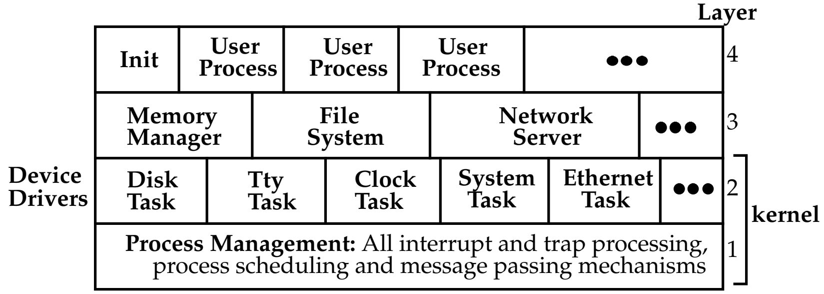
Minix Structure:

Minix is structured in four layers

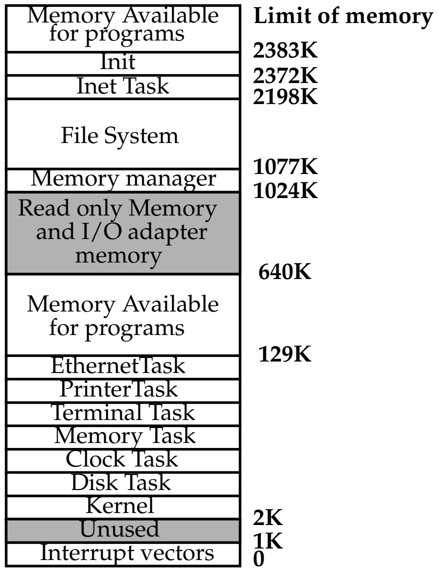


Layer 1: Highest privilege level, can access all of memory and execute any instruction.

Layer 2: Second highest privilege level; cannot access all of memory but can do I/O and access memory regions of less privileged processes.

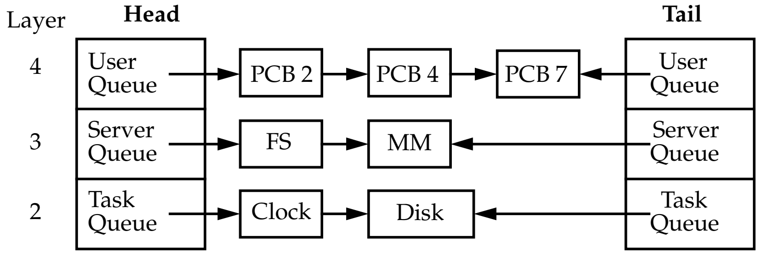
Layer 3: Have same privileges as layer 4 but run at higher priority.

Minix Memory map:



Minix scheduling:

Ready Queue

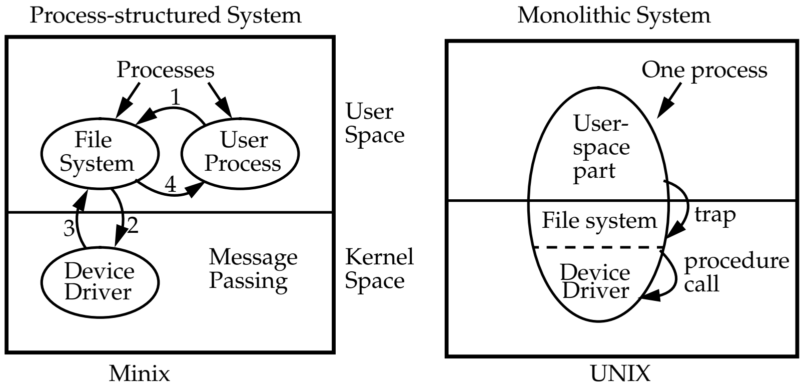


User processes scheduled round robin.

Server and Tasks run until they block.

Lower layers have higher priority. Layer 2 processes run first (until they block) followed by layer 3 processes (until they block). Layer 4 processes either block or are preempted.

Minix vs. UNIX



Process-structured system:

File system process contains all device-independent I/O code.

- Interfaces with drivers (tasks) using standardized interface
- Does buffering
- Block allocation

Device Drivers in Minix

Device drivers (tasks) are full-fledged processes.

Device drivers communicate with the file system and with each other using message passing.

