Lecture 04: Creating and Coordinating Processes

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 as 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 *multiprocessing*, 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.

Lecture 04: Creating and Coordinating Processes

- **New system call: `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 right here.

```
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;
}
```

  - Here's the output of two consecutive runs of the above program.

```
myth60$ ./basic-fork
Greetings from process 29686! (parent 29351)
Bye-bye from process 29686! (parent 29351)
Bye-bye from process 29687! (parent 29686)
myth60$ ./basic-fork
Greetings from process 29688! (parent 29351)
Bye-bye from process 29688! (parent 29351)
Bye-bye from process 29689! (parent 29688
```

Lecture 04: Creating and Coordinating Processes

- `fork` is called once, but it returns twice.
  - `fork` knows how to clone the calling process, synthesize a nearly identical copy of it, and schedule the copy to run as if it's been running all along.
    - Think of it as a form of process meiosis, where one process becomes twins.
    - All segments (data, bss, init, stack, heap, text) are faithfully replicated to form an independent, protected virtual address space.
    - All open file descriptors are replicated, and these copies are donated to the clone.
  - As a result, the output of our program is the output of two processes.
    - We should expect to see a single greeting but two separate bye-byes.
    - Each bye-bye is inserted into the console by two different processes. The OS's process scheduler dictates whether the child or the parent gets to print its bye-bye first.
  - `getpid` and `getppid` return the process id of the caller and the process id of the caller's parent, respectively.

Lecture 04: Creating and Coordinating Processes

- Here's why the program output makes sense:
  - Process ids are generally assigned consecutively. That's why 29686 and 29687 are relevant to the first run, and why 29688 and 29689 are relevant to the second.
  - The 29351 is the pid of the terminal itself, and you can see that the initial `basic-fork` processes—with pids of 29686 and 29688—are direct child processes of the terminal. The output tells us so.
  - The clones of the originals are assigned pids of 29687 and 29689, and the output is clear about the parent-child relationship between 29686 and 29687, and then again between 29688 and 29689.

Lecture 04: Creating and Coordinating Processes

- Differences between parent calling `fork` and child generated by it:
  - The most obvious difference is that each gets its own 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 to easily self-identify as the child.
    - 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).

Lecture 04: Creating and Coordinating Processes

- 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 at the `fork`.
  - `info inferiors`
    - This lists the processes that `gdb` has captured.
  - `inferior X`
    - Switch to a different process.
  - `detach inferior X`
    - Tell `gdb` to stop watching the process before continuing it
  - You can see an entire debugging session on the `basic-fork` program right here.

Lecture 04: Creating and Coordinating Processes

- **`fork` so far:**
  - **`fork`** is a system call that spawns an almost complete duplicate of the current process.
  - In the parent process, the return value of **`fork`** is the child's `pid`, and in the child, the return value is 0. This enables both the parent and the child to determine which process they are.
  - **All** data segments are replicated. Aside from checking the return value of **`fork`**, there is virtually no difference in the two processes, and they both continue after **`fork`** as if they were the original process.
  - There is **no** default sharing of data between the two processes, though the parent process can **`wait`** (more below) 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.

Lecture 04: Creating and Coordinating Processes

- **Second example: A tree of `fork` calls**
  - 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`**. The full program can be viewed right here.

```
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\n", kTrail[i]);
    pid_t pid = fork();
    assert(pid >= 0);
  }
  return 0;
}
```

Lecture 04: Creating and Coordinating Processes

- **Second example: A tree of `fork` calls**
  - Two samples runs on the right
  - Reasonably obvious: A single `a` is printed by the soon-to-be-great-grandaddy 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.
  - 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?

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

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

Lecture 04: Creating and Coordinating Processes

- **Third example: Synchronizing between parent and child using `waitpid`**
  - `waitpid` can be used to temporarily block one process until a child process exits.

```
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, 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.

Lecture 04: Creating and Coordinating Processes

- **Third example: Synchronizing between parent and child using `waitpid`**
  - Consider the following program, which is more representative of how `fork` really gets used in practice (full program, with error checking, is right here):

```
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;
  }
 }
```

Lecture 04: Creating and Coordinating Processes

- **Third example: Synchronizing between parent and child using `waitpid`**
  - In practice, the output is the same every single time the above program is executed.

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

  - The implementation directs the child process one way, the parent another.
  - The parent process uses **`waitpid`** to wait for the child to complete.
  - The parent lifts child exit information out of the **`waitpid`** call, and uses the **`WIFEXITED`** macro to confirm the process exited normally and uses **`WEXITSTATUS`** to to produce the child's return value (which we can see is, and should be, 110).
  - In theory, the child could print **`"After."`** and its **`"I'm the child..."`** line before the parent even returns from its **`fork`** call, but the OS so heavily biases toward the process calling **`fork`** to continue running that I've never seen that happen in practice.

Lecture 04: Creating and Coordinating Processes

- **Synchronizing between parent and child using `waitpid`**
  - This next example is more of a brain teaser, but it illustrates just how deep a clone the process created by **`fork`** really is (full program, with more error checking, is right here).

