CS111, Lecture 10
Pipes

😷 masks strongly recommended

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 → Managing processes and running other programs → Inter-process communication with pipes

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

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

Plan For Today

- **<u>Recap</u>: fork, waitpid, execvp and our first shell**
- **Shell Feature Demo:** pipes
- **pipe()** system call
- **Example:** Parent-child pipe

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

Our Goal: Shell

A *shell* is a program that prompts the user for a command to run, runs that command, waits for the command to finish, and then prompts the user again.

```
while (true) {
      char *user_command = … // user input
      pid_t pidOrZero = fork();
      if (pidOrZero == 0) {
            // run user's command in the child, then terminate
                        execvp
      }

      // parent waits for child before continuing
                  waitpid
}
```

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 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

Make sure to clean up after your zombie children.

(wait, what?)

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

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.

First Shell Solution

```
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);
}
```

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**

More About fork()

When you fork off a child process, the child will keep running like any other program, until it eventually returns from main (or exits using **exit()**).

- E.g. if you call fork() in a helper function (which is allowed), if the child process does not e.g. call **exit** in that helper function when it’s done, it will also return from the function and continue executing.
- We must be careful to ensure that the child doesn’t accidentally run code intended only for the parent!

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! (DEMO)

Plan For Today

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

```
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

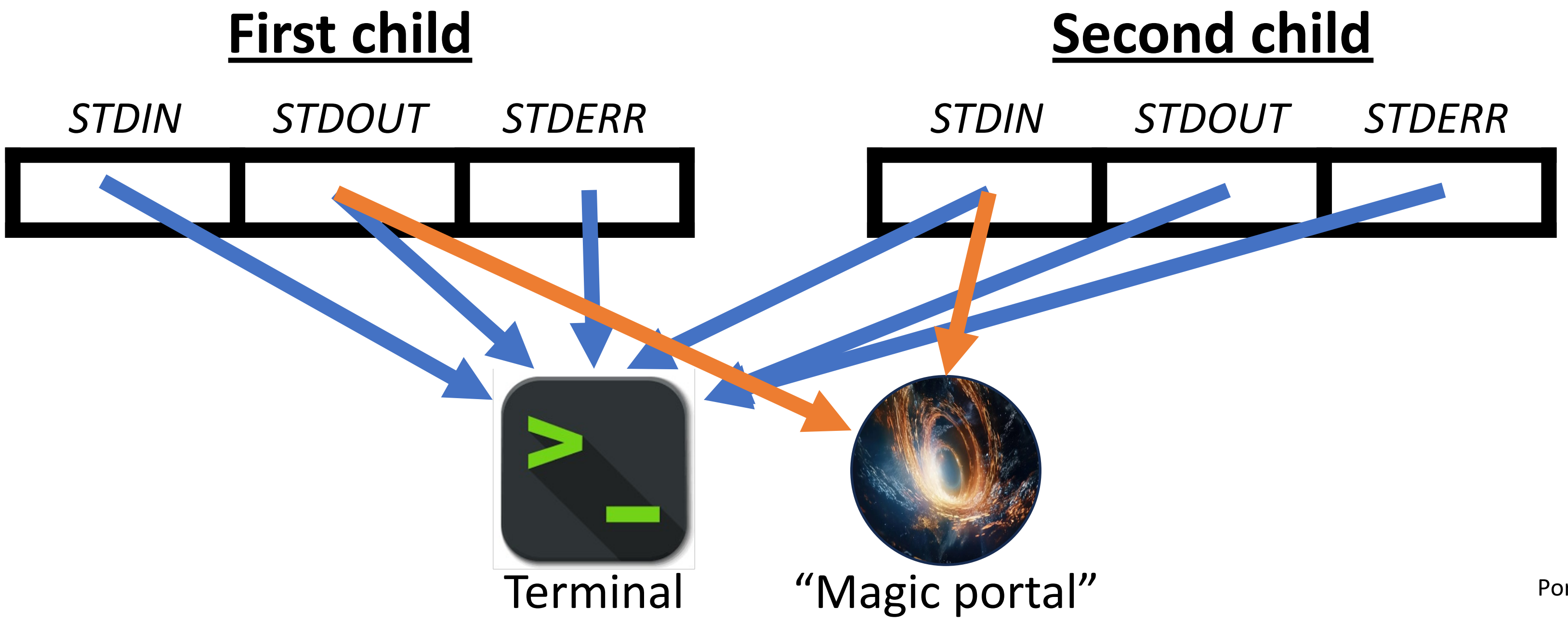
<u>Key Unix idea:</u> chaining the output (STDOUT) of one command to be the input (STDIN) of another.

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. 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

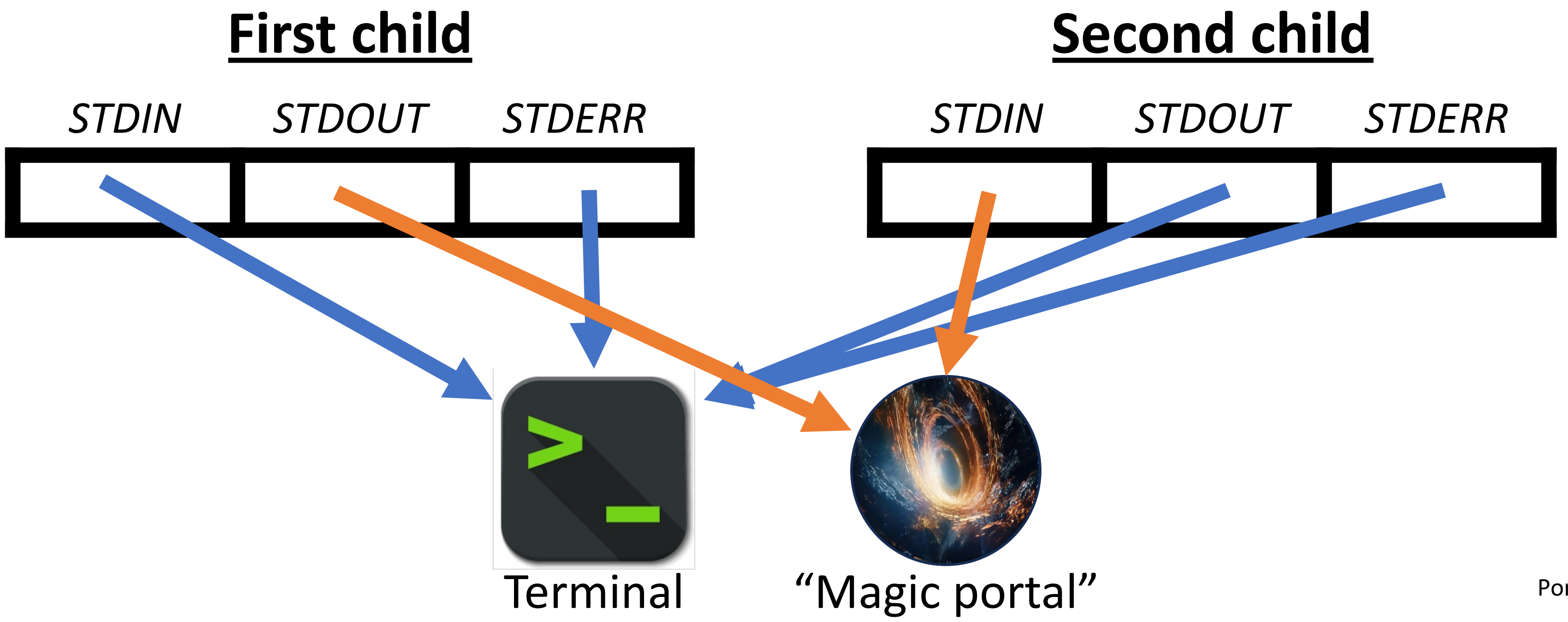


Portal image unmodified, CC credit

How do we implement shell pipelines?

Three key questions:

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

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

Plan For Today

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

```
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

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!

