CS111, Lecture 10
Pipes

CS111 Topic 2: Multiprocessing

Key Question: *How can our program create and interact with other programs? How does the operating system manage user programs?*

Lecture 8 **Lecture 9** **Today / Lecture 11**

assign3: implement your own shell!

Learning Goals

- Learn about **pipe** and how we can create a communication channel between processes
- Understand how file descriptors are duplicated across processes
- Learn the steps to implement pipelines in our shell

Plan For Today

- **Recap: fork, waitpid, execvp** and our first shell
- **Shell Feature Demo:** pipes
- **pipe()** system call
- **Example**: Parent-child pipe
- **dup2()** and rewiring file descriptors

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

Plan For Today

- **Recap: fork, waitpid, execvp and our first shell**
- **Shell Feature Demo:** pipes
- **pipe()** system call
- **Example**: Parent-child pipe
- **dup2()** and rewiring file descriptors

```
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 for any one child
- **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
- also cleans up the state of the child that was waited on

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 path, *completely cannibalizing the current process*.

- If successful, **execvp never returns** in the calling process
- If unsuccessful, **execvp** returns -1
- **argv** is the NULL-terminated arguments array to pass to the new program's **main** function.
- **path** should generally be **argv[0]** for our purposes (since program name is first argv argument)

Key idea: a parent can still wait on a child that calls 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

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

First Shell Solution

```
void runPipeline(const pipeline& p) {
        command cmd = p.commands[0];  // get tokenized version of command
        // Step 1: fork off a child process to run the command
        pid_t pidOrZero = fork();
        if (pidOrZero == 0) {
                // Step 2: if we are the child, execute the command
                execvp(cmd.argv[0], cmd.argv);
                // If the child gets here, there was an error
                throw STSHException(string(cmd.argv[0]) + ": Command not found.");
        }
        // Step 3: if we are the parent, wait for the child
        waitpid(pidOrZero, NULL, 0);
}
```

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**

Terminating the Child Process

