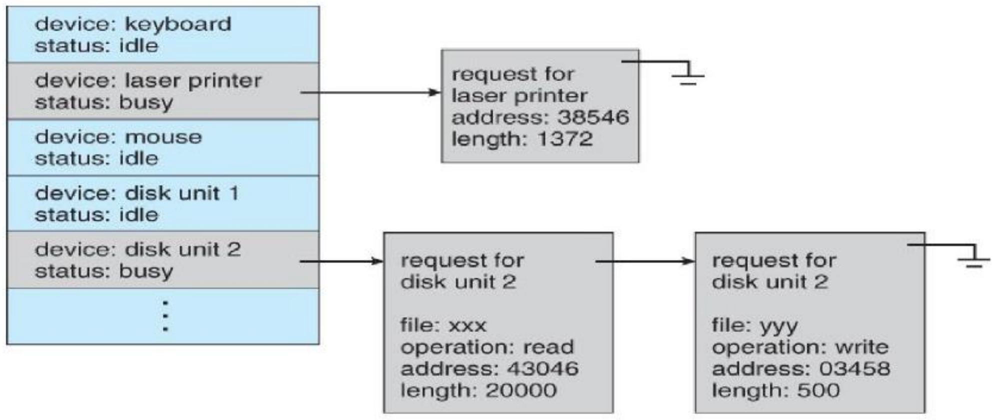
VI KERNAL I/O SYSTEM

I/O SCHEDULING

Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part in request scheduling. The classic example is the scheduling of disk accesses, as discussed in detail. Buffering and caching can also help, and can allow for more flexible scheduling options. On systems with many devices, separate request queues are often kept for each device:



Device-status table.

Buffering

Buffering of I/O is performed for (at least) 3 major reasons:

1.Speed differences between two devices. A slow device may write data into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once. So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full. This is known as *double buffering.*

2.Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side.

3.To support ***copy semantics***. For example, when an application makes a request for a disk write, the data is copied from the user's memory area into a kernel buffer. Now the application can change their copy of the data, but the data which eventually gets written out to disk is the version of the data at the time the write request was made.

Caching

Caching involves keeping a copy of data in a faster-access location than where the data is normally stored.

Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.

Buffering and caching go hand-in-hand, and often the same storage space may be used for both purposes. For example, after a buffer is written to disk, then the copy in memory can be used as a cached

copy, (until that buffer is needed for other purposes)

Spooling and Device Reservation

A *spool (Simultaneous Peripheral Operations On-Line)* buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.

If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.

Support is provided for viewing the spool queues, removing jobs from the queues, moving jobs from one queue to another queue, and in some cases changing the priorities of jobs in the queues.

Spool queues can be general (any laser printer) or specific (printer number 42.)

Error Handling

I/O requests can fail for many reasons, either transient (buffers overflow) or permanent (disk crash). I/O requests usually return an error bit (or more) indicating the problem. UNIX systems also set the global variable *errno* to one of a hundred or so well-defined values to indicate the specific error that has occurred. (Seeerrno.h for a complete listing, or man errno.)

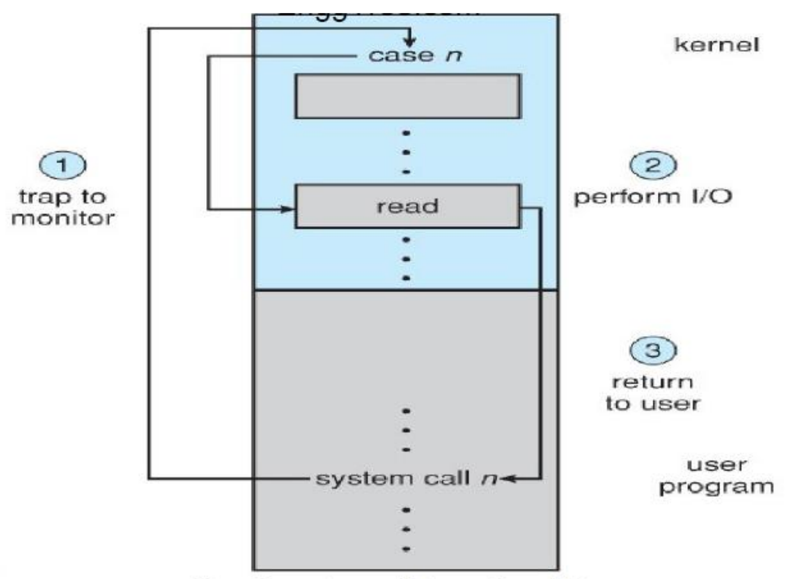
Some devices, such as SCSI devices, are capable of providing much more detailed information about errors, and even keep an on-board error log that can be requested by the host.

