CS111, Lecture 10
Pipes

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

assign3: implement your own shell!

Learning Goals

- Get more practice with using **fork()** and **execvp**
- Learn about **pipe** to create and manipulate file descriptors
- Understand how file descriptors are duplicated across processes

Plan For Today

- **Recap and continuing**: **waitpid and execvp**
- **Demo:** our first shell
- Pipes

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

Plan For Today

- **Recap and continuing**: **waitpid** and **execvp**
- **Demo:** our first shell
- Pipes

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

waitpid()

A system call that a parent can call to wait for its child to exit:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **pid**: the PID of the child to wait on, or -1 to wait on any of our children
- **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
- It's important to wait on all children to clean up system resources

execvp()

The most common use for **fork** is not to spawn multiple processes to split up work, but instead to run a *completely separate program* under your control and communicate with it.

- This is what a **shell** is; it is a program that prompts you for commands, and it executes those commands in separate processes.

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

To run another executable, we must specify the (NULL-terminated) arguments to be passed into its **main** function, via the **argv** parameter.

- For our programs, **path** and **argv[0]** will be the same

execvp has many variants (see **man execvp**) but we'll just be using **execvp**.

execvp()

```c
// execvp-demo.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    char *args[] = {"/bin/ls", "-l", "/usr/class/cs111/lecture-code",
                    NULL};
    execvp(args[0], args);
    printf("This only prints if an error occurred.\n");
    return 0;
}
```

```
$ ./execvp-demo
Hello, world!
total 4
drwx------ 2 troccoli operator 2048 Oct  9 16:21 lect5
drwx------ 2 troccoli operator 2048 Oct 13 22:19 lect9
```

Implementing a Shell

How is execvp useful?

- This is the way that we can run other programs
- However, we often don't want to cannibalize the current process
- Instead: we will usually fork off a child process and call execvp there. The child process will be consumed, but that's ok
- Key idea: the process is still the child process, and **the parent can still wait on it**. It's just running another program.

Implementing a Shell

A shell is essentially a program that repeats asking the user for a command and running that command

How do we run a command entered by the user?

1. Call **fork** to create a child process
2. In the child, call **execvp** with the command to execute
3. In the parent, wait for the child with **waitpid**

For assign3, you'll use this pattern to build your own shell, stsh ("Stanford shell") with various functionality of real Unix shells.

Demo: first-shell-soln.cc

Plan For Today

- Recap: **waitpid** and **execvp**
- **Demo:** our first shell
- **Pipes**

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

Is there a way that the parent and child processes can communicate?

(why is this useful? In a shell, we can use the pipe "|" character to feed the output of one command as the input of the next command.)

Pipes

How can we let two processes send arbitrary data back and forth?

- **Idea:** Create a shared file that one process could write to, and another process could read from?
- **Problem:** we don't want to clutter the filesystem with files every time two processes want to communicate.
- **Solution:** have the operating system set up an “imaginary shared file” for us.
  - It will give us two new file descriptors - one for writing, another for reading.
  - If someone writes data to the write FD, it can be read from the read FD.
  - It's *not actually a physical file on disk* - we are just using files as an abstraction. The OS maintains this pretend file for us in memory, but it appears to us just like a shared file.
- Core Unix principle: modeling things as “files”

pipe()

```
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()

