CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 06: Process Synchronization

- System Calls Introduced Last Time

```
pid_t fork();
```

  - The **fork** system call is used to create new processes. Calling **fork** from one process has the side effect of creating a second that's an exact replica of the first. Restated, somewhere in the middle of the **fork** call, one process becomes two nearly identical processes. Each continues from the same exact assembly code instruction with the same exact state of computer memory. All file descriptors are replicated as well.
    - Are there differences? Yes.
      - The two processes are nearly identical at the time of the split, but they each have different process ids, and their executions may diverge after the split.
      - The original call to **fork** returns the pid of the new process. But **fork** returns 0 from the new process.

Lecture 06: Process Synchronization

- System Calls Introduced Last Time

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

  - The waitpid system call is used to block a process until a child process finishes (or stops, or continues after being stopped).
    - The first argument specifies the *wait set*, which for now is just the pid of the child process that needs to complete before **waitpid** can return.
    - The second argument supplies the address of an integer where process termination information can be placed (or we can pass **NULL** if we don't need the information).
    - The third argument is a collection of bitwise-or'ed flags we'll study later. For the moment, we'll just go with 0 as the required parameter value, which means that **waitpid** should only return when the process with the given pid exits.
    - The return value is the pid of the process that successfully exited, or -1 if **waitpid** fails (perhaps because the pid is invalid, or you passed in other bogus arguments).

Lecture 06: Process Synchronization

- 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).
  - The parent process correctly waits for the child to complete using **waitpid**.

```
 1 int main(int argc, char *argv[]) {
 2   printf("Before.\n");
 3   pid_t pid = fork();
 4   printf("After.\n");
 5   if (pid == 0) {
 6     printf("I am the child, parent will wait for me.\n");
 7     return 110;
 8   } else {
 9     int status;
10     waitpid(pid, &status, 0);
11     if (WIFEXITED(status)) {
12       printf("Child exited with status %d.\n",
13              WEXITSTATUS(status));
14     } else {
15       printf("Child terminated abnormally.\n");
16     }
17     return 0;
18   }
19 }
```

- 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 (110 as expected).
- The **waitpid** call also donates child process-oriented resources back to the system.

Lecture 06: Process Synchronization

- The output on the left is most likely every single time the program is executed.
  - The parent generally continues running without halting when it calls **fork** (since all **fork** does is set up new data structures for the new process, return, and carry on).
  - However, it is theoretically possible to get the output on the right if the child runs as soon as it comes into existence.

```
myth59$ ./separate
Before.
After.
After.
I am the child, parent will wait for me.
Child exited with status 110.
myth59$
```

```
myth59$ ./separate
Before.
After.
I am the child, parent will wait for me.
After.
Child exited with status 110.
myth59$
```

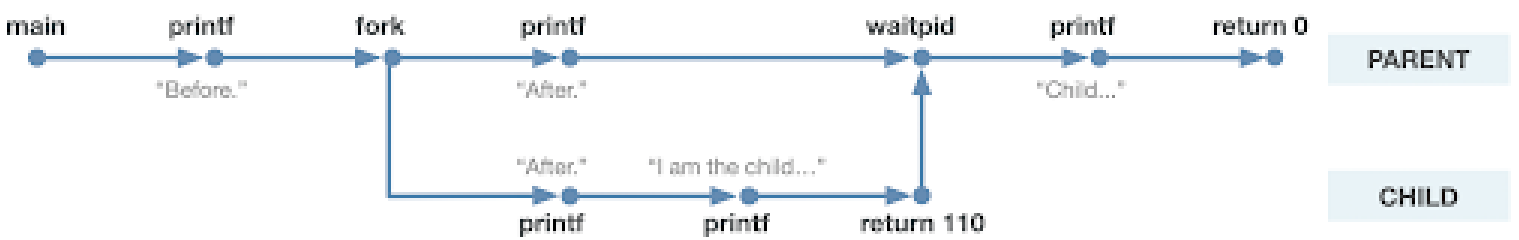

Illustration courtesy of Roz Cyrus.

