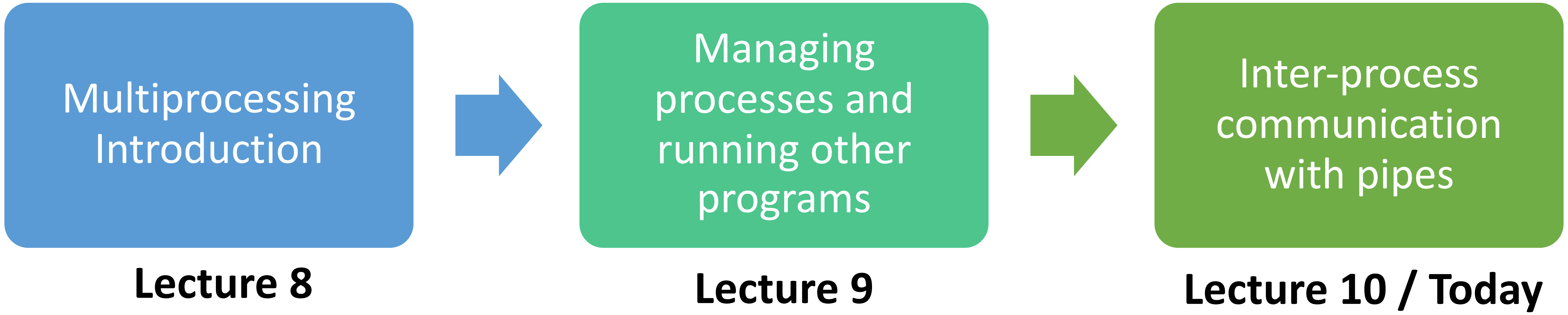
CS111, Lecture 11

Pipes, Continued

😷 masks strongly recommended

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

CS111 Topic 2: Multiprocessing


Multiprocessing Introduction
Lecture 8
Managing processes and running other programs
Lecture 9
Inter-process communication with pipes
Lecture 10 / Today

assign3: implement your own shell!

Learning Goals

- Learn about **pipe** and **dup2** to create and manipulate file descriptors
- Use pipes to redirect process input and output

Plan For Today

- **Recap**: Pipes so far
- Redirecting Process I/O
- ***Practice:*** implementing **subprocess**

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

Plan For Today

- **Recap: Pipes so far**
- Redirecting Process I/O
- ***Practice:*** implementing **subprocess**

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

Pipes

- A pipe is (sort of) like an “imaginary file” that we open twice, once for writing and once for reading. It consists of two file descriptors, one for reading and one for writing
- Whatever we write to the “write FD” we can read from the “read FD”
- To create these two file descriptors, we call the **pipe** system call

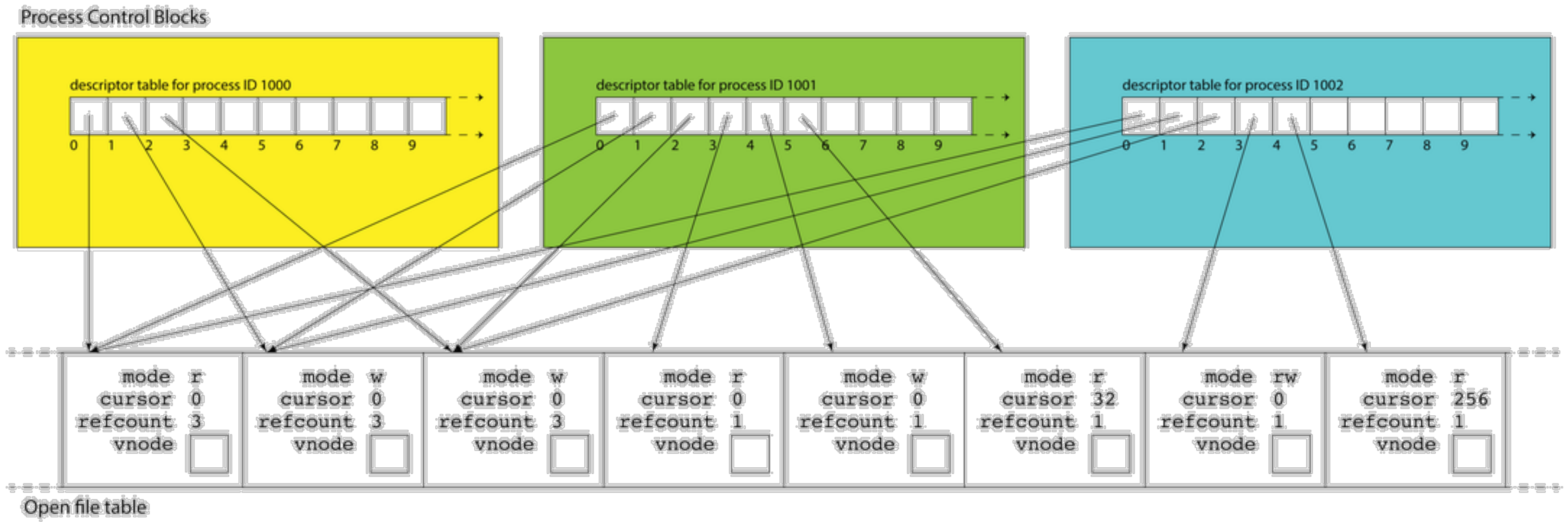
```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to fds[1] can be *read* from fds[0]. Returns 0 on success, or -1 on error.

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

pipe() and fork()

- **fork**() copies everything into the child, *including the file descriptor table*.
- We saw how each process's file descriptor table points to entries in a shared open file table. This explains how even though we copy file descriptors, they share state.
- E.g. if a parent creates a pipe and then calls fork, the child can access it, too, because its file descriptor table will point to the same open file table entries.

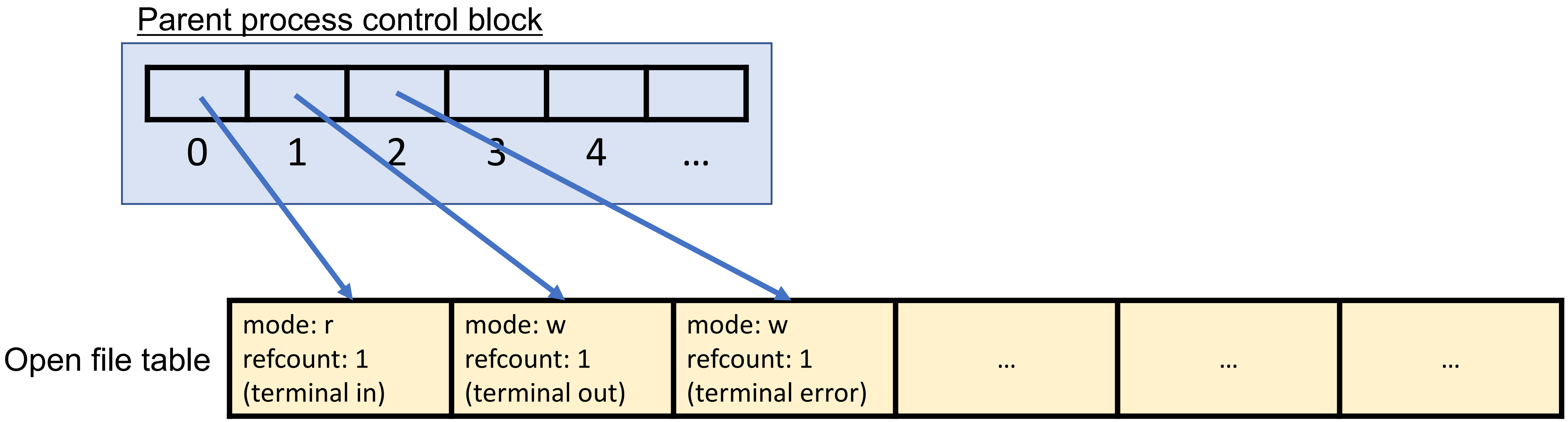


Process Control Blocks
descriptor table for process ID 1000
descriptor table for process ID 1001
descriptor table for process ID 1002
0 1 2 3 4 5 6 7 8 9
mode r cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode r cursor 0 refcount 1 vnode
mode w cursor 0 refcount 1 vnode
mode r cursor 32 refcount 1 vnode
mode rw cursor 0 refcount 1 vnode
mode r cursor 256 refcount 1 vnode
Open file table

Pipe Demo

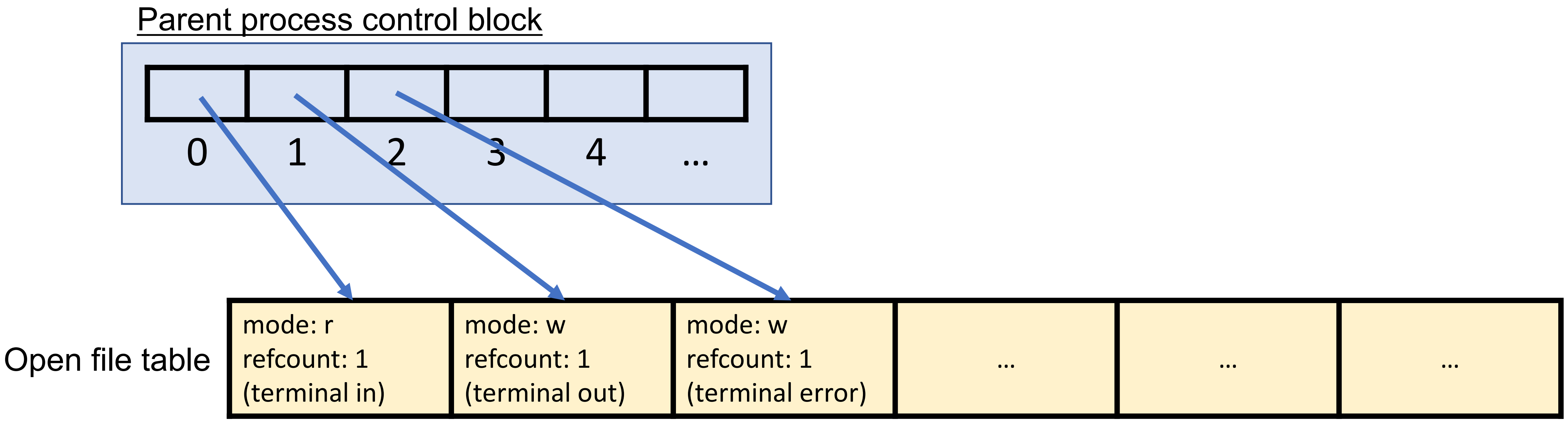
Let’s write a program where the parent sends a predetermined message to the child, which prints it out.

pipe()



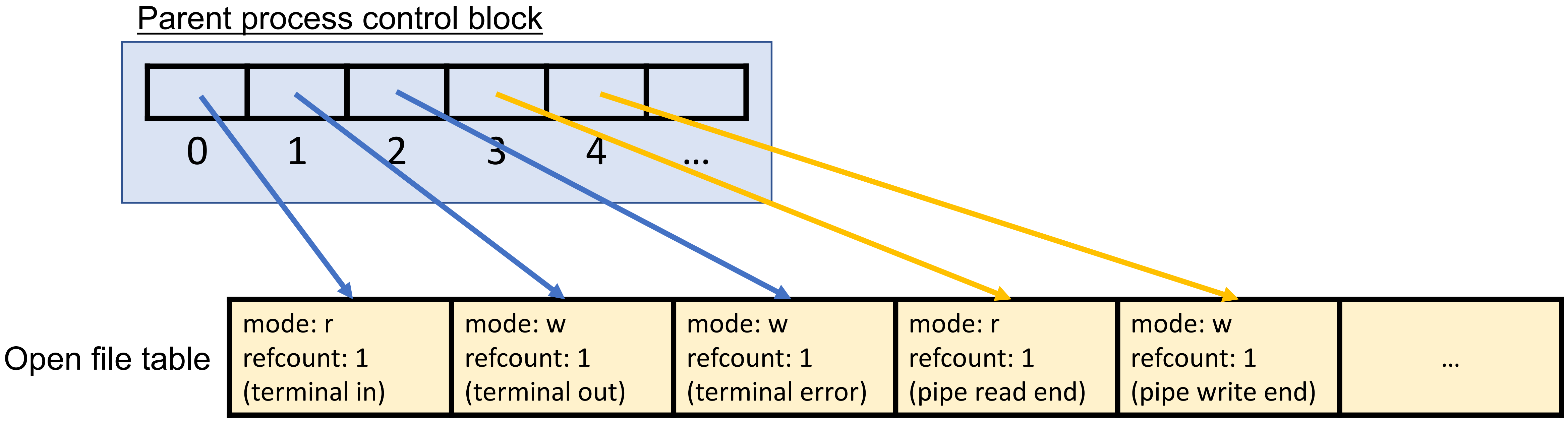
```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe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