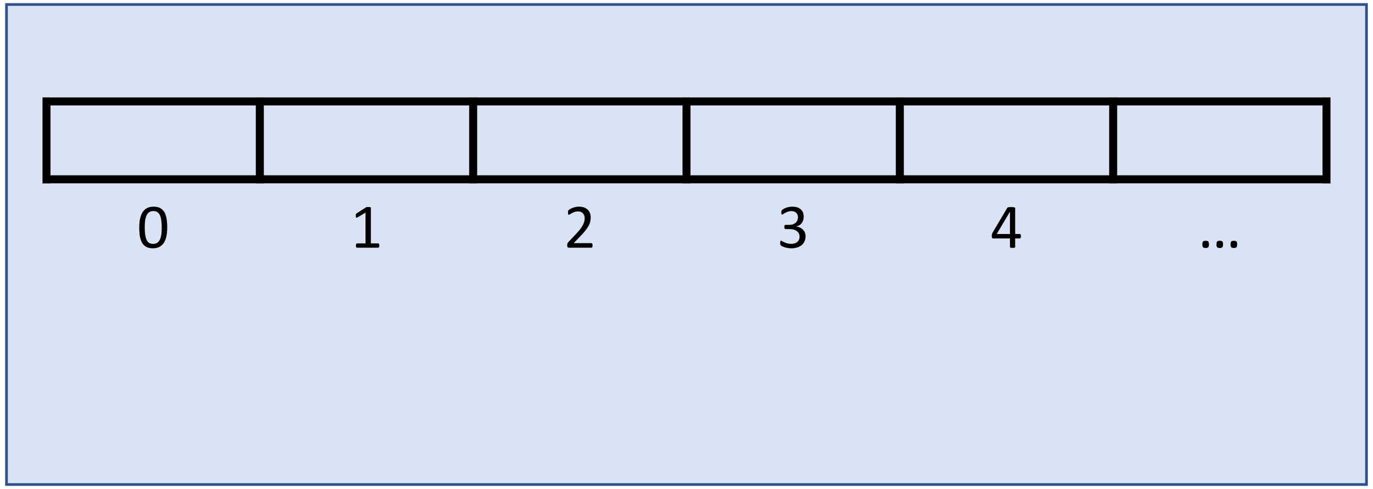
How does this file descriptor duplication work?

File Descriptor Table

The OS maintains a "Process Control Block" for each process containing info about it. This includes a process's *file descriptor table,* an array of info about open files/resources for this process.

Key idea: a file descriptor is **an index into that process's file descriptor table!**

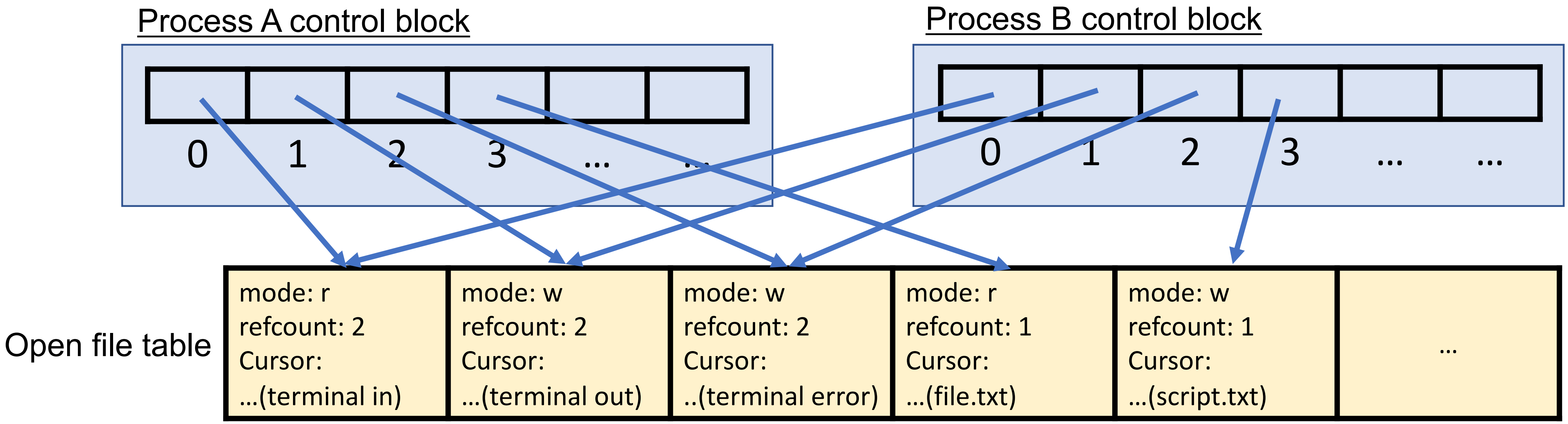
Process Control Block



File Descriptor Table

Key idea: a file descriptor is **an index into that process's file descriptor table**.

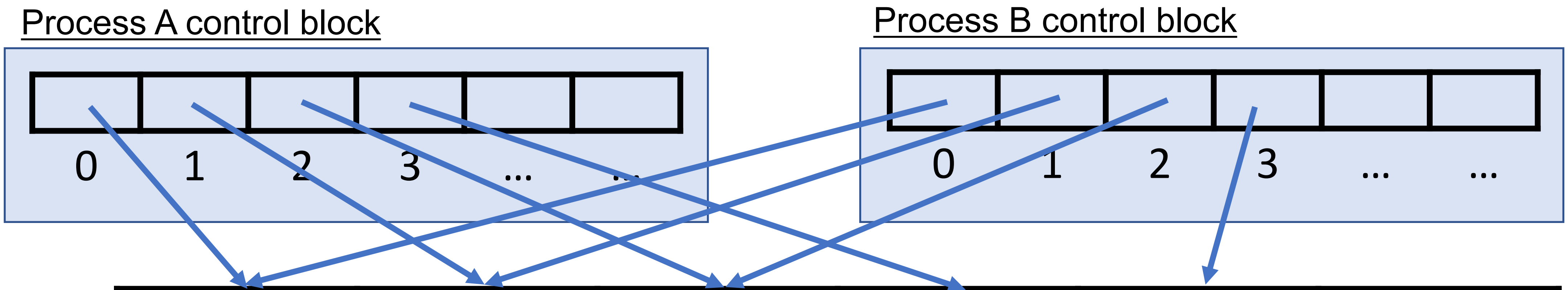
- An entry in a file descriptor table is really a *pointer* to an entry in another global table, the **open file table**.
- The **open file table** is one array of information about open files/resources across all processes.



File Descriptor Table

An open file table entry contains various information, such as:

- **mode:** e.g., read, write, read+write
- **Reference count:** the number of file descriptor table entries pointing to it
- **Cursor**: tracking where in the file it currently is

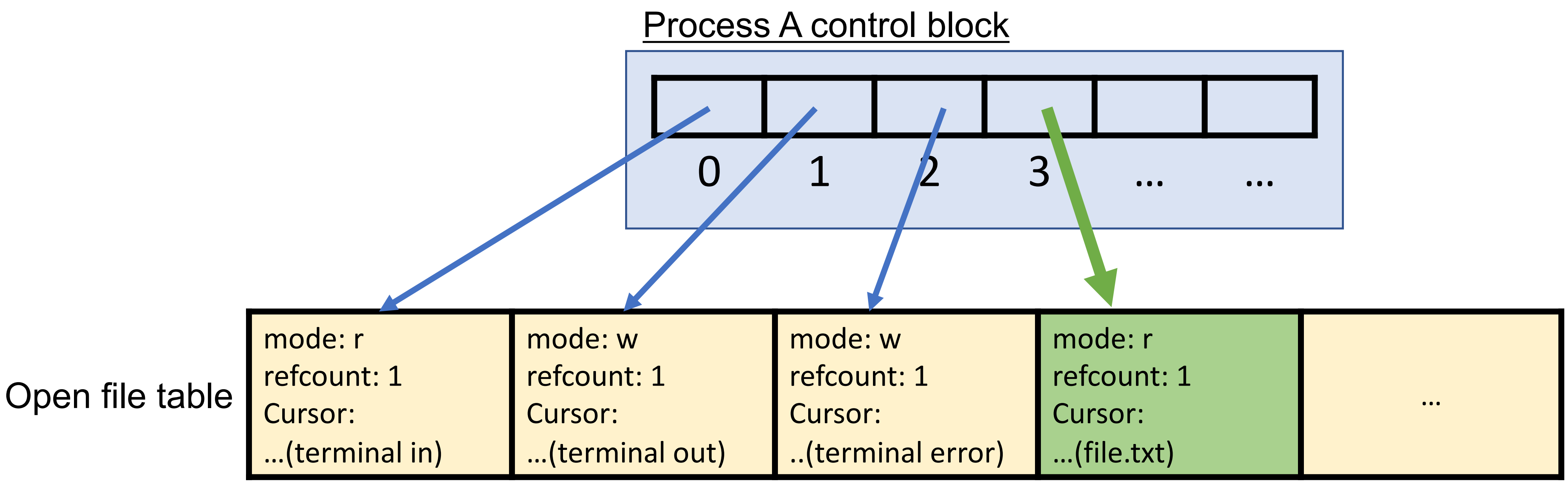


mode: r refcount: 2 Cursor: …(terminal in)	mode: w refcount: 2 Cursor: …(terminal out)	mode: w refcount: 2 Cursor: ..(terminal error)	mode: r refcount: 1 Cursor: …(file.txt)	mode: w refcount: 1 Cursor: …(script.txt)	…

Open File Table

Calling **open** creates a new open file table entry, and a new file descriptor index points to it.

