CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Introduction to UNIX Filesystems

- You should already be familiar with the Linux filesystem as a user. The filesystem uses a tree-based model to store files and directories of files. You can get details of a file in a particular directory with the **`ls`** command

```
poohbear@myth53:~/cs110/lecture-examples/filesystems$ ls
alphabet.txt  copy.c  Makefile  open.c  search.c  t.c  umask.c  vowels.txt
```

- You can get a more detailed listing with the **`ls -al`** command:

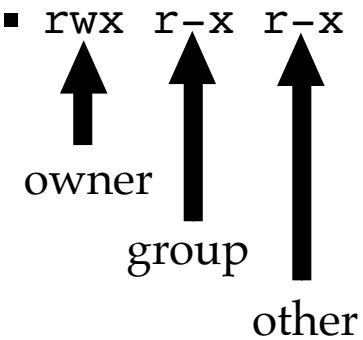
```
poohbear@myth62:~/cs110/lecture-examples/filesystems$ ls -la
total 16
-rw------- 1 poohbear operator   27 Sep 26 21:31 alphabet.txt
-rw------- 1 poohbear operator 1882 Sep 26 21:31 copy.c
-rw------- 1 poohbear operator  631 Sep 26 21:31 Makefile
-rw------- 1 poohbear operator  949 Sep 26 21:31 open.c
-rw------- 1 poohbear operator 2302 Sep 26 21:31 search.c
-rw------- 1 poohbear operator 1321 Sep 26 21:31 t.c
-rw------- 1 poohbear operator  286 Sep 26 21:31 umask.c
-rw------- 1 poohbear operator    6 Sep 26 21:31 vowels.txt
drwxr-xr-x 5 poohbear root     2048 Sep 26 21:29 ..
drwx------ 2 poohbear operator 2048 Sep 26 21:28 .
```

- There are two files listed as directories (**`d`**), "." and "..". These stand for:
  - "." is the *current* directory
  - ".." is the *parent* directory
- The **"rwx------"** designates the permissions for a file or directory, with "r" for read permission, "w" for write permission, and "x" for execute permission.

Introduction to UNIX Filesystems

```
poohbear@myth53:~/cs110/lecture-examples/filesystems$ ls -la search
-rwxr-xr-x 1 poohbear operator 22328 Sep 26 21:32 search
```

- There are actually three parts to the permissions line, each with the three permission types available:
  - `rwx r-x r-x`



In this case, the owner has read, write, and execute permissions, the group has only read and execute permissions, and the user also has only read and execute permissions.

- Because each individual set of permissions can be either r, w, or x, there are three bits of information per permission field. We can therefore, use *base 8* to designate a particular permission set. Let's see how this would work for the above example:
  - permissions: **`rwx r-x r-x`**
  - bits (base 2): **`111 101 101`**
  - base 8: **`7 5 5`**
- So, the permissions for the file would be recorded internally as **`755`**.

Introduction to UNIX Filesystems

- In C, a file can be created using the **open** *system call*, and you can set the permissions at that time, as well. We will discuss the idea of system calls soon, but for now, simply think of them as a function that can do systemsy stuff. The **open** function comes with the following signatures (and this works in C, even though C does not support function overloading! How, you ask? See here.)

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- There are many flags (see `man 2 open` for a list of them), and they can be bitwise `or`'d together. You must include exactly one of the following flags:
  - `O_RDONLY`: read only
  - `O_WRONLY`: write only
  - `O_RDWR`: read and write
- We employ one of the following flags when creating a file in either `O_WRONLY` or `O_RDWR` mode:
  - `O_CREAT`: create the file if it doesn't exist
  - `O_EXCL`: mandate the file be created from scratch, fail if file already exists
- When (and only when) you're creating a new file using `O_CREAT` do you include a third argument. This third argument is used to specific what the new file's permission set should be.

Introduction to UNIX Filesystems

- Today's lecture examples reside within **`/usr/class/cs110/lecture-examples/filesystems`**.
  - The **`/usr/class/cs110/lecture-examples`** directory is a **`git`** repository that will be updated with additional examples as the quarter progresses.
  - To get started, type **`git clone /usr/class/cs110/lecture-examples lecture-examples`** at the command prompt to create your own local copy.
  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type **`git pull`**. Doing so will update your local copy to match whatever the primary has become.

