CS111, Lecture 11

Pipes, Continued

masks strongly recommended

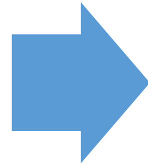
This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides and notes created by John Ousterhout, Jerry Cain, Chris Gregg, and others.
NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

CS111 Topic 2: Multiprocessing

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

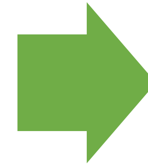
Multiprocessing Introduction

Lecture 8



Managing processes and running other programs

Lecture 9



Inter-process communication with pipes

Today / Lecture 11

assign3: implement your own shell!

Learning Goals

- Learn about **dup2** to rewire file descriptors
- See how to use **pipe** + **dup2** to implement pipelines
- Understand how to implement I/O redirection

Plan For Today

- **Recap**: Pipes so far
- Closing pipes
- **dup2()** and rewiring file descriptors
- Implementing pipelines
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

Plan For Today

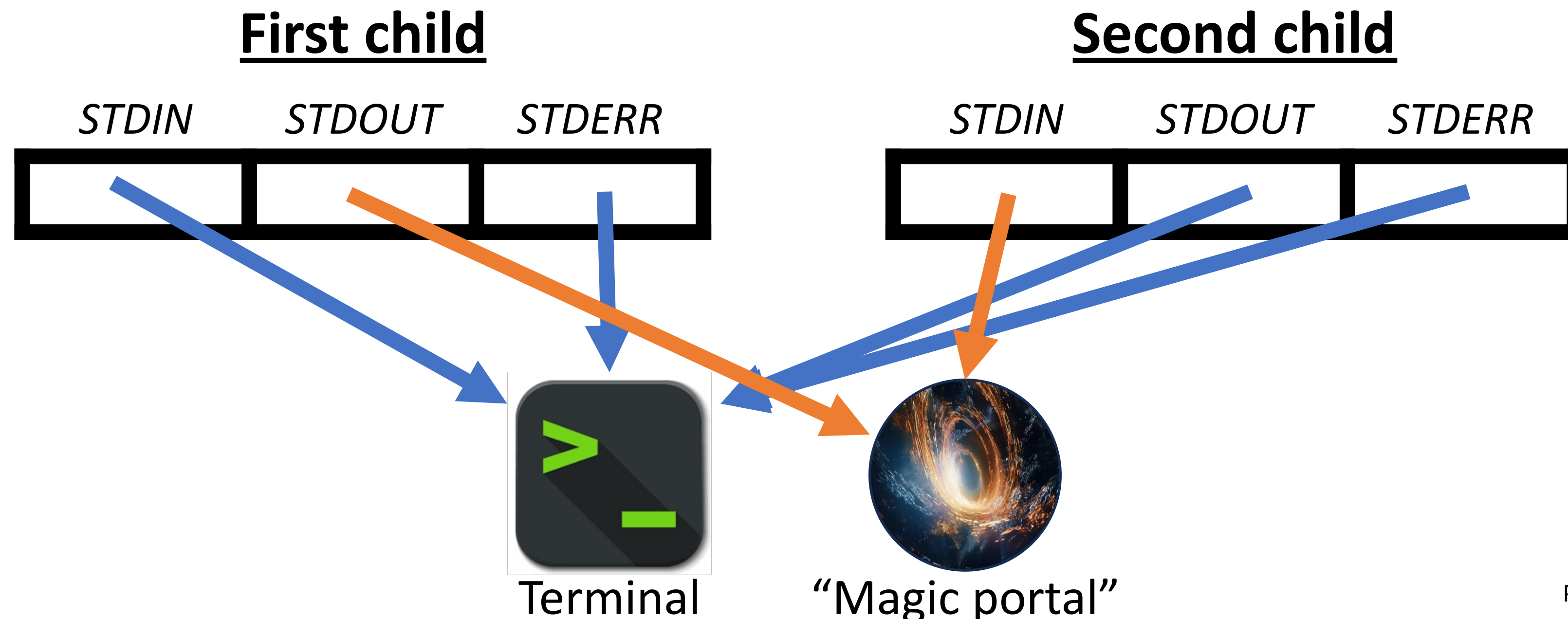
- **Recap: Pipes so far**
- Closing pipes
- **dup2()** and rewiring file descriptors
- Implementing pipelines
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

How do we implement shell pipelines?

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

1. Spawn 2 child processes (1 per command)
2. Create a “magic portal” that allows data to be sent between two processes
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?

   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

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**
3. How do we connect a process's STDIN/STDOUT to this "magic portal"?
   The **dup2()** system call

“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

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

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?

Key idea: the child accesses the same pipe because its file descriptor table is copied, which does not contain the actual pipe data; that is stored in the global "open file table" which is not duplicated on fork.