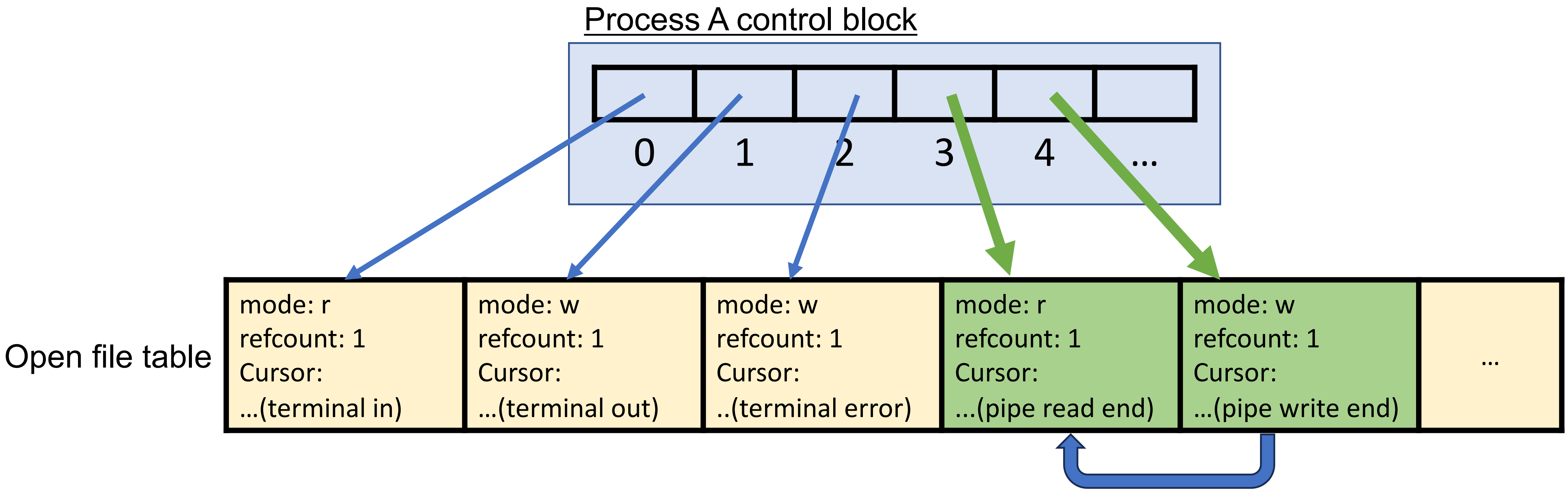
```
int fd = open("file.txt", O_RDONLY); // 3
```



Open File Table

Calling **pipe** creates 2 new open file table entries, and 2 new file descriptor indexes point to them. The open file table entries are linked behind the scenes.

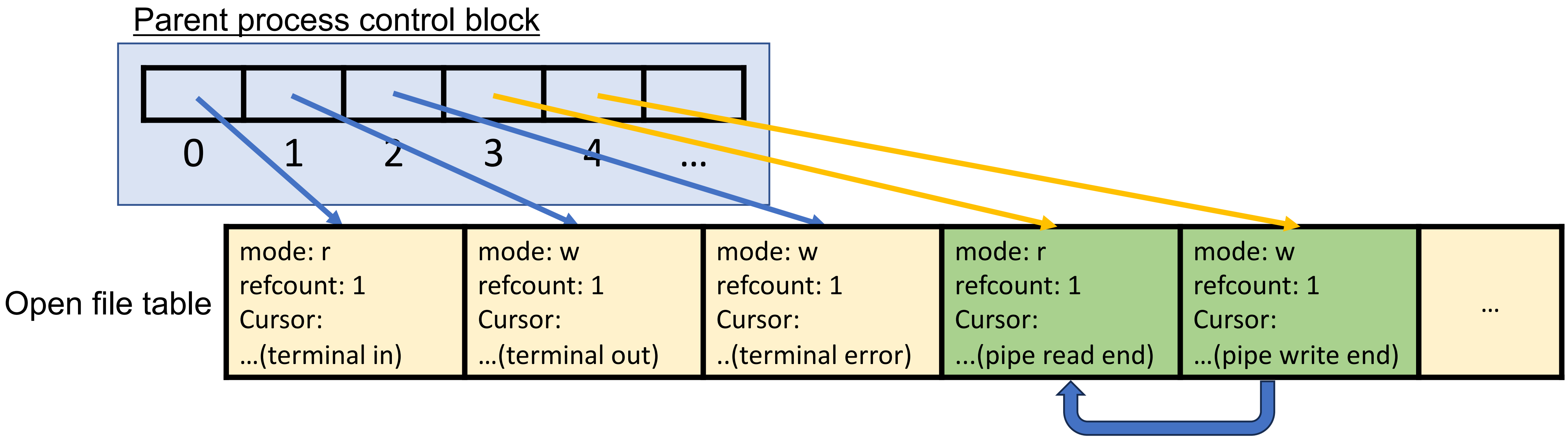
```
int fds[2];
pipe(fds); // afterwards, fds[0] = 3, fds[1] = 4
```



Open File Table

Calling **fork** means the OS creates a new Process Control Block with a copy of parent's FD table; so, all file descriptor indexes point to the same place!

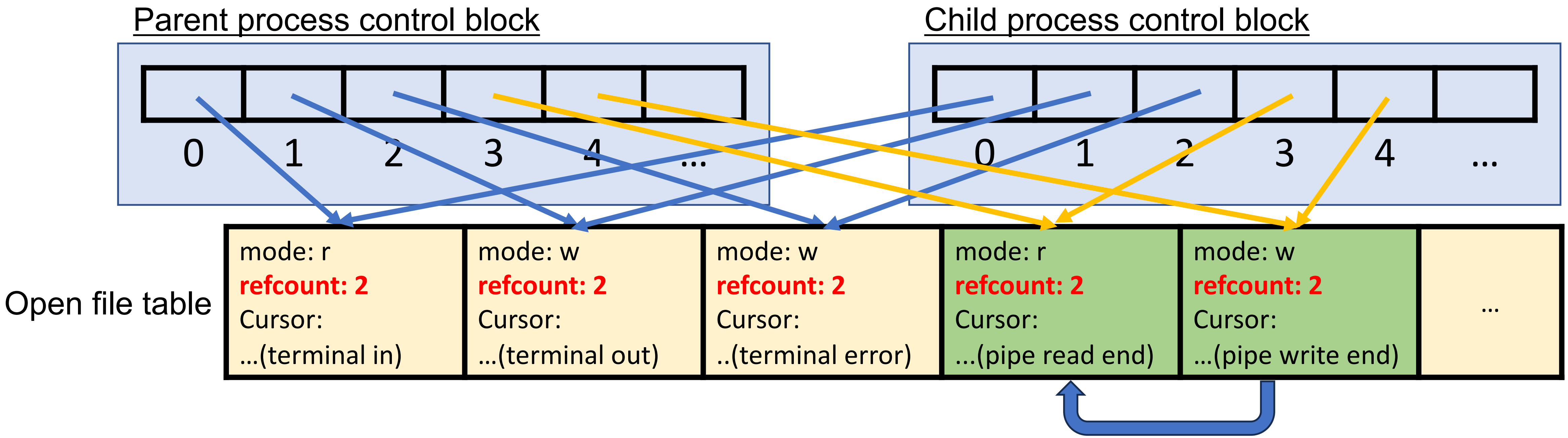
```
int fds[2];
pipe(fds); // afterwards, fds[0] = 3, fds[1] = 4
pid_t pidOrZero = fork();
```



Open File Table

Calling **fork** means the OS creates a new Process Control Block with a copy of parent's FD table; so, all file descriptor indexes point to the same place!

```
int fds[2];
pipe(fds); // afterwards, fds[0] = 3, fds[1] = 4
pid_t pidOrZero = fork();
```



Reference Count

- When we call **close**, that makes the file descriptor index no longer point to an open file table entry, and the old open file table entry's ref count is decremented.
- When open file table entry's ref count == 0, it's deleted

Practice: Reference Count

a) 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?

b) What about if a process spawns a child process and *then* opens a file?

