CHAPTER 10 THE I/O SUBSYSTEM

If we follow the diagram of kernel in chapter (2), then we will found that the "***File Subsystem***"(i.e. the I/O subsystem) allows a process to communicate with the hardware devices(i.e. peripheral devices) like disk, terminal, printers and networks etc. Some of these devices are "***Character Devices***" like printer and terminal, while some of them are "***Block Devices***" like hard disk.

The kernel programs, which control these devices, are together called as "***kernel modules***" or simply the "***Device Drivers***"(as the modules drive the device). Usually there is one device driver for one device type. Means one device driver can work for multiple hard disks (rapid array) on the same machine, but will not work for terminal of that machine.

Sometimes it may happen that, there are 3 hard disks of 2 different manufactures and hence may have 2 different device drivers. But this is occasional. Usually multiple devices of same type have one common device driver. Above occasional scenario may occur if both hard disks are controlled by different vendor specific architecture and commands.

As it is common to control multiple devices of one type by a common device, it is implied that the driver must distinguish multiple devices, separately (though of same type). In other words output for one device must not go to the output of another device of same type.

Above discussion was for actual physical devices. But a system may have so called "***software Device***", which does not correspond to any physical device. E.g.: RAM is not a peripheral device (it is actually inevitable part of any computer). Because peripheral device is that, which can be added later or which can never be added to the machine, but still machine can work when OS is loaded on to it. RAM is not like that. RAM is must for any type of computer to run. In some machines RAM can be replaced by so-called flash memory or EPROM can replace RAM, but then too they are RAM.

Coming back to our idea of "***software device***", the physical memory i.e. RAM can be treated as a "***device***" and a driver can be written for it to allow a process to access a physical memory location which is outside of its virtual address space.

Take *ps* command as an example. The ps command reads kernel data structures from physical memory to give you "***process statistics***"(*ps* stands for process statistics). Here *ps* will work, though kernel data structures are out of its virtual address space. In other words, *ps* work as driver, which reads that part of the physical memory, which is out of its address space. To do this it treats physical memory as a device and works itself as a driver for that physical memory device.

Similarly "***trace drivers***" can be written (ptrace for example) to write process trace records for debugging.

Not only this, but "***kernel profiler***", which we already discussed in last chapter of "***process scheduling***", is actually a "***software device***" driver. In which a process writes address of kernel routines, which found in kernel's symbol table and then reads profiling results accordingly.

Driver Interfaces

The Unix system contains 2 types of devices,

a) Block devices.
b) Character devices or also called as **RAW** devices.

Examples of block devices are *hard disk*, *floppy disk*, while examples of character devices are *printer*, *terminal*, *network card* etc.

The peculiar thing about block device is that, it can be treated as "***Random Access Storage Device***" and block devices may have character device interface too. Means a block device can be read or written block-by-block or character-by-character. In other words block device may have both "***Block Device Driver***" & as well as "***Character Device Driver***".

User looks at devices as files. Recall that, from user's eyes, "***devices are also files***". Obviously files have file system, hence devices also considered through the interface of file system.

That is why system calls that work with regular file(i.e. open(), read(), write(), close()) also work with device file too. In the file system, device is not a regular file, but a special file and it is not considered as "***leaf***", but considered as "***node***" in directory tree structure. Device file also has its unique inode.

The device file is distinguished from other files by the "***file type***" field in the inode. For device, this field is "***block special***" if the device is block device *or* this field is "***character special***" if the device is character device.

Here a question may arise, if a device has both "***block***" and "***character***" interfaces, then what will be the content of its "***file type***" field in its inode?

First it is very important to note that, when a device has both "***block***" & "***character***" interfaces, it will not have a single file representation. Instead the same device will have two files, one for its each interface and obviously two files will have two inodes and thus their "***file type***" fields in their inodes will have "***block special***" and "***character special***" identity respectively.

As mentioned earlier, from user perspective, regular file & devices file use same set of system calls that work for regular files. Though it is, there is a system call, known as ioctl()(means I/O control), which is made only for character devices(or for those block devices which have their character interfaces). This ioctl() will not work for regular files. Regular files have an analogous system call fcntl()(means file control) which works on "***file descriptor***" which is not possible for ioctl() though it has its one of the parameter "***file descriptor***".

This is because, the "***fd***" returned by open() system call for regular file and for device file have different interpretations. This is true for system V. Some implementations may consider & implement ioctl() transparent to all types of files including device files.

There is yet another important thing to mention worth. Though for user, open() is same for a regular file and for a device file, internally open() works differently for these two types. Same logic is applicable to all of those system calls, which work for both, regular file & device file.

However it is not at all mandatory for a device driver to support every system call interface.

System Configuration

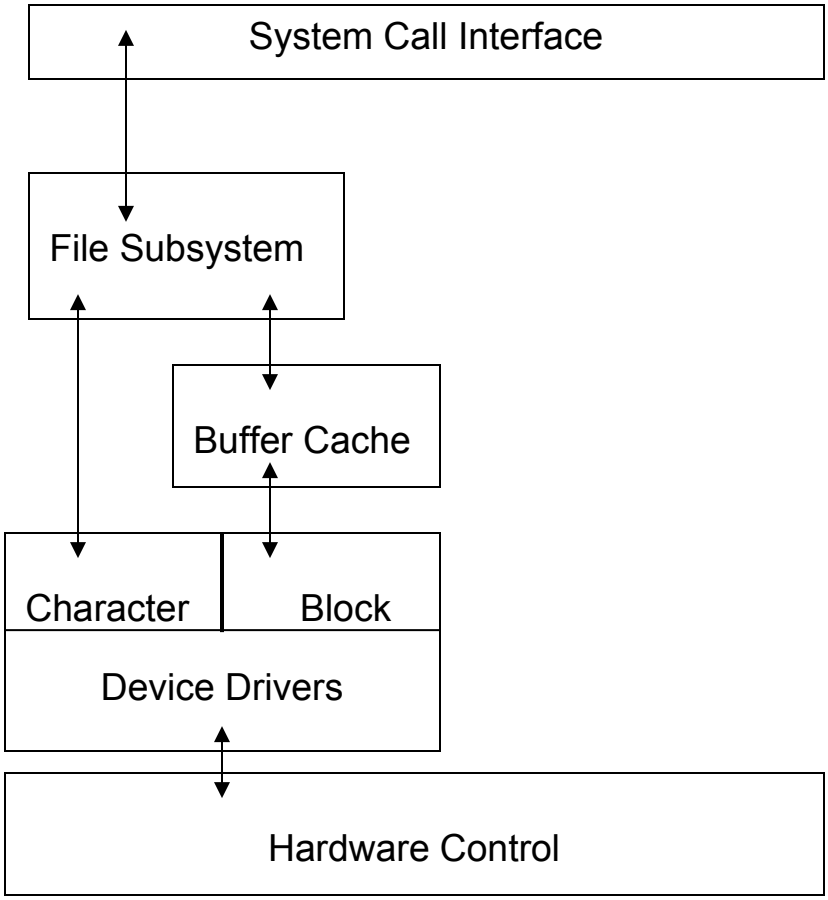
System configuration is the procedure by which administrators specify installation dependant settings. For a device concerned system configuration, the administrator may specify the number of installed devices on a machine to the kernel of loaded operating system. Here administrator usually specifies "***I/O port address***" of motherboard on which the device is plugged in.

