CS110 Spring 2021

Lecture 6: More on Multiprocessing

Principles of Computer Systems

Stanford University, Dept. Of Computer Science
Lecturer: Roslyn Michelle Cyrus
Content adapted from material by Jerry Cain.
Diagrams and pipe content by Roslyn.

Version 2

Lecture Overview

- More about **`waitpid`**
- Starting executables from disk with **`execvp`**
- Freeform communication between two processes using **pipes**

Reading

- My data flow blog post

Remember to download lecture slides! They usually have more slides/examples than I'm able to cover in class.

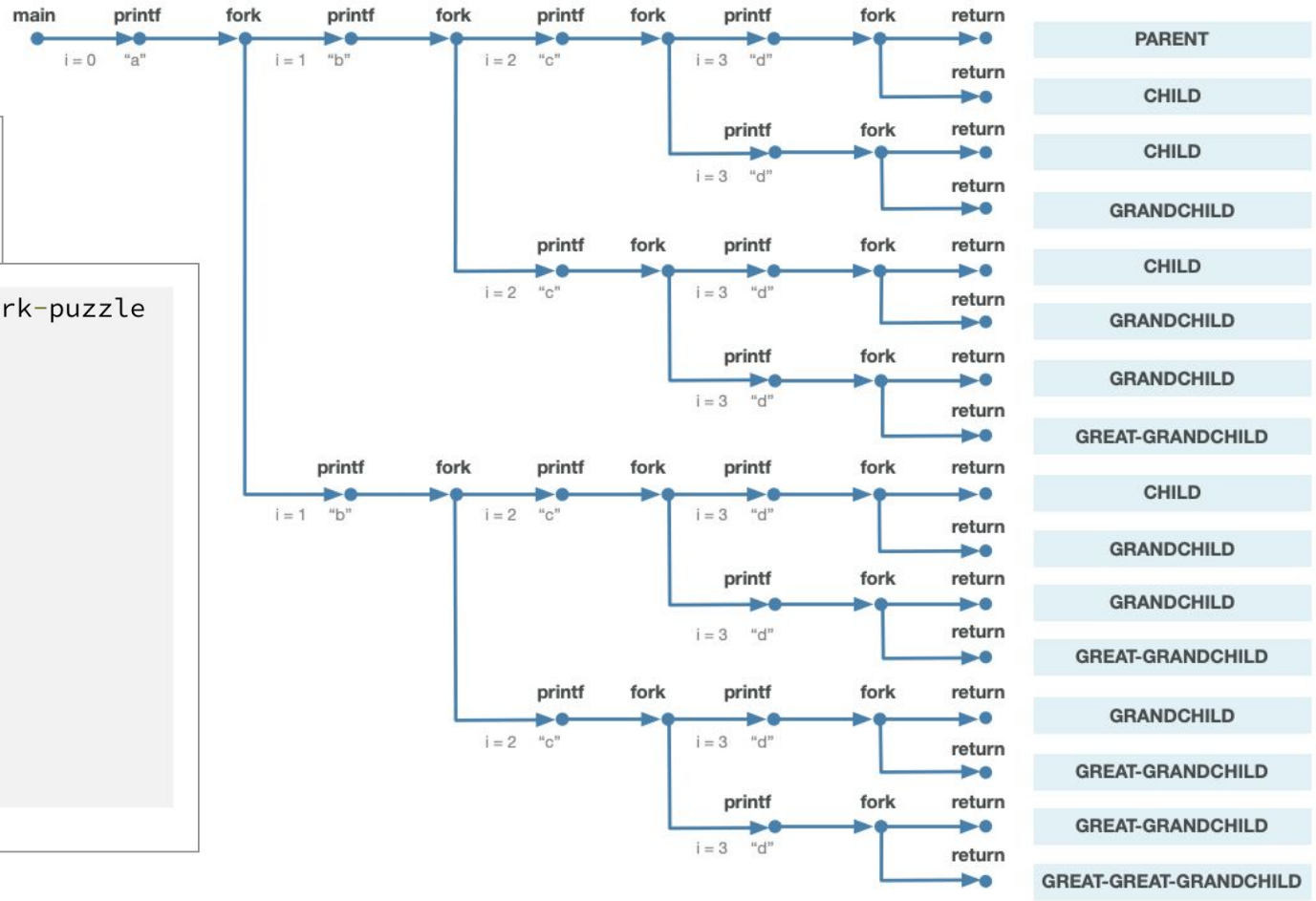
Accessing Code Examples

- Today's lecture examples reside within: **`/usr/class/cs110/lecture-examples/processes`**.
  - First **`ssh`** into a myth machine (ssh yourusername@myth.stanford.edu). When prompted for your password, it is normal for the text not to appear as you enter your password. Once logged onto a myth machine, **`cd`** into the above directory.
  - To get started, type: **`git clone /usr/class/cs110/lecture-examples cs110-lecture-examples`** at the command prompt to create a local copy of the master.
  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type **`git pull`**. Doing so will update your local copy to match whatever the master has become.

Fork-puzzle Recap

```
myth60$ ./fork-puzzle
a
b
c
b
d
c
d
c
c
d
d
d
d
d
d
myth60$
```

```
myth60$ ./fork-puzzle
a
b
b
c
d
c
d
c
d
d
c
d
myth60$ d
d
d
```



Recap: Waiting for Children to Finish

- Synchronization between parent and child can be done by using the system call **`waitpid`**. It can be used to temporarily block a process until a child process terminates or stops.

```
pid_t waitpid(pid_t pid, int *status, int options); // returns child PID if ok, 0 if WNOHANG, -1 if error
```

- The first argument specifies the **wait set,** which for the moment is just the ID of the child process that needs to complete before **`waitpid`** can return.
- The second argument supplies the address of an integer where termination information can be placed (or we can pass in **NULL** if we don't care for the information).
- The third argument is a collection of bitwise-or'ed flags we'll study later. For the time being, we'll just go with 0 as the required parameter value, which means that **`waitpid`** should only return when a process in the supplied wait set exits.
- The return value is the pid of the child that exited, or -1 if **`waitpid`** was called and there were no child processes in the supplied wait set.

Waiting For Children to Finish

Consider the following program, which is more representative of how **fork** really gets used in practice (full program, with error checking, is right here):

```
int main(int argc, char *argv[]) {
  printf("Before.\n");
  pid_t pid = fork();
  printf("After.\n");
  if (pid == 0) {
    printf("I am the child, and the parent will wait up for me.\n");
    return 110; // contrived exit status
  } else {
    int status;
    waitpid(pid, &status, 0);
    if (WIFEXITED(status)) {
      printf("Child exited with status %d.\n", WEXITSTATUS(status));
    } else {
      printf("Child terminated abnormally.\n");
    }
    return 0;
  }
 }
```

- The parent process correctly waits for the child to complete using **`waitpid`**.
- 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 (which is 110 as expected).
- The **`waitpid`** call also donates child process-oriented resources back to the system.

Reaping Zombie Processes

- When a process terminates for any reason, the kernel doesn't remove it from the system immediately. The terminated process is kept around until it is **reaped** (collected and cleaned up) by its parent. Thus the **`waitpid`** call donates child process-oriented resources back to the system, allowing the process's entry in the process table to be removed by the kernel.
  - When the parent reaps the terminated child, the kernel passes the child's exit status to the parent (to be processed by the **`waitpid`** call) and then discards the terminated process (at which point it ceases to exist).
- If **`waitpid`** isn't called, the terminated children processes become **zombies**. A zombie is a terminated process that has not yet been reaped by its parent. You can think of zombies as orphans.
  - When a parent process terminates without reaping its children, the kernel has the **`init`** process "adopt" and reap the parent's orphaned children processes.
  - **`init`** (which has process ID 1) is the ancestor of every process: it is created by the kernel when the system boots and it never terminates (note: **`init`** is now **`systemd`** on most systems).
- Long-running programs (like shells) should reap their zombie children since zombies still consume memory resources even though they are no longer running.

Reaping Zombie Processes

- Here's an example of a program that **doesn't** reap its child (like earlier fork examples from the last lecture).

```c
int main(int argc, char *argv[]) { // zombie.c
  printf("Currently running process: %d\n", getpid());

  pid_t pid = fork();
  assert(pid >= 0);

  if (pid == 0) { // in child
      printf("Child process (pid=%d) exiting.\n", getpid());
    exit(0); // terminate child
  } else {
      // in parent, which doesn't reap its child,
      // so while parent runs for a minute, the child
      // is in a zombie state once the child exits.
      // The child will be adopted by init when
      // the parent process ends.

      sleep(60);
  }
  return 0;
}
```

```
myth60$ ./zombie &
[1] 1613
Currently running process: 1613
Child process (pid=1614) exiting.
myth60$ ps -o pid,state,command | grep zombie
1613 S ./zombie
1614 Z [zombie] <defunct>
1626 S grep zombie
```

- **`return`** is an instruction of the language that returns from a function call.
- **`exit`** is a system call (not part of the language) that terminates the current process.
- The **`&`** runs the program in the background.
- The **`sleep`** call is there to give us time to check that the child was terminated but not reaped. Without it, the init process would reap the process quicker than we could check for a zombie process.