Open File Table

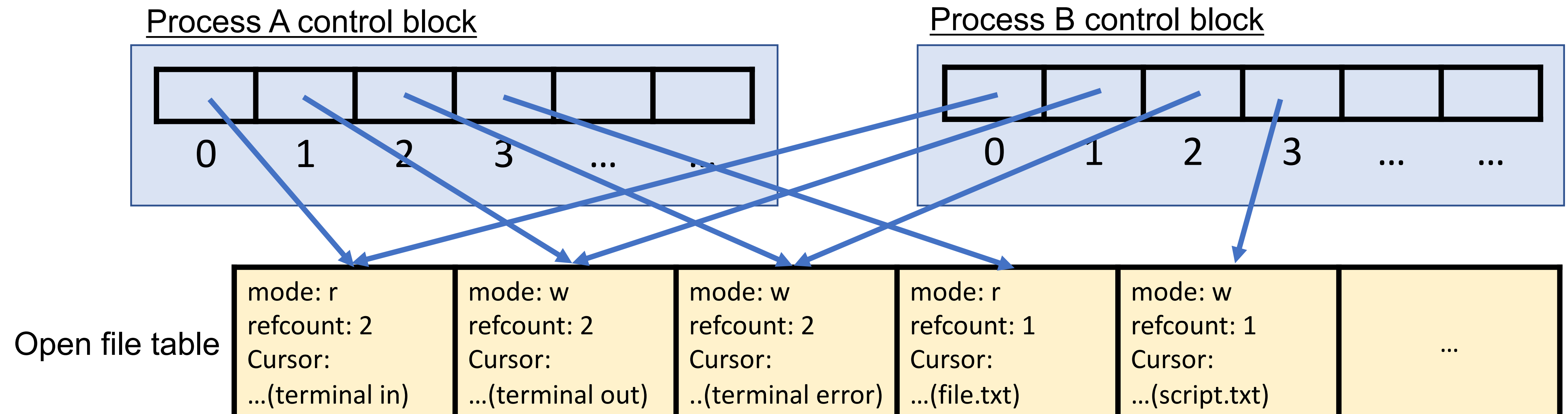
The open file table is complex! Here are the key takeaways of why it's useful to know about how it works:

- It explains why, when we fork off a child process, the child process gets "*shallow copies*" of all parent file descriptors. E.g. parent/child can share a pipe, or same cursor is advanced by both.
- It explains why we need to close those duplicated file descriptors in **both** the parent and the child.

File Descriptor Table

An entry in a file descriptor table is really a *pointer* to an entry in another global table, the **open file table**.

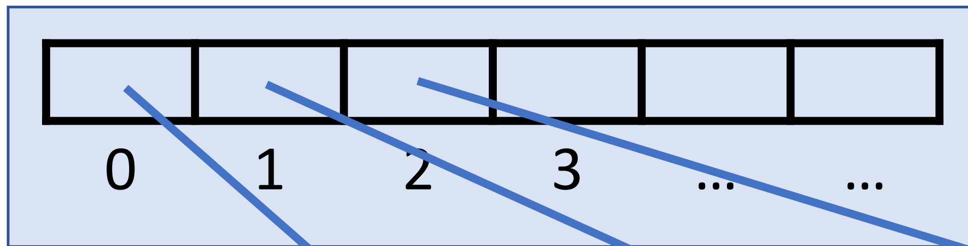
- The **open file table** is one array of information about open files/resources across all processes.
- There is one open file table entry per *session*, not per *file*



Practice: Reference Count

If a process opens a file, and then spawns a child process, what will the reference count be for the corresponding open file table entry?

```
int fd = open("file.txt", O_RDONLY); // fd = 3 here
pid_t pidOrZero = fork();
```



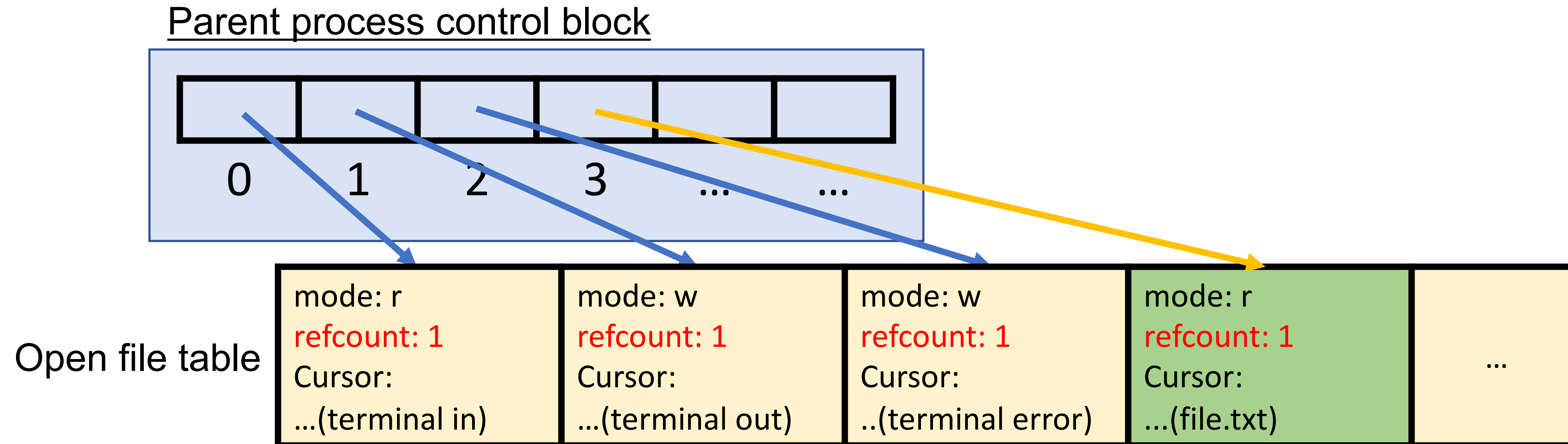
Open file table

mode: r refcount: 1 Cursor: …(terminal in)	mode: w refcount: 1 Cursor: …(terminal out)	mode: w refcount: 1 Cursor: ..(terminal error)	…	…

Practice: Reference Count

If a process opens a file, and then spawns a child process, what will the reference count be for the corresponding open file table entry?

