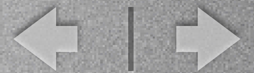
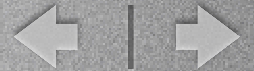


CS 415 OS Practicum

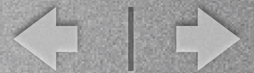
Project 3: Filesystems

Oliver Kennedy (okennedy@cs)



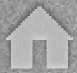
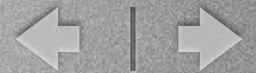
Review Questions

- Which stack are you running your alarms on?
    - Why?
- What is a well behaved process?
- Why do we want to expedite the execution of well behaved processes?
- Does the multi-level queueing scheme we use allow thread starvation?



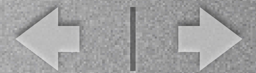
Project 3: Filesystems

- What are filesystems and why do we use them?
- How do we design our filesystem?
- What do you need to know to implement your filesystem?

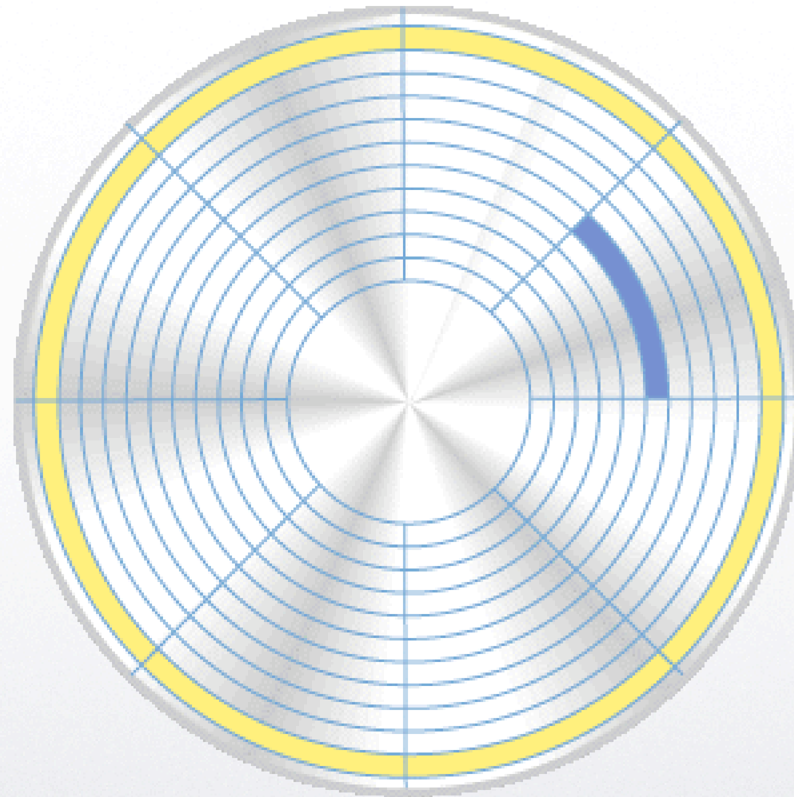



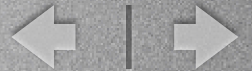
The Hard Disk

- A hard disk is composed of one or more platters.
- Each platter is divided up into concentric rings called tracks.
- Each track is broken up into chunks called sectors, or blocks.
  - Typically sectors on multiple tracks overlap.



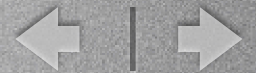
A Hard Disk





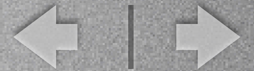
The Hard Disk

- Hard drives are high latency.
  - ... but also high bandwidth.
- Reading individual bytes is slow.
  - We read a sector at a time (4k).



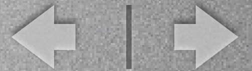
The Virtual Disk

- disk_create(*disk, name, size, flags)
    - Creates a virtual disk, backed to a file.
    - size is in 4k blocks.
- disk_startup(*disk, name)
    - Opens a virtual disk created with disk_create.
- disk_read_block(*disk, blocknum, buffer)
- disk_write_block(*disk, blocknum, buffer)



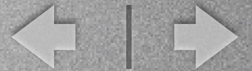
The Virtual Disk

- install_disk_handler(handler)
- disk_send_request(...)
  - The raw request interface; you don't need this.
- Note that the first parameter is passed by reference.
  - disk_create(&mydisk, "MYFS", 1024, myflags)
    - 4 MB disk



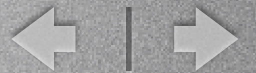
Filesystems

- This interface is not programmer-friendly.
  - Most programmers don't like asynchronous interfaces.
  - Restricting file sizes to 4k and filenames to block numbers is just bad.
- Filesystems place a friendly face on the raw disk interface.
- Note that the following is a very limited view of filesystems that ignores most of the work done in the past 20 years.

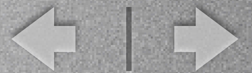


We need...