Introduction to UNIX Filesystems

- The program below creates an `"empty"` file, setting its permissions to **0644**:

```
#include <fcntl.h>    // for open
#include <unistd.h>   // for read, close
#include <stdio.h>
#include <sys/types.h> // for umask
#include <sys/stat.h>  // for umask
#include <errno.h>
static const char * const kFilename = "empty";
int main() {
    int fd = open(kFilename, O_WRONLY | O_CREAT | O_EXCL, 0664);
    if (fd == -1) {
        printf("There was a problem creating \"%s\"!\n", kFilename);
        if (errno == EEXIST) {
            printf("The file already exists.\n");
        } else {
            printf("Unknown errno: %d\n", errno);
        }
        return -1;
    }
    printf("Successfully opened the file called \"%s\", and about to close it.\n", kFilenam
    close(fd); // companion system call to open and releases the provided file descriptor
    return 0;
}
```

```
poohbear@myth62:/usr/class/cs110/lecture-examples/filesystems$ ./open
Successfully opened the file called "empty", and about to close it.
poohbear@myth62:/usr/class/cs110/lecture-examples/filesystems$ ./open
There was a problem creating 'empty'!
The file already exists.
poohbear@myth62:/usr/class/cs110/lecture-examples/filesystems$ ls -la empty
-rw-rw-r-- 1 poohbear operator 0 Sep 26 21:39 empty
```

Introduction to UNIX Filesystems

- So far we've seen two file system API calls: **`open`** and **`close`**.
- We need to look at other low-level operations that allow programmers to interact with file systems. We will focus here on the direct system calls, but when writing production code, you'll generally use directives like the **`FILE*`**, **`ifstream`**, and **`ofstream`** abstractions, whose implementations layer over *file descriptors*, calls to **`open`** and **`close`**, and related systems calls like **`read`** and **`write`** which we'll discuss right now.
- Requests to open a file, read from a file, extend the heap, etc., all eventually go through *system calls*, which are the only functions trustworthy enough to interact with the OS on your behalf. The OS *kernel* executes the code of a system call, isolating all system-level interactions from your (potentially buggy and harmful) program.

Implementing copy to emulate cp

- The implementation of **`copy`** (designed to mimic the behavior of **`cp`**) illustrates how to use **`open`**, **`read`**, **`write`**, and **`close`** and what the file descriptors are.
  `man` pages exist for all of these functions (e.g. **`man 2 open`**, **`man 2 read`**, etc.)
  Full implementation of our own **`copy`**, with exhaustive error checking, is right here.
  Simplified implementation, *sans* error checking, is on the next slide.

Back to file systems: Implementing copy

```
1  int main(int argc, char *argv[]) {
2    int fdin = open(argv[1], O_RDONLY);
3    int fdout = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0644);
4    char buffer[1024];
5    while (true) {
6      ssize_t bytesRead = read(fdin, buffer, sizeof(buffer));
7      if (bytesRead == 0) break;
8      size_t bytesWritten = 0;
9      while (bytesWritten < bytesRead) {
10       bytesWritten += write(fdout, buffer + bytesWritten, bytesRead - bytesWritten);
11     }
12   }
13   close(fdin);
14   close(fdout);
15   return 0;
16 }
```

- The **`read`** system call will block until at least one byte is available to be read. If **`read`** returns 0, there are no more bytes to read, presumably because you've reached the end of the file, or the file descriptor was closed.
- If **`write`** returns a value less than the value supplied as the third argument, it means that the system couldn't write all bytes at once, hence the **`while`** loop and the need to keep track of **`bytesRead`** and **`bytesWritten`**.
- You should close file descriptors as soon as you're done with them so that descriptors can be reused on behalf of future **`open`** calls and other syscalls—that's shorthand for system calls—that allocate descriptors. Some systems allow a surprisingly small number of descriptors to be open at any one time, so be sure to close them.

Pros and cons of file descriptors over alternatives

- The file descriptor abstraction provides direct, low-level access to a stream of data without the fuss of higher-level data structures or classes. It certainly can't be slower, and depending on what you're doing, it might be faster.
- **`FILE`** pointers and C++ **`iostream`**s work well when you know you're interacting with standard output, standard input, and local files.
  - They are less useful when the stream of bytes is associated with a network connection, which we'll soon learn is also supported via descriptors.
  - **`FILE`** pointers and C++ **`iostream`**s assume they can, in theory, rewind and move the file pointer back and forth freely, but that's not the case with file descriptors associated with network connections.
- File descriptors, however, work with **`read`** and **`write`**, but little else of use to CS110.
- C **`FILE`** pointers and C++ streams, on the other hand, provide automatic buffering and more elaborate formatting options.

