CS111, Lecture 11

Pipes, Continued

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** | **Lecture 10 / Today**

assign3: implement your own shell!

Learning Goals

- Implement an example of inter-process communication using **pipe** and **dup2**
- Explore why a pipe is shared when we call fork()

Plan For Today

- **<u>Recap</u>**: Pipes and **dup2** so far
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files
- Closing pipes
- Why are pipes shared when we call **fork**?

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

Plan For Today

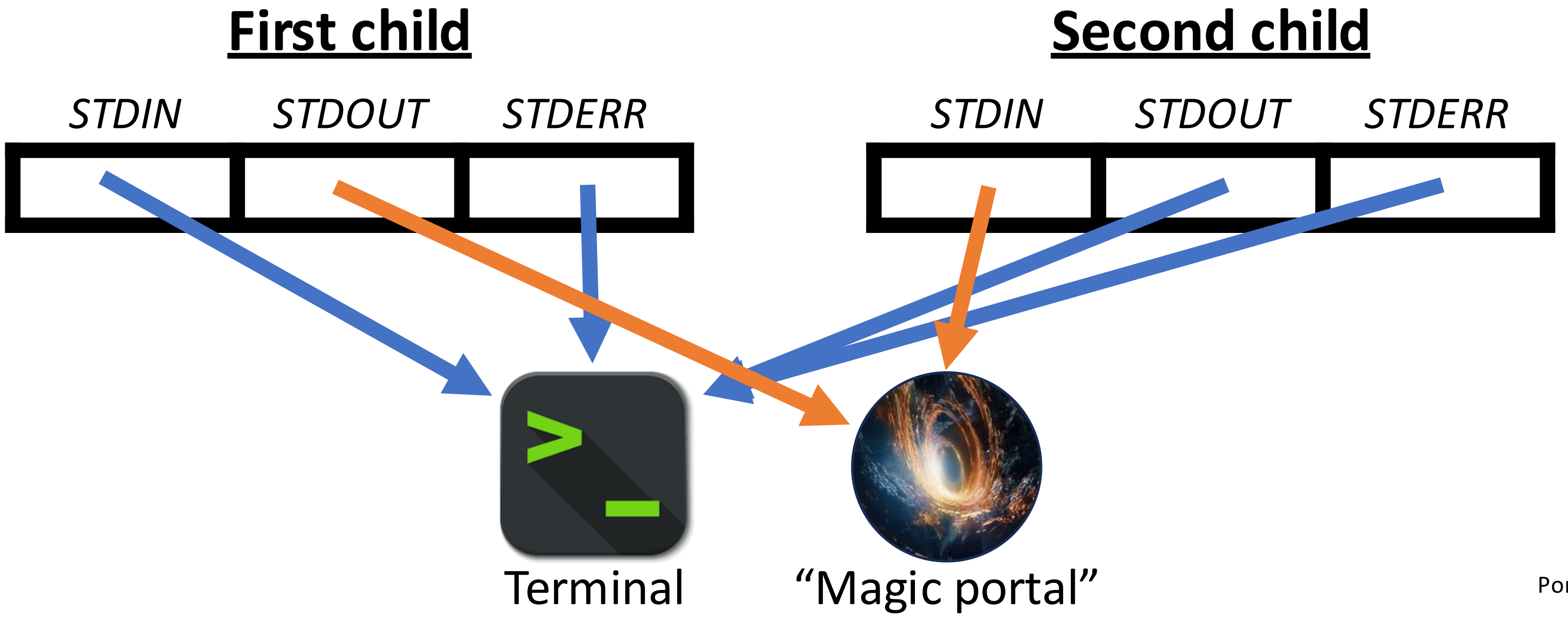
- **<u>Recap</u>: Pipes and dup2 so far**
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files
- Closing pipes
- Why are pipes shared when we call **fork**?

```
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. 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?

   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 populates the 2-element array **fds** with two file descriptors, where everything written to **fds[1]** can be read from **fds[0]**. (“*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!

We might expect the child gets a copy of the pipe – however, as we'll see today, the child gets access to the *same* file descriptor sessions at the time **fork** is called (sort of).

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;
}
```

Both the parent *and* the child must close the pipe FDs when they are done with them.

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

Redirecting Process I/O

dup2 copies over file descriptor information from one file descriptor number (srcfd) to another (dstfd). Both now refer to same thing (e.g. reading from one advances read position of the other).

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

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

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

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

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

A Secret About execvp

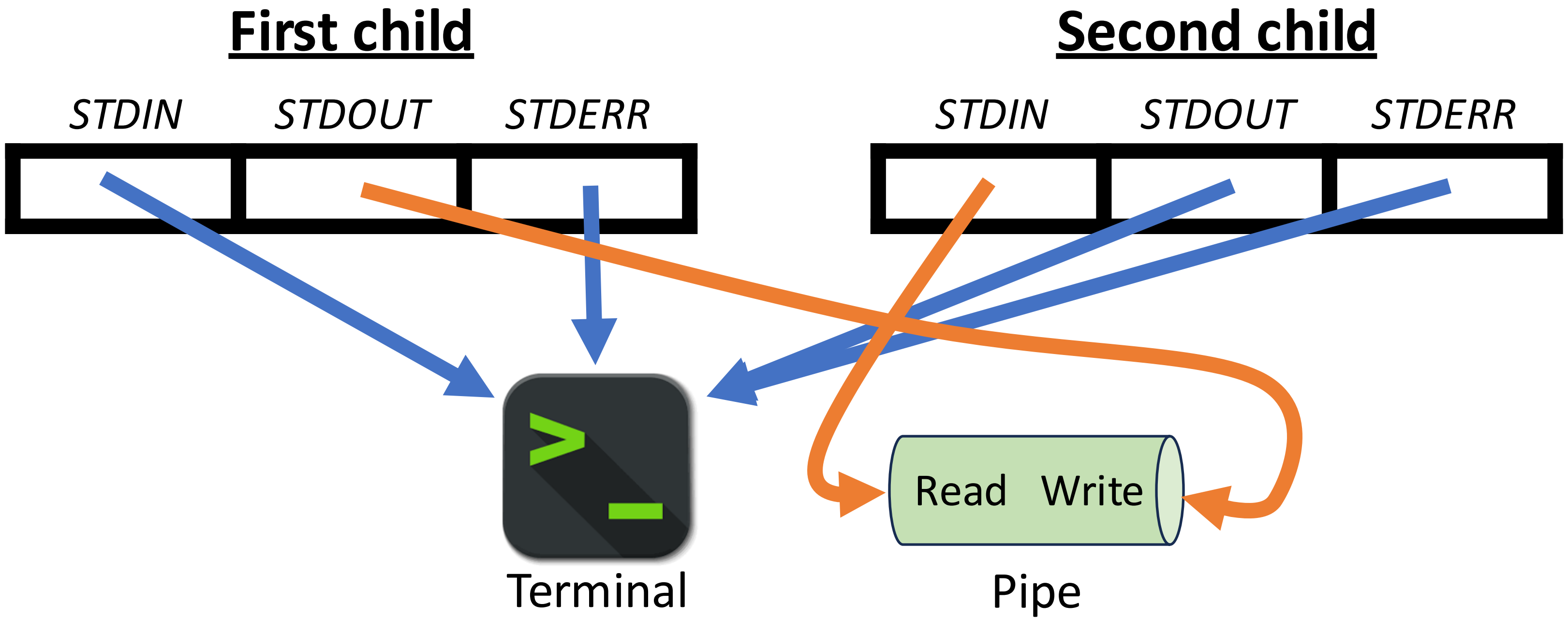
Problem: if we spawn a child and rewire its STDOUT to point to a pipe, won't everything get wiped anyway when we call **execvp**?

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

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.



Plan For Today

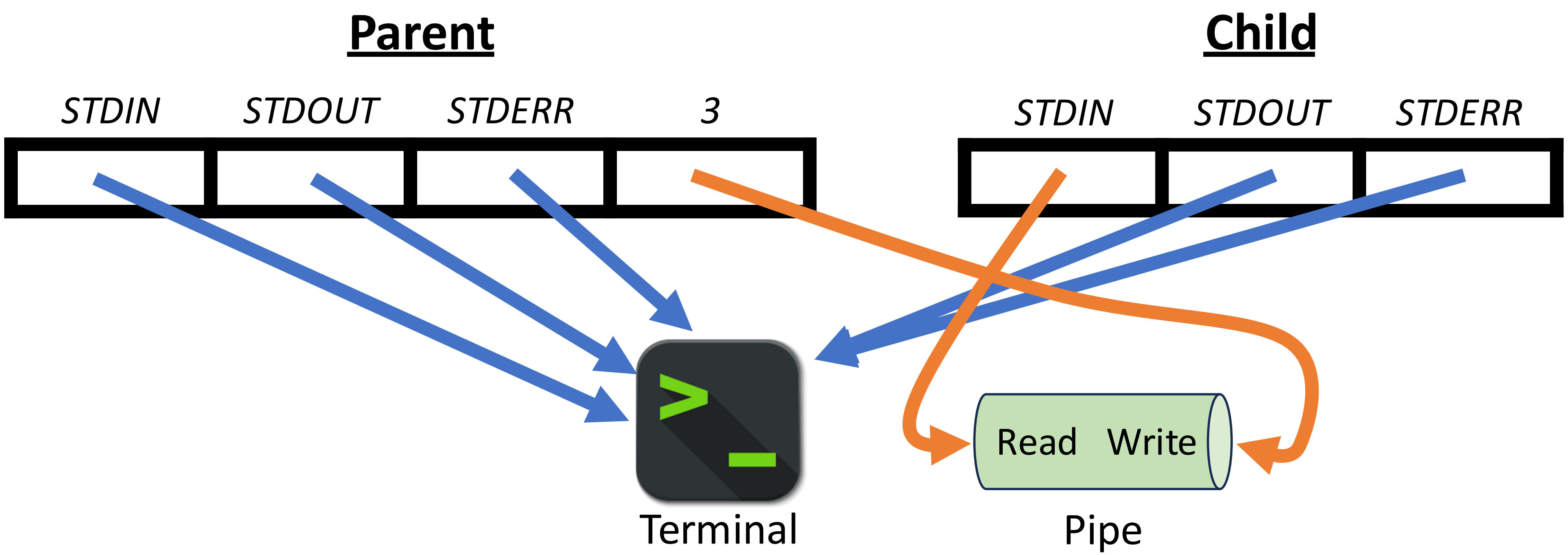
- Recap: Pipes and **dup2** so far
- ***Practice:* implementing subprocess**
- I/O Redirection with files
- Closing pipes
- Why are pipes shared when we call **fork**?

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

Practice: Subprocess

Let's implement the **subprocess** function, which spawns a child and connects a pipe such that the parent can write to the child's STDIN. (Slightly different from pipeline end-goal, where 2+ children are connected via pipes).

