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](#).

```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;
}
```

- 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 virtually identical copy of it, and schedule the copy as if it were running all along.
    - All segments (data, bss, init, stack, heap, text) are faithfully replicated.
    - All open file descriptors are duplicated and 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 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 than `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).
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.

```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\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?
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.

```c
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.
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](#)):

```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;
    }
}
```
Lecture 04: Creating and Coordinating Processes

- **Third example: Synchronizing between parent and child using `waitpid`**
  - 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 correctly waits for the child to complete using `waitpid`.
- The parent also 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 we can see is, and should be, 110).
Lecture 04: Creating and Coordinating Processes

- Final example: 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).

```c
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 seduce 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 `everything`. 