I/O Protection

The I/O system must protect against either accidental or deliberate erroneous I/O.

User applications are not allowed to perform I/O in user mode - All I/O requests are handled through system calls that must be performed in kernel mode.

Memory mapped areas and I/O ports must be protected by the memory management system, but access to these areas cannot be totally denied to user programs. (Video games and some other applications need to be able to write directly to video memory for optimal performance for example.) Instead the memory protection system restricts access so that only one process at a time can access particular parts of memory, such as the portion of the screen memory corresponding to a particular window.



Use of a system call to perform I/O.

Kernel Data Structures

The kernel maintains a number of important data structures pertaining to the I/O system, such as the open file table. These structures are object-oriented, and flexible to allow access to a wide variety of I/O devices through a common interface. (See Figure below.)Windows NT carries the object-orientation one step further, implementing I/O as a message-passing system from the source through various intermediaries to the device.

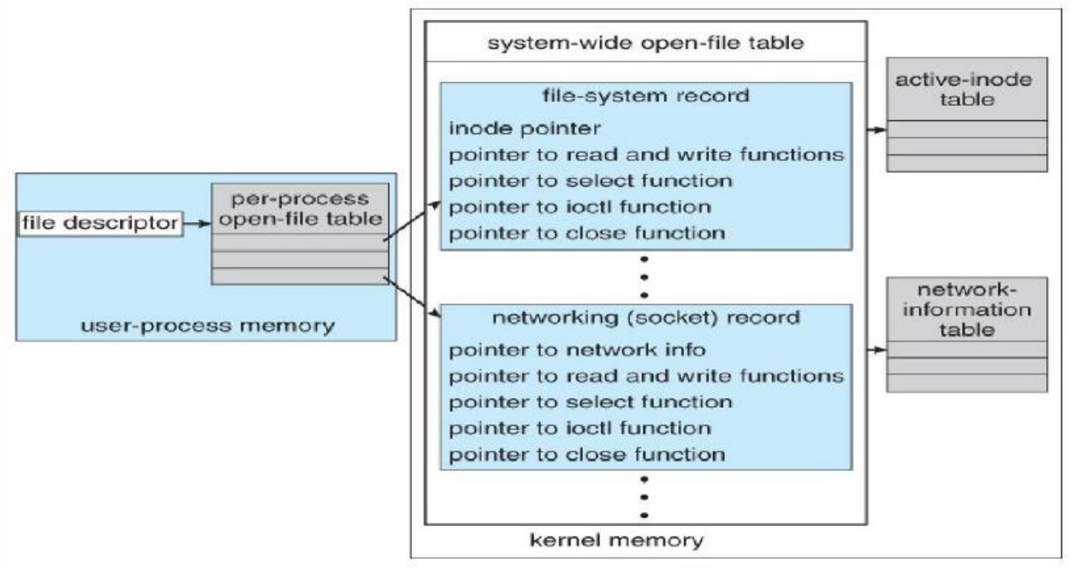
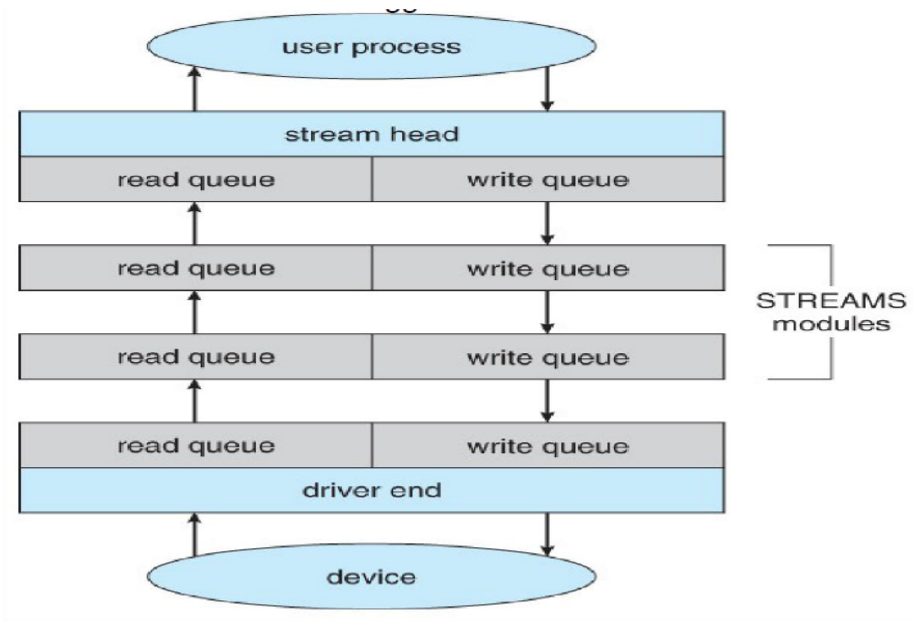

