Today's plan

- Minix RAM disk driver
- Minix hard disk driver
- Minix terminal driver

Partition Table

- a partition table for the x86 is a list of 4 partition entries at offset 446 (x1BE) in the first (boot) sector of a disk
- each partition entry has a 16 bytes of data (the last 2 bytes of the first sector hold the magic number 55AA, line 11571)
- the first 8 bytes include flags (e.g. 0x80 for a bootable disk) and geometry (heads, sectors, cylinders)
- the final 8 bytes are two 4-byte numbers recording the linear number of the start and size of the partition, in sectors
- partitions may be nested, e.g. the first partition may have another partition table in its first sector

Minix RAM disk

- RAM disk is block-oriented, just like the real disks, even the block size is the same
- RAM disk is stored in main memory: a 1MB RAM disk takes up 1MB of main memory
- write requests write directly to memory
- read requests copy data directly from memory
- six different Minix RAM devices, all with major device number 1 and using the same driver:
  - minor device 0: `/dev/ram`, a true RAM disk on which the root file system is usually mounted. Memory for this is allocated at boot time
  - minor device 1: `/dev/mem`, a pseudo RAM disk corresponding to the physical memory of the PC
  - minor device 2: `/dev/kmem`, a pseudo RAM disk corresponding to the kernel virtual memory of Minix
  - minor device 3: `/dev/null`, a pseudo RAM disk that discards all its data and is always empty
  - minor device 4: `/dev/boot`, a pseudo RAM disk that gives access to the in-memory copy of the boot device
  - minor device 5: `/dev/zero`, a pseudo RAM disk that discards all its data and returns as many bytes of zero as requested
- the first three RAM disks and `/dev/boo`t have some overlap: aliasing, different names for the same underlying objects

Minix RAM disk implementation

- pages 782-787
- many functions (e.g. `close`, `cleanup`) are no-ops
- `main` in the memory driver calls `m_init` (which sets up each device's pointers and sizes) and then goes to the shared `driver_task` function
  - open checks the device number
  - exactly one ioctl is supported, to set the size of a RAM disk (only FS may do this)
- doing this requires allocating memory, which eventually leads to a call to the process manager (PM)
- the code in `servers/vm/alloc.c` implements the first-fit memory allocation algorithm (`findbit`, lines 369-399)
  - current code calls this via `mmap` (line 588 in `drivers/storage/memory/memory.c`)
- `m_geometry` returns the (completely fake) geometry
- reading or writing of `/dev/null` is a question of returning the correct number of bytes read or written (0 read, all written)

Reading or writing a RAM disk block

- `m_transfer`, p. 783-785
- ignore the optional bit -- all operations on a RAM disk complete immediately
- one iteration per request in the vector
- for "normal" memory devices, check for address legality, i.e. within the RAM disk, and truncate the transfer if necessary
- select the appropriate segment
- perform the virtual or physical copy
- update the number of bytes copied

AT hard disk control flow

- `at_winchester_task` called
- `init_params` checks the boot parameters and the BIOS, to figure out which disks are present
- eventually, a DEV_OPEN message leads to calling `w_do_open`, which calls `w_prepare` (most messages, not just DEV_OPEN, lead to calling `w_prepare`), `w_identify`, and finally, if this is the first open for the device, `partition` (line 11426 onward). `partition` calls `transfer` (line 11566, in `get_part_table`) to get the boot sector
- `w_identify` does the hard work of requesting and decoding the device information
- the identification request is built by com_simple (line 13030), and sent to the device by `com_out` (line 12947)
- `com_out` waits for the controller to not be busy (line 12960), then selects the drive, then waits again
- the drive controller registers are shown in Figure 3-23 on p. 296
- the actual I/O to the controller is done in `kernel/system/do_vdevio.c`, line 108, using `outb`
- the driver will use cylinder/head/sector addressing if necessary, but LBA (Logical Block Addressing) if possible, with up to 48 bits for a block number
- the final step in `w_identify` enables the interrupt from the disk device (line 12701), specifying that interrupts should not remain blocked while the device driver executes