There are 3 stages at which an administrator can specify device configuration....

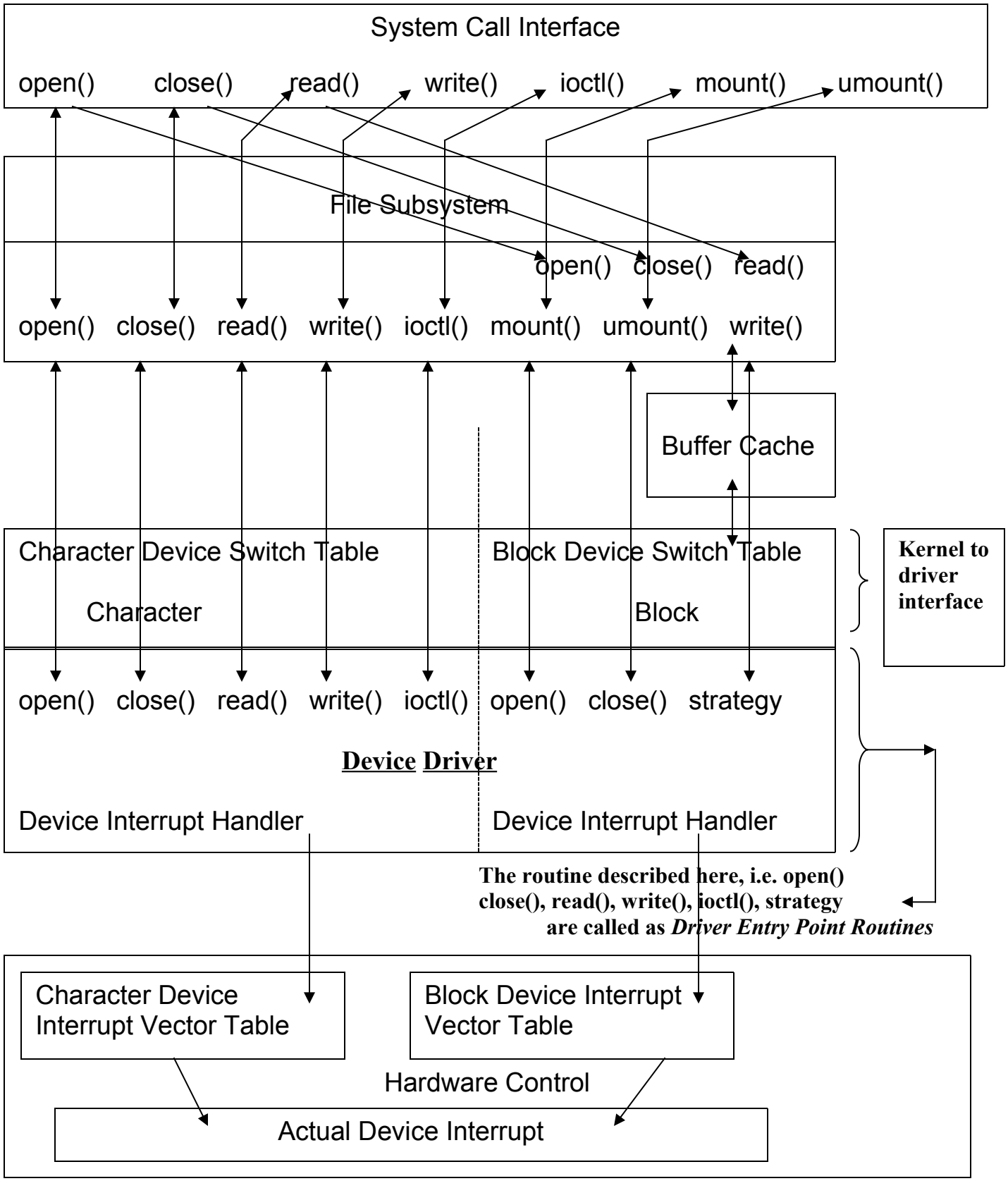
1) Administrator can hard code the device configuration concerned data into such "***conf***" files, which will be complied & linked statically when the kernel code is being built. Usually these files are plain text files, which the configuration program in kernel code, converts to "***compile-able and linkable code***" which is then patched to original kernel code. Obviously such devices, which are now permanent part of kernel, are supposed to be kept "***plugged in***" forever. In other words, if devices is removed or changed, the whole kernel has to be rebuilt, with modified code again.
2) System is already installed, i.e. kernel is already built and running and administrator is now specifying the device configuration related info. In such cases, the whole kernel is not rebuilt. Instead kernel updates "***internal configuration tables***" dynamically. Such types of devices are now a day called as Plug And Play devices or PNP devices.
3) Devices, themselves have necessary "***self-identification***" code burnt onto it already. Such devices permit the kernel to identify them dynamically when they are attached or removed. Here kernel reads "***hardware switches***" of devices and then re-configures itself to support such devices.

The common part to all above 3 stages is that, the process of device configuration creates some tables or if tables are already present, then fills those tables. These tables are part of kernel code.

Now recall the kernel diagram that we had already seen in chapter (2). The left side of the diagram is concerned with file subsystem & devices.



We will now elaborate this left side in much more detail....


System Call Interface
open() close() read() write() ioctl() mount() umount()
File Subsystem
open() close() read()
open() close() read() write() ioctl() mount() umount() write()
Buffer Cache
Character Device Switch Table
Block Device Switch Table
Character
Block
Kernel to driver interface
open() close() read() write() ioctl() open() close() strategy
Device Driver
Device Interrupt Handler
Device Interrupt Handler
The routine described here, i.e. open() close(), read(), write(), ioctl(), strategy are called as Driver Entry Point Routines
Character Device Interrupt Vector Table
Block Device Interrupt Vector Table
Hardware Control
Actual Device Interrupt

- If we look carefully at this figure, we can see that, when a user program calls a system call (open(), close(), read(), write(), ioctl(), mount(), umount() etc) from "***system call interface***" layer, it does not care much about the type of file, whether regular or device, because "***call***" are same.
- When flow arrives in "***file subsystem***", not much, but some filtration is done. For example...
    1) open(), close(), read(), write() for character & block devices arrive as they are. Same for mount() and umount().
    2) read() & write() for block device are filtered through buffer cache as buffer cache works only for reading & writing on block devices.
    3) Similarly ioctl() is filtered only to character device, as ioctl() works only for character devices in Sys V(UNIX System V).

- When flow arrives at "***device drivers***" layer, much of filtration is done. Before going to actual device driver part, the flow arrives at "***kernel To Driver Interface***". This has 2 sections called as "***Character Device Switch Table***" and "***Block Device Switch Table***". Each device has its unique entry in one of these tables according to its(i.e. device's) type. These entries in these tables actually direct the flow to an appropriate driver corresponding to the given system call. This means that...
    1) open() & close() calls when are in file subsystem, look for the "***file type***" field in the inode of specified file. If this field is "***character special***", then these 2 calls will be directed to "***character device switch table***". Similarly if "***file type***" field in the inode of specified file is "***block special***", then these 2 calls will be directed to "***Block Device Switch Table***". Same for read() and write() calls.
    2) mount() and umount() calls also get redirected to the "***Block Device Switch Table***", As mounting & unmounting is done for file systems and file system present on block device, when "***file type***" field has "***block special***" identity and calls are mount() or umount(), they are directed to "***Block Device Switch Table***".
    3) Ioctl() is directed only to "***character Device switch table***" by looking at "***file type***" field of specified file's inode.
- Now flow actually arrives at "***Device Driver***" part. This part is the real device driver's code. This part is also made up of different system call like low-level routines. Most of them even have same names as those of correspondingly given system call. Means in other words we can say that, when flow arrives in driver's part, after getting filtered through respective "***Switch Table***", it ultimately calls similar named routine in driver's part which corresponds with already given system call.

