CS111, Lecture 6
Crash Recovery

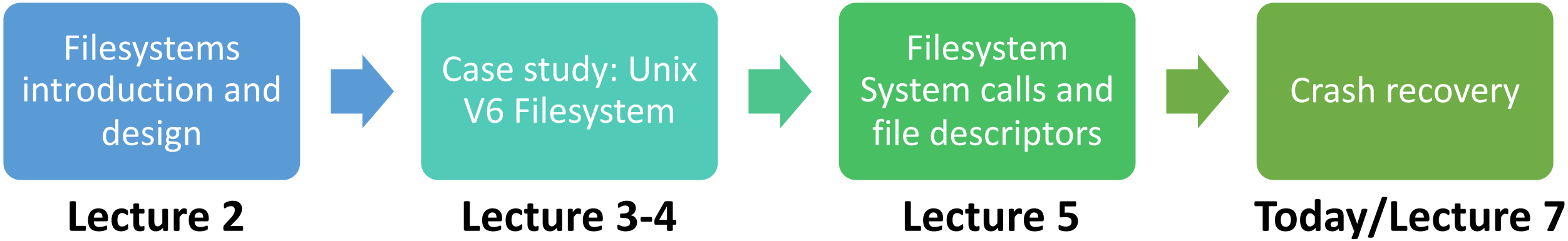
Optional reading:

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

😷 masks strongly recommended

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

- Learn about the role of the free map and block cache in filesystems
- Understand the goals of crash recovery and potential tradeoffs
- Compare and contrast different approaches to crash recovery

Plan For Today

- **Recap**: file descriptors and system calls
- Announcements
- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)

Plan For Today

- **Recap: file descriptors and system calls**
- Announcements
- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)

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 examples of system calls we use to interact with files.

File Descriptors

Filesystem system calls use **file descriptors** to identify open file sessions. 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)

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

`read()` and `write()`

We can call **read** or **write** to read/write bytes to an open file.

- **Key idea:** read/write may not read/write all the bytes you ask it to! The return value tells you how many were actually read/written.
- **Key idea #2:** the operating system keeps track of where in a file a file descriptor is reading or writing to. So the next time, it will pick up where you left off.

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.

```
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;
}
```

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?

```
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?

```
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
- **Announcements**
- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)

Announcements

- Section 1 grades posted to gradebook; solutions posted on course website
  - [NEW] for section accommodations (e.g. illness), please email your section TA
- assign1 updates:
  - Must match sample solution error messages to pass tests 23 and 50
  - Can't rely on inode_indexlookup error checking in directory layer to know when we've reached the end of the directory entries – must look at size instead
- Additional section slots posted
- [NEW] Assignments/Exams Accommodations TA (extensions, OAE, etc.): Michela Marchini (marchini @ stanford)
  - Please put "CS111" in subject line!
- Reminder: please let us know about any midterm conflicts by **Fri 10/13**

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)
- We'll learn about how free blocks on disk are tracked. This factors into crash recovery (e.g. free blocks not in a consistent state).
- We'll learn about the **block cache** in memory that stores frequently-used blocks accessed from disk.

Plan For Today

- **Recap**: file descriptors and system calls
- Announcements
- **Free space management**
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)

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 block 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 block 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

Plan For Today

- **Recap**: file descriptors and system calls
- Announcements
- Free space management
- **Block Cache**
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)

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 OSes 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. (not necessarily whole files, just individual blocks).

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?

One approach - 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.

The block cache allows us to *delay* operations until later (e.g. when modifying a single block many times, don't write to disk every time, but rather once later on). We'll see more about this in a bit!

Plan For Today

- **Recap**: file descriptors and system calls
- Announcements
- Free space management
- Block Cache
- **Crash Recovery Overview**
- Approach #1: Consistency check on reboot (**fsck**)

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, and explore some of the security and ethical implications of OSes / filesystems.

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 in the free list)

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

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

Tradeoffs Example: the Block Cache

The block cache is a great example of how we may have to choose between crash recovery abilities and filesystem performance.

Key question: 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?

Block Cache

Key Question: 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"):

- Safer: less risk (but not zero risk!) of data loss because it's written as soon as possible.
- Slow: program must wait to proceed until disk I/O completes

Block Cache

Key Question: 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 more 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)

Crash Recovery

We will discuss 3 approaches to crash recovery, building up to the most common one – **logging:**

1. Consistency Check on reboot (**fsck**)
2. Ordered Writes
3. Write-Ahead Logging (“Journaling”)

Plan For Today

- Recap: file descriptors and system calls
- Announcements
- Free space management
- Block Cache
- Crash Recovery Overview
- **Approach #1: Consistency check on reboot (fsck)**

Crash Recovery

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")

- Must check whether there was a clean shutdown (if so, no work to do). How do we know? **Set flag on disk on clean shutdown, clear flag on reboot**.
- If there wasn't, then scan disk contents, identify inconsistencies, repair them.
- Scans metadata (inodes, indirect blocks, free list, directories)
- Goals: restore consistency, minimize info loss

Possible fsck Scenarios

Example #1: block in file and also in free list?

Action: remove block from free list

Example 2: block a part of two different files (!!)

(Maybe deleting a file, then making a new file, but with the block cache the new file updates were written but not the old file updates)

Action: randomly pick which file should get it? Make a copy for each? Remove from both? (probably not, don't want to lose potentially-useful data)

Example 3: inode *reference count* (# times referenced by a directory entry) = 1, but not referenced in any directory.

Action: create link in special lost+found directory.

Limitations of fsck

What are the downsides/limitations of **fsck**?

- Time: can't restart system until **fsck** completes. Larger disks mean larger recovery time (Used to be manageable, but now to read every block sequentially in a 5TB disk -> 8 hours!)
- Restores consistency but doesn't prevent loss of information.
- Restores consistency but filesystem may still be unusable (e.g. a bunch of core system files moved to lost+found)
- Security issues: a block could migrate from a password file to some other random file.

Can we do better? What if we made design changes to the filesystem structure to implement crash recovery?

Recap

- **Recap**: file descriptors and system calls
- Announcements
- Free space management
- Block Cache
- Crash Recovery Overview
- Approach #1: Consistency check on reboot (**fsck**)

Next time: more about crash recovery

Lecture 6 takeaways: The free list tracks free blocks on disk and is commonly implemented using a bitmap. The block cache caches recently-accessed disk blocks. Crash recovery challenges include both data loss and inconsistency. **fsck** is one approach to crash recovery.