- A way to link blocks together for large files.
- A way to 'name' blocks.
- Some sort of directory structure.
- A way of keeping track of free blocks
- A way to make this structure persistent.

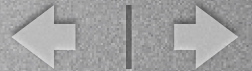


A way to link blocks...



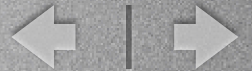
A way to link blocks...

- Let’s use some special indexing blocks; we can call them inodes.



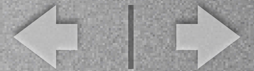
A way to link blocks...

- Let's use some special indexing blocks; we can call them inodes.
- Use an inode as an array of data block numbers.
    - but there's still a problem...



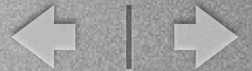
A way to link blocks...

- Let’s use some special indexing blocks; we can call them inodes.
- Use an inode as an array of data block numbers.
  - but there's still a problem...
- Use the 1024th block pointer as a pointer to the next inode.



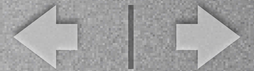
A way to link blocks...

- Let’s use some special indexing blocks; we can call them inodes.
- Use an inode as an array of data block numbers.
  - but there's still a problem...
- Use the 1024th block pointer as a pointer to the next inode.
- More advanced solutions:
  - Use indirection: Store pointers to 1024 inodes that point to blocks.
  - Hybrid approach: store 512 block pointers and 512 indirect pointers.



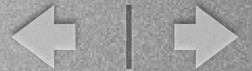
A way to name blocks...

- Directory inodes
  - Use an inode to store filename->file_inode mappings.
  - Like files, you can create an overflow pointer.
- But a directory is just another inode.
  - A directory can also link filenames to directory_inodes.
- You may want to store other things with these mappings
  - File/Directory size
  - File type = FILE | DIRECTORY



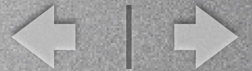
Some more inode types

- Free-list
  - Store all your free blocks as a linked list
  - Simple to initialize
  - Simple to maintain: O(1) on all modifications
- Root Inode/Superblock
  - Stores pointers to all the inodes you need to maintain the system.
  - Stored in a well known location: block 0



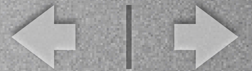
Summary

- Superblock
  - Pointer to the root directory
  - Pointer to the first free block
- Free Block
  - Pointer to the next free block
- Directory Inode
  - Header Info
  - Array of inode pointers with associated filename, type, etc...
  - Pointer to overflow inode



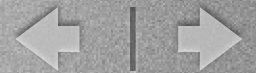
Summary

- File Inode
  - Array of inode pointers
  - Pointer to overflow inode
- Data Block
  - all user data



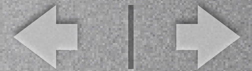
Synchronization caveats

- The hard disk controller may reorder requests as it sees fit.
- The hard disk may fail at any time
- It's ok to 'lose' free blocks
    - It's not ok to lose user data
    - All changes should be 'atomic'



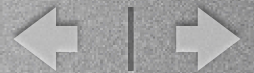
You there, in the back wake up!

- How do you...
  - Create a directory?
  - Create a file?
  - Write a block to a file



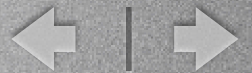
Project 3

- minifile_t minifile_creat(filename)
- minifile_t minifile_open(filename, mode) //r, w, or rw
- minifile_read(file, data, len) //should block until the read succeeds
- minifile_write(file, data, len) //should block until the write succeeds
- minifile_close(file)
- minifile_unlink(filename) //delete a file
- minifile_mkdir(dirname)
- minifile_rmdir(dirname) //delete a directory
- int minifile_stat(path) //0 = file; -1 = doesn't exist; -2 = dir
- char **minifile_ls(path) //assume the caller will free the array and each element
- interrupt handler



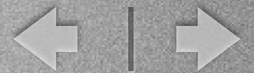
Working directories

- minifile_cd(path)
- char *minifile_pwd()
- Each thread has a separate working directory
  - It's ok to store this in minithread_t and access it via minithread_self()



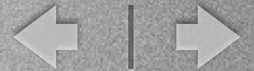
mkfs.exe

- disk_create()
- Create the superblock
- Create /
- Initialize the free list



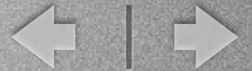
What’s an open file?

- Pointer to the file's inode
- Cursor
- File Size
- Open count
- Linked list?



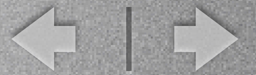
Some final words

- Writes may not be aligned to block boundaries
  - You may need to read one or more blocks before writing
- Synchronization, synchronization, synchronization
  - ... via semaphores/mutexes/interrupt disabling
  - ... via enforced ordering
- Don't assume a block has been written to disk until the write returns



More Final Words

- The disk must not enter an inconsistent state
  - 'lost' free blocks are ok
  - Inodes pointing to free blocks are not
  - Write data first, then propagate updates back up the directory tree.
- Synchronize your root block!



fin