Reaping Zombie Processes

- Here's an example of a program that **does** reap its child (using **`waitpid`**).

```c
int main(int argc, char *argv[]) { // zombie_reaped.c
  printf("Currently running process: %d\n", getpid());

  pid_t pid = fork();
  assert(pid >= 0);

  if (pid == 0) { // in child
    printf("Child process (pid=%d) exiting.\n",
getpid());
    exit(0); // terminate child
  } else { // in parent, which reaps its child
    int status;
    waitpid(pid, &status, 0);

    sleep(60);
  }

  return 0;
}
```

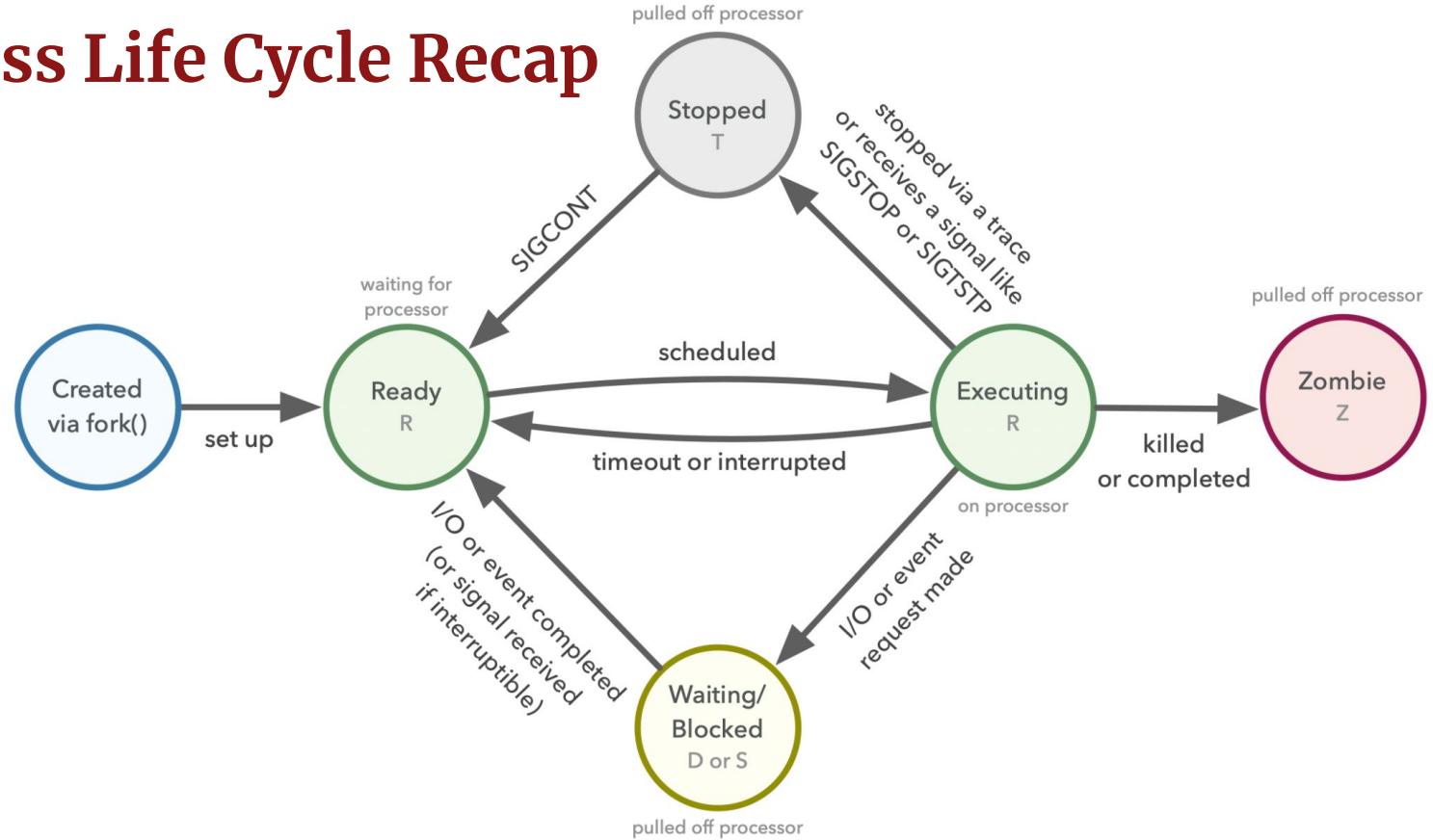
```
myth60$ ./zombie_reaped &
[1] 1896
Currently running process: 1896
Child process (pid=1898) exiting.

myth60$ ps -o pid,state,command | grep zombie
1896 S ./zombie_reaped
1938 S grep zombie
```

- The child was successfully reaped by the parent, so there's no zombie process running.

Question: if we moved the `sleep(60)` to the child's code (before its `exit` call), what states would you expect the parent to be in within that ~minute that the child is sleeping?

Process Life Cycle Recap



- **Answer**: the parent and child would both be in the waiting state. To be clear, **`waitpid`** does not influence the scheduling of processes; calling waitpid on a process does not mean that the process gets a higher priority. **`waitpid`** simply blocks the parent process until the specified child process has finished executing.

More Details About The `waitpid` Arguments

Waitpid: the pid argument

```
pid_t waitpid(pid_t pid, int *status, int options); // returns child PID if ok, 0 if WNOHANG, -1 if error
```

- As shown in the previous example, by default (when options = 0), **waitpid** suspends execution of the calling process until a child process in its **wait set** terminates.
- If a process in the wait set has already terminated at the time of the call, then **waitpid** returns immediately.
- **waitpid** returns the pid of the terminated child that caused **waitpid** to return, at which point the terminated child has been reaped and removed from the system by the kernel (it is removed from the process table).
- The wait set depends on the pid argument:
  - **pid > 0**: the wait set is the single child process that has that pid
  - **pid = -1**: the wait set is all of the parent's child processes
  - **pid = 0**: any child process whose process group ID is equal to that of the calling process.
  - **pid < -1**: any child process whose process group ID is equal to the absolute value of pid.

Waitpid: the status argument

```
pid_t waitpid(pid_t pid, int *status, int options); // returns child PID if ok, 0 if WNOHANG, -1 if error
```

If the **status** argument isn't **NULL**, then **waitpid** encodes the child's status information in the value pointed to by **status**. Several macros are defined for interpreting **status**:

WIFEXITED(status)	Returns true if the child terminated normally via a call to **exit** or a return.
WEXITSTATUS(status)	Returns the exit status of a normally terminated child (only defined if **WIFEXITED(status)** returned true).
WIFSIGNALED(status)	Returns true if the child process terminated because of a signal that wasn't caught.
WTERMSIG(status)	Returns the number of the signal that caused the child process to terminate (only defined if **WIFSIGNALED(status)** returned true).
WIFSTOPPED(status)	Returns true if the child that caused the return is currently stopped.
WSTOPSIG(status)	Returns the number of the signal that caused the child to stop (only defined if **WIFSTOPPED(status)** returned true).
WIFCONTINUED(status)	Returns true if the child process was restarted by receipt of a SIGCONT signal.

Waitpid: the options argument

```
pid_t waitpid(pid_t pid, int *status, int options); // returns child PID if ok, 0 if WNOHANG, -1 if error
```

- Recall that the default behavior of **waitpid** (when options is 0) is to suspend execution of the calling process until a child process in its wait set terminates.
- To change the default behavior, set **options** to **OR**ed combinations of the following constants:

WNOHANG	Instead of waiting, return immediately (with a return value of 0) if none of the child processes in the wait set has terminated yet. Useful if you want to keep doing useful work while waiting for a child to terminate.
WUNTRACED	Suspend execution of the calling process until a process in the wait set becomes either terminated OR stopped. Return the PID of the child that caused the return. Useful when you want to check for both terminated and stopped children.
WCONTINUED	Suspend execution of the calling process until a running process in the wait set is terminated or until a stopped process in the wait set has been resumed by the receipt of a **SIGCONT** signal.