Lecture 06: Process Synchronization

- This example is more of a brain teaser, but it illustrates just how **deep** a clone the process created by **`fork`** really is (full program is also online right here).

```
1  int main(int argc, char *argv[]) {
2    printf("I get printed once!\n");
3    pid_t pid = fork(); // returns 0 within child, returns pid of child within fo
4    bool parent = pid != 0;
5    if ((random() % 2 == 0) == parent) {
6      sleep(1); // force exactly one of the two to sleep
7      printf("Ah, naps are the best!\n"); // brag
8    }
9    if (parent) waitpid(pid, NULL, 0); // parent shouldn't exit until it knows it
10   printf("I get printed twice (this one is being printed from the %s).\n",
11          parent  ? "parent" : "child");
12   return 0;
13 }
```

  - 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.
    - Question: Why can't both parent and child call **sleep(1)** in any given run?
  - The parent waits for the child to exit before it allows itself to exit. Whether or not the parent sleeps for one second or the child sleeps for one second is up to the random number generator.
  - 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 06: Process Synchronization

- A process can call **`fork`** multiple times, provided it reaps the child processes (via **`waitpid`**) once they exit. Note that we reap processes as they exit without worrying about the order they were spawned! Full program is also online right here.

```
 1 int main(int argc, char *argv[]) {
 2   for (size_t i = 0; i < 8; i++) {
 3     pid_t pid = fork();
 4     assert(pid >= 0);
 5     if (pid == 0) exit(110 + i);
 6   }
 7   for (size_t i = 0; i < 8; i++) {
 8     int status;
 9     pid_t pid = waitpid(-1, &status, 0);
10     assert(pid > 0);
11     if (WIFEXITED(status)) {
12       printf("Child %d exited: status %d\n", pid, WEXITSTATUS(status));
13     } else {
14       printf("Child %d exited abnormally.\n", pid);
15     }
16   }
17   assert(waitpid(-1, NULL, 0) == -1 && errno == ECHILD);
18   return 0;
19 }
```

  - We feed -1 as the first argument to **`waitpid`**. The -1 means want to hear about **any** child as it exits, so that pids are returned in the order their processes finish.
  - 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.

Lecture 06: Process Synchronization

- Presented below are two sample runs on myth and two samples runs on cplayground.

```
myth60$ ./reap-as-they-exit
Child 3778291 exited: status 110
Child 3778292 exited: status 111
Child 3778293 exited: status 112
Child 3778294 exited: status 113
Child 3778295 exited: status 114
Child 3778296 exited: status 115
Child 3778297 exited: status 116
Child 3778298 exited: status 117
```

```
myth60$ ./reap-as-they-exit
Child 3778306 exited: status 110
Child 3778307 exited: status 111
Child 3778308 exited: status 112
Child 3778309 exited: status 113
Child 3778310 exited: status 114
Child 3778311 exited: status 115
Child 3778312 exited: status 116
Child 3778313 exited: status 117
```

***myth**: The OS appears to be fairly regimented and round-robin in how it selects child processes to run. I ran reap-as-they-exit some 100 times and always got this output.*

```
cplayground$ ./reap-as-they-exit
Child 12 exited: status 110
Child 13 exited: status 111
Child 14 exited: status 112
Child 15 exited: status 113
Child 16 exited: status 114
Child 18 exited: status 116
Child 19 exited: status 117
Child 17 exited: status 115
```

```
cplayground$ ./reap-as-they-exit
Child 11 exited: status 110
Child 12 exited: status 111
Child 13 exited: status 112
Child 15 exited: status 114
Child 14 exited: status 113
Child 17 exited: status 116
Child 18 exited: status 117
Child 16 exited: status 115
```

***cplayground**: The OS looks to randomly schedule child processes, allowing them to complete and be reaped in a somewhat unpredictable order.*

Lecture 06: Process Synchronization

- We can do the same thing we did in the previous program, but this time 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):

```
 1 int main(int argc, char *argv[]) {
 2   pid_t children[8];
 3   for (size_t i = 0; i < 8; i++) {
 4     if ((children[i] = fork()) == 0) exit(110 + i);
 5   }
 6   for (size_t i = 0; i < 8; i++) {
 7     int status;
 8     pid_t pid = waitpid(children[i], &status, 0);
 9     assert(pid == children[i]);
10     assert(WIFEXITED(status) && (WEXITSTATUS(status) == (110 + i)));
11     printf("Child with pid %d accounted for (return status of %d).\n",
12            children[i], WEXITSTATUS(status));
13   }
14   return 0;
15 }
```

  - Note that the child processes aren't required to **exit** in first-spawned 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 child process zombies—yes, that's what they're called—are reaped in the order they were forked.
  - **waitpid**, just prior to returning, deallocates a zombie's process control block and removes all traces of it from the OS.

