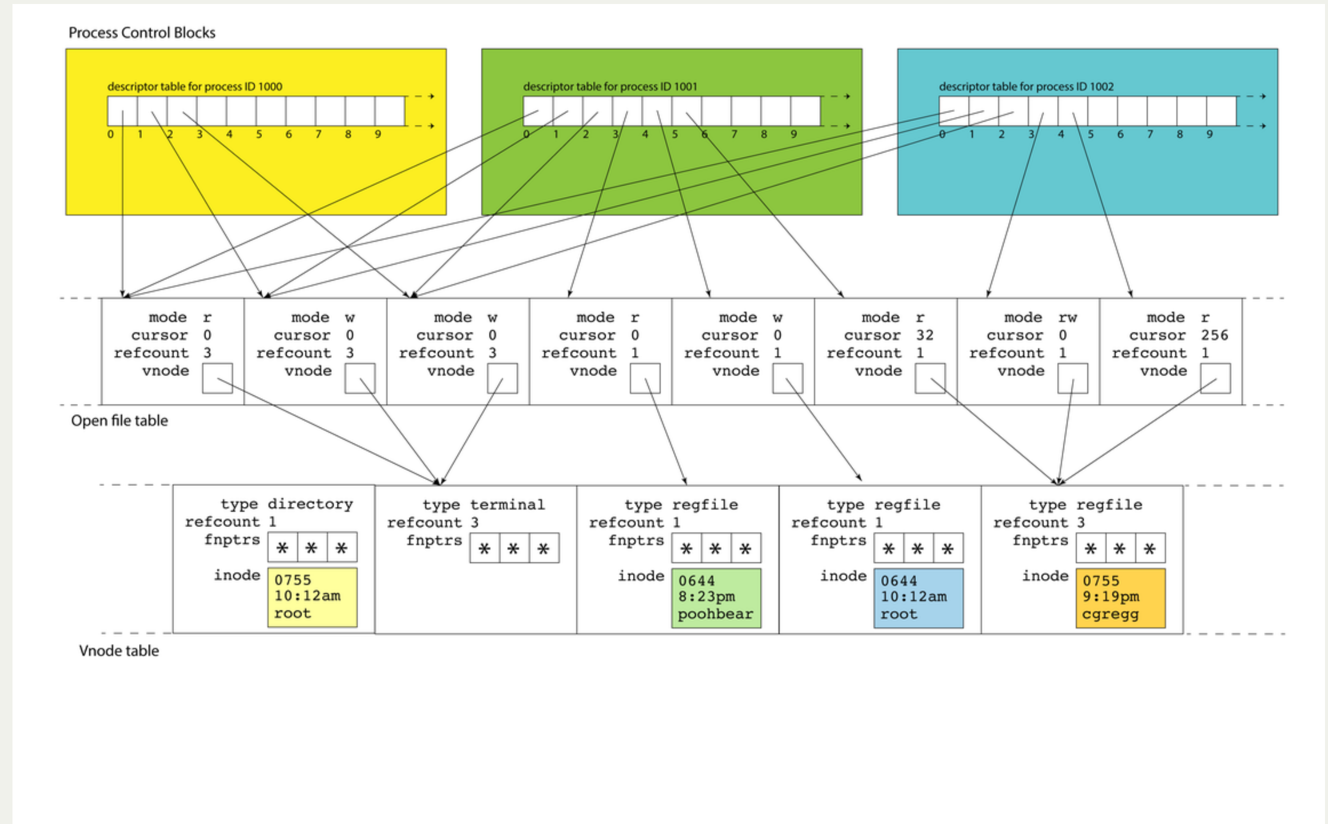
CS110 Lecture 5: File Descriptors, System Calls and Multiprocessing

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

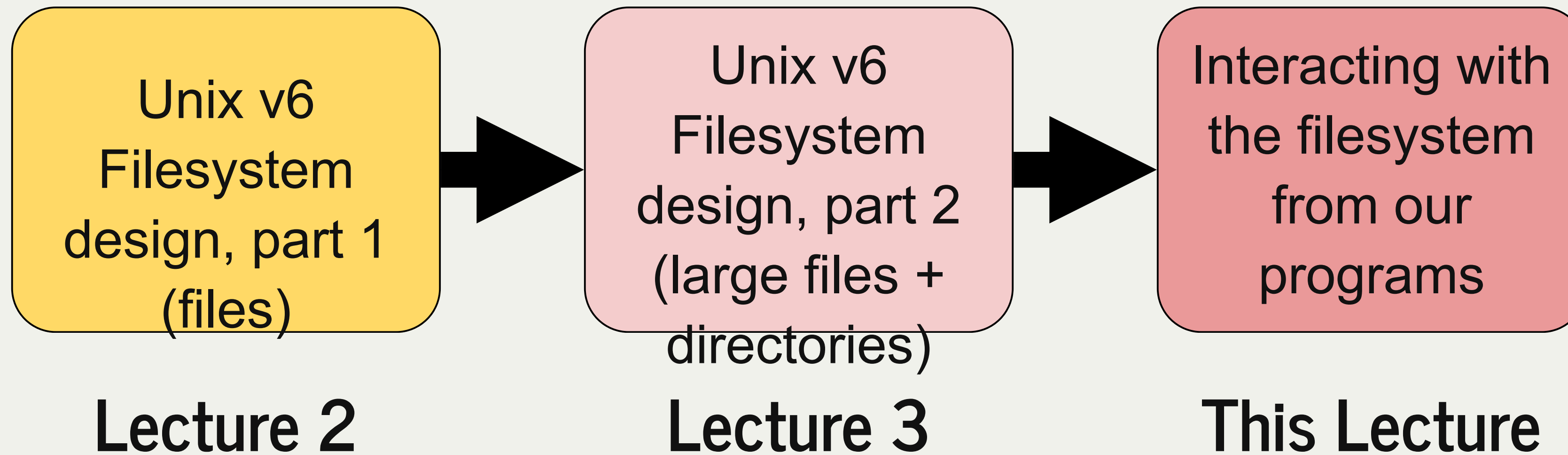
Instructors: Nick Troccoli and Jerry Cain



PDF of this presentation

CS110 Topic 1: How can we design filesystems to store and manipulate files on disk, and how can we interact with the filesystem in our programs?