Waitpid "errors"

```
pid_t waitpid(pid_t pid, int *status, int options); // returns child PID if ok, 0 if WNOHANG, -1 if error
```

- If the calling process has no children, then **waitpid** returns -1 and sets **errno** to ECHILD.
- If **waitpid** was interrupted by a signal, then it returns -1 and sets **errno** to EINTR.

Wait

```
pid_t wait(int *status);
```

- This is a simpler version of **waitpid**.
  - **wait(&status)** is the same as **waitpid(-1, &status, 0)**.

Note: waitpid can only be called on direct child processes (not parent processes, or grandchild processes, or anything else).

Bonus: Another `waitpid` example

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

```
int main(int argc, char *argv[]) { // parent-child.c
  srandom(time(NULL)); // for changing random number seed
  printf("I'm unique and just get printed once.\n");
  pid_t pid = fork();
  bool isparent = pid != 0;
  if ((random() % 2 == 0) == isparent) sleep(1); // force exactly one of the two to sleep
  if (isparent) waitpid(pid, NULL, 0); // parent shouldn't exit until child has finished
  printf("I get printed twice (this one is being printed from the %s).\n",
         isparent  ? "parent" : "child");
  return 0;
}
```

- The code emulates a coin flip to get exactly one of the two processes to sleep for a second, which is more than enough time for the child process to finish. Since the seed is the same in both the parent and child, both processes will get the same result from the **`random`** call! This is how we ensure that only one process sleeps.
- The parent waits for the child to exit before it allows itself to exit.
- The final **`printf`** gets executed twice. **Who will always execute it first?**
    - The child, because the parent is blocked in its **`waitpid`** call until the child executes **everything**.

What if a process has more than one child?

Reaping Several Children

- If a parent calls **fork** multiple times, it should reap all of the child processes (via **waitpid**) once they exit.
- If we want to reap processes as they exit without concern for the order they were spawned, then this does the trick (full program checking right here):

```c
int main(int argc, char *argv[]) { // reap-as-they-exit.c
  for (size_t i = 0; i < 8; i++) {
    if (fork() == 0) exit(110 + i);
  }
  while (true) {
    int status;
    pid_t pid = waitpid(-1, &status, 0);
    if (pid == -1) { assert(errno == ECHILD); break; }
    if (WIFEXITED(status)) {
      printf("Child %d exited: status %d\n", pid, WEXITSTATUS(status));
    } else {
      printf("Child %d exited abnormally.\n", pid);
    }
  }
  return 0;
}
```

This calls **waitpid** a total of 9 times (it returns child PIDs 8 times, then returns -1 to indicate that there are no remaining children).

Note that we feed a -1 as the first argument to **waitpid**. That -1 states that we want to hear about **any** child as it exits, and pids are returned in the order their processes finish.

Eventually, all children exit and **waitpid** correctly returns -1 to signal there are no more processes under the parent's jurisdiction.

Reaping Several Children (continued)

```
rice11$ ./reap-as-they-exit
Child 1209 exited: status 110
Child 1210 exited: status 111
Child 1211 exited: status 112
Child 1216 exited: status 117
Child 1212 exited: status 113
Child 1213 exited: status 114
Child 1214 exited: status 115
Child 1215 exited: status 116
rice11$
```

```
rice11$ ./reap-as-they-exit
Child 1453 exited: status 115
Child 1449 exited: status 111
Child 1448 exited: status 110
Child 1450 exited: status 112
Child 1451 exited: status 113
Child 1452 exited: status 114
Child 1455 exited: status 117
Child 1454 exited: status 116
rice11$
```

- When **waitpid** returns -1, it sets a global variable called **errno** to the constant **ECHILD** to signal that **waitpid** returned -1 because all child processes have terminated. That's the "error" we want.
- FYI: the myth machines were updated recently, so running this program on a myth likely will result in the children being reaped in the order they were created! Try running the program on a rice machine (connect via `ssh yourusername@rice.stanford.edu`) to see different behavior.

Reaping Several Children In Order

- We can do the same thing we did in the first program, but monitor and reap the child processes in the order they are forked (full program with error checking right here):

```
int main(int argc, char *argv[]) { // reap-in-fork-order.c
  pid_t children[8];
  for (size_t i = 0; i < 8; i++) {
    if ((children[i] = fork()) == 0) exit(110 + i);
  }
  for (size_t i = 0; i < 8; i++) {
    int status;
    pid_t pid = waitpid(children[i], &status, 0);
    assert(pid == children[i]);
    assert(WIFEXITED(status) && (WEXITSTATUS(status) == (110 + i)));
    printf("Child with pid %d accounted for (return status of %d).\n",
           children[i], WEXITSTATUS(status));
  }
  return 0;
}
```

Reaping Several Children In Order (continued)

```
myth60$ ./reap-as-they-exit
Child with pid 4689 accounted for (return status of 110).
Child with pid 4690 accounted for (return status of 111).
Child with pid 4691 accounted for (return status of 112).
Child with pid 4692 accounted for (return status of 113).
Child with pid 4693 accounted for (return status of 114).
Child with pid 4694 accounted for (return status of 115).
Child with pid 4695 accounted for (return status of 116).
Child with pid 4696 accounted for (return status of 117).
myth60$
```

- This version spawns and reaps processes in some first-spawned-first-reaped (FSFR) manner.
- In general, the child processes aren't required to exit in FSFR order.
- In theory, the first child could finish last, and the reap loop could be held up on its very first iteration until the first child really is done. But in our example, the process zombies are reaped in the order they were forked.
- Above is a sample run of the **`reap-in-fork-order`** executable. Only the pids change between runs.

Recap: Why Create Processes?

- So, now we know how to create processes... but why would we do that in the first place? There are three major reasons:
  - performance (ability to use multiple CPUs)
  - security (isolation of possibly sensitive components of an application)
  - **starting executables from disk**

Enter execvp!

About execvp

- It is possible to have a **fork**ed process simply do other work that you program. In other words, you have two processes, each doing work concurrently, and you've programmed the code for both processes. These are the examples we've seen so far.

- However, this is actually *not* the most common use for **fork**. Most often, a programmer wants to run a *completely separate program*, but wants to maintain control over the program, and may also (quite frequently) want to send data to the program through **stdin** and capture the output of the program through its **stdout**.
- E.g. the shell program: when you type a command, it executes that program and waits for it to end.

About execvp (continued)

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

Enter the **execvp** system call!

- **execvp** effectively reboots a process to run a different program from scratch. Here is the prototype:
  - **path** identifies the name of the executable to be invoked.
  - **argv** is the argument vector that should be funneled through to the new executable's **main** function.
  - For the purposes of CS110, **path** and **argv[0]** 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 in the calling process.
    - Thus unlike **fork** which is called once but returns twice, **execvp** is called once and never returns.
  - **execvp** has many variants (**execle**, **execlp**, and so forth. Type **man execvp** to see all of them). We generally rely on **execvp** in this course.

About execvp (continued)

```
static int mysystem(const char *command)
```

- Our first example using **execvp** is our implementation of the function **mysystem** to emulate the behavior of the libc function called **system**.
- The function 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 the negative of that signal number (e.g. -11 for **SIGSEGV**).

About execvp (continued)

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

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

- **`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 terminates via the **`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.

About execvp (continued)

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

```
static const size_t kMaxLine = 2048;
int main(int argc, char *argv[]) {
  char command[kMaxLine];
  while (true) {
    printf("> ");
    fgets(command, kMaxLine, stdin);
    if (feof(stdin)) break;
    command[strlen(command) - 1] = '\0'; // overwrite '\n'
    printf("retcode = %d\n", mysystem(command));
  }

  printf("\n");
  return 0;
}
```

- **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 when using this function so we need to chomp that newline off before calling **mysystem**. To exit the loop, press CTRL-D to send EOF.
- For a more involved version of a shell we implemented, review this code (shown on slides 29-31 when you download the PDF of these slides). You'll be writing an even more robust shell for a future assignment.

About execvp (continued)

- 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 third 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**, etc.) you've been using since you started using Unix.
- Note that the program that **execvp** loads and runs is in the context of the process it was run in.
  - It overwrites the address space of the current process but it doesn't create a new process.
  - The new program still has the same PID and it inherits all of the file descriptors that were open at the time of the call to **execvp**.

BONUS: Another shell example: `simplesh`

Let's work through the implementation of a more sophisticated shell: **`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 child proceeds to replace its own 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. That means we can launch other programs from the foreground before that background process finishes.
- 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 lecture).