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

Let’s implement the **subprocess** function, which spawns a child and connects a pipe such that the parent can write to the child’s STDIN.

```
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

```
typedef struct subprocess_t {
    pid_t pid;
    int supplyfd;
} subprocess_t;
```

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

    // We want to feed these lines as input to grep to print only sunny days
    const char * recent_weather[] = { "Sunny 72", "Rainy 55", "Cloudy 62",
      "Sunny 80", "Sunny 75", "Cloudy 61", "Sunny 68", "Rainy 60", "Sunny 85"
    };
    size_t nelems = sizeof(recent_weather) / sizeof(recent_weather[0]);

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

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

Demo: subprocess

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
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);
        exit(1);
    }
    ...
```

Wait – there's no **read** call here! Where do we read from the pipe? *Key idea: the execvp'ed program will read from the pipe when it reads from its STDIN!*

subprocess

```
subprocess_t subprocess(const char *command) {
    ...
    close(fds[0]);

    // Package up PID and pipe write end to return together in a struct
    subprocess_t returnStruct;
    returnStruct.pid = pidOrZero;
    returnStruct.supplyfd = fds[1];
    return returnStruct;
}
```

```
typedef struct subprocess_t {
    pid_t pid;
    int supplyfd;
} subprocess_t;
```

Wait – why don't we call **waitpid** here? *Key idea: the caller will be sending data to the child, so child needs to keep running after subprocess finishes. It's caller's responsibility to waitpid for the child.*

subprocess

```c
typedef struct subprocess_t {
    pid_t pid;
    int supplyfd;
} subprocess_t;
```

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

    // We want to feed these lines as input to grep to print only sunny days
    const char * recent_weather[] = { "Sunny 72", "Rainy 55", "Cloudy 62",
      "Sunny 80", "Sunny 75", "Cloudy 61", "Sunny 68", "Rainy 60", "Sunny 85"
    };
    size_t nelems = sizeof(recent_weather) / sizeof(recent_weather[0]);

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

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

Plan For Today

- **Recap**: Pipes and **dup2** so far
- ***Practice:*** implementing **subprocess**
- **I/O Redirection with files**
- Closing pipes
- Why are pipes shared when we call **fork**?

```
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
```

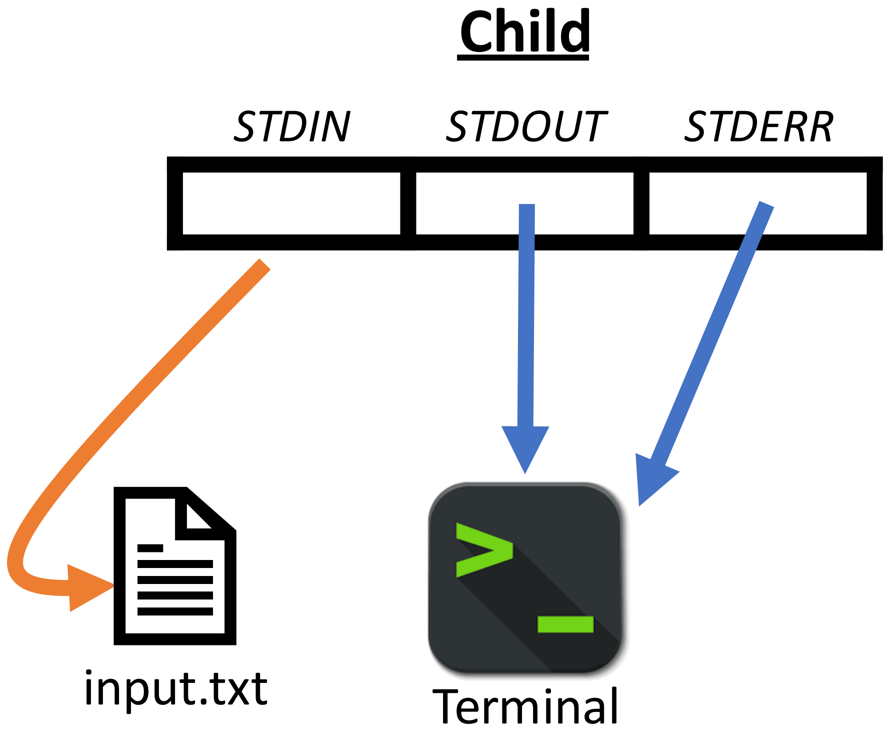
This reads input from a file instead of reading from the terminal

```
sort < input.txt
```

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

Redirecting Process I/O to/from a File

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

You're encouraged to unify code across milestones where possible!

Plan For Today

- **Recap**: Pipes and **dup2** so far
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files
- **Closing pipes**
- Why are pipes shared when we call **fork**?

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

Pipe Stalling

Not closing write ends of pipes can cause functionality issues. If a process calls **read** and there's nothing more to read, but the write end is still open, it will block until it gets more input!

- 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
- E.g. if the child reads until there's nothing left, but the write end was not closed everywhere, it will stall.

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;
}
```

Parent Doesn't Send Message ☹ (still finishes)

```
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); // program will still terminate
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Ex: Child reads, parent doesn't write or close

```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;
}
```

child stuck here!