AT hard disk read and write

- on a `w_transfer` (line 12848), adjacent requests (up to wn->max_count) are done in a single operation
- the transfer request is built by `do_transfer` (line 12814), and sent to the device by `com_out`
- after `do_transfer`, there is a loop that is executed once per sector, beginning on line 12890
- read calls `at_intr_wait` (line 12909) which calls `w_intr_wait` which calls receive
- on a read or a write (line 12908), `w_waitfor` reads the status from the disk controller (line 13190) until the status is STATUS_DRQ (data transfer request, line 12190)
- bytes are finally copied with `sys_insw` or `sys_outsw`, which is programmed I/O rather than DMA
- a write now waits for an interrupt (line 12920), to check whether everything was written
- data is always transferred one sector at a time
- for either reads or writes, the next I/O descriptor is then selected if this I/O descriptor has been satisfied
- in case of timeout (`w_need_reset`, line 12999, called from line 13198), set a bit and request a re-initialization of the device
- in case of timeout on the interrupt, `w_timeout`, line 13046, (called from line 13134) decreases the maximum size given to `do_transfer` from n to 8 sectors, and from 8 sectors to 1, in case that was what tripped up the drive

Minix Terminal Driver

- very complex: supports memory-mapped keyboard and displays, RS-232 serial terminals, and network-based logins (Pseudo-ttys, or PTYs, on other systems)
- character devices, but with two-dimensional positioning capabilities (screens)
- screens can be character-based or pixel-based (minix only supports character-based)
- buffering can be reserved per terminal (as is the case in Minix) or shared among all terminals (central buffer pool)
- terminal driver must perform **line editing functions** to honor erase characters and possibly change newlines into CR-LF or viceversa (or other combinations)
  - modern systems support more advanced line editing

Terminal modes

- editors will do their own screen redrawing, can handle erase characters, so should be given the **raw** stream of characters the user enters
- most programs take line input and would prefer to have the operating system take care of editing: **canonical** or **cooked** mode
- special characters can control freezing (^S) or restarting (^Q) the output, mark end of file (^D), end of line, etc
- in non-canonical mode, Minix allows the specification of a minimum number of characters to read and of a timeout for terminal reads -- if either is satisfied, the read call completes

Reading from the terminal

- book, figure 3-33, p. 319

1. user process sends message to file system
2. file system sends message to TTY task, which may directly go to (6), but most likely
3. replies asking the file system to suspend the process
4. when a key is pressed, the generic interrupt handler notifies the TTY task
5. the TTY task reads the I/O ports to determine which key was pressed, and adds the character to a queue
6. at the top of the TTY task, the task copies available data directly to the user space using physical memory copying
7. the TTY task tells the file server (whenever it is ready to receive the message) that it may wake up the user process
8. the FS server wakes up the user process

- this complex technique gives acceptable performance for large bursts of characters from the serial port on slow hardware, since the user process and the file system are only involved when enough characters (or a timeout) have been received
- serial line controllers may be configured to only interrupt after receiving several characters, rather than once per character

Bitmapped Displays

- each pixel (usually 1, 8, 16, 24, or 32 bits) represents one dot on the display: one scan sweep position or one pixel on an LCD display
- more resolution requires more memory (1280x1024x24 requires 4MB for each display, without counting virtual desktop space)
- data can be arranged in various fashions in memory, but usually such that adjacent elements of a row are adjacent in memory
- basic display operations include moving a block (**bitblt**), drawing a point or a line, or filling in a rectangle (can also be done with bitblt)
- the device controller may include a fast data-parallel (SIMD) computer to operate on many bits at once -- the **GPU**
- a **window manager**, which may be part of the OS, must create windows, associate them with processes, and support opening, closing, moving, iconifying, etc
- the **X-window system** is a user-level program (an **X server**) that supports basic window operations for (possibly remote) client programs
- one of these client programs is the X window manager
- other client programs show the time, check for mail, allow for user command-line input (by running shells, perhaps remote shells), support surfing the web, etc.