BONUS: Another shell example: `simplesh`

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

```
// simplesh.c
int main(int argc, char *argv[]) {
    while (true) {
        char command[kMaxCommandLength + 1];
        readCommand(command, kMaxCommandLength);
        char *arguments[kMaxArgumentCount + 1];
        int count = parseCommandLine(command, arguments, kMaxArgumentCount);
        if (count == 0) continue;
        if (strcmp(arguments[0], "quit") ==) break; // hardcoded builtin to exit shell
        bool isbg = strcmp(arguments[count - 1], "&") == 0;
        if (isbg) arguments[--count] = NULL; // overwrite "&"
        pid_t pid = fork();
        if (pid == 0) execvp(arguments[0], arguments);
        if (isbg) { // background process, don't wait for child to finish
            printf("%d %s\n", pid, command);
        } else { // otherwise block until child process is complete
            waitpid(pid, NULL, 0);
        }
    }
    printf("\n");
    return 0;
}
```

BONUS: Another shell example: `simplesh`

- Take a closer look at the highlighted line. Do you see the problem? We'll fix it in a future lecture!

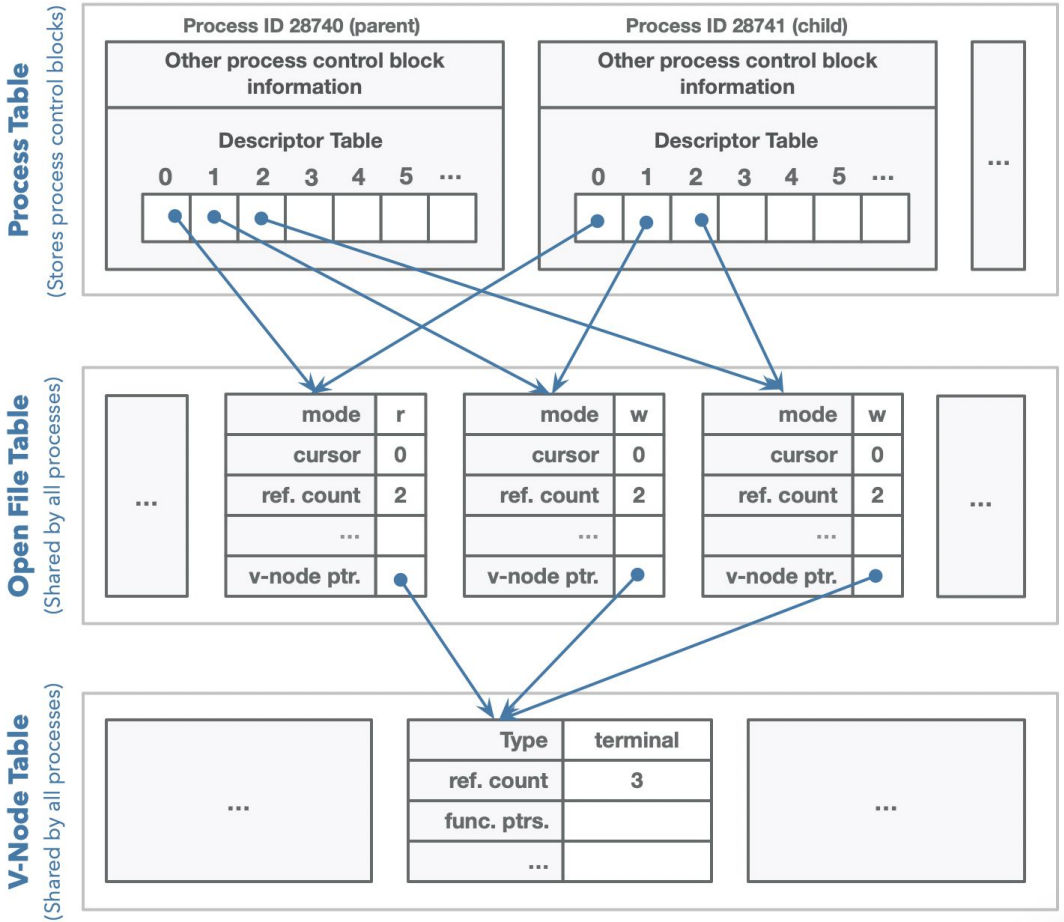
```
// simplesh.c
int main(int argc, char *argv[]) {
    while (true) {
        char command[kMaxCommandLength + 1];
        readCommand(command, kMaxCommandLength);
        char *arguments[kMaxArgumentCount + 1];
        int count = parseCommandLine(command, arguments, kMaxArgumentCount);
        if (count == 0) continue;
        if (strcmp(arguments[0], "quit") ==) break; // hardcoded builtin to exit shell
        bool isbg = strcmp(arguments[count - 1], "&") == 0;
        if (isbg) arguments[--count] = NULL; // overwrite "&"
        pid_t pid = fork();
        if (pid == 0) execvp(arguments[0], arguments);
        if (isbg) { // background process, don't wait for child to finish (we should, and we'll learn how later!)
            printf("%d %s\n", pid, command);
        } else { // otherwise block until child process is complete
            waitpid(pid, NULL, 0);
        }
    }
    printf("\n");
    return 0;
}
```

Introducing Pipes

Now it's time to 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.

Descriptors and `fork`

- Recall that the parent process' file descriptor table is cloned on **`fork`** and preserved across **`execvp`** boundaries.
- Thus on **`fork`**, a child process inherits the stdout linked to the terminal, and if it calls **`execvp`**, the new executable can still write to the terminal.
- Remember these virtual files stored in the v-node table? We can now introduce another type of virtual file: a **pipe**. These are **unnamed pipes** (we won't discuss named pipes in this class) that allow the output of one source to be passed as input to another source. They are an important type of **interprocess communication**.



Understanding Data Flow

Reading: My data flow blog post

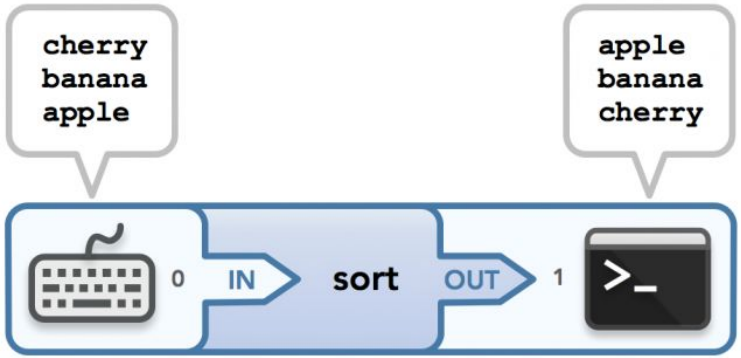
- The below figure represents the default setup of the standard input, output, and error streams.

- I use the words “in” and “out” to represent data that goes into one area and out of another area, respectively. The keyboard (used to model the “readable” terminal) passes data to the program that runs the command (from the command’s program’s perspective, it receives input via stdin), and that program sends output to the “writable” terminal via stdout.
- Generally, data flowing “into” something is considered **input** (and is being read in from a source via a file descriptor) and data flowing “out” of something is considered **output** (and is being written out to a source via a file descriptor).
    - Put another way: input is read from somewhere; output is written somewhere.

Understanding Data Flow

Reading: My data flow blog post

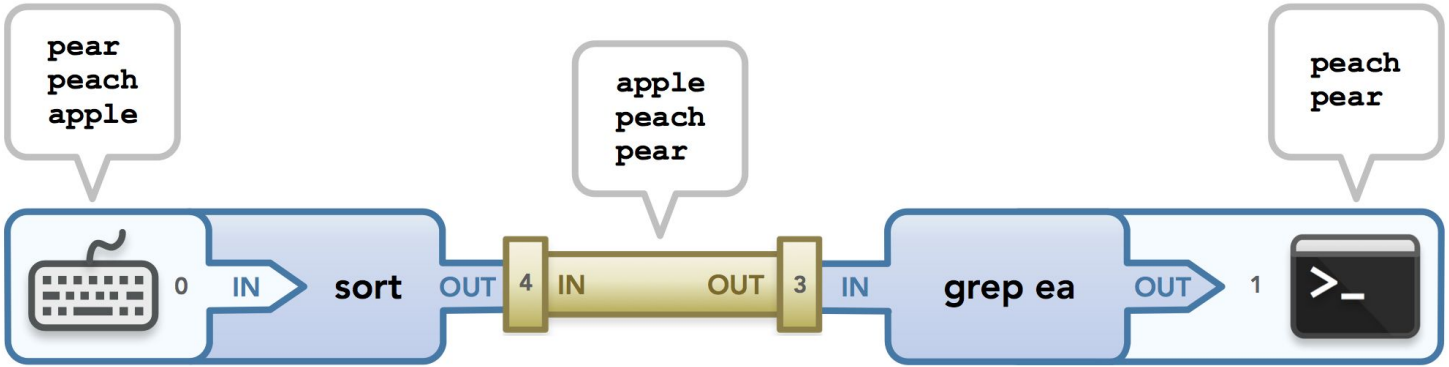
- Simple example (when using **sort** this way, hit CTRL-D to let it know that you're done sending input):




```
myth60$ sort
cherry
banana
apple
apple
banana
cherry
myth60$
```