    For example, ...
    1) open(), close(), read(), write(), ioctl() system calls for character device ultimately call similar named open(), close(), read(), write(), ioctl() routines in character device driver part after getting filtered through "***character Device switch table***".
    2) open(), close() of those block devices, which are filtered through "***Block Device Switch Table***", ultimately, call open() & close() routines in Block device driver part.
    3) But for mount() & umount() things are slightly different. As they are for block devices, when they filtered through "***Block Device Switch Table***", they call open() & close() routines of block device driver part respectively. This is because mounting & umounting of block device is nothing but to open or close the block device respectively.
    4) Similarly when read() and write() system calls for block devices get filtered through "***Block Device Switch Table***", ultimately do not call corresponding read() & write() routines in Block device driver part. Instead they call "***strategy algorithms***" as they are first filtered through buffer cache. Through this is true for system V, some implementations may have read() & write() routines in block devices

driver part and these two routines may internally implement strategy algorithms. But system V does not have read() & write() driver routines for block device. Instead it has "***Device Strategy Procedures***".

After these two switch tables, flow arrives at the "***Hardware To Device Driver***" interface. This part consists of "***Machine Dependent Control Registers***" or "***Machine Dependent I/O Instructions***". This part actually manipulates devices and Interrupt Vector Tables. Thus when a device interrupt occurs, the system (kernel) identifies the device (which had caused interrupt) and then calls the appropriate interrupt handler.

"***Software devices***" such as "***kernel profile driver***", does not have any hardware interface (because it does not have any hardware associated with it). For such "***software devices***", there may be a "***software interrupt handler***" which is called. E.g.: The clock interrupt handler calls the kernel profile interrupt handler directly as there is no intermediate real profile hardware.

- mknod() : Administrators set up device special file(i.e. create device files for specific hardware) by using mknod() command.

  E.g.: mknod() /dev/tty13 c 2 13

  In above example, "***/dev/tty13***" is a filename of device, "***c***" is type of device, which stands for the "***character***" type. Here 2 is the major number and 13 is the minor number.
  The major number specifies type of the device, which corresponds to the appropriate entry in the respective "***Device Switch Table***"(i.e. Block or character).
  The minor number indicates a unit of the device.
- Device switch Tables: Consider the figures of "***Block Device Switch Table***" and "***Character Device Switch Table***" on next page.

Block Device Switch Table

Entry	open	close	strategy
0	gdopen()	gdclose()	gdstrategy()
1	gtopen()	gtclose()	gtstrategy()

Now if a process opens the block special file say ***"/dev/dsk1***" with its major number 0, then kernel goes to "***Block Device Switch Table***", uses major number as "***entry number in table***" and searches for corresponding entry to "***open***". Now in above figure "***open***" for 0th entry number corresponds to gdopen() routine. Thus kernel finally calls gdopen() to open the /dev/dsk1 block device.

As another example, if a process calls "***read***" to read a character device having character device having character special file "***/dev/mem***" and if its major number is 3 then…

Character Device Switch Table

Entry	open	close	read	write	ioctl
0	conopen()	conclose()	conread()	conwrite()	conioctl()
1	dzbclose()	dzbclose()	dzbread()	dzbwrite()	dzbioctl()
2	syopen()	nulldev()	syread()	sywrite()	syioctl()
3	nulldev()	nulldev()	mmread()	mmwrite()	nodev()
4	`gdopen()	gdclose()	gdread()	gdwrite()	gdioctl()
5	gtopen()	gtclose()	gtread()	gtwrite()	gtioctl()

Kernel goes to "***Character Device Switch Table***", uses major number as "***entry number in table***" and searches for corresponding entry to "***read***". So here "***read***" for 3rd entry number corresponds to mmread() routine. Thus kernel finally calls mmread() to read from /dev/mem character device

The routine name nulldev()(e.g.: 3rd entry corresponding to "***open***" column) is an "***empty***" routine. This is used when there is no need of a particular device function.

Many peripheral devices may have same major number, and then to distinguish them, minor number is used.

NOTE THAT, device special files need not to be created every time when the system is booted. They just need to be changed is particular device configuration changes. Hence they are mainly changed when device is plugged in or removed.

System calls And The Driver Interface

In this section we will discuss the things seen in previous section, in much more detail.
All those system calls, which use file descriptor as there one of the parameters, follow the same path of logic as for the regular file. Means as usual, kernel follows pointers from user file descriptor table, to file table, then to incore inode table. After getting the inode, it examines "***file type***" field and accordingly access the respective type devices switch table. From the same inode it gets major number & minor number of the device too. Now kernel has 2 things, one the switch table and second major/minor numbers. Then kernel uses major number as index into the switch table and finds out that device specific routine, which corresponds to the system call. Now it calls this found routine by passing minor number and other required parameters (taken from the parameter of system call) to it.

The most important difference between system call working for regular file and system call working for a device is that, the inode of regular file is always locked at the beginning of system call and is released while returning from it. But the inode of device file is never locked in the system call when system call is executing the driver code. This is because drivers frequently sleep when they wait for establishment of connection with the actual hardware or they wait for arrival of data from the already connected device. Now it is impossible to determine when hardware/device will respond for above two things and the process, which is calling this system call, may sleep for long time as its driver is slept for long time.

Considering this real scenario, if inode of device file is kept locked (as of regular file), other processes which want this inode(e.g. for stat() system call) would sleep for indefinite time waiting that inode to get unlocked.

Another important thing is about the parameters of driver routine (in switch table), as we know user process calls same system call for both regular file and device file. So when a system call works for both regular file and device file and when it internally calls driver specific routine (chosen from switch table by using major number), then same parameters are used by these routines too.

Thus first duty of driver routine is to interpret the system call's parameter, into its (routine's) understandable form.

Specific devices require specific type of data (e.g.- sound card requires audio data & capture cord requires both video and audio data). Hence device driver of such device keep its own private data structures to deal with such driver specific data. These internal data structures tell current state of the device or one of its units. Thus all related driver functions & interrupt handlers execute according to this current state of device and resulting action depends on it.

Now we will see all such possible device specific routines. Remember that, most of them have same names as there corresponding system calls; their internal implementations differ considerably.

(1) open() :

In the device specific open() routine, kernel follows the same procedure of opening regular file by open system call. Means it allocates/finds in-core inode, then increments reference count field in in-core inode, then assigns a file table entry and as usual assigns a user file descriptor entry too. Then it returns this user file descriptor table index to the caller. In short we can say that kernel opens a device so that it looks like opening a regular file.

How ever, inside device specific open() routine it actually does the device driver specific part. The algorithm is like follows...