```
void runPipeline(const pipeline& p) {
        command cmd = p.commands[0];
        // Step 1: fork off a child process to run the command
        pid_t pidOrZero = fork();
        if (pidOrZero == 0) {
                // Step 2: if we are the child, execute the command
                execvp(cmd.argv[0], cmd.argv);
                // If the child gets here, there was an error
                throw STSHException(string(cmd.argv[0]) + ": Command not found.");
        }
        // Step 3: if we are the parent, wait for the child
        waitpid(pidOrZero, NULL, 0);
}
```

assign3 starter terminates any child process that throws an exception. If we omit this line and execvp fails, the child will continue executing – calling **waitpid**, returning back to **main**, itself then also running the prompting code intended only for the parent!

Plan For Today

- Recap: fork, waitpid, execvp and our first shell
- **Shell Feature Demo:** pipes
- **pipe()** system call
- **Example**: Parent-child pipe
- **dup2()** and rewiring file descriptors

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

Additional Shell Features

There are many more features from full shells that our shell could support:

- Running commands in the background (put ”&” after command)
- Ctl-c to terminate a program
- Chaining multiple commands together (a “pipeline”)
- Saving a command’s output to a file, or reading a command’s input from a file

Additional Shell Features

There are many more features from full shells that our shell could support:

- Running commands in the background (put "&" after command)
- Ctl-c to terminate a program
- **Chaining multiple commands together (a "pipeline")**
- Saving a command's output to a file, or reading a command's input from a file **(next time)**

You'll get to fully implement both features on assign3!

Demo: shell pipelines

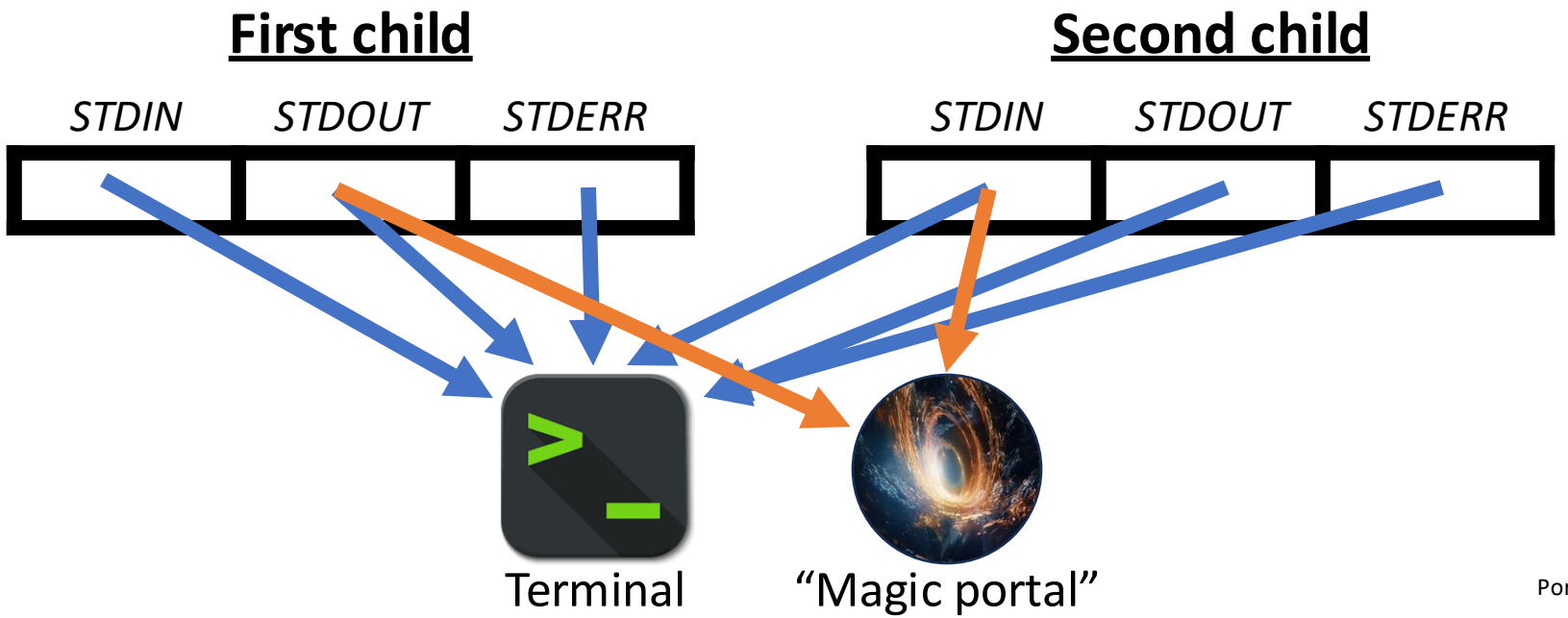
Key Unix idea: chaining the output (STDOUT) of one command to be the input (STDIN) of another.

Each command doesn't need to know it's part of a pipeline!

How do we implement shell pipelines?

Let's focus on two-command pipelines for now. How can we implement this?

1. Create a "magic portal" that allows data to be sent between two processes
2. Spawn 2 child processes (1 per command)