pipe()



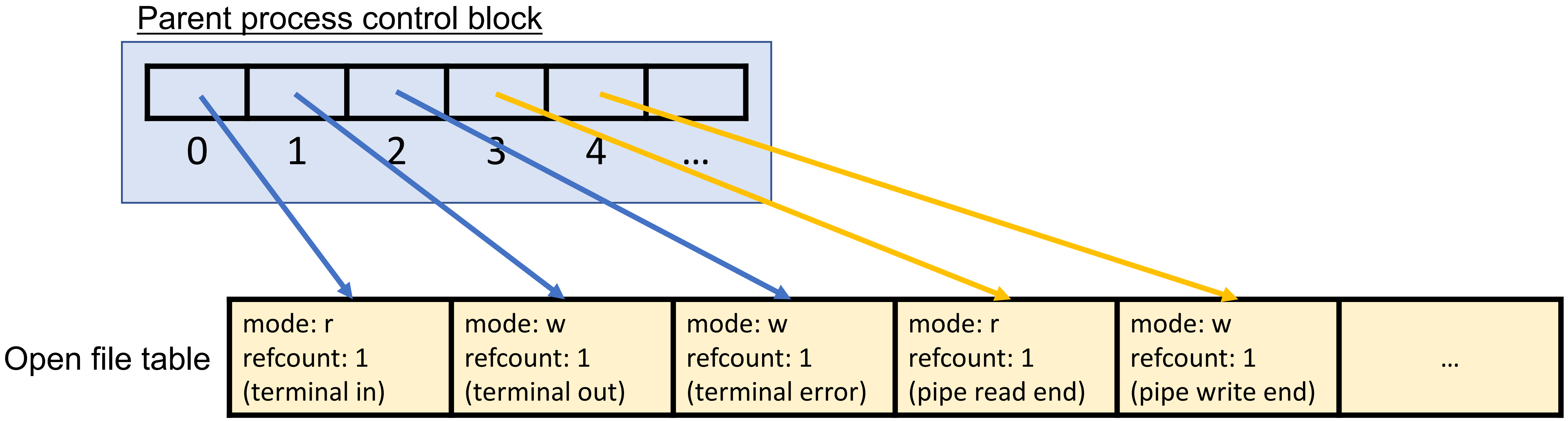
```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe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

pipe()



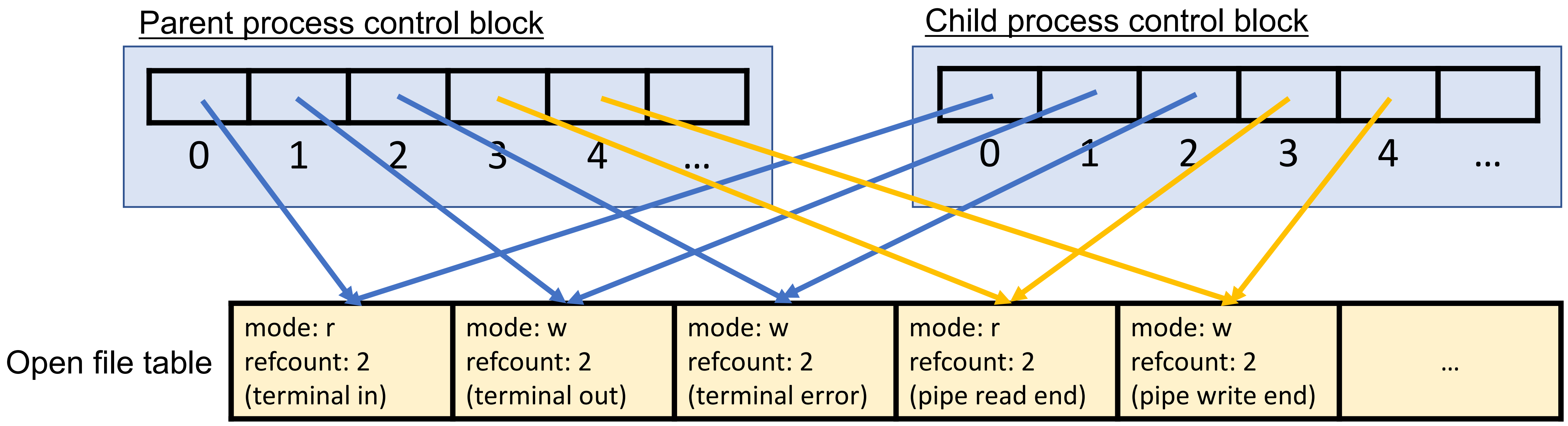
```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe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

pipe()