Usual open() system call → device specific open call → driver specific open() call
E.g. gdopen()

```
01 : algorithm : open
02 : input :(a) pathname of device file
03 :         (b) mode of opening
04 : {
05 :    convert path to respective inode by namei(), increment inode's reference count, allocate file table
06 :    entry, allocate user file descriptor table entry… do everything just like opening a regular file;
07 :    from the found inode(in above step) get the major and minor numbers of the device ;
08 :      save the current context(by algorithm setjmp) to be useful when/if device does longjmp;

09 :   if(device is a block device)
10 :   {
11 :          use major number as index into the "Block Device Switch Table" to find out corresponding
12 :           device specific open routine;
13 :          then call this device specific open routine and pass minor number(found above) and open
14 :           modes(parameter(b)) as parameter to this driver specific open routine;
15 :   }
16 :   else /* Means device is a character device */
17 :   {
18 :         use major number as index into the "Character Device Switch Table" to find out
19 :          corresponding device specific open routine;
20 :   }
21 :   if(driver specific routine fails inside its algorithm)
22 :            undo UFDT, file table, incore-inode table changes;
23 : }
24 : output : User File Descriptor
```

It uses the major number (which it finds from the found in-core inode) as index into the respective "***switch Table***"(i.e. Block or character – according to type of device) and finds out corresponding driver specific routine and then calls it.

Sometimes the requested device may have both "***Block***" & "***character***" interfaces. In this case discussion is made on the user supplied first parameter to open() means by using pathname it decides whether user wants to open "***Block***" type or "***character***" type of this device.

Now we will see what happens inside the "***Driver specific routine***" decide to be invoked after taking it from respective type "***Switch Table***". First kernel tries to establish a connection between the "***calling process***" & "***the device***". Then it initializes private driver data structures.

For example, suppose a user is trying to log in. means it wants the "***terminal***" as a device. If the terminal is free, then no problem to make connection of user's login process to the terminal device. But if right now terminal is busy and thus machine yet not detected that a user wants to log in, then the process, which wants this terminal, goes to sleep. It wakes up when machine detects "***a***

hardware carrier signal". After waking up the process, now kernel initializes respective terminal's private driver data structures such as terminal's baud rate.

Sometimes, if the device is of "***software device***" type, kernel may not need to do these initializations. E.g.: RAM

Now consider Error / exception case: In the given algorithm there is a step of saving current context by setjmp(). Why this is? We know that, when system booting completes *getty* process opens terminal and login process waits for any user to login. Takes this as example, suppose no user logs in for long time and *getty* sleeps for this indefinite long time,(Means event of waking up getty never occurred). To avoid this, kernel must be able to wake up *getty* process. Now recall from sleep() & wakeup() algorithms, that, for such "***indefinitely long time slept***" process, system sends a signal and after recipient of this signal process wakes up and proceeds by throwing appropriate error or exception. While doing this the process must be able to "***undo***" the changes that it made before going to sleep, as after throwing error or exception, process is going to exit.

To allow to "***undoing***" the changes, they must be saved some where so that when time comes to make "***undo***", process will retrieve them and completes its undoing task. That is why kernel saves current context of the process by setjmp() before entering into the driver specific open routine. In this saved context there are UFDT, file table & in-core inode table changes that it had already made in step 1 of the algorithm.

When driver's open routine fails to open the device, driver calls longjmp(), kernel retrieves or restores the saved context and undo the changes made in UFDT, file table & inode table.

This is specified by last two statements (i.e. "***if block***") in the algorithm.(For more details see the algorithm of sleep() and its statements concerned with setjmp() & longjmp())

NOTE THAT, while "***undoing***" the changes modes in saved context, it also undo the changes that it had made in driver's open routine, such as initialization of private driver data structures. Thus it releases all initialized private driver data structure too.

All above things will occur whenever any errors or exceptions occur including the user's attempt to open such a device, which is not, yet configured and thus open fails.

What about "***no delay***" option? In pipe, we saw that a device (say pipe device) can be opened by "***no delay***" option. Can we use this facility to by pass the long sleep mentioned above? Yes! We can! But it is not a good idea. Because in this case when open() is called for a device with "***no delay***" option, open() returns immediately and user process(which called open()) can proceed as usual. But it will never know whether the requested device is really opened or not. Obviously next read() like calls will fail.

Is it possible to open a device multiple times? A device from the user process's perspective is like any other regular file. So as a regular file can be opened multiple times, a device also can be opened multiple times. The manipulation of inode, a file table & UFDT entry by the kernel for a device file is also similar to that of a regular file.

Obviously driver specific open() routine is also called multiple times in such a case. But internally, device driver keeps a count of how many times the device was opened and the call fails if the threshold count is in appropriate. Means, the terminal is such a type of character device, which can be used by multiple processes as a device of inter-process communication. So it makes perfect sense that multiple processes are opening a single terminal for such IPC and driver's open() routine is thus called multiple times and processes can write on that terminal to achieve their IPC.(E.g.: two-background servers are responding to their multiple clients on same console i.e. same terminal). But it makes no sense to allow multiple processes to write on a single printer, because in such case multiple processes will overwrite each other's data resulting in a garbled output, which cannot satisfy any process. Obviously count in the printer driver will be such that, allowing single process to write on a printer at a given time. Rather venders of printers keep a special spooler process, which accepts all process's request of writing and fulfill them one by one as appropriate.

Concluding, we can say that, when open() is called by multiple process for a single device, the driver specific open() routine is also called multiple times, but behaviors depends upon an internal

count of open() and according to the type of opened device, either such multiple calls to open will succeed or will fail, in case of in-appropriate count.

NOTE THAT c*lose() for device specific closes the device for whole system, not for a particular process like regular file*.

(2) close() :

When close() system call is used to close a device, first it internally calls device specific close() & then in turn device specific close() will call driver specific close(). close() system call for device disconnects the device connection from the kernel.

This is very important to note that, a device can be opened multiple times but disconnection will be made only once by device specific close(). In other words, the user programs will close() for a device, but if there are say 5 user programs using a same device and they all call close(), the system call close will be executed 5 times, but device specific close() will be called only once and the last close() system call will call device specific close().

Means device specific close() will disconnect the hardware from the system only when no other processes have the device opened. Obviously device's close() will wait until it is sure that no other processes are accessing the device. Thus kernel may call device's open() multiple times but calls device's close() only once.

Above necessity creates some problems. The driver is not able to know, how many processes opened the device. That is why the device driver designer must code the driver very carefully.

If suppose, device driver sleeps in close(), means "***shutting connection down***" is not yet completed and suppose some other process opens the same device meanwhile, then after waking up, close() may get completed and the process which opened the device will face problems for read/write on it.