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;
```

child stuck here!

Plan For Today

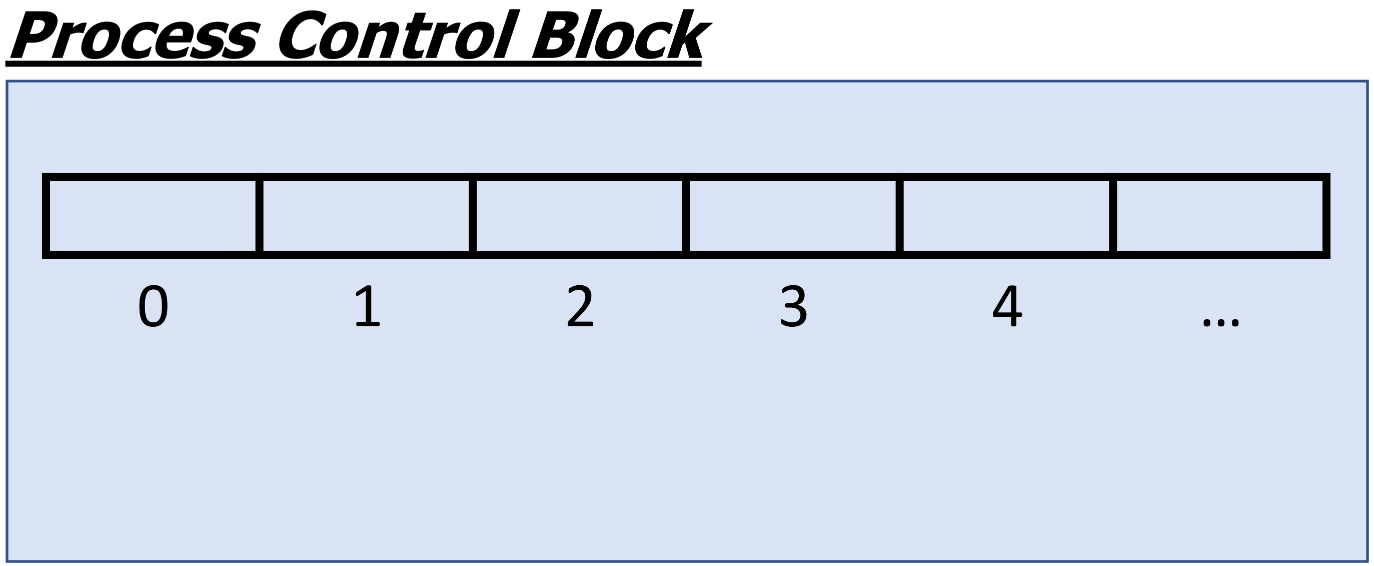
- **Recap**: Pipes and **dup2** so far
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files
- Closing pipes
- **Why are pipes shared when we call fork?**

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

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.

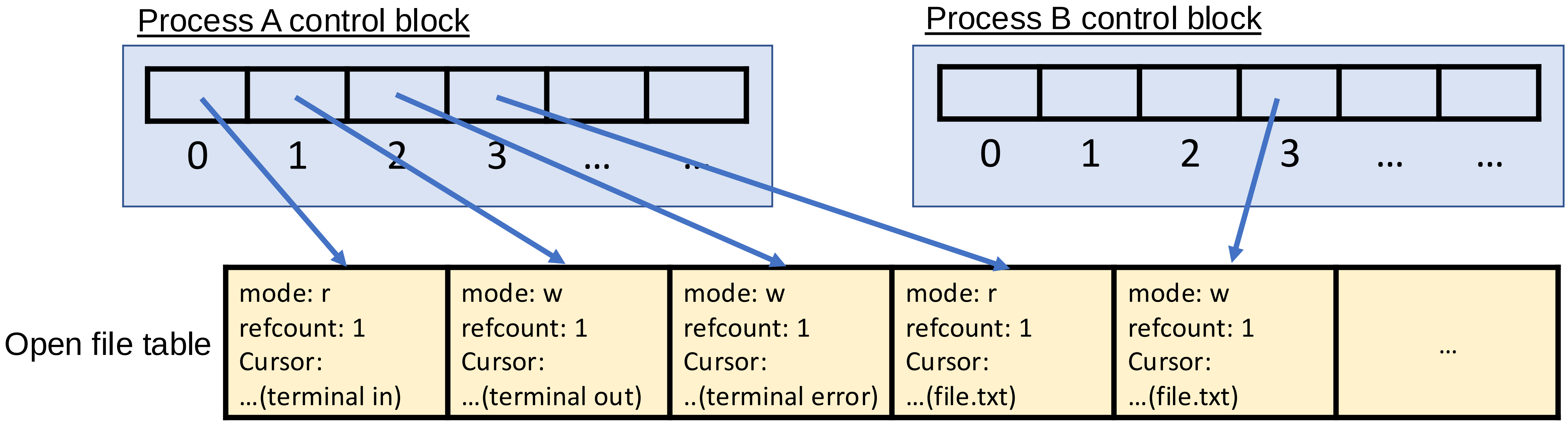
Key idea: a file descriptor is **an index into that process's file descriptor table!**



File Descriptor Table

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

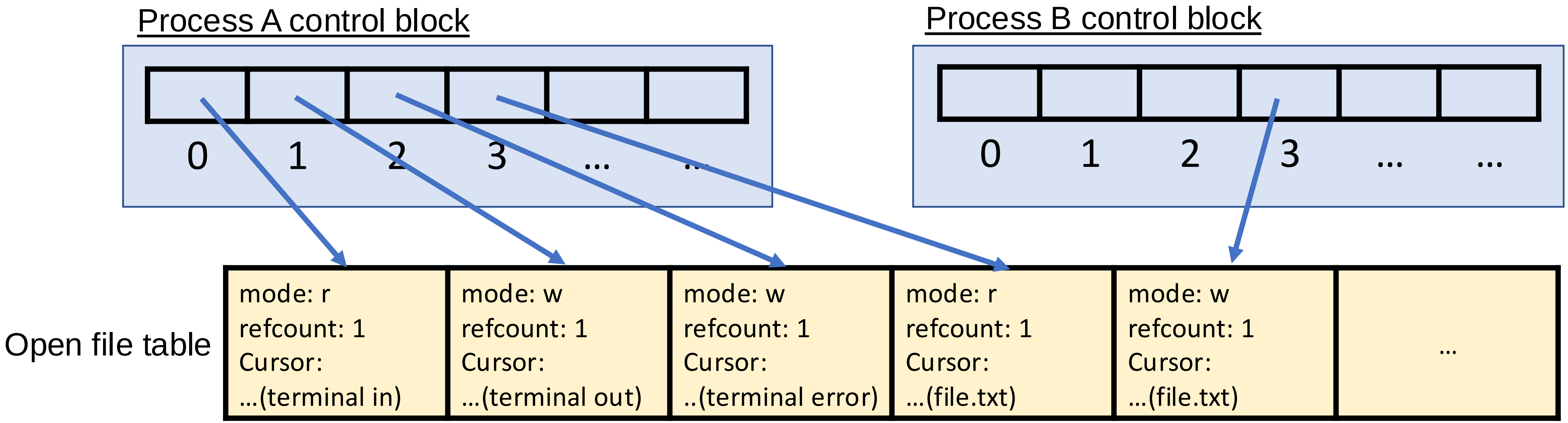
- An entry in a file descriptor table is really a *pointer* to an entry in another global table, the **open file table**.
- The **open file table** is one array of information about open files/resources across all processes. There's one open file table entry per *session* (not per *file).*



File Descriptor Table

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

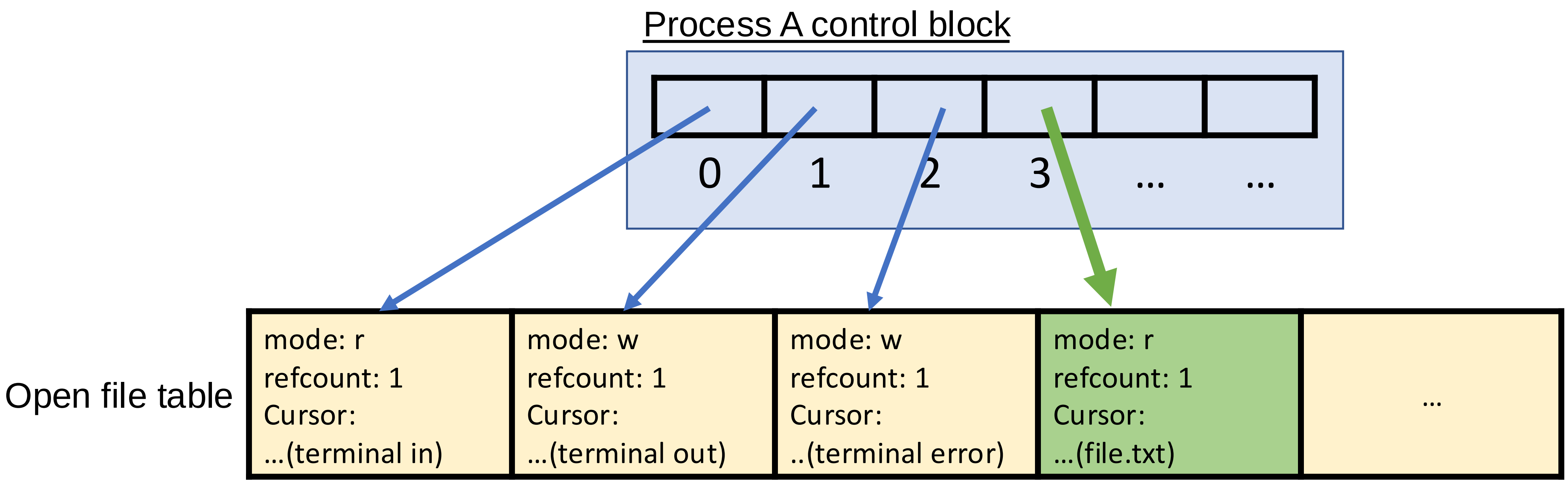
- **mode:** e.g., read, write, read+write
- **Reference count:** the number of file descriptor table entries pointing to it
- **Cursor**: tracking where in the file it currently is



Open File Table

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

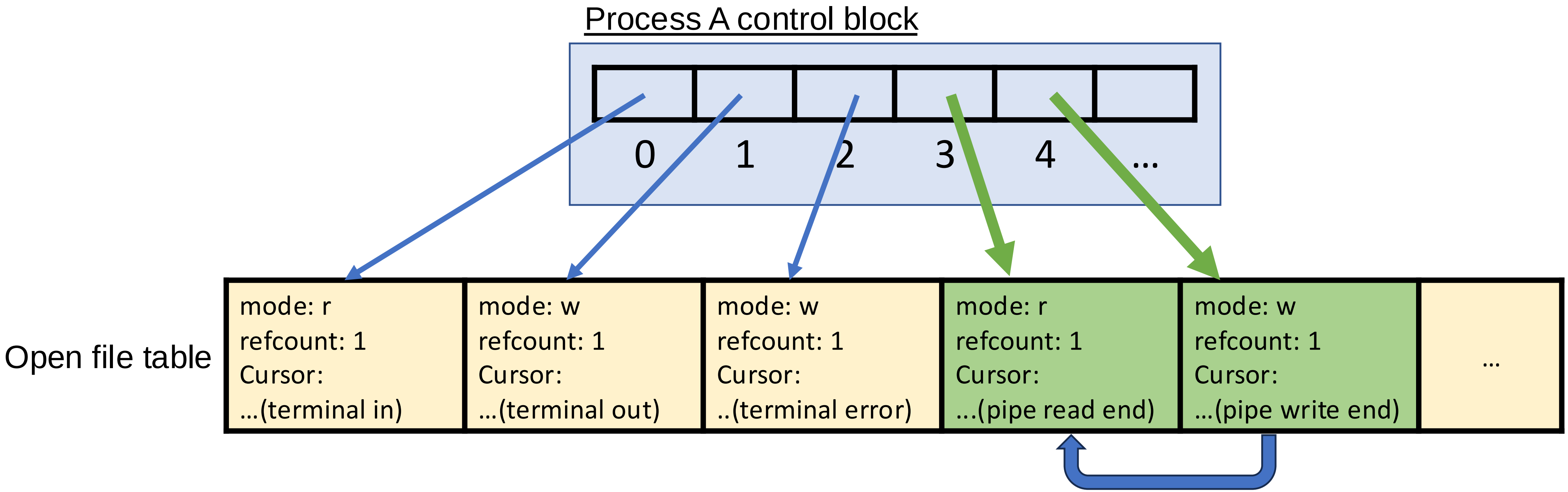
```
int fd = open("file.txt", O_RDONLY); // 3
```



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.

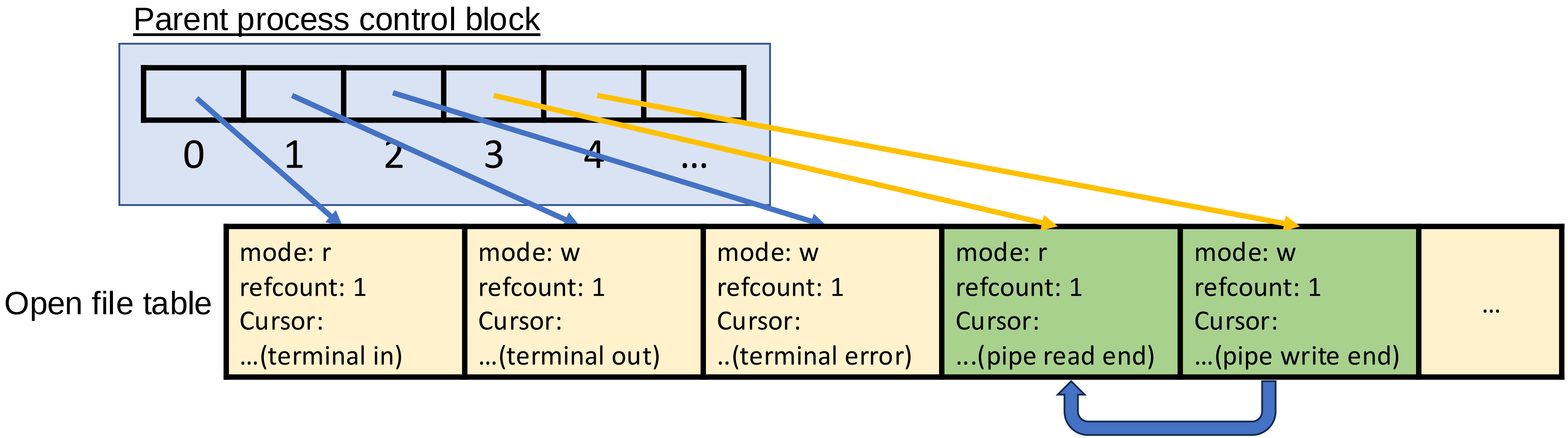
```
int fds[2];
pipe(fds); // afterwards, fds[0] = 3, fds[1] = 4
```



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!

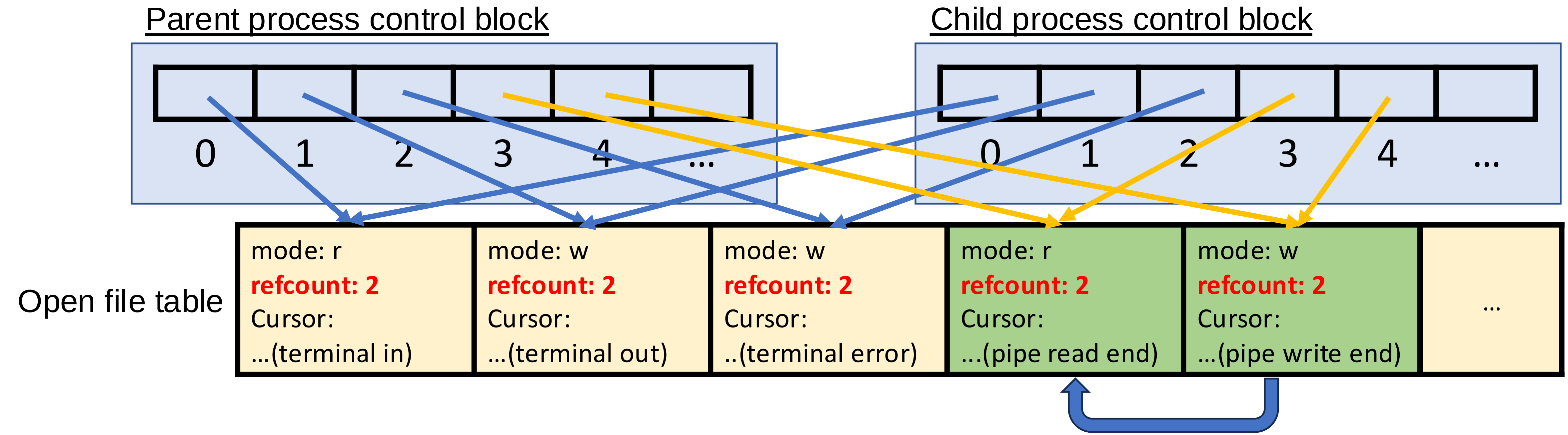
```
int fds[2];
pipe(fds); // afterwards, fds[0] = 3, fds[1] = 4
pid_t pidOrZero = fork();
```



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();
```



Key Idea: on **fork**, the child process gets "shallow copies" of all parent file descriptors. This is how a parent and child can share the same pipe even though it's "copied" on fork.

Reference Count

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

Key Idea: parent-child duplicated file descriptors must be closed in both the parent *and* child because both parent and child are referencing them.

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(ies)?

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

Respond on PollEv:
pollev.com/cs111

What will the reference counts be in each case?

a) 2, b) 2

0%

a) 2, b) 1

0%

a) 1, b) 2

0%

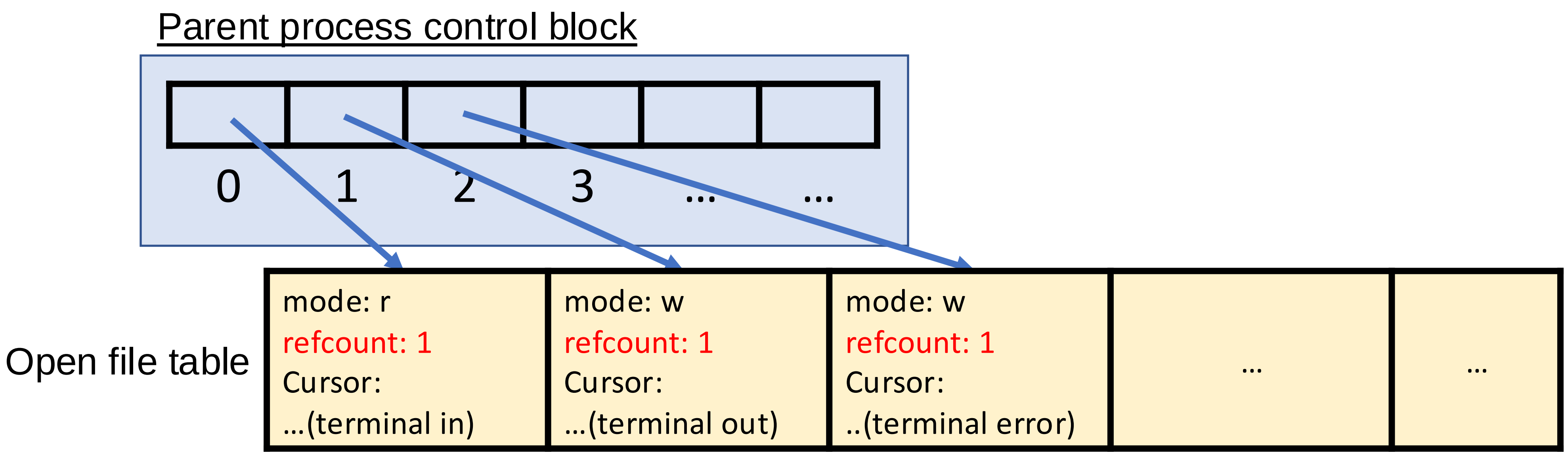
a) 1, b) 1

0%

Practice: Reference Count

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(ies)?

