

CS111, Lecture 6

Crash Recovery + Filesystem System Calls

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 14
through 14.1

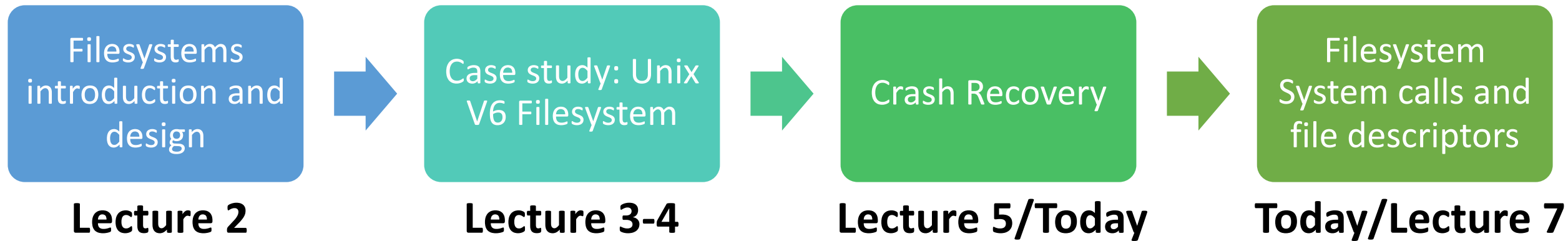
This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under
Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides and notes created by John Ousterhout, Jerry Cain, Chris Gregg, and others.

NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared,
uploaded, or distributed. (without expressed written permission)

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?*



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

- Understand the write-ahead logging approach to crash recovery
- Compare and contrast different crash recovery approaches
- Become familiar writing programs that read, write and create files

Plan For Today

- **Recap:** Crash Recovery so far
- Approach #3: Write-Ahead Logging (“Journaling”)
- Interacting with the filesystem in user programs
 - System calls
 - **open()** and **close()**
 - **Practice:** creating files
 - **read()** and **write()**
 - **Practice:** copying files

Plan For Today

- **Recap: Crash Recovery so far**
- Approach #3: Write-Ahead Logging (“Journaling”)
- Interacting with the filesystem in user programs
 - System calls
 - **open()** and **close()**
 - **Practice:** creating files
 - **read()** and **write()**
 - **Practice:** copying files

Free List and Block Cache

Filesystems commonly use a **bitmap** to track free blocks (1 bit / block , 1 = free, 0 = used).

- During allocation, search bit map for block close to previous block in file
 - Want *locality* – data likely used next is close by (linked list not as good)
- To ensure a minimum number of free blocks, we can pretend the disk is fuller than it actually is (!)

Many OSes have a block cache that stores recently-accessed disk blocks.

- Helps us avoid always having to read a block from disk – check cache first
- “least recently used” one way to manage cache – if full, kick out least recently used block

Crash Recovery

Challenge #1 – data loss: crashes can happen at any time, and not all data might have been saved to disk.

- E.g. if you saved a file but it hadn't actually been written to disk yet.

Challenge #2 - inconsistency: Crashes could happen even in the middle of operations, and this could leave the disk in an inconsistent state.

- E.g. adding block to file: inode was written to store block number, but block wasn't marked in the filesystem as used (it's still listed in the free list)

Approach #1: fsck

Idea #1: don't make any design changes to the filesystem structure to implement crash recovery. Instead, let's write a program that runs on bootup to check the filesystem for consistency and repair any problems it can.

Example: Unix **fsck** ("file system check")

- Runs when we reboot after a crash - scan metadata, identify inconsistencies, repair them, with goal of restoring consistency, minimizing info loss.
- Downsides:
 - Time – can't restart system until **fsck** completes
 - Restores consistency, but other limitations – e.g. can't prevent info loss, system may be consistent but unusable (e.g. system files moved to lost+found), security issues migrating a block to another file

Approach #2: Ordered Writes

Idea #2: Make design changes to the filesystem structure to implement crash recovery. Specifically, prevent certain kinds of inconsistencies by making updates in a particular order.

In general:

- Always initialize target before initializing new reference (e.g. initialize inode before adding directory entry to it)
- Never reuse a resource (inode, disk block, etc.) before nullifying all existing references to it (e.g. adding block to free list)
- To preserve data, never clear last reference to a live resource before setting new reference, preserving data so you don't lose it (e.g. moving a file)

Result: eliminate the need to wait for **fsck** on reboot!

Ordered Writes

Downside #1: *performance*. Forces synchronous metadata writes in the middle of operations, partially defeating the point of the block cache.

- **Example:** adding block to file (update free list + inode) – we must write free list to disk before adding block to inode
- *Improvement:* instead of synchronous writes, track dependencies in the block cache for later. **Tricky – circular dependencies possible! (A -> B -> C -> A)**

Downside #2: can leak resources (e.g. free block removed from free list but never used)

- *Improvement:* run **fsck** in the background to reclaim leaked resources (**fsck** can run in background because filesystem is repaired, but resources have leaked)

Can we do better? E.g., can we avoid leaking data?

