Today's plan

- Minix FS in-memory data structures
- Minix FS disk layout
- Minix buffer cache
- Minix inodes and superblock
- Minix opening files

Minix File System Implementation: in-memory data structures

- the file system process table (line 21500) includes the current and root directories, process IDs, an array of pointers to open files, and information that allows saving information for slow reads/writes (e.g. on a pipe or a terminal)
- inodes on disk (21100) have up to 8 direct block pointer ("zones"), one indirect, one double-indirect -- total, 64 bytes
- an inode block is read from disk whenever the file corresponding to at least one of the inodes is opened
- the corresponding inode representation in memory (21900) has additional references to make it easy to locate the device and the device's superblock, whether the inode is dirty, whether the file is special in some way (pipe, mount, etc)
- the superblock array (22100) stores information about each mounted file system, particularly the sizes of the inode and zone bitmaps. The in-memory copy of the structure also stores mount information, including the device and the byte-order
- an open file structure (21700), pointed to by the open files in the process table, contains a pointer to the inode, the position being read or written, and a few other items. These structures may be shared, e.g. by a parent and a child after a fork.
- the lock array (21800) contains information for each file lock set -- this is checked every time a lock is requested
- the device array (04220) contains function pointers for opening, closing, and reading/writing data, and the number of the corresponding device task

Minix File System: structures on disk

- boot block, including the partition table if any
- superblock, including information about the layout of the file system on disk
- bitmap for free inodes
- bitmap for free zones (blocks)
- inodes
- zones (blocks)

Minix File System: buffer cache

- a buffer may contain (p. 926) a data page, a directory page, an indirect block, an inode block, **or** a bitmap block
- free buffers are kept in a hash table, to make it easy to locate them quickly. The hash table is indexed by the low bits of the block number
- most free buffers are maintained in an LRU doubly-linked list, but some free buffers are placed at the head of the list because they are very unlikely to be needed again (e.g. superblock buffers)
- when a new buffer is needed, it is taken from the head of the list
- when reading a block from disk (or when writing part of a block), the hash table is searched to see if the block might be in the buffer -- if so, no need to access the disk
- when writing a full block to disk, the block then goes to the head of the LRU list
- most dirty buffers on the LRU list are only written back when:
  - the block reaches the head of the list, or
  - another block on the same device is written back
- when writing any block to disk, Minix writes all blocks from that device, in sorted (elevator) order

Minix Buffer Cache Implementation

- `get_block` (22426) searches through the list, returning it (line 22463) if found, and if not found, allocates a new block (by recycling the head of the LRU, 22473) and, if necessary, fetches it from disk (line 22511, also 22641)
- recycling the first block on the LRU chain may require writing back a dirty block, in which case all blocks are written back (`flushall`, 22694, `rw_scattered`, 22711)
- `put_block` (22520) puts the block at the rear or at the front of the LRU list, possibly writing it back immediately
- `alloc_zone` (22580) and `free_zone` (22621) manage the bitmaps for zones, reading the bitmap from disk and saving it back to disk as appropriate -- see alloc_bit (23324) and free_bit (23400)
- `rw_scattered` (22711) does the actual sorting of reads/writes and performs the I/O requests, freeing the corresponding blocks (by calling `put_block`, lines 22779 and 22791) and clearing the dirty bits

Minix Buffer Cache Summary

- blocks are in LRU queue and also hash table
- blocks in buffer cache are not in use, may be reused soon
- blocks in buffer cache may be dirty, in which case they should be written back before they may be used
- free blocks are taken from the head of the queue
- newly freed blocks are put at the end of the LRU queue, except in special cases
- newly freed dirty inode blocks are written back immediately

Minix Inode and Superblock Implementation

- inodes have a link count and an in-memory reference count:
  - if the reference count becomes zero, the file should be closed
  - if the link count becomes zero, the file should be removed
- duplicating an inode (as in `dup` or `dup2`) simply requires incrementing the reference count
- when creating a new inode, the block containing the inode must be read into memory, since the other inodes on the block might already exist (as an optimization, the bitmap could be checked to avoid the read if all other inodes in the block are free... but minix doesn't do this)
- inode blocks are written back immediately
- to avoid calling the clock unnecessarily, the times are only updated at most once when the inode is written back (p. 944, 23099)
- superblock management includes allocating and freeing bits in the two bit maps. Allocating starts from the last position searched or freed

Minix Opening and Closing

- `common_open` (p. 966, 24573)
- may create a new file, by calling `new_node` (p. 968, 24697), which allocates an inode then adds the name and the inode to the directory entry
- allocates an in-memory inode and filp and initializes them
- checks protections
- does type-specific operations for regular files, directories, block- and character-special files, and pipes
- e.g. for regular files, truncates them (by returning all the blocks and clearing the inode) if the open requested truncation
- `mknod` and `mkdir` (p. 970) do what is needed, e.g. `mkdir` creates the "." and ".." links
- `search_dir` (p. 997) edits or serches the directory
- `last_dir` (p. 994) returns the inode of the last directory in the path