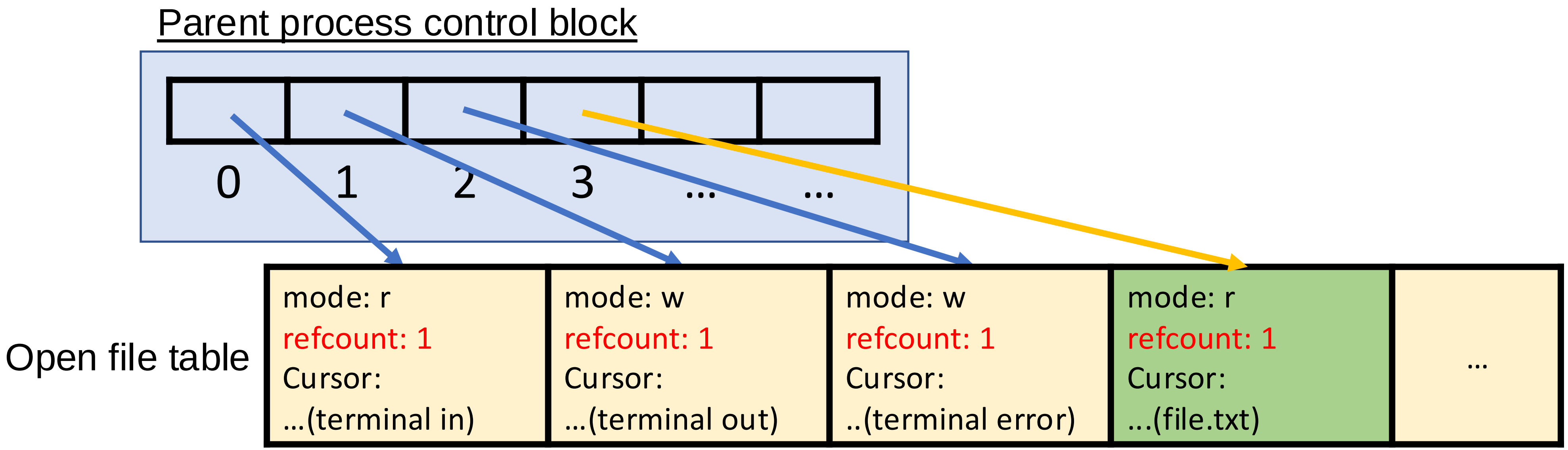
```
int fd = open("file.txt", O_RDONLY); // fd = 3 here
pid_t pidOrZero = fork();
```



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(ies)?

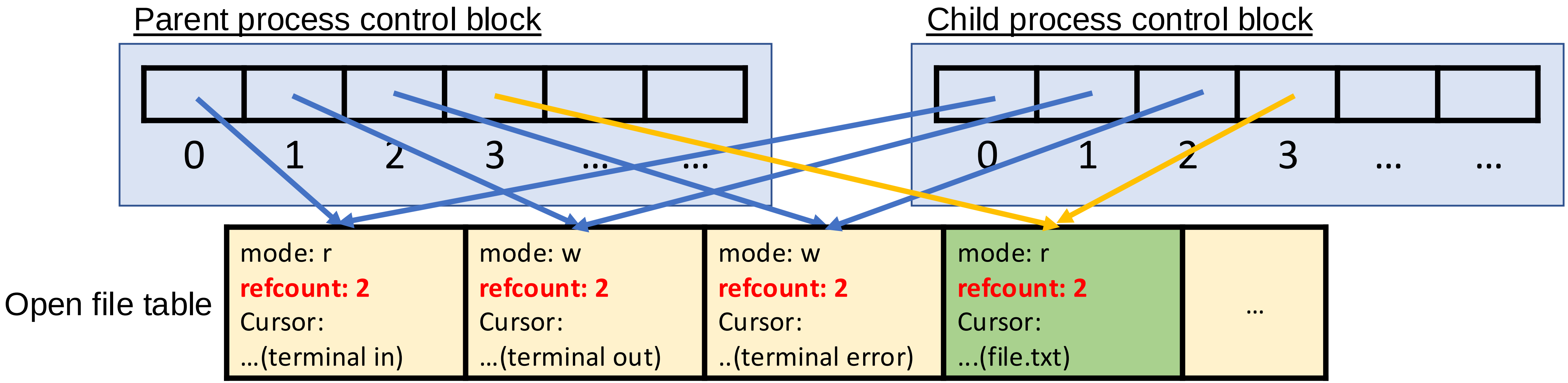
```
int fd = open("file.txt", O_RDONLY); // fd = 3 here
pid_t pidOrZero = fork();
```



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(ies)?

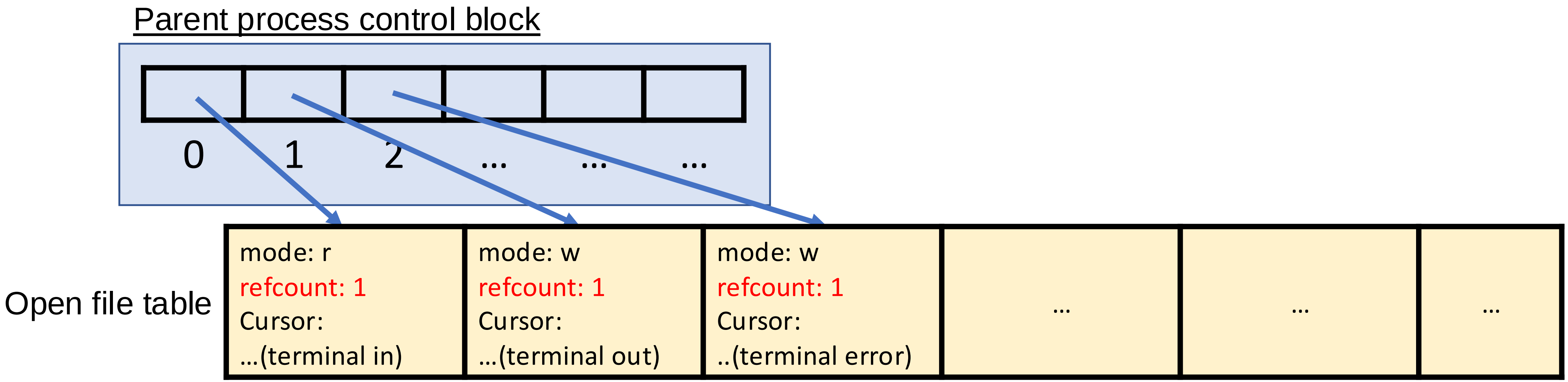
```
int fd = open("file.txt", O_RDONLY); // fd = 3 here
pid_t pidOrZero = fork();
```



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(ies)?

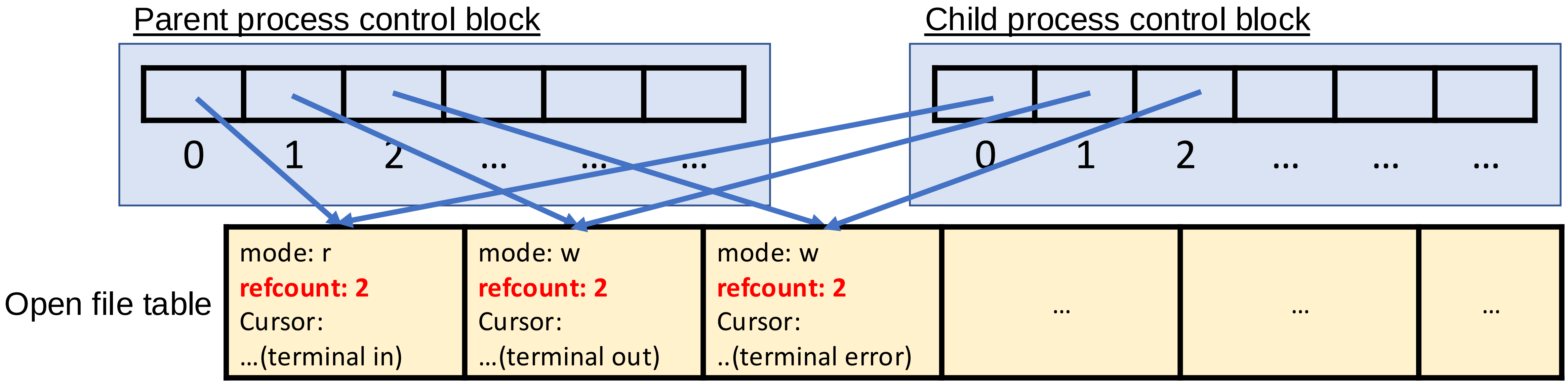
```
pid_t pidOrZero = fork();
int fd = open("file.txt", O_RDONLY); // fd = 3 here
```



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(ies)?

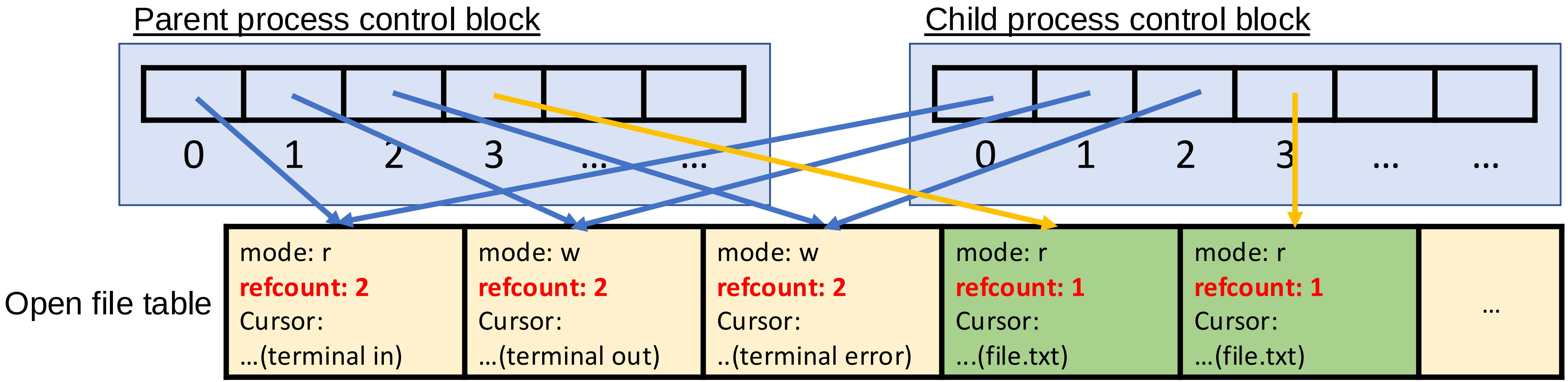
```
pid_t pidOrZero = fork();
int fd = open("file.txt", O_RDONLY); // fd = 3 here
```



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(ies)?