Implementing t to emulate tee

- The **`tee`** program that ships with Linux copies everything from standard input to standard output, making zero or more *extra* copies in the named files supplied as user program arguments.
  - For example, if the file contains 27 bytes—the 26 letters of the English alphabet followed by a newline character—then the following would print the alphabet to standard output and to three files named **`one.txt`**, **`two.txt`**, and **`three.txt`**.

```
$ cat alphabet.txt | tee one.txt two.txt three.txt
abcdefghijklmnopqrstuvwxyz
$ cat one.txt
abcdefghijklmnopqrstuvwxyz
$ cat two.txt
abcdefghijklmnopqrstuvwxyz
$ diff one.txt two.txt
$ diff one.txt three.txt
$
```

- If the file **`vowels.txt`** contains the five vowels and the newline character, and **`tee`** is invoked as follows, **`one.txt`** would be rewritten to contain only the English vowels.

```
$ cat vowels.txt | ./tee one.txt
aeiou
$ cat one.txt
aeiou
```

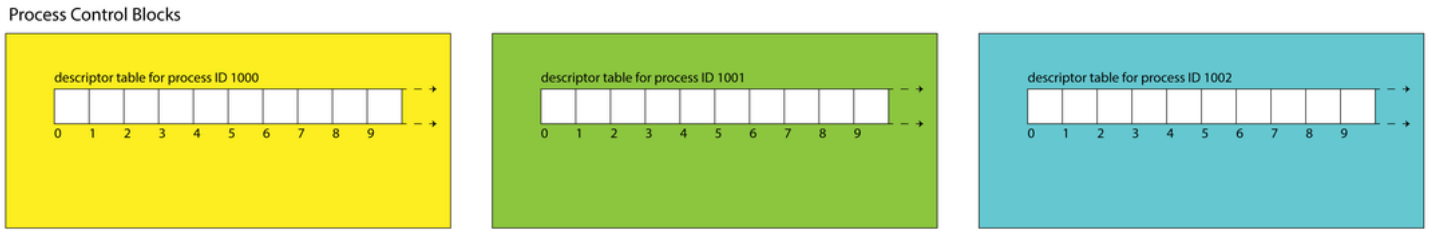
- Full implementation of our own **`t`** executable, with error checking, is right here.
- Implementation replicates much of what **`copy.c`** does, but it illustrates how you can use low-level I/O to manage many sessions with multiple files at the same time. The implementation, but without erroring checking, is presented on the next slide.

Implementing t to emulate tee

```
 1 int main(int argc, char *argv[]) {
 2   int fds[argc];
 3   fds[0] = STDOUT_FILENO;
 4   for (size_t i = 1; i < argc; i++)
 5     fds[i] = open(argv[i], O_WRONLY | O_CREAT | O_TRUNC, 0644);
 6
 7   char buffer[2048];
 8   while (true) {
 9     ssize_t numRead = read(STDIN_FILENO, buffer, sizeof(buffer));
10     if (numRead == 0) break;
11     for (size_t i = 0; i < argc; i++) writeall(fds[i], buffer, numRead);
12   }
13
14   for (size_t i = 1; i < argc; i++) close(fds[i]);
15   return 0;
16 }
17
18 static void writeall(int fd, const char buffer[], size_t len) {
19   size_t numWritten = 0;
20   while (numWritten < len) {
21     numWritten += write(fd, buffer + numWritten, len - numWritten);
22   }
23 }
```

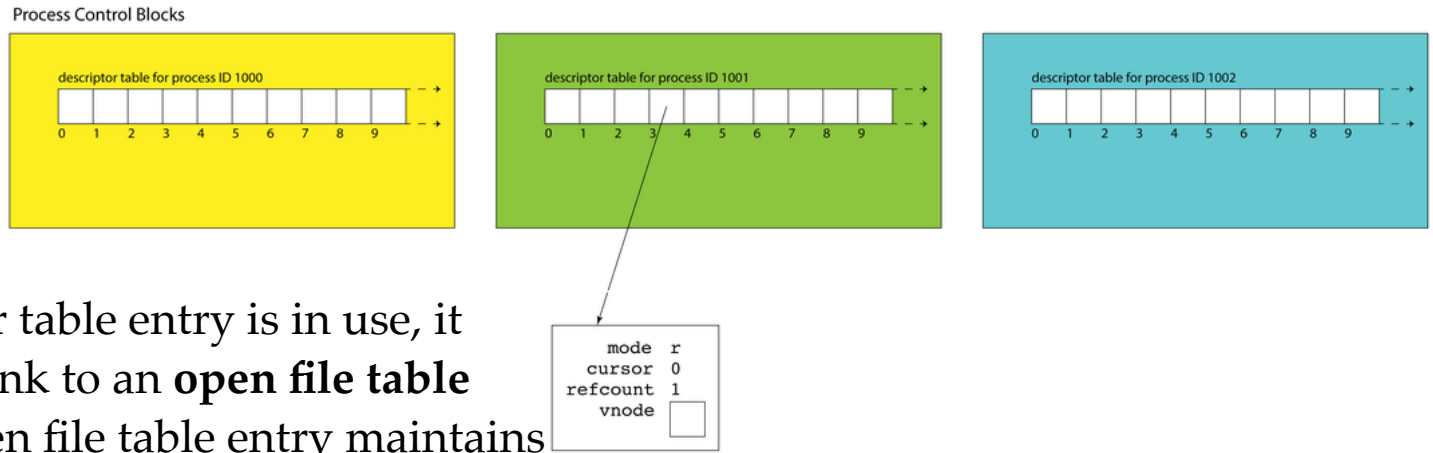
- Note that **`argc`** incidentally equals the number of descriptors we need to write to. That's why we declare an **`int`** array (or rather, a descriptor array) of length **`argc`**.
- **`STDIN_FILENO`** is a built-in constant for the number 0, which is the descriptor normally linked to standard input. **`STDOUT_FILENO`** is a constant for the number 1, which is the default descriptor bound to standard output.

