Today's plan

- Minix a.out and kernel executable formats
- booting
- I/O hardware
- DMA
- structure of I/O software

`a.out` format

- described in `include/a.out.h` (not in book)
- 2-byte magic number helps avoid inadvertent execution of text and other random files
- 1-byte flags field identifies different styles of executables
- 1-byte CPU field identifies architecture on which it is meant to run
- 1-byte header length allows for header variability
- segment lengths for text, data, bss segments (below)
- entry point records the address to jump to when executing the file
- text segment contains executable code
- data segment contains initialized data, with initial values
- bss segment contains data initialized to zero, so only the size is recorded in the file, no actual data is stored

memory layout of Minix kernel

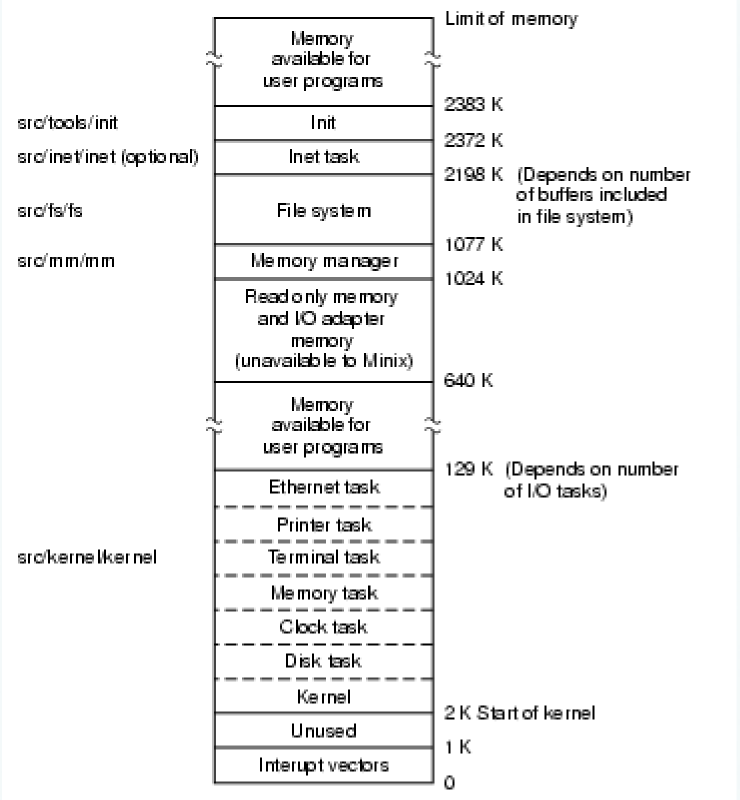

Figure 2-27. Memory layout after MINIX has been loaded from the disk into memory.

figure 2-31 on p. 129

kernel file format

- kernel on disk is simply a concatenation of:
  - one a.out for the kernel and the tasks and some drivers
  - one a.out for the memory manager (pm)
  - one a.out for the file system (fs)
  - one a.out for the inet server
  - one a.out for the rs process
  - one a.out for the init process
  - anything else the OS is configured for
- entry point is the label MINIX in `mpx386.s` (p. 709), which then calls `cstart` (p. 716) and finally `main` (p. 718)

booting

- ROM built-in to the machine (the BIOS) loads the first sector (512 bytes) from the floppy disk and executes it
- or reads a floppy's worth of data from a CD drive and treats it as a floppy
- or reads the partition table from the hard disk, locates the active partition, loads the boot block (512 bytes) from the active partition, and executes it
- boot sector program is hardcoded with the sectors of a program called `boot` which does the actual initialization -- a different boot sector is written by `installboot` depending on where `boot` is on the disk
- `boot` understands the minix file system, and searches for a file named `/boot/image` or, in a `/boot/image/ directory`, the newest file, or whatever file is specified by the boot parameters
- `boot` copies this file to memory (at location 2K for Minix) and jumps to the entry point
- **diskless workstations** need enough networking in the ROM to request a kernel image from a server

I/O hardware

- input and output devices have a range of hardware
- for example, a disk drive has a spinning disk and a moving head
- many printers have a sliding print head and mechanisms (e.g. ink pumps) to get the ink to the paper
- network and display devices achieve their effects by moving electrons
- all these mechanisms are electrically driven and controlled
- logic and drive hardware converts control bits into output motions, output bits into output signals, and input signals into input bits

