CS110 Lecture 8: Pipes and Interprocess Communication, Part 1

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

Instructors: Nick Troccoli and Jerry Cain

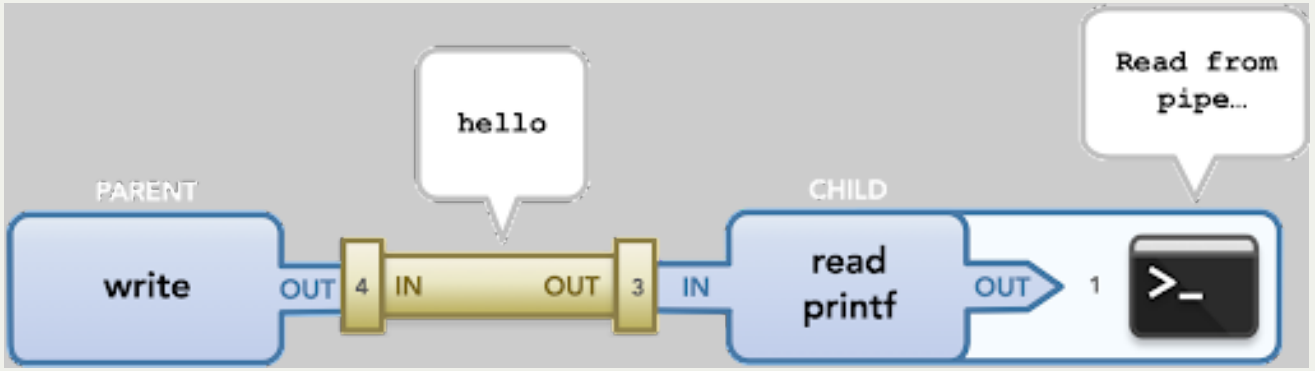
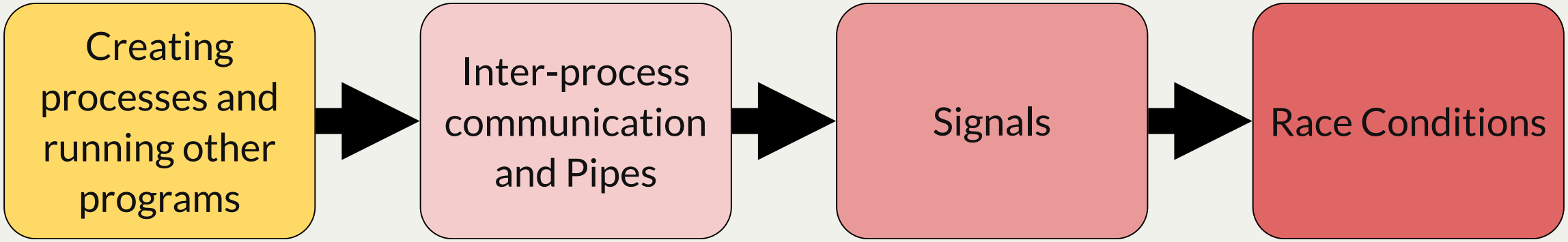


Illustration courtesy of Roz Cyrus.

PDF of this presentation

CS110 Topic 2: How can our program create and interact with other programs?

Learning About Processes



Lecture 6/7 | **This/next lecture** | **Lecture 10/11** | **Lecture 11**

assign3: implement multiprocessing programs like "trace" (to trace another program's behavior) and "farm" (parallelize tasks)

assign4: implement your own shell!

Learning Goals

- Get more practice with using **fork()** and **execvp**
- Learn about **pipe** and **dup2** to create and manipulate file descriptors
- Use pipes to redirect process input and output

Lecture Plan

- Review: our first shell
- Running in the background
- Introducing Pipes
  - What are pipes?
  - Pipes between processes

Lecture Plan

- **Review: our first shell**
- Running in the background
- Introducing Pipes
  - What are pipes?
  - Pipes between processes

`fork()`

- A system call that creates a new *child process*
- The "parent" is the process that creates the other "child" process
- From then on, both processes are running the code after the fork
- The child process is *identical* to the parent, except:
  - it has a new Process ID (PID)
  - for the parent, fork() returns the PID of the child; for the child, fork() returns 0
  - fork() is **called once**, but **returns twice**

```
1 pid_t pidOrZero = fork();
2 // both parent and child run code here onwards
3 printf("This is printed by two processes.\n");
```

waitpid()

A function that a parent can call to wait for its child to exit:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **pid:** the PID of the child to wait on, or -1 to wait on any of our children
- **status:** where to put info about the child's termination (or NULL)
- **options:** optional flags to customize behavior (always 0 for now)

The function returns when the specified **child process** exits.

- the return value is the PID of the child that exited, or -1 on error (e.g. no child to wait on)
- If the child process has already exited, this returns immediately - otherwise, it blocks
- It's important to wait on all children to clean up system resources

`execvp()`

`execvp` is a function that lets us run *another program* in the current process.

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

It runs the executable at the specified **path**, *completely cannibalizing the current process.*

- If successful, **execvp never returns** in the calling process
- If unsuccessful, **execvp** returns -1

To run another executable, we must specify the (NULL-terminated) arguments to be passed into its **main** function, via the **argv** parameter.

- For our programs, **path** and **argv[0]** will be the same

`execvp` has many variants (**`execle`**, **`execlp`**, and so forth. Type **`man execvp`** for more). We rely on **`execvp`** in CS110.