Filesystem Data Structures



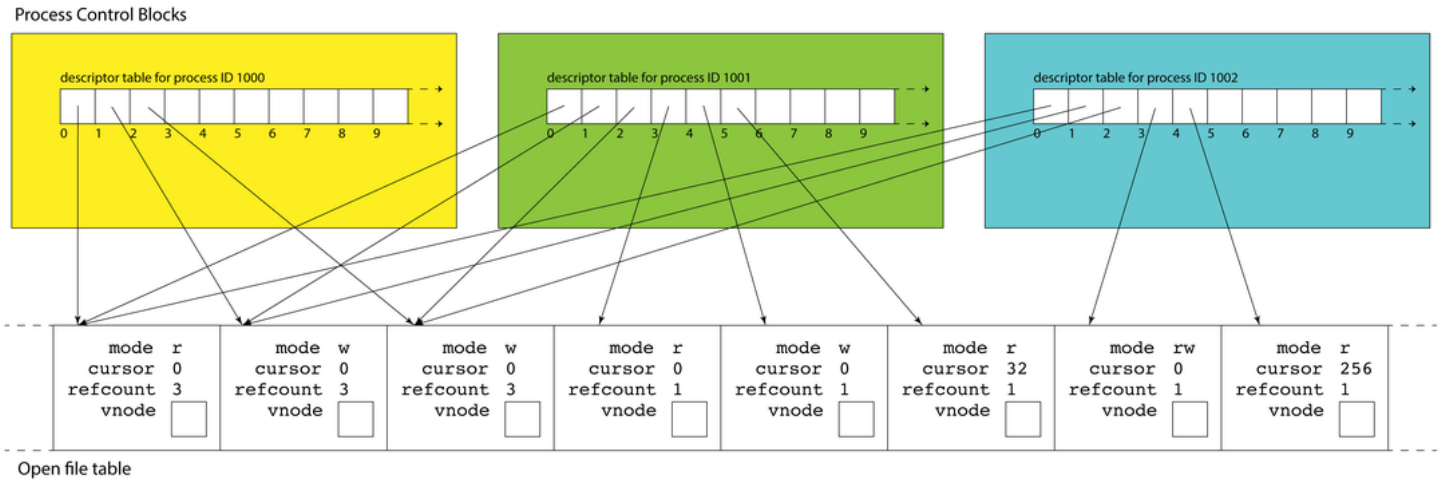
- The OS maintains a data structure for each active process. These data structures are called **process control blocks** and are stored in a **process table**.
- Process control blocks store many things (the user who launched it, the time it was launched, etc.). Among the many items it stores: the **descriptor table**.
- Each process maintains its own set of descriptors. Descriptors 0, 1, and 2 generally refer to standard input, standard output, and standard error, but there are no predefined meanings for descriptors 3 and up. Descriptors 0, 1, and 2 are most often bound to the terminal.
- A user program treats the descriptor as the identifier needed to interact with a resource (most often a file) via **`read`**, **`write`** and **`close`** calls. Internally, that descriptor is an index into the descriptor table.
- The process control block tracks which descriptors are in use and which ones aren't. When allocating a new descriptor for a process, the OS typically chooses the smallest available number in that process's descriptor table.