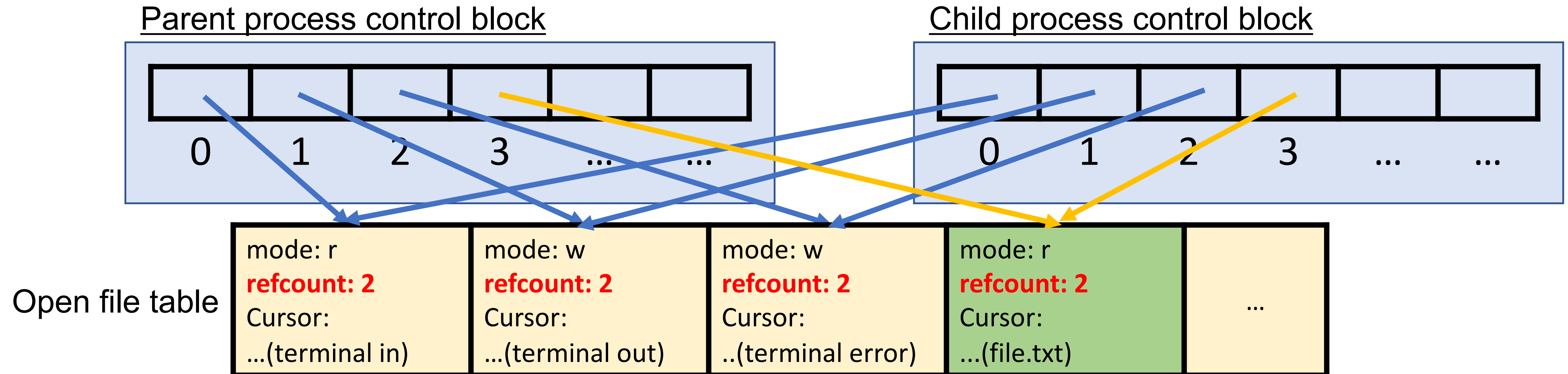
```
int fd = open("file.txt", O_RDONLY); // fd = 3 here
pid_t pidOrZero = fork();
```



Practice: Reference Count

If a process opens a file, and then spawns a child process, what will the reference count be for the corresponding open file table entry?

```
int fd = open("file.txt", O_RDONLY); // fd = 3 here
pid_t pidOrZero = fork();
```

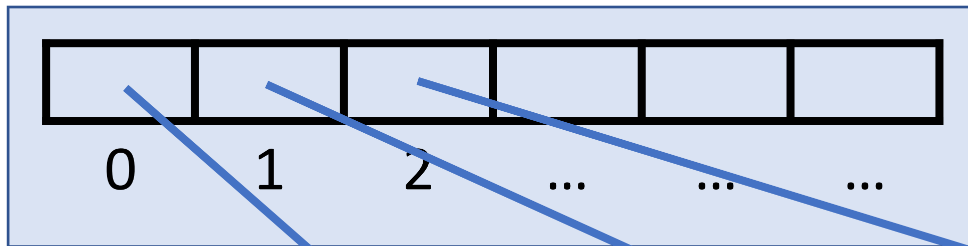


Practice: Reference Count

If a process spawns a child process, and then opens a file, what will the reference count be for the corresponding open file table entry?

```
pid_t pidOrZero = fork();
int fd = open("file.txt", O_RDONLY); // fd = 3 here
```

Parent process control block



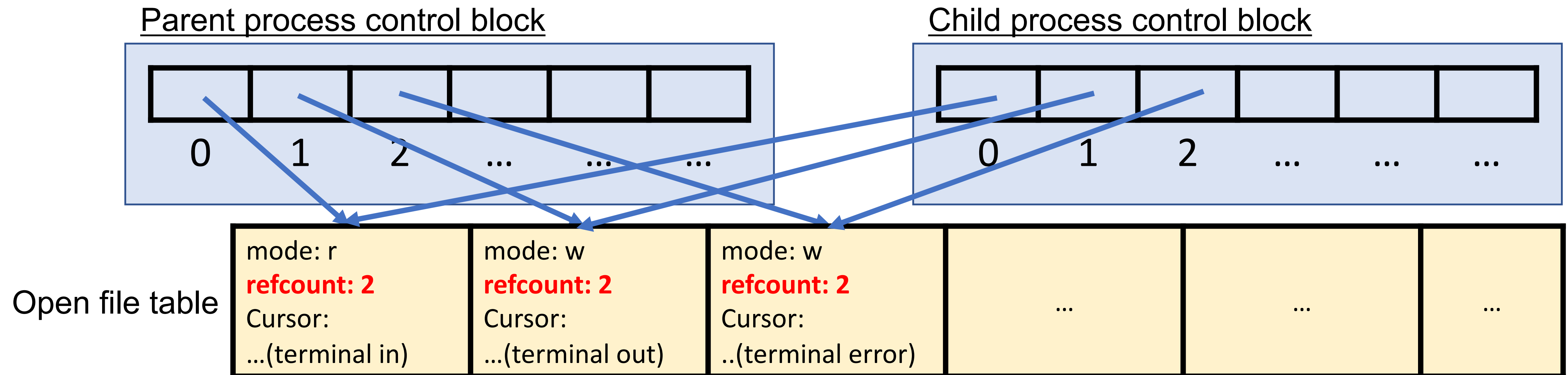
Open file table

mode: r refcount: 1 Cursor: …(terminal in)	mode: w refcount: 1 Cursor: …(terminal out)	mode: w refcount: 1 Cursor: ..(terminal error)	…	…	…

Practice: Reference Count

If a process spawns a child process, and then opens a file, what will the reference count be for the corresponding open file table entry?

