Lecture 06: `execvp`, Pipes, and Interprocess Communication

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg

PDF of this presentation

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- First example using **`execvp`**? An implementation **`mysystem`** to emulate the behavior of the 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.
  - If the execution of **`command`** exits normally (either via an **`exit`** system call, or via a normal return statement from **`main`**), then our **`mysystem`** implementation should return that exact same exit value.
  - If the execution exits abnormally (e.g. it segfaults), then we'll assume it aborted because some signal was ignored, and we'll return that negative of that signal number (e.g. -11 for **`SIGSEGV`**).

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- Here's the implementation, with minimal error checking (the full version is right here, and a working version is on the next slide):

```
1  static int mysystem(const char *command) {
2    pid_t pid = fork();
3    if (pid == 0) {
4      char *arguments[] = {"/bin/sh", "-c", (char *) command, NULL};
5      execvp(arguments[0], arguments);
6      printf("Failed to invoke /bin/sh to execute the supplied command.");
7      exit(0);
8    }
9    int status;
10   waitpid(pid, &status, 0);
11   return WIFEXITED(status) ? WEXITSTATUS(status) : -WTERMSIG(status);
12 }
```

- Instead of calling a subroutine to perform some task and waiting for it to complete, **`mysystem`** spawns a **child process** to perform some task and waits for it to complete.
- 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 parent and forgo the child process altogether? Because **`execvp`** would consume the calling process, and that's not what we want.

Lecture 06: `execvp`, Pipes, and Interprocess Communication

https://cplayground.com/?p=squirrel-gnat-mongoose

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- Here's a test harness that we can run 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      printf("retcode = %d\n", mysystem(command));
10   }
11
12   printf("\n");
13   return 0;
14 }
```

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

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- The **`mysystem`** function is the first example I've provided 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 terminal.
  - We need to continue implementing a few additional mini-terminals 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 (**`csh`**, **`bash`**, **`zsh`**, **`tcsh`**, etc.) you've been using since you started using Unix.
- We'll introduce the notion of a pipe, the **`pipe`** and **`dup2`** system calls, and how they can be used to introduce communication channels between the different processes.

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- Let's work through the implementation of a more sophisticated shell: the **`simplesh`**.
  - This is the best example of **`fork`**, **`waitpid`**, and **`execvp`** I can think of: a miniature shell not unlike those you've been using since the day you first logged into a **`myth`** machine.
  - **`simplesh`** operates as a read-eval-print loop—often called a *repl*—which itself responds to the many things we type in by forking off child processes.
    - Each child process is initially a deep clone of the **`simplesh`** process.
    - Each proceeds to replace its own process image with the new one we specify, e.g. **`ls`**, **`cp`**, our own CS110 **`search`** (which we wrote during our second lecture), 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.
  - Implementation of **`simplesh`** is presented on the next slide. Where helper functions don't rely on CS110 concepts, I omit their implementations.

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- Here's the core implementation of **`simplesh`** (full implementation is right here, and you can run the code on the following slide):

```
1  int main(int argc, char *argv[]) {
2    while (true) {
3      char command[kMaxCommandLength + 1];
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 06: `execvp`, Pipes, and Interprocess Communication

https://cplayground.com/?p=fox-chimpanzee-hawk

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- Introducing the **`pipe`** system call.
  - The **`pipe`** system call takes an uninitialized array of two integers—let's 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.
    - That's because the file descriptor table of the parent is cloned, and that clone is installed in the child.
    - That means the open file table entries references by the parent's pipe endpoints are also referenced by the child's copies of them.

Lecture 06: `execvp`, Pipes, and 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 (which you can find here, or run on the next slide):

```
int main(int argc, char *argv[]) {
  int fds[2];
  pipe(fds);
  pid_t pid = fork();
  if (pid == 0) {
    close(fds[1]);
    char buffer[6];
    read(fds[0], buffer, sizeof(buffer));
    printf("Read from pipe bridging processes: %s.\n", buffer);
    close(fds[0]);
    return 0;
  }
  close(fds[0]);
  write(fds[1], "hello", 6);
  waitpid(pid, NULL, 0);
  close(fds[1]);
  return 0;
}
```

Lecture 06: `execvp`, Pipes, and Interprocess Communication

https://cplayground.com/?p=okapi-grasshopper-bear

Lecture 06: `execvp`, Pipes, and 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 draw from a resource and the second to publish to that same resource.
  - **`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 06: `execvp`, Pipes, and 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 interest in reading anything from the pipe.
  - Similarly, the child closes **`fds[1]`** before it reads from **`fds[0]`** to emphasize the fact that the 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.
  - As is the case with all programs, I make the 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.