Filesystem Data Structures



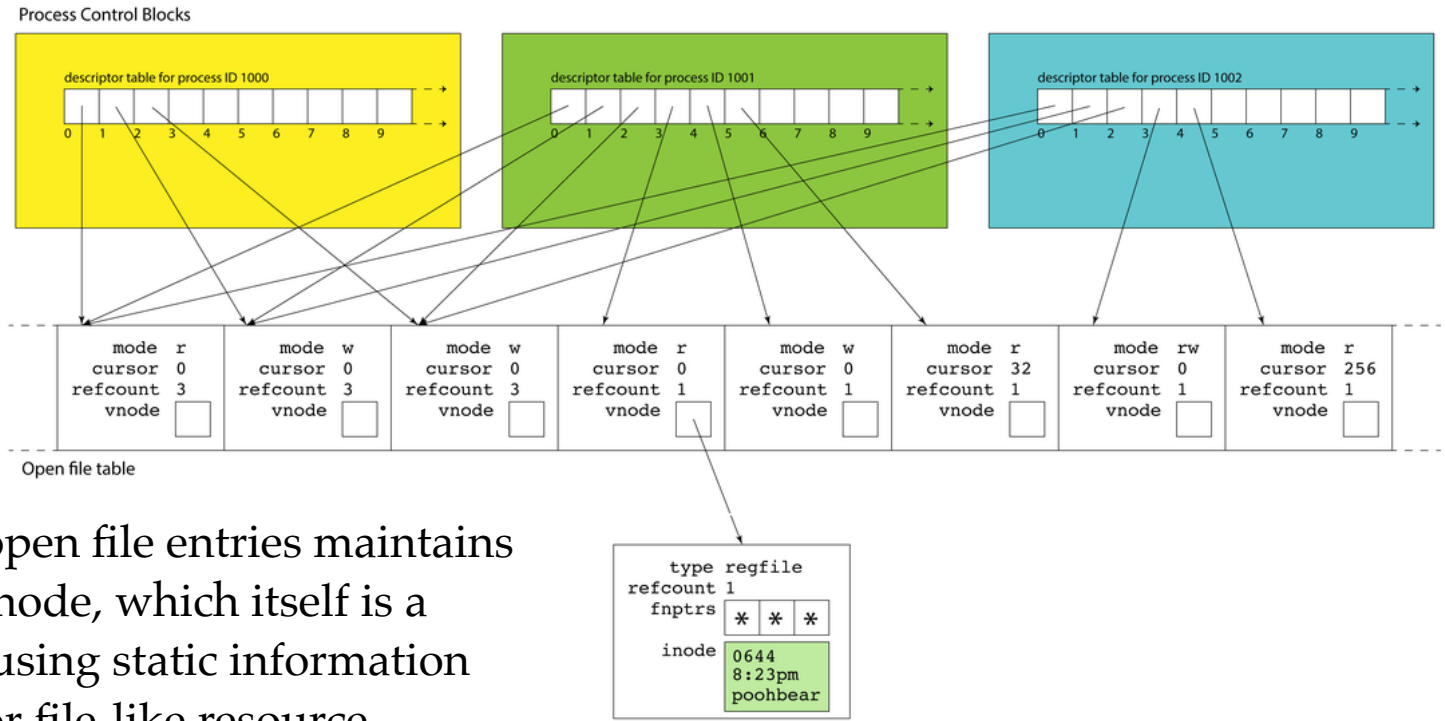
- If a descriptor table entry is in use, it maintains a link to an **open file table entry**. An open file table entry maintains information about an active session with a file (or something that behaves like a file, like terminal, or a network connection).
- Each table entry tracks information specific to the dynamics of that session. **`mode`** tracks whether we're reading, writing, or both. **`cursor`** tracks a position within the file payload. **`refcount`** tracks the number of descriptors across all processes that refer to that entry. (We'll discuss the **`vnode`** field in a moment.)
- The illustration here calls out one file table entry referenced by process 1001, descriptor 3. A call to **`open(filename, O_RDONLY)`** from that process might result in the above.