```
int main(int argc, char *argv[]) {
  printf("I'm unique and just get printed once.\n");
  bool parent = fork() != 0;
  if ((random() % 2 == 0) == parent) sleep(1); // force exactly one of the two to sleep
  if (parent) waitpid(pid, NULL, 0); // parent shouldn't exit until child has finished
  printf("I get printed twice (this one is being printed from the %s).\n",
         parent  ? "parent" : "child");
  return 0;
}
```

  - The code emulates a coin flip to instruct exactly one of the two processes to sleep for a second, which is more than enough time for the child process to finish.
  - The parent waits for the child to exit before it allows itself to exit. It's akin to the parent not being able to fall asleep until he/she knows the child has, and it's emblematic of the types of synchronization directives we'll be seeing a lot of this quarter.
  - The final **`printf`** gets executed twice. The child is always the first to execute it, because the parent is blocked in its **`waitpid`** call until the child executes in full.

Lecture 04: Creating and Coordinating Processes

- **Spawning and synchronizing with multiple child processes**
  - A parent can call **`fork`** multiple times, provided it reaps the child processes (via **`waitpid`**) once they exit. If we want to reap processes as they exit without concern for the order they were spawned, then this does the trick (full program checking right here):

```
int main(int argc, char *argv[]) {
  for (size_t i = 0; i < 8; i++) {
    if (fork() == 0) exit(110 + i);
  }
  while (true) {
    int status;
    pid_t pid = waitpid(-1, &status, 0);
    if (pid == -1) { assert(errno == ECHILD); break; }
    if (WIFEXITED(status)) {
      printf("Child %d exited: status %d\n", pid, WEXITSTATUS(status));
    } else {
      printf("Child %d exited abnormally.\n", pid);
    }
  }
  return 0;
}
```

Lecture 04: Creating and Coordinating Processes

- **Spawning and synchronizing with multiple child processes**
  - Note we feed a -1 as the first argument to `waitpid`. That -1 states we want to hear about **any** child as it exits, and pids are returned in the order their processes finish.
  - Eventually, all children exit and `waitpid` correctly returns -1 to signal there are no more processes under the parent's jurisdiction.

  - When `waitpid` returns -1, it sets a global variable called `errno` to the constant `ECHILD` to signal `waitpid` returned -1 because all child processes have terminated. That's the "error" we want.

```
myth60$ ./reap-as-they-exit
Child 1209 exited: status 110
Child 1210 exited: status 111
Child 1211 exited: status 112
Child 1216 exited: status 117
Child 1212 exited: status 113
Child 1213 exited: status 114
Child 1214 exited: status 115
Child 1215 exited: status 116
myth60$
```

```
myth60$ ./reap-as-they-exit
Child 1453 exited: status 115
Child 1449 exited: status 111
Child 1448 exited: status 110
Child 1450 exited: status 112
Child 1451 exited: status 113
Child 1452 exited: status 114
Child 1455 exited: status 117
Child 1454 exited: status 116
myth60$
```

Lecture 04: Creating and Coordinating Processes

- **Spawning and synchronizing with multiple child processes**
  - We can do the same thing we did in the first program, but monitor and reap the child processes in the order they are forked.
  - Check out the abbreviated program below (full program with error checking right here):

```
int main(int argc, char *argv[]) {
  pid_t children[8];
  for (size_t i = 0; i < 8; i++) {
    if ((children[i] = fork()) == 0) exit(110 + i);
  }
  for (size_t i = 0; i < 8; i++) {
    int status;
    pid_t pid = waitpid(children[i], &status, 0);
    assert(pid == children[i]);
    assert(WIFEXITED(status) && (WEXITSTATUS(status) == (110 + i)));
    printf("Child with pid %d accounted for (return status of %d).\n",
           children[i], WEXITSTATUS(status));
  }
  return 0;
}
```

Lecture 04: Creating and Coordinating Processes

- **Spawning and synchronizing with multiple child processes**
  - This version spawns and reaps processes in some first-spawned-first-reaped manner.
  - The child processes aren't required to exit in FSFR order.
  - In theory, the first child thread could finish last, and the reap loop could be held up on its very first iteration until the first child really is done. But the process zombies–yes, that's what they're called–are reaped in the order they were forked.

  - Below is a sample run of the `reap-in-fork-order` executable. The pids change between runs, but even those are guaranteed to be published in increasing order.

```
myth60$ ./reap-as-they-exit
Child with pid 4689 accounted for (return status of 110).
Child with pid 4690 accounted for (return status of 111).
Child with pid 4691 accounted for (return status of 112).
Child with pid 4692 accounted for (return status of 113).
Child with pid 4693 accounted for (return status of 114).
Child with pid 4694 accounted for (return status of 115).
Child with pid 4695 accounted for (return status of 116).
Child with pid 4696 accounted for (return status of 117).
myth60$
```