CS111, Lecture 5
File Descriptors and System Calls

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Sections 13.1-13.2

😷 masks strongly recommended

Topic 1: Filesystems - How can we design filesystems to manage files on disk, and what are the tradeoffs inherent in designing them? How can we interact with the filesystem in our programs?

CS111 Topic 1: Filesystems

Key Question: *How can we design filesystems to manage files on disk, and what are the tradeoffs inherent in designing them? How can we interact with the filesystem in our programs?*

Filesystems introduction and design → Case study: Unix V6 Filesystem → Filesystem System calls and file descriptors → Crash recovery

Lecture 2 | **Lecture 3-4** | **Today** | **Lectures 6-7**

assign1: implement portions of the Unix v6 filesystem!

Learning Goals

- Evaluate the tradeoffs of the Unix v6 Filesystem design overall
- Learn about the **open**, **close**, **read** and **write** functions that let us interact with files
- Get familiar writing programs that read, write and create files
- Learn what the operating system manages for us so that we can interact with files

Plan For Today

- **Recap**: filesystem design and modern filesystems
- Interacting with the filesystem in user programs
  - System calls
  - **open()** and **close()**
  - **Practice:** creating files
  - **read()** and **write()**
  - **Practice:** copying files

```
cp -r /afs/ir/class/cs111/lecture-code/lect5 .
```

Plan For Today

- **Recap: filesystem design and modern filesystems**
- Interacting with the filesystem in user programs
  - System calls
  - **open()** and **close()**
  - **Practice:** creating files
  - **read()** and **write()**
  - **Practice**: copying files

```
cp -r /afs/ir/class/cs111/lecture-code/lect5 .
```

Recap From Last Time

- Small files have up to 8 direct block numbers stored in their inode
- Large files have up to 7 singly-indirect and 1 doubly-indirect block number stored in their inode
- Directories are "just files" and are layered on top of files. Directories store directory entries, which contain info about each file/folder directly within that directory. **Note:** name is at most 14 bytes, which may not be null terminated!
- The lookup process traverses through each directory in the path until we reach the file we're looking for (or don't find it)

Unix V6 Filesystem

We built layers on top of the low-level **readSector** and **writeSector** to implement a higher-level filesystem. We encountered several design ideas:

- **Modularity** –subdivision of a larger system into a collection of smaller subsystems, which themselves may be further subdivided
- **Layering** –the organization of several modules that interact in some hierarchical manner where each layer typically only opens its interface to the module above it
- **Name resolution** – system resolves human-friendly names (paths) to machine-friendly names (inumbers). Names let us refer to system resources.
- **Virtualization** – making one thing look like another (e.g. disk is just an array of sectors)

Unix V6 Filesystem

The Unix V6 Filesystem is one example of a “multi-level index” filesystem design.

- What are the benefits / drawbacks of the Unix V6 Filesystem design?

<u>Advantages</u>

- Can access all block numbers for a file
- Still supports easy sequential access
- Easy to grow files

Unix V6 Filesystem

The Unix V6 Filesystem is one example of a “multi-level index” filesystem design.

- What are the benefits / drawbacks of the Unix V6 Filesystem design?

Disadvantages

- More steps and disk reads to get block data for large files
- More disk space taken up by metadata
- Upper limit on file size (though if larger than disk, doesn’t matter)
- Size change requires restructuring the inode

Multi-level Indexes

There are many alternative designs that could be used – some alterations you could propose might be:

- What if the block size was different?
- What if inodes stored a different number of block numbers?
- What if the file size scheme (small / large) worked differently?

Example: 4.3 BSD Unix filesystem (evolutionary descendent of V6)

- 4Kb block size
- Inodes store 14 block numbers
- First 12 block numbers always direct, 13th always singly indirect, 14th always doubly indirect (no small vs. large schemes)

Other Filesystem Design Ideas

Larger block size? Improves efficiency of I/O and inodes but worsens internal fragmentation. Generally: challenges with both large and small files coexisting.

