V DIRECTORY IMPLEMENTATION:

Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

Linear List

A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks. Finding a file (or verifying one does not already exist upon creation) requires a linear search.

Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.

Sorting the list makes searches faster, at the expense of more complex insertions and deletions.

A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.

More complex data structures, such as B-trees, could also be considered.

Hash Table

A hash table can also be used to speed up searches.

Hash tables are generally implemented *in addition to* a linear or other structure

Allocation Methods:

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

Contiguous

Allocation Linked

Allocation

Indexed Allocation

The main idea behind these methods is to provide:

Efficient disk space utilization.

Fast access to the file blocks.

All the **three methods** have their own advantages and disadvantages as discussed below:

Contiguous Allocation

Contiguous Allocation requires that all blocks of a file be kept together contiguously.

Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.

Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc.) The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.

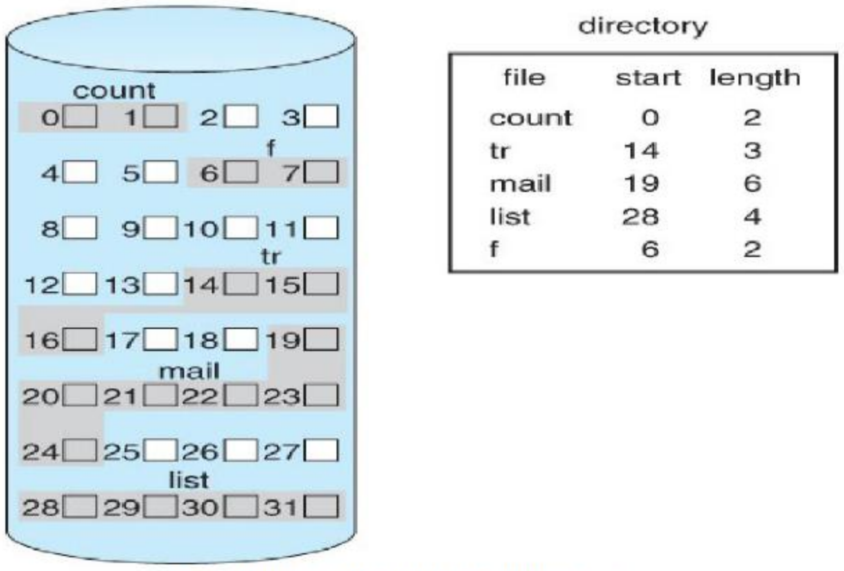
In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: b, b+1, b+2,……b+n-1.This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.

The directory entry for a file with contiguous allocation contains

Address of starting block

Length of the allocated portion.

The file 'mail' in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies 19, 20, 21, 22, 23, 24 blocks.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation of disk space.

Advantages:

Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).

This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

Disadvantages:

This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.

Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

Linked Allocation

Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)

Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.

Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.

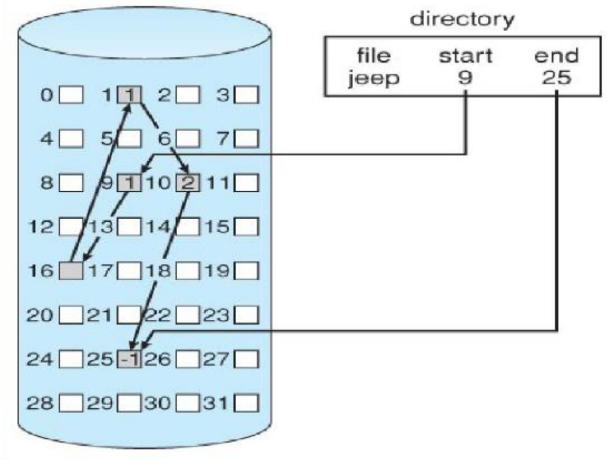
Allocating *clusters* of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.

Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.

In this scheme, each file is a linked list of disk blocks which need not be contiguous. The disk blocks can be scattered anywhere on the disk.

The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.



Linked allocation of disk space.

Advantages:

This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.