Learning About Filesystems



assign2: implement portions of a filesystem!

Learning Goals

- Understand the use and versatility of file descriptors
- Learn how file descriptors are used by the operating system to manage open files
- Learn how system calls are made while preserving privacy and security
- Become familiar with how to write a program that spawns another program

Lecture Plan

- Recap: File descriptors, **open()**, **close()**, **read()** and **write()**
- Operating system data structures
- How are system calls made?
- Introduction to multiprocessing

Today's Ed Thread: https://edstem.org/us/courses/16701/discussion/996212

Lecture Plan

- **Recap: File descriptors, open(), close(), read() and write()**
- Operating system data structures
- How are system calls made?
- Introduction to multiprocessing

System Calls

- Functions to interact with the operating system are part of a group of functions called **system calls**.
- A system call is a public function provided by the operating system. They are tasks the operating system can do for us that we can't do ourselves.
- **open()**, **close()**, **read()** and **write()** are 4 system calls we use to interact with files.

open()

A function that a program can call to open a file, and potentially create a file:

```c
// if opening an existing file
int open(const char *pathname, int flags);

// if there's potential to create a new file
int open(const char *pathname, int flags, mode_t mode);
```

- **pathname:** the path to the file you wish to open
- **flags:** a bitwise OR of options specifying the behavior for opening the file
- **mode** (if applicable): the permissions to attempt to set for a created file
- the return value is a **file descriptor** representing the opened file, or -1 on error

Many possible flags (see man page). You must include exactly one of **O_RDONLY, O_WRONLY, O_RDWR**.

O_TRUNC: if the file exists already, clear it ("truncate it").

O_CREAT: if the file doesn't exist, create it

File Descriptors

- A **file descriptor** is like a "ticket number" representing your currently-open file.
- It is a unique number assigned by the operating system to refer to that file
- Each program has its own file descriptors
- When you wish to refer to the file (e.g. read from it, write to it) you must provide the file descriptor.
- [NEW] file descriptors are assigned in ascending order (next FD is lowest unused)

close()

A function that a program can call to close a file when done with it.

```
int close(int fd);
```

It's important to close files when you are done with them to preserve system resources.

- **fd:** the file descriptor you'd like to close.

You can use **valgrind** to check if you forgot to close any files.

`read()` and `write()`

```c
// read bytes from an open file
ssize_t read(int fd, void *buf, size_t count);
```

- **fd:** the file descriptor for the file you'd like to read from
- **buf**: the memory location where the read-in bytes should be put
- **count:** the number of bytes you wish to read
- The function returns -1 on error, 0 if at end of file, or nonzero if bytes were read (may not read all bytes you ask it to!)

```c
// write bytes to an open file
ssize_t write(int fd, const void *buf, size_t count);
```

*Same as **read()**, except the function writes the **count** bytes in **buf** to the file, and returns the number of bytes written.*

Example: Copy

The **copy** program emulates **cp**; it copies the contents of a source file to a specified destination.

```
1  void copyContents(int sourceFD, int destinationFD) {
2    char buffer[kCopyIncrement];
3    while (true) {
4      ssize_t bytesRead = read(sourceFD, buffer, sizeof(buffer));
5      if (bytesRead == 0) break;
6
7      size_t bytesWritten = 0;
8      while (bytesWritten < bytesRead) {
9        ssize_t count = write(destinationFD, buffer + bytesWritten, bytesRead - bytesWritten);
10       bytesWritten += count;
11     }
12   }
13 }
14
15 int main(int argc, char *argv[]) {
16   int sourceFD = open(argv[1], O_RDONLY);
17   int destinationFD = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, kDefaultPermissions);
18
19   copyContents(sourceFD, destinationFD);
20
21   close(sourceFD);
22   close(destinationFD);
23   return 0;
24 }
```

`copy-soln.c` and `copy-soln-full.c` (with error checking)

File Descriptors

File descriptors are just integers - for that reason, we can store and access them just like integers.

- If you're interacting with many files, it may be helpful to have an *array of file descriptors*

There are 3 special file descriptors provided by default to each program:

- 0: standard input (user input from the terminal) - STDIN_FILENO
- 1: standard output (output to the terminal) - STDOUT_FILENO
- 2: standard error (error output to the terminal) - STDERR_FILENO

[NEW] Programs always assume that 0,1,2 represent STDIN/STDOUT/STDERR. Even if we change them! (eg. we close FD 1, then open a new file). (this is how `cat in.txt > out.txt` works)

Example: Copy Extended

The **copy-extended** program emulates **tee**; it copies the contents of a source file to specified destination(s), and also outputs it to the terminal.

```
1  // difference #1: an array of destination file descriptors
2  int destinationFDs[argc - 1];
3
4  // Include the terminal (STDOUT) as the first "file" so it's also printed
5  destinationFDs[0] = STDOUT_FILENO;
6
7  for (size_t i = 2; i < argc; i++) {
8    destinationFDs[i - 1] = open(argv[i], O_WRONLY | O_CREAT | O_EXCL, kDefaultPermissions);
9  }
10 ...
```

```
1 // difference #2: we write each chunk to every destination
2 for (size_t i = 0; i < numDestinationFDs; i++) {
3   size_t bytesWritten = 0;
4   while (bytesWritten < bytesRead) {
5     ssize_t count = write(destinationFDs[i], buffer + bytesWritten, bytesRead - bytesWritten);
6     bytesWritten += count;
7   }
8 }
9 ...
```

`copy-extended-soln.c` and `copy-extended-soln-full.c` (with error checking)