3. Connect one end of that portal to the first child's STDOUT, and the other end to the second child's STDIN

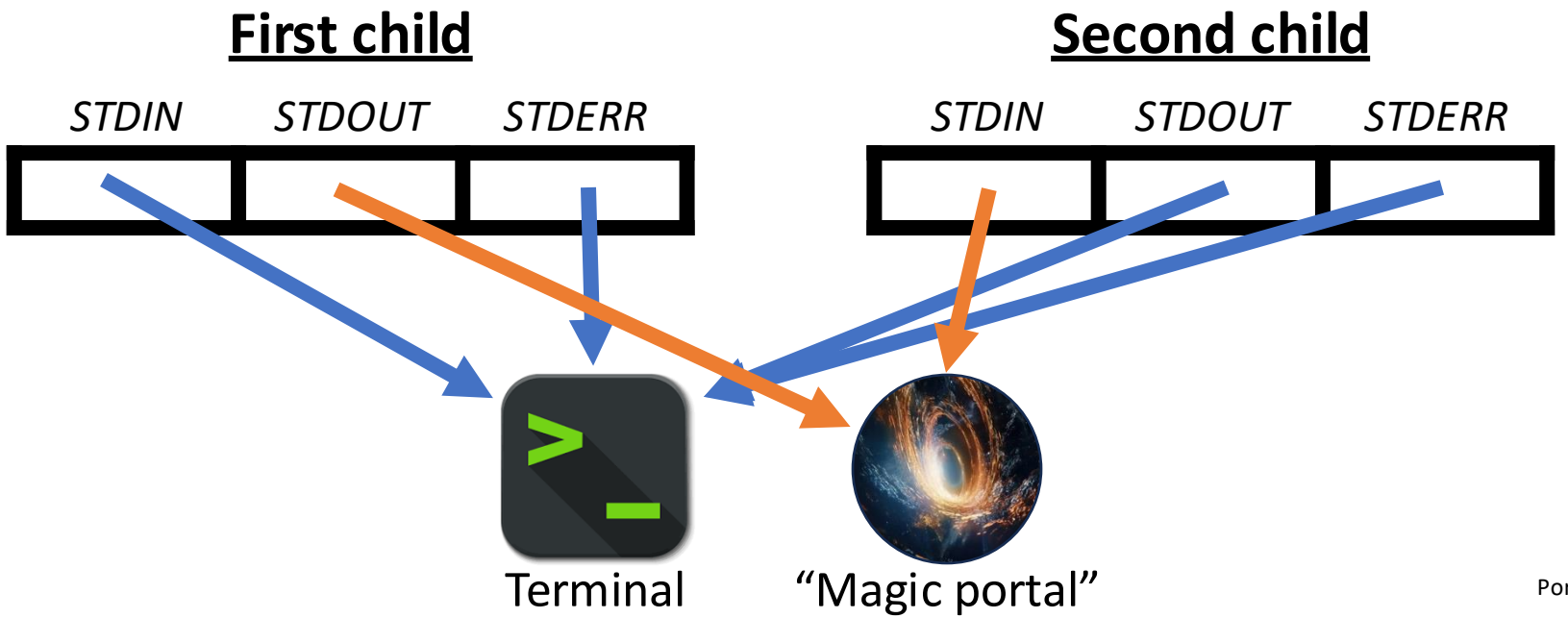


Portal image unmodified, CC credit

How do we implement shell pipelines?

Three key questions:

1. What the heck is a “magic portal” and how do we create one?
2. How do we share this “magic portal” between processes?
3. How do we connect a process’s STDIN/STDOUT to this ”magic portal”?



Portal image unmodified, CC credit

How do we implement shell pipelines?

Three key questions:

1. What the heck is a “magic portal” and how do we create one?

   The **pipe()** system call

2. How do we share this “magic portal” between processes?

   Relying on cloning that happens on **fork()** (plus a new property of execvp we’ll see later)

3. How do we connect a process’s STDIN/STDOUT to this ”magic portal”?

   The **dup2()** system call

How do we implement shell pipelines?

Three key questions:

1. **What the heck is a "magic portal" and how do we create one?**
   The pipe() system call
2. How do we share this "magic portal" between processes?
   Relying on cloning that happens on **fork()** (plus a new property of **execvp** we'll see later)
3. How do we connect a process's STDIN/STDOUT to this "magic portal"?
   The **dup2()** system call

Plan For Today

- Recap: **fork, waitpid, execvp** and our first shell
- **Shell Feature Demo:** pipes
- **pipe() system call**
- **Example**: Parent-child pipe
- **dup2()** and rewiring file descriptors

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

"Magic Portal": pipe() System Call

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

The **pipe** system call gives us back two file descriptors, where everything written to one can be read from the other.

- Specifically: populates the 2-element array **fds** with the two file descriptors. Everything *written* to fds[1] can be *read* from fds[0]. ***Tip**: you learn to read before you learn to write (read = fds[0], write = fds[1]).*
- Returns 0 on success, or -1 on error.

Imagine: like opening the same file twice, once for reading and once for writing.

Why doesn't it give back 1 read/write file descriptor? Can be at different places reading vs. writing.

pipe() Within 1 Process