All About Pipes

Reading: My data flow blog post

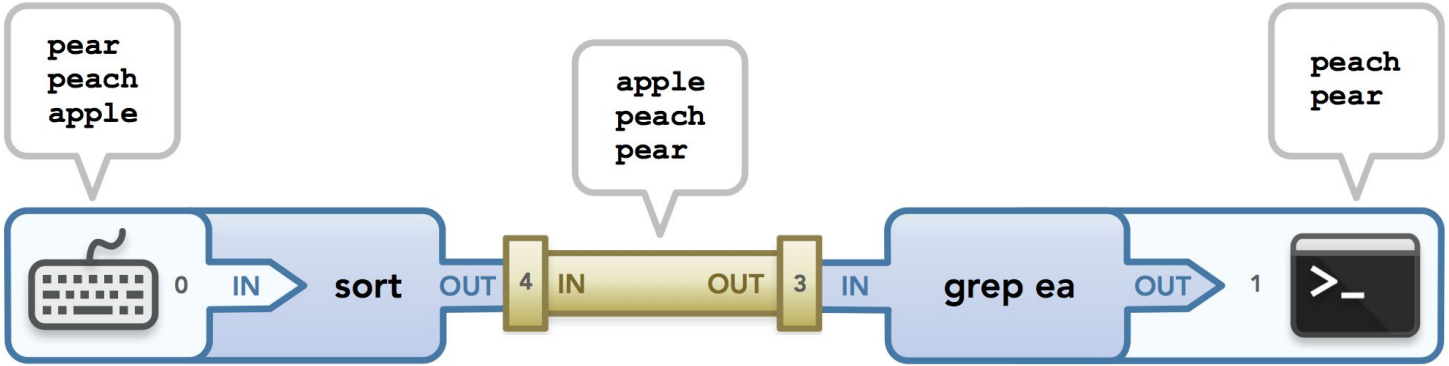



```
echo -e "peach\npear\napple" | sort | grep ea
```

- **Pipes** allow data from one process to be passed to another (via **unidirectional** data flow) so that commands can be chained together by their streams.
- This chaining of processes can be represented by a **pipeline**: commands in a pipeline are connected via pipes, where data is shared between processes by flowing from one end of each pipe to the other.
- Since each command in the pipeline is run in a separate process, each with a separate memory space, we need a way to allow those processes to communicate with each other. This is is exactly the behavior that the **`pipe()`** system call provides.

All About Pipes

Reading: My data flow blog post

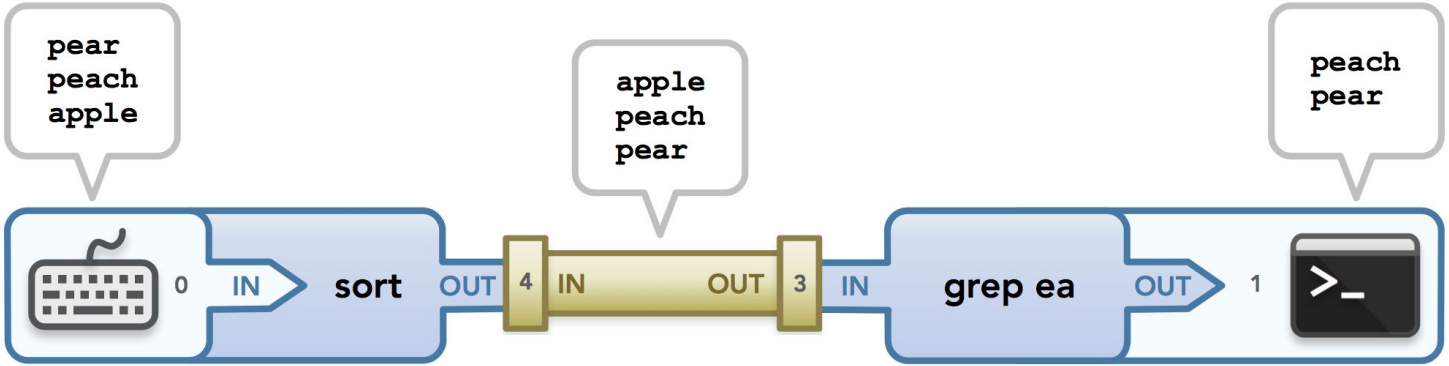



```
echo -e "peach\npear\napple" | sort | grep ea
```

- Physical pipes are naturally a great analogy for this abstraction.
- We can think of the data stream that starts in one process as water in an isolated environment, and the only way to allow the water to flow to the environment of the next process is to connect the environments with a pipe.
- In this way, the water (data) flows from the first environment (process) into the pipe, filling up the pipe with all its water and then draining its water into the other environment.

All About Pipes

Reading: My data flow blog post

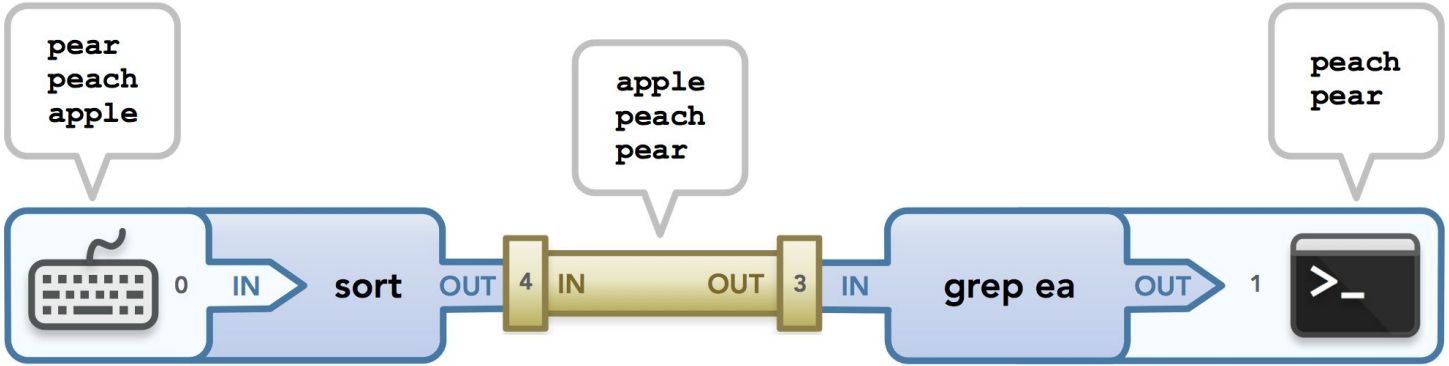



```
echo -e "peach\npear\napple" | sort | grep ea
```

- Ignoring file descriptors 3 and 4 for a moment, look at the “in” and “out” words: we see that data flows **out of** the **`sort`** process and **into** the pipe, where it then is passed **out of** the pipe and **into** the grep process. “In” and “out” are phrased based on the context they are used in: either inside the pipe or outside of it.

All About Pipes

Reading: My data flow blog post



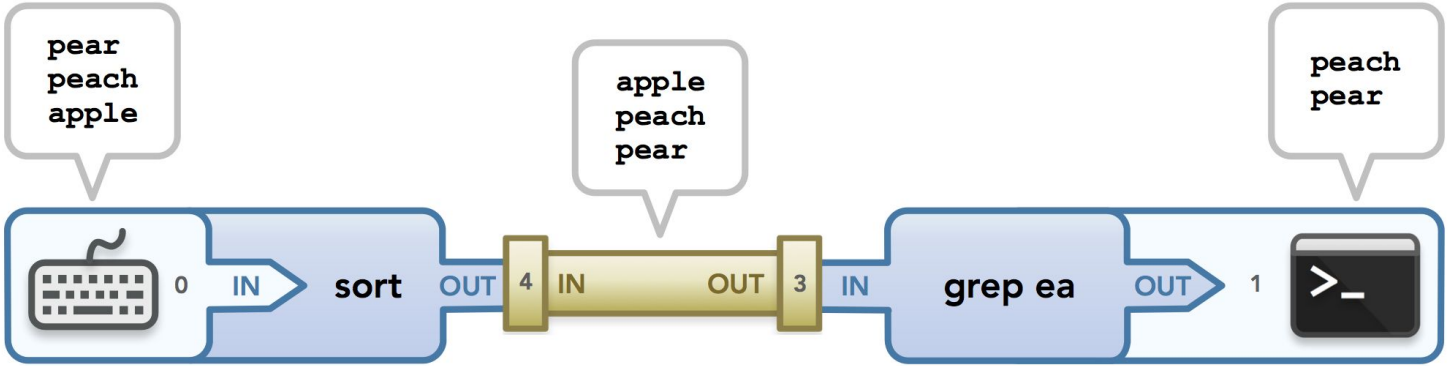

```
echo -e "peach\npear\napple" | sort | grep ea
```

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

- 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]**.
- **pipe** is particularly useful for allowing parent processes to communicate with spawned child processes because the file descriptor table of the parent is cloned in the child. That means the open file table entries referenced by the parent's pipe endpoints are also referenced by the child's copies of them.