Device Controllers

- in theory, a computer can directly provide the output and control bits and read the input bits
- in practice, the computer can appear to be faster if it doesn't have to do so
- most I/O devices therefore include some sort of intelligent device controller which:
  - provides **control registers** and **data registers** that the computer can read from and write to
  - executes **operations** in response to bits in its control registers
  - may **interrupt** the computer once an operation is complete

Device Controller Example

- A disk drive controller provides bits to the motor logic and to the write head logic, and reads bits from the read head logic
- these bits execute commands (for the motors) or read or write disk blocks on the disk
- when a disk block is read in, it is stored (usually) in the controller's internal memory
- from there, the controller can copy it directly into the main memory of the system: Direct Memory Access, or **DMA**
- or the controller can DMA a block from the main memory into its buffer, prior to writing it to the actual disk
- after the DMA is complete, the controller will interrupt the CPU to let it know it is done (even though the data may not yet be on disk)
- if DMA cannot be overlapped with a disk transfer, reading successive blocks from the disk will result in large delays (an entire disk rotation), so **logically** successive blocks are **interleaved** on the actual track so they can be read or written in quick succession

Programming a Device Controller

- the registers for each device controller are accessible to the CPU either through memory read/write operations (**memory mapped I/O**) or through special operations (I/O ports)
- for example, the x86 `inb` and `outb` operations are used to access specific ports on the I/O bus
- complex device controllers will take a list of commands at once, will copy the list from main memory, execute each of the commands, and interrupt when done
- simple device controllers will do exactly one thing in response to a command

Specific Example of Programming a Device Controller

- the PIC 12F675 microcontroller has a built-in, 4-input A/D (analog to digital) converter
- to take a sample, a program must:
  - enable A/D on the given port (by setting a bit in each of two device register)
  - select a channel by setting two of the bits in a device register. If another channel was previously selected, wait a settling time to avoid being affected by the input voltage on the other channel
  - select a voltage reference for the conversion (the result of the conversion reflects what fraction of the reference the input is), by setting a bit in a device register
  - select a conversion clock based both on the CPU clock frequency and the desired speed of conversion, by setting 3 bits in a device register
  - select a format for the 10-bit result (8+2 bits, or 2+8 bits)
  - determine whether the end of the conversion will cause an interrupt, by setting a bit in a device register
  - set the GO bit in a device register
  - either wait for an interrupt, or repeatedly sample the DONE bit to figure out when the conversion is complete

Programming an A/D controller, continued

- many of these bits are set once at the beginning of the program (e.g. input ports and clock frequency), others must be done for every conversion
- many of these operations can be combined by writing an entire device register at once
- the author of the device driver must understand these operations and execute them in the appropriate sequence
- note that A/D converters are usually simpler than disk drives -- for example, there are no error conditions

Structure of I/O Software

- many devices have identical (or nearly identical) interfaces
- these devices should be handled by a single **device driver**
- devices with similar functions but different interfaces (e.g. floppies, hard disks, and CD drives) should be handled by a **device-independent layer** that uses the device drivers to perform its operations
- ultimately, all this is in the service of **user-level software** that knows what operations need to be performed
- each of these may or may not be in the kernel

Device Drivers

- a device driver should accept device-independent requests and convert them to commands to the device controller
  - this is the **top half** or **upper half** of the device driver
- for example, a device-independent request might be to write a disk block
- the device driver has to program the device controller to write the disk
- after the operation is complete (perhaps as signalled by an interrupt), the device driver should check for errors, and perhaps retry the operation or return an error code
- the device driver can then free any resources, e.g. memory buffers
- interrupts are handled by the **lower half** or **bottom half** of the device driver

device-independent software

- a file system should not be designed only for one type of disk device
- a networking stack should not be designed only for one type of interface
- a windowing system should work with a variety of displays and graphic cards
- the device independent software should provide high-level operations such as naming, file abstractions, storage allocation, reliable transmission, routing, etc -- it is these abstractions that ultimately give the operating system API

user software I/O

- user software knows what contents to put in what files, what files to read, what data to send, what to display on the screen, etc
- some I/O functions, such as spooling print files, running routing protocols, or network services, is usually implemented by user-level processes known as **daemons** or **servers**