Filesystem Data Structures



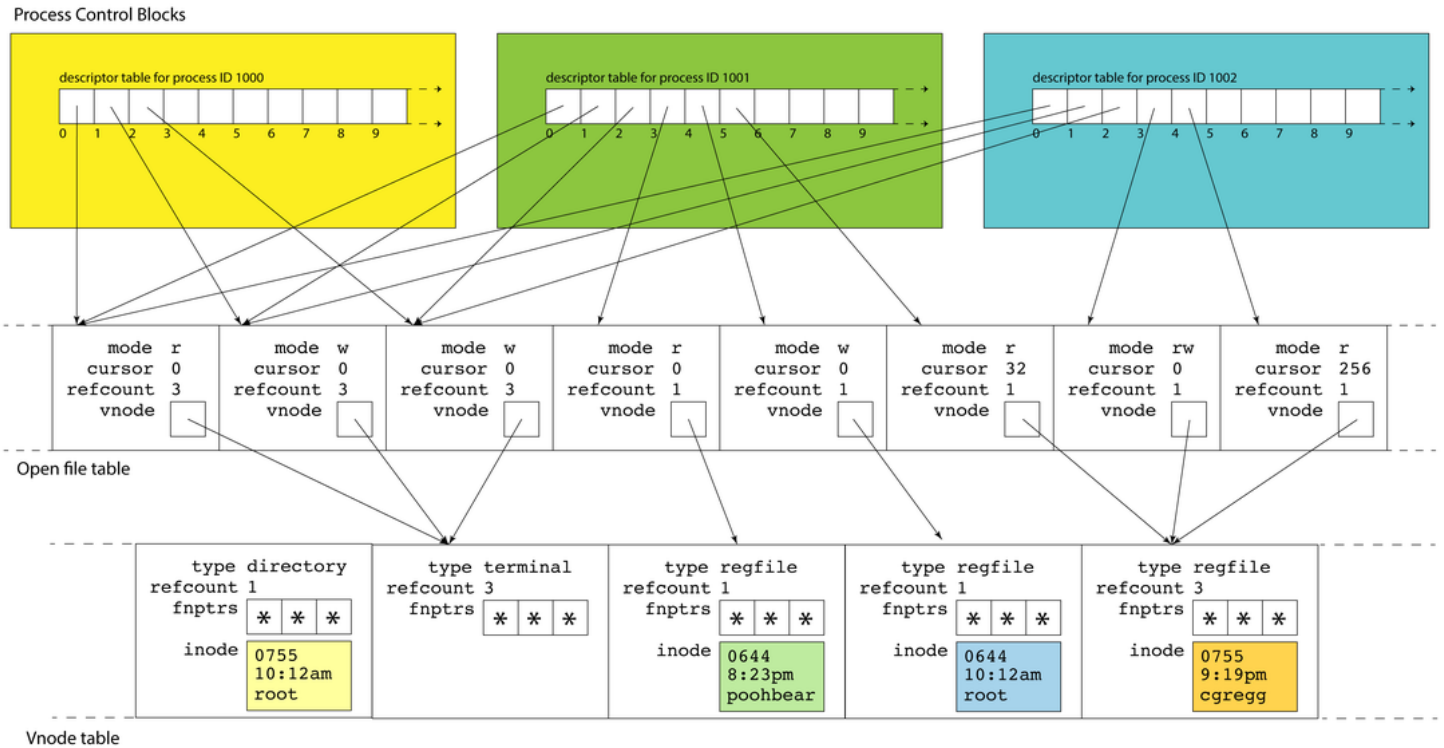
- At any one time, there are multiple active processes, and each typically has at least three open descriptors, and possibly more.
- Each process maintains its own descriptor table, but there is only one, system-wide open file table. This allows for file resources to be shared between processes, and we'll soon see just how common shared file resources really are.
- As drawn above, descriptors 0, 1, and 2 in each of the three PCBs alias the same three sessions. That's why each of the referred table entries have refcounts of 3 instead of 1.
  - This shouldn't surprise you. If your **`bash`** shell calls **`make`**, which itself calls **`g++`**, each of them inserts text into the same terminal window.