```
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 (assume all bytes read immediately)
    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.
```

How do we implement shell pipelines?

Three key questions:

1. What the heck is a “magic portal” and how do we create one?

   The pipe() system call

2. **How do we share this “magic portal” between processes?**

   Relying on cloning that happens on fork() (plus a new property of execvp we’ll see later)

3. How do we connect a process’s STDIN/STDOUT to this ”magic portal”?

   The **dup2()** system call

pipe() and fork()

Key idea: a pipe can facilitate parent-child communication because file descriptors are duplicated on **`fork()`**. Thus, a pipe created prior to **`fork()`** will also be accessible in the child!

But wait – isn't the child a *copy* of the parent? So wouldn't it get a *copy* of the pipe, not share the same one?

No, as it turns out – the child *also gets access to the same* file descriptor sessions open at the time **fork** is called (sort of - we'll see how this is possible later!)

pipe()

With this sharing, **pipe** can allow processes to communicate!

1. Create a pipe in the parent
2. Fork off a child process – important this comes *second*; only a pipe created prior to the fork is accessible by the child! Otherwise, each has their own separate pipe.
3. Parent and child can read/write with pipe (e.g. parent writes, child reads)

Plan For Today

- Recap: **fork, waitpid, execvp** and our first shell
- **Shell Feature Demo:** pipes
- **pipe()** system call
- **Example: Parent-child pipe**
- **dup2()** and rewiring file descriptors

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

Demo: Parent Child Pipe

Let's write a program where the parent sends a predetermined message to the child, which prints it out.

Note: can use a pipe in any way we'd like, e.g. child could write and parent could read. Though usually pipes are used *unidirectionally* – e.g. if you want bidirectional communication, we usually create 2 pipes.

