CS111, Lecture 7

File Descriptors and System Calls

Optional reading:

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

CS198 Section Leading!

cs198@cs.stanford.edu

cs198.stanford.edu – application due 2/1

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	→	Crash Recovery	→	Filesystem System calls and file descriptors
Lecture 2		**Lecture 3-4**		**Lecture 5-6**		**Lecture 6/Today**

assign2: implement a program that can repair a filesystem after a crash, and explore some of the security and ethical implications of OSes / filesystems.

Learning Goals

- 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

- System calls
- **open()** and **close()**
- **Practice:** creating files
- **read()** and **write()**
- **Practice:** copying files
- More about file descriptors

```
cp -r /afs/ir/class/cs111/lecture-code/lect7 .
```

Plan For Today

- **System calls**
- **open()** and **close()**
- **Practice:** creating files
- **read()** and **write()**
- **Practice:** copying files
- More about file descriptors

```
cp -r /afs/ir/class/cs111/lecture-code/lect7 .
```

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 instance of that file in this program.
- Each program has its own file descriptors
- You can have multiple file descriptors for the same file - every time you call open, you get a new file descriptor.
- 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)

Plan For Today

- System calls
- **open()** and **close()**
- **Practice: creating files**
- **read()** and **write()**
- **Practice:** copying files
- More about file descriptors

```
cp -r /afs/ir/class/cs111/lecture-code/lect7 .
```

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 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

- System calls
- **open()** and **close()**
- **Practice:** creating files
- **read() and write()**
- **Practice:** copying files
- More about file descriptors

```
cp -r /afs/ir/class/cs111/lecture-code/lect7 .
```

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 (will never return 0 but not be at end of file)

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

<u>**`copy-soln.c`**</u> 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.

```
static const int kDefaultPermissions = 0644;

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.

```
static const int kDefaultPermissions = 0644;

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, in
    char buffer[kCopyIncrement];
    ...
}
```

Read in chunks of **kCopyIncrement** bytes (arbitrary amount)

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

        ...
    }
}
```

Cool behavior: the next time through the loop when we call **read**, it will automatically read the next chunk of bytes from the file!

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

Cool behavior: each time through the loop, **write** knows where we left off writing in the file from before. However, it doesn't know *what* to write – we must do pointer arithmetic to specify that.

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;
        }
    }
}
```

Plan For Today

- System calls
- **open()** and **close()**
- **Practice:** creating files
- **read()** and **write()**
- **Practice:** copying files
- **More about file descriptors**

```
cp -r /afs/ir/class/cs111/lecture-code/lect7 .
```

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

File Descriptors and I/O

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

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).

Example: Copy

What is the smallest 1 line change/hack we could make to this code to make it print the contents of the source file to the terminal instead of copying it to the destination file?

```
static const int kDefaultPermissions = 0644;
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;
}
```

Respond on PollEv: pollev.com/cs111
or text CS111 to 22333 once to join.

How can we modify the copy program to print to the terminal instead of copying to the destination file?

Nobody has responded yet.

Hang tight! Responses are coming in.

Example: Copy

What is the smallest 1 line change/hack we could make to this code to make it print the contents of the source file to the terminal instead of copying it to the destination file?

```
static const int kDefaultPermissions = 0644;
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, STDOUT_FILENO);

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

Example: Copy

What is the smallest 1 line change/hack we could make to this code to make it print the contents of the source file to the terminal instead of copying it to the destination file?

```
static const int kDefaultPermissions = 0644;
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, STDOUT_FILENO);

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

Recap

- System calls
- **open()** and **close()**
- **Practice:** creating files
- **read()** and **write()**
- **Practice:** copying files
- More about file descriptors

Next time: introduction to multiprocessing

Lecture 7 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.

```
cp -r /afs/ir/class/cs111/lecture-code/lect7 .
```