Input and Output

Goal

To learn how input and output are done, and how the medium being used affects the operations

Kernel-Level I/O Routines

Moving data to or from secondary storage is done by kernel routiness called *device drivers*. Each device driver is associated with one (type of) device, and all processes access the same set of device drivers by making appropriate system calls. This leads to the issue of how the processes view devices; the most basic issue is *transparency* ; the processes don't care about the manufacturer, the model of the device, and in some cases, the type of the device (*i.e.*, whether it is a printer, tape drive, or another process).
example: virtual devices: these are devices simulated by the kernel, with data kept either in the main store or elsewhere; for example, IBM's VM/CMS partitions the real disks into much smaller *minidisks*, and stores data on those.
example: a printing spooler system; the program may think it is talking to a printer, but in reality it is writing the data to a disk, from which it will be sent to the printer.

We shall examine the issues involved in several steps:

(1) *goals*; what should a good process/device interface do?
(2) *device hardware*; what does a device look like?
(3) *device interface*; how are the devices connected to the computer?
(4) *device drivers*; what do the kernel modules interacting with devices look like?
(5) *process interface*; how do the processes access devices?

Goals of Kernel-Level I/O Routines

(1) Character code independence
The kernel I/O subsystem must translate character codes from various different devices to a uniform internal representation; this is done by the kernel just after the characters arrive in memory and before they are passed to the process so that the programmer need not worry about it.
  • internal codes may be ASCII, EBCDIC, or something else.
(2) Device Independence
The process should not depend on one particular device; the operating system should be free to assign any device of the right type as appropriate; *ie*, the process should not need to say “lp0” to get printer number 0, but should be able to say “lp” and let the kernel select the printer.

- It is very desireable that as far as possible, programs should be independent of the type of the device; *i.e.*, it should not matter if input is taken from a tape, a disk, or a card reader.

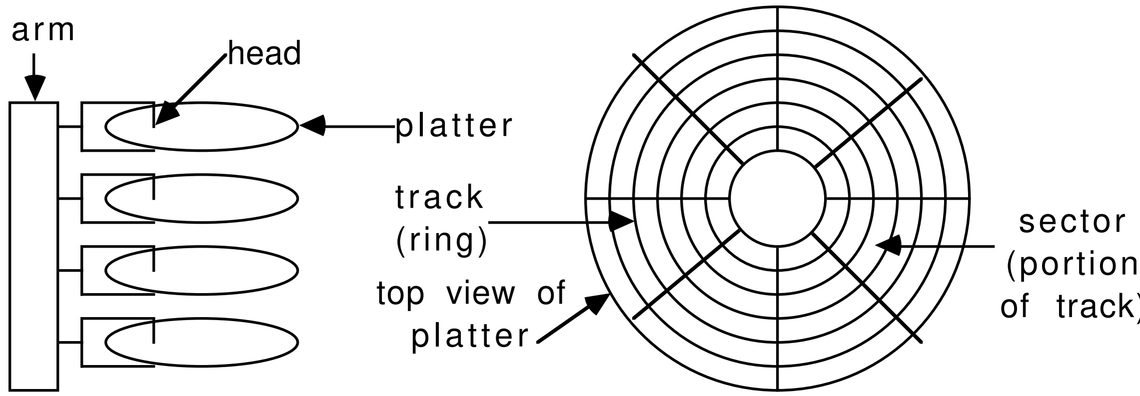
(3) Efficiency
I/O operations are often a bottleneck; this should be minimized.

(4) Uniform treatment of devices
The intent is to keep device handling simple and error-free; but it may be difficult in practice to handle all devices alike.

Device Hardware

Disks

Disks are collected in a pack like stacked phonograph records:



A *cylinder* is the same track on all disks in the pack (think of chopping out one track on each of the platters by cutting straight down, and you'll see where the name comes from). It is relevant because the arm moves all the heads over the tracks with the same number in the different platters.

Characteristics are:

- data transfer rate: 2 megabits/sec
- data is transferred in blocks with between 256 and 1024 bytes (typically); data is *always* transferred in multiples of a block.
- the platters spin at 60 rotations per second.
- each platter typically has 200 tracks (but some have more; the IBM 3380 has 900 tracks/platter, the new Winchesters have 1500.)
- each track typically has 30 sectors.

Floppies consist of one removable platter, and rotate more slowly than larger disks.

A *sector* contains:

- data
- a bit indicating whether or not the sector is usable
- error checking information
- (possibly) the sector number.

A *cylinder* is simply the set of tracks with the same track number on all platters.

The operating system needs to know how the disk is formatted:

- the number of sectors per track
- the number of bytes per sector

The operating system may do formatting when the disk brought into service; it can also discover bad sectors and mark them as unusable.

