CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 08: Pipes and 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!
  - Our first example was presented in the last lecture slide deck right here.

Lecture 08: Pipes and Interprocess Communication

- On **fork**, a child process inherits a properly wired file descriptor 1—you know, **stdout**—that's linked to the terminal.
- If the child process calls **execvp**, then the freshly installed executable still directs anything published to descriptor 1 to your screen.
- Remember the notion of a virtual file from our discussion of the vnode table?
  - We can think of pipes as virtual files.
  - In general, pipes collect bytes of information much as traditional files do, except that anything published to the write end of the pipe is readable from the other end of the pipe, and there's no persistent storage of the written but unread bytes beyond the pipe's lifetime. *You can pretend there's an unnamed file behind the scenes if it helps you understand where the data might be held.*
- It's because descriptors are duplicated across **fork** boundaries that pipes can bridge multiple processes.
- These bridges between processes enable an ability to pass arbitrary data from one side to the other.

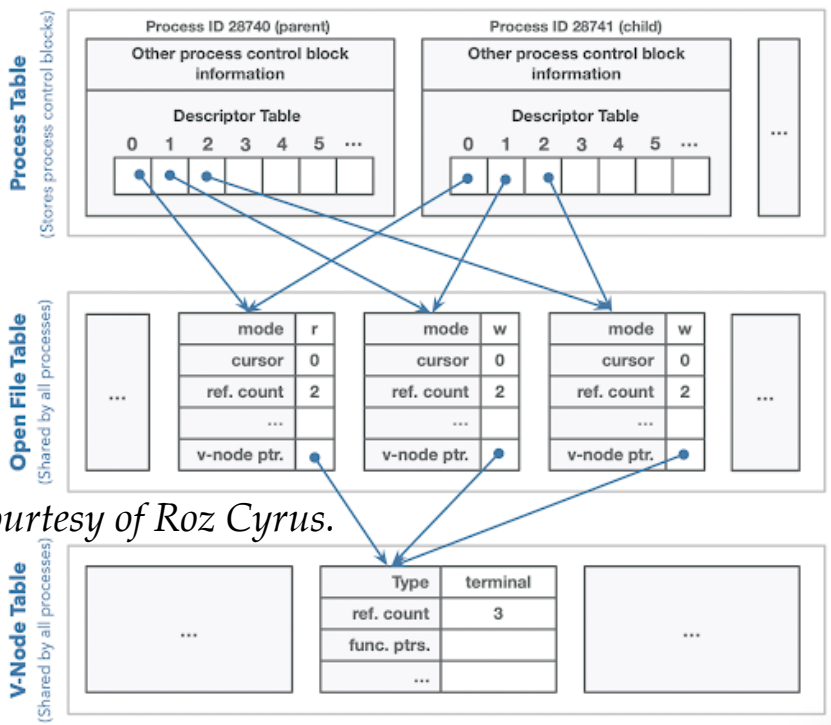


Illustration courtesy of Roz Cyrus.

Lecture 08: Pipes and Interprocess Communication

- Let's take this pipe thing seriously and try to build some intuition around data flow from one process to another and where we may have seen this before.

Illustration by Roz.

- The illustration above depicts how file descriptors 0, 1, and 2—that is, **stdin**, **stdout**, and **stderr**, respectively—are traditionally wired.
- We use IN and OUT to represent data supplied by some source and consumed by others. Here, the keyboard supplies data to the process running **command**, and that program published output to the terminal.
  - Generally, data flowing in is considered input and data flowing out of something is considered output.
  - Restated, input is read from somewhere; output is written somewhere.
- By protocol, Unix executables typically read from descriptor 0 and write to descriptors 1 and 2. We'll soon learn how to rewrite 0, 1, and 2 so that data is read and written from and to different resources.

Lecture 08: Pipes and Interprocess Communication

- Shell pipes—courtesy of the vertical bar—allow processes to be daisy-chained so that the standard output of one process feeds the standard input of a subsequent one, as with:

```
myth58$ echo -e "peach\npear\napple" | sort | grep ea
```

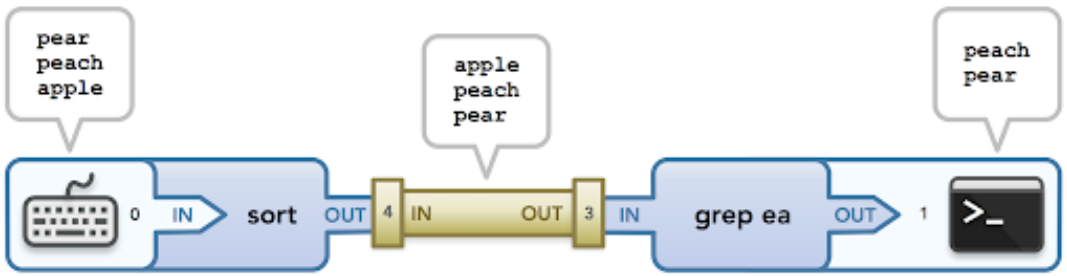


Illustration by Roz.