Revisiting `mysystem`

mysystem is our own version of the built-in function **system**.

- It takes in a terminal command (e.g. "ls -l /usr/class/cs110"), executes it in a separate process, and returns when that process is finished.
  - We can use **fork** to create the child process
  - We can use **execvp** in that child process to execute the terminal command
  - We can use **waitpid** in the parent process to wait for the child to terminate

Revisiting `first-shell`

```
1  int main(int argc, char *argv[]) {
2      char command[kMaxLineLength];
3      while (true) {
4          printf("> ");
5          fgets(command, sizeof(command), stdin);
6
7          // If the user entered Ctl-d, stop
8          if (feof(stdin)) {
9              break;
10         }
11
12         // Remove the \n that fgets puts at the end
13         command[strlen(command) - 1] = '\0';
14
15         int commandReturnCode = mysystem(command);
16         printf("return code = %d\n", commandReturnCode);
17     }
18
19     printf("\n");
20     return 0;
21 }
```

`first-shell-soln.c`

Our **first-shell** program is a loop in **main** that parses the user input and passes it to **mysystem**.

Revisiting `first-shell`

```
1  static int mysystem(char *command) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          char *arguments[] = {"/bin/sh", "-c", command, NULL};
5          execvp(arguments[0], arguments);
6          // If the child gets here, there was an error
7          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
8      }
9
10     // If we are the parent, wait for the child
11     int status;
12     waitpid(pidOrZero, &status, 0);
13     return WIFEXITED(status) ? WEXITSTATUS(status) : -WTERMSIG(status);
14 }
```

`first-shell-soln.c`

`first-shell` Takeaways

- A shell is a program that repeats: read command from the user, execute that command
- In order to execute a program and continue running the shell afterwards, we fork off another process and run the program in that process
- We rely on **`fork`**, **`execvp`**, and **`waitpid`** to do this!
- Real shells have more advanced functionality that we will add going forward.
- For your fourth assignment, you'll build on this with your own shell, **`stsh`** ("Stanford shell") with much of the functionality of real Unix shells.

Lecture Plan

- Review: our first shell
- **Running in the background**
- Introducing Pipes
  - What are pipes?
  - Pipes between processes

Supporting Background Execution

Shells usually also let you run a command in the *background* by adding "&" at the end:

- e.g. **sort myfile.txt &** - create a sort process and run it in the background
- only difference is specifying & with command
- shell immediately re-prompts the user
- process doesn't know "foreground" vs. "background"; the "&" just specifies whether or not the shell waits

`first-shell-soln-bg.c`

Supporting Background Execution

Let's make an updated version of **mysystem** called **executeCommand**.

- Takes an additional parameter **bool inBackground**
  - If **false**, same behavior as **mysystem** (spawn child, execvp, wait for child)
  - If **true**, spawn child, execvp, but *don't wait for child*

`first-shell-soln-bg.c`

Supporting Background Execution

```
1  static void executeCommand(char *command, bool inBackground) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, either wait or return immediately
12     if (inBackground) {
13         printf("%d %s\n", pidOrZero, command);
14     } else {
15         waitpid(pidOrZero, NULL, 0);
16     }
17 }
```

first-shell-soln-bg.c

Supporting Background Execution

```
1  static void executeCommand(char *command, bool inBackground) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, either wait or return immediately
12     if (inBackground) {
13         printf("%d %s\n", pidOrZero, command);
14     } else {
15         waitpid(pidOrZero, NULL, 0);
16     }
17 }
```

Line 1: Now, the caller can optionally run the command in the background.

first-shell-soln-bg.c

Supporting Background Execution

```
1  static void executeCommand(char *command, bool inBackground) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, either wait or return immediately
12     if (inBackground) {
13         printf("%d %s\n", pidOrZero, command);
14     } else {
15         waitpid(pidOrZero, NULL, 0);
16     }
17 }
```

Lines 11-16: The parent waits on a foreground child, but not a background child.

first-shell-soln-bg.c

Supporting Background Execution

```
1  int main(int argc, char *argv[]) {
2      char command[kMaxLineLength];
3      while (true) {
4          printf("> ");
5          fgets(command, sizeof(command), stdin);
6
7          // If the user entered Ctl-d, stop
8          if (feof(stdin)) {
9              break;
10         }
11
12         // Remove the \n that fgets puts at the end
13         command[strlen(command) - 1] = '\0';
14
15         if (strcmp(command, "quit") == 0) break;
16
17         bool isbg = command[strlen(command) - 1] == '&';
18         if (isbg) {
19             command[strlen(command) - 1] = '\0';
20         }
21
22         executeCommand(command, isbg);
23     }
24
25     printf("\n");
26     return 0;
27 }
```

In main, we add two additional things:

- Check for the "quit" command to exit
- Allow the user to add "&" at the end of a command to run that command in the background

Note that a background child isn't reaped! This is a problem - one we'll learn how to fix soon.

first-shell-soln-bg.c

Lecture Plan

- Review: our first shell
- Running in the background
- Introducing Pipes
  - **What are pipes?**
  - Pipes between processes

Is there a way that the parent and child processes can communicate?

Interprocess Communication

- It's useful for a parent process to communicate with its child (and vice versa)
- There are two key ways we will learn to do this: **pipes** and **signals**
  - **Pipes** let two processes send and receive arbitrary data
  - **Signals** let two processes send and receive certain "signals" that indicate something special has happened.