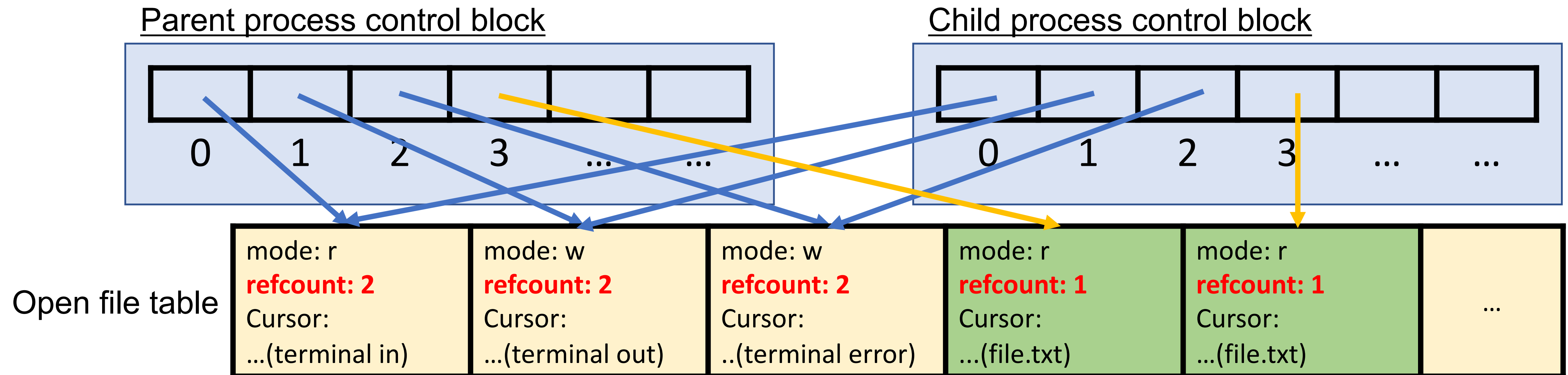
```
pid_t pidOrZero = fork();
int fd = open("file.txt", O_RDONLY); // fd = 3 here
```



Practice: Reference Count

If a process spawns a child process, and then opens a file, what will the reference count be for the corresponding open file table entry?

```
pid_t pidOrZero = fork();
int fd = open("file.txt", O_RDONLY); // fd = 3 here
```



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;
}
```

Plan For Today

- **Recap**: Pipes so far
- **Closing pipes**
- **dup2()** and rewiring file descriptors
- Implementing pipelines
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

Pipe Stalling

Not closing write ends of pipes can cause functionality issues.

- E.g. if the child reads from a pipe, but the parent waits for the child to finish before writing anything, the child will stall waiting for more input.
- E.g. if the child reads until there's nothing left, but the write end was not closed everywhere, it will stall.

Ex: Child reads, parent writes after waitpid

```
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]);
    waitpid(pidOrZero, NULL, 0);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    return 0;
}
```

child stuck here!

Ex: Child reads continually, parent doesn't close

```
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];
        while (true) {
            ssize_t ret = read(fds[0], buffer, sizeof(buffer));
            if (ret == 0) break;
            printf("Message from parent: %s\n", buffer);
        }
        close(fds[0]);
        return 0;
    }
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    waitpid(pidOrZero, NULL, 0);
    close(fds[1]);
    return 0;
```

child stuck here!

Ex: Child reads continually, forgets to close write end itself

```
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];
        while (true) {
            ssize_t ret = read(fds[0], buffer, sizeof(buffer));
            if (ret == 0) break;
            printf("Message from parent: %s\n", buffer);
        }
        close(fds[0]);
        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 line `close(fds[1]);` in the child is struck through.)

child stuck here! (arrow pointing to the `read(...)` line)

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

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

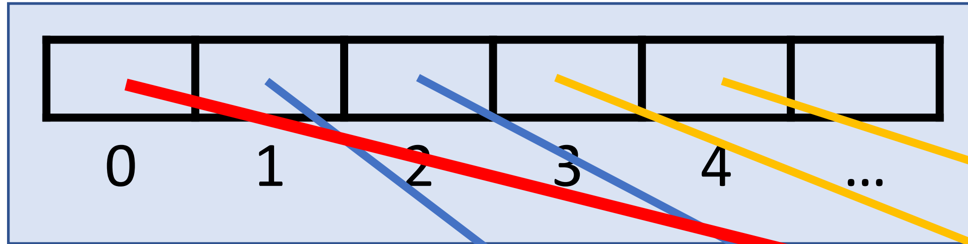
   The dup2() system call

Plan For Today

- **Recap**: Pipes so far
- Closing pipes
- **dup2() and rewiring file descriptors**
- Implementing pipelines
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

First Goal: Rewiring STDIN



Open file table

mode: r refcount: X (terminal in)	mode: w refcount: X (terminal out)	mode: w refcount: X (terminal error)	mode: r refcount: 2 (pipe read end)	mode: w refcount: 1 (pipe write end)	...

```
dup2(srcfd, dstfd)
```

```
e.g. dup2(3, 0)
```

Redirecting Process I/O

dup2 makes a copy of a file descriptor entry and puts it in another file descriptor index. This means both will now point to the same open file table entry. If the second parameter is an already-open file descriptor, it is closed before being used.