Hard disks, floppy disks and RAM disk device drivers have two parts:

- Hardware-independent part; share a set of common routines.
- Hardware-dependent part; each have a separate source file and run as a separate process.

Main difference between device drivers and 'normal' processes:

Share a common address space (data).
Share common procedures, and must therefore be re-entrant.

In Minix, disk tasks are sequential for simplicity.

They issue a hardware request to the controller (on behalf of the user).
They do a RECEIVE operation waiting for an interrupt message.
New requests for work (from user processes) are kept waiting.

Block device drivers or tasks - Disk driver task:

One copy of main loop code in the Minix kernel.
Each driver passes to the main loop a parameter.

ep (entry_points) pointer to a table of addresses of functions used for indirect function calls:

```
message m;  /* Message buffer */
void  shared_io_task( struct driver_table *ep )
/* Initialization done by each task beforehand. */
   while (TRUE)  /* Loop forever */
      {
      receive(ANY, &m);
      caller = m.source;
      switch ( m.type )
         {
         case READ:  rcode =(*ep->dev_read)(&m);break;
         case WRITE: rcode =(*ep->dev_write)(&m);break;
/* Other cases go here, including OPEN, CLOSE etc. */
         default: rcode = ERROR;
      ...
```

Each of processes at different points in the code but each has its own data/stack.

Device operations:

OPEN
CLOSE
READ
WRITE
IOCTL:

Change device operational parameters (e.g. for terminal devices, the baud or parity).

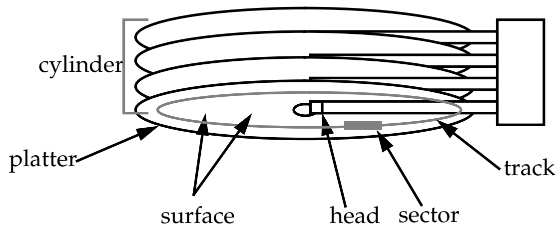
SCATTERED_IO:

OS anticipates future requests for data. Device driver given option of retrieving more blocks than requested.

Disk Hardware:

Disk components:

- platter: the metal (or plastic) disks
- surface: oxide on one or both sides of the disk
- head: magnetic coil that moves over the surface
- track: path on surface under an (unmoving) head
- sectors: sections into which a track is divided
- cylinder: a stack of tracks



A **disk block** is usually just a sector. (Sometimes disk blocks are groups of sectors; we will use the terms synonymously, here).
A sector (or block) is the smallest addressable unit on a disk.

A disk block address has three components

- a = sector number
- b = surface (head)
- c = cylinder (or track, since heads move together)

Suppose there are:

s sectors per track; t tracks per cylinder

Then the sector (or block) address is

c * s * t + b * s + a

For example, suppose we have a disk with

s = 17 sectors per track and

t = 10 tracks per cylinder

The block with disk address 542 can be found at

c = cylinder = 542 / s * t = 3

b = surface = (542 mod s * t) / s = 32 / 17 = 1

a = sector = (542 mod s) = 15

Checking, we have 3 * 17 * 10 + 1 * 17 + 15 = 542

IDE (Integrated Device Electronics)

Actual disk has more sectors per track on the outer track than on the inner tracks, e.g,

3000 cylinders, 4 heads, and 57-105 sectors in 12 zones.

Control electronics masks these differences and driver 'sees' **physical** representation -> same number of sectors on each track, e.g.,

1048 cylinders, 4 heads, 255 sectors.

Additional translation due to ROM BIOS limitations, 6 bit field allows max of 63 sectors, **logical** representation, e.g.,

1048 cylinders, 16 heads, 63 sectors.

Other disk types:

MFM

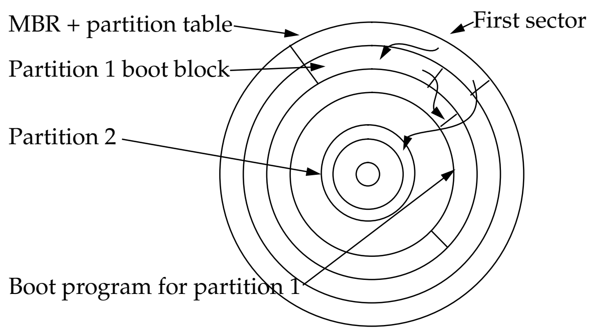
SCSI

ESDI

Disk Partitioning:

Hard disk is physically divided into a number of distinct partitions.

