiple processes communicate with pipes.