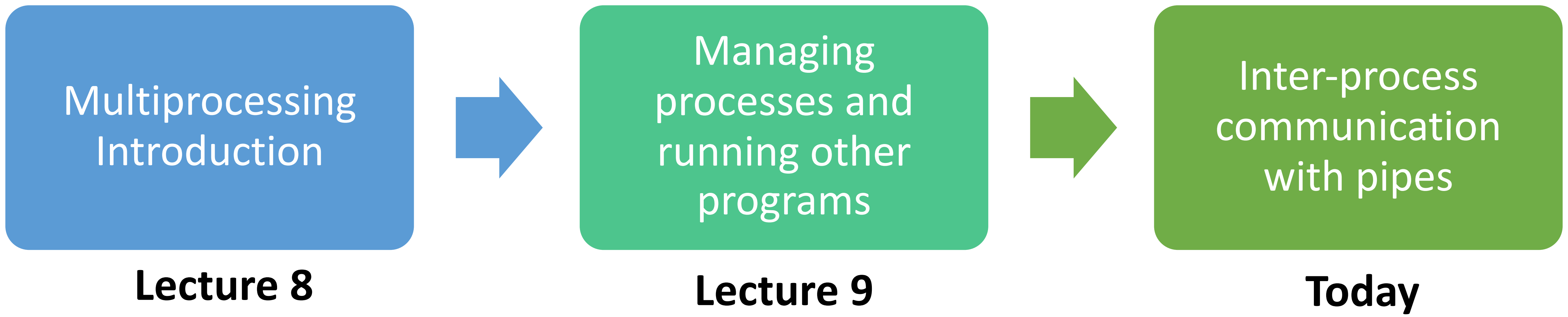
CS111, Lecture 10

Pipes

😷 masks required

Topic 2: Multiprocessing - How can our program create and interact with other programs? How does the operating system manage user programs?

CS111 Topic 2: Multiprocessing


Multiprocessing Introduction
Lecture 8
Managing processes and running other programs
Lecture 9
Inter-process communication with pipes
Today

assign3: implement your own shell!

Learning Goals

- Get more practice with using **fork()** and **execvp**
- Learn about **pipe** to create and manipulate file descriptors

Plan For Today

- **Recap**: **waitpid** and **execvp**
- **Demo:** our first shell
- Pipes

```
cp -r /afs/ir/class/cs111/lecture-code/lect10 .
```

Plan For Today

- **Recap**: **waitpid** and **execvp**
- **Demo:** our first shell
- Pipes

```
cp -r /afs/ir/class/cs111/lecture-code/lect10 .
```

waitpid()

A system call 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

- Returns 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 given <u>path</u>, *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 **<u>argv</u>** parameter.

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

execvp has many variants (see **man execvp**) but we'll just be using **execvp**.

execvp()

```c
// execvp-demo.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    char *args[] = {"/bin/ls", "-l", "/usr/class/cs111/lecture-code",
                    NULL};
    execvp(args[0], args);
    printf("This only prints if an error occurred.\n");
    return 0;
}
```

```
$ ./execvp-demo
Hello, world!
total 4
drwx------ 2 troccoli operator 2048 Oct  9 16:21 lect5
drwx------ 2 troccoli operator 2048 Oct 13 22:19 lect9
```

Implementing a Shell

How is execvp useful?

- This is the way that we can run other programs
- However, we often don't want to cannibalize the current process
- Instead: we will usually fork off a child process and call execvp there. The child process will be consumed, but that's ok

Implementing a Shell

A shell is essentially a program that repeats asking the user for a command and running that command

How do we run a command entered by the user?

1. Call **fork** to create a child process
2. In the child, call **execvp** with the command to execute
3. In the parent, wait for the child with **waitpid**

For assign3, you'll use this pattern to build your own shell, stsh ("Stanford shell") with various functionality of real Unix shells.

Demo: first-shell-soln.cc

Plan For Today

- **Recap**: **waitpid** and **execvp**
- **Demo:** our first shell
- **Pipes**

```
cp -r /afs/ir/class/cs111/lecture-code/lect10 .
```

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

(why is this useful? Shell piping and I/O redirection)

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.

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

pipe()

```c
static const char * kPipeMessage = "this message is coming via a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);

    // Write message to pipe (assuming all bytes written immediately)
    write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
    close(fds[1]);

    // Read message from pipe
    char receivedMessage[strlen(kPipeMessage) + 1];
    read(fds[0], receivedMessage, sizeof(receivedMessage));
    close(fds[0]);
    printf("Message read: %s\n", receivedMessage);

    return 0;
}
```