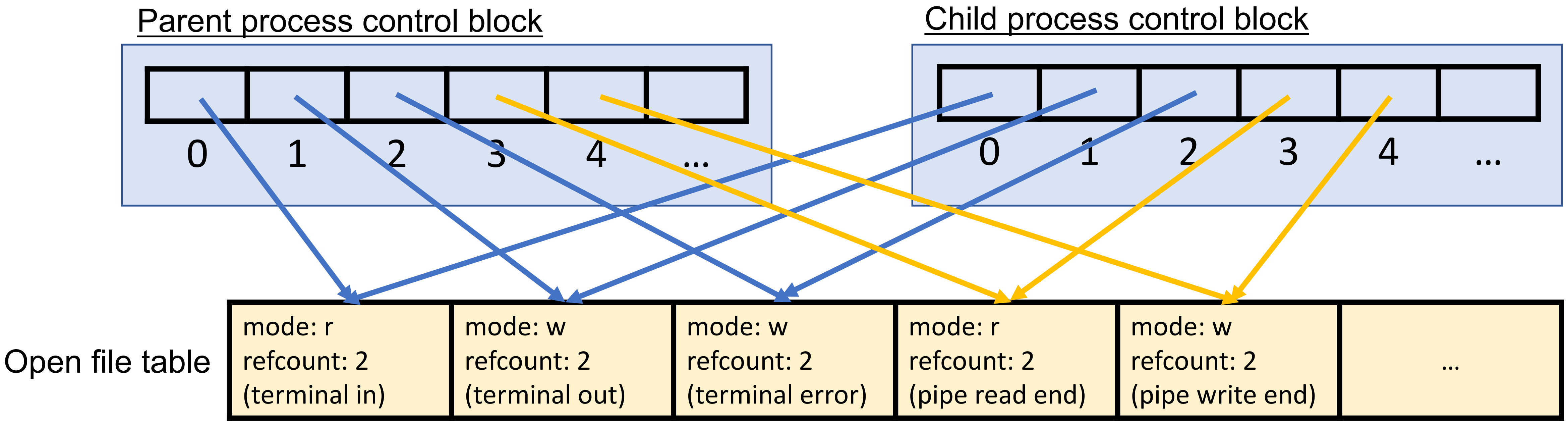
```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe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

pipe()



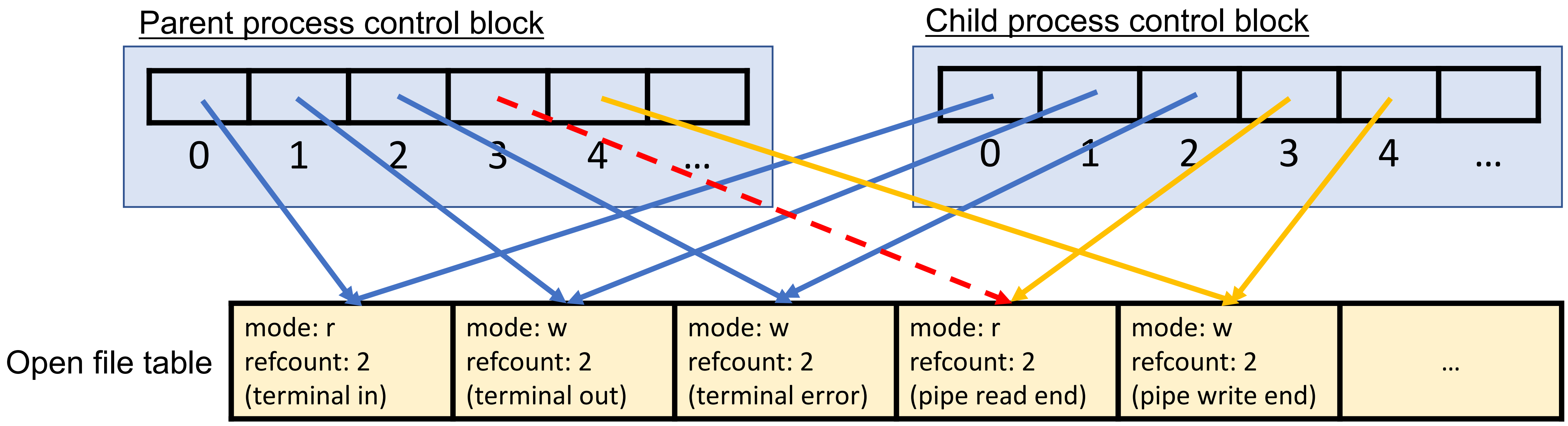
```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe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

pipe()



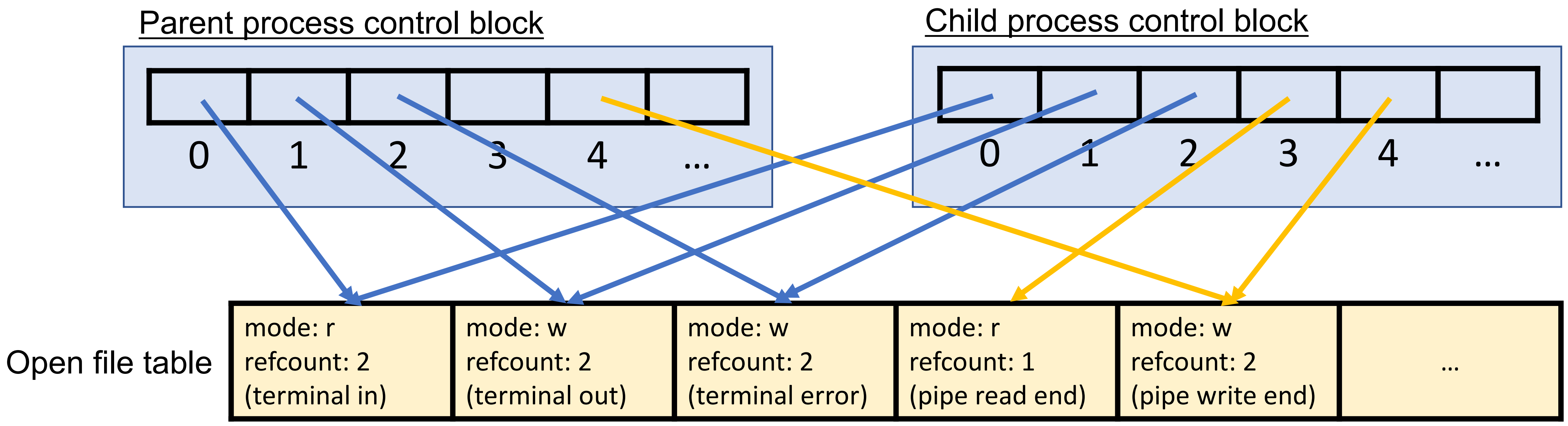
```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe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

pipe()



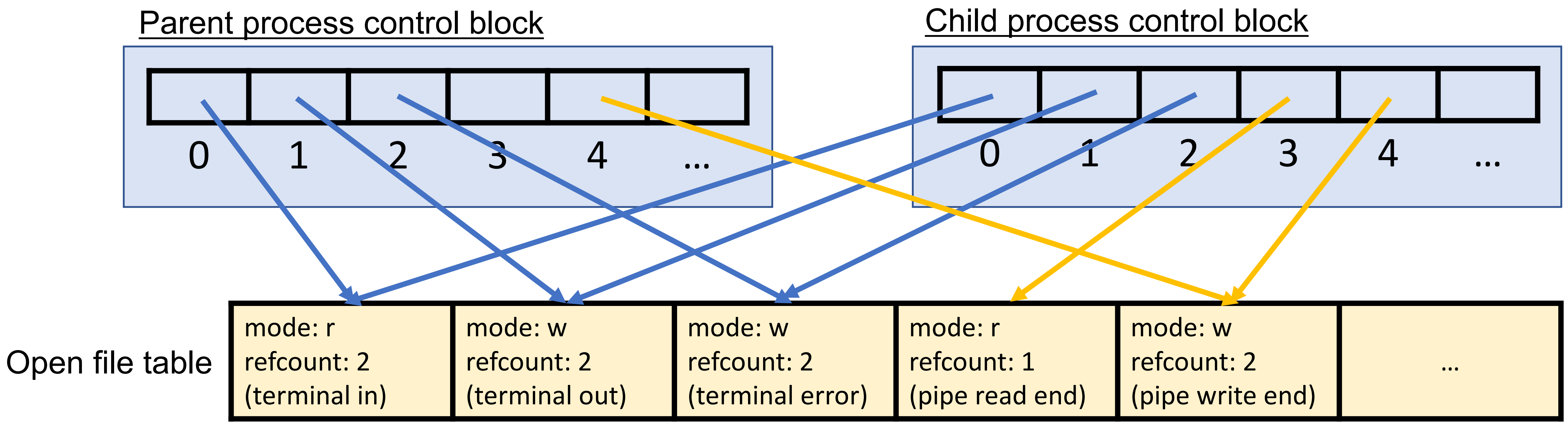
```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

pipe()