Partitions are generally made at cylinder boundaries.



ROM reads first sector and executes code found there which reads partition table and loads boot block of active partition. Boot block loads the boot program.

Reasons for partitioning:

- Disk capacity is cheaper per unit in large disks. If you want multiple OSs, you can create multiple partitions, one for each OS,
  e.g. Windows95, Linux and MINIX
- OSs have limits to the device size they can handle. Partitioning allows otherwise wasted disk space to be made usable.
  e.g., MINIX can not address disk partitions larger than 1 GB.
- Two different file systems may be used by one OS.
  e.g. ext2fs and swap for Linux.
- The same file system may be partitioned to make it convenient for backups and space management, e.g., ext2fs
  / (partition 1 contains system software, kernel, libraries, utilities, etc.)
  /home (partition 2 contains user home directories)

Typically a disk has 4 primary partitions.

Extended partitions act as containers that permit the creation of more than 4 partitions, e.g. logical partitions.

Disk Read/Write Time:

Seek time: Time to move head over cylinder.
Latency: Time for sector to rotate under head.
Data transfer time: time to read the desired sector(s) which is a function of disk density and rotational speed.

Seek time dominates.

Disk Arm Scheduling Algorithms:

Objective: Optimize the seek time.

FCFS: No optimization.

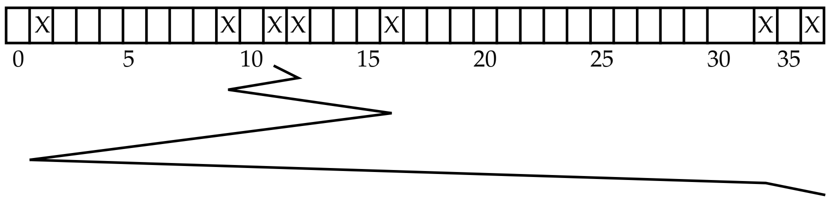
Alternatively, multiple requests (from multiple processes) can be queued up in a table indexed by cylinder number.

Shortest Seek First algorithm selects closest cylinder (nearest neighbor) to service next.

For example,

Assume sector 11 is requested.

Over the time period defined by the seek operation, assume requests 1, 36, 16, 34, 9 and 12 arrive in that order.



FCFS moves over 111 cylinders to service all requests.
Shortest Seek First moves over 61 cylinders.
Problem:

If requests keep arriving, arm will tend to remain in the middle of the disk

Cylinder requests along the edges will be starved (assuming accesses are uniformly distributed).

Solution:

Elevator algorithm: Keep moving arm in one direction until edge of disk is reached and then switch directions.

For example, service order for previous requests would occur as,

11, 12, 16, 34, 36, 9 and 1 = 60 cylinder moves.

In general, the algorithm usually requires more cylinder moves than SSF.

Disk Error Handling:

Sources of error:

- Programming error: request for a non-existent cylinder/track/sector. Usually handled by controller terminating request with an error.

- Transient error in CRC: Controller makes error transparent by retrying.

- Permanent error in CRC: Could handle using a bad block file where a file is created that is composed of all the bad blocks on the disk.

  Problem #1: Back-up software that backs up the disk by reading tracks will read the bad block file resulting in an error.

  Problem #2: Bad block file strategy does not work if bad block occurs in the first cylinder since almost all file systems have at least one data structure that occupies a fixed location on the disk.

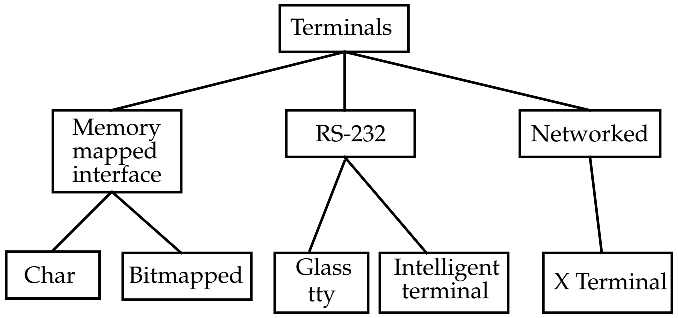
  Other solutions:

  Intelligent controllers use redundant tracks.
  New errors handled transparently by controller which substitutes blocks in for the unrecoverable errors.
  In these cases, the driver software rarely sees any bad blocks.