This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

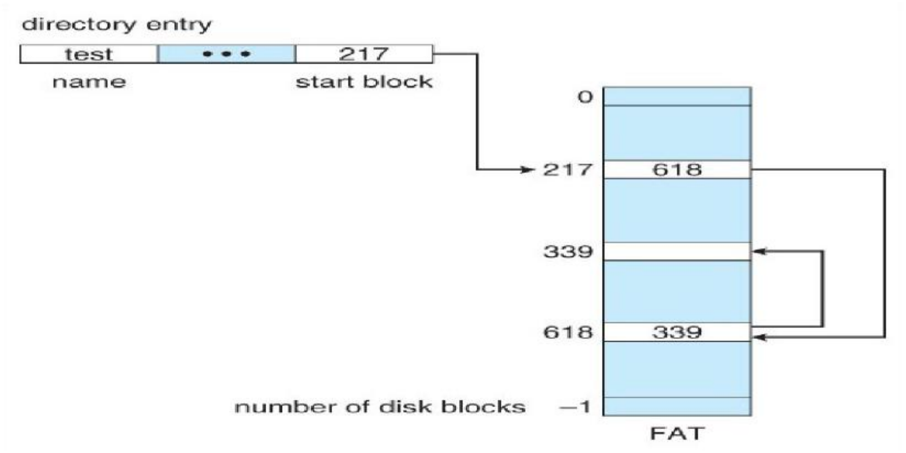
Disadvantages:

Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.

It does not support random or direct access. We cannot directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.

Pointers required in the linked allocation incur some extra overhead.

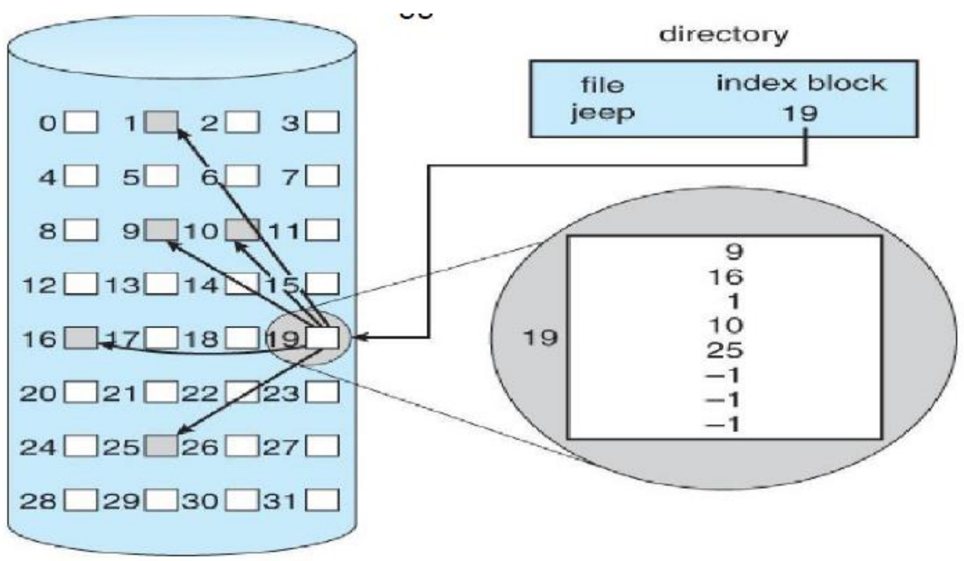
The ***File Allocation Table, FAT,*** used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.



File-allocation table.

Indexed Allocation

Indexed Allocation combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.



Indexed allocation of disk space.

Advantages:

This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.

It overcomes the problem of external fragmentation.

Disadvantages:

The pointer overhead for indexed allocation is greater than linked allocation.

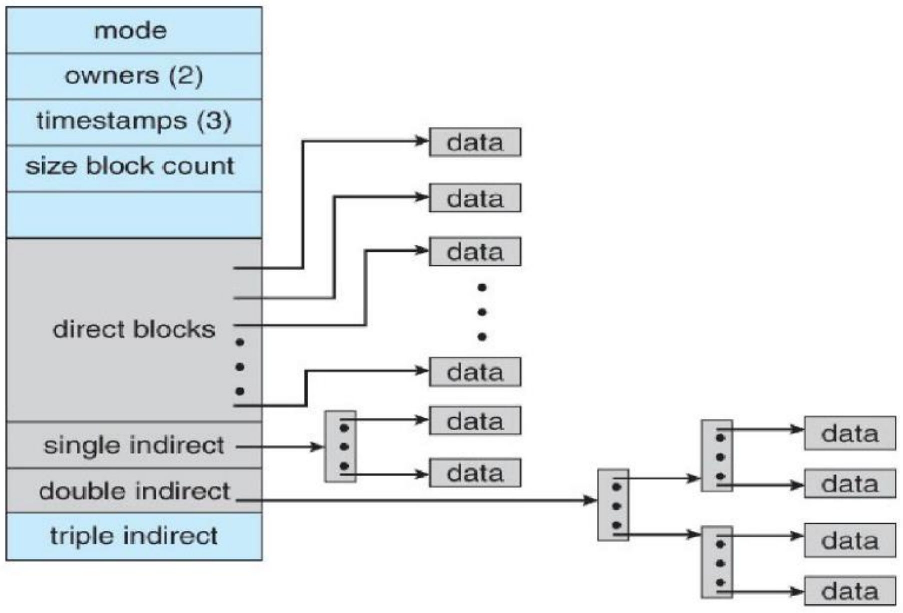
For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