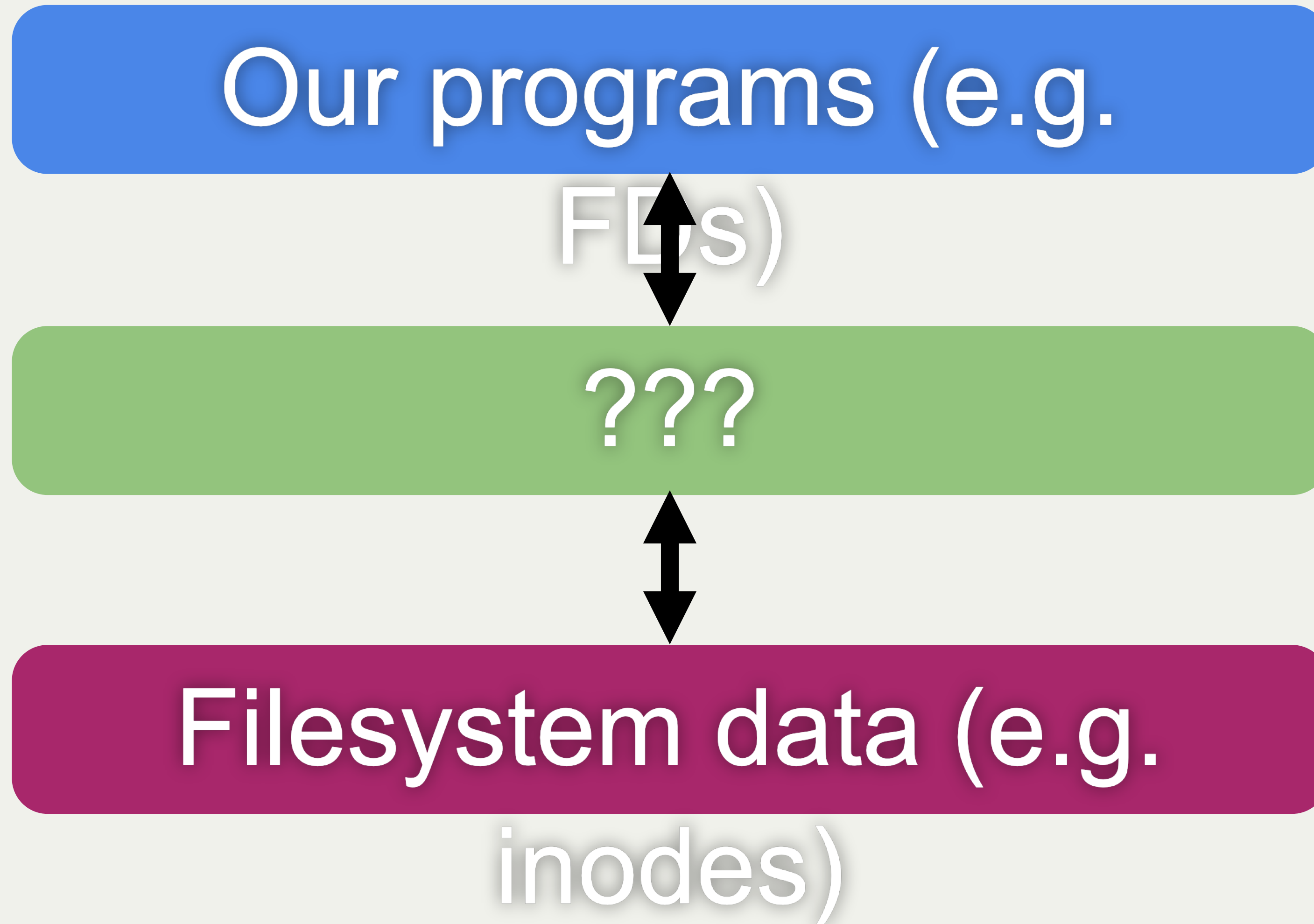
File descriptors are a powerful abstraction for working with files and other resources. They are used for files, networking and user input/output!

Lecture Plan

- Recap: File descriptors, **open()**, **close()**, **read()** and **write()**
- **Operating system data structures**
- How are system calls made?
- Introduction to multiprocessing

What is a file descriptor really mapping to behind the scenes? How does the operating system manage open files?