- Seek errors: formatting causes cylinder numbers to be written in sector of cylinders therefore the controller can verify the arms position.

Terminals

3 categories based on how OS communicates with them:

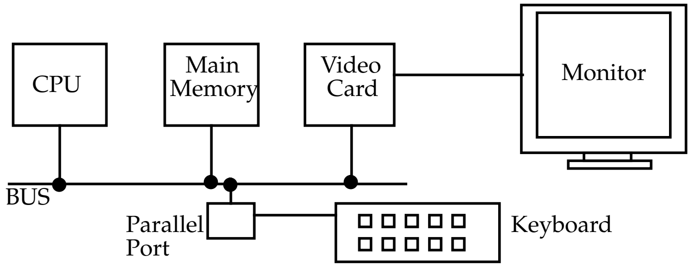


Memory mapped: Interfaced via a special memory called **video RAM** which is part of the computers address space.

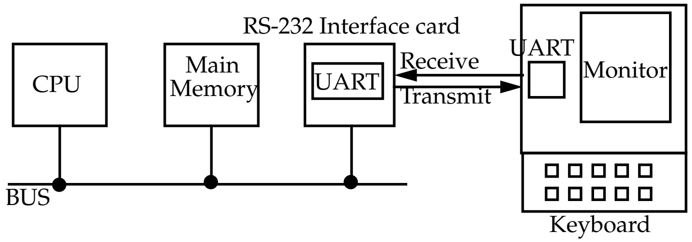
RS-232: a keyboard and character oriented display, communicating over a serial line.

An **Xterm** is a virtual RS232 terminal.

Memory Mapped configuration:



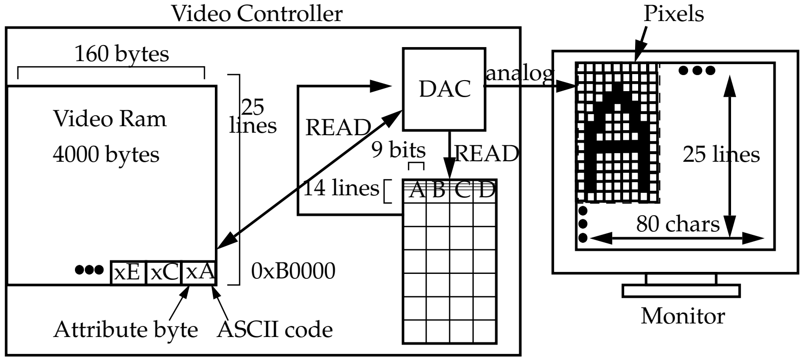
RS-232 terminal configuration:



Memory Mapped: Video controller contains video RAM and a chip (DAC) that translates video ram bits to analog video signals which drive the display or monitor.

Character Oriented Displays:

For character displays, characters are represented by, e.g., 9X14 pixel boxes.

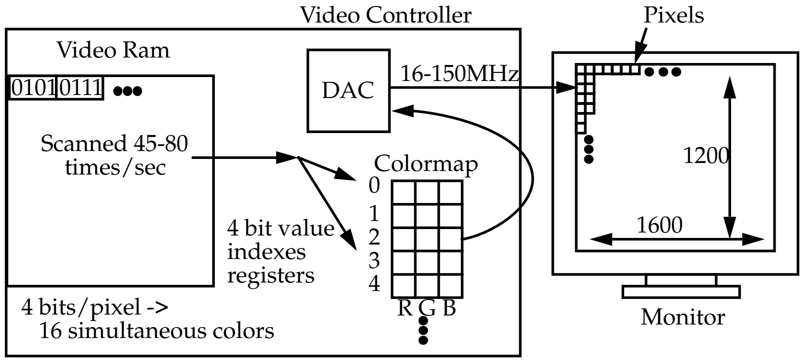


8 bits of address (ASCII code) combined with 4 bits to address each of the 9 bit scan lines in Char ROM.

Bit-mapped Displays:

In bit-mapped displays, special `dual ported' memory is often used, that can be both:

- Written or read as regular memory.
- Read by the display as pixels, with one or more bits/pixel.



To modify the screen one simply writes to the video ram.

Fonts and graphics are done in software.

RS-232 Terminals:

Keyboard and display communicate with the CPU using a serial interface (one bit at a time).