The algorithm for device's close() is as follows .....

```
01 : algorithm : close()
02 : input : file descriptor
03 : {
04 :    do usual "close" algorithm as that of regular file;
05 :    if(file table reference count is not 0)
06 :            goto finish;
07 :    if(there is another open file and if its major and minor numbers are same as that of the device
08 :       which is to be closed)
09 :    {
10 :            /* this is not last close() call */
11 :            goto finish;
12 :    }
13 :    if(device is a character device type)
14 :    {
15 :            use major number as index into the "Character Device Switch Table" to find out
16 :            corresponding driver specific close routine;
17 :            then call this driver specific close routine and pass minor number as parameter to it;
18 :    }
19 :    if(device is a block device type)
20 :    {
21 :            if(device is mounted)
22 :                    goto finish;
23 :            write device block in buffer cache to device;
```

```
24 :            use major number as index into the "Block Device Switch Table" to find out corresponding
25 :            driver specific close routine;
26 :            then call this driver specific close routine and pass minor number as parameter to it;
27 :            invalidate device block still in buffer cache;
28 :    }
29 :    finish :
30 :            release the inode of the device file;
31 : }
32 : output : nothing
```

Algorithm's philosophy is same as that of the regular file, but before releasing the inode of the device file, kernel does some tasks common to both "***block***" & "***character***" type devices, while some tasks are unique to respective types.

1. Kernel searches the file table entries to make sure that no other processes still have the device in opened state. This is not sufficient to rely on this file table count to call last close(), because several processes may access the same device by different file table entries. We may think that, to enforce above thing we can do inode table search to see reference count. But this is also not sufficient. Because one device can be accessed via two different files (recall that some devices may have both block & character interfaces or same device may have different linked file name-symbolic links) having two different, independent inodes. E.g.: Suppose there are two files say /dev/tty01 and /dev/fty01, if we issue a command like ls –l, then output may be as follows .....

   crw--w--w- 1 root vis 9,1 Aug 6 1994 /dev/tty01
   crw--w--w- 1 root unix 9,1 may 3 15:02 /dev/fty01

   Here the beginning "***c***" indicates that the two files "***vis***" & "***unix***" are of "***character special***" device files. There device names /dev/tty01 & /dev/fty01 are also different. But as both have major number 9 and minor number 1, they are concerned with a single device. Here we may rely on another point that these two files are actually links and thus will have same inode. But this is also not true, because link count for these two files in above command shows 1. Every file has link count at least 1(for its name) in its inode and if and only if it has some other links then count becomes greater than one. This shows that the link count 1 clearly indicating the 2 independent inodes for these 2 files of same device. That is why we cannot rely on inode reference count for calling the last close(). Thus further step is necessary ...
2. Get major number & minor number of closable device, and look for all inodes, if any resemble with same major number & minor number. If it is, then don't call last close().
3. **For character Device:** Kernel just calls device close() routine and returns to user mode as shown in the algorithm from line no 13 – 19.

   For block device: But for block device this process is not so simple. As block device can be used for mounting of file system, some extra precautions & measures are necessary.
   - Kernel searches mount table, to see, whether this closable device does not contain any mounted file system. If it is, then last close() is not called, because as device already contains a mounted file system indicating that some other process is using the device and thus this close() is not the "***last***" device's close().
   - Above measure is not also the "***enough***" measure. Because sometimes it may happen that, a process "***A***" unmounted the device and thus process B when calls close() for the device, finds that there is no mounted file system on it. But as it is a block device, it uses buffer cache for read & write and buffer cache may still contain some buffers for previously mounted file system (i.e. which was just unmounted by process A). Such buffers are usually "***delayed write***" buffers. Thus before calling "***last***" close() for device(given by process - B), kernel searches whole buffer cache and if finds such

"***delayed write***" buffers for this device it writes them back to the device, before calling "***last***" device's close() procedure.

- Now device's "***last***" close() is called. But this is not the end. Kernel immediately goes to buffer cache again and invalidates all those buffers(i.e. delayed write buffers) concerned with this closed device. Obviously as per the strategy for "***delayed write***" buffers, invalidating them after so long time allows kernel to keep useful data as long as possible.

4. Finally kernel releases the inode of the device file and thus now the device connection is cut and the driver is ready to accept next device open() call by re-initializing the driver's internal private data structures.

(3) read() & write()

Reading from a device and writing onto a device is logically quite similar with the reading & writing of regular file respectively.

FOR CHARACTER DEVICE user given read() & write() system calls ultimately call device specific read() & write() routines. As seen in the kernel diagram, there is no "***buffer cache***" step between system calls & character driver. But usually device drivers themselves buffer data internally in their implementations. Although there are some important situations where device driver may bypass their internal buffering mechanism and thus can transmit & receive data directly to & from the user address space and the device. E.g.: Terminal device drivers use a data structure called as *clists* to buffer the data internally. Here driver allocates memory for a buffer(here, for a *clist*) and copies data from user address space into this buffer during the process of writing on device. Then the data in this internal buffer is actually written on the device and we get output on terminal(i.e. on console).

Sometimes(rather most of the times) process writes data to its internal buffer very fast. Obviously data from this process's internal buffer(e.g. char Buffer [1024]) gets copied to driver's internal buffer very fast. This is because both process & driver are in RAM and data transmission also takes place in RAM. Means driver's internal buffer gets data from the process very fast. But transmission of data from driver's internal buffer(i.e. *clist*) onto the actual device is never so fast because device may be slower in performance than the RAM.

Thus if process is generating & transmitting the data to the driver faster than the device's capability of outputting it, then the device's write() routine puts such faster performing process to sleep until the device becomes capable of accepting more data. Above story concludes waiting on a device.

This is called as Flow Control or Throttling of data transmission.

During read() from a device, the driver receives data from the device into its internal buffer and then copies contents of this internal buffer to the process's address space i.e. to the process's internal buffer.(E.g.: char Buffer [1024]).

Above discussion gives us much information of transmission of data between user process and the device driver. But how actually driver communicates with its real device is a different story. This depends completely upon the hardware of device. The commonest mechanisms are memory mapped I/O, programmed I/O, Raw I/O etc.

! The memory mapped I/O means certain addresses mentioned in kernel's address space are not locations in physical memory(i.e. RAM) but are actually special registers of a hardware device, which are responsible for controlling that particular device.

The device driver for such type of above device pass values(called as control parameters) to these registers and thus control the activity of device accordingly. Obviously these values are according to the hardware specifications given by hardware designers.

For Example, VAX-11 computer's I/O controllers contain some special registers like...

a) Registers for recording status of the device – called as control and status registers.

b) Registers for data Transmission – called as data Buffer registers. Here the data buffer registers, which receive data, are Receive Data buffer registers and which transmit data are Transmit Data Buffer registers.
In machine line VAX-DZ11; all above registers are configured(or say mapped) to some physical locations in RAM.

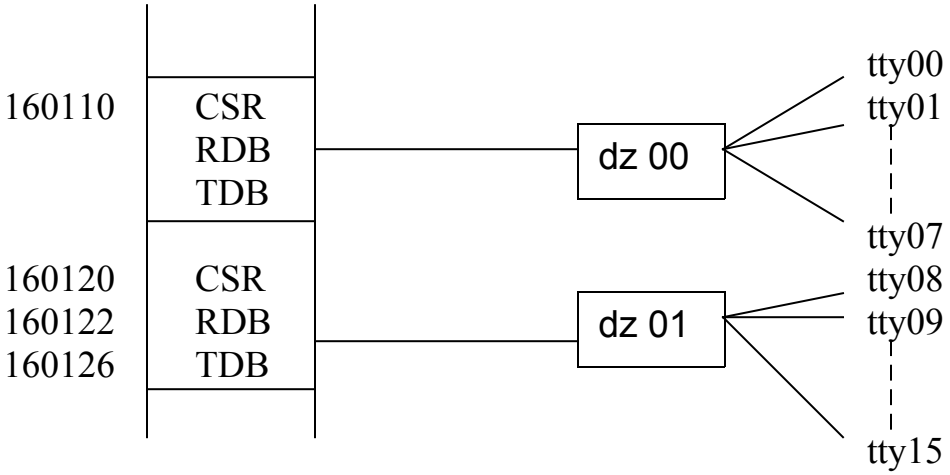


Figure: memory mapped I/O with VAX-DZ11 controller

Consider above figure and assume that VAX-DZ11 terminal controller controls 8 asynchronous lines for terminal I/O communication. Also assume that there are 2 chips of VAX-DZ11 controller on a machine, named dz00 and dz01 respectively.