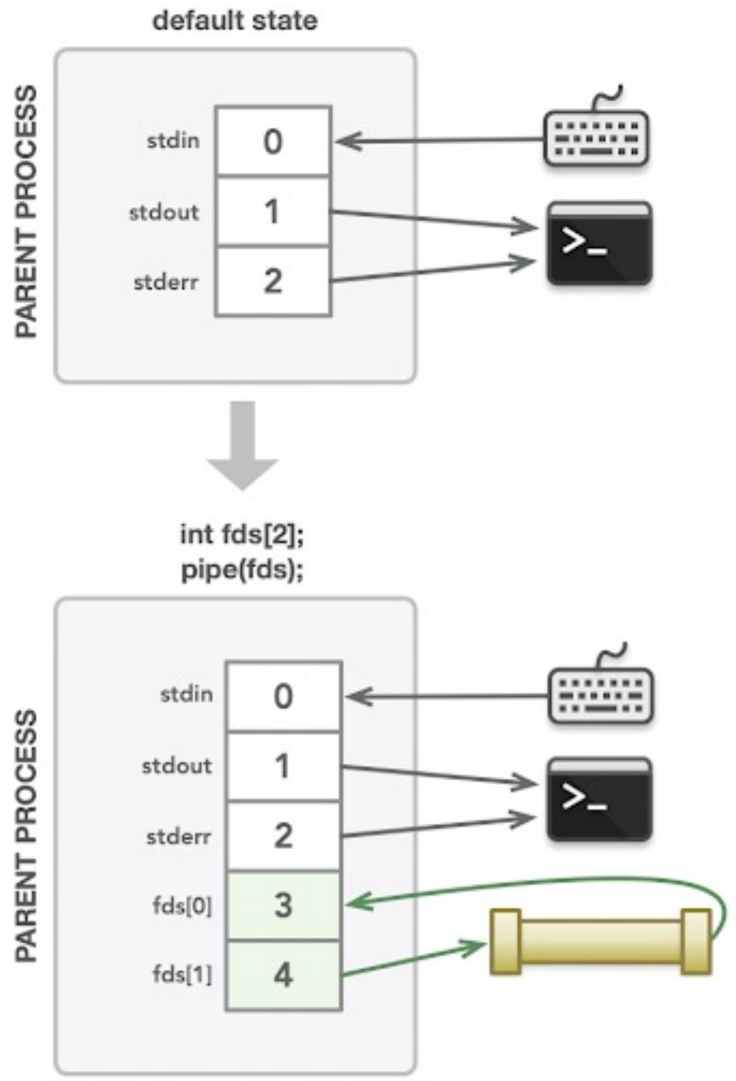
```
$ ./pipe-demo
Message read: this message is coming via a pipe.
```

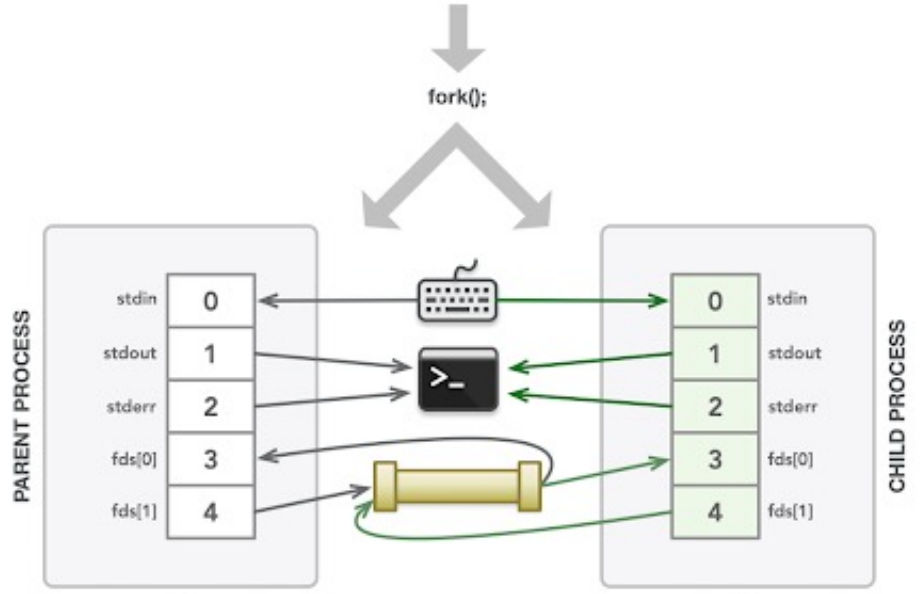
pipe()

pipe can allow processes to communicate! (how?)

- When fork is called, everything is cloned – *even* the file descriptors, which are **replicated in the child process**. This means if the parent creates a pipe and then calls fork(), both processes can use the pipe!
- 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)
- **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). So if one process is reading, it will wait until the other writes.

