Lecture 3: Intro to Multiprocessing

Note: Reading these lecture notes is not a substitute for watching the lecture. I frequently go off script, and you are responsible for understanding everything I talk about in lecture unless I specify otherwise.

Creating processes

When the `fork()` syscall is invoked, a new process is created as an *exact replica* of the original process. The original process's virtual address space and registers are cloned as exact copies, as some sort of "process meiosis." As such, `fork` gets called once but returns twice; the child process continues execution from immediately after the `fork` call, just as the parent does.

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

The above code prints one line of `Greetings`, but two `Bye-bye` lines (one from the parent, one from the child).

```
Greetings from process 31384! (parent 27623)
Bye-bye from parent process 31384! (parent 27623)
Bye-bye from child process 31385! (parent 31384)
```

`fork` returns a number (of type `pid_t`). In the parent process, this is the PID of the new child process; in the child, it returns 0. (This is the only real difference in execution between the `parent` and `child` after the fork call.) We can use this to easily distinguish between the two processes:

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

`fork` *so thoroughly* duplicates the process memory that it *even* duplicates the random seed of a process (used to generate random numbers in the `random` function). One might think the following process prints 3 random numbers, but it only prints two:

```c
int main(int argc, char *argv[]) {
    printf("%ld\n", random());
    fork();
    printf("%ld\n", random());
    return 0;
}
```

```
1804289383
846930886
846930886
```

Scheduling

Consider the following program, which prints a letter, forks, and continues the loop:

```c
int main(int argc, char *argv[]) {
    const char *letters = "abcd";
    for (size_t i = 0; i < strlen(letters); i++) {
        printf("%c\n", letters[i]);
        fork();
    }
    return 0;
}
```

You might reasonably guess that the program outputs the following:

```
a
b
b
c
c
c
c
d
d
d
d
d
d
d
d
```

However, the order is not necessarily preserved. I get a different ordering of the letters every time, but this is one example output:

```
a
b
c
b
d
c
d
c
d
d
c
d
d
d
d
```

This is the effect of the **process scheduler** at work. The above code creates 8 processes, but we may only have 2 CPU cores with which to run those processes. In order to provide the illusion of running many processes simultaneously, the operating system scheduler does

the following:

- A process is allowed to run on a particular CPU core. After some short interval – a handful of microseconds – the OS pauses the process, copies the contents of registers into a *process control block*, and adds that data structure to a queue of running processes, called the *ready queue*.
- The OS then selects another process from the ready queue, loads the saved registers back into the CPU, and resumes execution of that process as if it had never stopped.
- Occasionally, a process needs to wait for something (e.g. it calls `sleep`, or it waits for a network request to come in, or it waits for a different process to do something). In this case, the process is removed from the ready queue and is instead moved to the *blocked set*. From there, the process isn't considered for scheduling. (Eventually, it is moved from the blocked set back to the ready queue when the thing it was waiting for is ready.) We will talk more about these situations in the next few lectures.

Note that the ready queue isn't a simple ordered queue; we may have high-priority processes that should get more CPU time. The scheduler employs a sophisticated algorithm to balance the needs of various processes, and, as a result, processes may not run in the order you expect them to. You are never given any guarantees about process scheduling, other than the fact that your process will be scheduled and will be executed eventually.

Basics of synchronization: the `waitpid` syscall

The `waitpid` system call can be used to wait until a particular child process is finished executing. (It's actually a more versatile syscall than that, and we will discuss its various uses next week, but consider this basic use case for now.)

In the following code, a process forks, and then the parent process waits for the child to exit:

```
int main(int argc, char *argv[]) {
    pid_t pid = fork();
    if (pid == 0) {
        // Child process
        sleep(1);
        printf("CHILD: Child process exiting...\n");
        return 0;
    }

    // Parent process
```

```
    printf("PARENT: Waiting for child process...\n");
    waitpid(pid, NULL, 0);
    printf("PARENT: Child process exited!\n");
    return 0;
}
```

Note: waitpid can only be called on *direct child processes* (not parent processes, or grandchild processes, or anything else).

Getting the return code from a process

The number returned from `main` is the *return code* or *exit status code* of a process. We can pass a second argument to `waitpid` to get information about the child process's execution, including its return code:

```
int main(int argc, char *argv[]) {
    pid_t pid = fork();
    if (pid == 0) {
        // Child process
        sleep(1);
        printf("CHILD: Child process exiting...\n");
        return 0;
    }

    // Parent process
    printf("PARENT: Waiting for child process...\n");
    int status;
    waitpid(pid, &status, 0);
    if (WIFEXITED(status)) {
        printf("PARENT: Child process exited with return code %d!\n",
               WEXITSTATUS(status));
    } else {
        printf("PARENT: Child process terminated abnormally!\n");
    }
    return 0;
}
```

We can now modify the `return 0;` of the child code to return some other number, or even

to segfault (in which case, `WIFEXITED(status)` will return false).

Calling waitpid without a specific child PID

You can call waitpid passing `-1` instead of a child's PID, and it will wait for *any* child process to finish (and subsequently return the PID of that process). If there are no child processes remaining, `waitpid` returns -1 and sets the global variable `errno` to `ECHILD` (to be specific about the "error condition." It can return -1 for other reasons, such as passing an invalid 3rd argument.)

This example creates several processes without keeping track of their PIDs, then calls `waitpid` until the parent has no more child processes that it hasn't already called `waitpid` on:

```
int main(int argc, char *argv[]) {
    for (size_t i = 0; i < 8; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            // Child process
            return 110 + i;
        }
    }

    while (true) {
        int status;
        pid_t pid = waitpid(-1, &status, 0);
        if (pid == -1) break;
        printf("Process %d exited with status %d.\n", pid,
               WEXITSTATUS(status));
    }
    assert(errno == ECHILD);
    return 0;
}
```

This calls `waitpid` a total of 9 times (it returns child PIDs 8 times, then returns -1 to indicate that there are no remaining children).

When `-1` is passed as the first argument, waitpid returns children in a somewhat arbitrary order. If several child processes have exited by the time you call waitpid, it will choose an

arbitrary child from that set. Otherwise, if you call waitpid before any child processes have stopped, it will wait for at least one of the running children to exit.

waitpid and scheduling

To be clear, waitpid does ***not*** influence the scheduling of processes. Calling waitpid on a process does not tell the OS, “hey, I am waiting on this process, so please give it higher priority.” It simply blocks the parent process until the specified child process has finished executing.

What’s the point of all of this?

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). On Monday, we’ll talk about a third reason: starting executables from disk.