One idea: multiple block sizes

- Large blocks are 4KB, *fragments* are 512 bytes (8 fragments fit in a block)
- The last block in a file can be a fragment (0-7 fragments)
- One large block can hold fragments from multiple files
- Get the time efficiency benefit of larger blocks, but the internal fragmentation benefit of smaller blocks (small files can use fragments)

Filesystem Techniques Today

- Filesystem design is a hard problem! Tradeoffs, challenges with large and small files.
- Even larger block sizes (16KB large blocks, 2KB fragments) – disk space cheap, internal fragmentation doesn't matter as much
- Reallocate files as blocks grow – initially allocate blocks one at a time, but when a file reaches a certain size, reallocate blocks looking for large contiguous clusters
- ext4 is a popular current Linux filesystem – you may notice similarities!
- NTFS (replacement for FAT) is the current Windows filesystem
- APFS ("Apple Filesystem") is the filesystem for Apple devices

Plan For Today

- **Recap**: filesystem design and modern filesystems
- **Interacting with the filesystem in user programs**
  - System calls
  - **open()** and **close()**
  - **Practice:** creating files
  - **read()** and **write()**
  - **Practice**: copying files

```
cp -r /afs/ir/class/cs111/lecture-code/lect5 .
```

OS vs. User Mode

- The operating system runs code in a privileged “kernel mode” where it can do things and access data that regular user programs cannot. E.g. only OS can call **readSector**.
- System tracks whether it is in “user mode” or “kernel mode”
- The OS provides public functions that we can call in our user programs – **system calls**. When these functions are called, it switches over to “kernel mode”.

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.
- The operating system handles these tasks because they require special privileges that we do not have in our programs. When a system call runs, it runs in **kernel mode**, and we switch back to user mode when it's done.
- The operating system *kernel* runs the code for a system call, completely isolating the system-level interaction from your (potentially harmful) program.
- We are going to examine the system calls for interacting with files. When writing production code, you will often use higher-level methods that build on these (like C++ streams or FILE *), but let's see how they work!

open()

Call **open** to open a file:

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

- **pathname**: the path to the file you wish to open
- **flags**: a bitwise OR of options specifying the behavior for opening the file
- returns a **file descriptor** representing the opened file, or -1 on error

Many possible flags (see manual page for full list). You must include exactly one of the following flags: **O_RDONLY** (read-only), **O_WRONLY** (write-only), **O_RDWR** (read and write). These say how you will use the file in this program.

Another useful flag: **O_TRUNC** means if the file exists already, truncate (clear) it.

open()

Call **open** to open a file:

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

You can also create a new file if the specified file doesn't exist, by including **O_CREAT** as one of the flags. You must also specify a third **mode** parameter.

- **mode**: the permissions to attempt to set for a created file

open()

Call **open** to open a file:

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

You can also create a new file if the specified file doesn't exist, by including **O_CREAT** as one of the flags. You must also specify a third **mode** parameter.

- **mode**: the permissions to attempt to set for a created file

Another useful flag: **O_EXCL**, which says the file must be created from scratch, and to fail if the file already exists.

Aside: how are there multiple signatures for **open** in C? See here.

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 in this program.
- 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.
- file descriptors are assigned in ascending order (next FD is lowest unused)
- The OS remembers information associated with each of your file descriptors, like where in the file you currently are (if reading/writing)

close()

Call **close** to close a file when you're done with it:

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

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

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

- You can use **valgrind** to check if you forgot to close any files. (--track-fds=yes)

Example: Creating a File (touch)

```c
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

touch.c

Example: Creating a File (touch)

```c
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

Open the file to be written to

touch.c

Example: Creating a File (touch)

```c
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

If the file doesn't exist, create it

touch.c

Example: Creating a File (touch)

```c
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

If it does exist, throw an error

touch.c

Example: Creating a File (touch)

```c
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

If we create a new file, it should have these permissions (don't worry about specifics for now)

touch.c

Example: Creating a File (touch)

```c
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

touch.c

Example: Creating a File (touch)

```
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now            e done with it
  close(fd);
  return 0;
}
```

Specify how we are going to use this file in *this program*

touch.c

Example: Creating a File (touch)

```c
// ./touch newfile.txt
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);

  // If an error occurs, print out an error message
  if (fd == -1) {
    printf("There was a problem creating \"%s\"!\n", argv[1]);
    return 1;
  }

  // Close the file now that we are done with it
  close(fd);
  return 0;
}
```

Specify permissions for everyone on disk if this call creates a new file

touch.c

Plan For Today

- Recap: filesystem design and modern filesystems
- **Interacting with the filesystem in user programs**
  - System calls
  - **open()** and **close()**
  - **Practice:** creating files
  - **read()** and **write()**
  - **Practice**: copying files

```
cp -r /afs/ir/class/cs111/lecture-code/lect5 .
```

read()

Call **read** to 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
- returns -1 on error, 0 if at end of file, or nonzero if bytes were read

Key idea: read may not read all the bytes you ask it to! The return value tells you how many were actually read. E.g. if there aren’t that many bytes, or if interrupted)

Key idea #2: the operating system keeps track of where in a file a file descriptor is reading from. So the next time you read, it will resume where you left off.