Interprocess Communication

- It's useful for a parent process to communicate with its child (and vice versa)
- There are two key ways we will learn to do this: **pipes** and **signals**
  - **Pipes** let two processes send and receive arbitrary data
  - **Signals** let two processes send and receive certain "signals" that indicate something special has happened.

Pipes

- How can we let two processes send arbitrary data back and forth?
- A core Unix principle is modeling things as *files*. Could we use a "file"?
- **Idea:** a file that one process could write, and another process could read?
- **Problem:** we don't want to clutter the filesystem with actual files every time two processes want to communicate.
- **Solution:** have the operating system set this up for us.
  - It will give us two new file descriptors - one for writing, another for reading.
  - If someone writes data to the write FD, it can be read from the read FD.
  - It's *not actually a physical file on disk* - we are just using files as an abstraction

pipe()

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

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

```
1 $ ./pipe-demo
2 Message read: Hello, this message is coming through a pipe.
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

Lecture Plan

- Review: **fork()** and **execvp()**
- Running in the background
- Introducing Pipes
  - What are pipes?
  - Pipes between processes

`pipe()`

`pipe` can allow processes to communicate!

- The parent's file descriptor table is **replicated in the child** - both have pipe access (increasing reference counts in open file table)
- E.g. the parent can write to the "write" end and the child can read from the "read" end
- Because they're file descriptors, there's no global name for the pipe (another process can't "connect" to the pipe).
- Each pipe is uni-directional (one end is read, the other write)

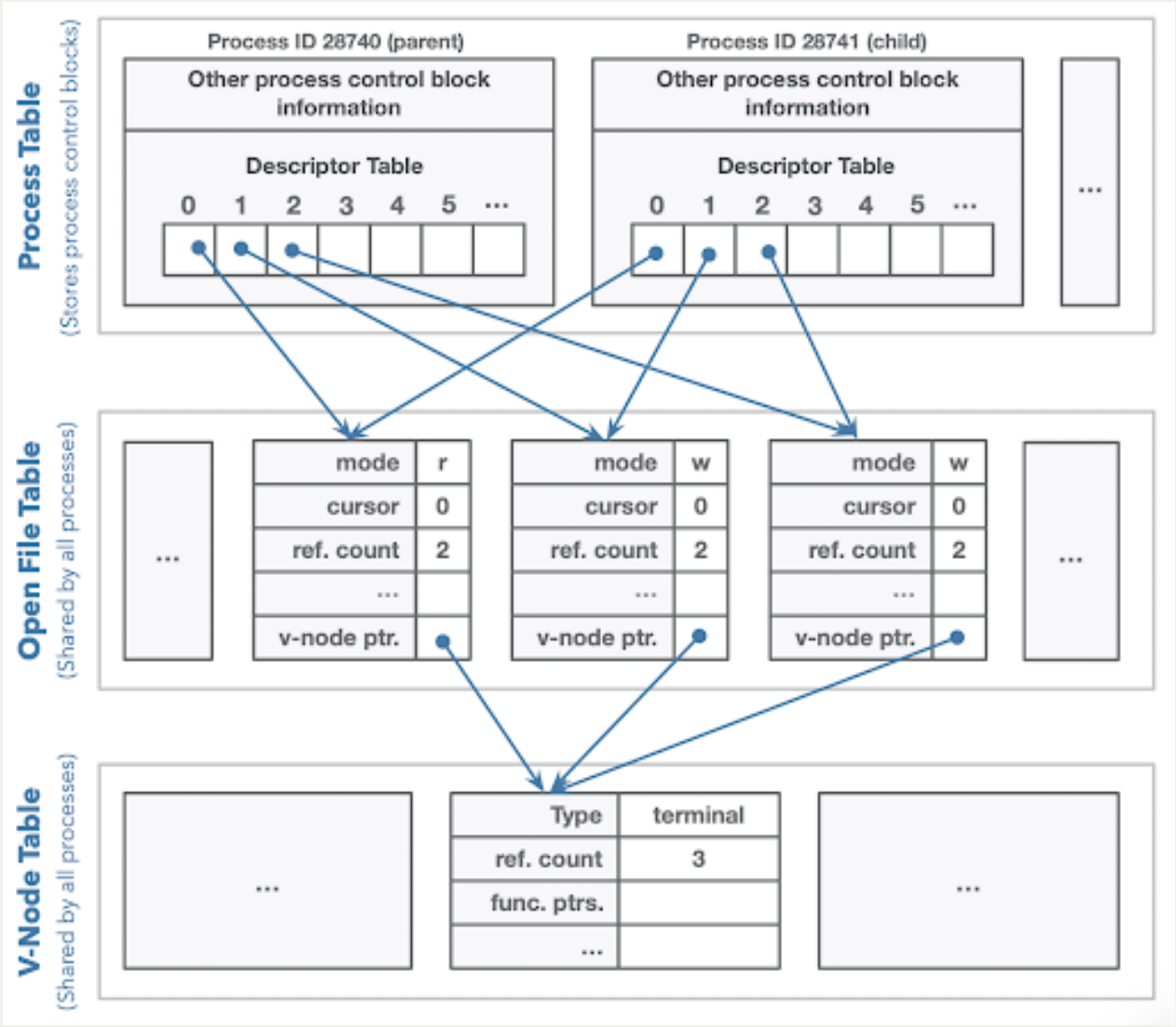


Illustration courtesy of Roz Cyrus.

Let’s write a program where the parent writes something to the pipe, and the child reads that from the pipe.

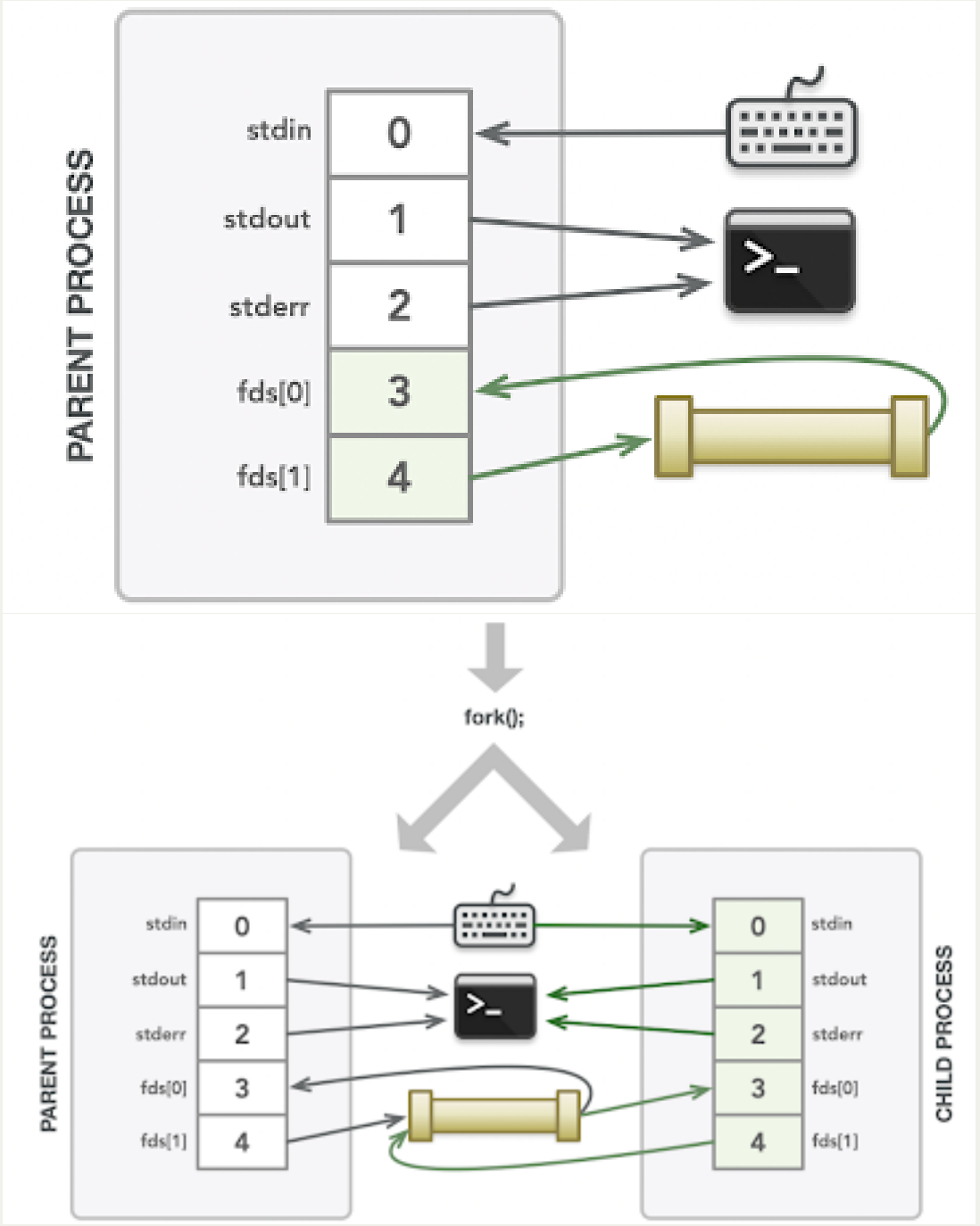