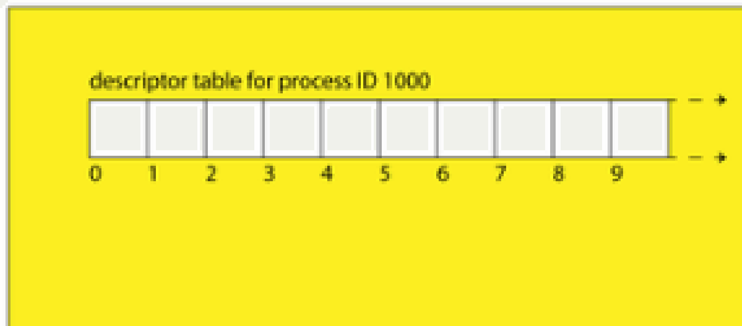
Filesystem Interaction: Layers

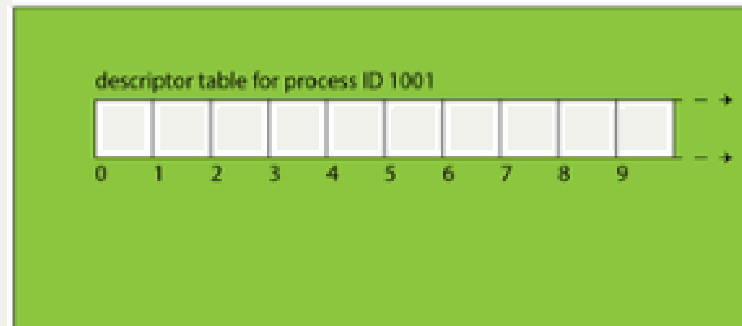


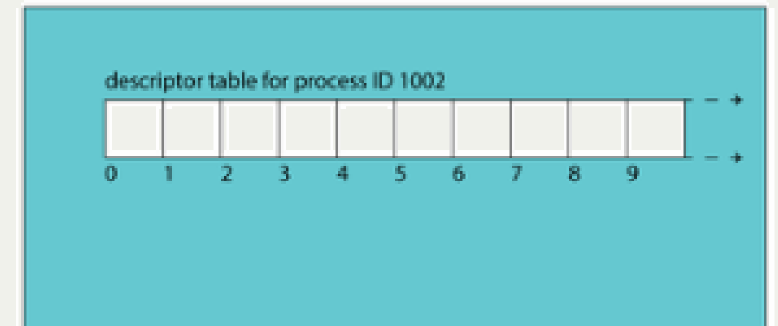
Operating System Data Structures

- A **process** is a single instance of a program running
- For each process, Linux maintains a **process control block** - a set of relevant information about its execution (user who launched it, CPU state, etc.). These blocks live in the **process table**.
- A process control block also stores a **file descriptor table**. This is a list of info about open files/resources for this process.
- **Key idea:** a file descriptor is just **an index into a file descriptor table!**

Process Control Blocks

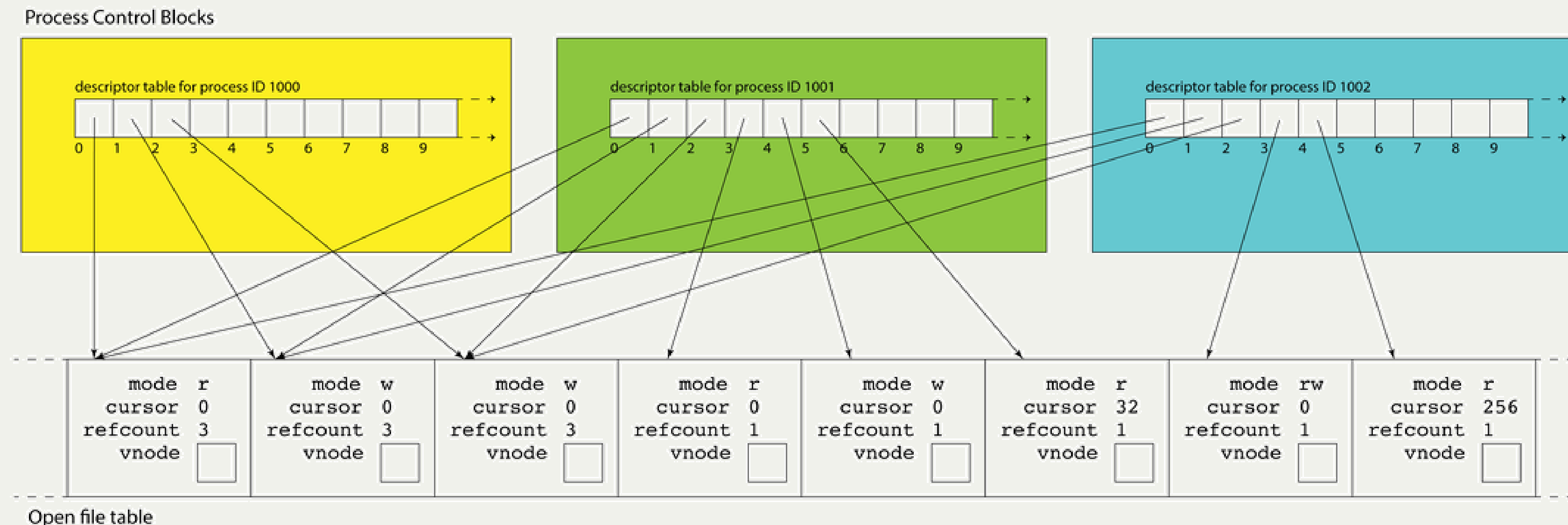






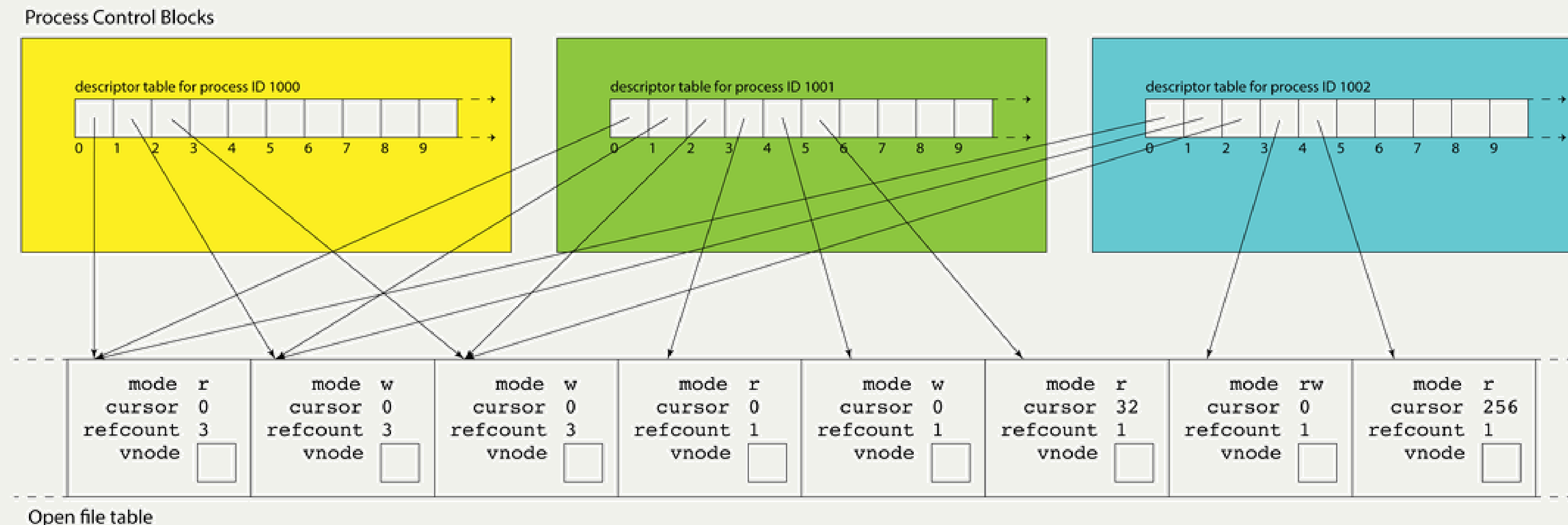
Operating System Data Structures

- 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 across all processes.
- Multiple file descriptor entries (even across processes!) can point to the same open file table entry.
- An open file table entry stores changing info like "cursor" (how far into file are we?)