Plan For Today

- **Recap:** Crash Recovery so far
- **Approach #3: Write-Ahead Logging (“Journaling”)**
- Interacting with the filesystem in user programs
 - System calls
 - **open()** and **close()**
 - **Practice:** creating files
 - **read()** and **write()**
 - **Practice:** copying files

Write-Ahead Logging (Journaling)

Let's keep a "paper trail" of disk operations that we can revisit in the event of a crash.

- Have an append-only log on disk that stores information about disk operations
- Before performing an operation, record its info in the log, and write that to disk *before* doing the operation itself ("write-ahead")
 - E.g. "I am adding block 4267 to inode 27, index 5"
- Then, the actual block updates can be carried out later, in any order
- If a crash occurs, replay the log to make sure all updates are completed on disk. Thus, we can detect/fix inconsistencies without a full disk scan.

Write-Ahead Logging (Journaling)

- Typically we only log *metadata operations*, not actual file data operations (data is much more expensive, since much more written to log). Tradeoff!
- Most modern filesystems do some sort of logging (e.g. Windows NTFS) – many allow choice whether you want data logging or not.
- Logs one of the most important data structures used in systems today

assign2 Log Example

[offset 33562846]

* LSN 1838326418

LogBlockAlloc

blockno: 1027

zero_on_replay: 0

[offset 33562862]

* LSN 1838326419

LogPatch

blockno: 8

offset_in_block: 136

bytes: 0304

inode #52 (i_addr[0] = block pointer 1035)

Write-Ahead Logging (“Journaling”)

What are the downsides/limitations of our logging design so far?

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



What are the downsides/limitations of our logging design so far?

Nobody has responded yet.

Hang tight! Responses are coming in.

Write-Ahead Logging (“Journaling”)

Problem: log can get long!

Solution: occasional “checkpoints” – truncate the log occasionally once we confirm that portion of the log is no longer needed.

Problem: could be multiple log entries for a single “operation” that should happen atomically.

Solution: have a log mechanism to track “transactions” (atomic operations) and only replay those if the entire transaction is fully entered into the log. (assign2 wraps each transaction with LogBegin and LogCommit)

Problem: we could replay a log operation that has already happened.

Solution: make all log entries *idempotent* (doing multiple times has same effect as doing once). E.g. “append block X to file” (bad) vs. “set block number X to Y”

Write-Ahead Logging (“Journaling”)

Problem: log entries must be written synchronously before the operations

Solution: delay writes for log, too (i.e. build log, but don't write immediately; when a block cache block is written, write relevant log entries then). Though this risks losing some log entries.

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)

Crash Recovery

Ultimately, tradeoffs between *durability*, *consistency* and *performance*

- E.g. if you want durability, you're going to have to sacrifice performance
- E.g. if you want highest performance, you're going to have to give up some crash recovery capability
- What kinds of failures are most important to recover from, and how much are you willing to trade off other benefits (e.g. performance)?

Still lingering problems – e.g. disks themselves can fail

Crash Recovery

We've discussed 3 main approaches to crash recovery:

- 1. Consistency check on reboot (fsck)** – no filesystem changes, run program on boot to repair whatever we can. But can't restore everything and may take a while.
- 2. Ordered Writes** – modify the write operations to always happen in particular orders, eliminating various kinds of inconsistencies. But requires doing synchronous writes or tracking dependencies and can leak resources.
- 3. Write-Ahead Logging** – log metadata (and optionally file data) operations before doing the operations to create a paper trail we can redo in case of a crash.

assign2

- Assign2 tools let you simulate real filesystems, make them crash, and experiment with recovery tools
- Implement a program that replays a log after a crash
- Mix of filesystem exploration (playing around with simulated filesystems, viewing logs and filesystem state) and coding (about ~10-15 lines total)
- Also kicks off embedded ethics discussions about OS trust and security
- You'll have a chance to play with these tools in the assignment and in section

Plan For Today

- **Recap:** Crash Recovery so far
- Approach #3: Write-Ahead Logging (“Journaling”)
- **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/lect6 .
```

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`)

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)

```
// ./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)

```
// ./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)

```
// ./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)

```
// ./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)

```
// ./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 that we are 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)

```
// ./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



Plan For Today

- **Recap:** Crash Recovery so far
- Approach #3: Write-Ahead Logging (“Journaling”)
- **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/lect6 .
```

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.

```
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, int destFD,
    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 destinationFD) {
    char buffer[kCopyIncrement];
    while (true) {
        ssize_t bytesRead = read(sourceFD, buffer, kCopyIncrement);
        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, buffer, kCopyIncrement);
        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;
        }
    }
}
```

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;
}
```

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

- **Recap:** Crash Recovery so far
- Approach #3: Write-Ahead Logging (“Journaling”)
- Interacting with the filesystem in user programs
 - System calls
 - **open()** and **close()**
 - **Practice:** creating files
 - **read()** and **write()**
 - **Practice:** copying files

Next time: more about filesystem system calls

Lecture 6 takeaways: There are various ways to implement crash recovery, such as logging, each with tradeoffs between durability, consistency and performance. System calls like **open** and **close** are functions provided by the operating system to do tasks we cannot do ourselves.