CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 07: Process Transformation

- System Call Introduced Last Time

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

  - **execvp** effectively reboots a process to run a different program from scratch.
    - **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.
  - Our first example was included in last Friday's slide deck, and we'll be working through that first.

Lecture 07: Process Transformation

- This **`mysystem`** function is just the first example where **`fork`**, **`execvp`**, and **`waitpid`** all work together to do something genuinely useful.
  - The test harness we used to exercise **`mysystem`** is operationally a miniature shell.
  - We need to continue implementing a few additional mini-shells to fully demonstrate how **`fork`**, **`waitpid`**, and **`execvp`** work in practice.
  - All of this is paying it forward to your fourth assignment, where you'll implement your own shell—we call it **`stsh`**, for Stanford shell—to imitate the functionality of the shell (c-shell aka **`csh`**, or bash-shell aka **`bash`**, or z-shell aka **`zsh`**, or tc-shell aka **`tcsh`**, etc. are all different shell implementations) you've been using since you started using Unix.

Lecture 07: Process Transformation

- Let's work through the implementation of a more sophisticated shell: the **`simplesh`**.
  - This is the best introductory example of **`fork`**, **`waitpid`**, and **`execvp`** that I can think of: a miniature shell not unlike those you've been using since the first time you logged into a **`myth`**.
  - **`simplesh`** operates as a read-eval-print loop—often called a *repl*—which itself responds to the many things we type in, typically by **forking off child processes**.
    - Each child process is initially a deep clone of the **`simplesh`** process.
    - Each child proceeds to replace its own image with the new one we specify, e.g. **`ls`**, **`cp`**, **find**, **make**, or even **`emacs`**.
    - As with traditional shells, a trailing ampersand—e.g. as with **`emacs &`**—is an instruction to execute the new process in the *background* without forcing the shell to wait for it to finish. That means we can launch other programs from the *foreground* before that background process finishes.
  - Our implementation of **`simplesh`** is presented on the next slide. Where helper functions don't rely on CS110 concepts, I omit their implementations (but describe them in adequate detail in lecture).

Lecture 07: Process Transformation

- Here's the core implementation of **`simplesh`** (full implementation is right here):

```
 1 int main(int argc, char *argv[]) {
 2   while (true) {
 3     char command[kMaxCommandLength + 1]; // room for \0 as well
 4     readCommand(command, kMaxCommandLength);
 5     char *arguments[kMaxArgumentCount + 1];
 6     int count = parseCommandLine(command, arguments, kMaxArgumentCount);
 7     if (count == 0) continue;
 8     if (strcmp(arguments[0], "quit") ==) break; // hardcoded builtin to exit shell
 9     bool isbg = strcmp(arguments[count - 1], "&") == 0;
10     if (isbg) arguments[--count] = NULL; // overwrite "&"
11     pid_t pid = fork();
12     if (pid == 0) execvp(arguments[0], arguments);
13     if (isbg) { // background process, don't wait for child to finish
14       printf("%d %s\n", pid, command);
15     } else {    // otherwise block until child process is complete
16       waitpid(pid, NULL, 0);
17     }
18   }
19   printf("\n");
20   return 0;
21 }
```

Lecture 07: Process Transformation without fork!

- **exargs** (type **man xargs** for the full read) is useful when one program is needed to programmatically generate the argument vector for a second.
  - **xargs** reads tokens from standard input (delimited by spaces and newlines).
  - **xargs** then appends those tokens to the end of its original argument list and executes the full list of arguments—original plus those read from standard input—as if we typed them all in by hand.
  - To illustrate the basic idea, consider the **factor** program, which prints out the prime factorizations of all of its numeric arguments, as with:

```
poohbear@myth62:~$ factor 720
720: 2 2 2 2 3 3 5
poohbear@myth62:~$ factor 9 16 2047 870037764750
9: 3 3
16: 2 2 2 2
2047: 23 89
870037764750: 2 3 3 5 5 5 7 7 7 7 11 11 11 11 11
poohbear@myth62:~$ printf "720" | ./xargs factor
720: 2 2 2 2 3 3 5
poohbear@myth62:~$ printf "2047 1000\n870037764750" | ./xargs factor 9 16
9: 3 3
16: 2 2 2 2
2047: 23 89
1000: 2 2 2 5 5 5
870037764750: 2 3 3 5 5 5 7 7 7 7 11 11 11 11 11
poohbear@myth62:~$
```

Lecture 07: Process Transformation without fork!