Now suppose for a particular chip, say dz01, its control & status register (CSR) is mapped to address 160120, its receive Data Buffer Register (RDB) is mapped to address 160122 and its transmit data buffer register (TDB) is mapped to address 160126 in RAM.

Figure also tells us that first chip i.e. dz00 controls 8 terminals from tty00 to tty07 and second chip i.e. dz01 controls 8 terminals from tttty08 to tty15.

Now, when driver wants to write one character to a terminal of dz01 say tty09, the terminal driver first writes 1 to a specified bit position in CSR and then writes the desired actual character in TDB. As soon as data is written in TDB, it is the stimulation to TDB to transmit the data. CSR contains many specific bits. One of its bits is the Transmit Interrupt Enable Bit and other one is the Done Bit. When above data transmission of TDB completes, the done Bit in CSR is set, indicating that controller is ready to accept more data from the terminal. Now controller can set Transmit Interrupt Enable Bit, which cause the controller to interrupt the system when controller is ready to accept more data.

Above discussion is about waiting the data to DZ11 controller device. Reading data from this 11 controller is also similar process.

! **The Programmed I/O** means the hardware itself contains instructions to control the device(ROM burned) Drivers of such devices just execute those hard wired instructions and control the operation of the device.

For example, the IBM 370 computer has a hard wired start I/O instruction. Driver just executes this instruction and initiates the I/O operation of such device. Here the method of driver-device communication is "***open***" to the user. Means user process/programmer can directly tell the driver to do so. E.g.: programmers to manipulate various drivers can use Int896 functions in DOS.

As we saw up till now, the driver-device communication is hardware architecture dependent and hence there are no standard interface (i.e. set of programmable API's) exits between them.

We also saw that driver may control the device by keeping its internal buffering mechanism (as there is buffer cache for character devices).

Sometimes the internal buffering mechanism of drive can be an issue of device's performance. In such cases considering rapid data availability of a device, driver can bypass its internal buffering

and can setup Direct Memory Access (DMA) between the device and RAM. This can be done for both memory mapped I/O and programmed I/O. For DMA, driver's task is just to setup correct locations in RAM so that the device itself can access them. Some operating systems may allow bulk DMA transfer of data between device & RAM in parallel to other CPU operations.(This is some sort of parallel processing). When such a transfer of data gets completed, the device itself interrupts the system for accepting any more available data.

! **THE RAW I/O** another way to avoid buffering, is to avoid intermediate step of kernel's buffer cache mechanism. High-speed devices (though block devices) can sometimes transfer data directly between the device and the user address space (i.e. program's internal buffer say, char buffer [255]) without intermediate step of kernel's buffer cache. As there is one less copy operation in this, the transfer speed is high and most importantly the amount of data transfer does not need to depend upon the kernel's buffer size (recall that buffer cache mechanism uses fixed sized buffers corresponding to fixed size blocks. Say 1024 in Sys V). Thus the amount of data can be bigger than kernel's buffer size. This mechanism is called RAW I/O. This method initially sounds compatible for block devices (which require high speed) only. But recall that some character devices may behave dual as they have their "***block***" counterpart too. In such types of character devices, drivers invoke strategy interfaces first though they are called from character device specific read() and write() routines. Thus we can say that, those character devices, which have their "***block device***" counterpart due to their dual nature, can mainly use raw I/O mechanism.

This completes the discussion of reading & writing of device driver for character type devices. Now we will discuss about reading & writing of block device drivers.

! **FOR A BLOCK DEVICE** kernel has intermediate step of buffer cache mechanism. The device driver uses strategy interface to transmit data between buffer cache and the device.(Recall that "***block device***" counter part of dual natured character device, also can use strategy interface to transmit data between device & user address space directly and faster).

The strategy interface APIs usually create queue of I/O jobs for a device, which is currently under work. Drivers, as per need, set up physical address of buffer from which data is going to be transmitted to the device. Driver may do this by considering one physical address in buffer at a time or may choose multiple addresses in buffer.

On behalf of a running process, when data is to be transferred to a device, kernel passes address of buffer header (of that buffer which contains the transferable data) to the strategy interface procedures. As we know, header contains list of physical addresses (i.e. pages) of data locations and also contains sizes of transferable data. This is the same method used by "***swapping***" system of memory management used for swapping of memory pages. The difference is that, when buffer caching is to be used, the transfer of data is by one page at a time and when swapping is to be done, then transfer of data can be of many pages at a time.

NOTE THAT, when data is to be transferred from the device to the user address space (i.e. to the program's internal buffer say char buffer [1024]), to the process for which this is going must be locked by the driver. So driver writers must keep this thing in mind as a crucial measure. If suppose whole, process cannot be locked, then atleast concerned pages must be locked, until whole I/O is complete and data is transferred.

For Example, after mounting a file system, the kernel identifies every file in the mounted file system by its device number (obviously mounted device's device number) and inode number. This device number is "***encoded combination***" of mounted device's major number and minor number. When kernel wants to access a block from a file(from mounted file system), it copies this device number and block number into the buffer header(recall from the chapter of "***Buffer Cache***" that, buffer header contains device number & block number). So when buffer cache algorithms like bread() or bwrite() access the mounted disk, they internally invoke the strategy interface procedures uses the device minor number and block number fields in the buffer header to identify the location of data on the mounted device. After finding data, it copies the data to the buffer using that buffer's address in

memory. Same logic can be applied when a process directly accesses a block device(i.e. opening, reading & writing a block device). Here too, buffer cache algorithms & strategy interface procedures work similarly as described in above example.

(4) ioctl

The ioctl() system call in this Unix system is generalization of stty() [i.e. set tele type terminal settings] and getty() [i.e. get teletype terminal settings] system calls in old UNIX systems.

This system call gives common "***entry point***" for device specific commands. Thus in general it allows 2 things...

i) Allows a process to set device's hardware specific options and
ii) Allows a process to set device's device driver specific software options.

This specific actions vary/differ according to the type of device and thus these specific actions are defined in respective device's respective device driver. Hence programs those use ioctl() system call must know about the type of device(i.e. device file in /dev directory) with which they are dealing. This situation is an exception to the rule that UNIX system does not differentiate between different file types.

The syntax of ioctl() system call is

ioctl(fd, command, arg);

Where "***fd***" is the file descriptor returned by prior open system call. Then "***command***" is the request of the driver to due desired action and "***arg***" is a parameter (usually pointer to a structure) for the 2nd parameter. Here the "***command***" i.e. 2nd parameter is driver specific, and thus depends upon internal specification of the device. Obviously format of structure variable i.e. 3rd parameter depends upon the 2nd parameter's device specificity.