Operating System Data Structures

- This structure allows the OS to share resources across processes.
- Also explains common behavior: e.g. multiple programs interleaving terminal output. This is because all FDs 0,1,2 usually point to the same open file table entries!
- **Problem:** we could have multiple open file table entries referring to the same file. It seems wasteful to store that file's static info many times....

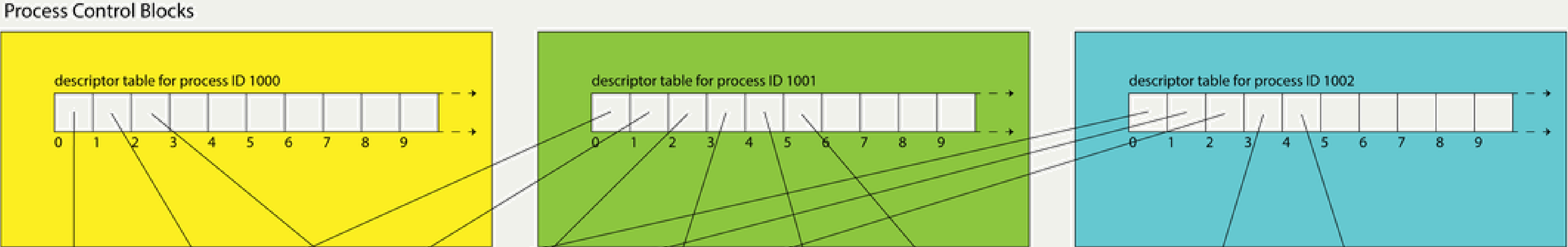


Operating System Data Structures

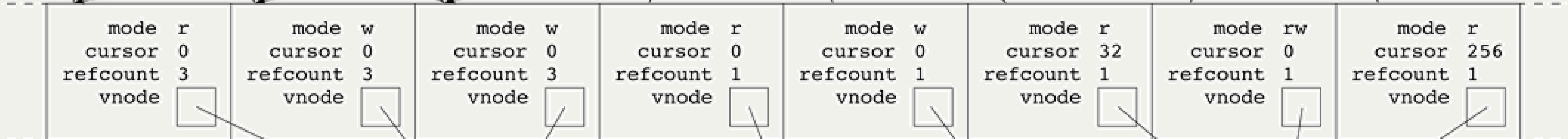
- Each open file table entry has a **vnode** field; a pointer to the file's static information.
- **vnodes** live in the **vnode table**; a single table referenced by all open file table entries. A vnode is an abstraction of a file; it includes information on what kind of file it is, how many file table entries reference it, and function pointers for performing operations. Also cache of inode (if exists).
- These resources are all freed over time:
  - Free a file table entry when the last file descriptor closes it
  - Free a vnode when the last file table entry is freed
  - Free a file when its reference count is 0 and there is no vnode

Operating System Data Structures

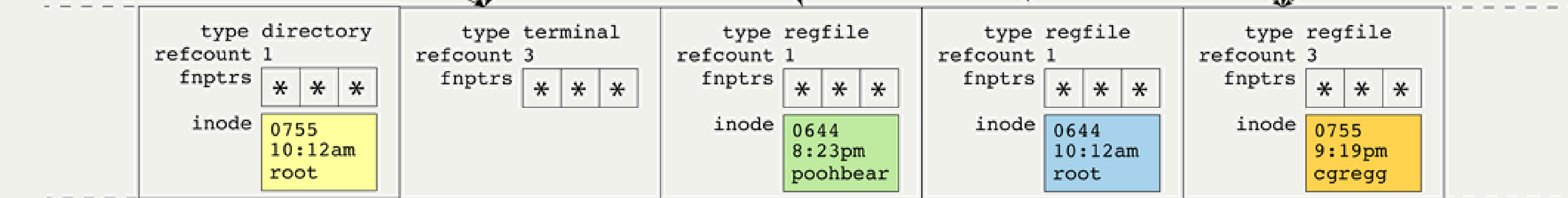
FD table(s)



