CS111, Lecture 6
Crash Recovery

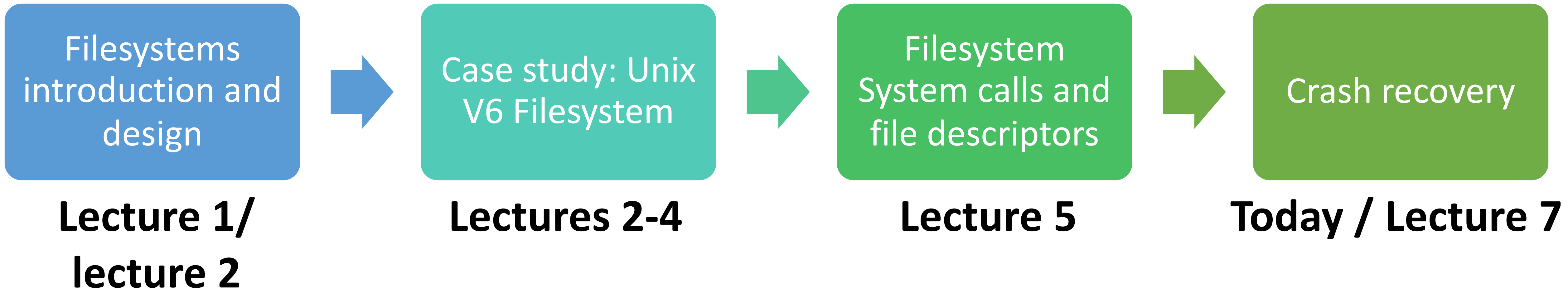
Optional reading:

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

😷 masks required

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


Filesystems introduction and design
Lecture 1/ lecture 2
Case study: Unix V6 Filesystem
Lectures 2-4
Filesystem System calls and file descriptors
Lecture 5
Crash recovery
Today / Lecture 7

Learning Goals

- Get practice working with file descriptors in programs
- Understand the goals of crash recovery and potential tradeoffs
- Learn about the role of the free map and block cache in filesystems

Plan For Today

- **Recap**: file descriptors and system calls
- Crash Recovery Overview
- Free space management
- Block cache

Plan For Today

- **Recap**: file descriptors and system calls
- Crash Recovery Overview
- Free space management
- Block cache

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. They are tasks the operating system can do for us that we can't do ourselves.
- **open()**, **close()**, **read()** and **write()** are 4 system calls we use to interact with files.

open()

A function that a program can call to open a file, and potentially create a file:

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

- **pathname**: the path to the file you wish to open
- **flags**: a bitwise OR of options specifying the behavior for opening the file
- **mode** (if applicable): the permissions to attempt to set for a created file
- returns a **file descriptor** representing the opened file, or -1 on error

Many possible flags (see man page). You must include exactly one of **O_RDONLY, O_WRONLY, O_RDWR**, which specifies how you'll use the file in this program.

- **O_TRUNC:** if the file exists already, clear it ("truncate it").
- **O_CREAT:** if the file doesn't exist, create it
- **O_EXCL:** the file must be created from scratch, fail if 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.
- Each program has its own file descriptors
- You can have multiple file descriptors for the same file
- 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)

`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)

read() and write()

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 (may not read all bytes you ask it to! E.g. if there aren't that many bytes, or if interrupted)

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

- *Same as **read()**, except the function writes the **count** bytes in **buf** to the file, and returns the number of bytes written.*

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) {
    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;
        }
    }
}
```

Example: Copy

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

Read in chunks of **kCopyIncrement** bytes

```
void copyContents(int sourceFD, in
    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;
        }
    }
}
```

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 < bytesRe
            ssize_t count = write(des                            itten,
                                  byt
            bytesWritten += count;
        }
    }
}
```

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) {
            ssize_t count = write(destinationFD, buffer + bytesWritten,
                                  bytesRead - bytesWritten);
            bytesWritten += count;
        }
    }
}
```

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;
        }
    }
}
```

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?

```
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?

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?