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



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(ies)? **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);
}
```

Practice: Reference Count

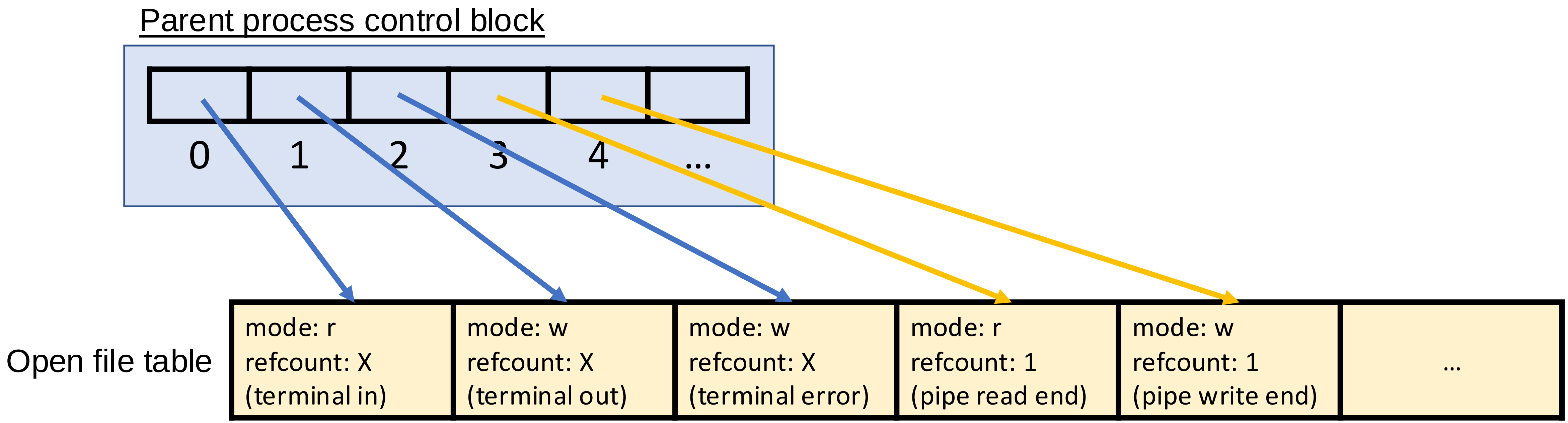
a) If a process opens a file, and then spawns a child process, what will the reference count be for the corresponding open file table entry(ies)? **2**.

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

Both together explain why, to share a pipe, we must create it prior to fork().

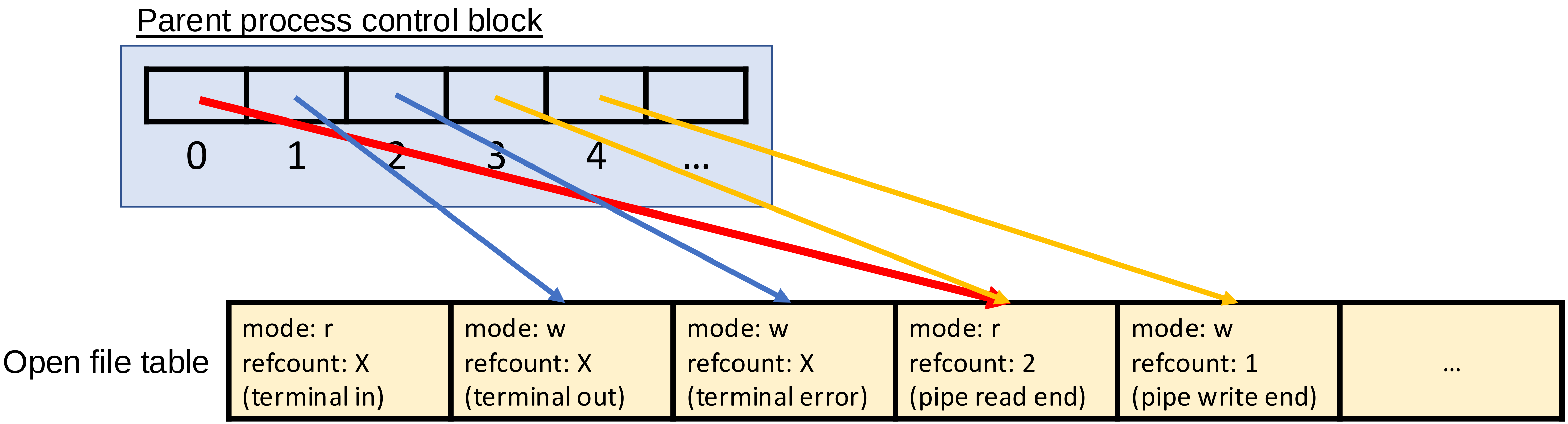
```
pid_t pidOrZero = fork();
int fds[2];
pipe(fds);  // uh oh - parent and child have separate pipes!
```

dup2 and Open File Table



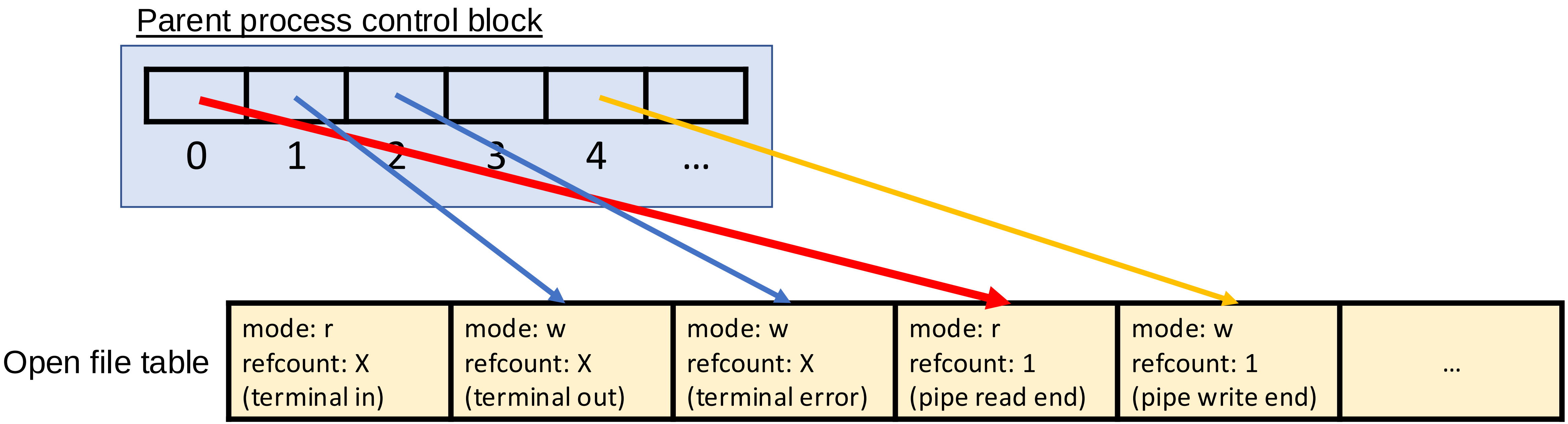