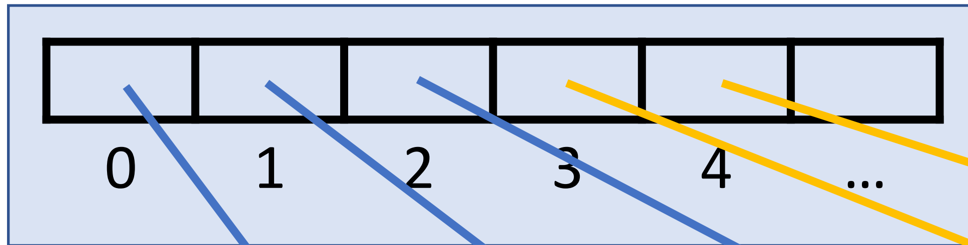
```
int dup2(int srcfd, int dstfd);
```

Example: we can use **dup2** to copy the pipe read file descriptor into standard input! Then we can close the original pipe read file descriptor.

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

If we change file descriptors 0-2, we can redirect STDIN/STDOUT/STDERR to be something else without the program knowing!

Redirecting Process I/O

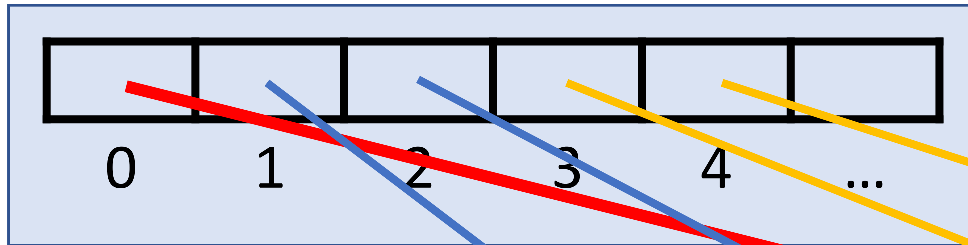


Open file table

mode: r refcount: X (terminal in)	mode: w refcount: X (terminal out)	mode: w refcount: X (terminal error)	mode: r refcount: 1 (pipe read end)	mode: w refcount: 1 (pipe write end)	...

```
int fds[2];
pipe(fds); // assume fds[0] is 3 and fds[1] is 4
dup2(fds[0], STDIN_FILENO);
close(fds[0]);
```

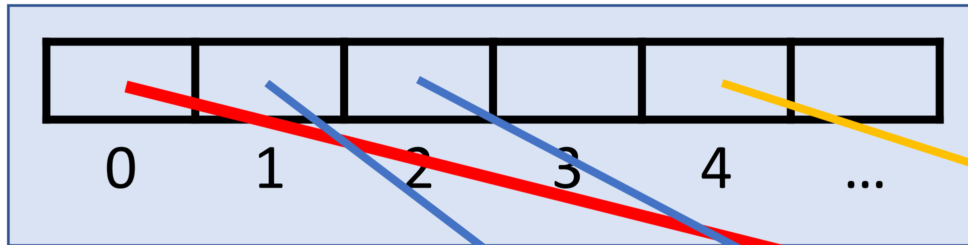
Redirecting Process I/O



Open file table					
mode: r refcount: X (terminal in)	mode: w refcount: X (terminal out)	mode: w refcount: X (terminal error)	mode: r refcount: 2 (pipe read end)	mode: w refcount: 1 (pipe write end)	...

```
int fds[2];
pipe(fds); // assume fds[0] is 3 and fds[1] is 4
dup2(fds[0], STDIN_FILENO);
close(fds[0]);
```