Lecture 06: Process Synchronization

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

```
 1 int main(int argc, char *argv[]) {
 2   pid_t children[8];
 3   for (size_t i = 0; i < 8; i++) {
 4     if ((children[i] = fork()) == 0) exit(110 + i);
 5   }
 6   for (size_t i = 0; i < 8; i++) {
 7     int status;
 8     pid_t pid = waitpid(children[i], &status, 0);
 9     assert(pid == children[i]);
10     assert(WIFEXITED(status) && (WEXITSTATUS(status) == (110 + i)));
11     printf("Child with pid %d accounted for (return status of %d).\n",
12            children[i], WEXITSTATUS(status));
13   }
14   return 0;
15 }
```

```
myth60$ ./reap-as-they-exit
Child with pid 3787749 accounted for (return status of 110).
Child with pid 3787750 accounted for (return status of 111).
Child with pid 3787751 accounted for (return status of 112).
Child with pid 3787752 accounted for (return status of 113).
Child with pid 3787753 accounted for (return status of 114).
Child with pid 3787754 accounted for (return status of 115).
Child with pid 3787755 accounted for (return status of 116).
Child with pid 3787756 accounted for (return status of 117).
myth60$
```

Lecture 06: Process Transformation via execvp

- Enter the **execvp** system call!
  - **execvp** effectively reboots a process to run a different program from scratch. Here is the full prototype:

```
int execvp(const char *path, char *argv[]);
```

    - **path** is relative or absolute pathname of the executable to be invoked.
    - **argv** is the argument vector that should be funneled through to the new executable's **main** function.
    - **path** and **argv[0]** generally end up being the same exact string.
    - If **execvp** fails to cannibalize the process and install a new executable image within it, it returns -1 to express failure.
    - If **execvp** succeeds, it 😱 never returns 😱.
  - **execvp** has many variants (**execle**, **execlp**, and so forth. Type **man execvp** to see all of them). We typically rely on **execvp** in this course.

Lecture 06: Process Transformation

- First example using **execvp**? An implementation **mysystem** to imitate a similar **libc** function called **system**.
  - Here we present our own implementation of the **mysystem** function, which executes the supplied **command** as if we typed it out in the terminal ourselves, ultimately returning once the surrogate **command** has finished.

```
1  static void mysystem(char *command) {
2    pid_t pid = fork();
3    if (pid == 0) {
4      char *arguments[] = {"/bin/sh", "-c", command, NULL};
5      execvp(arguments[0], arguments);
6      printf("Failed to invoked /bin/sh to execute the supplied command.\n");
7      exit(0);
8    }
9    int status;
10   waitpid(pid, &status, 0);
11 }
```

  - Disclaimer: **libc**'s **system** function is unambiguously dangerous when executed on arbitrary commands. Check out this stack overflow post for a lightweight discussion illustrating why. By extension, our **mysystem** function is unsafe too.
  - We're merely speaking of **system** here because it's a simple vehicle for learning how **execvp** does its job and how **fork**, **execvp**, and **waitpid** are combined in multiprocessing scenarios.

Lecture 06: Process Transformation

- Here's the implementation, with minimal error checking (the full version is right here):

```
1  static void mysystem(char *command) {
2    pid_t pid = fork();
3    if (pid == 0) {
4      char *arguments[] = {"/bin/sh", "-c", command, NULL};
5      execvp(arguments[0], arguments);
6      printf("Failed to invoked /bin/sh to execute the supplied command.\n");
7      exit(0);
8    }
9    int status;
10   waitpid(pid, &status, 0);
11 }
```

  - Instead of calling a subroutine to perform some task and waiting for it to complete, **`mysystem`** spawns a **child process** to execute that task and waits for it to complete.
    - Very often the functionality we want to invoke is packages in executable form, not library function form. When that's the case, it's common to use something like our **`mysystem`**.
  - We don't bother checking the return value of **`execvp`**, because we know that if it returns at all, it returns a -1. If that happens, we need to handle the error and make sure the child process terminates via an exposed **`exit(0)`** call.
  - Why not call **`execvp`** inside the parent and forgo the child process altogether? Because **`execvp`** would consume the calling process, and that's not what we want.

Lecture 06: Process Transformation

- Here's a test harness that we'll run during lecture to confirm our **`mysystem`** implementation is working as expected:

```
 1 static const size_t kMaxLine = 2048;
 2 int main(int argc, char *argv[]) {
 3   char command[kMaxLine];
 4   while (true) {
 5     printf("> ");
 6     fgets(command, kMaxLine, stdin);
 7     if (feof(stdin)) break;
 8     command[strlen(command) - 1] = '\0'; // overwrite '\n'
 9     mysystem(command);
10   }
11
12   printf("\n");
13   return 0;
14 }
```

  - **`fgets`** is an overflow-safe variant on **`scanf`** that knows to read everything up through and including the newline character.
    - The newline is retained, so we need to chomp it off before calling **`mysystem`**.