Respond on PollEv: pollev.com/cs111fall23
or text CS111FALL23 to 22333 once to join.

What will the reference counts be in each case?

- a) 2, b) 2 — 0%
- a) 2, b) 1 — 0%
- a) 1, b) 2 — 0%
- a) 1, b) 1 — 0%

Practice: Reference Count

a) 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? **2**.

b) What about if a process spawns a child process and *then* opens a file? **1**.

(a) explains why we must close this file in both the parent *and* child.

```
int fd = open(…);
pid_t pidOrZero = fork();
if (pidOrZero == 0) {
      …
      close(fd);
} else {
      …
      close(fd);
}
```

pipe()

pipe can allow processes to communicate!

- 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 (or vice versa)
- **Key Idea: read()** *blocks* until:
  a) At least 1 byte is available, OR
  b) "End of file" is reached (for pipe, means all pipe write ends are closed, so no more can be written to it)

Plan For Today

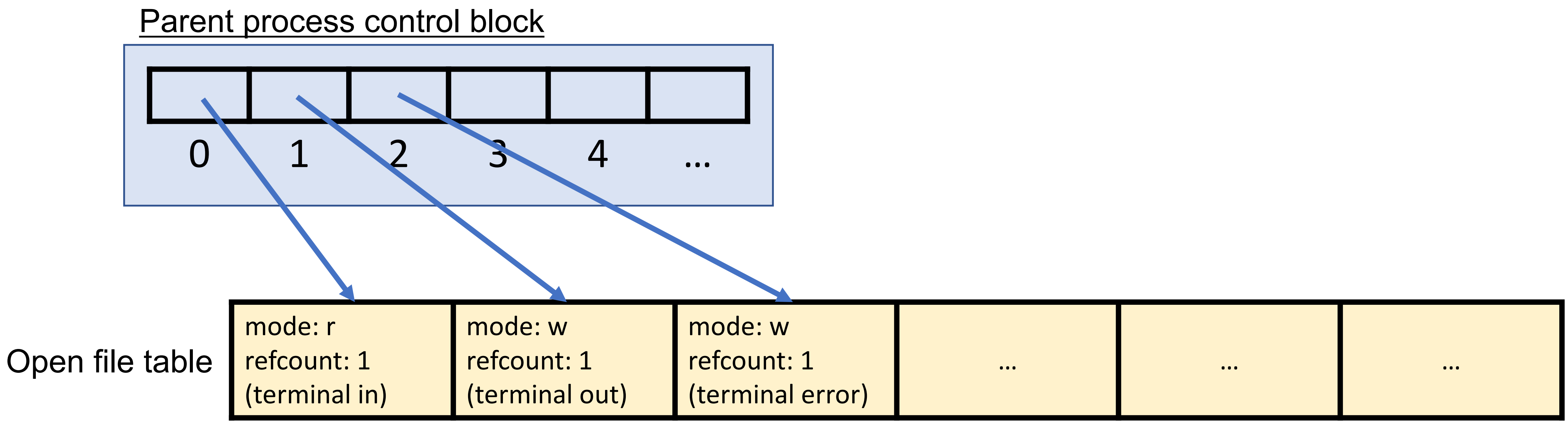
- Recap: **fork, waitpid, execvp** and our first shell
- **Shell Feature Demo:** pipes
- **pipe()** system call
- **Example: Parent-child pipe**

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

Demo: Parent Child Pipe

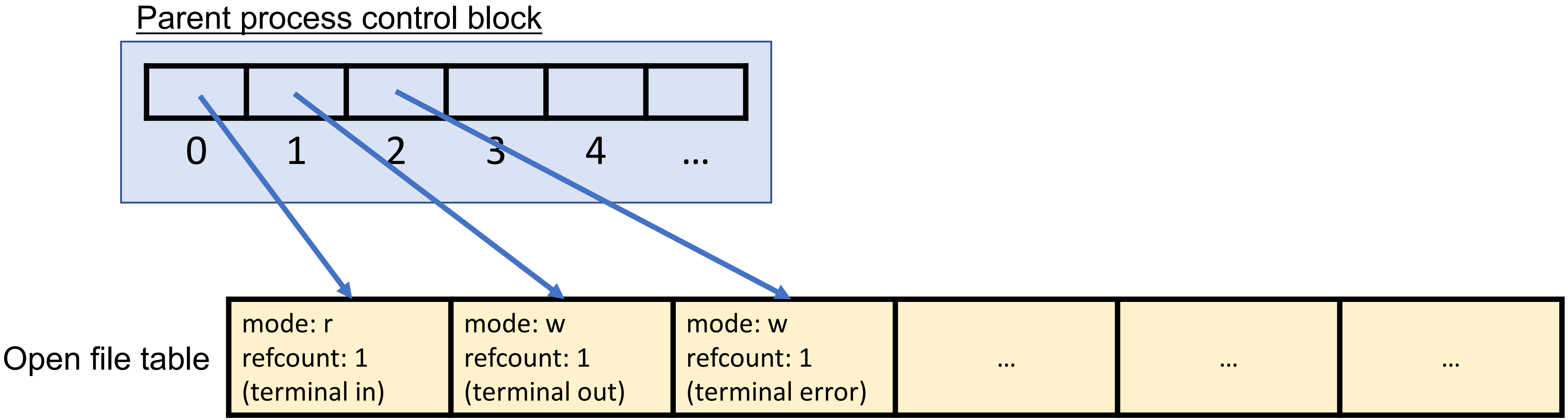
Let's write a program where the parent sends a predetermined message to the child, which prints it out.

