Agenda for Today

1. Lab Q&A
2. Filesystem Implementation
3. Tolerating Crashes
P2P Chat Lab Q&A

When is the lab actually due?
Monday, March 22nd at 11:59pm.
Watch for an update about office hours on Monday. I’m going to try to hold them.

Is it safe to call the ui_display function from a thread without extra locking?
Yes, it is. This property of being safe to call concurrently from multiple threads is known as “thread safe”. That doesn’t mean the interface implements the atomicity you need, but it at least won’t break the interface.

What part of this program has to run forever?
Almost all of it. Your program will loop indefinitely waiting for the user to type input. It will also loop forever waiting for peers to connect. You’ll also be looping to read from any connected peers, until that connection fails (because it was closed). You should close sockets for connections to peers that stop working, but you won’t ever call pthread_join in this lab.
What is the difference between socket_connect and server_socket_open?
Both of these give you a socket FD, but they’re logically different.
server_socket_open creates a special server socket we’ll call server_fd, which you can use only to accept new connections. You have to call listen() on the socket first, but after that point the only thing you can do with it is call server_socket_accept(server_fd)

socket_connect establishes a connection to a server socket listening on a specific port at some machine address (we call this hostname).

For this work, the server has to have called server_socket_accept. When the connection is established, socket_connect returns fd1 to its caller and server_socket_accept returns fd2 to its caller. Anything you write to fd1 will be read from fd2, and anything you write to fd2 will be read from fd1.
P2P Chat Lab Q&A

1. If the user at peer 1 types a message, peer 1 needs to display it and write it to fd1.

2. Peer 0 receives the message by reading fd2. Peer 2 displays the message and forwards it to any other peers.
Filesystem Implementation
Elements of a Filesystem

- Superblock
- Inodes
- Data Blocks
- Bitmap
- Data
- Inode
- Bitmap
**A Simple File System**

**Disk:**
- 256 blocks, each 4KB
- 4 byte disk addresses

**inode:**
- 52 bytes of inode metadata
- 8 direct block pointers

**layout:**
- 1 superblock
- 1 block for inode bitmap
- 1 block for data block bitmap
- 4 blocks for inode entries

---

What is the total size of this disk?

\[
256 \text{ blocks} \times 4 \text{ KB/block} = 1024 \text{ KB} = 1 \text{ MB}
\]

How many inode entries or data blocks can be tracked with a single bitmap block?

\[
4096 \text{ bytes/block} \times 8 \text{ bits/byte} = 32768 \text{ bits/block}
\]

What is the size of an inode entry?

\[
52 \text{ bytes} + 4 \text{ bytes/ptr} \times 8 \text{ ptrs} = 84 \text{ bytes}
\]

How many inodes does this file system support?

\[
4 \text{ blocks} \times 4096 \text{ bytes/block} / 84 \text{ bytes/inode entry} = 195 \text{ inodes}
\]

How many data blocks does this file system have?

\[
256 \text{ blocks} - 7 \text{ reserved blocks} = 249 \text{ data blocks}
\]

What is the largest supported file size?

\[
4 \text{ KB/block} \times 8 \text{ blocks/inode} = 32 \text{ KB/inode}
\]
File System Operations

Work with your breakout group to answer each of the following questions.

What steps does the OS need to take to...

1. Read an entire file off the disk, given an inode number
2. Write a new file to disk in some directory, given the directory's inode number
3. Delete a file from the disk, given its inode number
File System Operations (responses)

Read an entire file off disk given its inode number

“/home/curtsinger/csc213/labs/p2pchat/p2pchat.c” has already been traversed and turned into an inode number.

1. Read the inode entry from the disk
2. Make sure the user has permission to read this file. If not, return an error.
3. Update the last-accessed time of the file in the inode entry’s metadata
4. Set up a file descriptor and reserve some memory to hold the file data
5. For each non-null direct block pointer in the inode entry:
   A. Read the block referenced by the block pointer
   B. Store the data in the memory reserved in step 4
Write a new file to disk in directory D, given D's inode number.

1. Load the directory’s inode from disk
2. Load all of the directory’s blocks using the inode direct block pointers
3. Check the directory content: is there space to add the new entry? Let’s assume there is.
4. Load the inode bitmap and find the first available inode (indicated by a zero bit). Set the bit to 1 to claim the inode. This inode will be the new file.
5. Add a link to the directory’s content that points to the new inode.
6. Find enough free data block to store the new file using the data block bitmap, and then claim them by writing ones to those bits.
7. Initialize the inode entry with some default metadata
8. Add the data blocks to the new inode entry’s direct block pointers
9. Write out whatever initial content the file has to its data blocks.
Delete a file from the disk, given its inode number

1. Load the directory inode entry from the disk
2. Load the data blocks that hold the directory’s content (its links)
3. Find the named entry and the file’s inode number in the directory content.
4. Remove the directory entry from the directory’s content.
5. Decrement the file inode’s link count. If it’s >0, we’re done. But if it goes down to zero we are removing the last link to this file. Continue.
   A. Mark the file’s data blocks as unused in the data block bitmap
   B. Mark the file’s inode entry as unused in the inode bitmap
A Better Simple File System

Disk: 256 blocks, each 4KB
4 byte disk addresses

inodes: 52 bytes of metadata
8 direct block pointers
1 *indirect block pointer*

layout: 1 superblock
1 block for inode bitmap
1 block for data block bitmap
4 blocks for inode entries

Consider the same filesystem we looked at earlier, but now each inode entry contains an additional pointer to an indirect block.

What is the effect of adding this indirect block pointer?
Tolerating Crashes
Crash Consequences

Creating a new file incurs at least five separate writes. These writes could be performed in any order (the disk decides).

What happens if a crash means the drive does not successfully...

- write to the inode bitmap?
- write to the data block bitmap?
- write to the inode entry?
- write to the new file's data block?
- write to the containing directory's data block?
Crash Consequences

Lost write to inode bitmap

Lost write to data block bitmap
Crash Consequences

(responses)

Lost write to inode entry

Lost write to file data block(s)

Lost write to directory entry
What does fsck do?

Can fsck detect all of the problems that arise from lost writes?

Can fsck repair all of the problems it detects?
Journaling

What is journaling?

What problems can journaling prevent or repair?

What is metadata journaling? Why would we use it instead of full journaling?