Interprocess Communication

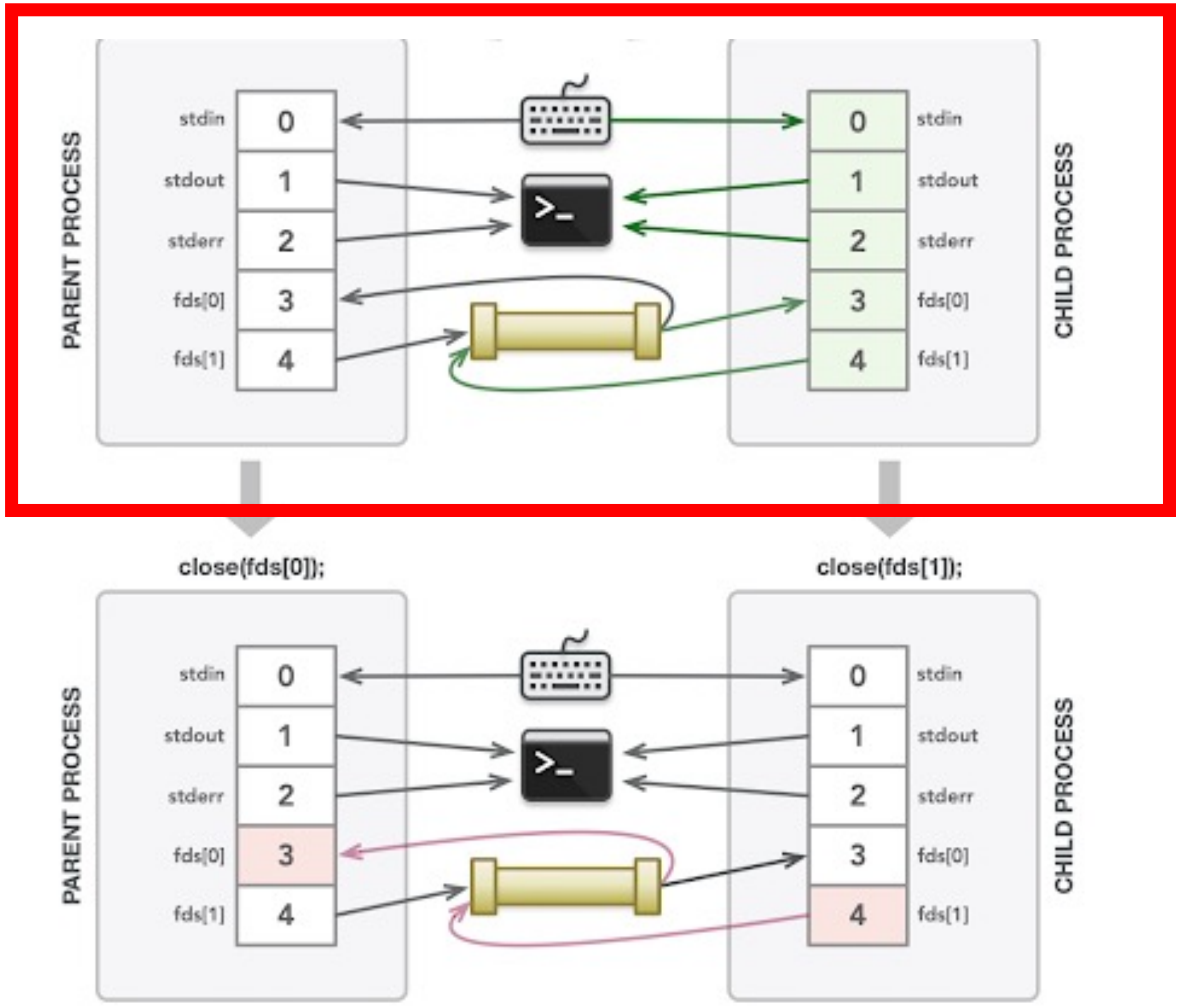




Illustrations courtesy of Roz Cyrus.

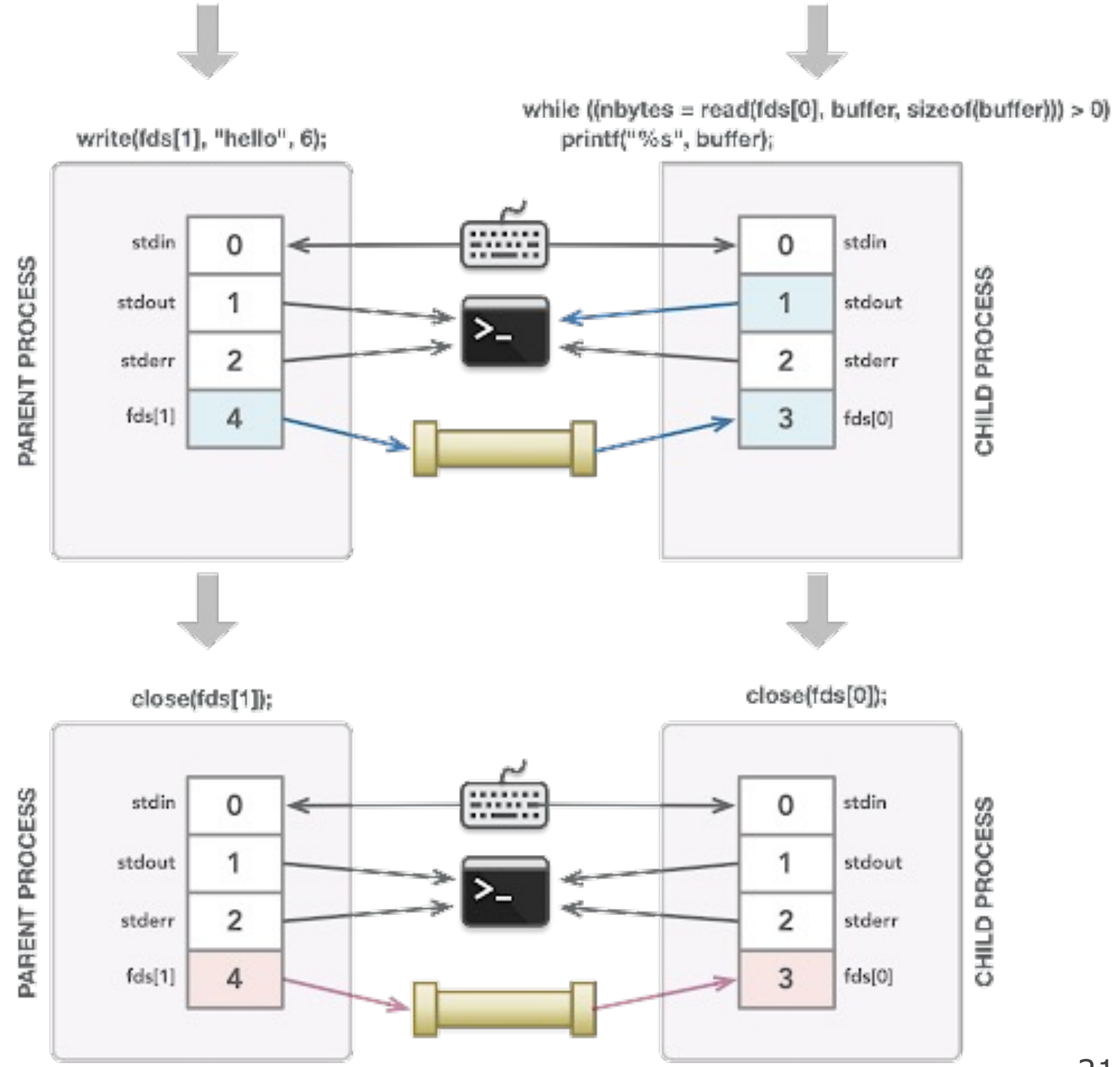
Key Idea: because the pipe file descriptors are duplicated in the child, we need to close the 2 pipe ends ***in both the parent and the child.***