```
int fds[2];
pipe(fds); // assume fds[0] is 3 and fds[1] is 4
dup2(fds[0], STDIN_FILENO);
close(fds[0]);
```

dup2 and Open File Table




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

dup2 and Open File Table




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

Summary: File Desciptors / Open File Table

- Per-process **file descriptor table** + global **open file table**. Entries in file descriptor tables point to entries in the open file table.
- One open file table entry for each session (e.g. every **open** call), with refcount.
- If a pipe is created and then we call fork, 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.

Recap

- **<u>Recap</u>**: Pipes and **dup2** so far
- ***Practice:*** implementing **subprocess**
- I/O Redirection with files
- Closing pipes
- Why are pipes shared when we call **fork**?

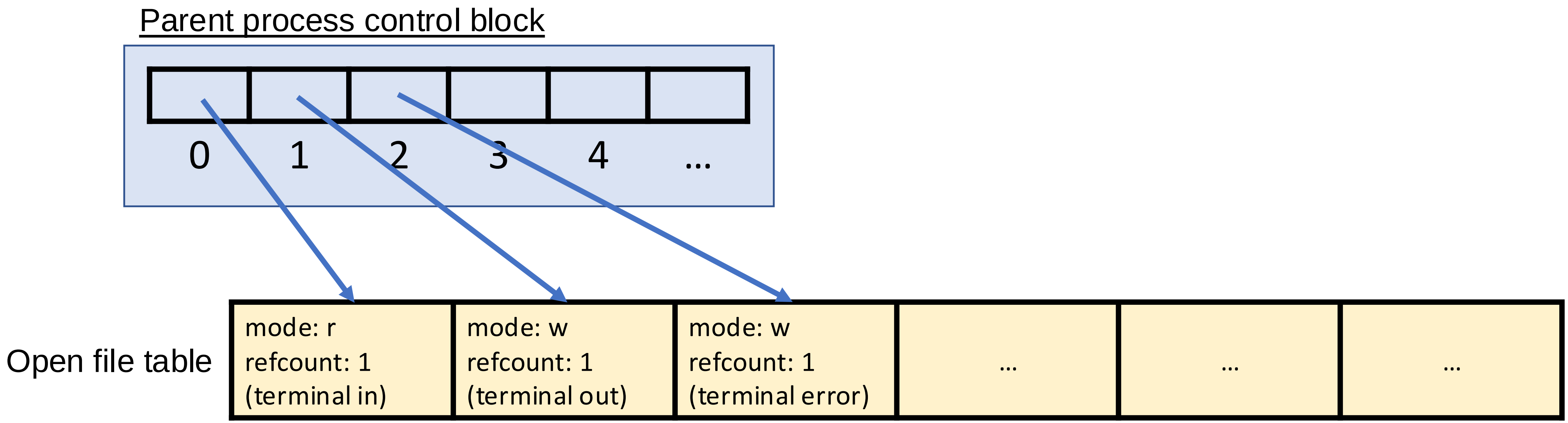
Next time: introduction to multithreading

Lecture 11 takeaway: We can share pipes with child processes and change FDs 0-2 to connect processes and redirect their I/O. File descriptors are shared on fork because the file descriptor table, which is copied, contains pointers to a shared open file table, which is not copied.

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

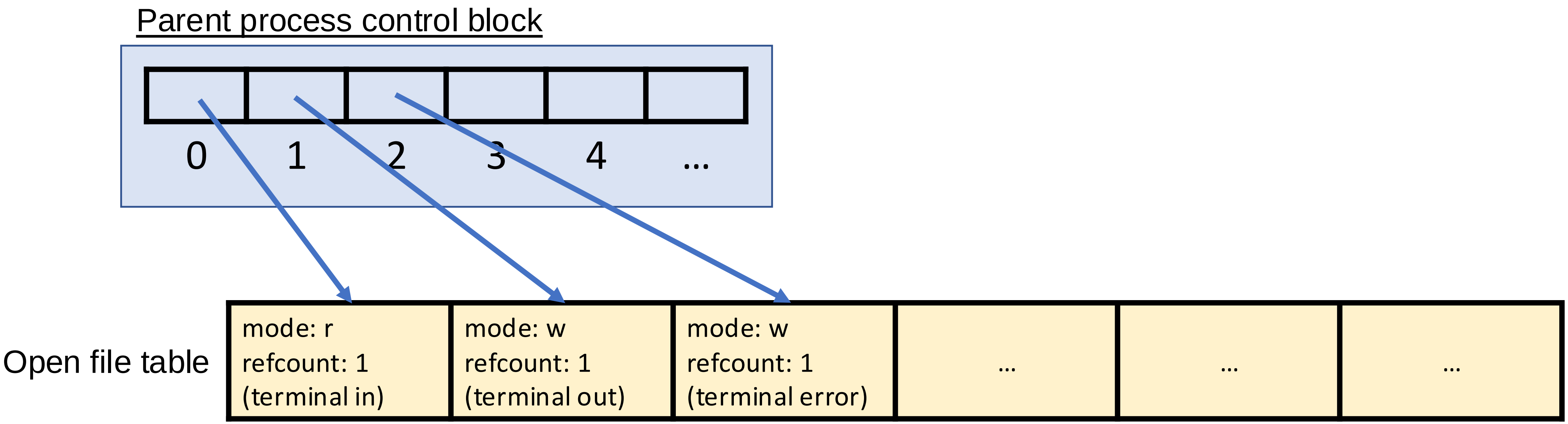
Extra Slides

Demo: Parent Child Pipe



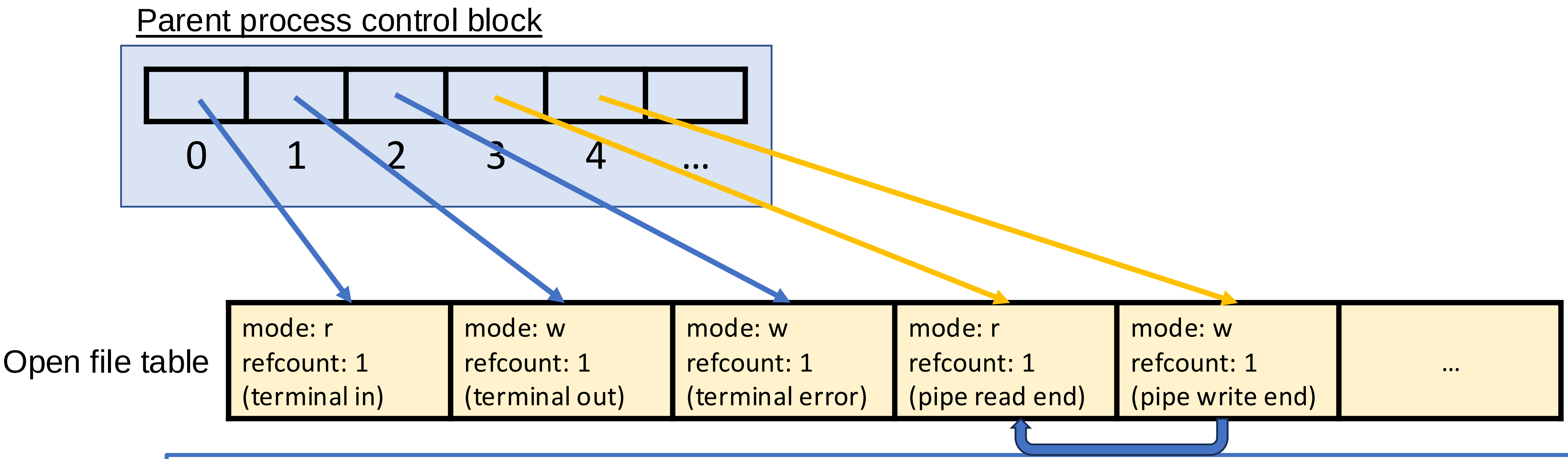

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe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

Demo: Parent Child Pipe



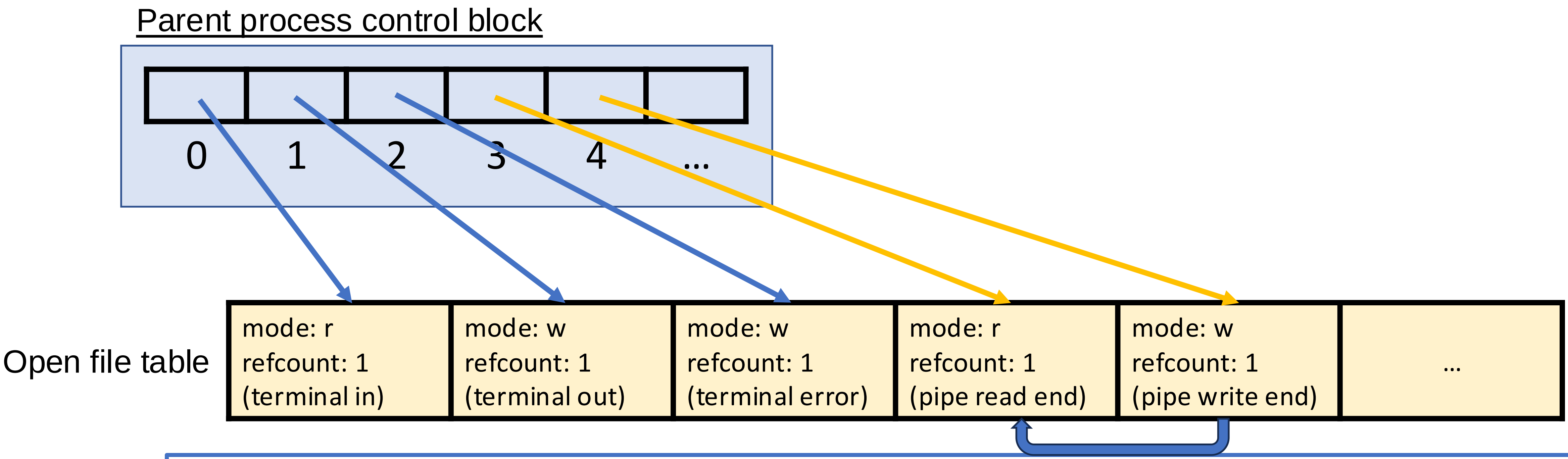

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe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

Demo: Parent Child Pipe



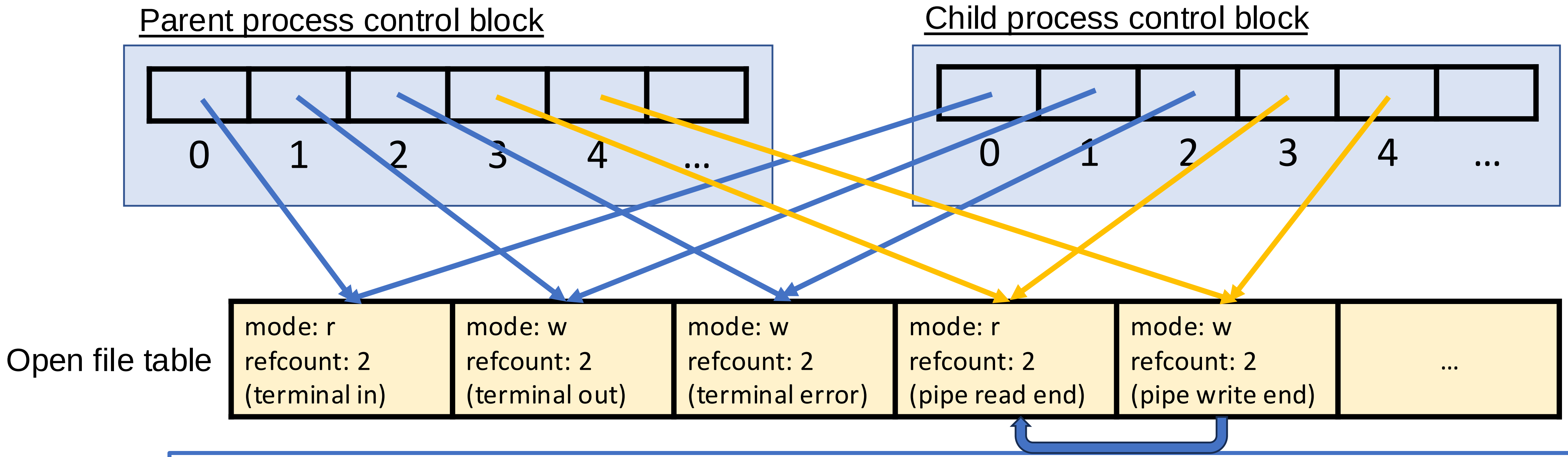

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);  // here, fds[0] = 3, fds[1] = 4
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the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

Demo: Parent Child Pipe



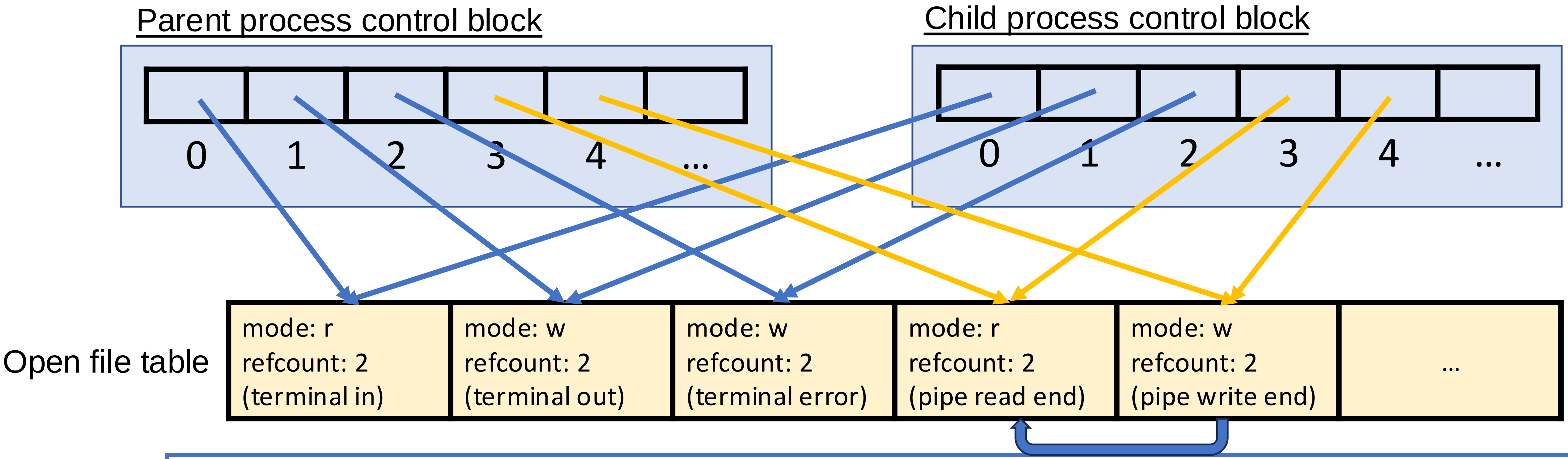

```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);  // here, fds[0] = 3, fds[1] = 4
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the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

Demo: Parent Child Pipe



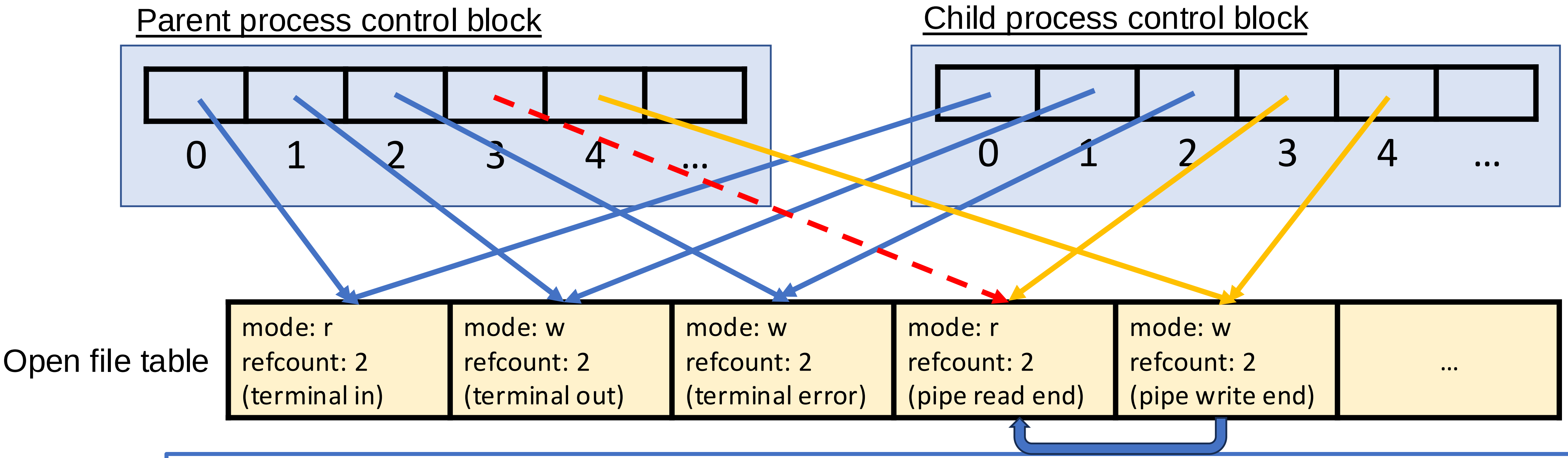

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);  // here, fds[0] = 3, fds[1] = 4
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the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