Demo: Parent Child 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) { // Child only reads from pipe (assume everything is read)
        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 Summary

- Both the parent *and* the child must close the pipe FDs when they are done with them.
- If someone tries calling **read** from a pipe and no data has been written, it will block until some data is available, or until the pipe write end is closed everywhere (“end of file” for pipes).

How do we implement shell pipelines?

Three key questions:

1. What the heck is a “magic portal” and how do we create one?

   The pipe() system call

2. How do we share this “magic portal” between processes?

   Relying on cloning that happens on fork() (plus a new property of execvp we’ll see later)

3. **How do we connect a process’s STDIN/STDOUT to this ”magic portal”?**

   The dup2() system call

Plan For Today

- Recap: **fork, waitpid, execvp** and our first shell
- **Shell Feature Demo:** pipes
- **pipe()** system call
- **Example:** Parent-child pipe
- **dup2()** and rewiring file descriptors

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

Rewiring File Descriptors

The previous parent-child pipe program requires manually reading and writing from new file descriptors created for the pipe.

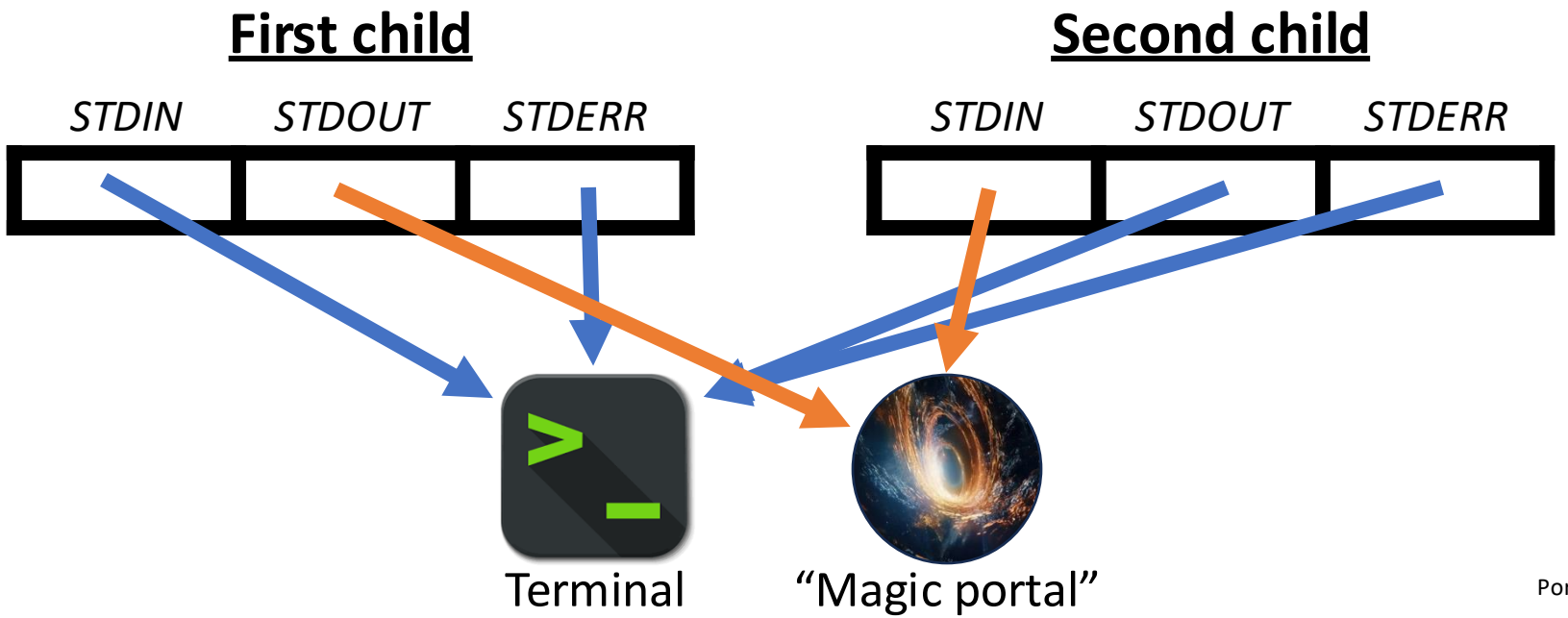
Ultimate goal: invisibly connect pipe to STDOUT/STDIN without the programs knowing.

We want anything the ”writer process” prints to the terminal to be automatically written to the pipe. And want whatever the “reader process” reads from its STDIN to actually be read from the pipe.

How do we implement shell pipelines?

To implement two-process pipelines, we must do the following:

1. Create a “magic portal” that allows data to be sent between two processes
2. Spawn 2 child processes (1 per command)
3. Connect one end of that portal to the first child’s STDOUT, and the other end to the second child’s STDIN.



Portal image unmodified, CC credit

Redirecting Process I/O

dup2 duplicates an open resource session from one file descriptor number (srcfd) to another (dstfd).