Linked Scheme - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.

Multi-Level Index - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.

Combined Scheme - This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. (See below) The advantage of this scheme is that for small files (which many are), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)



The UNIX inode

Performance

The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.

Some systems support more than one allocation method, which may require specifying how the file is to be used (sequential or random access) at the time it is allocated. Such systems also provide conversion utilities. Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.

And of course some systems adjust their allocation schemes (e.g. block sizes) to best match the characteristics of the hardware for optimum performance.

Free Space Management:

Another important aspect of disk management is keeping track of and allocating free space.

Bit Vector

One simple approach is to use a *bit vector*, in which each bit represents a disk block, set to 1 if free or 0 if allocated.

Fast algorithms exist for quickly finding contiguous blocks of a given size

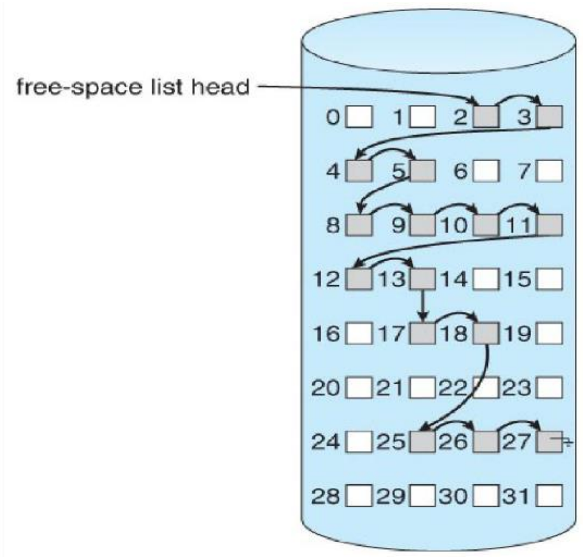
The down side is that a 40GB disk requires over 5MB just to store the bitmap. (For example.)

Linked List

A linked list can also be used to keep track of all free blocks.

Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.

The **FAT** table keeps track of the free list as just one more linked list on the table.



Linked free-space list on disk

Grouping

A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

Counting

When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered.)

Space Maps (New)

- Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems.
- The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) *metaslabs* of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

Efficiency and Performance:

Efficiency

- UNIX pre-allocates inodes, which occupies space even before any files are created. UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data.
- Some systems use variable size clusters depending on the file size.
- The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have to be re-written. As technology advances, addressing schemes have had to grow as well.

- Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

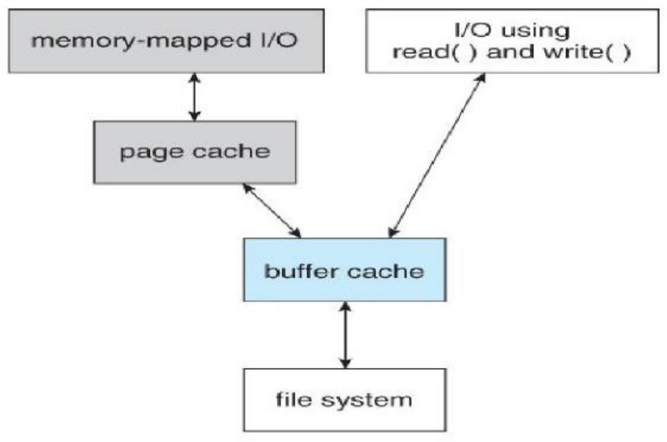
Performance

- Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads (reducing latency.) The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.
- Some OSes cache disk blocks they expect to need again in a ***buffer cache.***

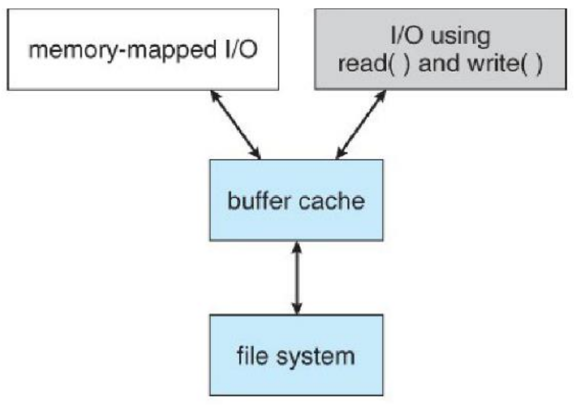
A ***page cache*** connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.