Demo: Parent Child Pipe



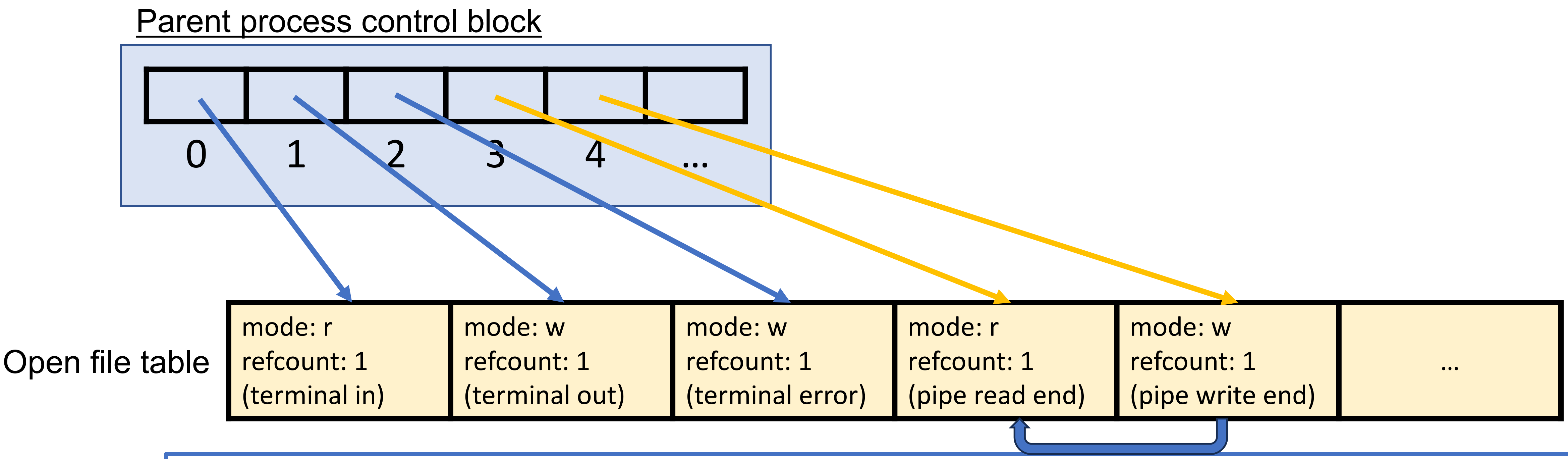

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe
    ...
}
```

Demo: Parent Child Pipe



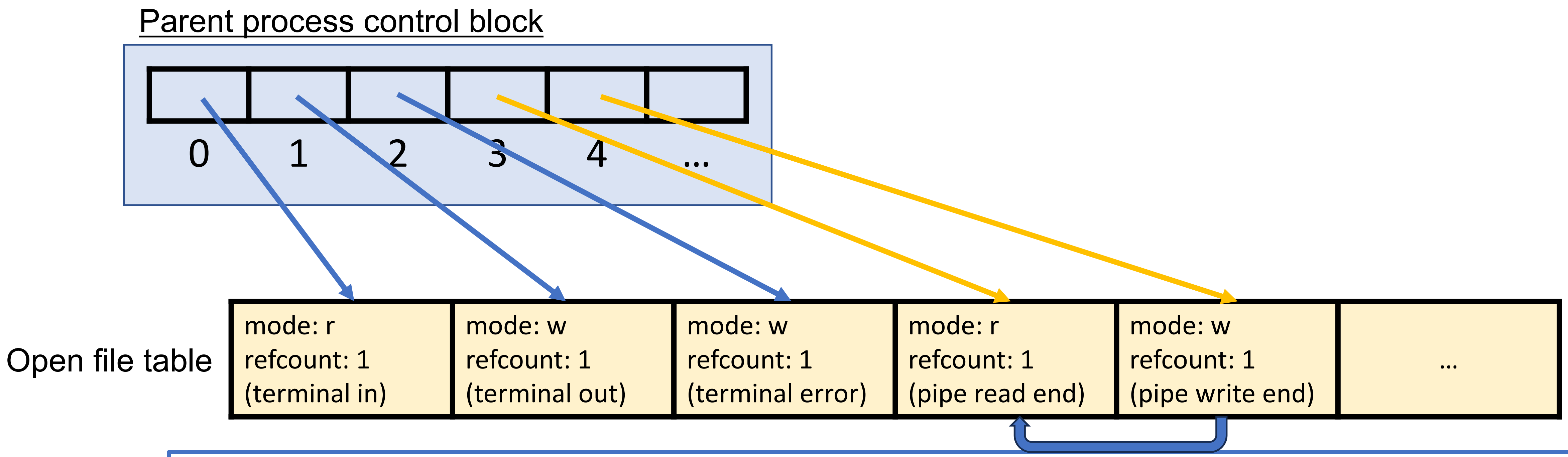

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe
    ...
}
```

Demo: Parent Child Pipe



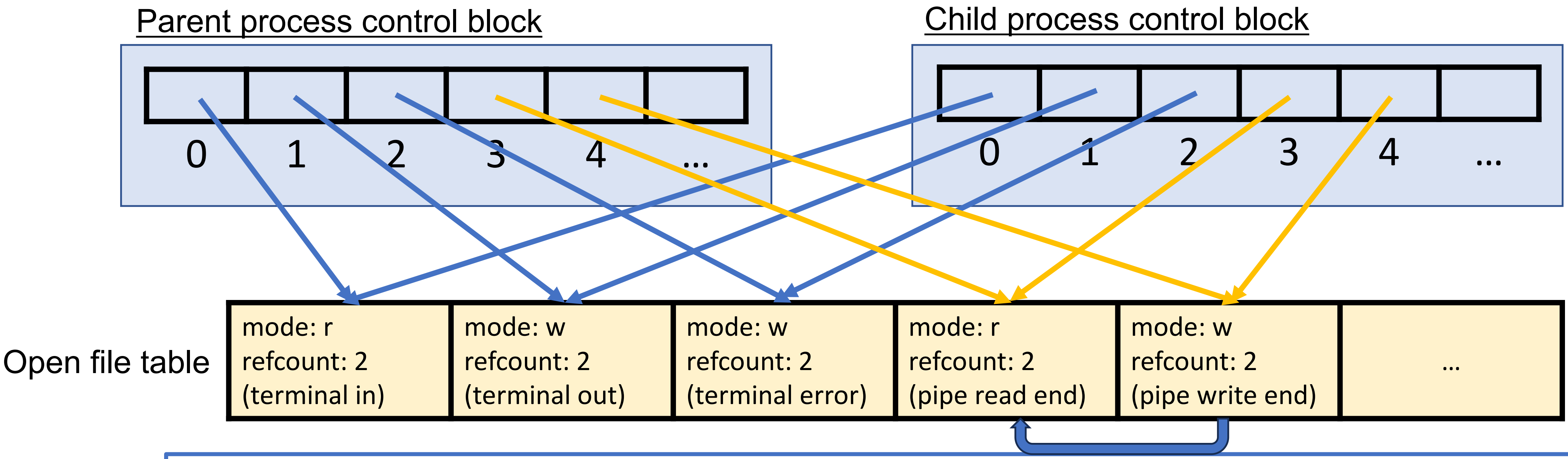

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);  // here, fds[0] = 3, fds[1] = 4
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe
    ...
}
```

Demo: Parent Child Pipe



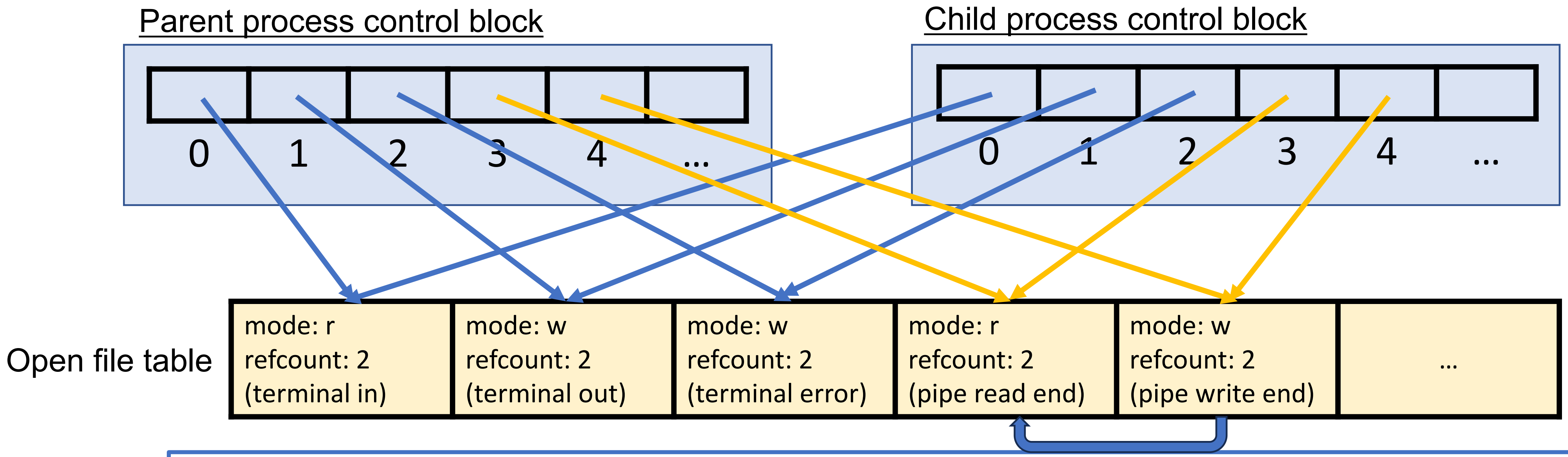

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);  // here, fds[0] = 3, fds[1] = 4
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe
    ...
}
```

