CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 09: Pipes and Interprocess Communication II

- Last lecture was dedicated to the **pipe** system call, with a fleeting mention of **dup2** toward the end. Here are the prototypes:

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

```
int dup2(int source, int target);
```

  - The **pipe** call allocates two descriptors and places them in **fds[0]** and **fds[1]**. The descriptors are configured so that everything written to **fds[1]** is readable via **fds[0]**. Any text that's been written to **fds[1]** but not yet read through **fds[0]** accumulates in a virtual file managed by the OS.
  - The **dup2** call rewires the **target** descriptor so that it references the same file session resources—that is, the same open file table entry—referenced by **source**.
  - Today's first example resides in the previous lecture deck, starting right here.
- Apparently they went with **dup2** as a name because **dup** was already taken.

```
int dup(int source);
```

  - **dup** allocates a new descriptor and configures it to reference the same file session that source references.
  - The primary difference is that you choose the target descriptor number with **dup2**, whereas the OS chooses when you call **dup**.
  - We'll use **dup2** more often in CS110, though there are benefits to knowing of both.

Lecture 09: Pipes and Interprocess Communication II

- The **subprocess** example illustrates how an entire executable can be tapped to manage a sizeable fraction of a larger executable's overall goal.
  - The relationship between the client of **subprocess** and the literal process created by calling it is very clearly a parent-child one.
- The Unix pipeline is supported by most shells—in fact, you'll see when you implement one for Assignment 4—by creating two or more sibling concurrent processes so they're chained together via their standard input and output streams.
  - More specifically, the standard output of one process is forwarded on to and consumed as the standard input of the next process in the sequence.
  - Some examples:

```
poohbear@myth51:~$ cat /usr/include/tar.h | wc
    112     600    3786
poohbear@myth51:~$ echo -e "pear\ngrape\npeach\napricot\nbanana\napple" | sort | grep ap
apple
apricot
grape
poohbear@myth51:~$ time sleep 5 | sleep 10
real    0m10.004s
user    0m0.006s
sys     0m0.001s
poohbear@myth55:~$ curl -sL "http://cs110.stanford.edu/odyssey.txt" | sed 's/[^a-zA-Z ]/ /g' |
> tr 'A-Z ' 'a-z\n' | grep [a-z] | sort -u |
> comm -23 - <(sort /usr/share/dict/words) | less
poohbear@myth55:~$
```

  - We're going to focus on pipelines of length 2 right now, and we'll leave pipelines of arbitrary length to Assignment 4.

Lecture 09: Pipes and Interprocess Communication II

- Let's implement a binary **pipeline** function that codes to the following interface:

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]);
```

- **pipeline** accepts two argument vectors and, assuming both vectors are valid, spawns off twin processes with the added bonus that the standard output of the first is directed to the standard input of the second.
  - For simplicity, we'll assume all **pipeline** calls are well-formed and work as expected. **argv1** and **argv2** are each valid, **NULL**-terminated argument vectors, and **pids** is the base address of an array of length two.
  - We'll also assume all calls to **pipe**, **dup2**, **close**, **execvp**, and so forth succeed so that you needn't do any error checking whatsoever.
  - **pipeline** should return without waiting for either of the child processes to finish, and the pids of the two processes are dropped into **pids[0]** and **pids[1]**.
  - We'll be sure the two processes running in parallel, so that

```
pid_t pids[2];
pipeline({"sleep", "10", NULL}, {"sleep", "10", NULL}, pids);
```

    takes about 10 seconds.

Lecture 09: Pipes and Interprocess Communication II

- Here's the implementation of my own **pipeline** function:

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
  int fds[2];
  pipe(fds);

  pids[0] = fork();
  if (pids[0] == 0) {
    close(fds[0]);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(argv1[0], argv1);
  }
  close(fds[1]); // was only relevant to first child, so close before second fork

  pids[1] = fork();
  if (pids[1] == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    execvp(argv2[0], argv2);
  }
  close(fds[0]);
}
```

  - The key difference between this function and prior ones is that **two** child processes are created—true twinsies—to run executables identified by **argv1** and **argv2**.
  - The **fork** and **execvp** work is more straightforward than the descriptor work. Note, however, that the child pids are dropped in the two slots of the supplied **pids** array.

Lecture 09: Pipes and Interprocess Communication II

- Here's the implementation of my own **pipeline** function:

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
  int fds[2];
  pipe(fds);

  pids[0] = fork();
  if (pids[0] == 0) {
    close(fds[0]);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(argv1[0], argv1);
  }
  close(fds[1]); // was only relevant to first child, so close before second fork

  pids[1] = fork();
  if (pids[1] == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    execvp(argv2[0], argv2);
  }
  close(fds[0]);
}
```

  - The **pipe** call must come before the **fork** calls, because the pipe endpoints need to exist at the time the child processes come into being. The ordering here is crucial, else the child processes won't inherit endpoint descriptors related to the pipe.

Lecture 09: Pipes and Interprocess Communication II

- Here's the implementation of my own **pipeline** function:

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
  int fds[2];
  pipe(fds);

  pids[0] = fork();
  if (pids[0] == 0) {
    close(fds[0]);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(argv1[0], argv1);
  }
  close(fds[1]); // was only relevant to first child, so close before second fork

  pids[1] = fork();
  if (pids[1] == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    execvp(argv2[0], argv2);
  }
  close(fds[0]);
}
```

  - The first child process inherits clones of the two pipe endpoints. The first child doesn't need **fds[0]** for anything meaningful, so it's closed right away. **fds[1]** is also closed, but only after **dup2** uses it to rewire the child's standard output to reference the write end of the pipe.