Interprocess Communication



```
close(fds[0]);
```

```
close(fds[1]);
```



```
write(fds[1], "hello", 6);
```

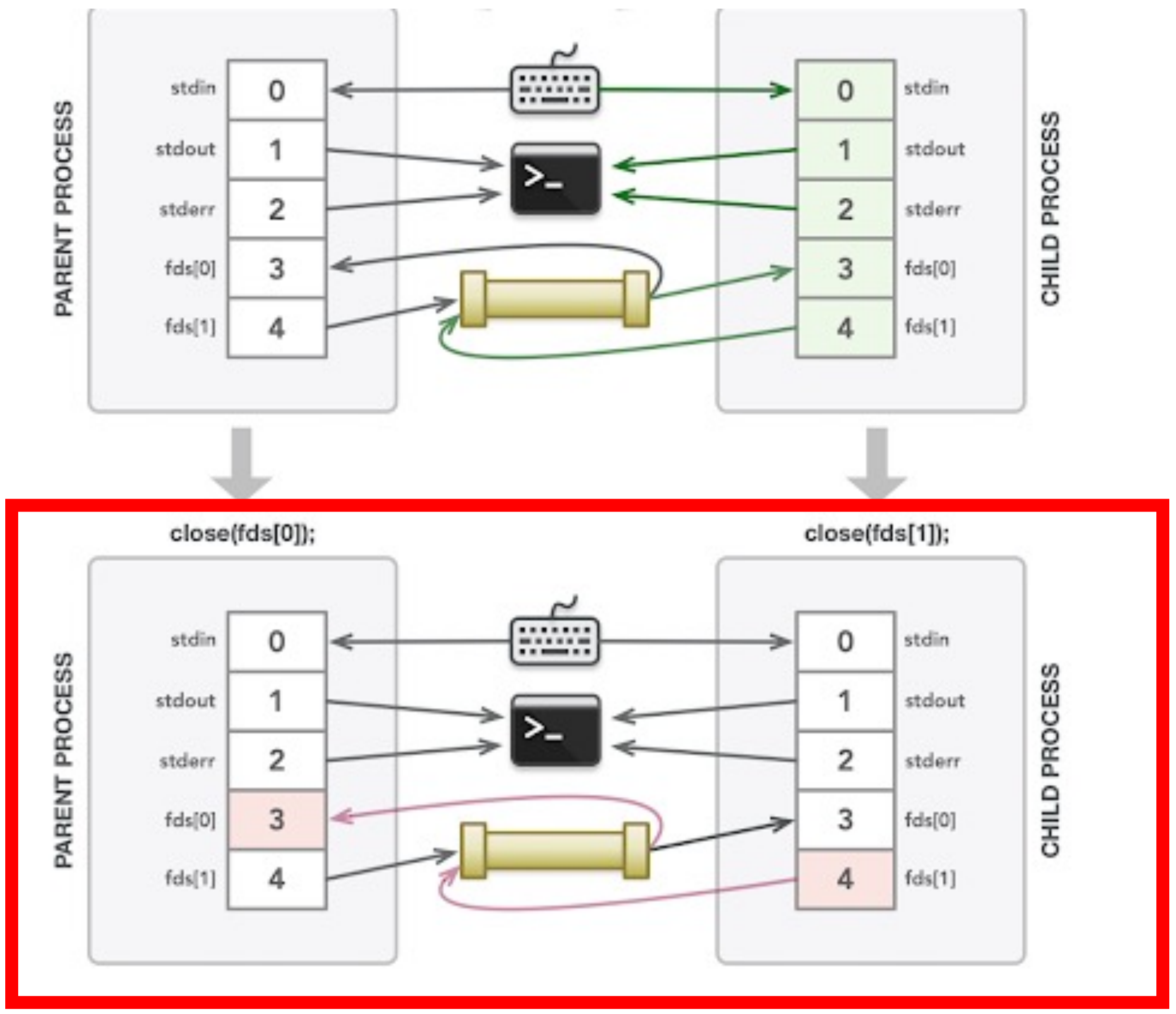
```
while ((nbytes = read(fds[0], buffer, sizeof(buffer))) > 0)
    printf("%s", buffer);
```

```
close(fds[1]);
```

```
close(fds[0]);
```