Some systems (Solaris, Linux, Windows 2000, NT, XP) use page caching for both process pages and file data in a ***unified virtual memory.***

Figures show the advantages of the *unified buffer cache* found in some versions of UNIX and Linux - Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.



I/O without a unified buffer cache.



I/O using a unified buffer cache.

Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages. Solaris, for example, has gone through many variations, resulting in *priority paging* giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.

Another issue affecting performance is the question of whether to implement *synchronous writes* or *asynchronous writes.*

On the other hand, we can expect to need the next page in the file fairly soon. For this reason sequential access files often take advantage of two special policies:

- ***Free-behind*** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
- *Read-ahead* reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future.

Recovery

Consistency Checking

- The storing of certain data structures (e.g. directories and inodes) in memory and the caching of disk operations can speed up performance, but what happens in the result of a system crash? All volatile memory structures are lost, and the information stored on the hard drive may be left in an inconsistent state.
- A ***Consistency Checker*** is often run at boot time or mount time, particularly if a filesystem was not closed down properly. Some of the problems that these tools look for include:
  - Disk blocks allocated to files and also listed on the free list.
  - Disk blocks neither allocated to files nor on the free list.

Disk blocks allocated to more than one file.

The number of disk blocks allocated to a file inconsistent with the file's stated size. Properly allocated files / inodes which do not appear in any directory entry.

Log-Structured File Systems

- *Log-based transaction-oriented* filesystems borrow techniques developed for databases, guaranteeing that any given transaction either completes successfully or can be rolled back to a safe state before the transaction commenced:
- All metadata changes are written sequentially to a log. A set of changes for performing a specific task (e.g. moving a file) is a *transaction*.
- As changes are written to the log they are said to be *committed*, allowing the system to return to its work. In the meantime, the changes from the log are carried out on the actual filesystem, and a pointer keeps track of which changes in the log have been completed and which have not yet been completed.

.

Other Solutions (New)

- Sun's ZFS and Network Appliance's WAFL file systems take a different approach to file system consistency. No blocks of data are ever over-written in place. Rather the new data is written into fresh new blocks, and after the transaction is complete, the metadata (data block pointers) is updated to point to the new blocks.
- The old blocks can then be freed up for future use.

- Alternatively, if the old blocks and old metadata are saved, then a *snapshot* of the system in its original state is preserved. This approach is taken by WAFL.
- ZFS combines this with check-summing of all metadata and data blocks, and RAID, to ensure that no inconsistencies are possible, and therefore ZFS does not incorporate a consistency checker.

Backup and Restore

- In order to recover lost data in the event of a disk crash, it is important to conduct backups regularly. Files should be copied to some removable medium, such as magnetic tapes, CDs, DVDs, or external removable hard drives.
- A full backup copies every file on a file system. Incremental backups copy only files which have changed since some previous time.
- A combination of full and incremental backups can offer a compromise between full recoverability, the number and size of backup tapes needed, and the number of tapes that need to be used to do a full restore. For example, one strategy might be:
  - At the beginning of the month do a full backup.
  - At the end of the first and again at the end of the second week, backup all files which have changed since the beginning of the month.
  - At the end of the third week, backup all files that have changed since the end of the second week.
  - Every day of the month not listed above, do an incremental backup of all files that have changed since the most recent of the weekly backups described above.

Backup tapes are often reused, particularly for daily backups, but there are limits to how many times the same tape can be used.

Every so often a full backup should be made that is kept "forever" and not overwritten.

Backup tapes should be tested, to ensure that they are readable!

For optimal security, backup tapes should be kept off-premises, so that a fire or burglary cannot destroy both the system and the backups. There are companies (e.g. Iron Mountain) that specialize in the secure off-site storage of critical backup information.