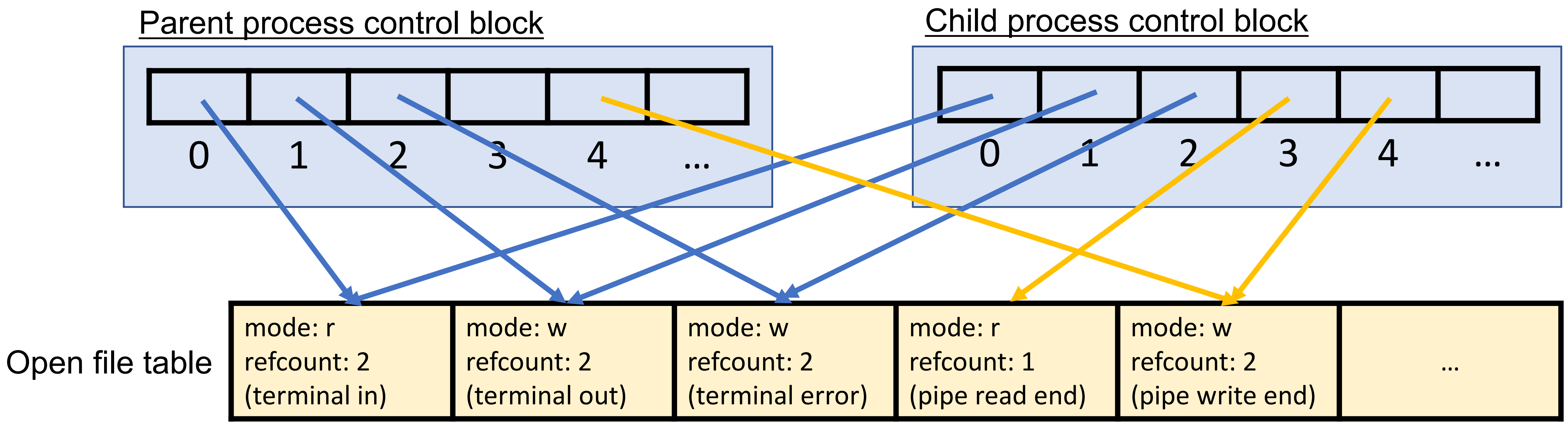
```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

pipe()



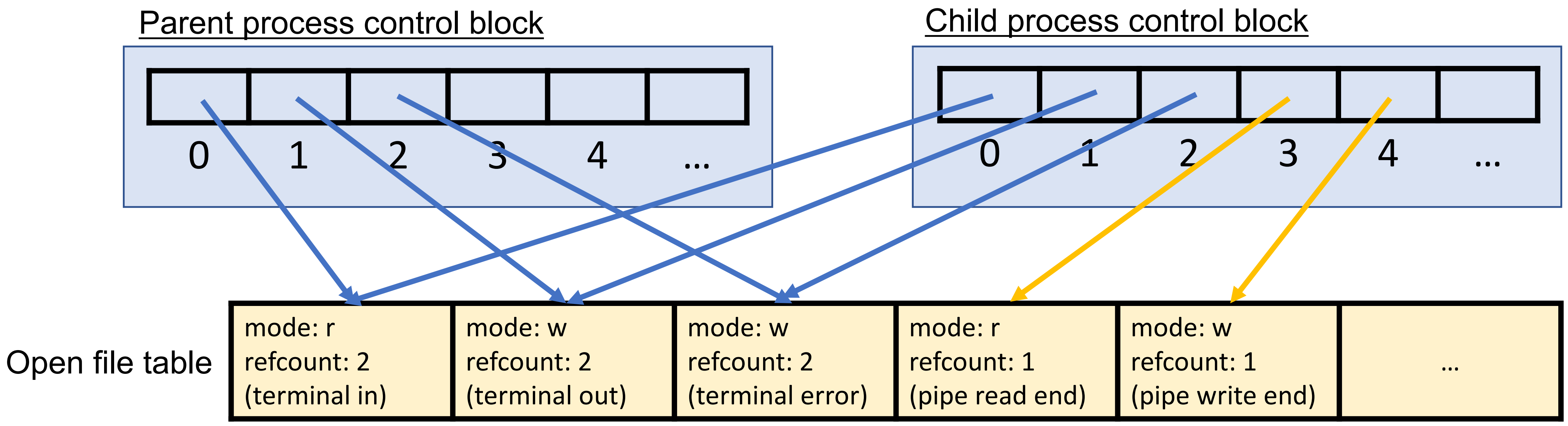
```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

pipe()



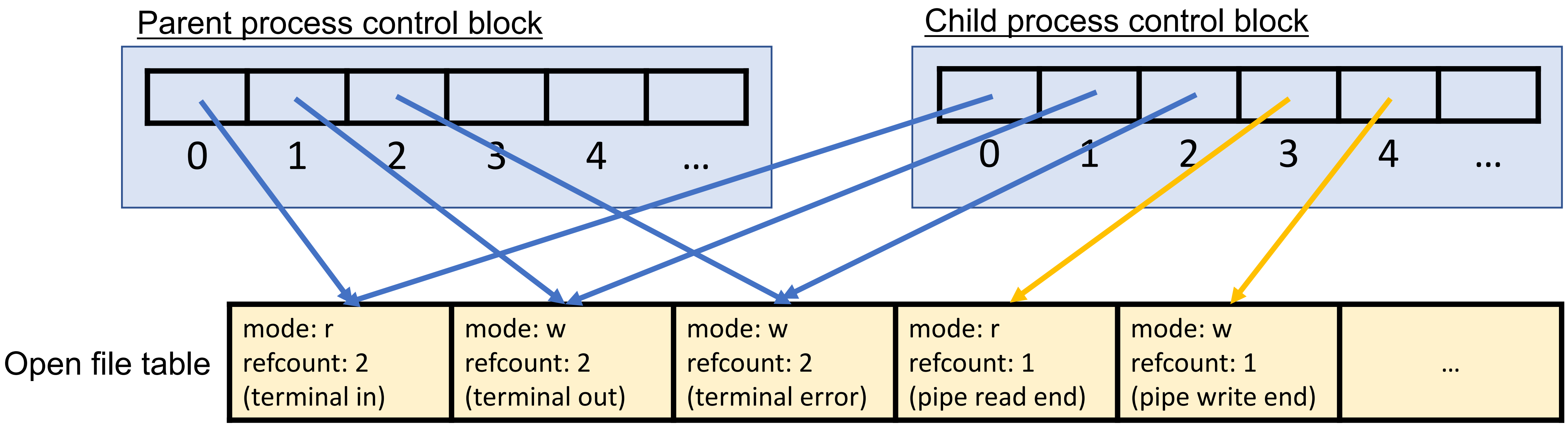
```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

pipe()



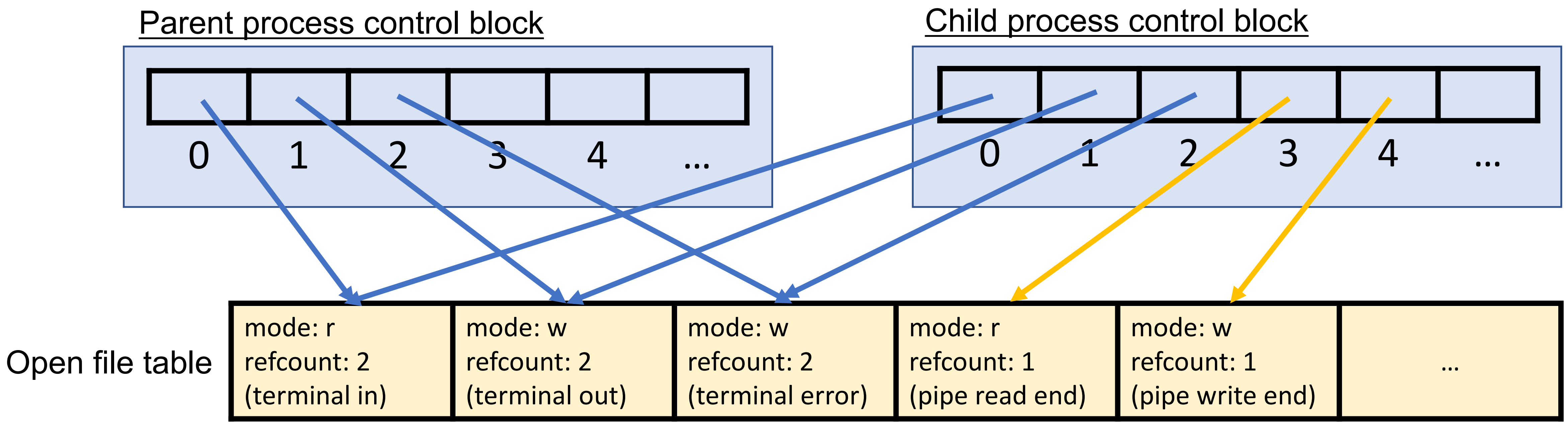
```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