Open file table

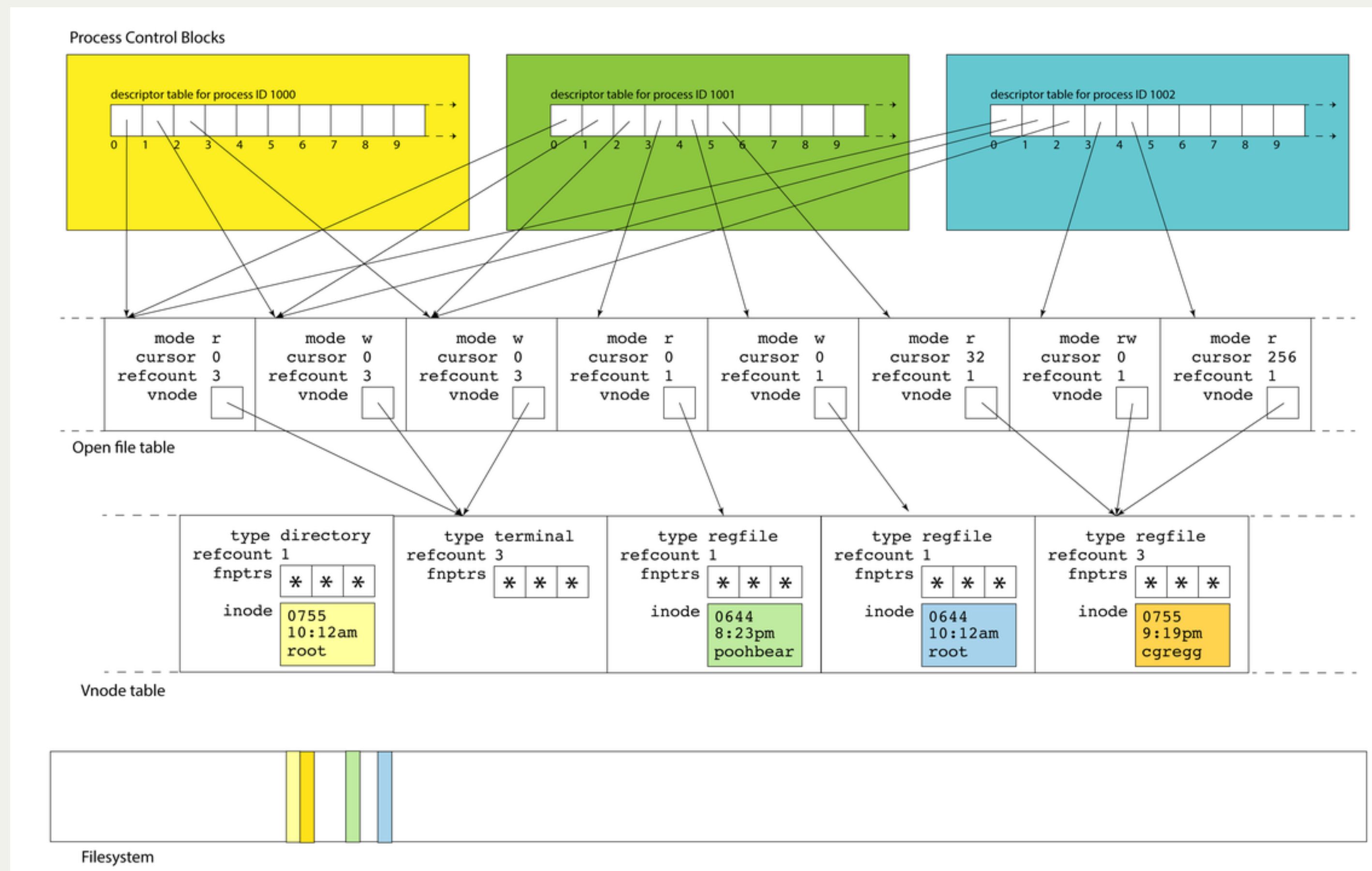


Vnode table



Operating System Data Structures

All of these data structures are private to the operating system. They are layered on top of the filesystem data itself.



Lecture Plan

- Recap: File descriptors, **open()**, **close()**, **read()** and **write()**
- Operating system data structures
- **How are system calls made?**
- Introduction to multiprocessing

System Call Execution

- **Key idea:** the OS performs private, privileged tasks that regular user programs cannot do, with data that user programs cannot access.
- **Problem:** because of this, we can't have system calls behave like regular function calls - there are security risks to having OS data in user-accessible memory!

E.g. *loadFiles* can poke around in *main*'s stack frame, or *main* can poke around in the values left behind by *loadFiles* after it finishes.
Functions are supposed to be modular, but the function call and return protocol's support for modularity and privacy is pretty soft.

Refresher: Function Call Semantics

- **Refresher:** for a normal function call, the stack grows downwards to add a new stack frame. Parameters are passed in registers like %rdi and %rsi, and the return value (if any) is put in %rax.
- This means stack frames are adjacent, and can in theory be manipulated via pointer arithmetic when they're not supposed to
- **Solution:** a range of addresses will be reserved as "kernel space"; user programs cannot access this memory. Instead of using the user stack and memory space, system calls will use kernel space and execute in a "privileged mode". *But this means function calls must work differently!*

System Call Semantics

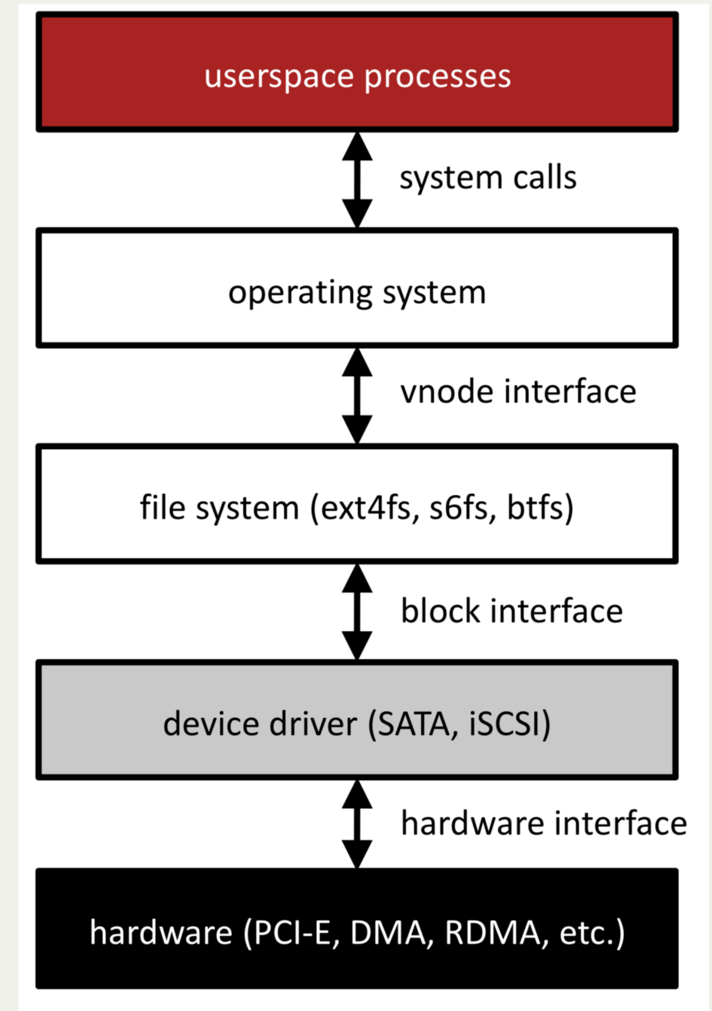
New approach for calling functions if they are system calls:

- put the system call "opcode" in %rax (e.g. 0 for **`read`**, 1 for **`write`**, 2 for **`open`**, 3 for **`close`**, and so forth). Each has its own unique opcode.
- put up to 6 parameters in normal registers *except for %rcx* (use %r10 instead)
- store the address of the next user program instruction in %rcx instead of %rip
- The **syscall** assembly instruction triggers a *software interrupt* that switches execution over to "superuser" mode.
- The system call executes in privileged mode and puts its return value in %rax, and returns (using **iretq**, "interrupt" version of **retq**)
- If %rax is negative, the global **errno** is set to abs(%rax), and %rax is changed to -1.
- The system transfers control back to the user program.

CS110 Topic 1: How can we design filesystems to store and manipulate files on disk, and how can we interact with the filesystem in our programs?

CS110 Filesystems Recap

- User programs interact with the filesystem via *file descriptors* and the *system calls* **open**, **close**, **read** and **write**
- The operating system stores a per-process file descriptor table with pointers to open file table entries containing info about the open files
- The open file table entries point to vnodes, which cache inodes
- inodes are file system representations of files/directories. We can look at an inode to find the blocks containing the file/directory data.
- Inodes can use indirect addressing to support large files/directories
- Key principles: *abstraction*, *layers*, *naming*



Lecture Plan

- Recap: File descriptors, **open()**, **close()**, **read()** and **write()**
- Operating system data structures
- How are system calls made?
- **Introduction to multiprocessing**

CS110 Topic 2: How can our program create and interact with other programs?

Multiprocessing Terminology

Program: code you write to execute tasks

Process: an instance of your program running; consists of program and execution state.

Key idea: multiple processes can run the same program

Process 5621

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     printf("Goodbye!\n");
4     return 0;
5 }
```

Multiprocessing

Your computer runs many processes simultaneously - even with just 1 processor core (how?)

- "simultaneously" = switch between them so fast humans don't notice
- Your program thinks it's the only thing running
- OS *schedules* processes - who gets to run when
- Each process gets a little time, then has to wait
- Many times, waiting is good! E.g. waiting for key press, waiting for disk
- *Caveat*: multicore computers can truly multitask

Playing With Processes

When you run a program from the terminal, it runs in a new process.

- The OS gives each process a unique "process ID" number (PID)
- PIDs are useful once we start managing multiple processes
- **getpid()** returns the PID of the current process

```
1  // getpid.c
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6      pid_t myPid = getpid();
7      printf("My process ID is %d\n", myPid);
8      return 0;
9  }
```

```
$ ./getpid
My process ID is 18814

$ ./getpid
My process ID is 18831
```

fork()

fork() creates a second process that is a **clone** of the firs

```
pid_t fork();
```

Process A

```
1  int main(int argc, char *argv[]) {
2      printf("Hello, world!\n");
3      fork();
4      printf("Goodbye!\n");
5      return 0;
6  }
```

```
$ ./myprogram
```

fork()

fork() creates a second process that is a **clone** of the firs

```
pid_t fork();
```

Process A

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the firs

```
pid_t fork();
```

Process A

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the firs

```
pid_t fork();
```

Process A

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     fork();
4     printf("Goodbye!\n");
5     return 0;
6 }
```

```
$ ./myprogram
Hello, world!
Goodbye!
Goodbye!
```

fork()

fork() creates a second process that is a **clone** of the firs

```
pid_t fork();
```

Process A

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

```
$ ./myprogram2
```

fork()

fork() creates a second process that is a **clone** of the firs

```
pid_t fork();
```

Process A

```
1  int main(int argc, char *argv[]) {
2      int x = 2;
3      printf("Hello, world!\n");
4      fork();
5      printf("Goodbye, %d!\n", x);
6      return 0;
7  }
```

```
$ ./myprogram2
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the firs

```
pid_t fork();
```

Process A

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

```
$ ./myprogram2
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the firs

```
pid_t fork();
```

Process A

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

```
$ ./myprogram2
Hello, world!
Goodbye, 2!
Goodbye, 2!
```

fork()

fork() creates a second process that is a clone of the firs

```
pid_t fork();
```

- **parent** (original) process forks off a **child** (new) process
- The child **starts** execution on the next program instruction. The parent **continues** execution with the next program instruction. *The order from now on is up to the OS!*
- **fork()** is called once, but returns twice (why?)
- Everything is duplicated in the child process
  - File descriptor table (increasing reference counts on open file table entries)
  - Mapped memory regions (the address space)
  - Regions like stack, heap, etc. are copied

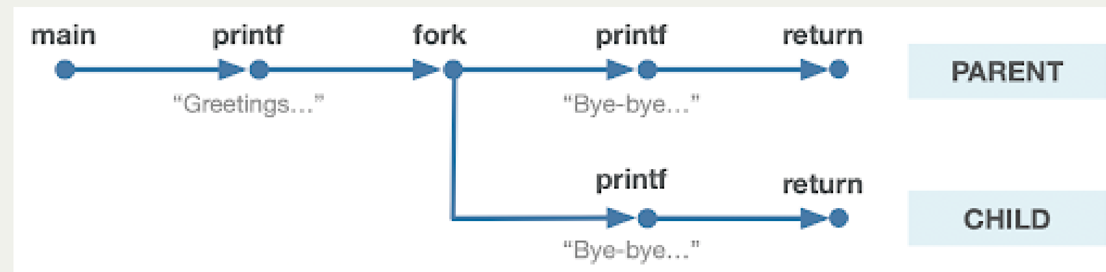


Illustration courtesy of Roz Cyrus.