Demo: Parent Child Pipe



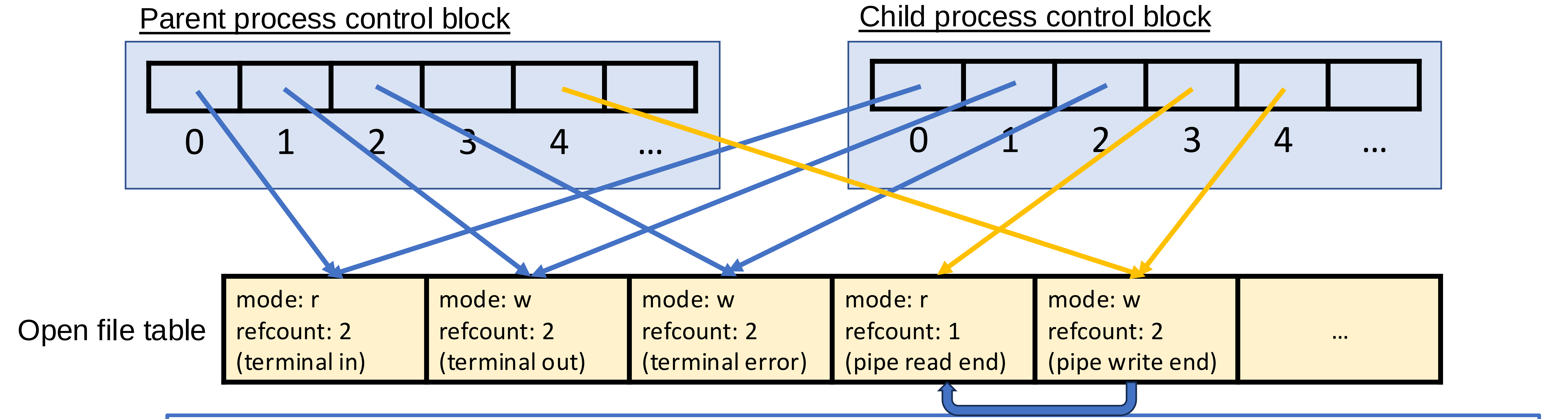

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);  // here, fds[0] = 3, fds[1] = 4
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the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