Redirecting Process I/O



Open file table					
mode: r refcount: X (terminal in)	mode: w refcount: X (terminal out)	mode: w refcount: X (terminal error)	mode: r refcount: 1 (pipe read end)	mode: w refcount: 1 (pipe write end)	…

```
int fds[2];
pipe(fds); // assume fds[0] is 3 and fds[1] is 4
dup2(fds[0], STDIN_FILENO);
close(fds[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
- 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

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Plan For Today

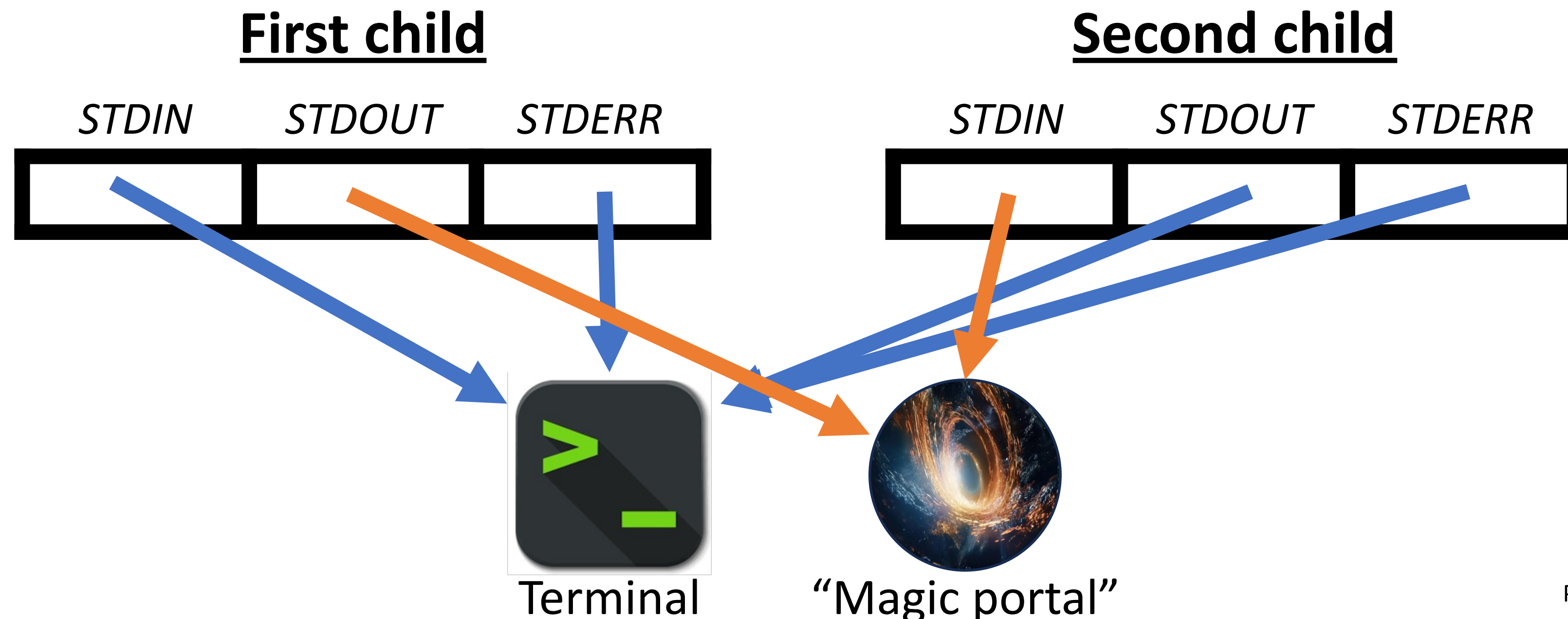
- **Recap**: Pipes so far
- Closing pipes
- **dup2()** and rewiring file descriptors
- **Implementing pipelines**
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

How do we implement shell pipelines?

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

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

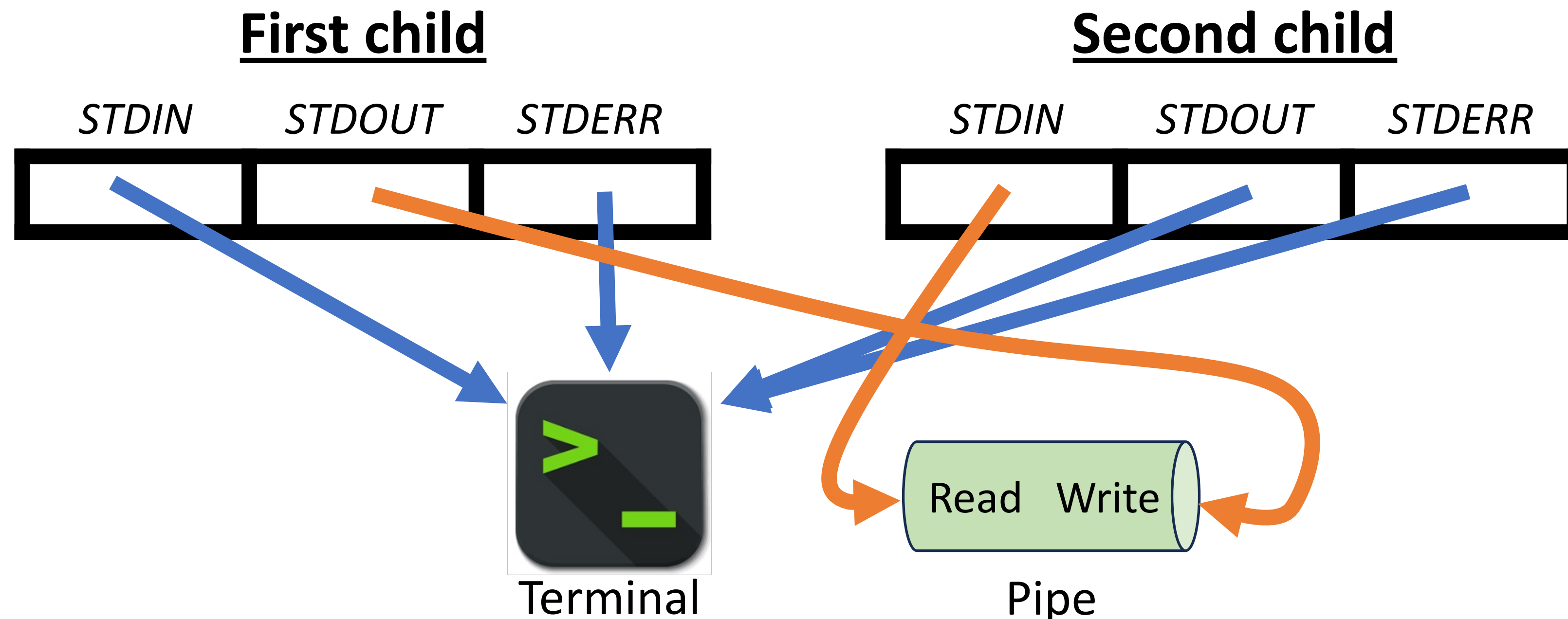