To read data, the operating system first *seeks for* the data by positioning the disk arm on the track on which the data sits; typically takes 20-40 ms. Hence there are three *latencies* (delays) relevant:
(1) *seek latency*: how long does it take the heads to get to the desired cylinder?
(2) *rotational latency*: once the heads are over the right track, how long does it take the right sector to rotate under the heads?
(3) *transfer latency*: once the heads are over the right sector, how long does it take to transfer data to or from the disk?

Drums

A drum is a cylinder divided into circular tracks; it is just like a disk except that there is one head per track, so the heads don't move; this eliminates seek latency.

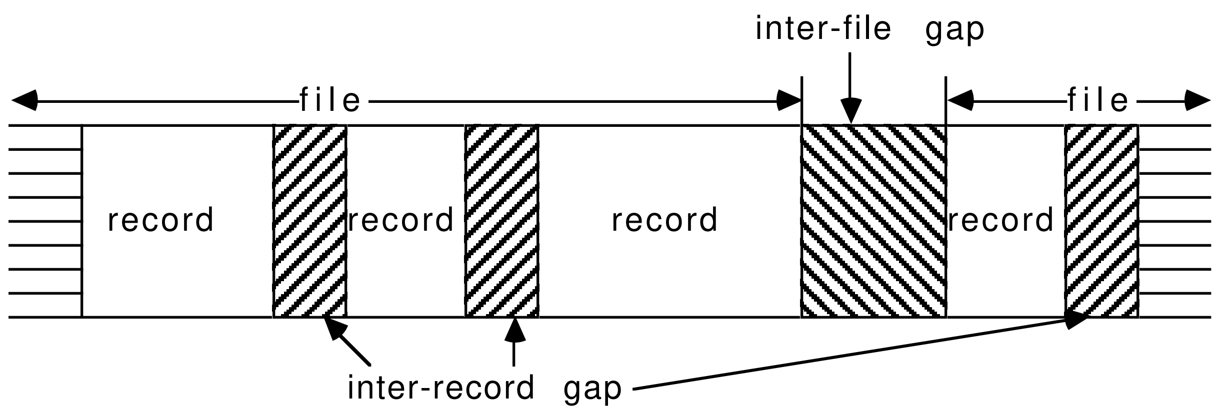
When drums are used, it is typically for swap space or as a backing store; nowadays, disks or banks of memory chips are used instead.

Magnetic tapes

Magnetic tapes are used for archiving data, transferring data between computers, and for intermediate storage of large amounts of data. They are usually <u>very</u> portable between different operating systems and machine architectures.

Some relevant characteristics:
- there are 9 regions across the width of the tape (hence the term *nine-track tape*); one bit is stored per region, so you can store 8 bits for the character and 1 parity bit. This is called a *frame*.
- *Tape density* is the number of frames per inch; usually 1600 or 6350 frames per inch. Density is the same throughout the tape.

Frames are grouped into records and records are grouped into files:



Records may be of any size, and successive records may have different lengths; so, when the tape is being reading, there must be enough room in the buffer to hold entire records.

Note you can only write at the end of a tape safely because the inter-record gaps are *not* of reproducible size. So writing in the middle of the tape overwrites a record followed by an inter-record gap, the latter of which may overwrite part of the next record.

Tapes usually contain a *label*, which is an initial record containing information that describes the tape's contents, its owner, its serial number, etc. Similarly, *headers* and *trailers* may surround files.

To read data, the operating system first *seeks for* the data by winding the tape. So there are two *latencies* (delays) relevant:

(1) *winding latency*: how long does it take to wind the tape to the desired place?

(2) *transfer latency*: once the heads are over the right sector, how long does it take to transfer data to or from the disk? (The times here are comparable to those of disks; ~2 megabits per second)

Communications lines

There are three types of lines:

- *simplex* lines transmit data in only one direction;
- *half-duplex* lines can transmit data in either direction, but only one direction at a time; and
- *duplex* lines can transmit data in both direcitons simultaneously.

Some relevant characteristics:

- *baud* is the number of electrical transitions per second on the line, which is usually (but not always) equivalent to “bits per second;” typical baud rates are:
  - for a terminal line: 110, 300, 1200, 2400, 4800, 9600, 19200
  - for a leased line (computer to computer): 56000
  - for an ethernet: 3 to 10 Megabaud
  - for a Hyperchannel: 50 Megabaud

  ASCII character codes take 7 or 8 bits; the remaining 2 or 3 bits are used as parity checks and to synchronize the sender and receiver.

All conversing parties must agree on conventions (called *protocols*) for formatting information, interpreting messages, etc. For synchronous transmissions, the sender and receiver share a common clock and know when to sample the wire for transmission.