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- Here's a more sophisticated example:
  - Using **`pipe`**, **`fork`**, **`dup2`**, **`execvp`**, **`close`**, and **`waitpid`**, we can implement the **`subprocess`** function, which relies on the following record definition and is implemented to the following prototype (full implementation of everything is right here):

```
1 typedef struct {
2   pid_t pid;
3   int supplyfd;
4 } subprocess_t;
5 subprocess_t subprocess(const char *command);
```

- The child process created by **`subprocess`** executes the provided **`command`** (assumed to be a **`'\0'`**-terminated C string) by calling **`"/bin/sh -c <command>"`** as we did in our **`mysystem`** implementation.
  - Rather than waiting for **`command`** to finish, **`subprocess`** returns a **`subprocess_t`** with the **`command`** process's **`pid`** and a single descriptor called **`supplyfd`**.
  - By design, arbitrary text can be published to the return value's **`supplyfd`** field with the understanding that that same data can be ingested verbatim by the child's **`stdin`**.

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- Let's first implement a test harness to illustrate how **`subprocess`** should work.
  - By understanding how **`subprocess`** works for us, we'll have an easier time understanding the details of its implementation.
  - Here's the program, which spawns a child process that reads from **`stdin`** and publishes everything it reads to its **`stdout`** in sorted order:

```
 1 int main(int argc, char *argv[]) {
 2   subprocess_t sp = subprocess("/usr/bin/sort");
 3   const char *words[] = {
 4     "felicity", "umbrage", "susurration", "halcyon",
 5     "pulchritude", "ablution", "somnolent", "indefatigable"
 6   };
 7   for (size_t i = 0; i < sizeof(words)/sizeof(words[0]); i++) {
 8     dprintf(sp.supplyfd, "%s\n", words[i]);
 9   }
10   close(sp.supplyfd);
11   int status;
12   pid_t pid = waitpid(sp.pid, &status, 0);
13   return pid == sp.pid && WIFEXITED(status) ? WEXITSTATUS(status) : -127;
14 }
```

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- Key features of the test harness:
  - The program creates a **`subprocess_t`** running **`sort`** and publishes eight fancy SAT words to **`supplyfd`**, knowing those words flow through the pipe to the child's **`stdin`**.
  - The parent shuts the **`supplyfd`** down by passing it to **`close`**. The reference count of the open file entry referenced by **`supplyfd`** is demoted from 1 to 0 with that **`close`** call. That sends an **EOF** to the process that tries in ingest data from the other end of the pipe.
  - The parent then blocks within a **`waitpid`** call until the child exits. When the child exits, the parent assumes all of the words have been printed in sorted order to **`stdout`**.

```
cgregg@myth60$ ./subprocess
ablution
felicity
halcyon
indefatigable
pulchritude
somnolent
susurration
umbrage
cgregg@myth60$
```

Lecture 06: `execvp`, Pipes, and Interprocess Communication

- Implementation of **`subprocess`** (error checking intentionally omitted for brevity):

```
subprocess_t subprocess(const char *command) {
  int fds[2];
  pipe(fds);
  subprocess_t process = { fork(), fds[1] };
  if (process.pid == 0) {
    close(fds[1]);
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    char *argv[] = {"/bin/sh", "-c", (char *) command, NULL};
    execvp(argv[0], argv);
  }
  close(fds[0]);
  return process;
}
```

  - The write end of the pipe is embedded into the **`subprocess_t`**. That way, the parent knows where to publish text so it flows to the read end of the pipe, across the parent process/child process boundary. This is bonafide interprocess communication.
  - The child process uses **`dup2`** to bind the read end of the pipe to its own standard input. Once the reassociation is complete, **`fds[0]`** can be closed.