All About Pipes

Reading: My data flow blog post




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

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

Important, and easy to get confused: The read and write actions defined by a `pipe` call are from the perspective of the two processes using the pipe, not the pipe itself!

This is why you see fd 3 (the read end) on the right of the pipe and fd 4 (the write end) on the left of the pipe. It may seem "backwards", but when you think about it from the perspective of the processes using the pipe, it makes sense.

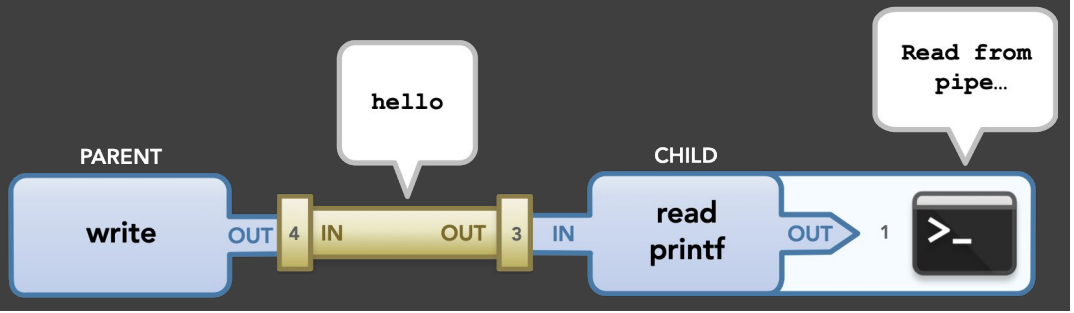
All About Pipes

Reading: My data flow blog post

Let's show how **pipe** works and how data can be passed from one process to another:

```c
int main(int argc, char *argv[]) {
  int fds[2];
  pipe(fds); // assume that fds = [3, 4]
  pid_t pid = fork();
  if (pid == 0) {
    close(fds[1]); // without this, program hangs
    printf("Read from pipe bridging processes: ");
    char buffer[6];
    int nbytes;
    while ((nbytes = read(fds[0], buffer, sizeof(buffer))) > 0)
      printf("%s", buffer);
    printf("\n");
    close(fds[0]);
    return 0;
  }

  close(fds[0]);
  write(fds[1], "hello", 6);
  close(fds[1]); // without this, program hangs
  waitpid(pid, NULL, 0);
  return 0;
}
```



All About Pipes

Reading: My data flow blog post

How do **pipe** and **fork** work together in the previous 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]**.

All About Pipes

Reading: My data flow blog post

How do **`pipe`** and **`fork`** work together in the previous 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.
- Remember: to check for file descriptor leaks in valgrind, add the following flag: **`--track-fds=yes`**

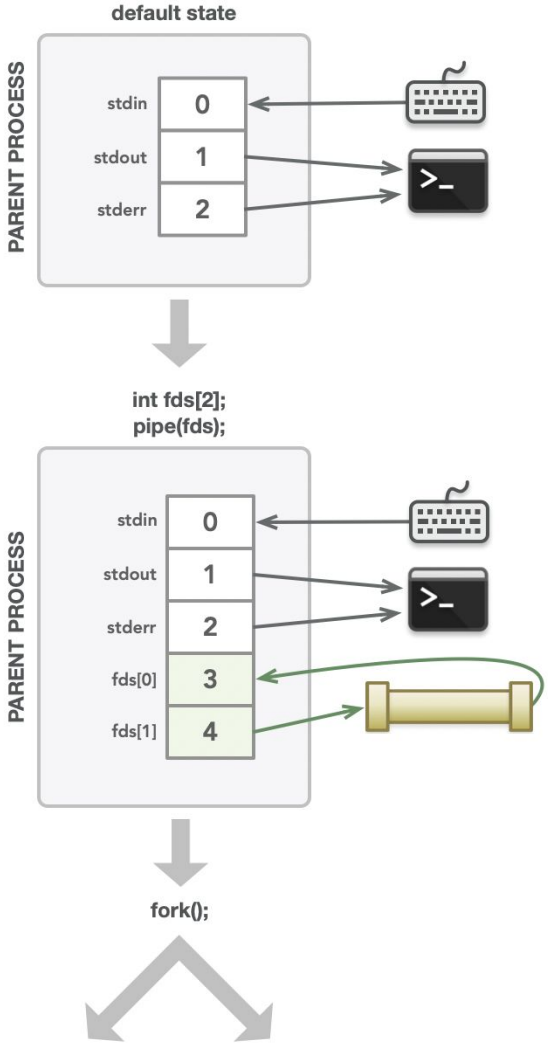
All About Pipes

Here are some more detailed illustrations of the previous code example.

The thin arrows represent data flow.

Note how the **`pipe`** call populates an array with file descriptors 3 and 4 in this example. 3 is hooked up so that a process can read data from that end of the pipe (the read end) and 4 is hooked up so that a process can write data to that end of the pipe (the write end).

Reading: My data flow blog post

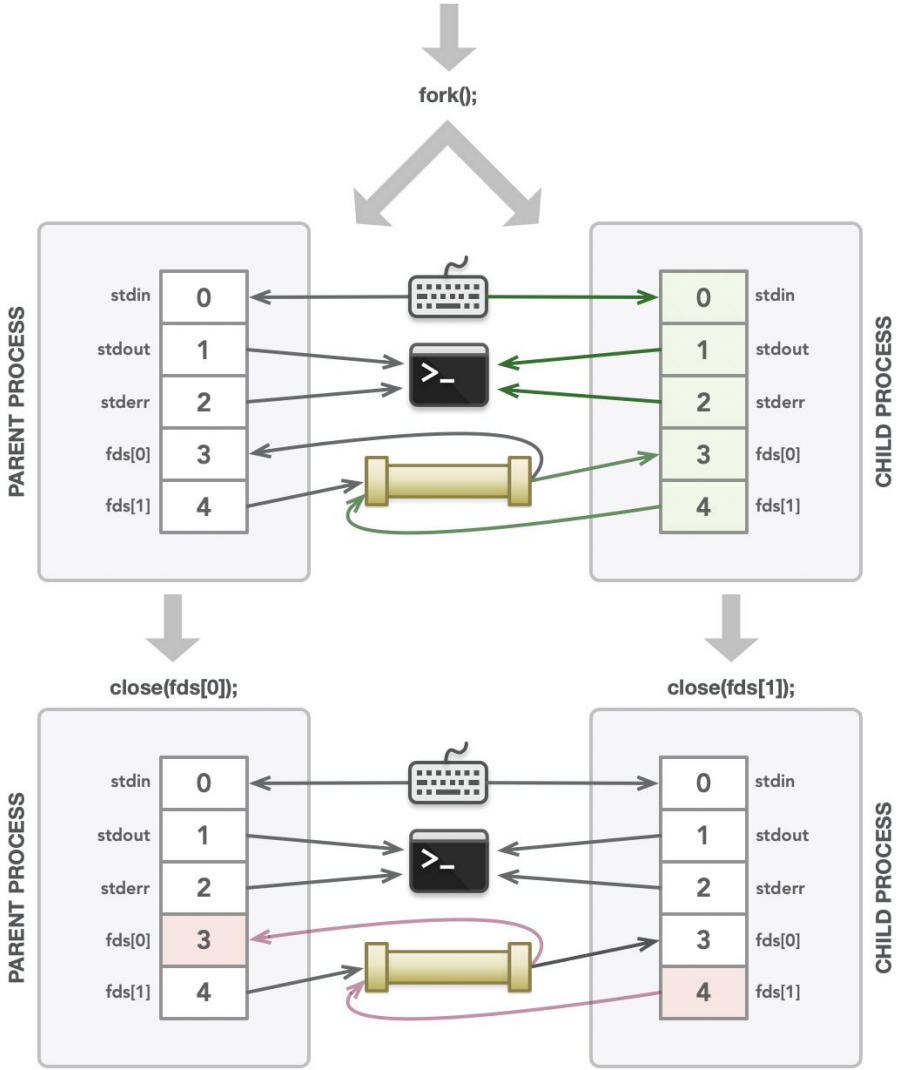


All About Pipes

Reading: My data flow blog post

Calling **fork** will create a copy of the parent process: the child. Recall that the file descriptor table is also copied to the child as shown.