```
int dup2(int srcfd, int dstfd);
```

e.g. dup2(3, 1); // duplicates from 3 -> 1; FD 1 and 3 now refer to same open session

- Both will now refer to the same open resource session (e.g. if you read from one, it will advance the position of the other).
- If **dstfd** is already open, it is closed before being used.

Example: have STDOUT really write to a pipe, instead of the terminal!

```
dup2(fds[1], STDOUT_FILENO);
```

Redirecting Process I/O

dup2 duplicates an open resource session from one file descriptor number (srcfd) to another (dstfd).

```
int dup2(int srcfd, int dstfd);
```

Once we dup2, it's common to close the original FD – we usually don't need 2 both referring to the same session.

Example: have STDOUT really write to a pipe – after dup2, close original pipe FD.

```
dup2(fds[1], STDOUT_FILENO);
close(fds[1]);
```

How would we wire up a pipe to feed the STDOUT of process A to the STDIN of process B?

connect A's STDOUT to pipe read end, connect B's STDIN to pipe write end

connect A's STDIN to pipe write end, connect B's STDOUT to pipe read end

connect A's STDOUT to pipe write end, connect B's STDIN to pipe read end

connect A's STDIN to pipe read end, connect B's STDOUT to pipe write end

How would we wire up a pipe to feed the STDOUT of process A to the STDIN of process B?

connect A's STDOUT to pipe read end, connect B's STDIN to pipe write end

0%

connect A's STDIN to pipe write end, connect B's STDOUT to pipe read end

0%

connect A's STDOUT to pipe write end, connect B's STDIN to pipe read end

0%

connect A's STDIN to pipe read end, connect B's STDOUT to pipe write end

0%

How would we wire up a pipe to feed the STDOUT of process A to the STDIN of process B?

connect A's STDOUT to pipe read end, connect B's STDIN to pipe write end

0%

connect A's STDIN to pipe write end, connect B's STDOUT to pipe read end

0%

connect A's STDOUT to pipe write end, connect B's STDIN to pipe read end

0%

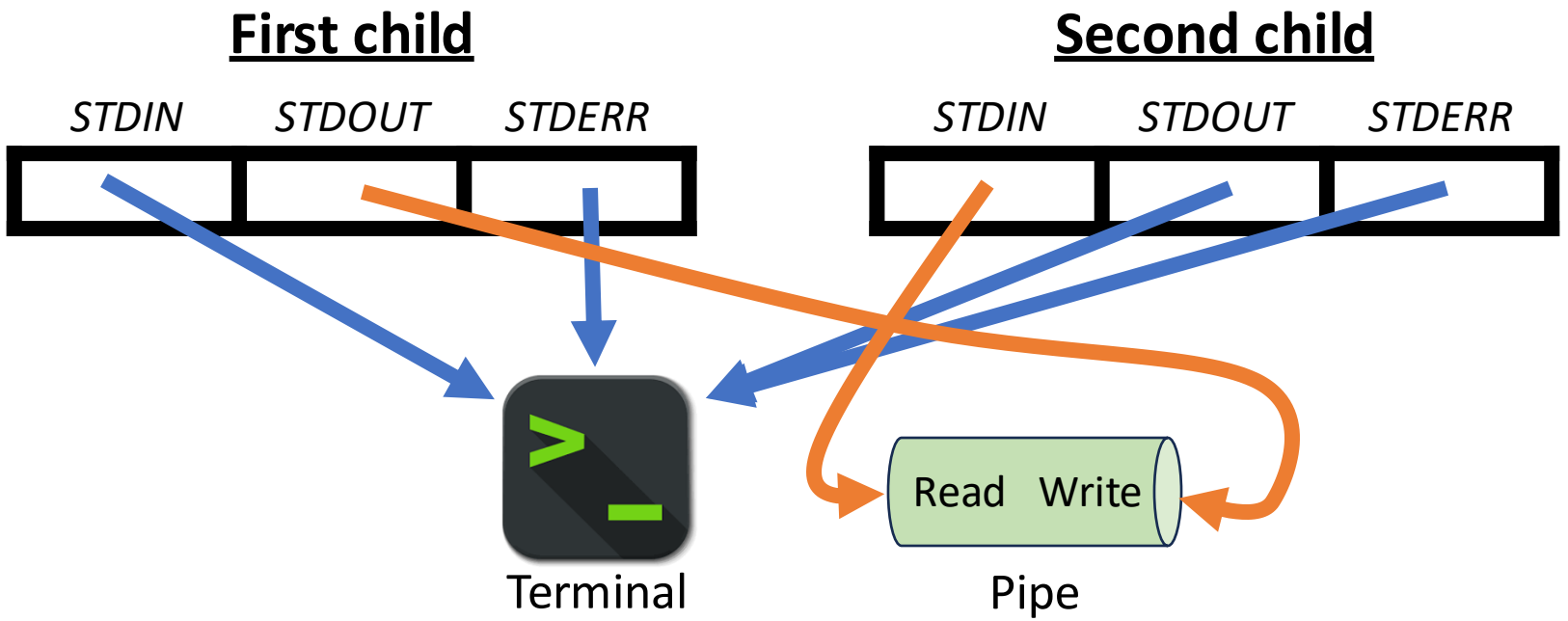
connect A's STDIN to pipe read end, connect B's STDOUT to pipe write end

0%

How do we implement shell pipelines?

To implement two-process pipelines, we must do the following:

1. Create a pipe *prior to spawning the child processes*
2. Spawn 2 child processes (1 per command)
3. Use **dup2** to connect the first child's STDOUT to the write end of the pipe. Use **dup2** again to connect the second child's STDIN to the read end of the pipe.



A Secret About execvp

Problem: if we spawn a child and rewire its STDOUT to point to a pipe, won't everything get wiped anyway when we call **execvp**?

New insight: execvp consumes the process but *leaves the file descriptor table intact!*

More next time…

Recap

- Recap: **fork, waitpid, execvp** and our first shell
- **Shell Feature Demo:** pipes
- **pipe()** system call
- **Example:** Parent-child pipe
- **dup2()** and rewiring file descriptors

Next time: more about rewiring STDIN/STDOUT, plus the details of how pipes are shared on fork()

Lecture 10 takeaway: 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 .
```