Filesystem Data Structures



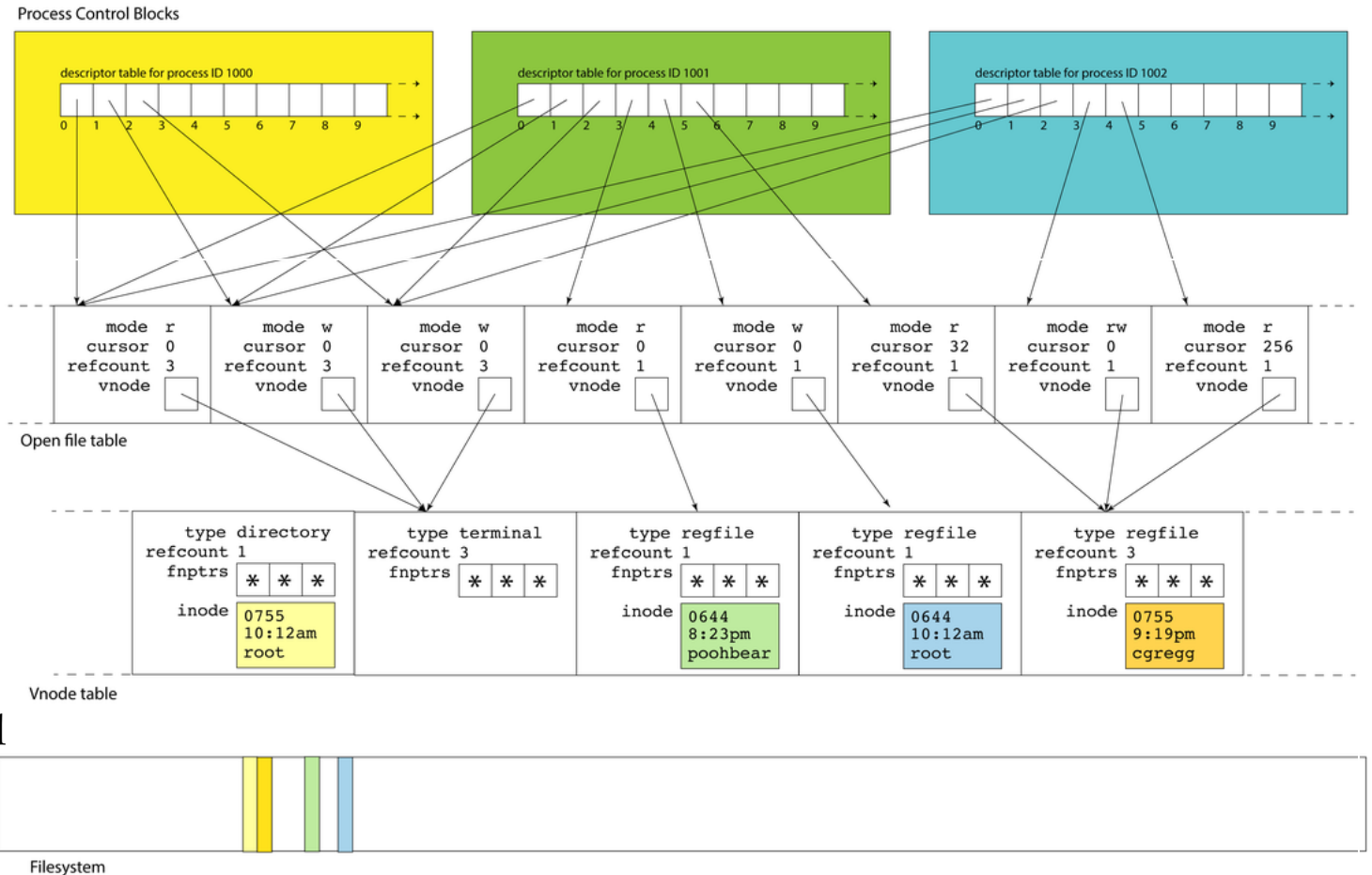
- Each of the open file entries maintains access to a vnode, which itself is a structure housing static information about a file or file-like resource.
- The data structure stores file type (e.g. regular file, directory, symlink, terminal), a refcount, the collection of function pointers that should be used to read, write, and otherwise interact with the resource, and, if applicable, a copy of the inode that resides on the filesystem on behalf of that file. In that sense, the vnode is an **inode cache** that stores information about the file (e.g. file size, owner, permissions, etc.) so that it can be accessed much more quickly.

Filesystem Data Structures



- There is one, system-wide vnode table for the same reason there is one system-wide open file table. Independent file sessions reading from or writing to the same file don't need independent copies of the vnode. They can all alias the same one.

Filesystem Data Structures



None of these kernel-resident data structures are visible to users. Note the filesystem itself is a completely different component, and that filesystem inodes of open files are loaded into vnode table entries. The yellow inode in the vnode is an in-memory replica of the yellow sliver of memory in the filesystem.

Optional: Using stat and lstat to extract file metadata

*I won't be formally covering **stat** in lecture, but I will refer to these in future lectures when **stat** is needed. Still, cool stuff!*

- **`stat`** and **`lstat`** are system calls that populate a **`struct stat`** with information about some named file. The prototypes of the two are:

```
int stat(const char *pathname, struct stat *st);
int lstat(const char *pathname, struct stat *st);
```

- **`stat`** and **`lstat`** operate exactly the same way, except when the named file is a *link*, **`stat`** returns information about the file the link ultimately references, and **`lstat`** returns information about the link itself.
- the **`struct stat`** looks like:

```
struct stat {
  dev_t st_dev;         // id of device containing file
  ino_t st_ino;         // id of data structure on device
  mode_t st_mode;       // mode of file
  // many other fields (file size, create time, etc.)
};
```

- The **`st_mode`** field—which is the only one we'll really pay much attention to—isn't so much a single value as it is a collection of bits encoding multiple pieces of information about file type and permissions. A collection of bit masks and macros can be used to extract information from this **`st_mode`** field.