Demo: Parent Child Pipe



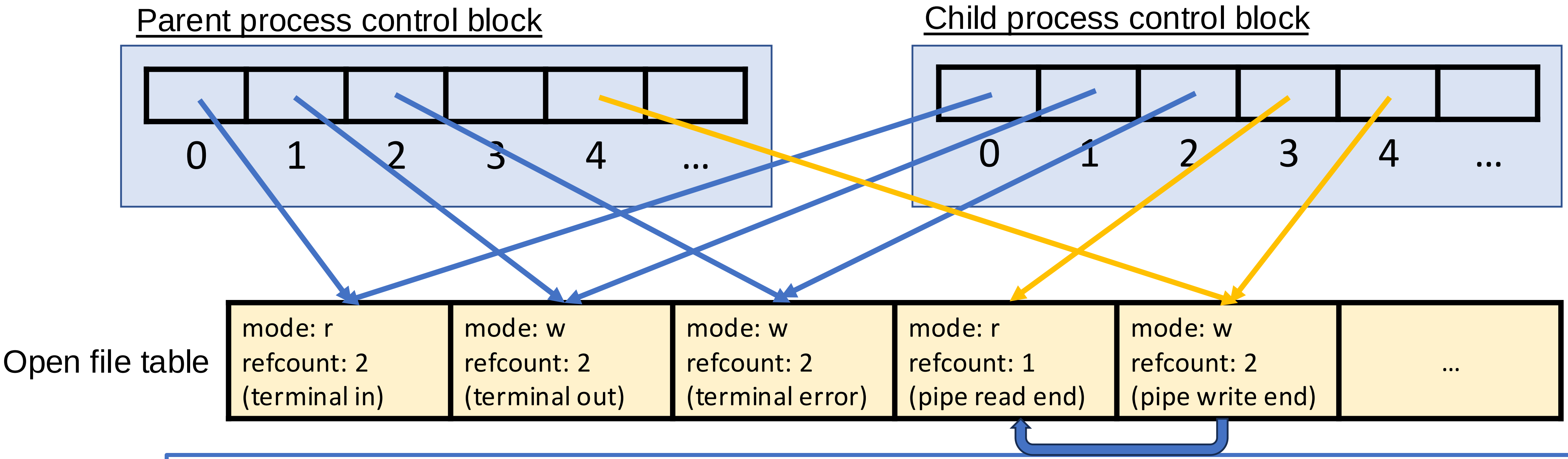

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



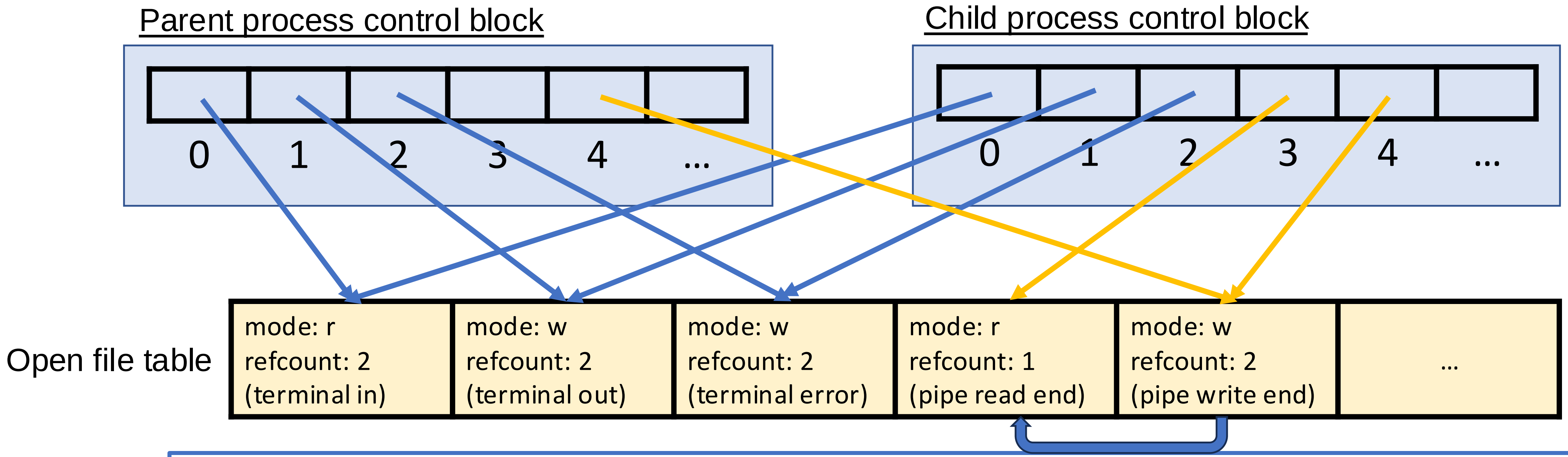

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



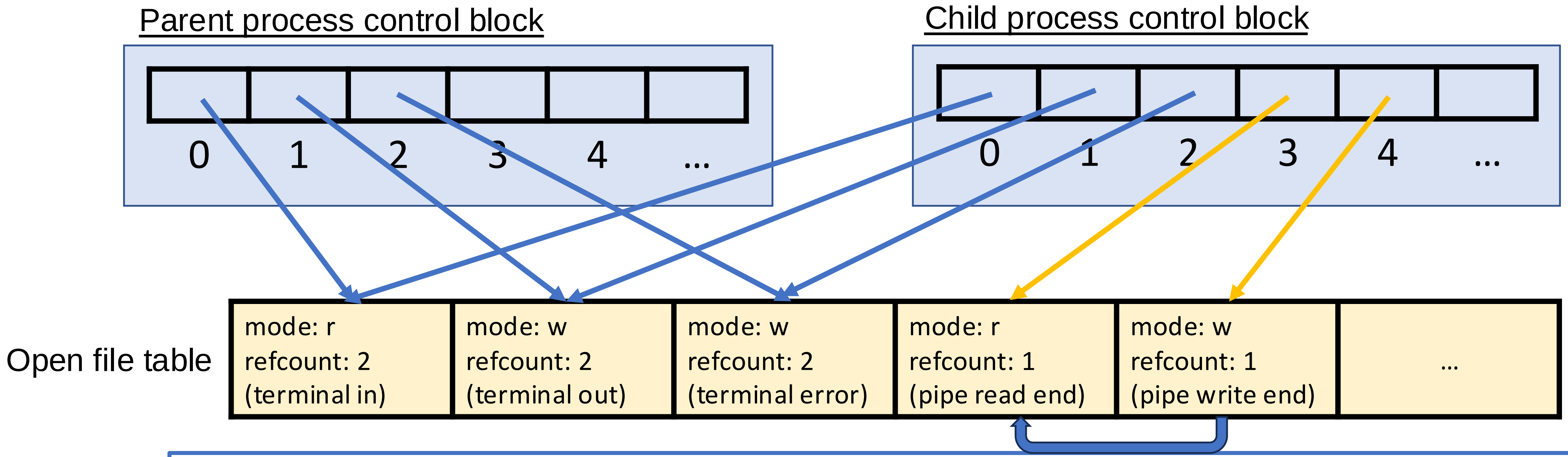

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



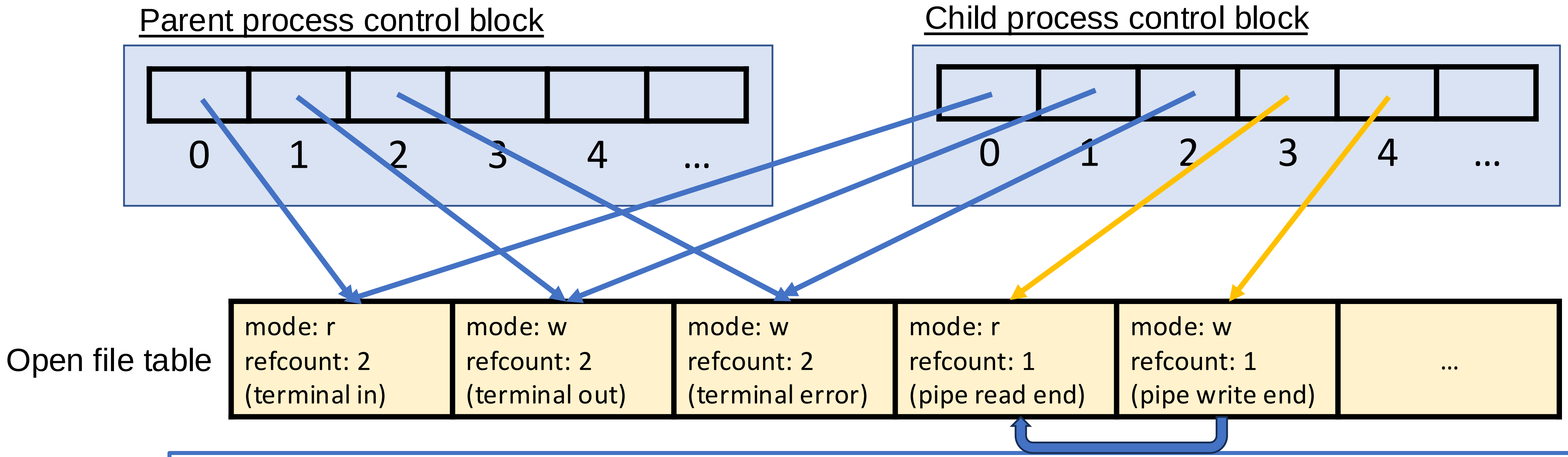

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



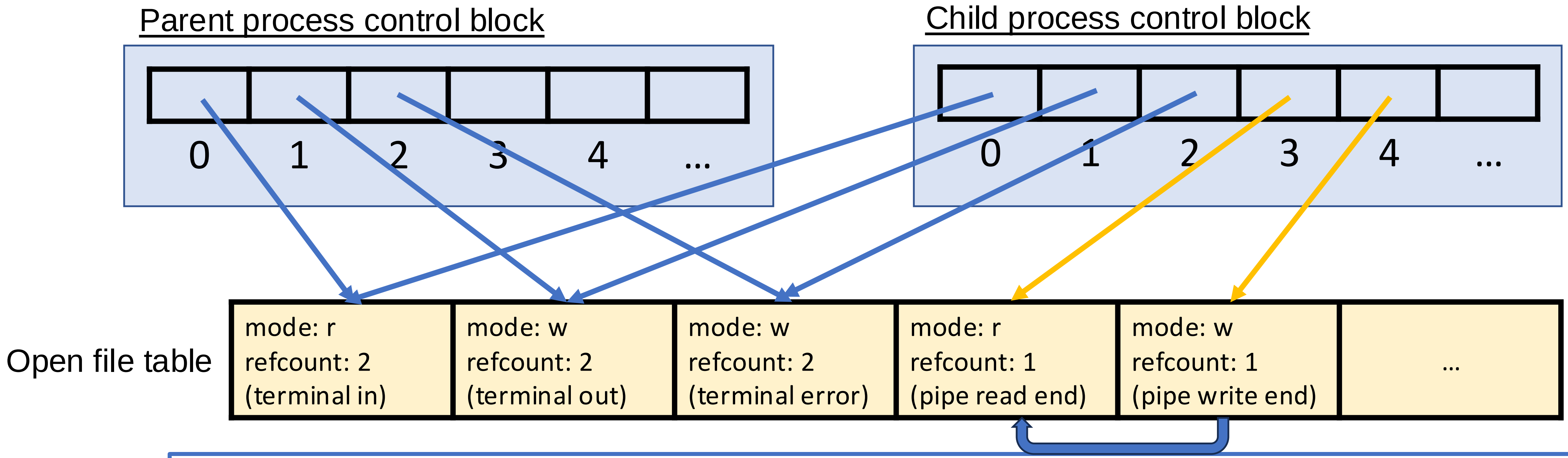

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



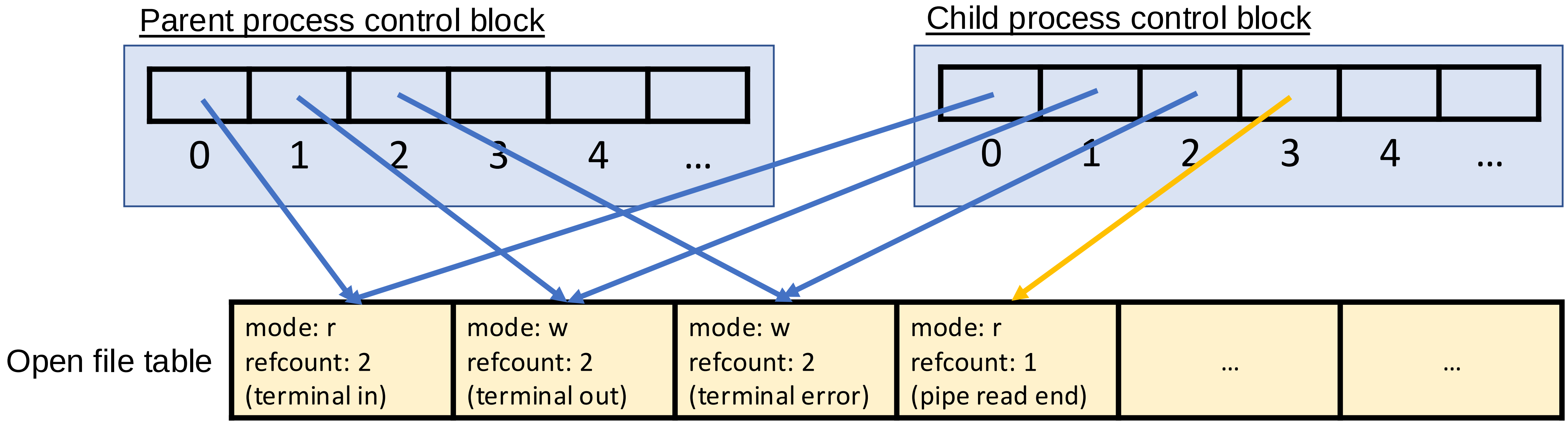

```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

Demo: Parent Child Pipe



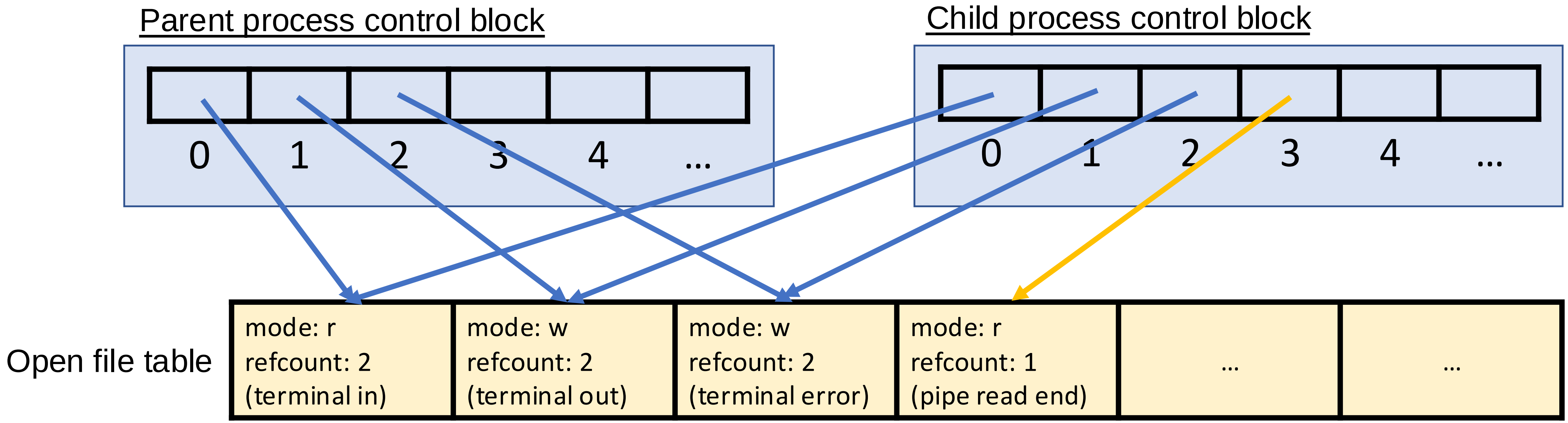

```
...
if (pidOrZero == 0) { // In the child, we only read from the pipe
    close(fds[1]);
    char buffer[bytesSent];
    read(fds[0], buffer, sizeof(buffer));
    close(fds[0]);
    printf("Message from parent: %s\n", buffer);
    return 0;
}  ...
```

Demo: Parent Child Pipe



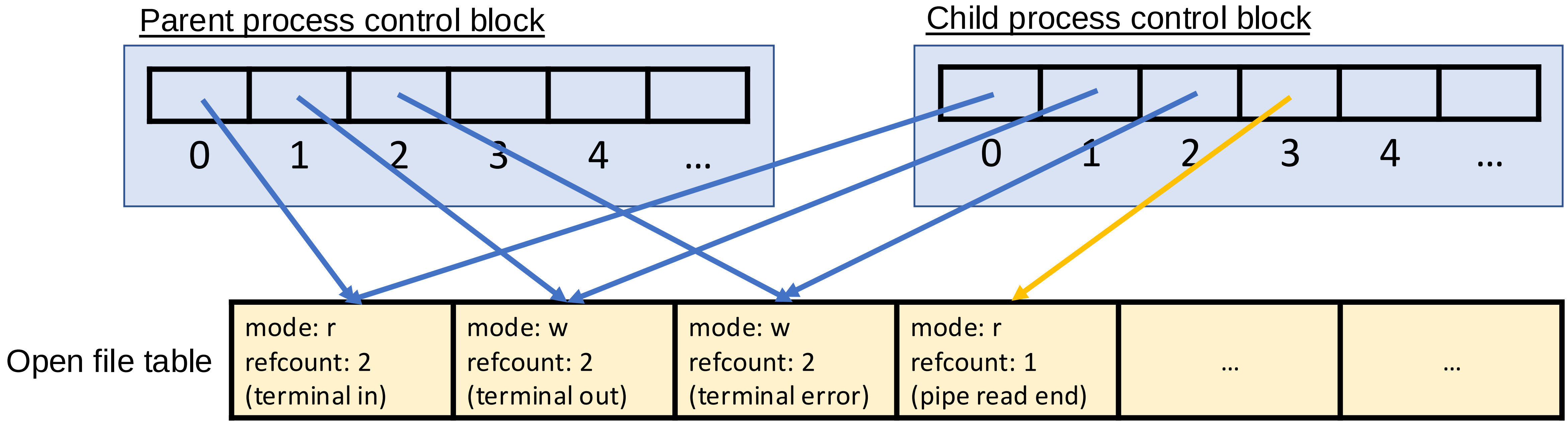

```
...
if (pidOrZero == 0) { // In the child, we only read from the pipe
    close(fds[1]);
    char buffer[bytesSent];
    read(fds[0], buffer, sizeof(buffer));
    close(fds[0]);
    printf("Message from parent: %s\n", buffer);
    return 0;
}  ...
```

Demo: Parent Child Pipe



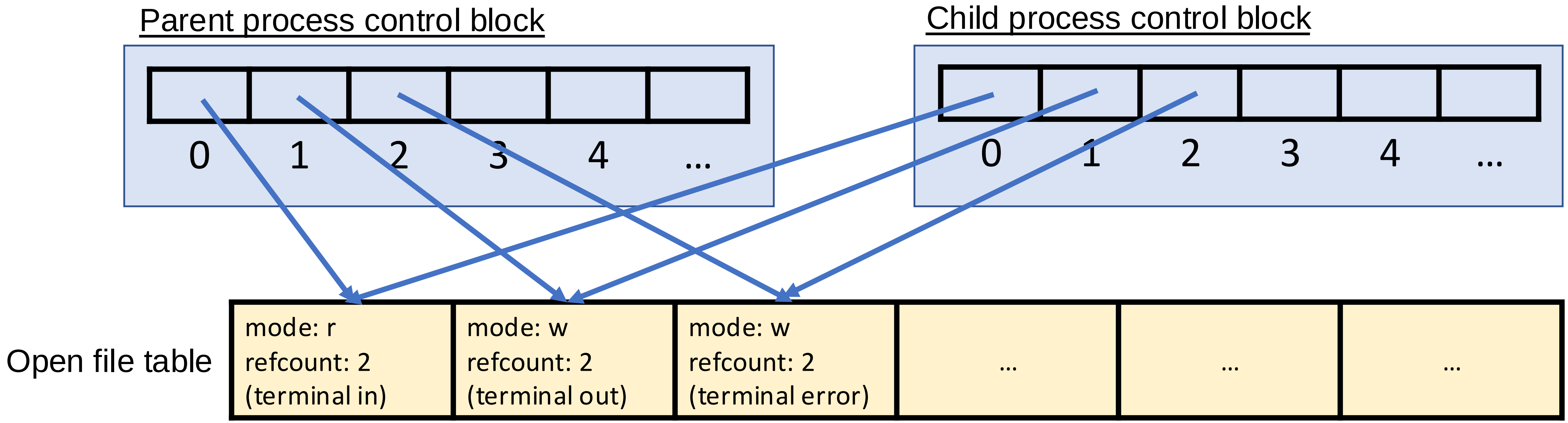

```c
...
if (pidOrZero == 0) { // In the child, we only read from the pipe
    close(fds[1]);
    char buffer[bytesSent];
    read(fds[0], buffer, sizeof(buffer));
    close(fds[0]);
    printf("Message from parent: %s\n", buffer);
    return 0;
}  ...
```

Demo: Parent Child Pipe




```
...
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 - CPlayground

https://cplayground.com/?p=hare-camel-buffalo&breakpoints=%5B11%5D