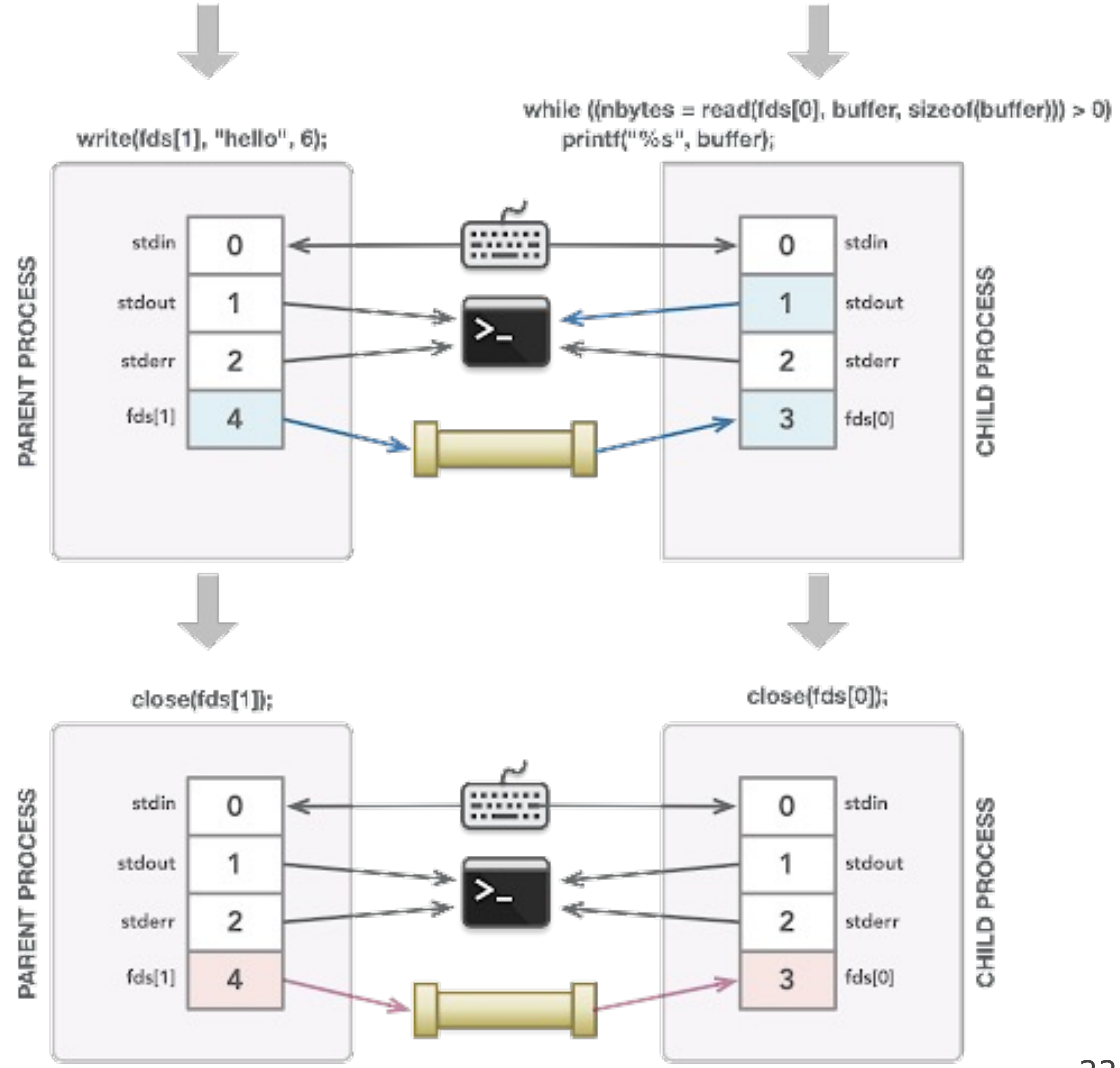
Illustrations courtesy of Roz Cyrus.

Interprocess Communication



```
close(fds[0]);
```

```
close(fds[1]);
```



```
write(fds[1], "hello", 6);
```

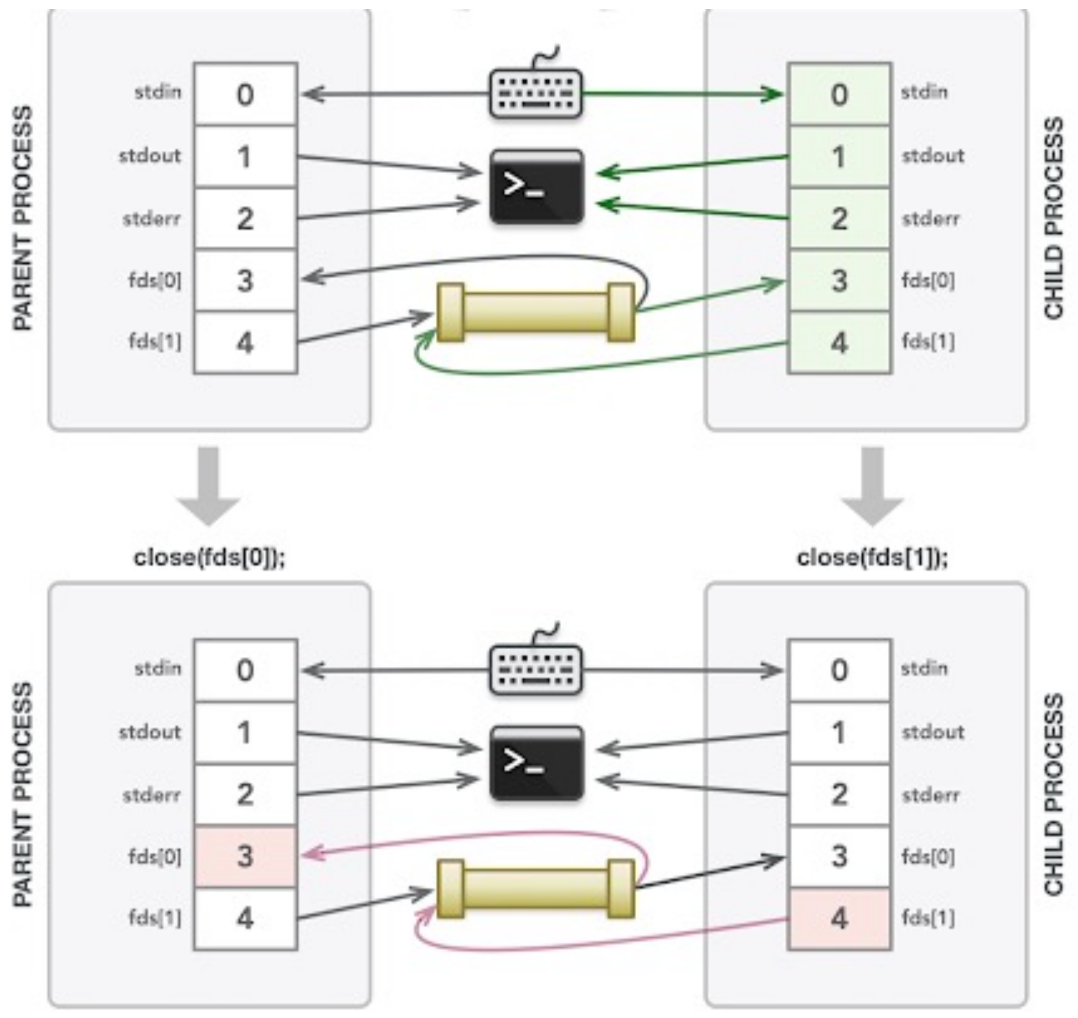
```
while ((nbytes = read(fds[0], buffer, sizeof(buffer))) > 0)
    printf("%s", buffer);
```

```
close(fds[1]);
```

```
close(fds[0]);
```