```
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;
}
```

Plan For Today

- **Recap**: file descriptors and system calls
- Crash Recovery Overview
- Free space management
- Block cache

Crash Recovery

Sometimes, computers crash or shut down unexpectedly. In those situations, we want to avoid filesystem data loss or corruption as much as possible.

How can we recover from crashes without losing file data or corrupting the disk?

assign2: implement a program that can repair a filesystem after a crash!

Crash Recovery

Key challenge: tradeoffs between *crash recovery abilities* and *filesystem performance.*

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. if a modification affects multiple blocks, a crash could occur when some of the blocks have been written to disk but not the others.
- 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 as free)

Ideally, filesystem operations would be **atomic,** meaning they happen in their entirety without interruption – they are never left in an incomplete state. But this isn’t fully possible, since crashes can happen at any time.

Crash Recovery

To understand crash recovery, we need to understand all places where filesystem data is stored and maintained.

- We know about most of the disk itself (e.g. Unix V6 layout), but not **how free blocks are tracked**. This factors into crash recovery (e.g. free blocks not in a consistent state).
- There is also the **block cache** in memory that stores frequently-used blocks accessed from disk. This factors into crash recovery (e.g. not all updates in block cache written to disk).

Plan For Today

- **Recap**: file descriptors and system calls
- Crash Recovery Overview
- **Free space management**
- Block cache

Free Space Management

- Early Unix systems (like Unix v6) used a linked list of free blocks
  - Initially sorted, so files allocated contiguously, but over time list becomes scrambled

More common: use a **bitmap**

- Array of bits, one per block: 1 means bock is free, 0 means in use
- Takes up some space – e.g. 1TB capacity -> 2^{28} 4KB blocks -> 32 MB bitmap
- 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)

Problem: slow if disk is nearly full, and files become very scattered

Free Space Management

More common: use a **bitmap** – an array of bits, one per block, where 1 means bock is free, 0 means in use.

- During allocation, search bit map for block close to previous block in file

Problem: slow if disk is nearly full, and blocks very scattered

- Expensive operation to find a free block on a mostly full disk
- Poor *locality* – data likely to be used next is not close by

Solution: don’t let disk fill up!

- E.g. Linux pretends disk has less capacity than it really has (try **df** on myth!)
- Increase disk cost, but for better performance

Free Space Management

The free list / bitmap is important to understand for crash recovery because it is a source of crash recovery issues.

- E.g. block assigned to file but not removed from free list / marked as allocated in bitmap

Plan For Today

- **Recap**: file descriptors and system calls
- Crash Recovery Overview
- Free space management
- **Block cache**

Block Cache

Problem: Accessing disk blocks is expensive, especially if we do it repeatedly for the same blocks.

Idea: use part of main memory to retain recently-accessed disk blocks. (Many OS-es do this).

- A *cache* is a space to store and quickly access recently- / frequently-used data.
- Frequently-referenced blocks (e.g. indirect blocks for large files) usually in block cache.

Challenge: cache size limited; how do we utilize it? What if it gets full?

Block Cache

Challenge: cache size limited; how do we utilize it? What if it gets full?

Least-recently-used "LRU" replacement – If we need something not in the cache, we read it from disk and then add it to the cache. If there's no room in the cache, we remove the least-recently-used element.

Another challenge: what happens when a block in the cache is modified? Do we stop and wait and immediately write it to disk? Or do we delay it slightly until later? Argue yes/no!

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

What happens when a block in the cache is modified? Do we stop and immediately write it to disk and wait? Why/why not?

Block Cache

Another challenge: what happens when a block in the cache is modified? Do we immediately write it to disk? **Pros/cons?**

If we immediately write to disk ("synchronous writes"):

- Safe: no data loss because it's written immediately
- Slow: program must wait to proceed until disk I/O completes

Block Cache

Another challenge: what happens when a block in the cache is modified? Do we immediately write it to disk? **Pros/cons?**

If we don’t immediately write to disk (“delayed writes”):

- Wait a while (Unix chose 30 seconds) in case there are more writes to that block, or it is deleted
- Fast and Efficient: writes return immediately, eliminates disk I/Os in many cases (e.g. many small writes to the same block)
- Dangerous: may lose data after a system crash! “Are you willing to lose your last 30sec of work in exchange for performance bump?”
- (Side note – **fsync** system call lets a program force a write to disk)

Recap

- **Recap**: file descriptors and system calls
- Crash Recovery Overview
- Free space management
- Block cache

Lecture 6 takeaway: Crash recovery requires tradeoffs with performance. Both the free list/map and block cache play a role in filesystem state and crash recovery.

Next time: approaches to crash recovery