Lecture 09: Pipes and Interprocess Communication II

- Here's the implementation of my own **pipeline** function:

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
  int fds[2];
  pipe(fds);

  pids[0] = fork();
  if (pids[0] == 0) {
    close(fds[0]);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(argv1[0], argv1);
  }
  close(fds[1]); // was only relevant to first child, so close before second fork

  pids[1] = fork();
  if (pids[1] == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    execvp(argv2[0], argv2);
  }
  close(fds[0]);
}
```

  - Once the first child has been spawned, **pipeline** can close **fds[1]**, since it's of no importance to the second child and doesn't need to be open for the second **fork** call.

Lecture 09: Pipes and Interprocess Communication II

- Here's the implementation of my own **pipeline** function:

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
  int fds[2];
  pipe(fds);

  pids[0] = fork();
  if (pids[0] == 0) {
    close(fds[0]);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(argv1[0], argv1);
  }
  close(fds[1]); // was only relevant to first child, so close before second fork

  pids[1] = fork();
  if (pids[1] == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    execvp(argv2[0], argv2);
  }
  close(fds[0]);
}
```

  - The second child's standard input is rewired to reference the read end of the pipe. Once **dup2** and **fds[0]** have been used to finagle that rewiring, **fds[0]** can be closed. Note that **fds[1]** wasn't even inherited across the fork boundary.

Lecture 09: Pipes and Interprocess Communication II

- Here's the implementation of my own **pipeline** function:

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
  int fds[2];
  pipe(fds);

  pids[0] = fork();
  if (pids[0] == 0) {
    close(fds[0]);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(argv1[0], argv1);
  }
  close(fds[1]); // was only relevant to first child, so close before second fork

  pids[1] = fork();
  if (pids[1] == 0) {
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    execvp(argv2[0], argv2);
  }
  close(fds[0]);
}
```

  - Finally, we close **fds[0]** in the primary process.
  - Note that there were a total of five **close** calls across three processes. This is the cost of creating a pipe ahead of multiple fork calls. There's simply no other programmatic way to enable unidirectional communication channels between sibling processes unless this type of thing is done.

Lecture 09: Pipes and Interprocess Communication II

- The cascade of **close** calls seen in the last example is fairly common when coding with multiple processes. In particular, the child processes generally need to close all **pipe** endpoints they've inherited ahead of their **execvp** calls.
- There's a second version of **pipe** called **pipe2** with the following prototype:

```
int pipe2(int fds[], int flags);
```

- A call to **pipe2(fds, 0)** is functionally equivalent to **pipe(fds)**. A call to **pipe2(fds, O_CLOEXEC)**, however, populates the supplied **fds** array with pipe endpoint descriptors *that automatically close when the surrounding processes calls* ***execvp***.

- Presented on the right is an arguably better version of our **pipeline** function, since this version allows to omit the close calls in the child processes. That's permitted, because the **execvp** system call automatically closes any descriptors that have been marked as **O_CLOEXEC**.

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
  int fds[2];
  pipe2(fds, O_CLOEXEC);
  pids[0] = fork();
  if (pids[0] == 0) {
    dup2(fds[1], STDOUT_FILENO);
    execvp(argv1[0], argv1);
  }
  close(fds[1]);
  pids[1] = fork();
  if (pids[1] == 0) {
    dup2(fds[0], STDIN_FILENO);
    execvp(argv2[0], argv2);
  }
  close(fds[0]);
}
```

Lecture 09: Preamble to Signals

- Processes start, cycle on/off the CPU, and eventually terminate, but they can also be paused at arbitrary points by what are called job control signals.
  - Maybe you're running a long, CPU-intensive program, and you want to pause it so you can quickly run some much shorter CPU-intensive program without competition.
  - Perhaps MacOS will send "pause" signals to programs when it starts running out of physical memory, prompting you to close some apps before resuming them.
- Job control is sometimes used programmatically to synchronize between processes; for example, process A might halt itself to wait for process B to catch up, and then process B will signal process A to continue when it's ready.

Lecture 09: Preamble to Signals

- We'll talk more about signals on Monday, so don't worry much about the details of how this works. Just know you can send a particular signal called **SIGSTOP** to pause a process and another signal called **SIGCONT** to continue it.

From the command line

```
# Pause PID 1234
myth61$ kill -STOP 1234
# Resume PID 1234
myth61$ kill -CONT 1234
```

Programmatically

```
// Pause PID 1234
kill(1234, SIGSTOP);
// Resume PID 1234
kill(1234, SIGCONT);
```

- **waitpid** can be used to *observe* when a program changes job control states (i.e. exits, crashes, stops, or continues). This is managed via that third **flags** parameter that, until now, we've always just set to 0.
  - Passing **WUNTRACED** as the third argument to **waitpid** tells that **waitpid** to return when a process in the supplied wait set either stops or exits.

```
pid_t pid = waitpid(-1, &status, WUNTRACED);
```

  - Passing **WCONTINUED** as the third argument to **waitpid** tells that **waitpid** to return when a process in the supplied wait set either continues or exits.

```
pid_t pid = waitpid(-1, &status, WCONTINUED)
```

  - Passing in **WUNTRACED | WCONTINUED** —yes, that's a bitwise or— tells **waitpid** to return because of any state change whatsoever.

```
pid_t pid = waitpid(-1, &status, WUNTRACED | WCONTINUED);
```