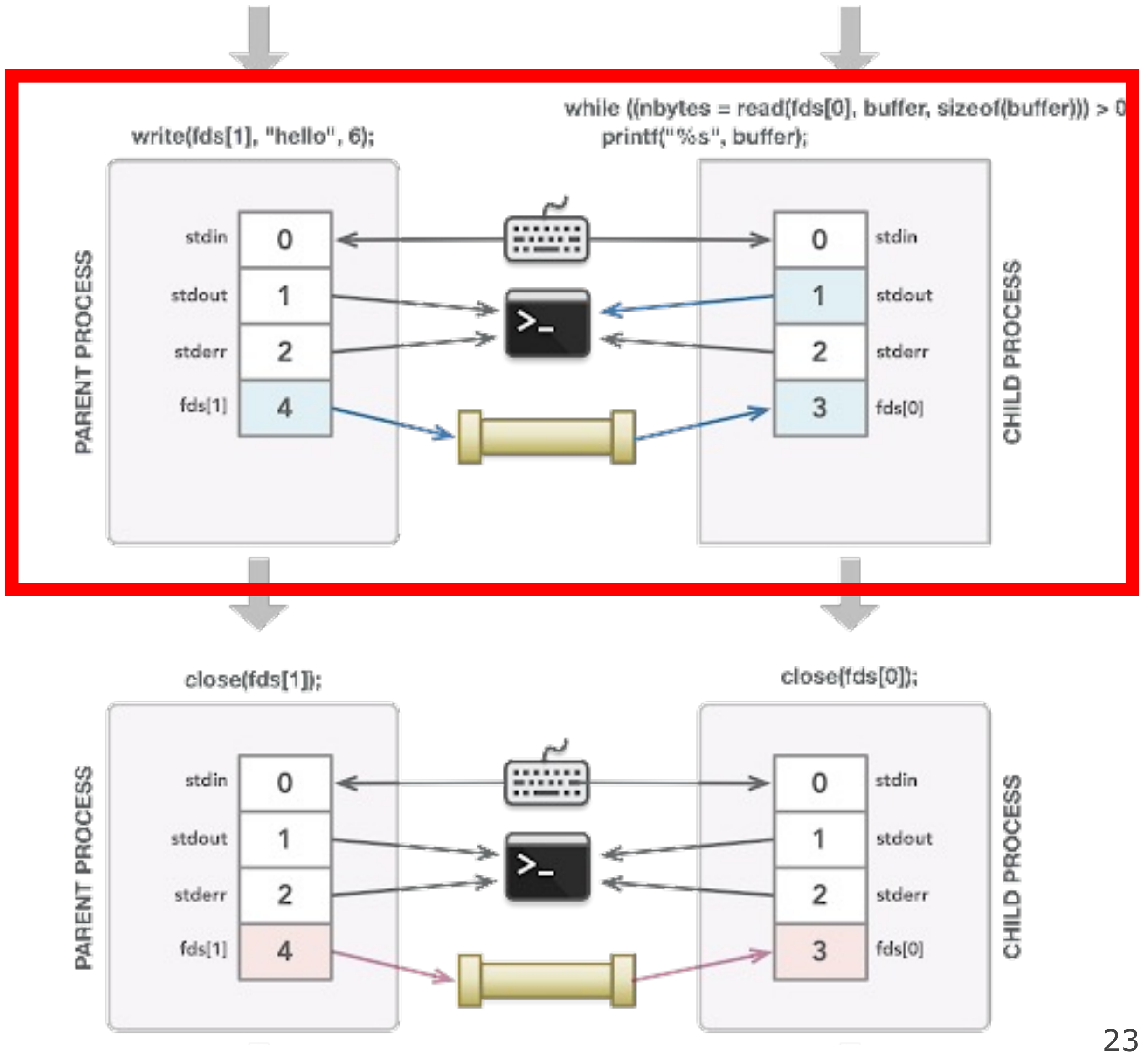
Illustrations courtesy of Roz Cyrus.

Interprocess Communication



```
close(fds[0]);
```

```
close(fds[1]);
```



```
write(fds[1], "hello", 6);
```