Demo: Parent Child Pipe



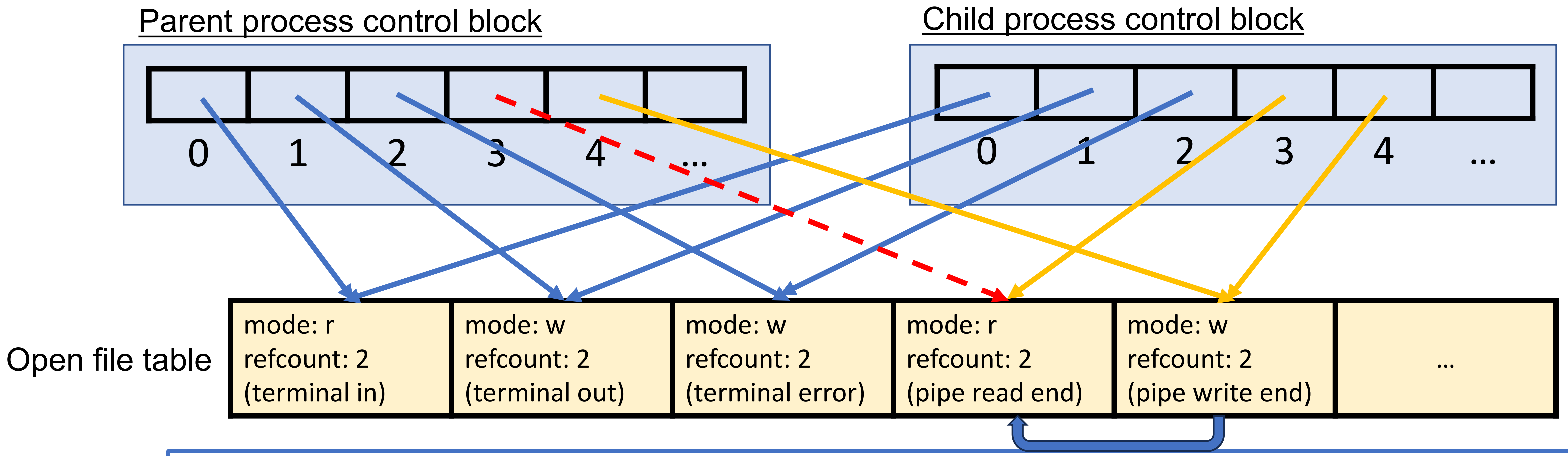

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);  // here, fds[0] = 3, fds[1] = 4
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe
    ...
}
```

Demo: Parent Child Pipe



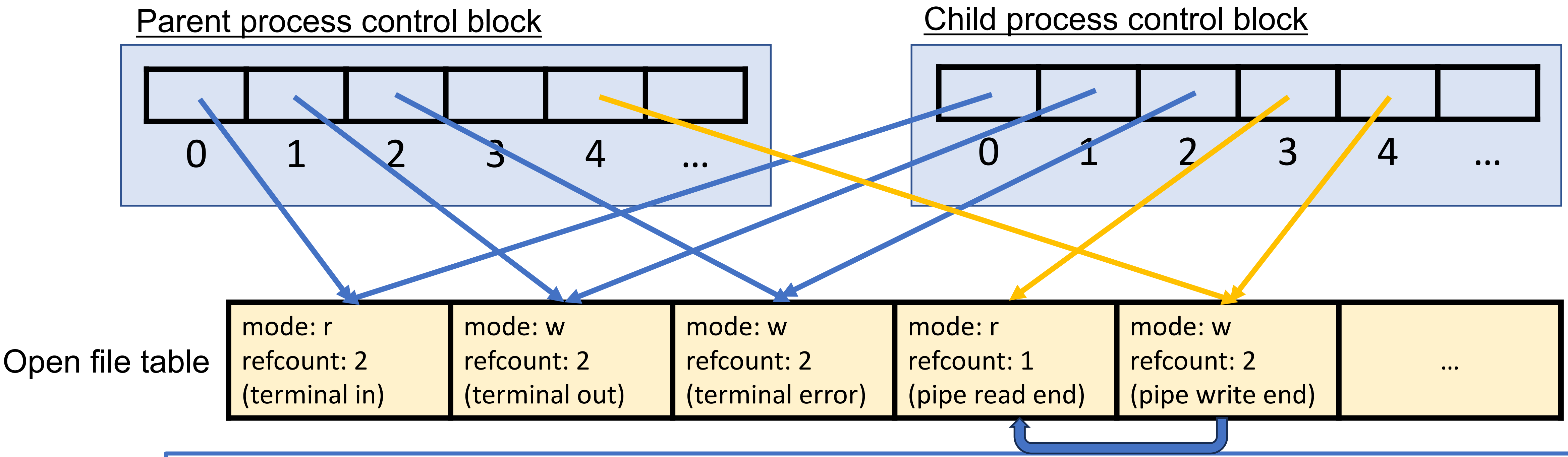

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);  // here, fds[0] = 3, fds[1] = 4
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe
    ...
}
```