Keep your backup tapes secure - The easiest way for a thief to steal all your data is to simply pocket your backup tapes!

Storing important files on more than one computer can be an alternate though less reliable form of ***backup.***

I/O System

Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)

I/O Subsystems must contend with two trends: (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.

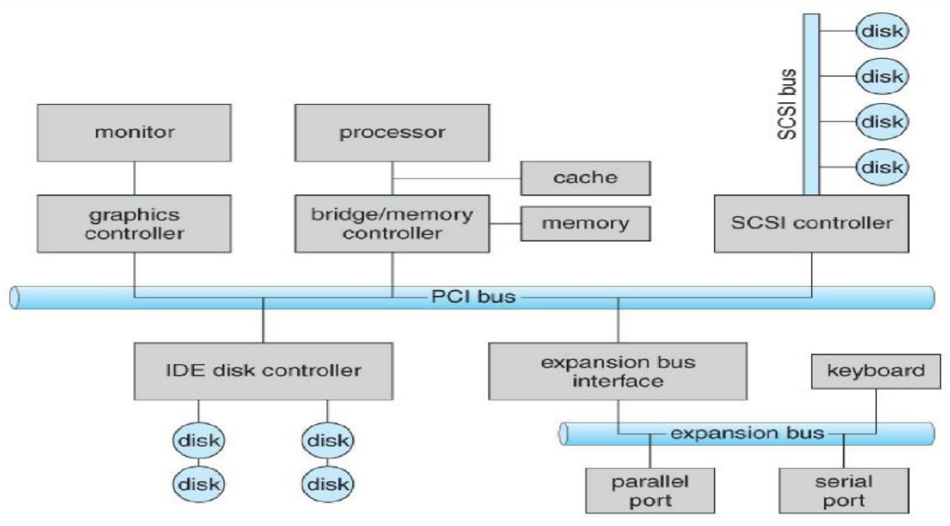
Device drivers are modules that can be plugged into an OS to handle a particular device or category of similar devices.

I/O Hardware:

I/O devices can be roughly categorized as storage, communications, user-interface, and other

Devices communicate with the computer via signals sent over wires or through the air. Devices connect with the computer via ***ports***, e.g. a serial or parallel port. A common set of wires connecting multiple devices is termed a ***bus.***

- Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
- Figure below illustrates three of the four bus types commonly found in a modern PC:
  1. The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.
  2. The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
  3. The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.
  4. A ***daisy-chain bus,*** (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.



<u>A typical PC bus structure</u>

One way of communicating with devices is through ***registers*** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:

1. The ***data-in register*** is read by the host to get input from the device.
2. The ***data-out register*** is written by the host to send output.
3. The ***status register*** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
4. The ***control register*** has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.

Figure shows some of the most common I/O port address ranges.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Device I/O port locations on PCs (partial).

Another technique for communicating with devices is ***memory-mapped I/O.***

- In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.
- Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
- Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.
- A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device.
- (Note: Memory-mapped I/O is not the same thing as direct memory access, DMA.)

Polling

One simple means of device *handshaking* involves polling:

1. The host repeatedly checks the *busy bit* on the device until it becomes clear.
2. The host writes a byte of data into the data-out register, and sets the *write bit* in the command register (in either order.)
3. The host sets the *command ready bit* in the command register to notify the device of the pending command.
4. When the device controller sees the command-ready bit set, it first sets the busy bit.
5. Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.
6. The device controller then clears the *error bit* in the status register, the command-ready bit, and finally clears the busy bit, signalling the completion of the operation.

Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

Interrupts

Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.