```
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
    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.
```

pipe()

The most common use for **pipe** is not to send and receive data within a single process, but to share a pipe between two processes, where one reads and the other writes. How is this possible?

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!

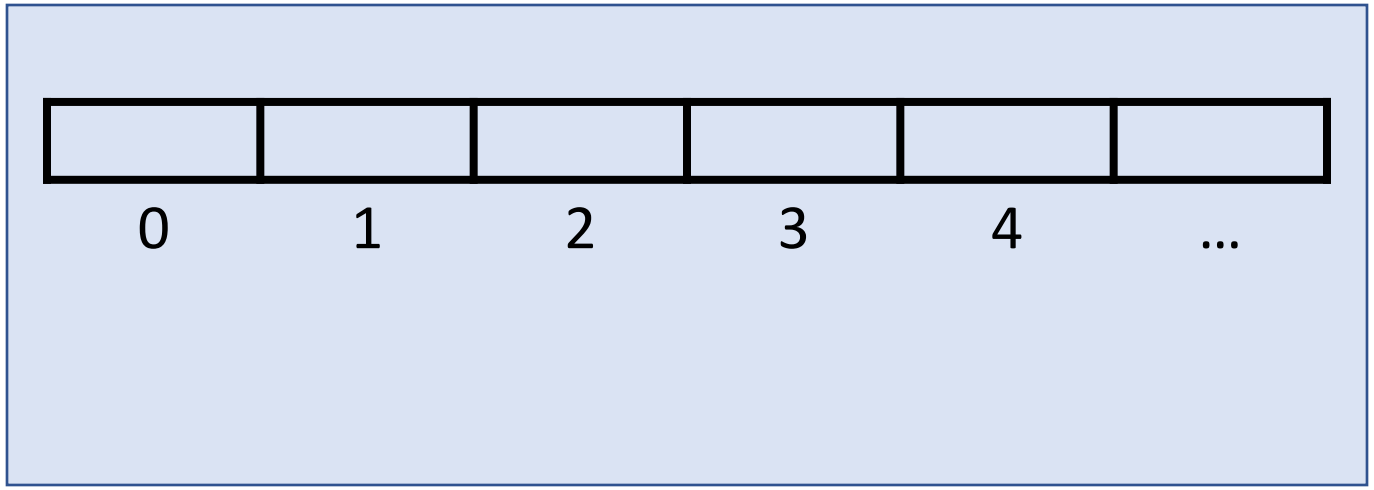
How does this file descriptor duplication work?

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!**

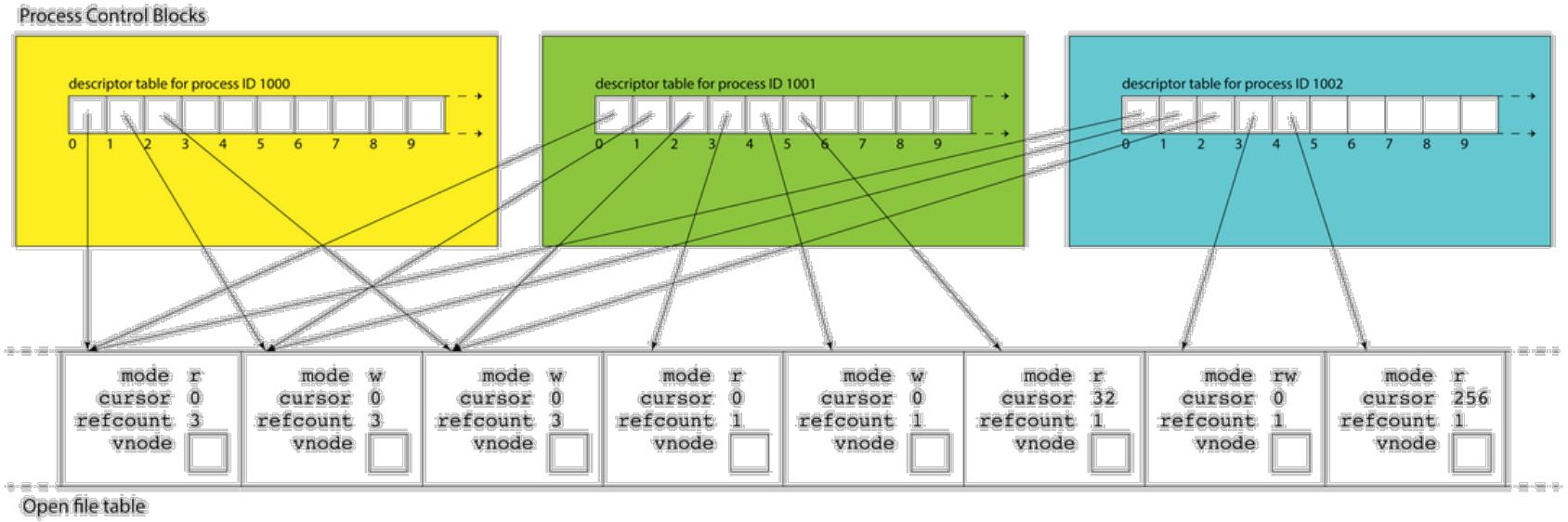
Process Control Block



File Descriptor Table

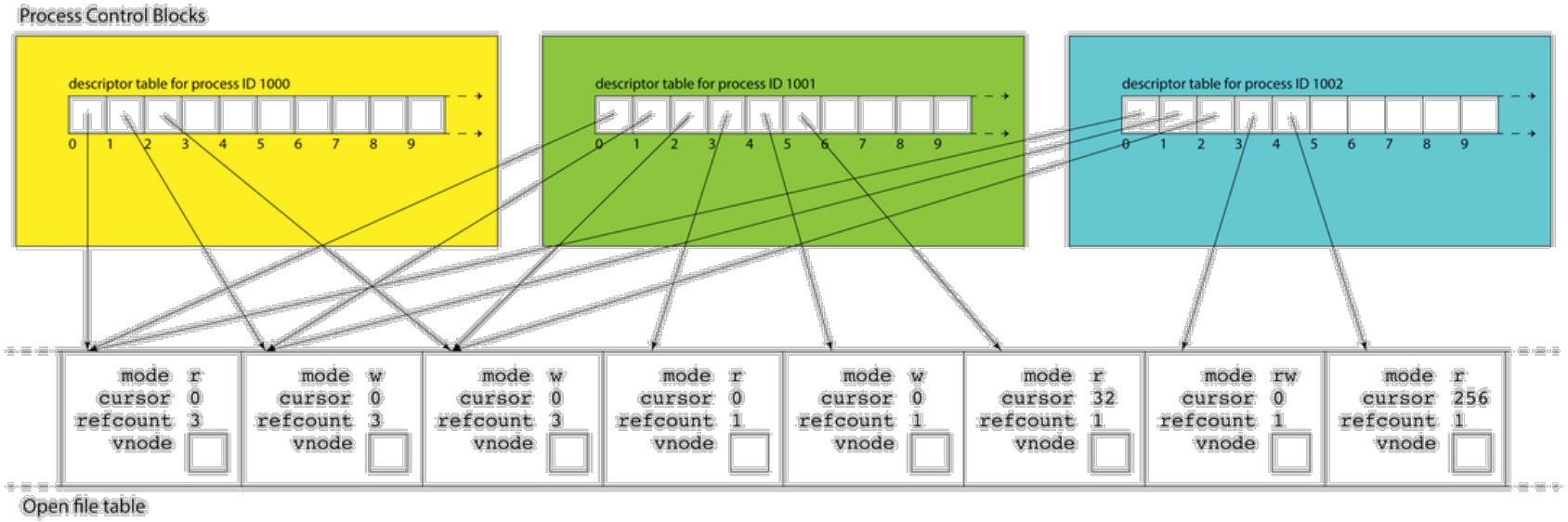
Key idea: a file descriptor is **an index into that process's file descriptor table**.

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



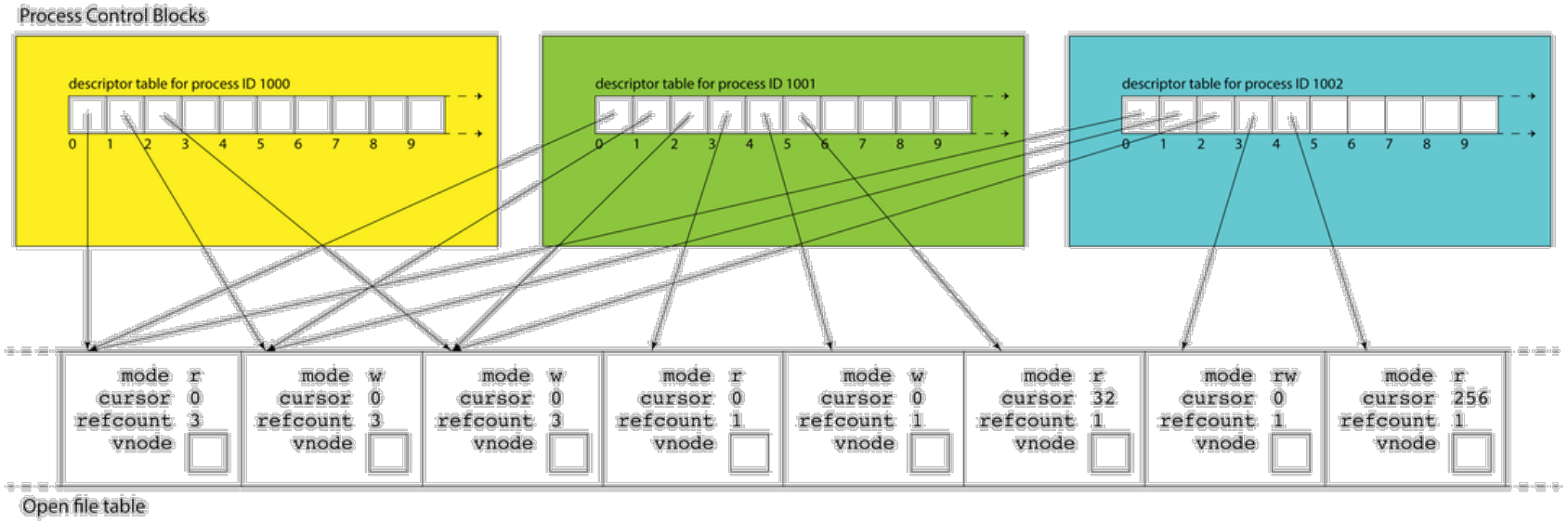
Open File Table

- Calling **open** creates a new open file table entry, and a new file descriptor index points to it.
- Calling **pipe** creates 2 new open file table entries, and 2 new file descriptor indexes point to them.
- 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!



Open File Table

- Each open file table entry keeps a *reference count*, a count of the number of file descriptor table entries pointing to it.
- This ref count increases whenever a new file descriptor index points to it.
- When we call **close** in our program, file descriptor index no longer points to open file table entry, open file table entry's ref count decremented.
- When open file table entry's ref count == 0, it's deleted



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?

2.

This 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)
- Because they're file descriptors, there's no global name for the pipe (another process can't "connect" to the pipe).
- Each pipe is uni-directional (one end is read, the other write)
- **Key Idea: read()** *blocks* until the bytes are available or there is no more to read (e.g. end of file or pipe write end closed). So if one process is reading, it will wait until the other writes.

Recap

- **Recap**: **waitpid** and **execvp**
- **Demo:** our first shell
- Pipes

Next time: more usage of pipes

Lecture 10 takeaway: shells work by spawning child processes that call execvp. Pipes are sets of file descriptors that let us read/write. We can share pipes with child processes to send arbitrary data back and forth.

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