- Each transmission will be preceded by a header containing a prearranged pattern of bits which enables the receiver to adjust its clock to match that of the sender.
- Transmissions are split into frames each of which is preceded by a header
- The header pattern cannot appear in the message, so:
  - if the transmission is bit-oriented, put an extra bit in; this bit will be stripped when the message is read (called *buit stuffing*)
  - if the transmission muct have multiples of some byte size, the transmission is character-oriented, so stuff a byte (called the *escape character*) instead of a bit. As an example, the BISYNC protocol sends messages with the following format:

    <SYN><SYN><SOH>header<STX>text<ETX>checksum

    The escape character is <DLE>; hence if the character <ETX> appears in the text, it must be escaped by sending <DLE><ETX> or else the receiver will mistake it for the end oif the text and become confused when it interprets text as the checksum.

Device Interface

The device interface is the lowest level of interaction between the machine and the I/O device; the device driver sits directly above it. The interface mechanism are the *device registers*, which are used for:

- transferring status information from the device to the CPU;
- transferring instructions from the CPU to the device;
- transferring data from the device to the CPU;
- transferring data from the CPU to the device.

Review the PDP-11 polling and interrupt-driven I/O schemes quickly. Complex devices are usually connected to a *controller*, and the controller to the CPU. The controller monitors the device status, knows the format of the medium, etc., and accepts orders from the CPU as well as accepting or returning data.

Channels are subsidiary CPU's that use a different machine language; the instructions are *commands* and sequences of commands are *channel programs*, which usually are stored in the main store as used by the CPU. They typically use DMA. *Command chaining* is the ability to send (or take) channel programs containing more than one command. The term *data chaining*(also called *scatter-gather*) is the ability to gather output data from, or scatter input data to, many places. A *selector channel* manages many devices, of which only one may transfer data at a time. A *multiplier channel* manages many devices, all of which may transfer data simultaneously.

Device Drivers

These serve three functions:

- they try to put a regular structure on those parts of the operating system that interact with devices;
- they provide a standard interface to the rest of the kernel;
- they serve the rest of the operating system and the device.

Think of a device driver as having two parts: the *lower* part deals with the device itself, and the *upper* part with the rest of the operating system:

- The upper part accepts requests from the operating system (*eg.*, the storage manager asks it to write out a page to backing store); this part then transforms these requests into entries in a pending work list for the lower part.
- The lower part wakes up when there is an interrupt, or when new work is added to the pending work list. When awakened, it disables other interrupts from the same device.

Example: Clock device driver

There are two types of clock devices:

line clock: generates an interrupt every tick ($\frac{1}{50}$ or $\frac{1}{60}$ second)

- It may have a register that counts ticks since the last reset.
- It has a backup battery (to handle short-term power failures).
- It may have a register counting ticks missed if interrupts are not serviced by the CPU (for priority reasons).

programmable clock:

- a *count register* can be set by software
- subtract 1 from the register per time interval (say, ms or tick)
- when the count register contains 0, an interrupt occurs

When an interrupt occurs, the following steps are done:

(1) the system's software time structure is incremented;
(2) if the clock is used for scheduling, decrement the remaining time field of the current job is decremented and if 0, the scheduler is invoked;
(3) accounting is done;
(4) if there is no programmable clock, decrement a counter for the next alarm; if this counter is 0, any kernel routine waiting for an alarm is invoked.
(5) if the current process is being profiled, figure out where it is (by looking at the program counter), and update the appropriate counter.
(6) return to interrupted process

Example: Disk device driver

These must provide the illusion of a virtual disk which is a linear array of sectors; to do this, the sectors are numbered like elements of an array. Thus, sector s on track t in cylinder c is numbered:

$$a = ((c \times (\#tracks/cylinder) + t) \times (\#sectors/track) + s$$

so other parts of the kernel can write to sector number a, rather than sector number (c, t, s).

The other requirement is that the disk driver reduce the effect of the latencies inherent in accessing the disk; this is typically done by:

(1) overlapping I/O and computation;
(2) storing large objects in contiguous regions on the disk, so once the first seek is done, no more seeks are needed to write out the object; and
(3) ordering outstanding disk requests.

The last is particularly important. Assume:

- only one disk drive (if there are more, schedule them separately);
- all I/O requests are for single, equal-sized blocks;
- the requested blocks are distributed randomly over the disk pack;
- the disk drive has only 1 moveable arm, with all heads on it
- the seek latency is linear in the number of tracks crossed (this is not true if the disk controller uses replaces bad sectors with those in tracks at the end of the disk);
- the disk controller does not introduce appreciable delays; and
- read and write requests are identically fast.