Lecture 06: Signals

- Introduction to Signals
  - A **signal** is a small message that notifies a process that an event of some type occurred. Signals are often sent by the kernel, but they can be sent from other processes as well.
  - A **signal handler** is a function that executes in response to the arrival and consumption of a signal.
  - You're already familiar with some types of signals, even if you've not referred to them by that name before.
    - You haven't truly programmed in C before unless you've unintentionally dereferenced a `NULL` pointer.
    - When that happens, the kernel delivers a signal of type `SIGSEGV`, informally known as a segmentation fault (or a **SEG**mentation **V**iolation, or `SIGSEGV`, for short).
    - Unless you install a custom signal handler to manage the signal differently, a `SIGSEGV` terminates the program and generates a core dump.
  - Each signal category (e.g. `SIGSEGV`) is represented internally by some number (e.g. 11). In fact, C `#define`s `SIGSEGV` to be the number 11.

Lecture 06: Signals

- Other signal types:
  - Whenever a process commits an integer-divide-by-zero (and, in some cases, a floating-point divide by zero on older architectures), the kernel hollers and issues a **`SIGFPE`** signal to the offending process. By default, the program handles the **`SIGFPE`** by printing an error message announcing the zero denominator and generating a core dump.
  - When you type ctrl-c, the kernel sends a **`SIGINT`** to the foreground process (and by default, that foreground is terminated).
  - When you type ctrl-z, the kernel issues a **`SIGTSTP`** to the foreground process (and by default, the foreground process is halted until a subsequent **`SIGCONT`** signal instructs it to continue).
  - When a process attempts to publish data to the write end of a pipe after the read end has been closed, the kernel delivers a **`SIGPIPE`** to the offending process. The default **`SIGPIPE`** handler prints a message identifying the pipe error and terminates the program.

Lecture 06: Signals

- One signal type most important to multiprocessing:
  - Whenever a child process **changes state**—that is, it exits, crashes, stops, or resumes from a stopped state, the kernel sends a `SIGCHLD` signal to the process's parent.
    - By default, the signal is ignored. In fact, we've ignored it until right now and gotten away with it.
    - This particular signal type is instrumental to allowing forked child processes to run in the background while the parent process moves on to do its own work without blocking on a `waitpid` call.
    - The parent process, however, is still required to reap child processes, so the parent will typically register a custom `SIGCHLD` handler to be asynchronously invoked whenever a child process changes state.
    - These custom `SIGCHLD` handlers almost always include calls to `waitpid`, which can be used to surface the pids of child processes that've changed state. If the child process of interest actually terminated, either normally or abnormally, the `waitpid` also culls the zombie the relevant child process has become.

Lecture 06: Signals

- Our first signal handler example: Disneyland
  - Here's a carefully coded example that illustrates how to implement and install a **`SIGCHLD`** handler.
  - The premise? Dad takes his five kids out to play. Each of the five children plays for a different length of time. When all five kids are done playing, the six of them all go home.
  - The parent process is modeling dad, and the five child processes are modeling his children. (Full program, with error checking, is right here.)

```
1  static const size_t kNumChildren = 5;
2  static size_t numDone = 0;
3
4  int main(int argc, char *argv[]) {
5    printf("Let my five children play while I take a nap.\n");
6    signal(SIGCHLD, reapChild);
7    for (size_t kid = 1; kid <= 5; kid++) {
8      if (fork() == 0) {
9        sleep(3 * kid); // sleep emulates "play" time
10       printf("Child #%zu tired... returns to dad.\n", kid);
11       return 0;
12     }
13   }
```

Lecture 06: Signals

- Our first signal handler example: Disneyland
  - The program is crafted so each child process exits at three-second intervals. **`reapChild`**, of course, handles each of the **`SIGCHLD`** signals delivered as each child process exits.
  - The **`signal`** prototype doesn't allow for state to be shared via parameters, so we have no choice but to use global variables.

```
1    // code below is a continuation of that presented on the previous slide
2    while (numDone < kNumChildren) {
3      printf("At least one child still playing, so dad nods off.\n");
4      sleep(5);
5      printf("Dad wakes up! ");
6    }
7    printf("All children accounted for.  Good job, dad!\n");
8    return 0;
9  }
10
11 static void reapChild(int unused) {
12   waitpid(-1, NULL, 0);
13   numDone++;
14 }
```