pipe()



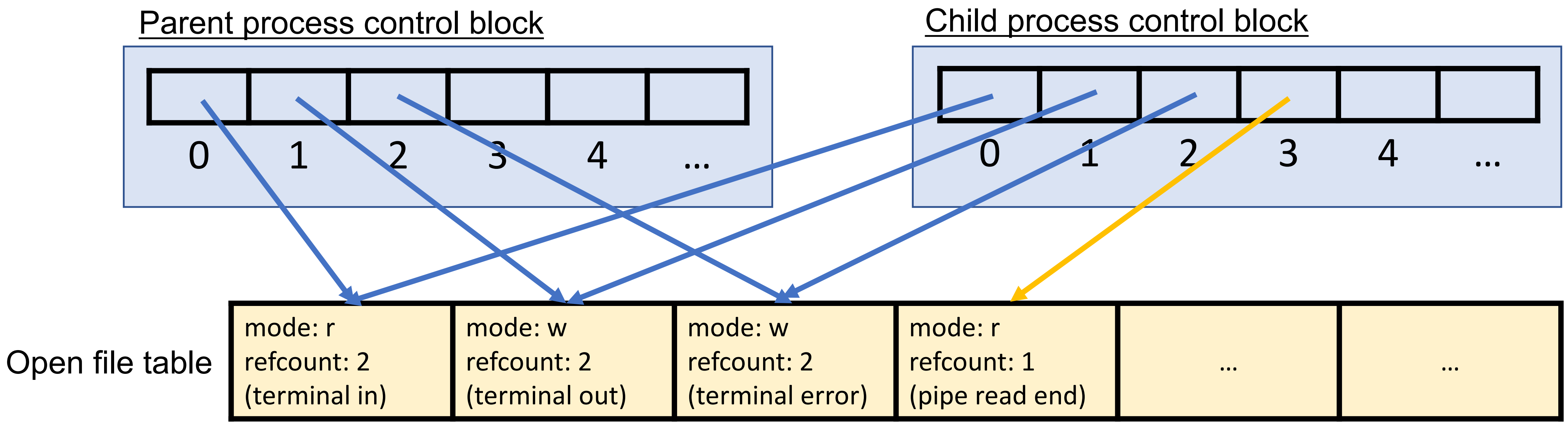
```
    ...
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

pipe()



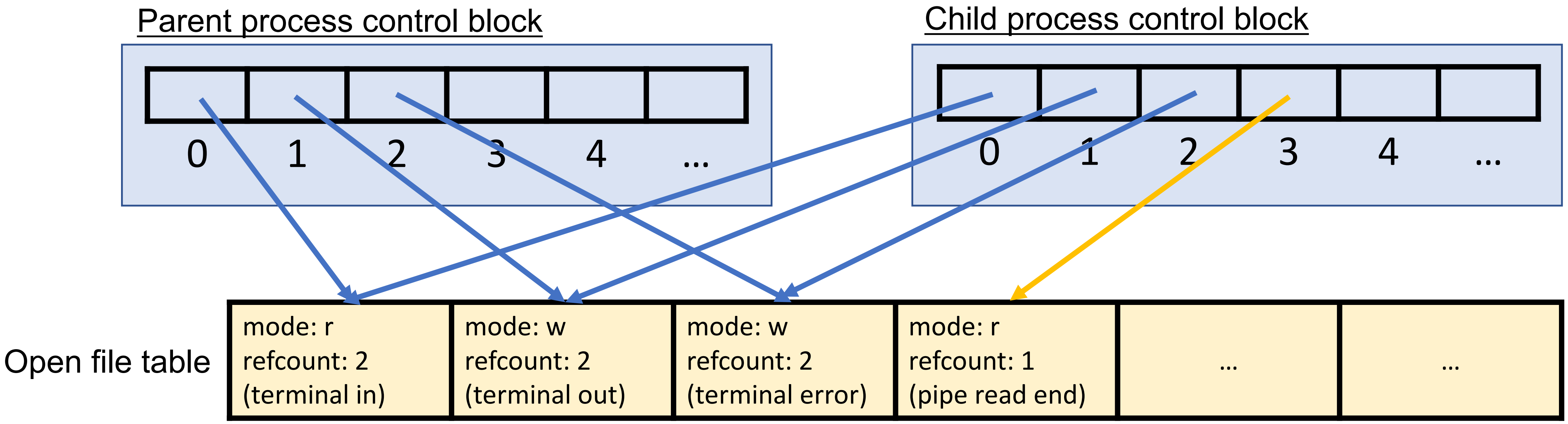
```
...
if (pidOrZero == 0) { // In the child, we only read from the pipe
    close(fds[1]);
    char buffer[bytesSent];
    read(fds[0], buffer, sizeof(buffer));
    close(fds[0]);
    printf("Message from parent: %s\n", buffer);
    return 0;
}  ...
```

pipe()



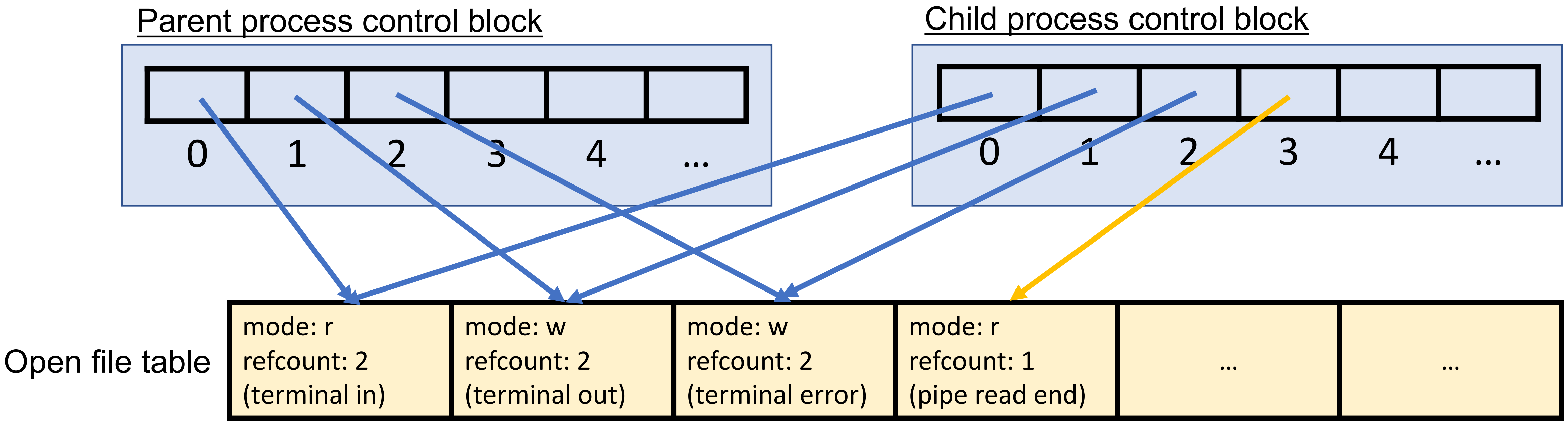
```
...
if (pidOrZero == 0) { // In the child, we only read from the pipe
    close(fds[1]);
    char buffer[bytesSent];
    read(fds[0], buffer, sizeof(buffer));
    close(fds[0]);
    printf("Message from parent: %s\n", buffer);
    return 0;
}  ...
```

pipe()



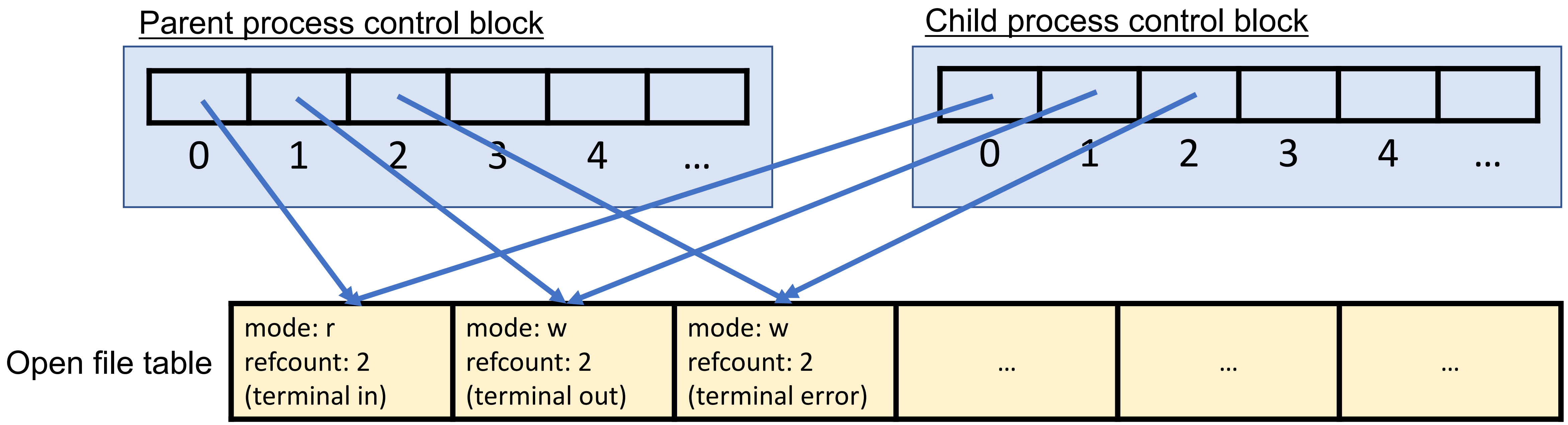
```
...
if (pidOrZero == 0) { // In the child, we only read from the pipe
    close(fds[1]);
    char buffer[bytesSent];
    read(fds[0], buffer, sizeof(buffer));
    close(fds[0]);
    printf("Message from parent: %s\n", buffer);
    return 0;
}  ...
```

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;
}  ...
```