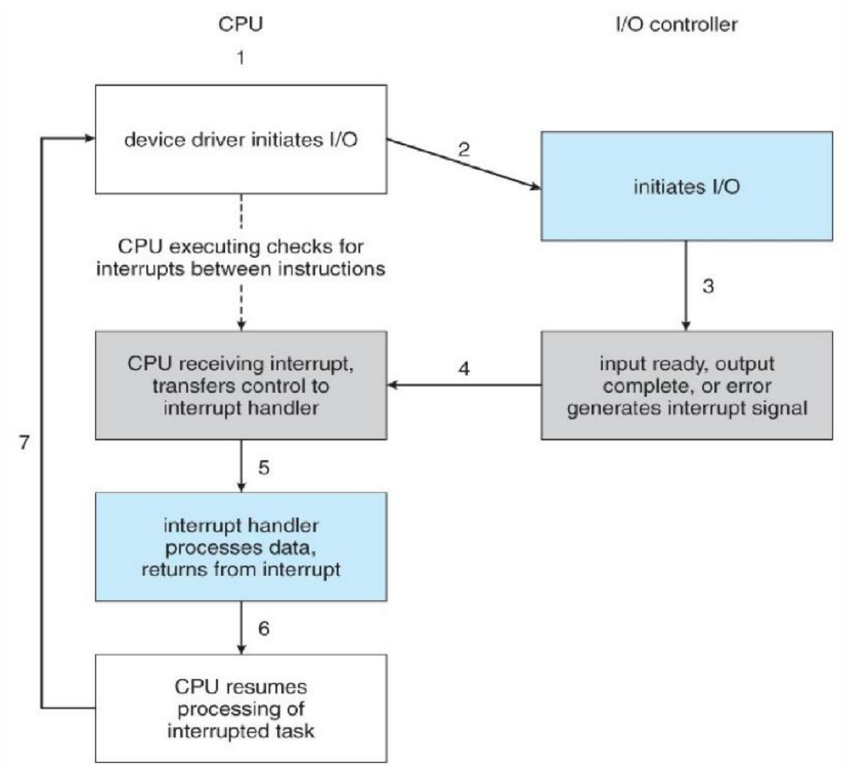
The CPU has an *interrupt-request line* that is sensed after every instruction.

- A device's controller *raises* an interrupt by asserting a signal on the interrupt request line.
- The CPU then performs a state save, and transfers control to the *interrupt handler* routine at a

fixed address in memory. (The CPU *catches* the interrupt and *dispatches* the interrupt handler.)

- The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a *return from interrupt* instruction to return control to the CPU. (The interrupt handler *clears* the interrupt by servicing the device.)

(Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing.)


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Interrupt-driven I/O procedure

<u>Interrupt-driven I/O cycle.</u>

The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:

1. The need to defer interrupt handling during critical processing,
2. The need to determine *which* interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.

These issues are handled in modern computer architectures with *interrupt-controller* hardware.

- Most CPUs now have two interrupt-request lines: One that is *non-maskable* for critical error conditions and one that is *maskable*, that the CPU can temporarily ignore during critical processing.
- The interrupt mechanism accepts an *address*, which is usually one of a small set of numbers for an offset into a table called the *interrupt vector*. This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.
- The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be *interrupt chained*. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
- Figure shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.Modern interrupt hardware also supports *interrupt priority levels*, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.
- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.
- During operation, devices signal errors or the completion of commands via interrupts

Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signalled via interrupts.

Time slicing and context switches can also be implemented using the interrupt mechanism.

- The scheduler sets a hardware timer before transferring control over to a user process.
- When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
- The scheduler does a state-restore of a *different* process before resetting the timer and issuing the return-from-interrupt instruction.

Direct Memory Access

For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.

Instead this work can be off-loaded to a special processor, known as the *Direct Memory Access, DMA, Controller.*

Below illustrates the **DMA** process.