- Since each command in the pipeline—**echo**, **sort** and **grep**, for instance—is run in a separate process, we need a way to allow those processes to pass information on to each other. The **pipe** system call was designed specifically to address this need.
- Above, we see data flows—like cool, refreshing water—out of **sort** and into one end of. the pipe. That same data flows out the other end and into **grep**.
  - The 3 and 4 you see decorating the pipe endpoints are, in practice, the descriptors tapped by the pipe system call that ultimately enables this left-to-right data flow.

Lecture 08: Pipes and Interprocess Communication

- Presented below is a replica of the first program we wrote today, but this time with a detailed illustration of how a pipe bridges the parent and child processes to allow a hello to pass from the first to the second.

```c
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)); // assume one call is enough
    printf("Read from pipe bridging processes: %s.\n", buffer);
    close(fds[0]);
    return 0;
  }
  close(fds[0]);
  write(fds[1], "hello", 6);
  close(fds[1]);
  waitpid(pid, NULL, 0);
  return 0;
}
```

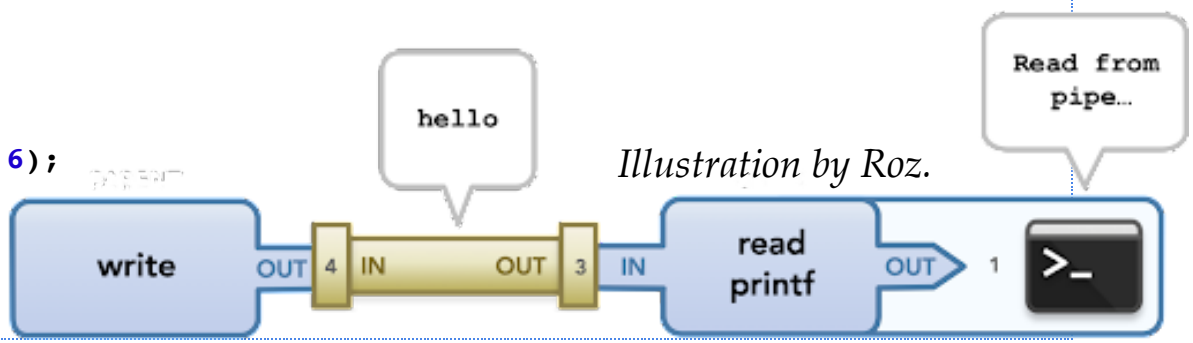


Illustration by Roz.

- The drawing emphasizes that **write** publishes the **"hello"** to the write end of the pipe, the read pulls that same **"hello"** from the read end of the pipe, and the **printf** feeds traditional standard out.

Lecture 08: Pipes and Interprocess Communication

- And here is a step-by-step walkthrough the full implementation to show the descriptor tables are populated, cloned, and otherwise manipulated as everything executes.

- The first of the two images on the right conveys the default state of our program just as it's launched, before the call to pipe. Descriptor 0 is configured to vacuum up characters as they're typed, and descriptors 1 and 2 are independently wired to pixelate our computer screen.
- After the call to **pipe** but pre-**fork**, descriptors 3 and 4—the lowest unused descriptor numbers at the time **pipe** is called—are allocated and associated with the write and read ends of the pipes, respectively. The 3 and the 4 are then dropped side by side into the **fds** array.
- The call to **fork** creates a new process in the image of the caller, and the descriptor table of the clone is a duplicate of the original.