Demo: Parent Child Pipe



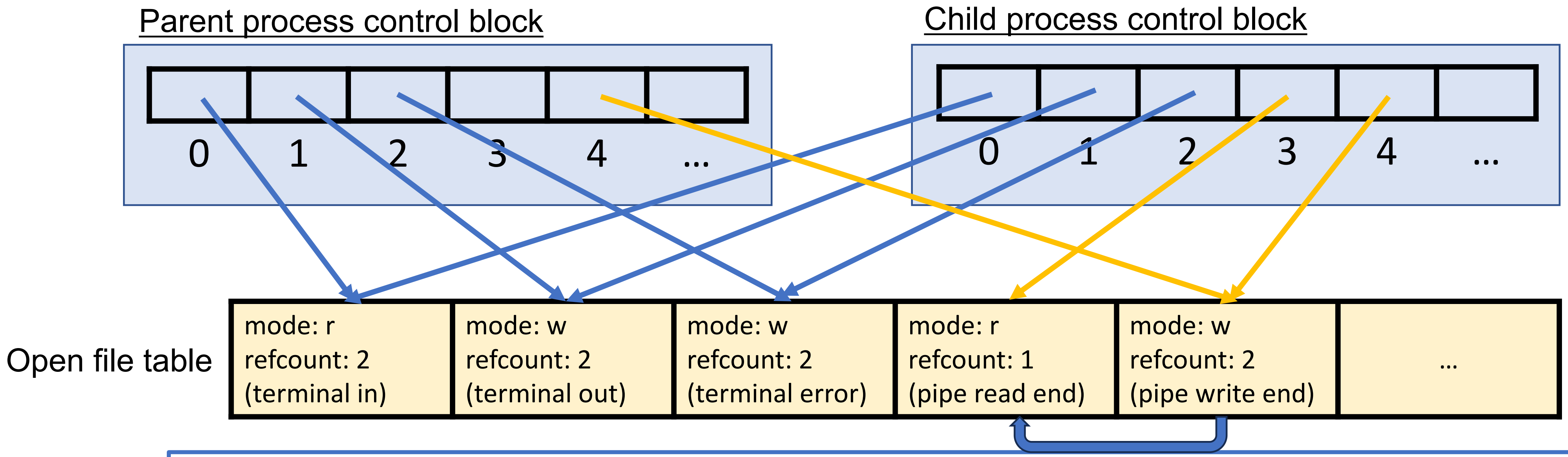

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



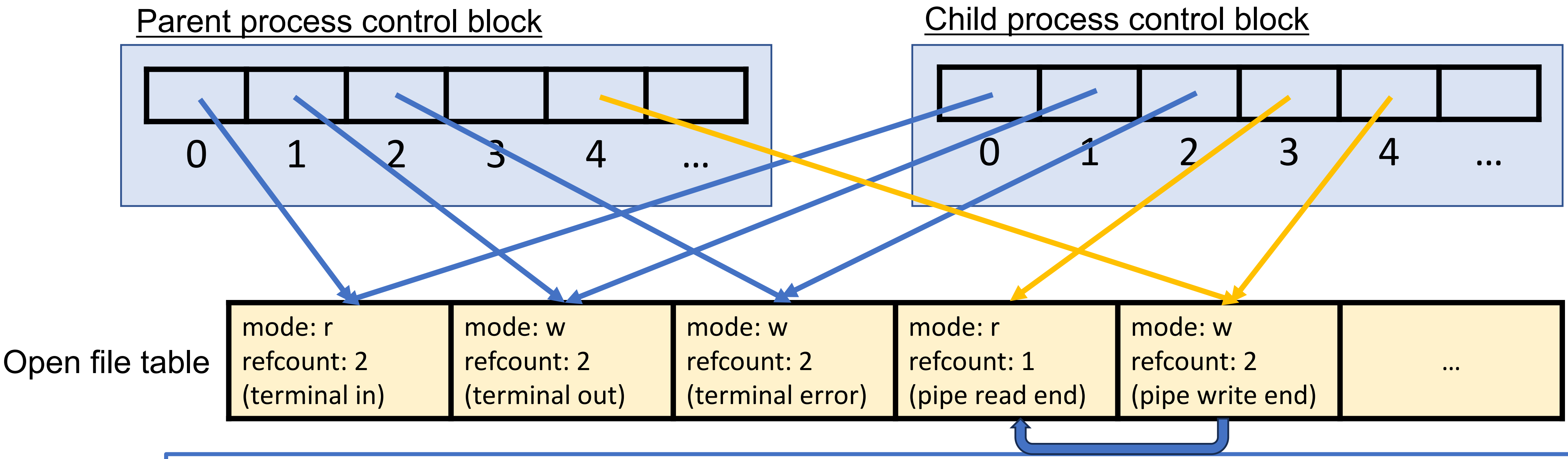

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



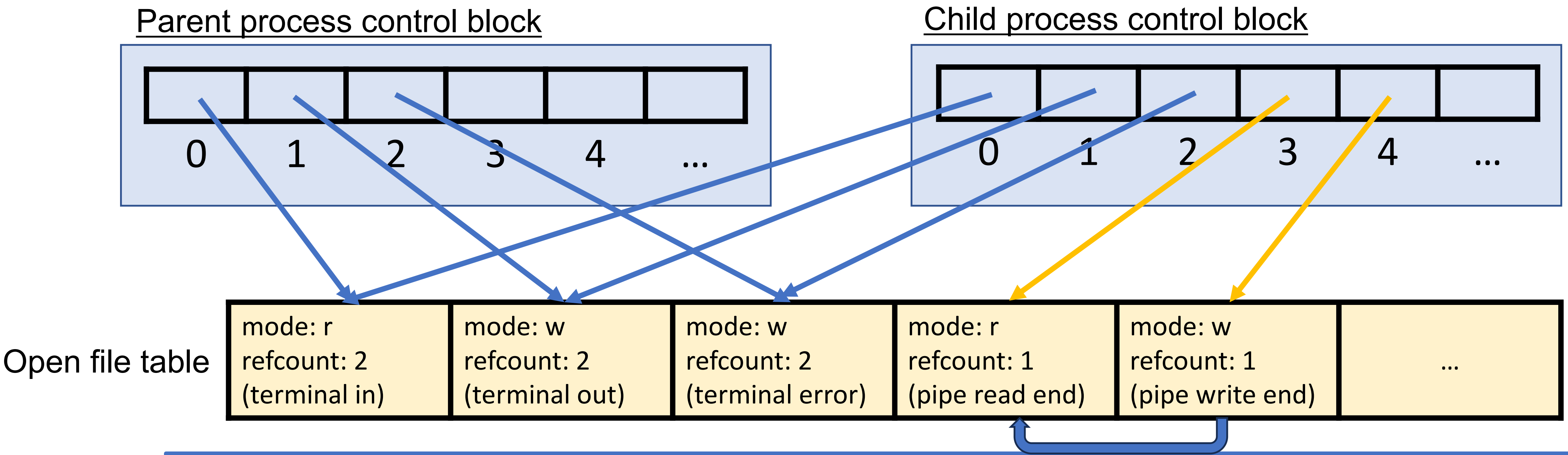

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



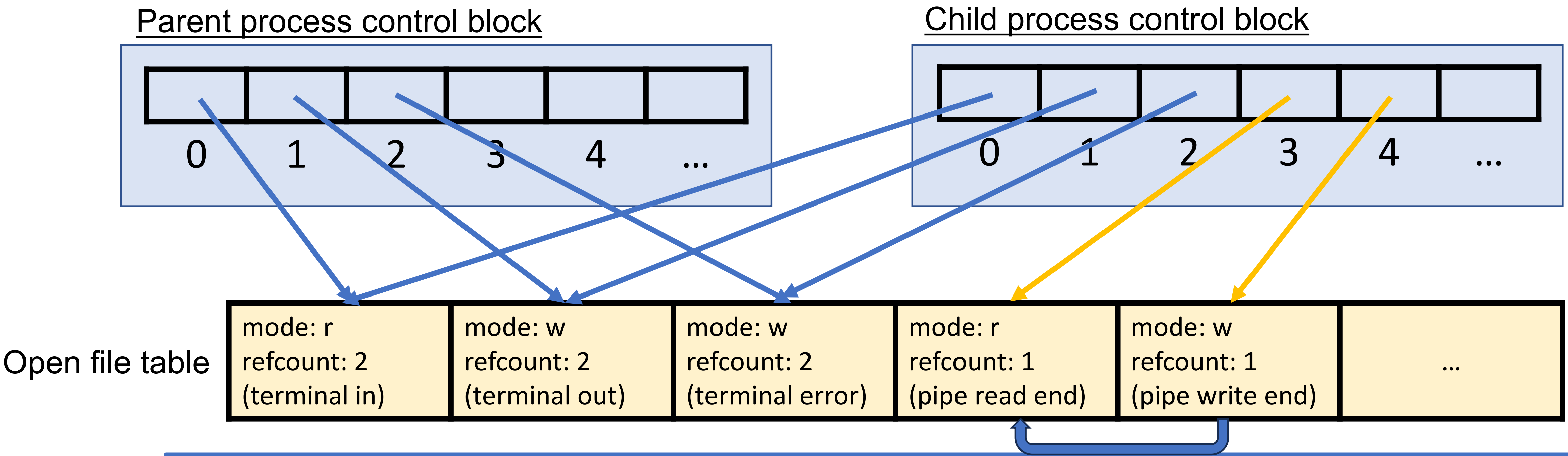

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



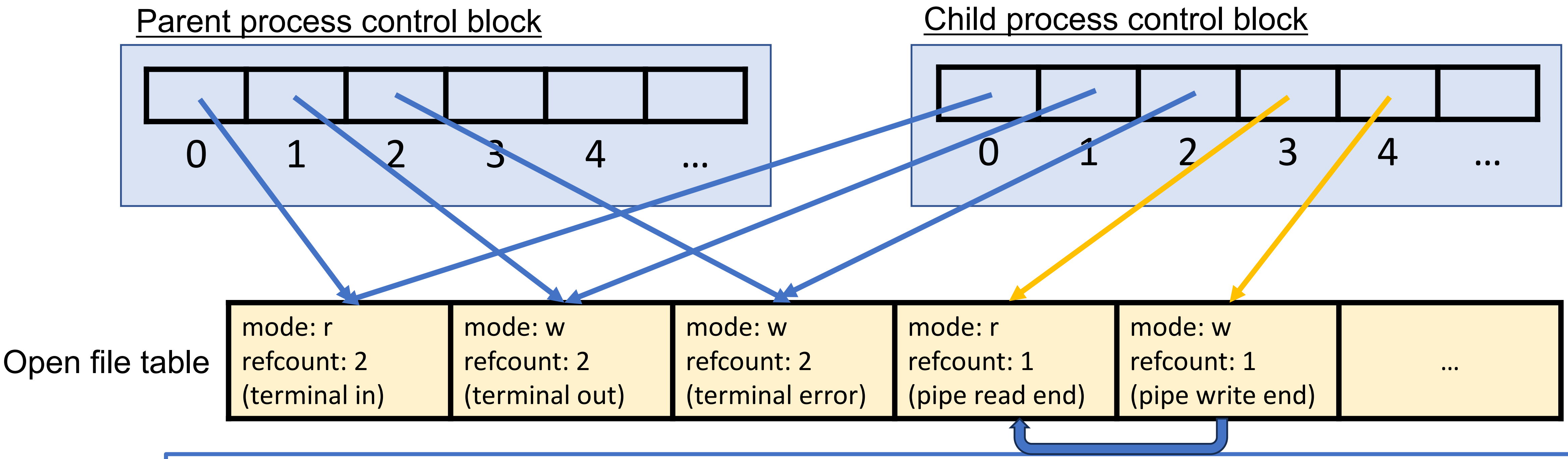

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



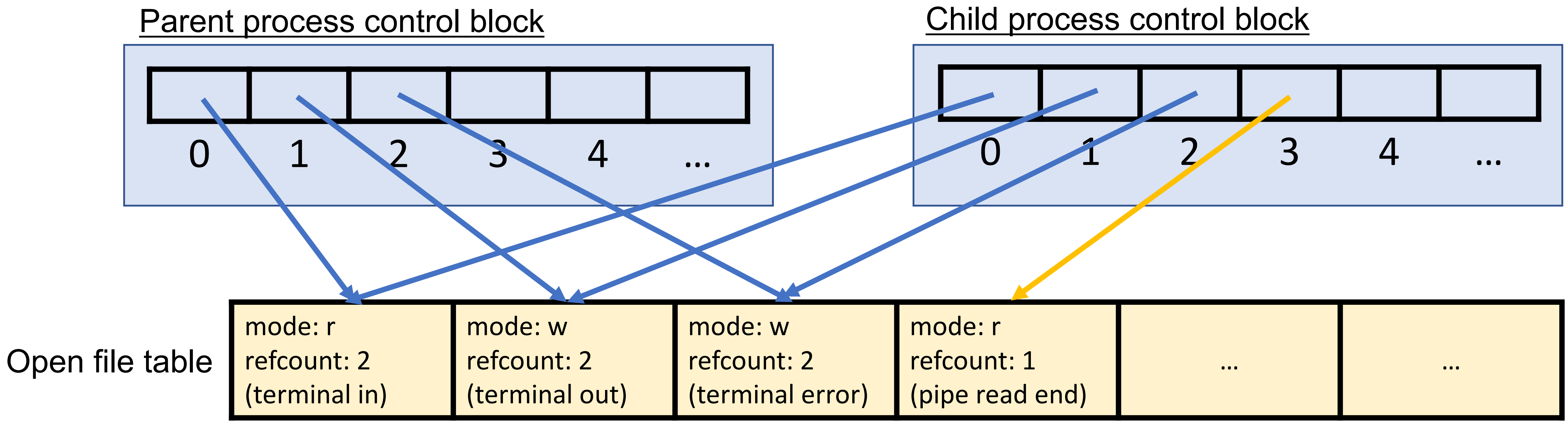

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



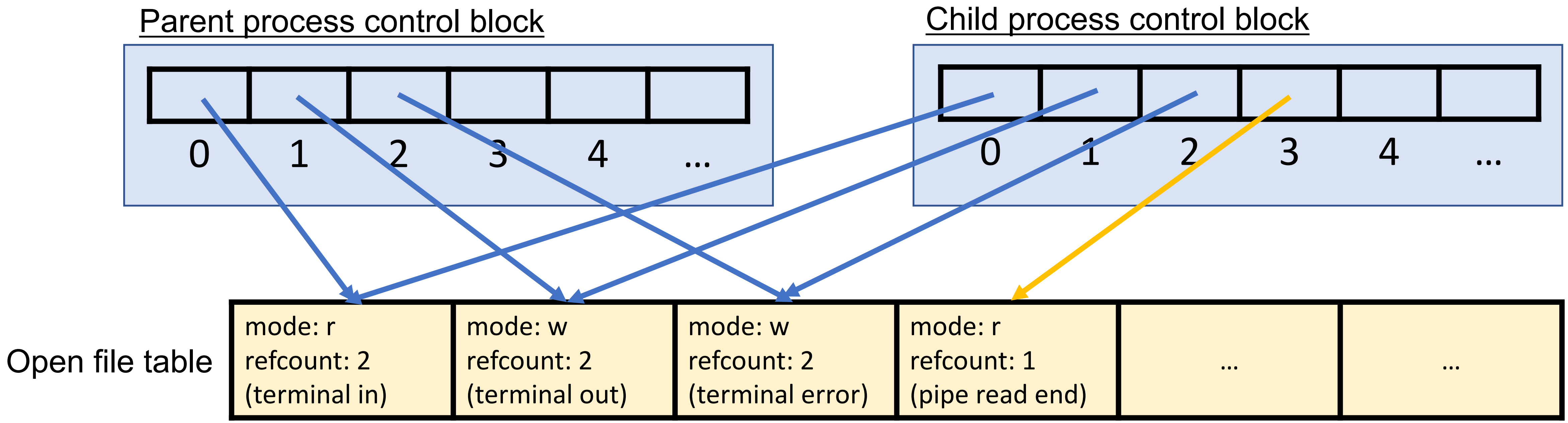

```c
...
if (pidOrZero == 0) { // In the child, we only read from the pipe
    close(fds[1]);
    char buffer[bytesSent];
    read(fds[0], buffer, sizeof(buffer));
    close(fds[0]);
    printf("Message from parent: %s\n", buffer);
    return 0;
}  ...
```

Demo: Parent Child Pipe



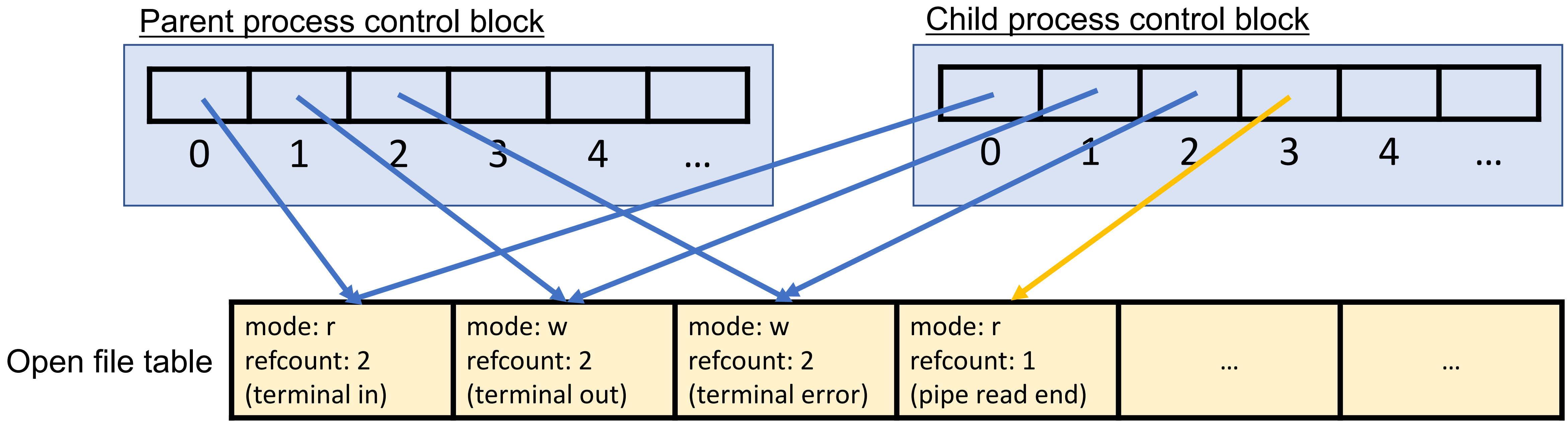

```c
...
if (pidOrZero == 0) { // In the child, we only read from the pipe
    close(fds[1]);
    char buffer[bytesSent];
    read(fds[0], buffer, sizeof(buffer));
    close(fds[0]);