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;
}  ...
```

pipe()

```
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) { // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }
    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

pipe() and fork()

- 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).

Key combined idea: 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.

Stalling Example

```
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

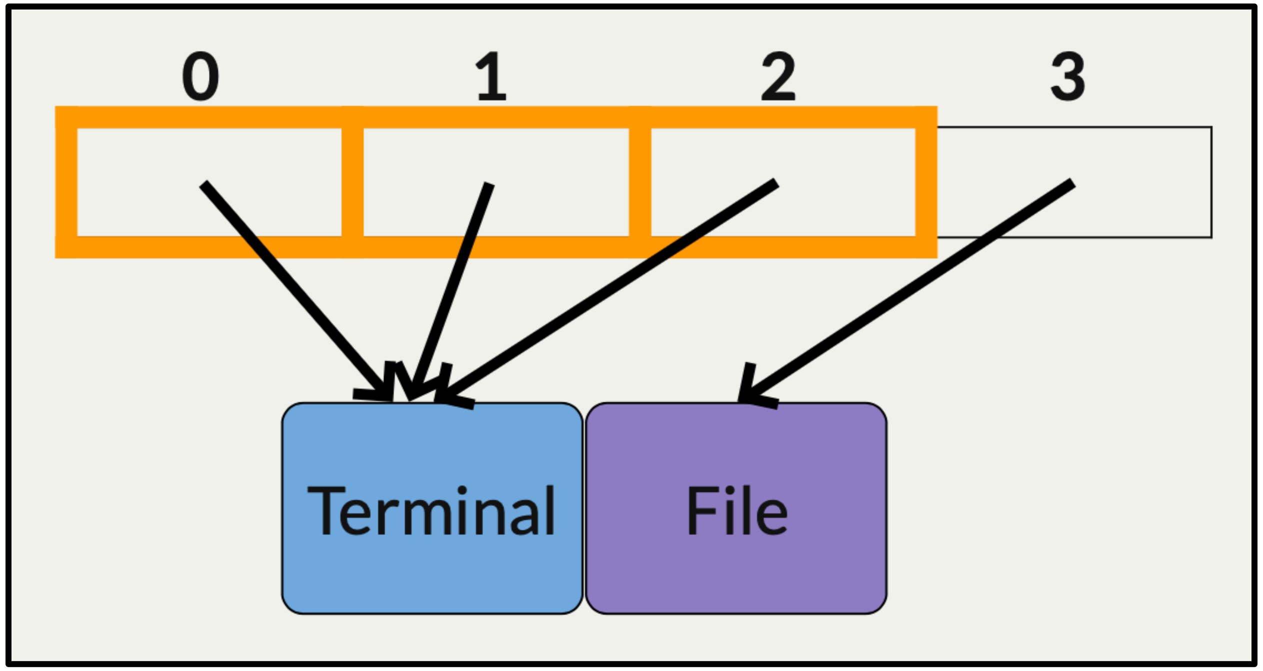
- **Recap**: Pipes so far
- **Redirecting Process I/O**
- ***Practice:*** implementing **subprocess**

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

Redirecting Process I/O

Each process has the special file descriptors STDIN (0), STDOUT (1) and STDERR (2)

- Processes assume these indexes are for these methods of communication (e.g. **printf** always outputs to file descriptor 1, STDOUT).



Idea: what happens if we change FD 1 to point somewhere else?

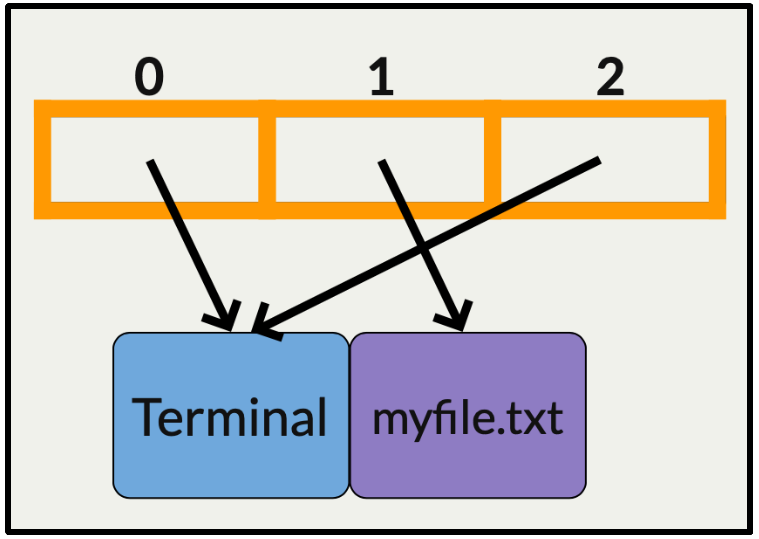
Redirecting Process I/O

Idea: what happens if we change FD 1 to point somewhere else?