Process Clones

The parent process' file descriptor table is **cloned** on **fork** and the reference counts within the relevant open file table entries are incremented. This explains how the child can still output to the same terminal!

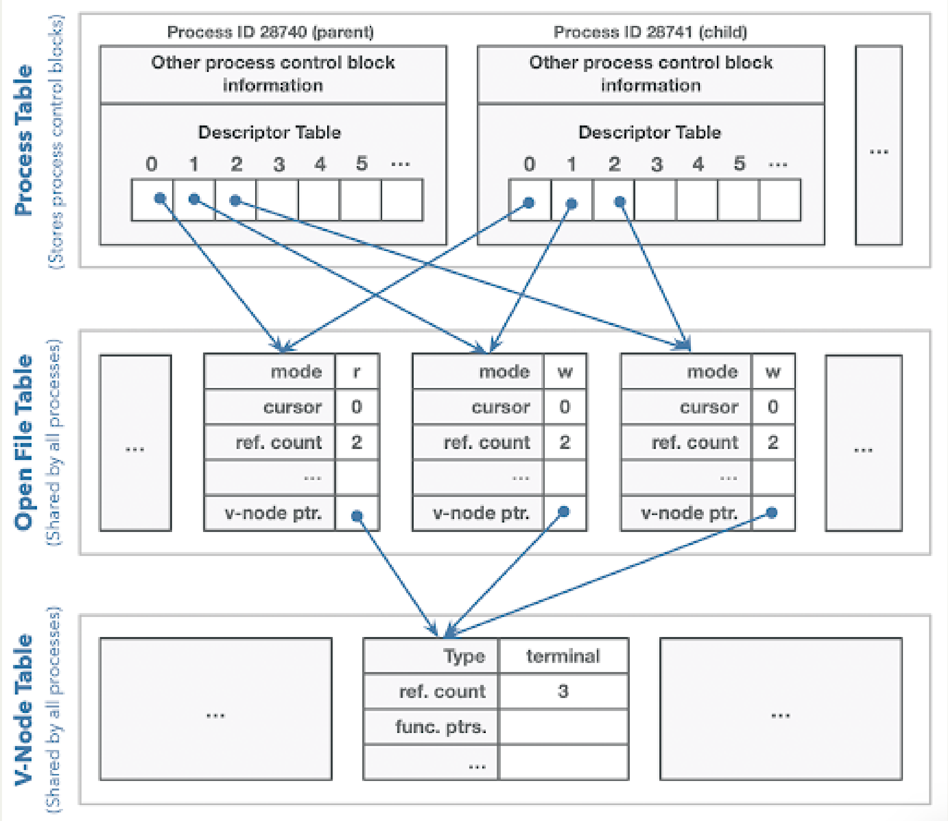


Illustration courtesy of Roz Cyrus.

fork()

(Am I the parent or the child?)

Process A

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

Process B

```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     printf("Hello, world!\n");
4     fork();
5     printf("Goodbye, %d!\n", x);
6     return 0;
7 }
```

Is there a way for the processes to tell which is the parent and which is the child?

Key Idea: **the return value of fork() is different in the parent and the child.**

fork()

fork() creates a second process that is a **clone** of the first:

```
pid_t fork();
```

- **parent** (original) process forks off a **child** (new) process
- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)

fork()

- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork();
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram
```

fork()

- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork();
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram2
Hello, world!
```

fork()

- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 111
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

Process 111

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 0
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram2
Hello, world!
```

fork()

- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 111
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

Process 111

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 0
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram
Hello, world!
fork returned 111
fork returned 0
```

fork()

- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 111
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

Process 111

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 0
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

```
$ ./myprogram
Hello, world!
fork returned 111
fork returned 0
```

OR

```
$ ./myprogram
Hello, world!
fork returned 0
fork returned 111
```

fork()

- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)

Process 110

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 111
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
6 }
```

Process 111

```
1 int main(int argc, char *argv[]) {
2     printf("Hello, world!\n");
3     pid_t pidOrZero = fork(); // 0
4     printf("fork returned %d\n", pidOrZero);
5     return 0;
```

We can no longer assume the order in which our program will execute! The OS decides the order.

```
$ ./myprogram
Hello, world!
fork returned 111
fork returned 0
```

OR

```
Hello, world!
fork returned 0
fork returned 111
```

fork()

- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)
- A process can use **getppid()** to get the PID of its parent
- if **fork()** returns < 0, that means an error occurred

```
1 // basic-fork.c
2 int main(int argc, char *argv[]) {
3     printf("Greetings from process %d! (parent %d)\n", getpid(), getppid());
4     pid_t pidOrZero = fork();
5     assert(pidOrZero >= 0);
6     printf("Bye-bye from process %d! (parent %d)\n", getpid(), getppid());
7     return 0;
8 }
```

```
$ ./basic-fork
Greetings from process 29686! (parent 29351)
Bye-bye from process 29686! (parent 29351)
Bye-bye from process 29687! (parent 29686)

$ ./basic-fork
Greetings from process 29688! (parent 29351)
Bye-bye from process 29689! (parent 29688
Bye-bye from process 29688! (parent 29351)
```

- The parent of the original process is the *shell* - the program that you run in the terminal.
- The ordering of the parent and child output is *nondeterministic*. Sometimes the parent prints first, and sometimes the child prints first!

Recap

- Recap: File descriptors, **open()**, **close()**, **read()** and **write()**
- Operating system data structures
- How are system calls made?
- Introduction to multiprocessing

Next time: more multiprocessing