Portal image unmodified, CC credit

How do we implement shell pipelines?

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.



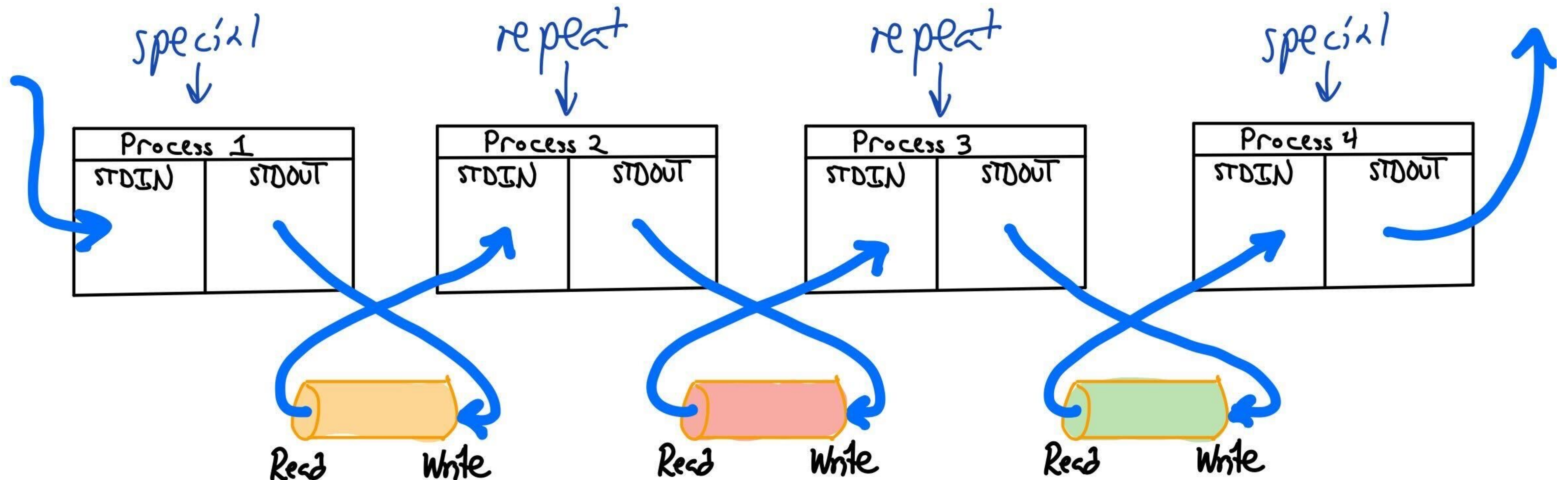
A Secret About execvp

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!*

Implementing Shell Pipelines

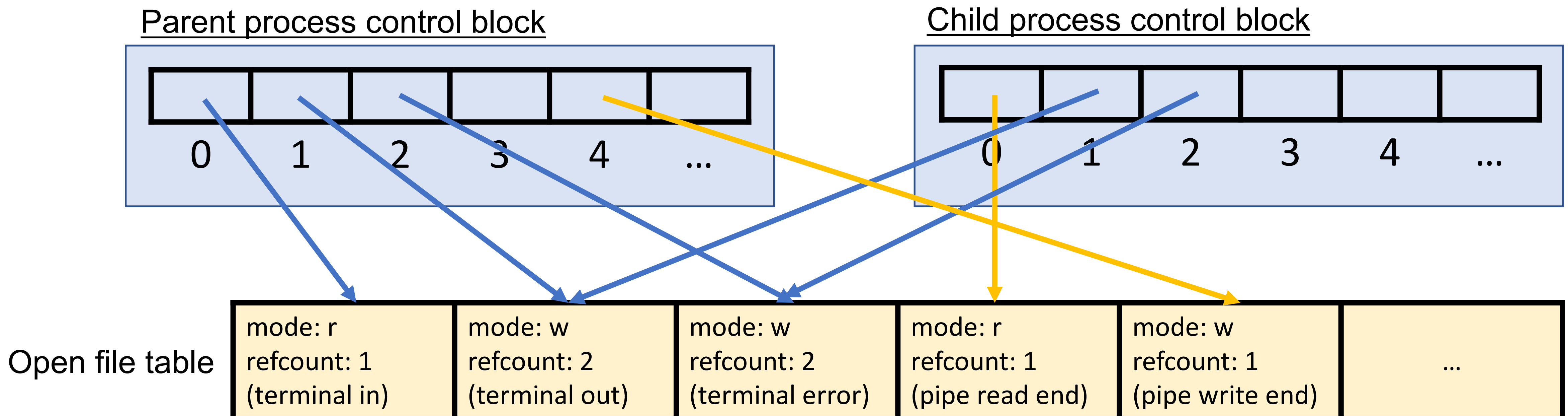
On assign3, you'll build up your shell implementation to support arbitrary-length pipelines. You'll need to spawn **N** child processes and **N-1** pipes.



Practice: Subprocess

To practice with **dup2** and rewiring, let's implement **subprocess**, a function that spawns a child and connects a pipe to it so that the parent can write to the pipe to send data to the child's STDIN.

This is useful because we can spawn and run any other program, even if we don't have the source code for it, and feed it input.



subprocess

To practice this piping technique, let's implement a custom function called **subprocess**.