    printf("Message from parent: %s\n", buffer);
    return 0;
}  ...
```

Demo: Parent Child Pipe



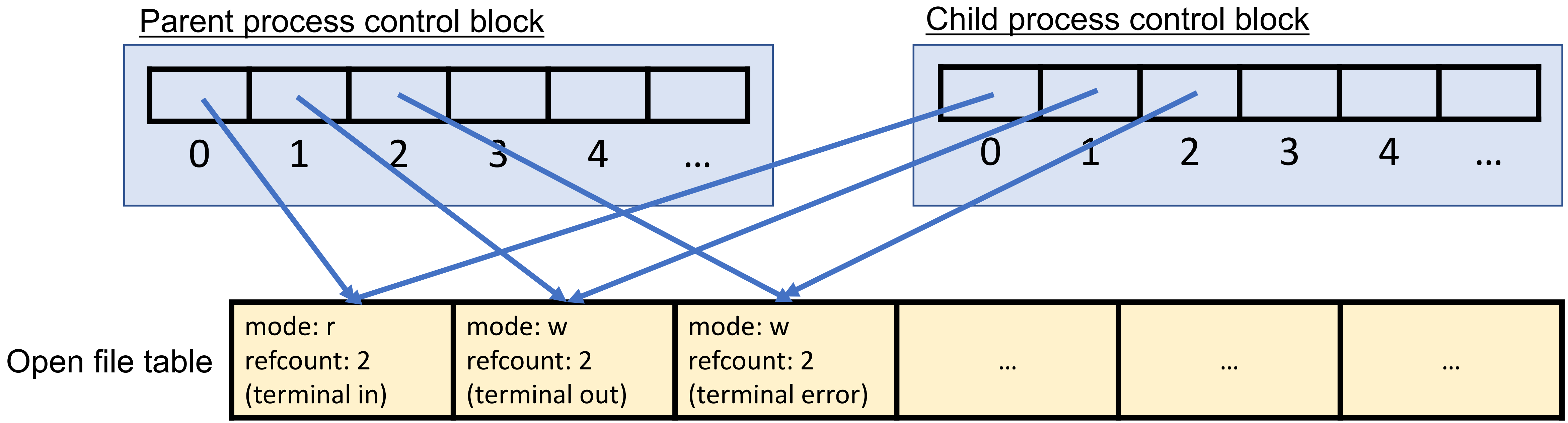

```c
...
if (pidOrZero == 0) { // In the child, we only read from the pipe
    close(fds[1]);
    char buffer[bytesSent];
    read(fds[0], buffer, sizeof(buffer));
    close(fds[0]);
    printf("Message from parent: %s\n", buffer);
    return 0;
}  ...
```

Demo: Parent Child Pipe




```
...
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;
}  ...
```

Demo: Parent Child Pipe




```c
...
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;
}  ...
```

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) { // 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;
}
```

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 the pipe write end is closed everywhere).

Recap

- <u>Recap</u>: **fork, waitpid, execvp** and our first shell
- **Shell Feature Demo:** pipes
- **pipe()** system call
- **Example:** Parent-child pipe

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.

Next time: how to connect pipes to STDIN/STDOUT, and how to implement file redirection

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