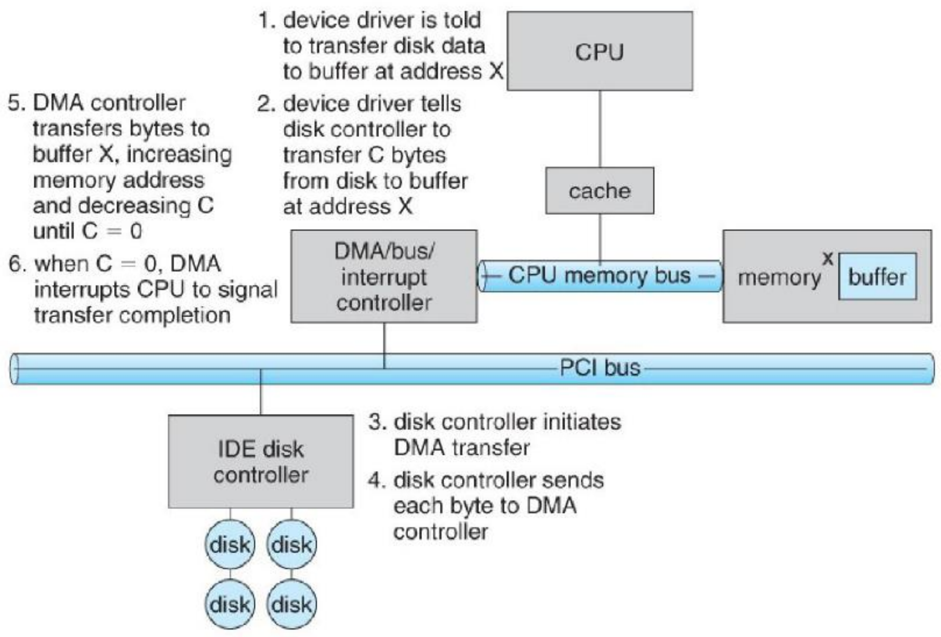
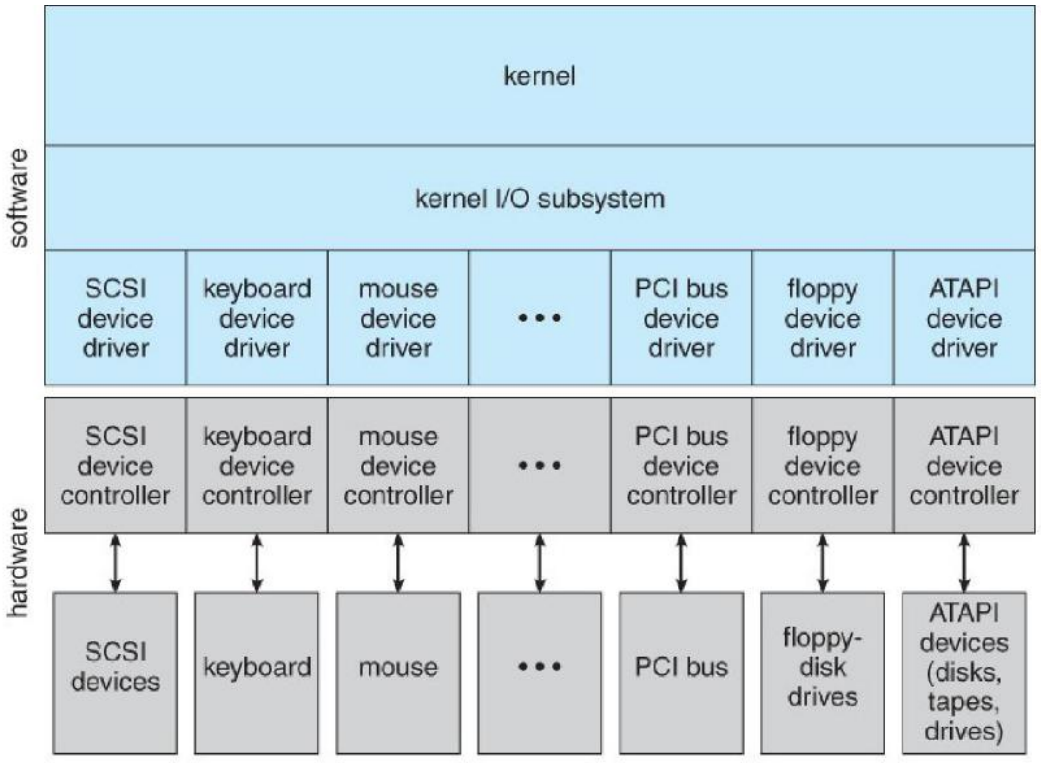

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
CPU
cache
DMA/bus/ interrupt controller
CPU memory bus
memory
x
buffer
PCI bus
IDE disk controller
disk
disk
disk
disk