Evauation of policies involves considering:

- how long requests must wait as a function of load (*ie*, the frequency of requests, measured in requests per second)
- the mean and variance of the waiting time for each request (for example, a low mean but high variance means some requests will take a long time).

There are several common policies; the handout gives an example of how they work:

(1) *First Come First Served* (*FCFS*): no starvation of requests possible; all eventually get serviced. It has a fairly low variance but becomes saturated easily (*saturation* occurs when the load becomes greater than the driver can handle, so there are always requests waiting).

Problems:

- every request is likely to require a seek;
- for low loads, it works fine, but for high loads, the latencies increase the mean of the waiting time appreciably.

(2) *Backup*: this is FCFS, but on the way to the track where the next request lies, any queued requests lying on an intermediate track are serviced. For high loads, this decreases the mean waiting time a bit.

(3) *Shortest Seek (Time) First* (*SSF, SSTF*):Service the request lying on the closest track. It saturates at the highest load of any of these policies.
Problems:
- Starvation is possible, but means that the disk can't keep up with disk requests, which usually indicates other, more severe problems (specifically, thrashing).
- The innermost and outermost tracks are discriminated against, leading to a variance larger than that of FCFS.

(4) *SCAN*: the head moves from the outermost track to the innermost one, then back out, then back in, …, servicing requests on the way. This variant of SSF reduces the problem of discrimination against the outermost and innermost tracks, thereby lowering the variance.
Problems:
- It is still subject to starvation.

(5) N-STEP SCAN: like SCAN, but only requests waiting when the disk heads begin a sweep are serviced; all others wait until the next sweep. This prevents starvation and lowers the variance even further.

(6) C-SCAN: This is like SCAN except in one direction only (hence the name, *Circular SCAN*); in it the head moves from the outermost track to the innermost and then jumps back to the outermost track. This variant of SCAN eliminates the problem of discrimination against the outermost and innermost tracks, and provides more uniform waiting times.

In practise, the last three are implemented by having the head move only as far as there are outstanding requests in each direction (so if the first request is at track 7, the second at track 9, and the arm is moving outward, the arm will stop at track 7 and then change direction, rather than continue to the outermost track and reverse direction there. These variants are called *LOOK*, *N-Step LOOK*, and *C-LOOK* respectively.

At very heavy loads it is useful to use a scheduling policy to minimize rotational latency (called *sector queueing*). This involves ordering requests for the same track so all such requests can be written in a minimum number of rotations of the disk. In practise, each sector has its own queue, and requests for a given sector are put into the appropriate queue; when that sector rotates under the head, the first request in its queue is serviced. It is most often used with fixed-arm devices such as drums.

Process Interface

Underlying this interface is the concept of a *file*; this concept (usually) does not exist at the lower level. Files provide the means for processes to interact with devices, and in some cases data structures such as kernel memory or a sink (*/dev/null* in UNIX). Given this metaphor, the system needs one extra system call to handle actions that can't fit into the metaphor, such as changing speed on a terminal line.

System Calls

open the file

device descriptor = open(device name, intent)

- the descriptor can then be used in other references to the file;
- this call may block until the device is ready, or return an error code;
- this call also checks privileges.

close the file

close(device descriptor)

- the device driver does any needed clean-up (*eg*, rewind tape)

position the file pointer

position(device descriptor, where)

- position the read/write pointer associated with the device in a certain way; *eg*, skip over records on a tape or move to the end of a file

read the file

read(device descriptor, memory address, amount)

- this call transfers data from the device to memory;
- it reads at most amount, and may read less

write the file

write(device descriptor, memory address, amount)

- this call transfers data from memory to the device

miscellaneous control commands

control(device descriptor, code)

- this is the call for all actions not fitting into the metaphor, and is used to do device-specific things.

The read and write commands have two forms, *blocking* and *non-blocking*.

- a blocking transfer is synchronous; when the next statement executes, the data has been transferred yo or from memory.
- a nonblocking transfer is asynchronous; the transfer may or may not be complete when the next statement is executed. To determine when the transfer completes, one cal use *polling* or a *virtual interrupt*. With the latter, the process requests an

interrupt from the kernel when the transfer is finished. This requires the system call

handle(device descriptor, routine)

which instructs that when the asynchronous input finishes, the process is to be interrupted and routine executed.

If the read or write system call is non-blocking and the process needs to do blocking I/O, two system calls are needed:

wait(device descriptor, timeout)

blocks the process until transfer done or for timeout time, whichever is least.

When doing non-blocking I/O, **do not modify that portion of memory involved in the transfer** or the results of the transfer will be undefined! (Some kernels may copy data to their own buffers as part of the system call, before control returns to the process.)