Illustration courtesy of Roz Cyrus.

Key Idea: because the pipe file descriptors are duplicated in the child, we need to close the 2 pipe ends ***in both the parent and the child.***

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Here's an example program showing how pipe works across processes (full program link at bottom).

parent-child-pipe.c

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Make a pipe just like before.

parent-child-pipe.c

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

The parent must close all its open FDs. It never uses the Read FD so we can close it here.

parent-child-pipe.c

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Write to the Write FD to send a message to the child.

parent-child-pipe.c

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

We are now done with the Write FD so we can close it here.

parent-child-pipe.c

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

We wait for the child to terminate.

parent-child-pipe.c

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

parent-child-pipe.c

Key Idea: when we call **fork**, the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

This duplication means the child's file descriptor table entries point to the same open file table entries as the parent. Thus, the open file table entries for the two pipe FDs both have reference counts of 2.

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

The child must close all its open FDs. It never uses the Write FD so we can close it here.

parent-child-pipe.c

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

parent-child-pipe.c

Read from the Read FD to read the message from the parent.

Key Idea: read() *blocks* until the bytes are available or there is no more to read (e.g. end of file or pipe write end closed). If the parent hasn't written yet, the child's call to **read()** will wait.

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

We are now done with the Read FD so we can close it here. Also print the received message.

parent-child-pipe.c

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Key Idea: the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