`write()`

Call **write** to write bytes to an open file:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- **fd**: the file descriptor for the file you'd like to write to
- **buf**: the memory location storing the bytes that should be written
- **count**: the number of bytes you wish to write from buf
- returns -1 on error, or otherwise the number of bytes that were written

Key idea: write may not write all the bytes you ask it to! The return value tells you how many were actually written. E.g. if not enough space, or if interrupted)

Key idea #2: the operating system keeps track of where in a file a file descriptor is writing to. So the next time you write, it will write to where you left off.

Example: Copy

Let's write an example program **copy** that emulates the built-in **cp** command. It takes in two command line arguments (file names) and copies the contents of the first file to the second.

E.g. **`./copy source.txt dest.txt`**

1. Open the source file and the destination file and get file descriptors
2. Read each chunk of data from the source file and write it to the destination file

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

Example: Copy

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

```
int main(int argc, char *argv[]) {
    int sourceFD = open(argv[1], O_RDONLY);
    int destinationFD = open(argv[2],
         O_WRONLY | O_CREAT | O_EXCL, kDefaultPermissions);

    copyContents(sourceFD, destinationFD);

    close(sourceFD);
    close(destinationFD);
    return 0;
}
```

Example: Copy

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

```
int main(int argc, char *argv[]) {
    int sourceFD = open(argv[1], O_RDONLY);
    int destinationFD = open(argv[2],
         O_WRONLY | O_CREAT | O_EXCL, kDefaultPermissions);

    copyContents(sourceFD, destinationFD);

    close(sourceFD);
    close(destinationFD);
    return 0;
}
```

”create the file to write to, and it must not already exist”

Example: Copy

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

```
void copyContents(int sourceFD, int destinationFD) {
      // Goal: while there’s more data from source, read the next
      // chunk and write it to the destination.
}
```

Example: Copy

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

```
void copyContents(int sourceFD, ir
    char buffer[kCopyIncrement];
    ...
}
```

Read in chunks of **kCopyIncrement** bytes

Example: Copy

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

```
void copyContents(int sourceFD, int destinationFD) {
    char buffer[kCopyIncrement];
    while (true) {
        ssize_t bytesRead = read(sourceFD, buffer, sizeof(buffer));
        if (bytesRead == 0) break;

        ...
    }
}
```

Read a chunk of bytes. It may not be **kCopyIncrement** bytes! If **read** returns 0, there are no more bytes to read.

Example: Copy

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

```
void copyContents(int sourceFD, int destinat
    char buffer[kCopyIncrement];
    while (true) {
        ssize_t bytesRead = read(sourceFD, b
        if (bytesRead == 0) break;
        size_t bytesWritten = 0;
        while (bytesWritten < bytesRead) {
            ...
        }
    }
}
```

Now we write this chunk of bytes to the destination file. We must loop until **write** writes them all.

Example: Copy

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

```
void copyContents(int sourceFD, int destinationFD) {
    char buffer[kCopyIncrement];
    while (true) {
        ssize_t bytesRead = read(sourceFD, bu
        if (bytesRead == 0) break;
        size_t bytesWritten = 0;
        while (bytesWritten < bytesRead) {
            ssize_t count = write(destinationFD, buffer + bytesWritten,
                                  bytesRead - bytesWritten);
            bytesWritten += count;
        }
    }
}
```

Since **write** may write only some of the bytes, we need to just give it the *rest* of the bytes that it hasn't written yet.

Example: Copy

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

```
void copyContents(int sourceFD, int destinationFD) {
    char buffer[kCopyIncrement];
    while (true) {
        ssize_t bytesRead = read(sourceFD, buffer, sizeof(buffer));
        if (bytesRead == 0) break;
        size_t bytesWritten = 0;
        while (bytesWritten < bytesRead) {
            ssize_t count = write(destinationFD, buffer + bytesWritten,
                                  bytesRead - bytesWritten);
            bytesWritten += count;
        }
    }
}
```

Recap

- **Recap**: filesystem design and modern filesystems
- Interacting with the filesystem in user programs
  - System calls
  - **open()** and **close()**
  - **Practice:** creating files
  - **read()** and **write()**
  - **Practice**: copying files

Lecture 5 takeaway: System calls are functions provided by the operating system to do tasks we cannot do ourselves. open, close, read and write are 4 system calls that work via file descriptors to work with files.