Lecture 06: Signals

- Here's the output of the above program.
  - Dad's wakeup times (at t = 5 sec, t = 10 sec, etc.) interleave the various finish times (3 sec, 6 sec, etc.) of the children, and the output published below reflects that.
  - Understand that the **`SIGCHLD`** handler is invoked 5 times, each in response to some child process finishing up.

```
cgregg@myth60$ ./five-children
Let my five children play while I take a nap.
At least one child still playing, so dad nods off.
Child #1 tired... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child #2 tired... returns to dad.
Child #3 tired... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child #4 tired... returns to dad.
Child #5 tired... returns to dad.
Dad wakes up! All children accounted for.  Good job, dad!
cgregg@myth60$
```

Lecture 06: Signals

- Advancing our understanding of signal delivery and handling.
  - Now consider the scenario where the five kids are the same age and run about Disneyland for the same amount of time. Restated, **`sleep(3 * kid)`** is now **`sleep(3)`** so all five children flashmob dad when they're all done.
  - The output presented below makes it clear dad never detects all five kids are present and accounted for, and the program runs forever because dad keeps going back to sleep.

```
cgregg*@myth60$ ./broken-pentuplets
Let my five children play while I take a nap.
At least one child still playing, so dad nods off.
Kid #1 done playing... runs back to dad.
Kid #2 done playing... runs back to dad.
Kid #3 done playing... runs back to dad.
Kid #4 done playing... runs back to dad.
Kid #5 done playing... runs back to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
^C # I needed to hit ctrl-c to kill the program that loops forever!
cgregg@myth60$
```

Lecture 06: Signals

- Advancing our understanding of signal delivery and handling.
  - The five children all return to dad at the same time, but dad can't tell.
  - Why? Because if multiple signals come in at the same time, the signal handler is only run once.
    - If three **`SIGCHLD`** signals are delivered while dad is off the processor, the operating system only records the fact that at one or more **`SIGCHLD`**s came in.
    - When the parent is forced to execute its **`SIGCHLD`** handler, it must do so on behalf of the **one or more signals that may have been delivered**since the last time it was on the processor.
  - That means our **`SIGCHLD`** handler needs to call **`waitpid`** in a loop, as with:

```
static void reapChild(int unused) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, 0);
    if (pid < 0) break;
    numDone++;
  }
}
```

Lecture 06: Signals

- Advancing our understanding of signal delivery and handling.
  - The improved **`reapChild`** implementation seemingly fixes the **`pentuplets`** program, but it changes the behavior of the first **`five-children`** program.
    - When the first child in the original program has exited, the other children are still out playing.
    - The **`SIGCHLD`** handler will call **`waitpid`** once, and it will return the pid of the first child.
    - The **`SIGCHLD`** handler will then loop around and call **`waitpid`** a second time.
    - This second call will **block** until the second child exits three seconds later, preventing dad from returning to his nap.
  - We need to instruct **`waitpid`** to only reap children that have exited but to return without blocking, even if there are more children still running. We use **`WNOHANG`** for this, as with:

```
static void reapChild(int unused) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break; // note the < is now a <=
    numDone++;
  }
}
```

Lecture 06: Signals

- All `SIGCHLD` handlers generally have this `while` loop structure.
  - Note we changed the `if (pid < 0)` test to `if (pid <= 0)`.
  - A return value of -1 typically means that there are no child processes left.
  - A return value of 0—that's a new possible return value for us—means there *are* other child processes, and we would have normally waited for them to exit, but we're returning instead because of the `WNOHANG` being passed in as the third argument.
- The third argument supplied to `waitpid` can include several flags bitwise-or'ed together.
  - `WUNTRACED` informs `waitpid` to block until some child process has either ended or been stopped.
  - `WCONTINUED` informs `waitpid` to block until some child process has either ended or resumed from a stopped state.
  - `WUNTRACED | WCONTINUED | WNOHANG` asks that `waitpid` return information about a child process that has changed state (i.e. exited, crashed, stopped, or continued) but to do so without blocking.