Device drivers can read this "***arg***" data structure from the user address space according to its predefined format or they can write device's settings into user address space at "***arg***".
For example, the ioctl() system call allows

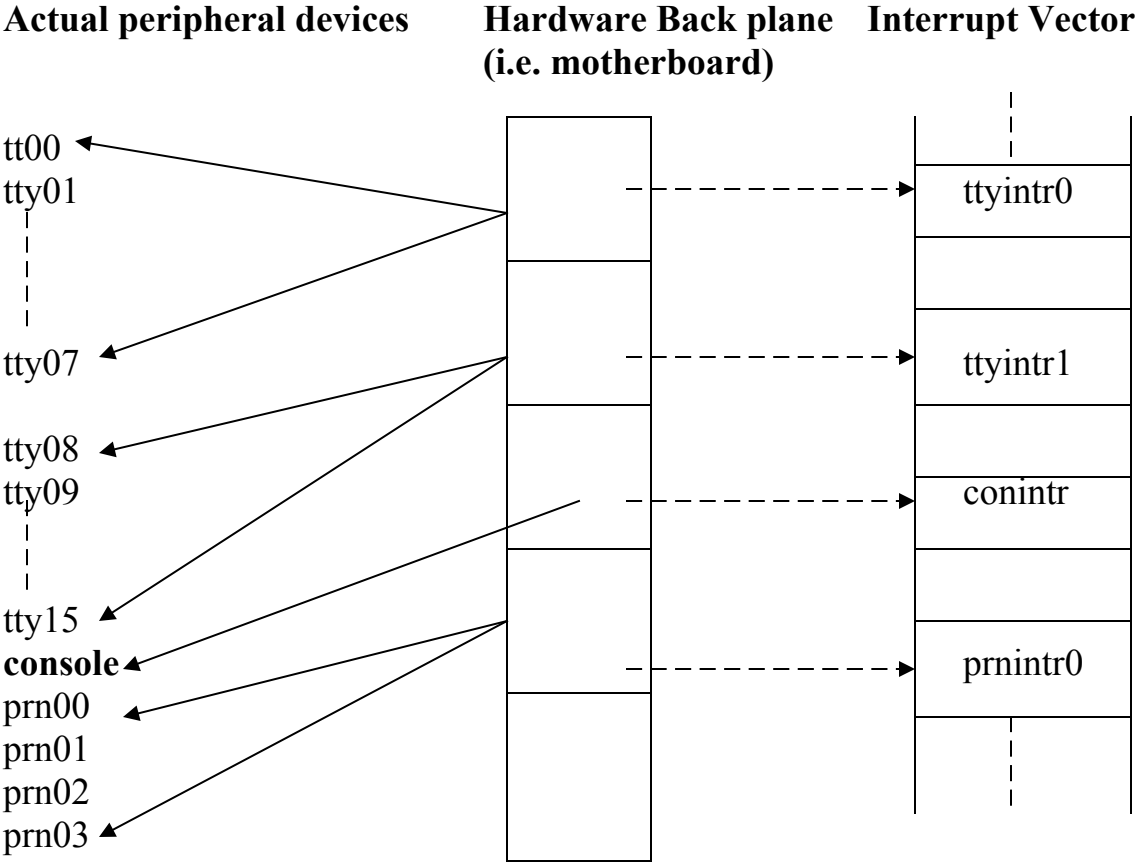
a) User programs to set terminal baud rates.
b) User programs to rewind tapes on a tape drive.
c) User programs to specify virtual circuit numbers and network addresses on a network operation.

(5) Other File System Related System Calls

File system calls such as stat(), chmod(), lseek() work for devices in the same way as they do for other type of files. Means they can manipulate device file's inode without accessing that device's device driver. For example, if lseek() system call is used for a tape on tape drive, the kernel as usual updates the respective file table offset without doing any device driver specific operation. When the same process later tries to "***read from***" or "***write to***" the tape from this changed position, the kernel moves the file table offset to process's u area (as like in regular file) and then the tape device physically seeks to the correct offset as mentioned in u area.

Interrupt Handlers

As explained before, when a device interrupt occurs, kernel execute that device's interrupt handler function. The name at this function is found by the kernel by using the information about the interrupting device and its relation with the offset in the interrupt vector table (IVT). When this interrupt handler is called kernel passes the interrupting device's device number or other device specific information as parameter to this interrupt handler.



In above figure there are 16 terminals in a networked system. Terminals are named by numbers (numbers names terminals). Terminals *tty0* to *tty07* have interrupt handler *ttyintero* in IVT, where terminals *tty08* t o *tty15* have interrupt handler *ttyintr1*. There position on the motherboard is important, because IVT is generated (by BIOS) in the same sequence as their positions. As many devices can be associated with one IVT entry (e.g. *tty00* to *tty07* terminals are associated with *ttyintr0*), it is the duty of device driver to resolve which device had caused the interrupt.

Similarly, the two terminal interrupt handlers, *ttyintr0* and *ttyintr1* are given the number postfix 0 &1, because kernel also distinguishes them by such index numbers and used these numbers as parameters when calling the respective interrupt handler. In short, when *tty06* causes the interrupt, *ttyintr0* will be called and when *tty09* causes the interrupt, *ttyintr1* will be called by passing 0 & 1 to respective *ttyintr* in IVT.

So in general, we can say that, the device number used by the interrupt handler is used for identification of device's hardware unit while the device minor number is used for identification of device(as a whole) to the kernel. The device driver actually establishes the relation between minor device number and major device number (i.e. hardware unit).

Disk driver

On UNIX system disks are partitioned into different sections and different sections in turn contain different file systems each. This is made purposely to facilitate the use of different file systems on disk. E.g.: An administrator now onwards can mount one of the file system as "***read only***", while can mount other file system as "***read-write***" and can keep some other file system un-mounted.

Here it is important to note that, though all file system can fit in single disk and though all can contain many files, files on un-mounted file system can not be accessed by file system related system calls. Also files cannot be added or modified or deleted when a file system is mounted as "***read only***".

This partitioning strategy has one other advantage each section (and thus each file system too) spans through contiguous tracks & cylinders of the disk, row it becomes easier to copy entire file system (i.e. file system as whole). This would not be possible if a single file system is kept scattered any where on entire disk.

The duty of the Disk Driver is to translate a file system address (which is made up of logical device number and logical block number) to a particular physical sector on the disk. Driver gets this file system address by one of the 2 ways...

a) Either the respective "***strategy interface procedure***" uses a buffer from the buffer pool and this buffer's header contains the file system address in the form of logical device number and logical block number
b) Or read() & write() procedures takes "***device minor number***" as their parameter. These procedures then use this number to identify the physical drive and particular section to be used. From this and from the internal tables they find the sector, which is the beginning of required disk section. We can call this as "***start sector number***", now we know that process's u-area (i.e. the process which is using read() or write() system calls) contains the byte offset from which next read or write operation should begin. From this by the offset bmap() is used to get the "***block address number***". Now if these 2 numbers (i.e. "***start sector number***" & "***block address number***") are added, we get the exact physical sector on disk from which I/o should start.