```
subprocess_t subprocess(char *command);
```

subprocess spawns a child to run the specified command and returns its PID as well as a file descriptor we can write to to write to its STDIN.

It returns a struct containing:

- the PID of the child process
- a file descriptor we can use to write to the child's STDIN

subprocess-soln.cc

subprocess

```
int main(int argc, char *argv[]) {
	// Spawn a child that is running the sort command
	subprocess_t sp = subprocess("/usr/bin/sort");

	// We would like to feed these words as input to sort
	const char *words[] = { "felicity", "umbrage", "susurration", "halcyon", 
"pulchritude", "ablution", "somnolent", "indefatigable" };

	// write each word on its own line to the STDIN of the child sort process
	for (size_t i = 0; i < sizeof(words) / sizeof(words[0]); i++) {
		dprintf(sp.supplyfd, "%s\n", words[i]);
	}

	// Close the write FD to indicate the input is closed
	close(sp.supplyfd);
	// Wait for the child to finish before exiting
	waitpid(sp.pid, NULL, 0);
	return 0;
}
```

subprocess

Implementing subprocess:

1. Create a pipe
2. Spawn a child process
3. That child process changes its STDIN to be the pipe read end (how?)
4. That child process calls **execvp** to run the specified command
5. We return the pipe write end to the caller along with the child's PID. That caller can write to the file descriptor, which appears to the child as its STDIN

subprocess

```
subprocess_t subprocess(const char *command) {
    // this line parses the command into a pipeline like is done for you on assign3
    pipeline p(command);

    // Make a pipe
    int fds[2];
    pipe(fds);

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // We are not writing to the pipe, only reading from it
        close(fds[1]);

        // Duplicate the read end of the pipe into STDIN
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);

        // Run the command
        execvp(p.commands[0].argv[0], p.commands[0].argv);
        exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
    }
    ...
```

Plan For Today

- **Recap**: Pipes so far
- Closing pipes
- **dup2()** and rewiring file descriptors
- Implementing pipelines
- ***Practice:*** implementing **subprocess**
- **I/O Redirection with files**

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

Redirecting Process I/O to/from a File

There is one final shell feature we can use our understanding of file descriptors to implement, I/O Redirection with a file:

This saves the output to a file instead of printing it to the terminal

```
sort file.txt > output.txt
```

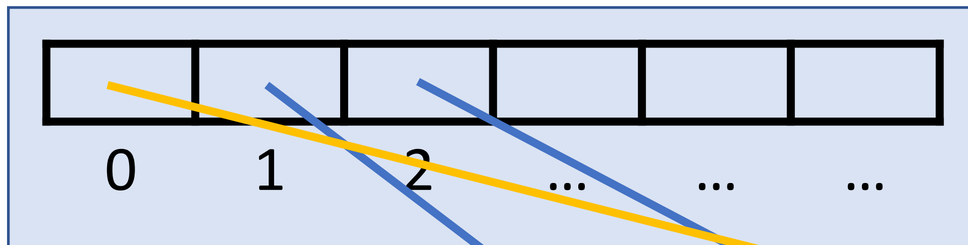
This reads input from a file instead of reading from the terminal

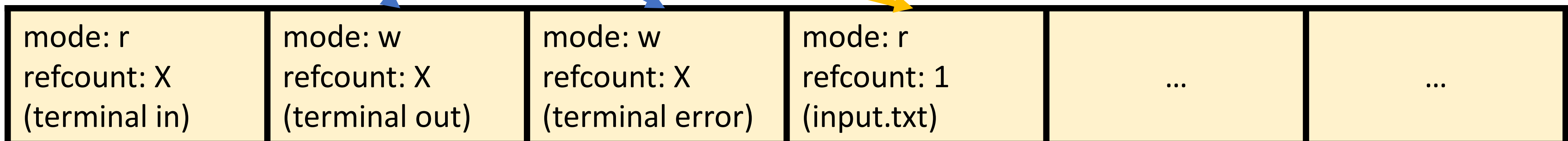
```
sort < input.txt
```

Consider how we can use our knowledge of file descriptors to implement this functionality on assign3!

Practice: Subprocess

Example: `sort < input.txt`





assign3

Implement your own shell! ("stsh" – Stanford Shell)

4 key features:

- Run a single command and wait for it to finish
- Run 2 commands connected via a pipe
- Run an arbitrary number of commands connected via pipes
- Have command input come from a file, or save command output to a file

Recap

- **Recap**: Pipes so far
- Closing pipes
- **dup2()** and rewiring file descriptors
- Implementing pipelines
- ***Practice:*** implementing **subprocess**
- I/O Redirection

Lecture 11 takeaway: We can share pipes with child processes and change FDs 0-2 to connect processes and redirect their I/O.

Next time: introduction to multithreading

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```