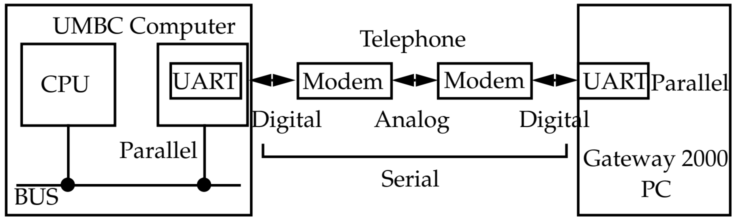
Glass ttys: ttys with a screen. The tty driver for a terminal must manage both the screen and keyboard.

Intelligent: Understand escape sequences which allows cursor control, scrolling, clear line, clear screen, etc.

RS-232 **connector** has 25 pins but only a subset of the pins are actually connected (receive, transmit and ground wires) and used to transmit data.

RS-232 is a **protocol** used between many sorts of devices (e.g., 2 computers, computer & terminal, terminal & modem).

RS-232 protocol is used between modems and UART:



UART (Universal Asynchronous Receiver Transmitter) used at each end which convert serial bit stream into bytes and vise versa.

Baud rates vary from 1,200 to 38,400.

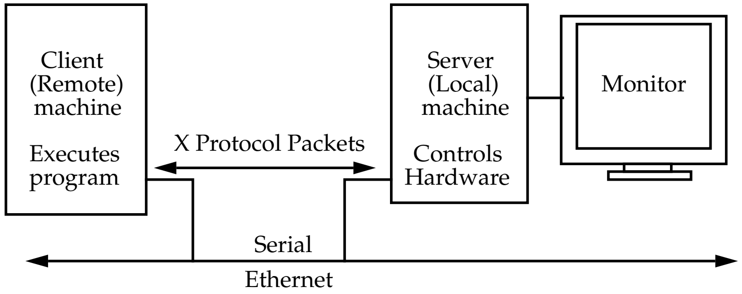
In the simplest case (no buffering):

1 interrupt at each received char.

1 interrupt after each successful transmission.

Typically, the serial card can buffer a small number of characters.

X Terminals:



X Server runs on local machine.

- Interprets packets sent by client program (possibly running on a remote machine).
- Converts client commands into the appropriate hardware commands to carry out requests of clients, e.g., draw blue rectangle at coordinates x, y.
- Converts keystrokes and mouse clicks into X packets to send to client.

Terminal Software:

Job of keyboard driver is to collect input from the keyboard and pass it to user programs.

Two philosophies can be adopted for the drivers:

- **Raw mode:** Pass each ASCII character along to user program as it is typed.
  Program manages updates (backspaces, etc.) that the user requests.
- **Canonical (cooked) mode**: Driver handles waits for carriage return before providing characters to user programs.
  Driver handles intraline editing and program receives only the corrected line.

In either case, typing a character at the keyboard generates an interrupt.

The ISR can read the hardware port directly and pass the key code to the driver.

The driver converts the key code to an ASCII value using a table.

Loadable keymaps allow the user to select choose the mapping between key codes and codes delivered to the application.

In either mode, we need buffers.

Raw: To handle typeahead for programs that have not yet requested a character.

Canonical: To allow intraline editing by the driver.

Most modern terminals do **not** display (echo) characters typed to the screen directly in hardware since it limits the flexability of the system, e.g., passwords are displayed.

Rather, the software or driver does the echoing.

Driver responsible for:

- Converting carriage return/line feed sequences from keyboard into proper screen operations.
- Expanding tabs.
- Wrapping lines.
- Accounting for delays in screen operations, e.g., scrolling.

Software to manage the keyboard and screen gets quite complex, e.g., there exists 30 times more terminal I/O code than scheduling code in MINIX.

Clocks

Two types of clocks:

Simple clock tied to the power supply, interrupt generated every 60 Hz.

Quartz clock oscillates from 5 to 100MHz:

Ticks decrement a register.

When register is zero, an interrupt is generated.

Register reloaded with value in holding register. Holding register can be set to value of time slice in nanoseconds.

Device driver software responsible for

Maintaining the time of day.

Preempting processes by calling scheduler.

Accounting for CPU usage.

Handling the ALARM system call, sends signal to a process after x secs.

Provide watchdog timers, e.g., for floppy disk spin-up delays.

Accounting and run-time profiling of code; may want to keep track of how much of each run burst is actually used.

Scheduling programs to run at specific times (e.g., UNIX chron utilities).