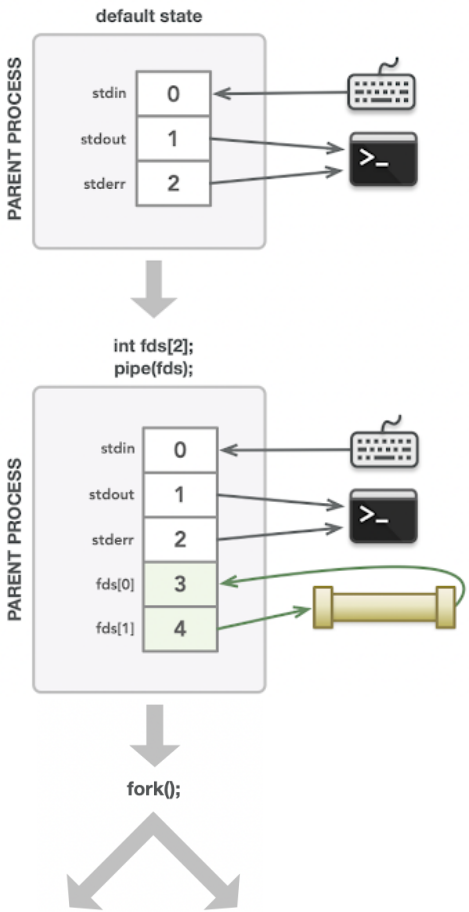


Illustration by Roz.

Lecture 08: Pipes and Interprocess Communication

- Repeated: The call to **fork** creates a new process in the image of the caller, and the descriptor table of the clone is a shallow copy of the original.

- Next, the parent and child close the descriptors they don't use. The parent is only writing, so it closes its access to the read end of the pipe. The child only reads so it closes its access to the write end.
- Note the illustration shades new descriptors in a cool mint green and descriptors being closed in cinnamon red.
- After the close calls in parent and child, the only process that can feed the pipe is the parent (through its **fds[1]** descriptor), and the only process that can draw from the pipe is the child (through its **fds[0]** descriptor).

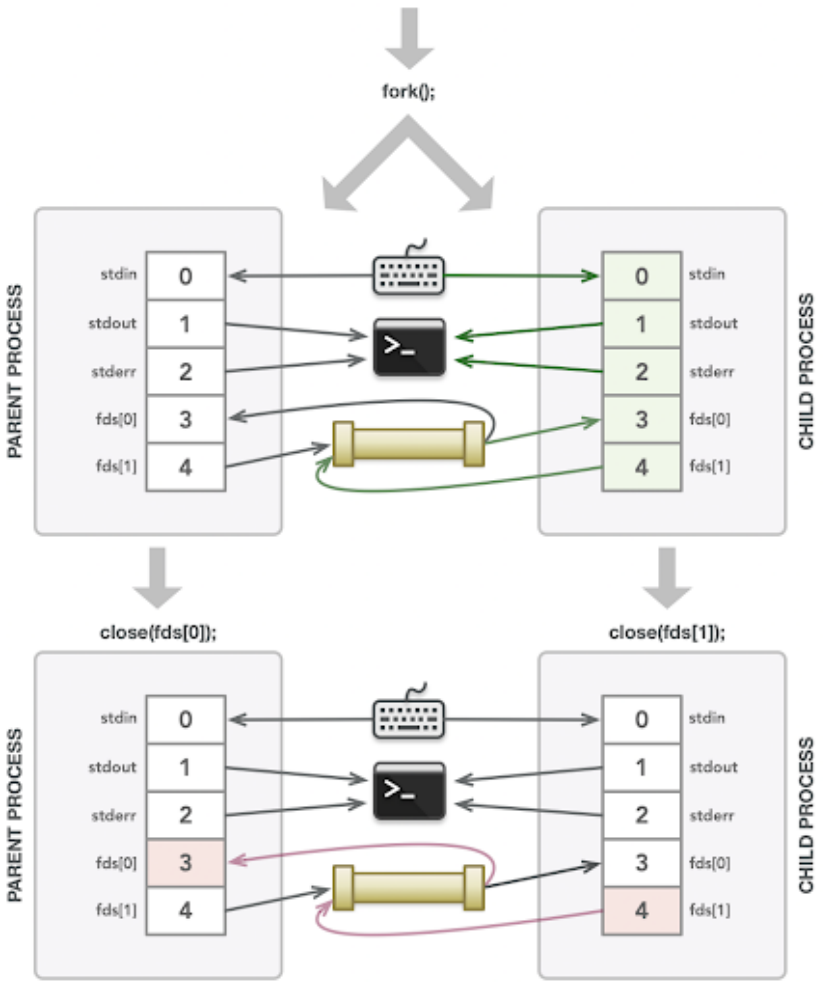


Illustration by Roz.