```
int main() {
    printf("This will print to the terminal\n");
    close(STDOUT_FILENO);

    // fd will always be 1
    int fd = open("myfile.txt", O_WRONLY | O_CREAT
| O_TRUNC, 0644);

    printf("This will print to myfile.txt!\n");
    close(fd);
    return 0;
}
```



Redirecting Process I/O

This is useful for both pipes (as we'll see) as well as another shell feature; redirecting input or output:

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

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

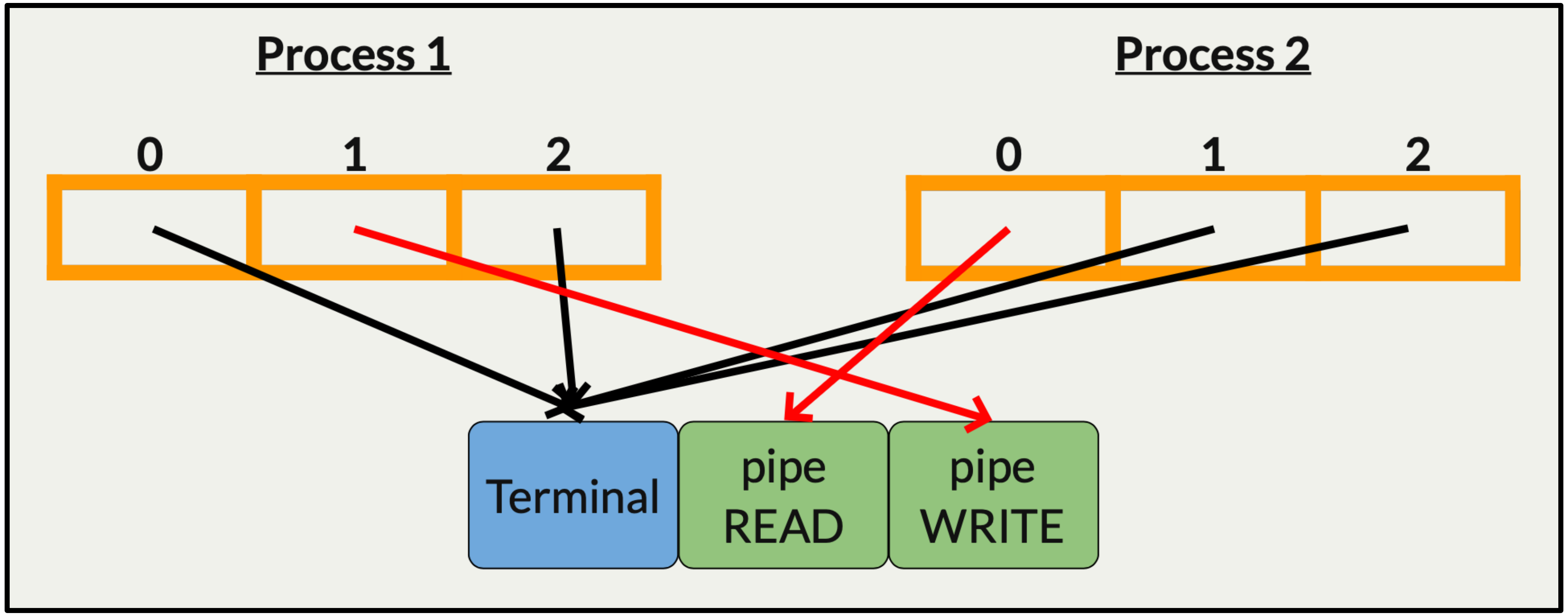
This reads input from a file instead of reading from the terminal

```
sort < input.txt
```

Redirecting Process I/O

Idea: what happens if we change a special FD to point somewhere else?

Could we do this with a pipe?



This is how piping works in the terminal! And the executables don't know they are using a pipe.

Redirecting Process I/O

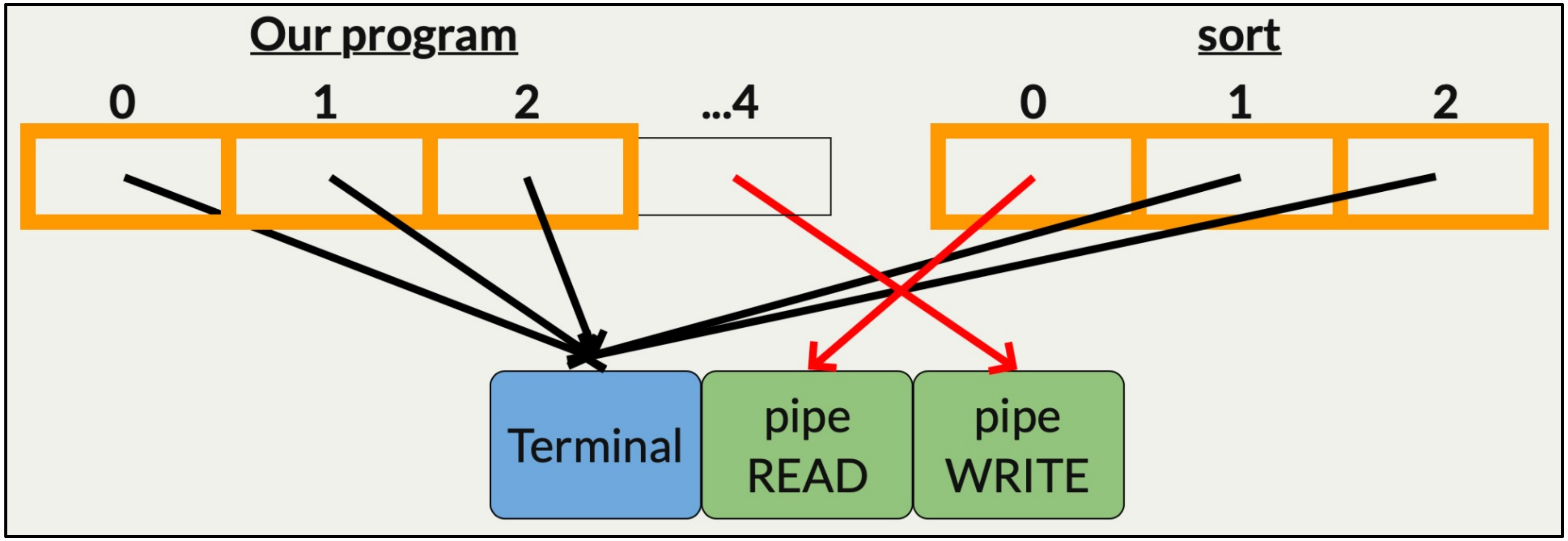
On assign3, milestone 3 tasks you with implementing this pipeline structure, where we spawn 2 child processes and connect the STDOUT of the first to the STDIN of the second via a pipe.

Today, we'll implement a stepping stone to this, a function **subprocess**. It will create 1 pipe and 1 child process such that if we write to the pipe's write end, it will send it to that 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.

Redirecting Process I/O

Stepping stone: our first goal is to write code that spawns another program and sends data to its STDIN via a separate file descriptor.



The **sort** executable has no idea its input is not coming from terminal entry!

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.

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

"Wait a minute...I thought execvp consumed the process? How do the file descriptors stick around?" **New insight: execvp** consumes the process, but *leaves the file descriptor table in tact!*

Redirecting Process I/O

One issue; how do we "connect" our pipe FDs to STDIN/STDOUT?

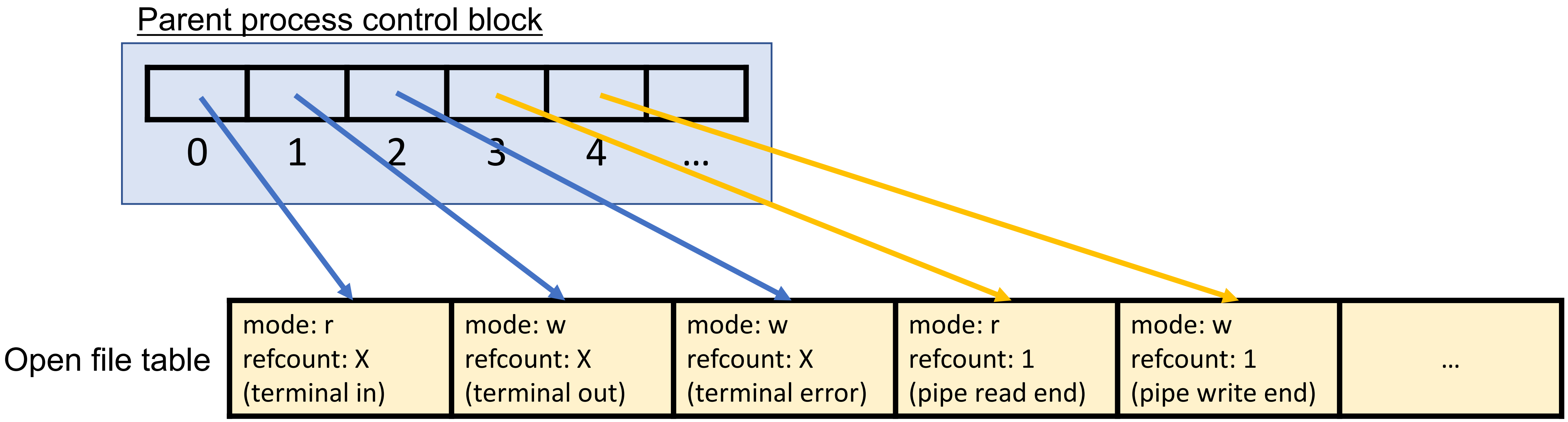
dup2 makes a copy of a file descriptor entry and puts it in another file descriptor index. This means both will now point to the same open file table entry. If the second parameter is an already-open file descriptor, it is closed before being used.

