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
https://docs.google.com/presentation/d/1jzb7xIxI0eTClhT839XT2MYUBeEETF6D80ViJ6ZP1Tw/edit?usp=sharing
cs198.stanford.edu – application due 1/26

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** | **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.

Crash Recovery

- The block cache can store any kind of block – inode block, block from the log, singly-indirect block, directory entries, etc.
- We must write the log entry for an operation to disk before writing the operation to disk. One way to do this is to remember dependencies in the block cache between the operation's blocks and their log entry blocks, to make sure to write the log entry first.
- For a transaction, we want to write all the transaction's log entries before doing any of its operations
- Logging doesn't guarantee that everything is preserved, but it does guarantee that what's there is consistent (separates *durability* – data will be preserved – from *consistency* – state is consistent)
- A chance we crash even mid-operation (e.g. mid-writing a single block)

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

```
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

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

System Calls and Kernel vs. User Mode

System calls are public functions implemented by the operating system that you can use in your program.

- System calls do privileged tasks – tasks we cannot do on our own (e.g. call **readSector**). OS can run code in privileged "kernel mode" where it can do things regular programs cannot. User programs run in regular "user mode".
  - System switches to "kernel mode" when system call runs, switches back to "user mode" after
- The operating system *kernel* runs the code for a system call, completely isolating the system-level interaction from the (potentially harmful) user 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**: path to open
- **flags**: 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).

- **Must have exactly 1 of**: **O_RDONLY** (read-only), **O_WRONLY** (write-only), **O_RDWR** (read and write). These say how you will use the file in this program.

Example: Opening a File

```
// open file to read from
... = open("filetoread.txt", O_RDONLY);

// open file to write to
... = open("filetowrite.txt", O_WRONLY);

// open file to write to and also clear any existing contents
... = open("filetowrite.txt", O_WRONLY | O_TRUNC);
```

*(**O_TRUNC** - 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, e.g. 0644 (octal!)

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

Example: Opening a File

```
// 0644 means "I can read/write the file, others can only read it"

// open file to write to and create if it doesn’t exist
... = open("filetowrite.txt", O_WRONLY | O_CREAT, 0644);

// open file to write to, which must not already exist
... = open("newfiletowrite.txt", O_WRONLY | O_CREAT | O_EXCL, 0644);
```

*(**O_EXCL** (pairs with **O_CREAT)** - the file must be created from scratch. Fail if the file already exists.)*

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.
- When you wish to refer to the file (e.g. read from it, write to it) you must provide the file descriptor.
- 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.
- The OS remembers information associated with each of your file descriptors, like where in the file you currently are (if reading/writing). E.g. separate locations in the file for each file descriptor.
- file descriptors are assigned in ascending order (next FD is lowest unused)

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.
- Returns: 0 on success, -1 on error (we usually won’t error-check **close**)

Important to close files when 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

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

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, we are done with it (no error checking)
  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, we are done with it (no error checking)
  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, we are done with it (no error checking)
  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, we are done with it (no error checking)
  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, we are done with it (no error checking)
  close(fd);
  return 0;
}
```

If we create a new file, it should have these permissions

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, we are done with it (no error checking)
  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, we [illegible]ith it (no error checking)
  close(fd);
  return 0;
}
```

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

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, we are done with it (no err
  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

```
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! This is not necessarily an error – e.g. if there aren't that many bytes, or if interrupted. The return value tells you how many were read. If we must have all bytes, we can call **read** more.

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 (nonzero assuming count > 0)

Key idea: write may not write all the bytes you ask it to! This is not necessarily an error – e.g. if not enough space, or if interrupted. The return value tells you how many were written. If we must write all bytes, we can call **write** more.

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

(*note: we won't worry as much about error-checking open/close/read/write, but full version here includes error checking).*

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

Example: Copy

The **copy** program 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);
    if (destinationFD == -1) {
        printf("%s: destination file could not be created.\n", argv[2]);
        return 1;
    }

    copyContents(sourceFD, destinationFD);

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

Example: Copy

The **copy** program 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);
    if (destinationFD == -1) {
        printf("%s: destination file could not be created.\n", argv[2]);
        return 1;
    }

    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.

```c
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 (With Error)

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

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

Example: Copy (With Error)

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

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

Loop through the source file one chunk at a time – for each chunk, write it to the destination file.

Example: Copy (With Error)

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

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

Read a chunk of at most **kCopyIncrement** (arbitrary amount) bytes at a time.

Example: Copy (With Error)

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

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

Read a chunk of bytes. If **read** returns 0, there are no more bytes to read.

Example: Copy (With Error)

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

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

Now we write this chunk of bytes to the destination file.

Example: Copy (With Error)

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

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

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 (With Error)

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

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

Problem: write may not actually write all **bytesRead** bytes! This could cause us to not transfer over all bytes.

Fix: call **write** in a loop, keep calling it until it writes all bytes.

Example: Copy (With Error)

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

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

Observation: read may not actually read **kCopyIncrement** bytes. Should we wrap the call to **read** in a loop as well?
Not necessary here: chunk size doesn't really matter - we'll get remaining bytes next time through the loop.

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

We cannot just call **write**, as it may not write all the bytes in one go – we will make a helper that calls **write** in a loop until all these bytes are written.

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) { … }

void writeAllBytes(int destinationFD, char buf[], int nbytes) {
    size_t bytesWritten = 0;
    while (bytesWritten < nbytes) {
            ...
    }
}
```

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) { … }

void writeAllBytes(int destinationFD, char buf[], int nbytes) {
    size_t bytesWritten = 0;
    while (bytesWritten < nbytes) {
        ssize_t count = write(...);
        bytesWritten += count;
    }
}
```

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) { … }

void writeAllBytes(int destinationFD, char buf[], int nbytes) {
    size_t bytesWritten = 0;
    while (bytesWritten < nbytes) {
        ssize_t count = write(destinationFD, buffer + bytesWritten,
                              nbytes - 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) { … }

void writeAllBytes(int destinationFD, char buf[], int nbytes) {
    size_t bytesWritten = 0;
    while (bytesWritten < nbytes) {
        ssize_t count = write(destinationFD, buffer + bytesWritten,
                              nbytes - 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) { … }

void writeAllBytes(int destinationFD, char buf[], int nbytes) {
    size_t bytesWritten = 0;
    while (bytesWritten < nbytes) {
        ssize_t count = write(destinationFD, buffer + bytesWritten,
                              nbytes - bytesWritten);
        bytesWritten += count;
    }
}
```

Example: Copy

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

Example: Copy

Would it also work if we used **sizeof(buffer)** like below?

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

Respond on PollEv:
pollev.com/cs111

Would using sizeof(buffer) in this way also work?

Yes - we are reading bytes into the buffer, and we want to write the entire contents of the buffer each time

0%

No - read may not read enough bytes to fill the whole buffer, and therefore we shouldn't write all bytes in the buffer

0%

No - read may read more bytes than fit in the buffer, and therefore sizeof(buffer) is not the right amount

0%

No - we should use (sizeof(buffer) - bytesRead) to ensure the correct number of bytes are written each time

0%

Recap

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

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.

Next time: introduction to multiprocessing

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