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. (Closes are shown with red arrows.)



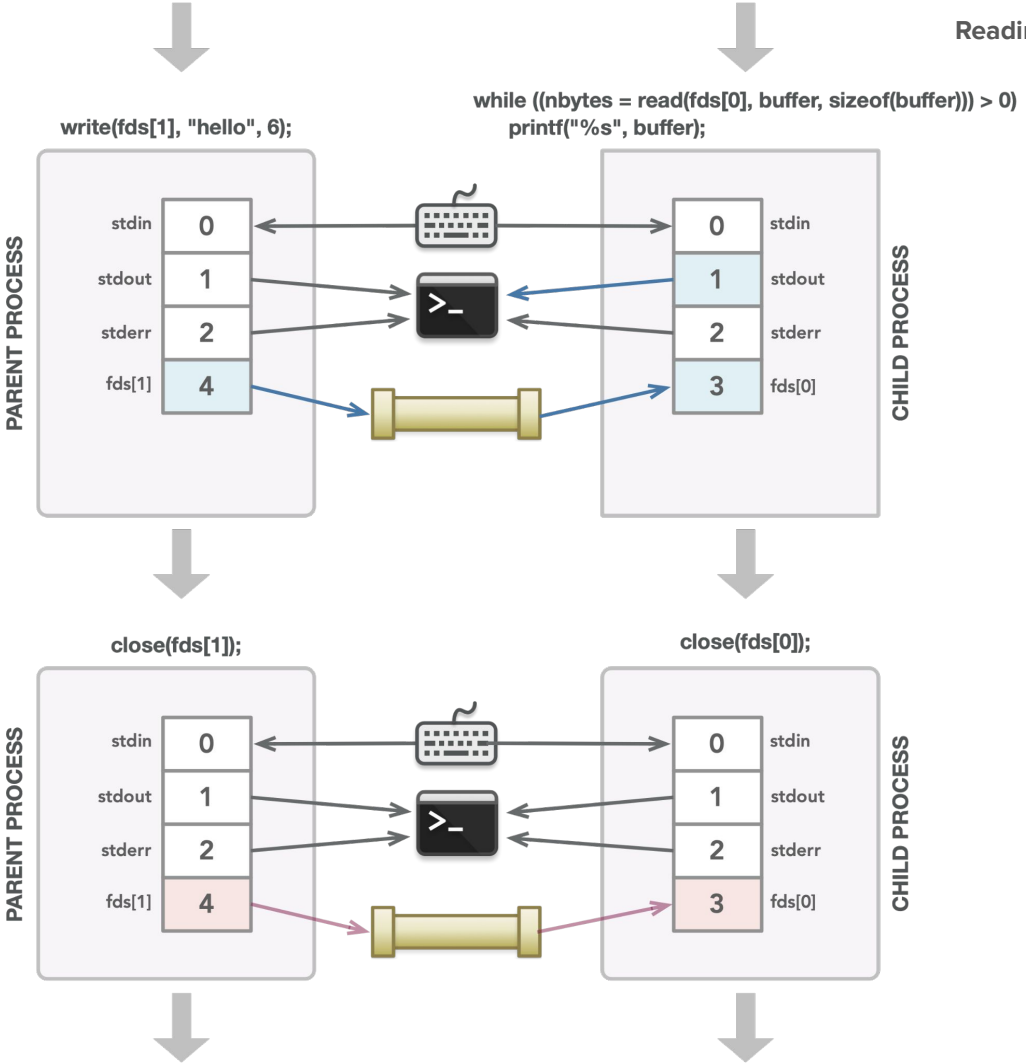
All About Pipes

Reading: My data flow blog post

The parent writes a string to the write end of the pipe.

The child reads in data until the parent closes the write end, which will tell the **`read`** call that no more data will be written since at that point, both pointers to the write end of the pipe have been closed.

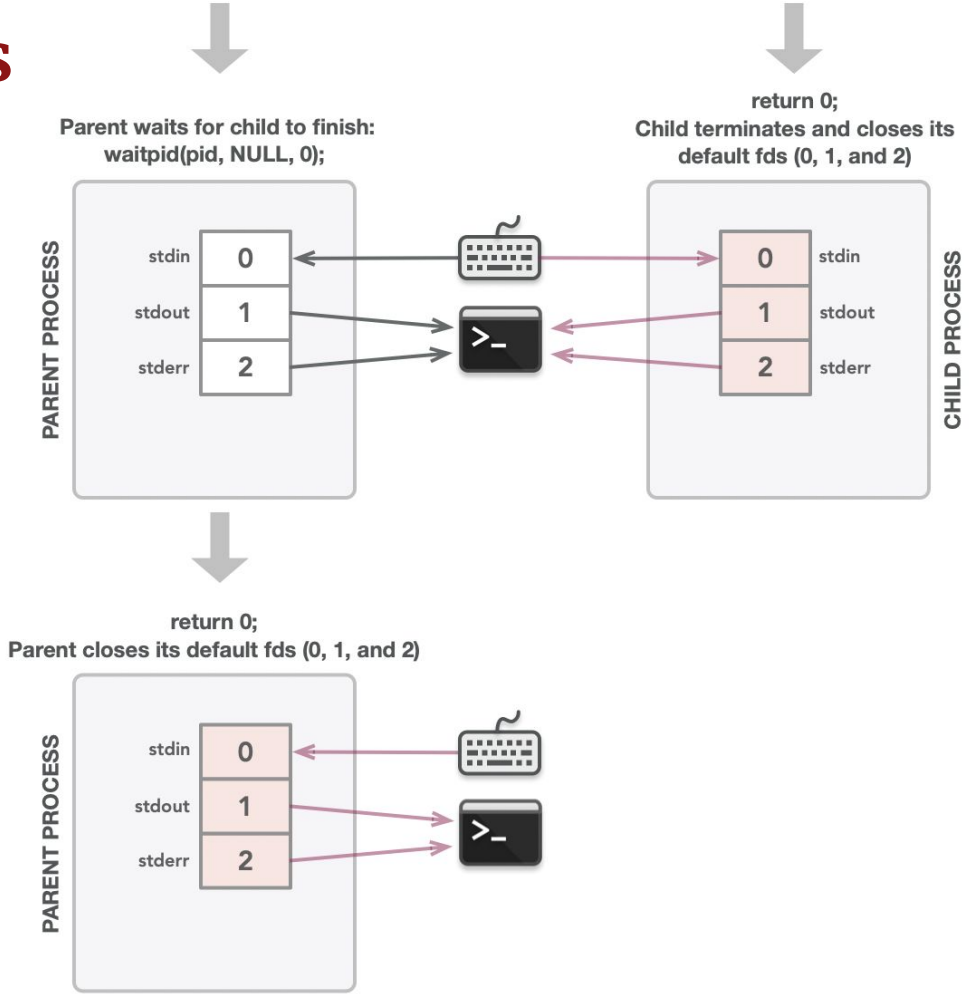
Once the child is done reading data, it closes the read end of the pipe to clean up after itself.



All About Pipes

Reading: My data flow blog post

After the child finishes, the parent can continue, at which point it returns and its default descriptors can be closed (the three default descriptors are cleaned up automatically).