Fig: Steps in a DMA transfer

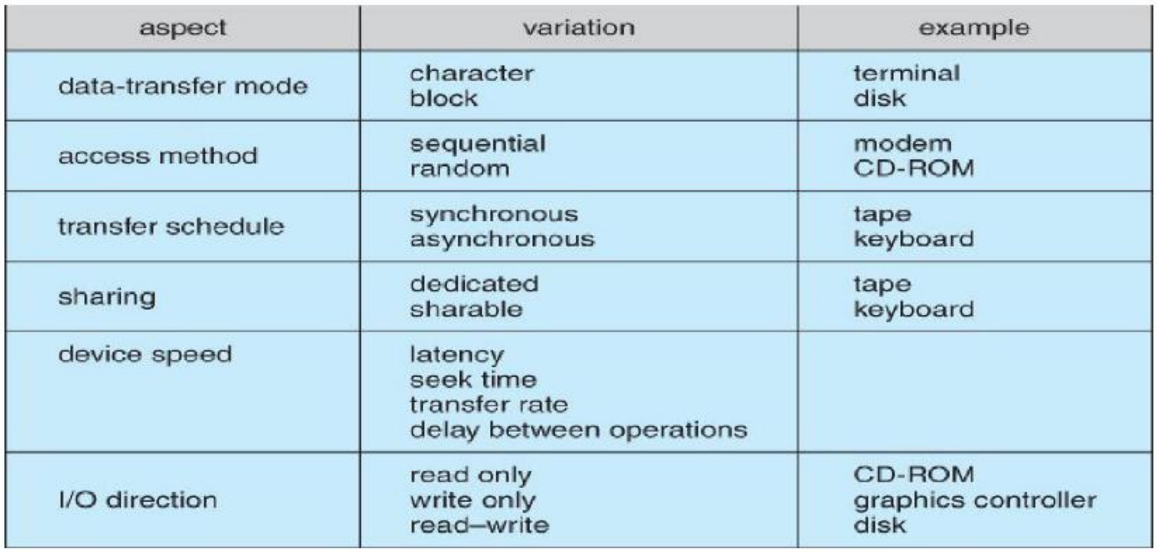
Application I/O interface:

User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into ***device drivers***, while application layers are presented with a common interface for all (or at least large general categories of) devices.



A kernel I/O structure.

Devices differ on many different dimensions, as outlined



aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Characteristics of I/O devices.

Block and Character Devices

Block devices are accessed a block at a time, and are indicated by a "b" as the first character in a long listing on UNIX systems. Operations supported include read( ), write( ), and seek( ).

- Accessing blocks on a hard drive directly (without going through the filesystem structure) is called ***raw I/O***, and can speed up certain operations by bypassing the buffering and locking normally conducted by the OS. (It then becomes the application's responsibility to manage those issues.)
- A new alternative is ***direct I/O,*** which uses the normal filesystem access, but which disables buffering and locking operations.

Character devices are accessed one byte at a time, and are indicated by a "c" in UNIX long listings. Supported operations include get( ) and put( ), with more advanced functionality such as reading an entire line supported by higher-level library routines.

Network Devices

Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.

One common and popular interface is the ***socket*** interface, which acts like a cable or pipeline connecting two networked entities. Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.

The select( ) system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

Clocks and Timers

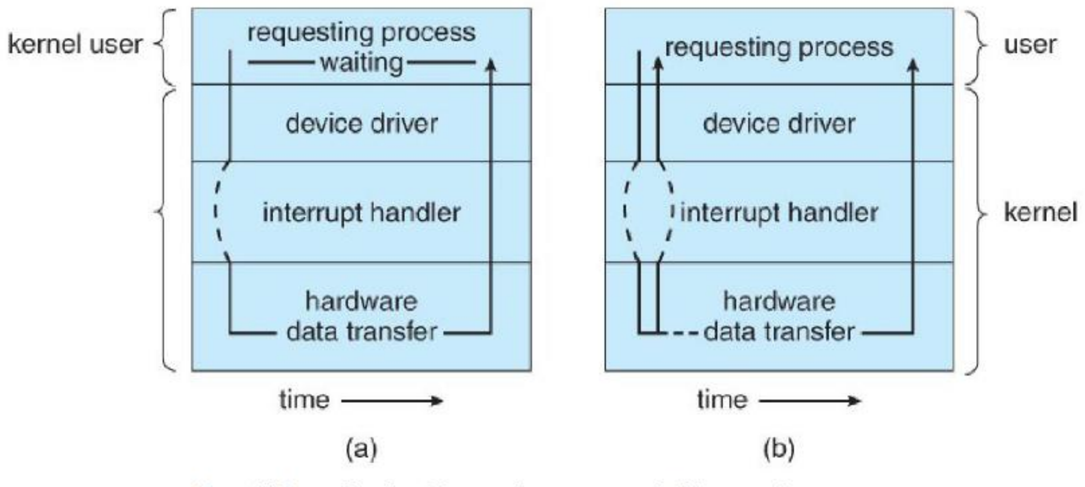
Three types of time services are commonly needed in modern systems: o Get the current time of day.

- Get the elapsed time (system or wall clock) since a previous event.
- Set a timer to trigger event X at time T.

Blocking and Non-blocking I/O

With ***blocking I/O*** a process is moved to the wait queue when an I/O request is made, and moved back to the ready queue when the request completes, allowing other processes to run in the meantime.

With ***non-blocking I/O*** the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not.



Two I/O methods: (a) synchronous and (b) asynchronous.