Implementing search to emulate find

- **search** is our own version of the **find** utility that ships with Linux. Compare the outputs of the following to be clear how search is supposed to work. In each of the two test runs below, an executable—one native to Linux, and a second we'll implement together—is invoked to find all files named **stdio.h** within **/usr/include** or any of its descendant directories.

```
poohbear@myth53$ find /usr/include -name stdio.h -print
/usr/include/x86_64-linux-gnu/bits/stdio.h
/usr/include/stdio.h
/usr/include/bsd/stdio.h
/usr/include/c++/7/tr1/stdio.h
/usr/include/c++/10/tr1/stdio.h
/usr/include/c++/8/tr1/stdio.h
/usr/include/c++/9/tr1/stdio.h
poohbear@myth53$ ./search /usr/include stdio.h
/usr/include/x86_64-linux-gnu/bits/stdio.h
/usr/include/stdio.h
/usr/include/bsd/stdio.h
/usr/include/c++/7/tr1/stdio.h
/usr/include/c++/10/tr1/stdio.h
/usr/include/c++/8/tr1/stdio.h
/usr/include/c++/9/tr1/stdio.h
poohbear@myth53$
```

- Nice! They match!

Implementing search to emulate find

- The following **main** relies on **listMatches**, which we'll implement in a second. The full program, complete with error checks we don't present below, is right here.

```
1  int main(int argc, char *argv[]) {
2    const char *directory = argv[1];
3    struct stat st;
4    stat(directory, &st);
5    if (!S_ISDIR(st.st_mode)) return 0;
6    size_t length = strlen(directory);
7    const char *pattern = argv[2];
8    char path[kMaxPath + 1];
9    strcpy(path, directory);
10   // buffer overflow impossible, directory length <= kMaxPath else stat fails
11   listMatches(path, length, pattern);
12   return 0;
13 }
```

- This is our first example that calls **stat** and **lstat**, each of which extracts information about the named file and populates the **struct stat** supplied by address.
- You'll also note the use of the **S_ISDIR** macro, which examines the upper four bits of the **st_mode** field to determine whether the named file is a directory.
- **S_ISDIR** has a few cousins: **S_ISREG** decides whether a file is a regular file, and **S_ISLNK** decided whether the file is a link.
- Most of what's algorithmically interesting falls under the jurisdiction of this **listMatches** function, which performs a depth-first tree traversal of the filesystem to determine what filenames just happen to match the name of interest.

Implementing search to emulate find

- The implementation of **listMatches** makes use of three library functions to iterate over all files within a directory. Let's play with those before tackling **listMatches**.

```
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

- Here's a relatively straightforward function—not **listMatches**, but something even simpler called **listEntries**—illustrating how these three functions above can be used to print all of the named entries within a supplied directory.
- **opendir** accepts the name of a directory and returns the address of an *opaque iterable* surfacing a series of **dirent**s records via a sequence of **readdir** calls.
  - If **opendir** gets anything other than an accessible directory, it returns **NULL**.
  - Once **de** has surfaced all entries, **readdir** returns **NULL**.

```
static void listEntries(const char *name) {
  struct stat st;
  stat(name, &st);
  if (!S_ISDIR(st.st_mode)) return;
  DIR *dir = opendir(name);
  while (true) {
    struct dirent *de = readdir(dir);
    if (de == NULL) break;
    printf("+ %s\n", de->d_name);
  }
  closedir(dir);
}
```

- The **struct dirent** is only guaranteed to contain a **d_name** field, which stores the entry's name as a C string. **.** and **..** are included in the sequence of named entries.
- **closedir** gets called to dispose of the resources allocated by **opendir**.

Implementing search to emulate find

- We can now leverage everything we've learned to implement **`listMatches`**.

```
static void listMatches(char path[], size_t length, const char *name) {
  DIR *dir = opendir(path);
  if (dir == NULL) return; // it's a directory, but permission to open was denied
  strcpy(path + length++, "/");
  while (true) {
    struct dirent *de = readdir(dir);
    if (de == NULL) break; // we've iterated over every directory entry, so stop
    if (strcmp(de->d_name, ".") == 0 || strcmp(de->d_name, "..") == 0) continue;
    if (length + strlen(de->d_name) > kMaxPath) continue;
    strcpy(path + length, de->d_name);
    struct stat st;
    lstat(path, &st);
    if (S_ISREG(st.st_mode)) {
      if (strcmp(de->d_name, name) == 0) printf("%s\n", path);
    } else if (S_ISDIR(st.st_mode)) {
      listMatches(path, length + strlen(de->d_name), name);
    }
  }
  closedir(dir);
}
```

- Note we brute-force ignore **`.`** and **`..`**, else we're threatened with infinite recursion.
- We use **`lstat`** instead of **`stat`** so we know whether an entry is a link. We ignore all links because, again, we want to avoid infinite recursion.
- If the **`stat`** record identifies something as a regular file, we print the entire path if and only if the entry name matches the name of interest.
- If the **`stat`** record identifies something as a directory, we recursively dip into it to see if any descendents match **`name`**.