Here, right before the **fork** call, the parent has 2 open file descriptors (besides 0-2): the pipe Read FD and Write FD.

parent-child-pipe.c

Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Key Idea: the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

Therefore, when the child is spawned, it *also* has the same 2 open file descriptors (besides 0-2): the pipe Read FD and Write FD.

parent-child-pipe.c

Parent-Child Communication

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

Key Idea: the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

We should close FDs when we are done with them. The **parent** closes them here.

parent-child-pipe.c

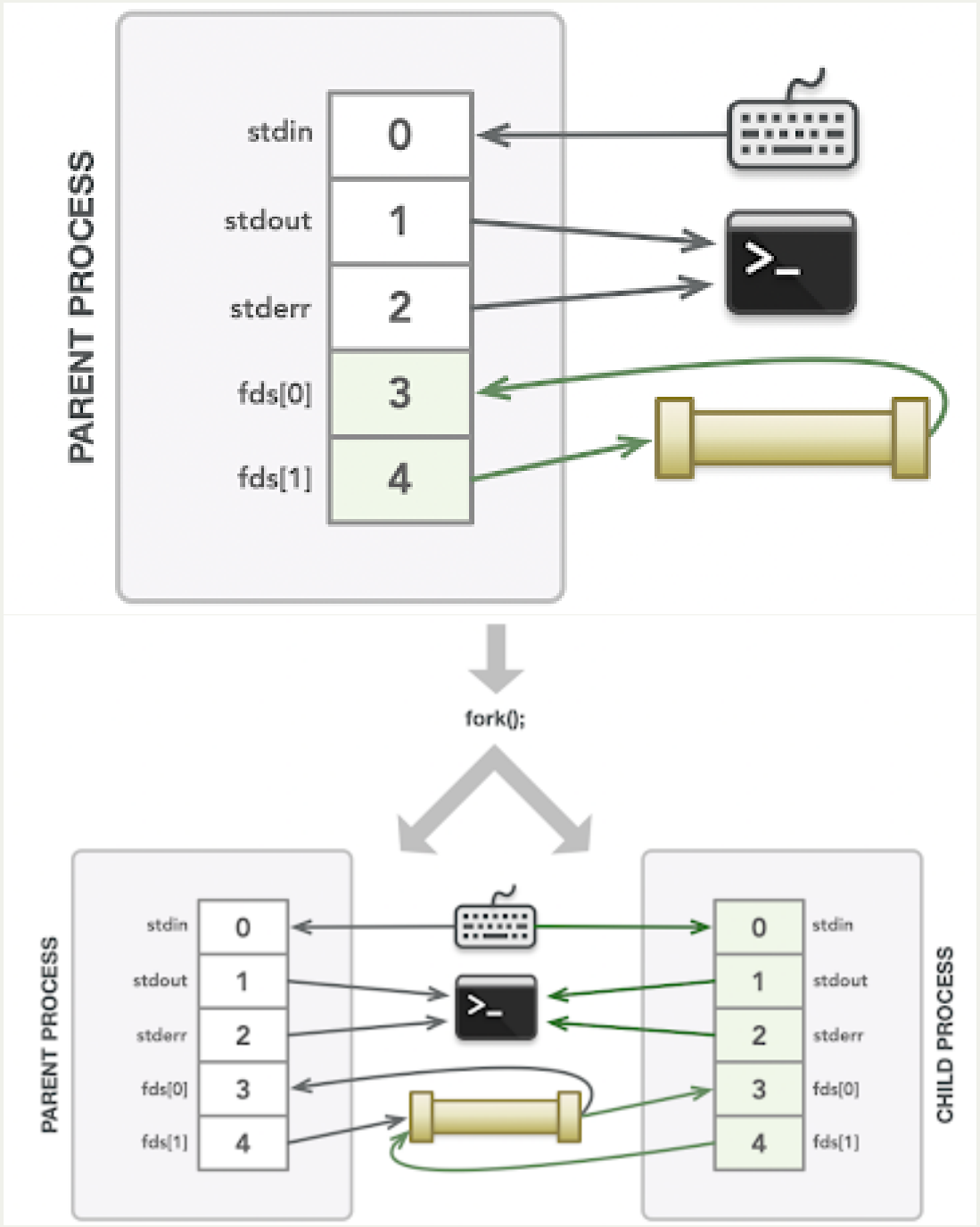
Parent-Child Communication

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

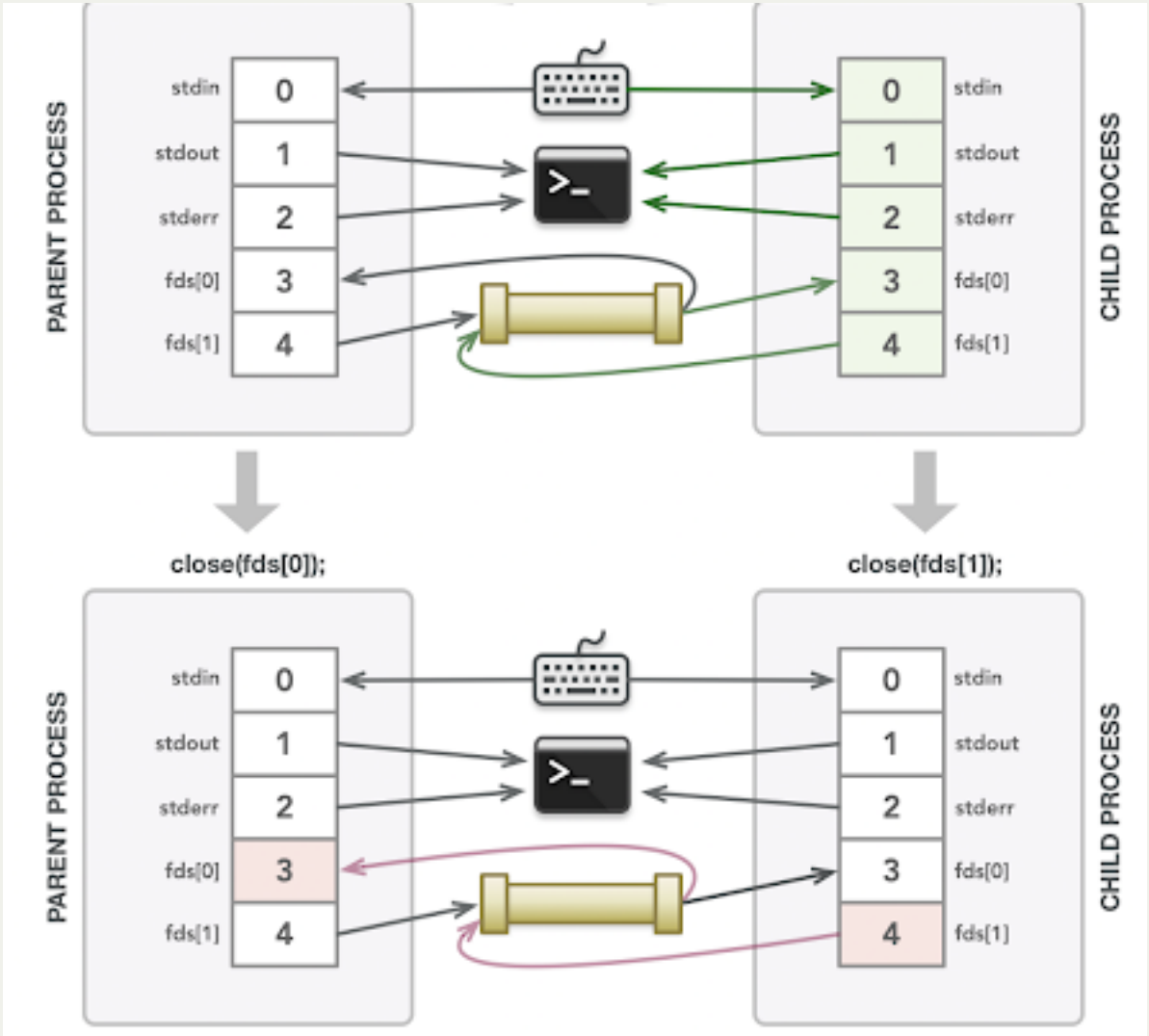
Key Idea: the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

We should close FDs when we are done with them. The **child** closes them here.

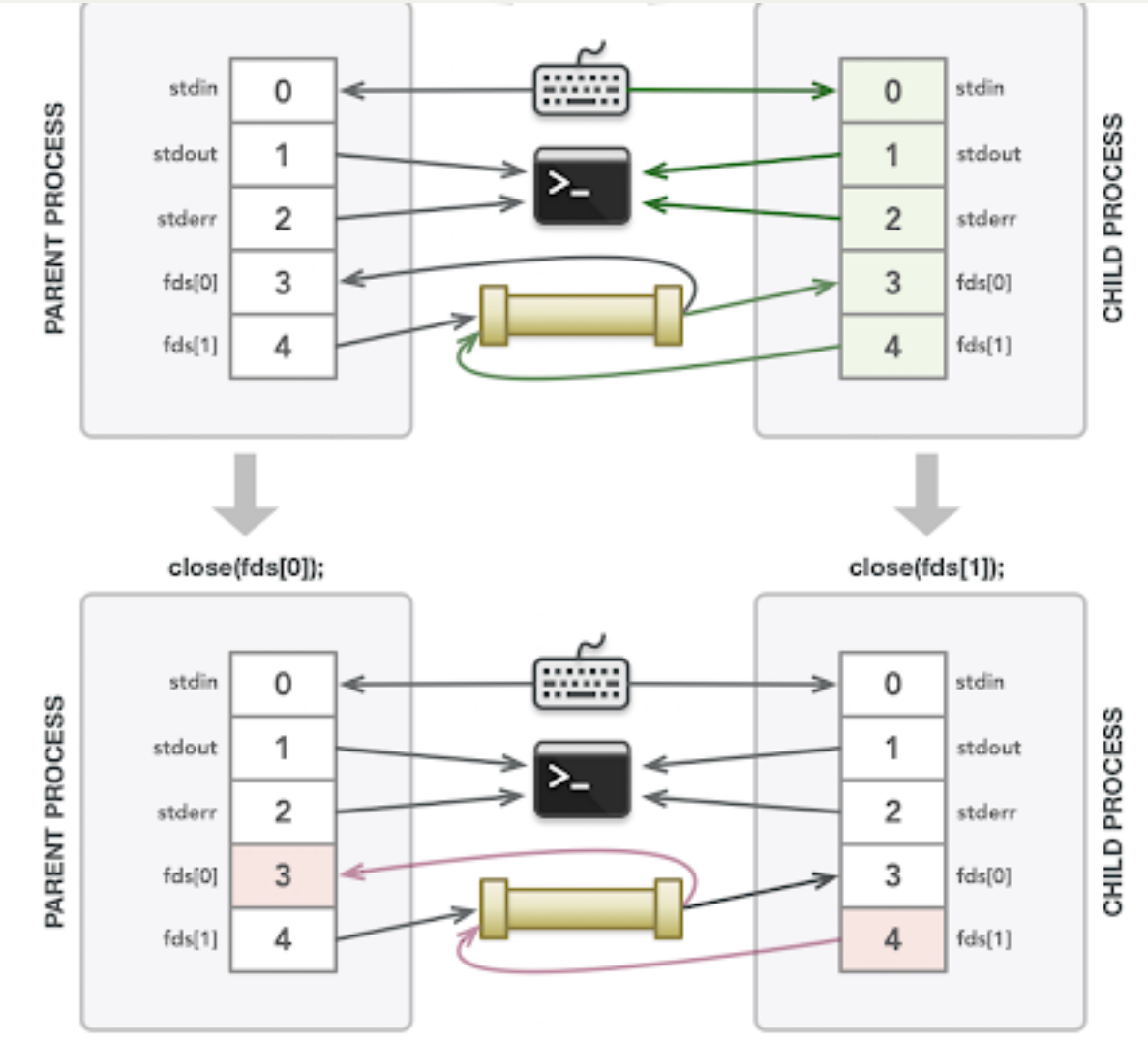