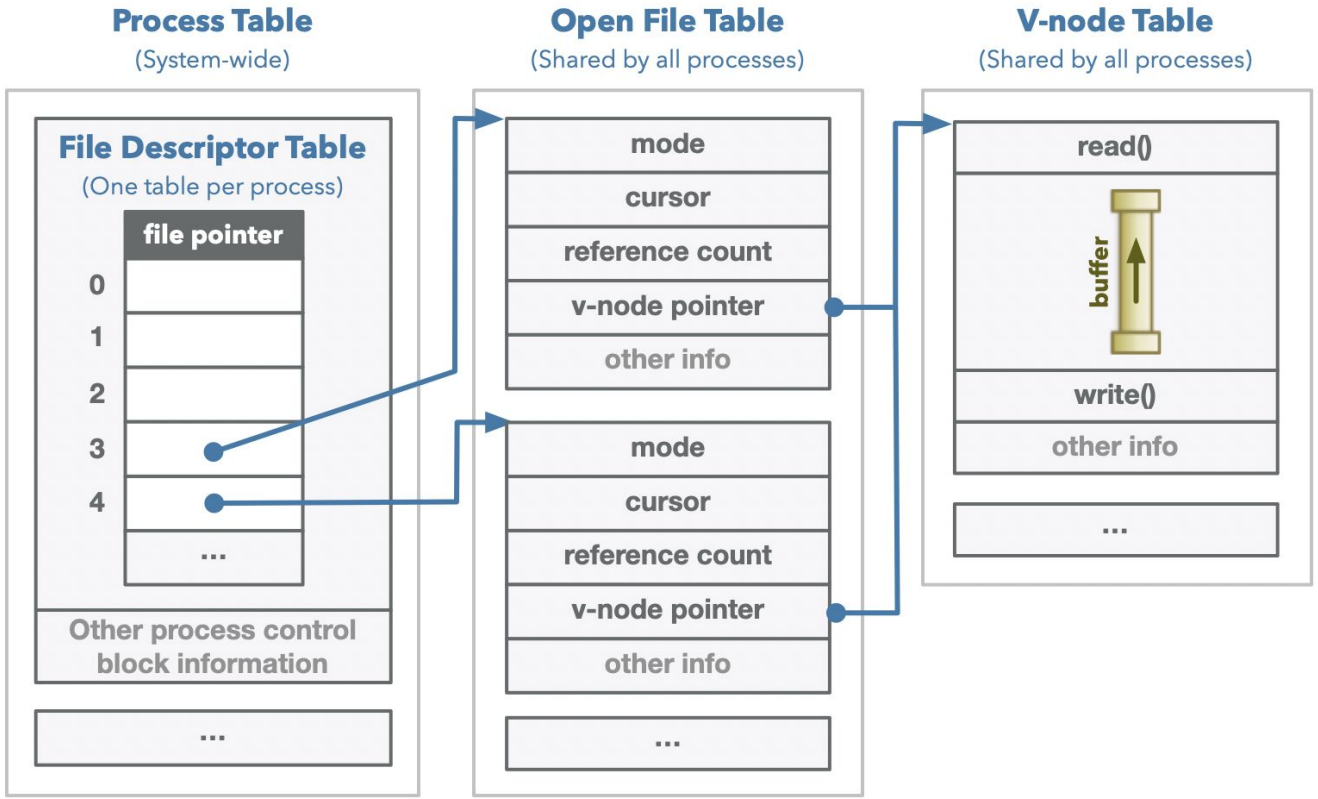


All About Pipes

Reading: My data flow blog post

Here's a way to visualize how the pipe is set up by the kernel. It is a virtual "file" stored in memory, not in the actual filesystem.

The **`read`** and **`write`** functions are inside the v-node (remember: the v-node entries store function pointers), where **`read`** pulls data out of the buffer and **`write`** puts stuff in. A vnode table entry is really a function lookup table with some associated data; the associated data would be a pointer to the buffer, and the read/write functions would know how to use the buffer.



Introducing The dup2 System Call

Reading: My data flow blog post

```
int dup2(int oldfd, int newfd);
```

- A pipe can be used to allow two processes (or even the same process!) to exchange data in one direction. However, a process may want to change the source of the data that gets written to the pipe or change where the data flows out of the pipe. The **dup2** call supports this.
- **dup2()** copies descriptor table entry **oldfd** to descriptor table entry **newfd**, overwriting the previous contents of descriptor table entry **newfd**. If newfd was already open, then **dup2** closes **newfd** before it copies **oldfd**.

The dup2 System Call: An Example

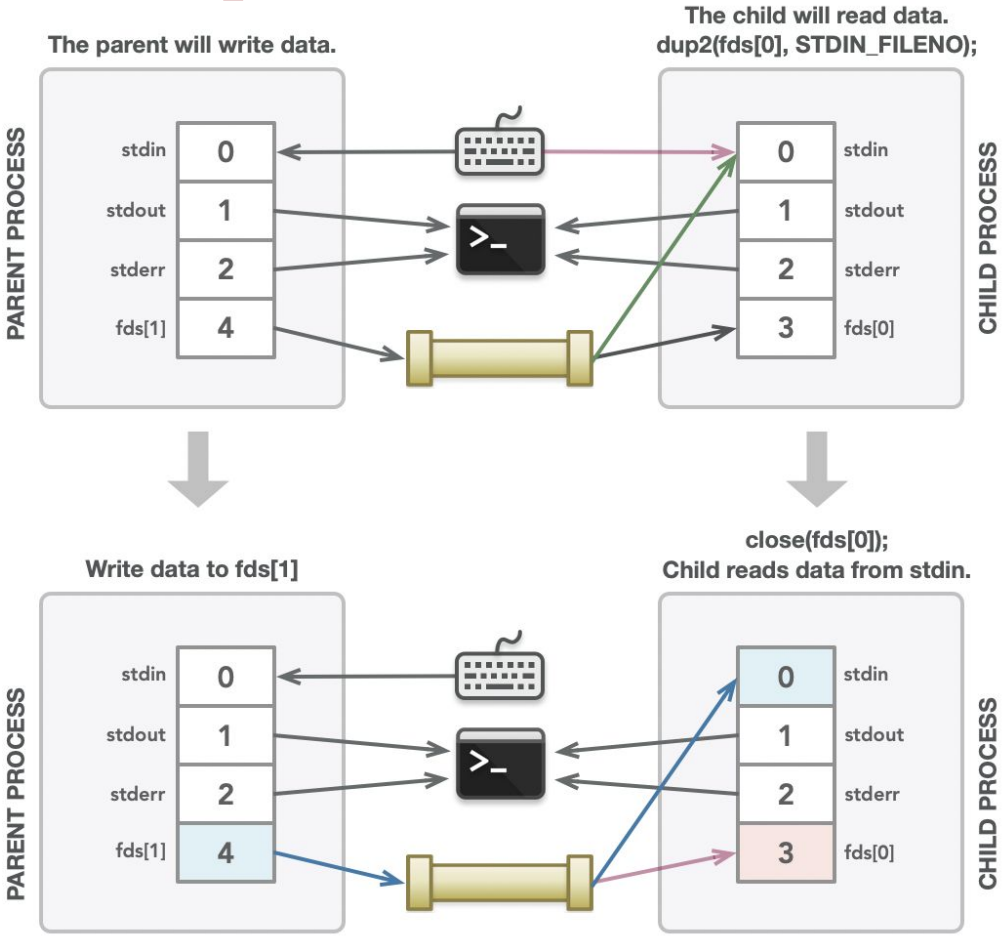
Reading: My data flow blog post

```
dup2(fds[0], STDIN_FILENO);
```

An important detail about the way **dup2()** works is that it will first close its second parameter, which is a file descriptor, if necessary.

Thus in this example, **stdin** (which is open by default) is first closed, which will remove its reference to the default terminal (keyboard) file. Then the child's **stdin** will be able to receive data in the pipe via **fds[0]** instead of from the keyboard. Remember that by default, **stdin** is where a program expects to get its input from, and **stdout** is where it expects to send it.

A use case for this is if the child is running the first command in a pipeline, so it will want to get input the default way via stdin but still get data from parent via a pipe.



Putting it all together! 🥳

fork | execvp | waitpid | pipe | dup2

Subprocess Routine

Here's a more sophisticated example that combines what we've learned so far:

- 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 is right here):

```
typedef struct {
  pid_t pid;
  int supplyfd;
} subprocess_t;

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**.

Subprocess Routine

Let's first implement a test harness to illustrate how **subprocess** should work so 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:

```
int main(int argc, char *argv[]) {
  subprocess_t sp = subprocess("/usr/bin/sort");
  const char *words[] = {
    "felicity", "umbrage", "susurration", "halcyon",
    "pulchritude", "ablution", "somnolent", "indefatigable"
  };
  for (size_t i = 0; i < sizeof(words)/sizeof(words[0]); i++) {
    dprintf(sp.supplyfd, "%s\n", words[i]); // dprintf prints to a file descriptor
  }
  close(sp.supplyfd); // necessary to communicate end-of-input
  int status;
  pid_t pid = waitpid(sp.pid, &status, 0);
  return pid == sp.pid && WIFEXITED(status) ? WEXITSTATUS(status) : -127; // 127: command not found
}
```

Subprocess Routine

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** to indicate that no more data will ever be written through that descriptor. The reference count of the relevant open file entry referenced by **supplyfd** is demoted from 1 to 0 with that **close** call. That sends an **EOF** to the process reading 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**.

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

Subprocess Routine

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]); // child isn't writing to pipe
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    char *argv[] = {"/bin/sh", "-c", (char *) command, NULL};
    execvp(argv[0], argv); // assume success, which means that execvp doesn't return
  }
  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.

End of Lecture 6