Lecture 08: Pipes and Interprocess Communication

- The parent presses **"hello"** and the trailing **'\0'**—that's six bytes—into the write end of the pipe via **fds[1]**. Those six bytes are pulled in via the child's **read(fds[0], buffer, sizeof(buffer))** call, and subsequently printed via **printf**, which uses file descriptor 1 as virtually all **printf**s do.
- After the two processes finish writing and reading, they each advance on to **close** access to pipe endpoints they have but no longer need.
- Make a concerted effort to donate all resources—e.g. unused descriptors—back to the system as aggressively as possible.
- The parent's workflow is gated by the call to **waitpid** to ensure the child runs to completion before allowing itself to exit.
- As each process exits, its three default descriptors are closed on your behalf. In particular, you don't need to close 0, 1, and 2 just because you're in CS110 now.

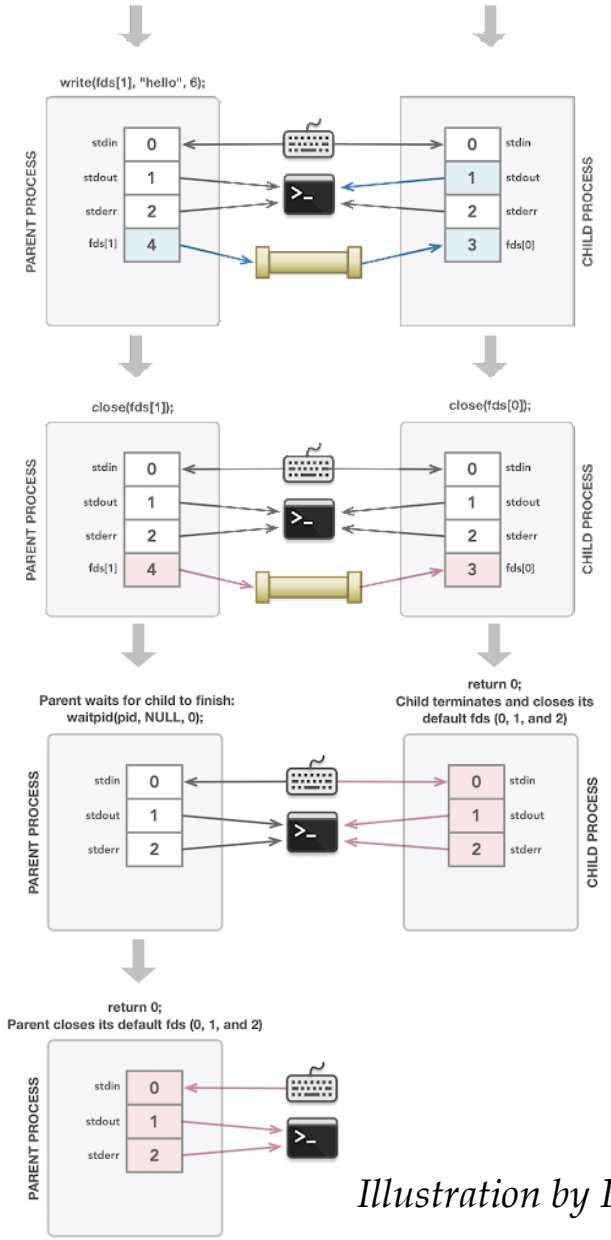
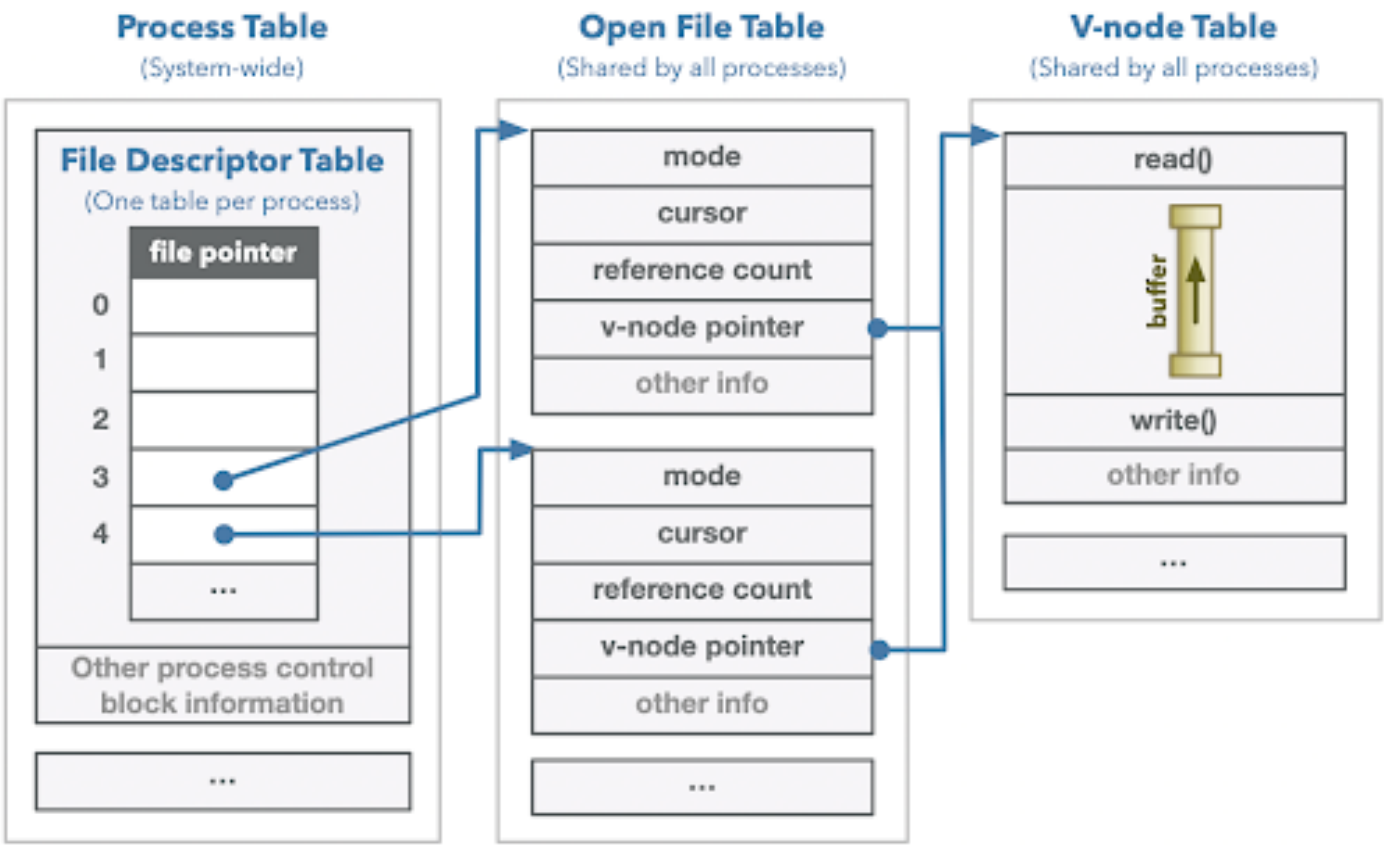