system-wide open-file table
file-system record
inode pointer
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
networking (socket) record
pointer to network info
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
kernel memory
active-inode table
network-information table
file descriptor
per-process open-file table
user-process memory

Fig : UNIX I/O kernel structure

<u>Streams</u>

- The ***streams*** mechanism in UNIX provides a bi-directional pipeline between a user process and a device driver, onto which additional modules can be added. The user process interacts with the ***stream head.***
- The device driver interacts with the ***device end.***

- Zero or more ***stream modules*** can be pushed onto the stream, using ioctl( ). These modules may filter and/or modify the data as it passes through the stream.
- Each module has a ***read queue*** and a ***write queue.***
- ***Flow control*** can be optionally supported, in which case each module will buffer data until the adjacent module is ready to receive it. Without flow control, data is passed along as soon as it is ready.
- User processes communicate with the stream head using either read( ) and write( ) (or putmsg( ) and getmsg( ) for message passing.)



The SREAMS structure.

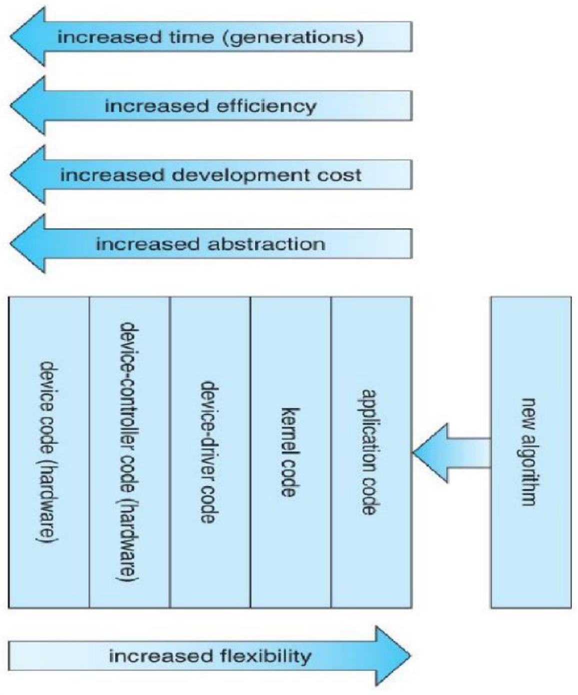
<u>Performance:</u>

- The I/O system is a major factor in overall system performance, and can place heavy loads on other major components of the system (interrupt handling, process switching, memory access, bus contention, and CPU load for device drivers just to name a few.)
- Other systems use ***front-end processors*** to off-load some of the work of I/O processing from the CPU. For example a ***terminal concentrator*** can multiplex with hundreds of terminals on a single port on a large computer.

Several principles can be employed to increase the overall efficiency of I/O processing:

1. Reduce the number of context switches.
2. Reduce the number of times data must be copied.
3. Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate
4. Increase concurrency using DMA.
5. Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.

6. Balance CPU, memory, bus, and I/O operations, so a bottleneck in one does not idle all the others.



Device functionality progression.