- Note that the first process in the pipeline—the **`printf`**—is a brute force representative of an executable capable of supplying or extending the argument vector of a second executable—in this case, **`factor`**—through **`xargs`**.
  - Of course, the two executables needn't be **`printf`** or **`factor`**; they can be anything that works.
  - If, for example, I'm interested in exposing how much code I wrote for my own **`assign2`** solution , I might use **`xargs`** to do this:

```
poohbear@myth62:~$ ls /usr/class/cs110/staff/master_repos/assign2/*.c | ./xargs wc
   78  1792    90 /usr/class/cs110/staff/master_repos/assign2/chksumfile.c
   35  1178   121 /usr/class/cs110/staff/master_repos/assign2/directory.c
  266  8015   111 /usr/class/cs110/staff/master_repos/assign2/diskimageaccess.c
   31   731    86 /usr/class/cs110/staff/master_repos/assign2/diskimg.c
   35  1193   144 /usr/class/cs110/staff/master_repos/assign2/file.c
   72  2751   134 /usr/class/cs110/staff/master_repos/assign2/inode.c
   33   987   152 /usr/class/cs110/staff/master_repos/assign2/pathname.c
   45  1287    91 /usr/class/cs110/staff/master_repos/assign2/unixfilesystem.c
  595 17934   152 total
```

  - For simplicity, we'll assume a working **pullAllTokens** function, which exhaustively pulls all content from the provided **istream**, tokenizes around newlines and whitespace, and populates the referenced **vector** with all tokens, in sequence.

```
static void pullAllTokens(istream& in, vector<string>& tokens);
```

Lecture 07: Process Transformation without fork!

- Here's our implementation of **xargs.cc**. Note that we're coding in C++, because the string processing is farcically easy compared compared to C.

```
 1 int main(int argc, char *argv[]) {
 2   vector<string> tokens;
 3   pullAllTokens(cin, tokens);
 4   char *xargsv[argc + tokens.size()];
 5   for (size_t i = 0; i < argc - 1; i++)
 6     xargsv[i] = argv[i + 1];
 7   for (size_t i = 0; i < tokens.size(); i++)
 8     xargsv[argc - 1 + i] = (char *) tokens[i].c_str();
 9   xargsv[argc + tokens.size() - 1] = NULL;
10   execvp(xargsv[0], xargsv);
11   cerr << xargsv[0] << ": command not found, so xargs can't do its job!" << endl;
12   return 0;
13 }
```

  - This is a rare example of a program that calls **execvp** without calling **fork** first.
    - The *real* program to be executed is supplied via **argv[1]**, and that's ultimately the executable we really want **xargs** to become.
    - The code preceding **execvp** is little more than argument vector construction.

Lecture 07: Interprocess Communication

- Introducing the **`pipe`** system call.
  - The **`pipe`** system call takes an uninitialized array of two integers—we'll call it **`fds`**—and populates it with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**.
  - Here's the prototype:

```
int pipe(int fds[]);
```

  - **`pipe`** is particularly useful for allowing parent processes to communicate with spawned child processes.
    - Recall that the file descriptor table of the parent is cloned across **fork** boundaries and preserved by **execvp** calls.
    - That means open file table entries referenced by the parent's **pipe** endpoints are also referenced by the child's copies of them. Neat!

Lecture 07: Interprocess Communication

- How does **`pipe`** work?
  - To illustrate how **`pipe`** works and how arbitrary data can be passed over from one process to a second, let's consider the following program (available for play right here):

```
 1 int main(int argc, char *argv[]) {
 2   int fds[2];
 3   pipe(fds);
 4   pid_t pid = fork();
 5   if (pid == 0) {
 6     close(fds[1]);
 7     char buffer[6];
 8     read(fds[0], buffer, sizeof(buffer)); // assume one call is enough
 9     printf("Read from pipe bridging processes: %s.\n", buffer);
10     close(fds[0]);
11     return 0;
12   }
13   close(fds[0]);
14   write(fds[1], "hello", 6);
15   waitpid(pid, NULL, 0);
16   close(fds[1]);
17   return 0;
18 }
```

Lecture 07: Interprocess Communication

- How do **`pipe`** and **`fork`** work together in this example?
  - The base address of a small integer array called **`fds`** is shared with the call to **`pipe`**.
  - **`pipe`** allocates two descriptors, setting the first to read from a resource and the second to write to that same resource. Think of this resource as an unnamed file that only the OS and its support for pipe know about.
  - **`pipe`** then plants copies of those two descriptors into indices 0 and 1 of the supplied array before it returns.
  - The **`fork`** call creates a child process, which itself inherits a shallow copy of the parent's **`fds`** array.
    - The reference counts in each of the two open file entries is promoted from 1 to 2 to reflect the fact that two descriptors—one in the parent, and a second in the child—reference each of them.
    - Immediately after the **`fork`** call, anything printed to **`fds[1]`** is readable from the parent's **`fds[0]`** *and* the child's **`fds[0]`**.
    - Similarly, both the parent and child are capable of publishing text to the same resource via their copies of **`fds[1]`**.

Lecture 07: Interprocess Communication

- How do **`pipe`** and **`fork`** work together in this example?
  - The parent closes **`fds[0]`** before it writes to anything to **`fds[1]`** to emphasize the fact that the parent has no need to read anything from the pipe.
  - Similarly, the child closes **`fds[1]`** before it reads from **`fds[0]`** to emphasize the fact that it has zero interest in publishing anything to the pipe. It's imperative all write endpoints of the pipe be closed if not being used, else the read end will never know if more text is to come or not.
  - For simplicity, I assume the one call to **`write`** in the parent presses all six bytes of **`"hello"`** (**`'\0'`** included) in a single call. Similarly, I assume the one call to **`read`** pulls in those same six bytes into its local **`buffer`** with just the one call.
  - I make a concerted effort to donate all resources back to the system before I exit. That's why I include as many **`close`** calls as I do in both the child and the parent before allowing them to exit.