Illustration by Roz.

Lecture 08: Pipes and Interprocess Communication

- Here's a respectable view on how a pipe is set up by the OS: The material written to a pipe is stored in a virtual file, but within main memory instead of the actual filesystem.
- The relevant read and write functions are reachable from the vnode (i.e. the vnode stores functions pointers). The read function extracts data from the buffer and the write function puts stuff in. A vnode table entry operates as a function lookup table with some associated metadata (e.g. the location of this buffer).



Lecture 08: Pipes and Interprocess Communication

- Introducing the **dup2** system call.
  - Pipes are typically used to allow one process to pass data to a second. However, a process may want to rewrite a particular descriptor—say, 0, 1, or 2—so that its bound not to the keyboard or the terminal, but rather to one of the pipe endpoints.
  - This is where the **dup2** system call comes into play, and this is its prototype.

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

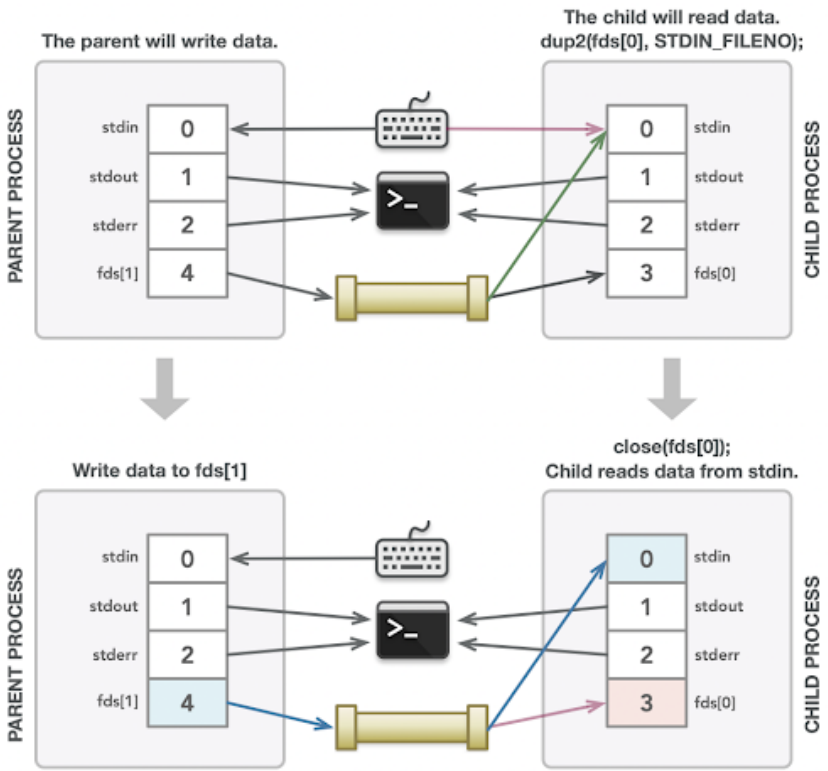
  - **dup2** rewires the **target** descriptor so that it references the same open file table entry that **source** does. So, after **dup2** returns, source and target are bound to the same resource. If **target** was referencing something else prior to the call, it is first closed before any rewiring is done.
  - The return value is simply the value of **target**, provided **dup2** manages the rewiring without drama. Otherwise, it returns -1 like any unhappy system call would.
    - Truth be told, I never look at the return value of this particular system call.

Lecture 08: Pipes and Interprocess Communication

- Here's a sample call to **dup2** and **close** that comes up quite a bit in practice:

```
dup2(fds[0], STDIN_FILENO); // STDIN_FILENO is a #define constant for 0
close(fds[0]);
```

- Here, the child's **FILENO_STDIN**—presumably open and linked to the screen—is closed, thereby detaching it from the terminal it's typically attached to. **FILENO_STDIN** is then bound to the same esource that **fds[0]** is, which, as per our illustration, is the read end of the pipe.
- The subsequent **close** call shuts down the read-end descriptor allocated by the original **ipe** call, leaving the child's descriptor 0 as the only descriptor through which material residing in this pipe can be read.
- This has the dramatic effect that all reads from file descriptor 0—a number which is effectively hardcoded into all **scanf**, **getline**, and **getc** calls—pull in bytes from a pipe instead of the terminal. The fact that the 0 descriptor was rewired under the hood is completely hidden from the implementation of all functions that read from it.



Lecture 08: 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 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
6 subprocess_t subprocess(char *command);
```

  - The child process created by **subprocess** executes the provided **command** 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**.
    - We'll implement subprocess so that arbitrary text can be published to this **supplyfd** field with the understanding that it can be ingested verbatim by the child's standard input.

Lecture 08: 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 own **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); // necessary to trigger end-of-input within child
11   waitpid(sp.pid, NULL, 0);
12   return 0;
13 }
```

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

Lecture 08: Pipes and Interprocess Communication

- Key features of the test harness:
  - The program creates a **subprocess_t** running **sort** and posts eight fancy words to **supplyfd**, knowing those words flow through some pipe to the child's **stdin**.
  - The parent shuts the **supplyfd** down by passing it to **close** to indicate that no more data will ever be posted via that descriptor. The reference count of the relevant open file entry referenced by **supplyfd** is demoted from 1 to 0 with that **close** call,

```
 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); // necessary to trigger end-of-input within child
11   waitpid(sp.pid, NULL, 0);
12   return 0;
13 }
```

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

  and that triggers an **EOF** in the child process reading data from its standard in.
  - The parent blocks on an **waitpid** call until the child exits, and then itself exits.

Lecture 08: Pipes and Interprocess Communication

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

```
 1 subprocess_t subprocess(char *command) {
 2   int fds[2];
 3   pipe(fds);
 4   subprocess_t process = { fork(), fds[1] };
 5   if (process.pid == 0) {
 6     close(fds[1]);
 7     dup2(fds[0], STDIN_FILENO);
 8     close(fds[0]);
 9     char *argv[] = {"/bin/sh", "-c", command, NULL};
10     execvp(argv[0], argv);
11   }
12   close(fds[0]);
13   return process;
14 }
```

  - The write end of the pipe is embedded with a **subprocess_t** so the parent knows where to publish text so it flows to the read end of the pipe, across the parent process/child process boundary.
  - 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.
  - All other descriptors are closed as aggressively as possible.
    - Note we don't close **fds[1]** in the parent! Why not?