```
int dup2(int oldfd, int newfd);
```

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.

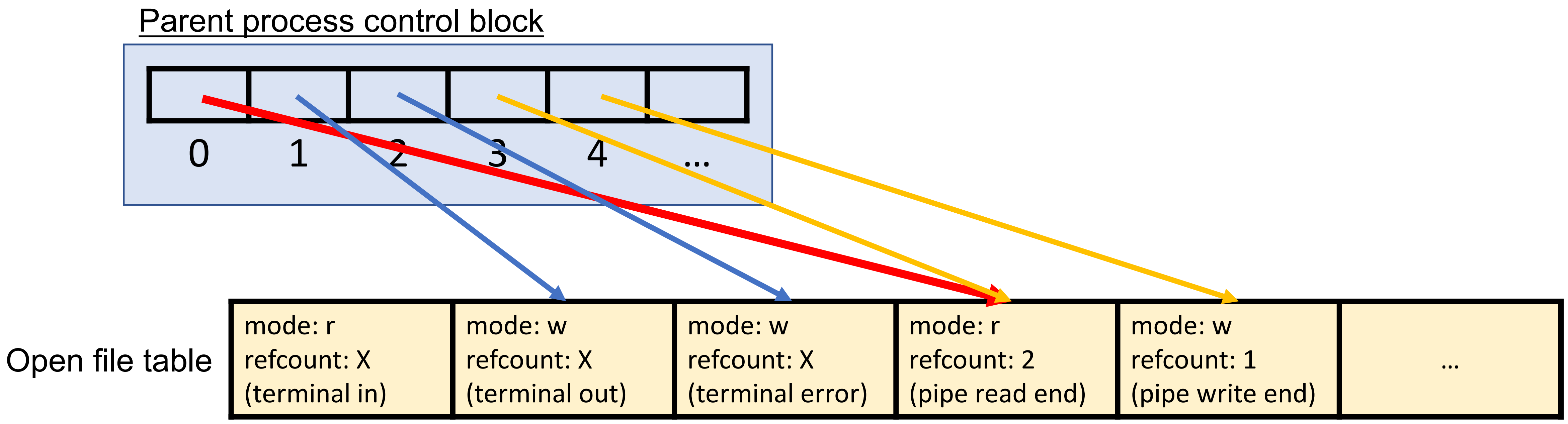
```
dup2(fds[0], STDIN_FILENO);
close(fds[0]);
```

Redirecting Process I/O



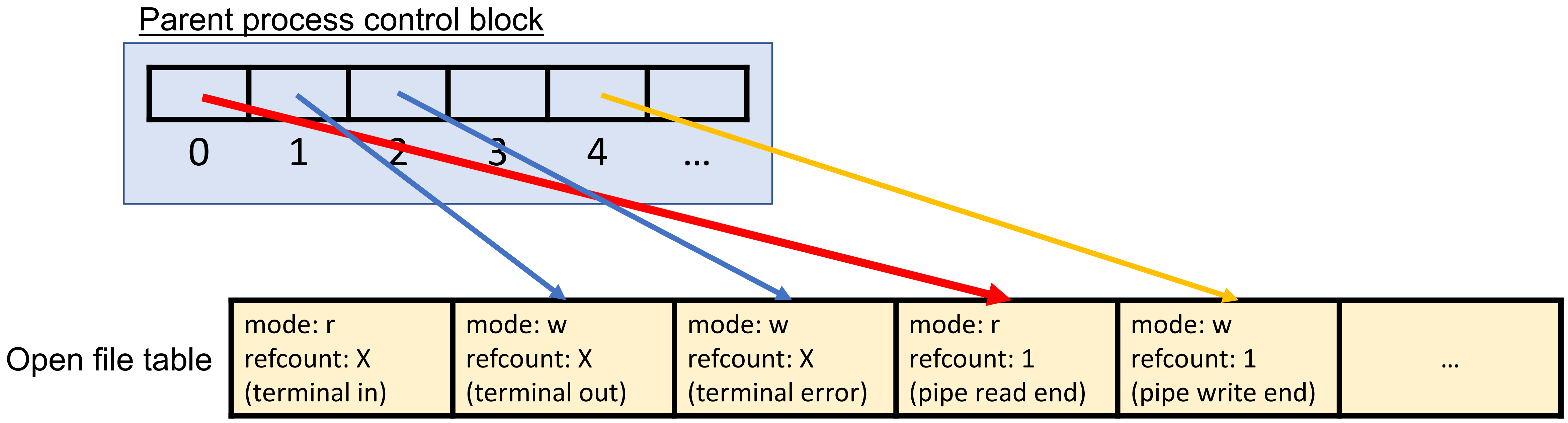
```
dup2(fds[0], STDIN_FILENO);  // assume fds[0] is 3
close(fds[0]);
```

Redirecting Process I/O



```
dup2(fds[0], STDIN_FILENO);  // assume fds[0] is 3
close(fds[0]);
```

Redirecting Process I/O



```
dup2(fds[0], STDIN_FILENO);  // assume fds[0] is 3
close(fds[0]);
```

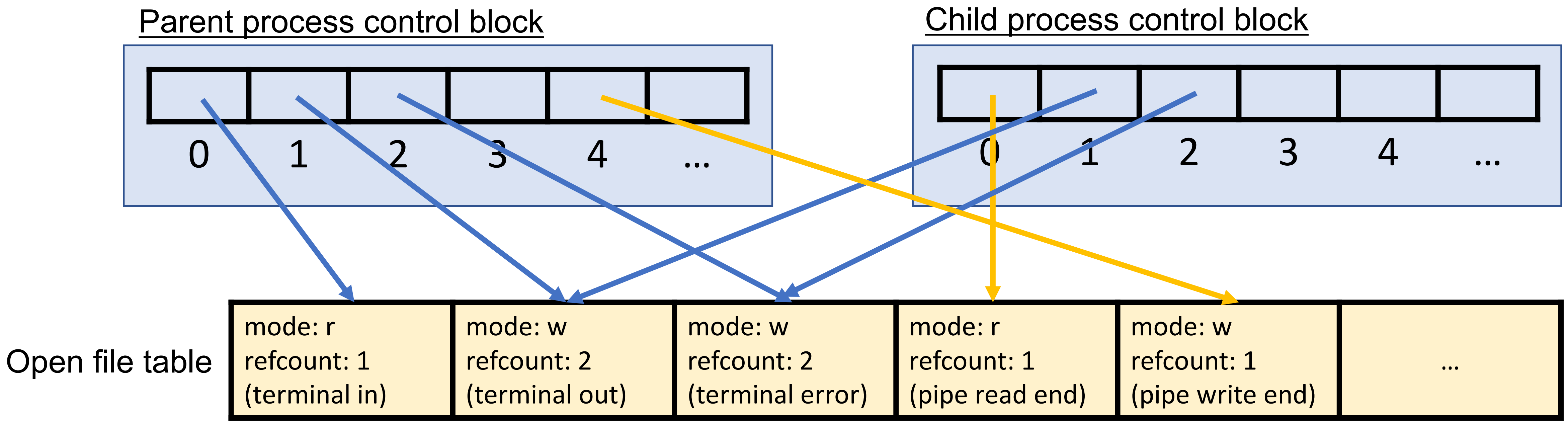
Plan For Today

- **Recap**: Pipes so far
- Redirecting Process I/O
- ***Practice:* implementing subprocess**

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

Redirecting Process I/O

Our stepping stone goal:



subprocess

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

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

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

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

YEAH Hours: this afternoon 3:10PM on Zoom!

Recap

- **<u>Recap</u>**: Pipes so far
- Redirecting Process I/O
- ***Practice:*** implementing **subprocess**

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 .
```