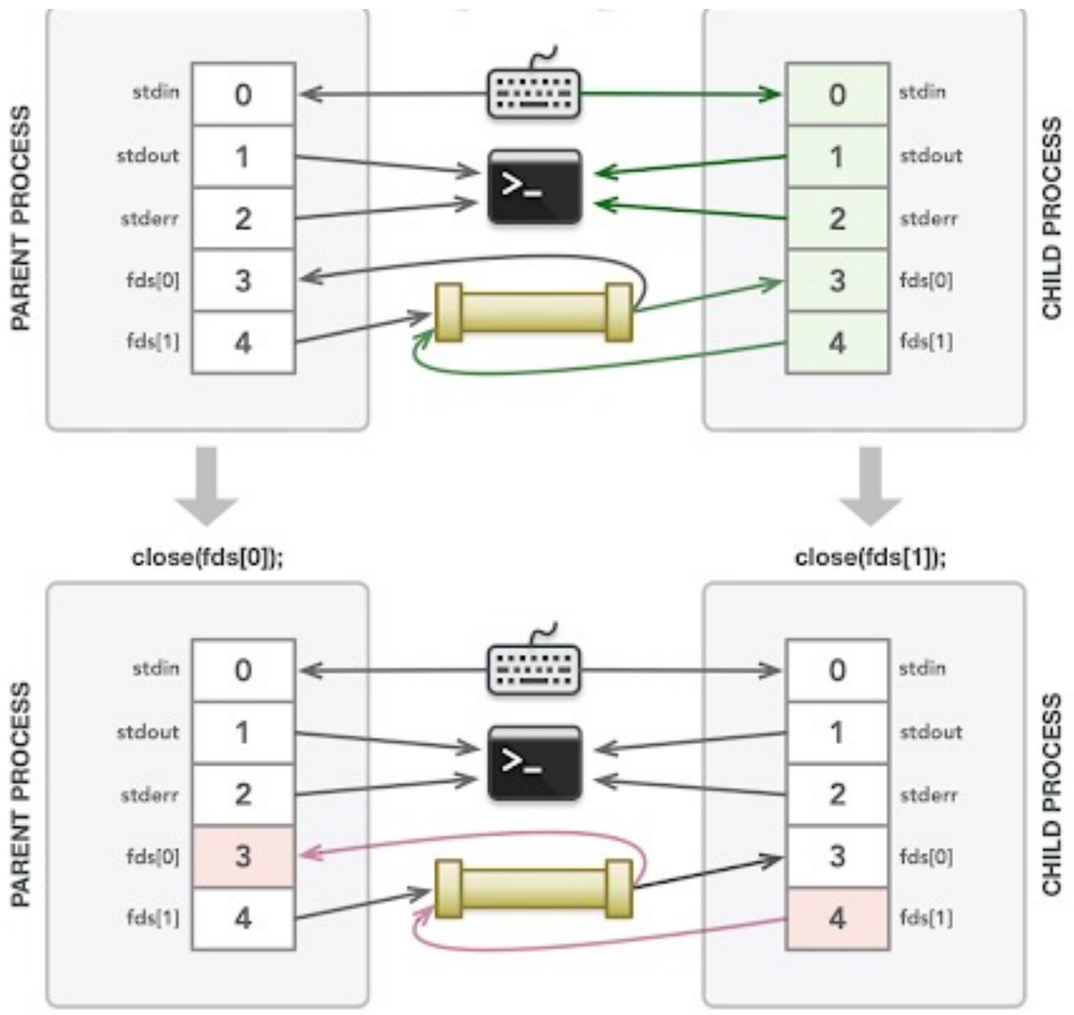
```
while ((nbytes = read(fds[0], buffer, sizeof(buffer))) > 0
    printf("%s", buffer);
```

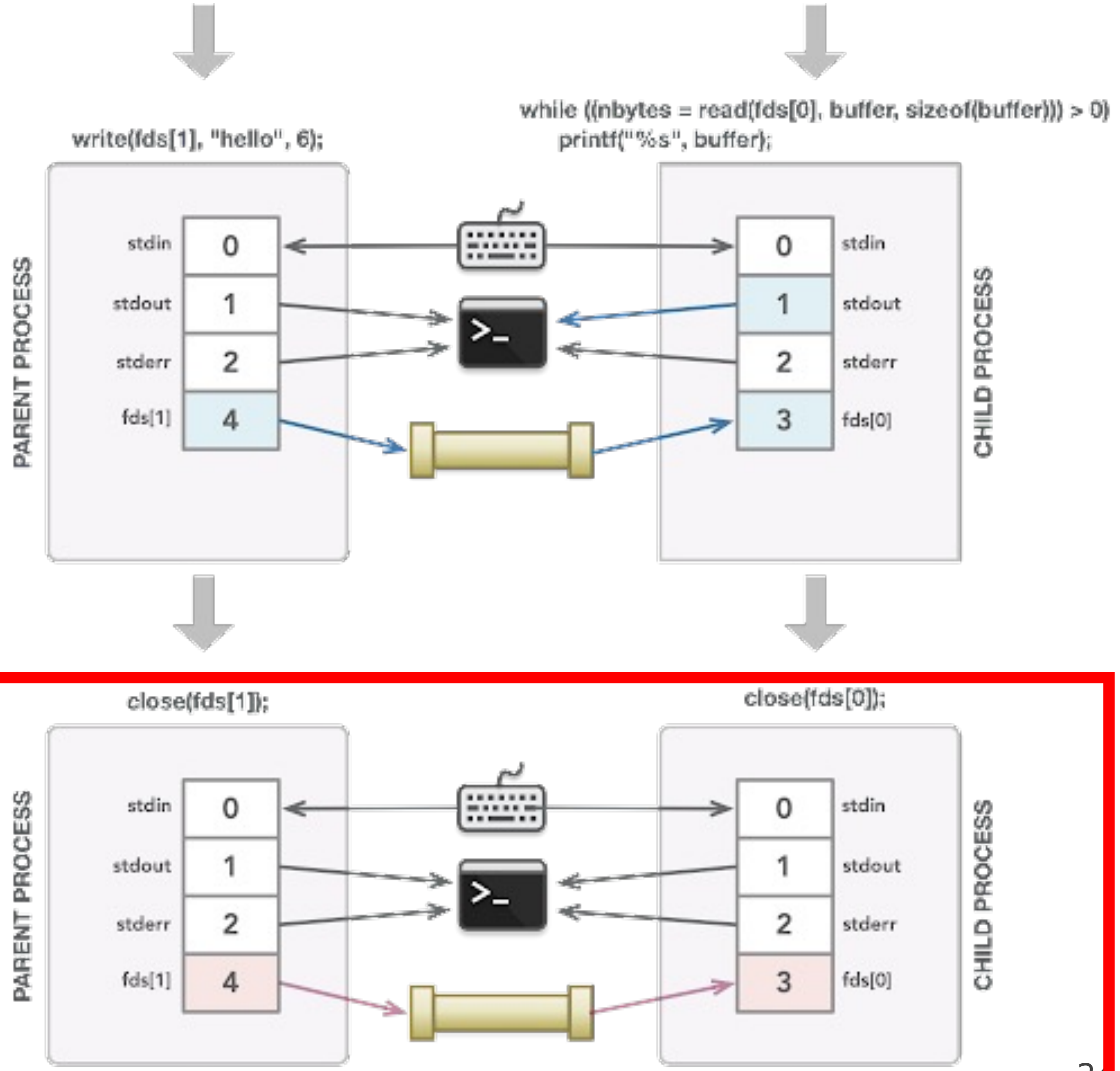
```
close(fds[1]);
```

```
close(fds[0]);
```