parent-child-pipe.c



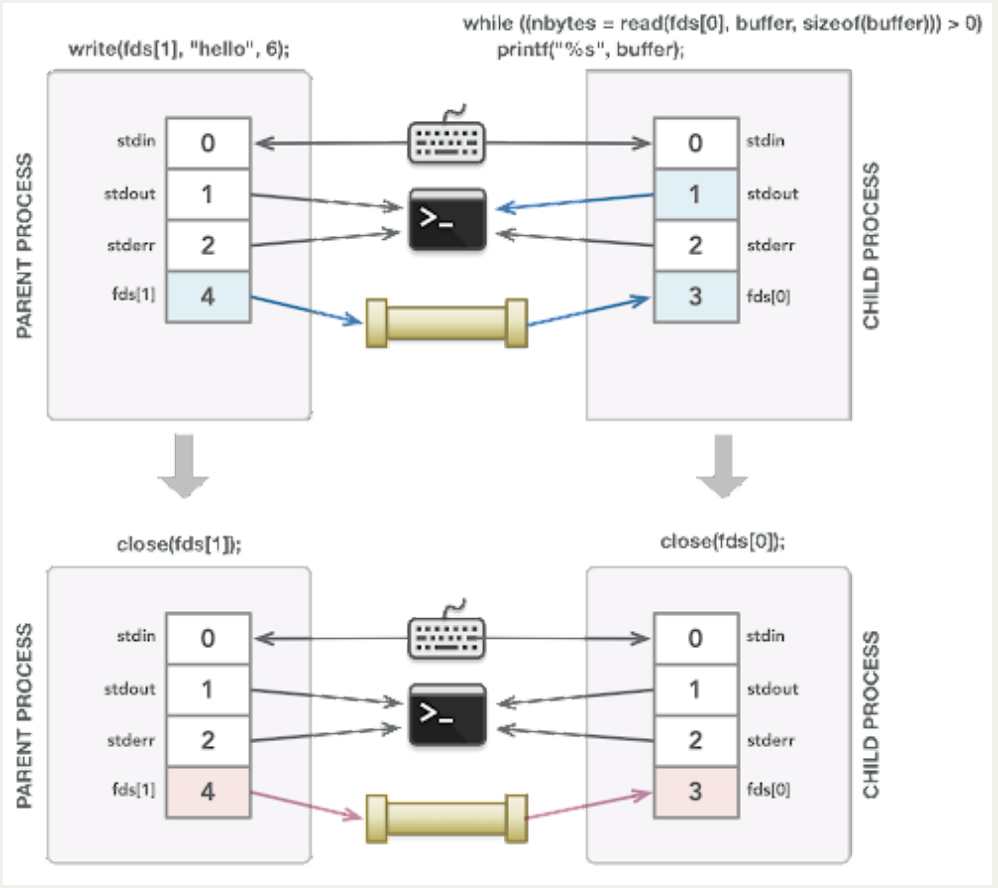
continued...



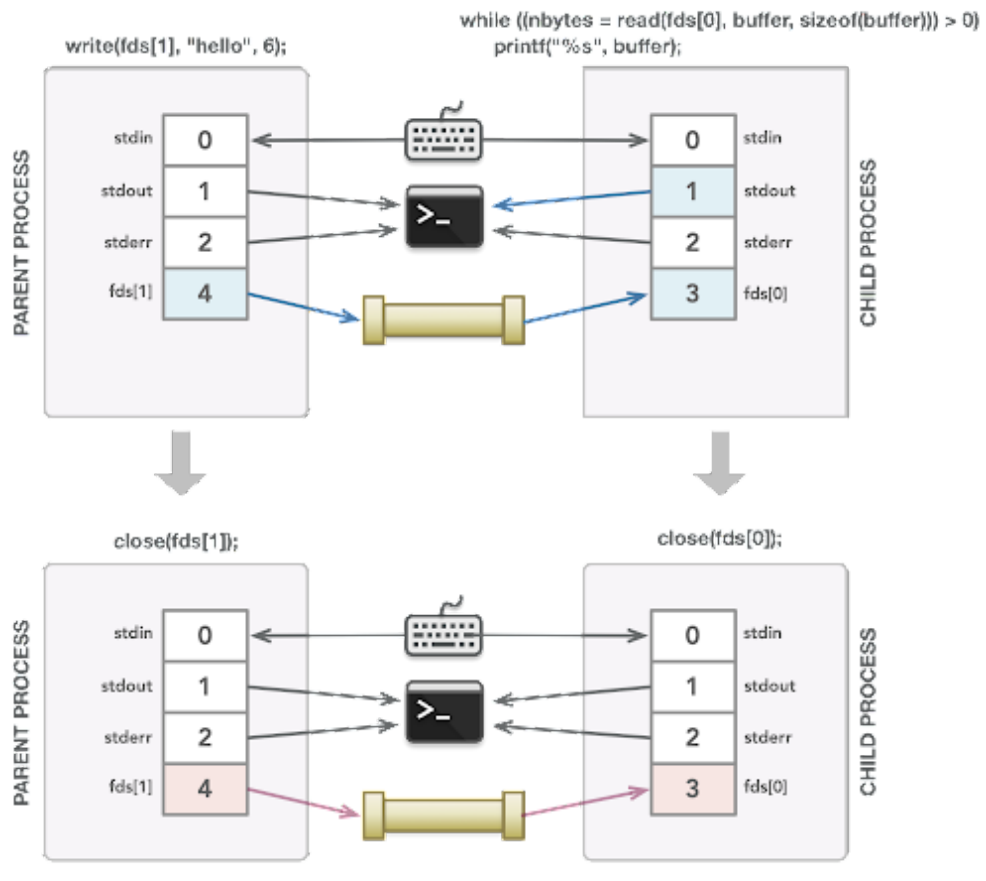
Illustrations courtesy of Roz Cyrus.



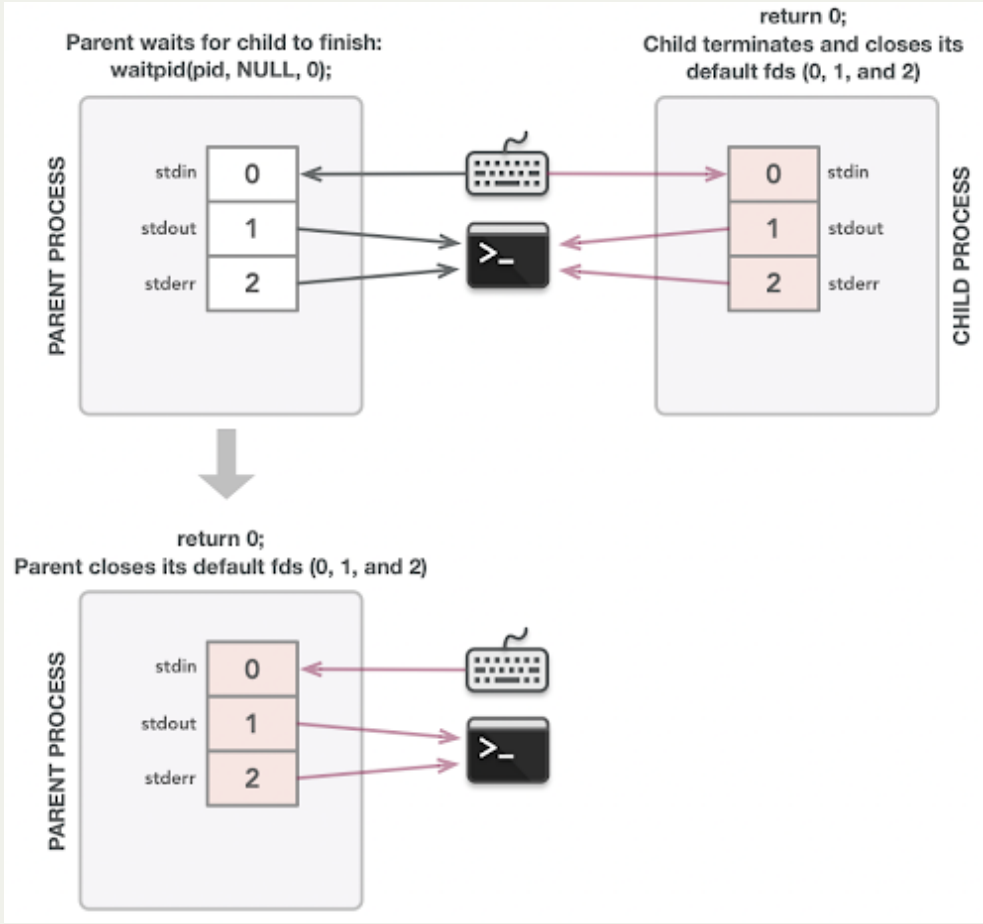
continued...



Illustrations courtesy of Roz Cyrus.



continued...



Illustrations courtesy of Roz Cyrus.

DEMO: Parent-Child Communication

https://cplayground.com/?p=eagle-fish-mouse

Pipes

This method of communication between processes relies on the fact that file descriptors are duplicated when forking.

- each process has its own copy of both file descriptors for the pipe
- both processes could read or write to the pipe if they wanted.
- each process must therefore close both file descriptors for the pipe when finished

This is the core idea behind how a shell can support piping between processes (e.g. **cat file.txt | uniq | sort**). Let's see how this works in a shell.

Lecture Recap

- Review: our first shell
- Running in the background
- Introducing Pipes
  - What are pipes?
  - Pipes between processes

Next time: more practice with pipes