Illustrations courtesy of Roz Cyrus.

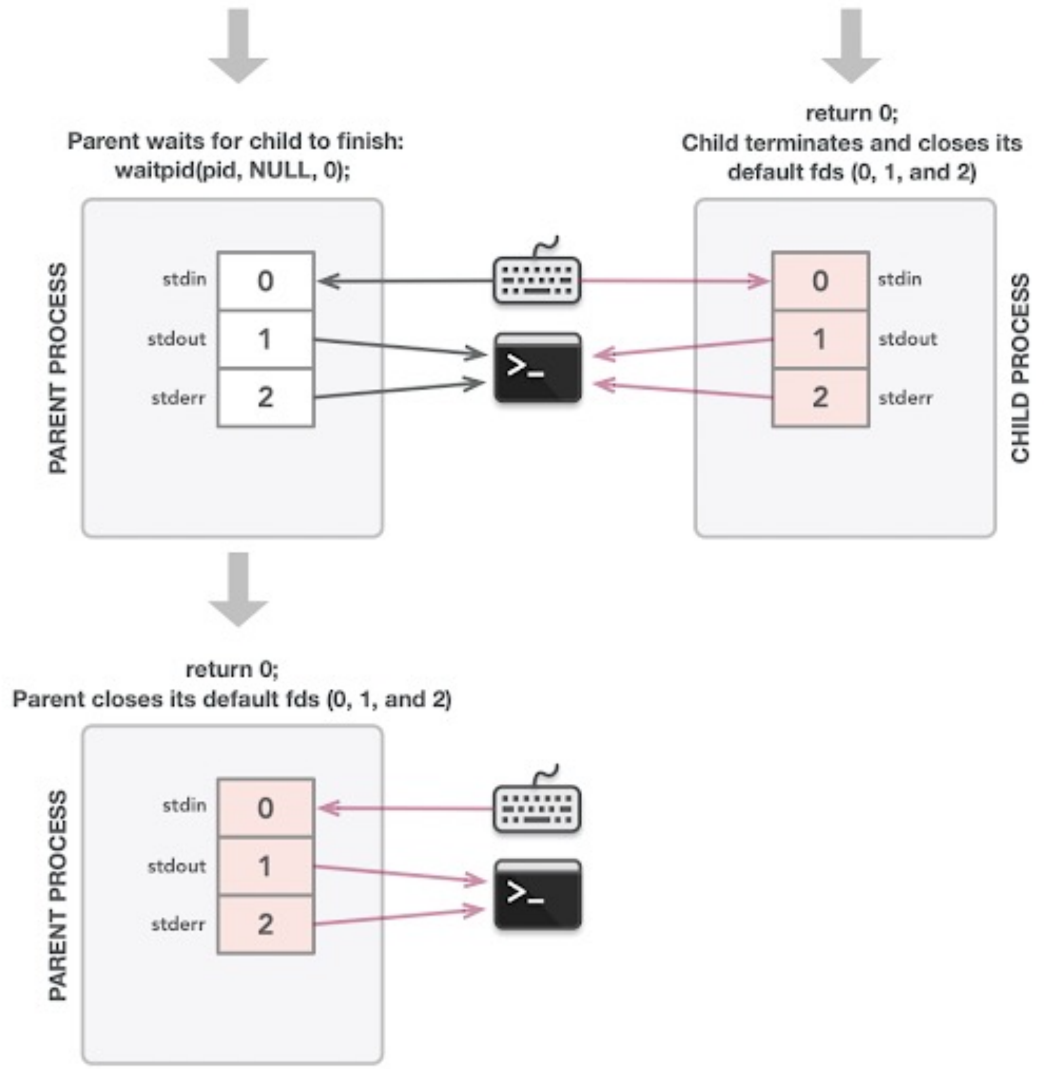
Interprocess Communication





Illustrations courtesy of Roz Cyrus.

Interprocess Communication



Parent waits for child to finish:
waitpid(pid, NULL, 0);
return 0;
Child terminates and closes its default fds (0, 1, and 2)
PARENT PROCESS
stdin
stdout
stderr
0
1
2
CHILD PROCESS
return 0;
Parent closes its default fds (0, 1, and 2)

Illustrations courtesy of Roz Cyrus.

pipe()

```
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;
}
```

pipe()

```
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;
}
```

pipe()

```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;
}
```

pipe()

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**).

Recap

- **Recap**: **waitpid** and **execvp**
- **Demo:** our first shell
- Pipes

Lecture 10 takeaway: shells work by spawning child processes that call execvp. Pipes are sets of file descriptors that let us read/write. We can share pipes with child processes to send arbitrary data back and forth.

```
cp -r /afs/ir/class/cs111/lecture-code/lect10 .
```