   Disk manufacturers maintain sizes & lengths of disk sections fixed according to the disk type.

Sections	Minor Device Number	Start Block (Size of block is 512 bytes)	Length In Blocks	
0	0	0	64000	→ /dev/dsk0
1	1	64000	944000	→ /dev/dsk1
2	2	168000	840000	→ /dev/dsk2
3	3	336000	672000	→ /dev/dsk3
4	⋮	504000	504000	
5	⋮	672000	336000	
6	⋮	840000	168000	
7	⋮	0	1008000	

Above figure is the partitioning of DEC RP07 disk. Files /dev/dsk0, /dev/dsk1, /dev/dsk2 & /dev/dsk3 correspond to sections (portions) 0, 1, 2 & 3 respectively. Assume that these device files have minor numbers 0, 1, 2 & 3 and also assume that the size of logical file system block & the disk block is same and it is 540 bytes. Now suppose kernel tries to access block 940of the file system in /dev/dsk3, then the disk driver translates the block no 940 as follows...

Starting block number of /dev/dsk3 = 336000
Requested block number in /dev/dsk3 = + 940

336940

So disk driver actually accesses block number 336940 on physical disk.

From above figure we also can see that sizes of sections (or partitions) may vary and administrator may configure different file systems on those sections accordingly means large file system will obviously require larger section and so on.

Also notice that different section on disk may overlap. E.g.: In the figure on last page, sections 0 &1 are separate but if added, their total size becomes 64000(size of section 0) + 944000(size of section 1) = 1008000 which is the whole size of disk. These 2 sections are not overlapping (i.e. separate), but now see section 7 whose size is 1008000, which also occupies whole disk. This clearly shows that section 7 is overlapping on combination of section 0 & 1 or vice versa.

This overlapping is not dangerous, provided that file system in section 0, 1 & 7 do not overlap on each other. This is something similar to extended DOS partitioned. The advantage of having such a section, which occupies whole disk, is that, the entire disk can be copied at once.

Use of fixed sections restricts the flexibility of disk configuration. So sizes of different sections on a disk should not be put in disk driver in hard coded from. Instead these sizes of configuration should be kept somewhere else and this configuration should be kept somewhere else and this configuration should be made configurable/customizable. UNIX system v keeps its disk configuration in 1st sector of the disk (logically called as Boot block as we saw in chapter 2). This configuration is in the form of table mentioning size file system type and limits. This table is called as *Partition Table* and a program like fdisk makes it customizable.

Though disk driver is not supposed to contain the hard coded information about section sizes, it may contain hard coded location of this partition table.

As Disk I/o is always heavy, most disk manufactures take care of disk job scheduling, positioning of disk arm and data transfer from disk to CPU. Manufacturers make this possible via their disk controller chips. But if these things are not taken care off by disk manufacturer, it is the duty of disk driver to do itself.

From chapter 4 & 5, we know that using different file system related system calls we can access disk data. But some programmers may bypass this way and can directly access disk data via raw or block interfaces. Unix system v itself comes bundled with such 2 programs *mkfs* & *fsck*,(they do not use file system related system calls) where *mkfs* formats a disk partition and creates super block, inode list, list of free blocks and root directory for a new file system. While *fsck* checks the consistency of an existing file system and corrects errors if any.

Suppose, in /dev directory, there are two files /dev/dsk15 and /dev/rdsk15. Both these files are pointing to same device, one via block interface(i.e. /dev/dsk15) and other via raw interface(i.e. /dev/rdsk15). Now we issue ls command which shows output like follows…

 # ls –l /dev/dsk15 /dev/rdsk15

br-------- 2 root root 0, 21 feb 2 15:40 /dev/dsk15
crw-rw---- 2 root root 7, 21 mar 7 09:29 /dev/rdsk15

This output shows that /dev/dsk15 is a block device (by starting latter '***b***' of permission string), owned by root and only root can read it (by second latter '***r***' of the permission string). This device has major number 0 & minor number 21. The second line of output shows that /dev/rdsk15 is a character device (by starting letter '***c***' of permission string), owned by root and root can read or write on it as well as group also can read or write on it. (But note that "***group***" is also "***root***" here). This device has major number 7 & minor number 21.

Now consider following program……

```
# include <fcntl.h>

void main(void)
{
        /* Variable declarations */
        char szBuffer1[4096], szBuffer2[4096];
        int fd1, fd2, I;
        /* code */
        if(((fd1 = open("/dev/dsk15", O_RDONLY)) == -1) ||
            ((fd2 = open("/dev/rdsk15, O_RDONLY)) == -1))
        {
                printf("Failed to open\n");
                exit(1);
        }
        lseek(fd1, 8192L, 0);
        lseek(fd2, 8192L, 0);
        if((read(fd1, szBuffer1, sizeof(szBuffer1)) == -1) ||
           (read(fd2, szBuffer2, sizeof(szBuffer2)) == -1))
                {
                        printf("Failed to read\n");
                        exit(1);
                }
         for(I=0;I<sizeof(szBuffer1);I++)
                {
                        if(szBuffer[I] != szbuffer2[I])
                        {
                                printf("Different at offset %d \n", I);
                                exit();
                        }
                }
         printf("read matches\n");
}
```

According to this program, when open() system call is called with /dev/dsk15 file, access to this block device will be through "***block device switch table***", and when open() is called with /dev/rdsk15 file, then access to this character device will be through "***character device switch table***". From the output of previous "***ls***" command, it is clear that both these files have minor number 21, which tells the driver which section (partition) of the disk is accessed. In much circumstances "***21***" is made up of 2 & 1, which correspond to "***physical drive 2, partitions 1",*** obviously both files are referring to same section of same disk. (Note: here it is assumed that now system has one disk device. Because there is no other way, except "***partition table***" & driver's source code, to tell that both drivers (i.e. block device driver & character device driver are referring to same device). Obviously above program is opening the same driver (low level) twice through different interfiles code next lseeks to byte offset 8192 in the opened devices and then read data from that offset. Now if we assume that there is no other interfering file activity concerned with these two files, the reading must be identical and output must be "***Read Matches***".

Programs like *mkfs* & *fsck*, which read or write the disk "***directly***", are dangerous, because they can read or write "***sensitive data***" collapsing the system security. Thus system administrators must protect such "***sensitive data***" by giving appropriate permissions to disk device files. E.g. files of above example /dev/dsk15 & /dev/rdsk15 must be protected by giving "***read***" permission only to "***root***" and neither "***read***" nor "***write***" to anyone else.

Another important drawback of such "***direct access***" is that, it can destroy "***file system consistency***" maintained by algorithms of chapter 3, 4 & 5. So free block list, free inode list, pointer to direct & indirect disk blocks may get corrupted. Superblock also gets corrupted. As "***direct access***" mechanisms bypass the file system algorithms, even if they are coded carefully, chances of file system structure corruption are there. Chances are much more if they are running when other file system activities are still going on. This is the reason that, *fsck* is never used on an active file system.

The major difference between the two drive interfaces (i.e. block device interface & character device interface) is whether they use "***buffer cache***" or not. When accessing a block device interface, kernel follows the same algorithms as that of regular files, except when bmap() coverts logical byte offset to logical block offset, the logical block offset itself is then treated as physical block number in the file system. Then it accesses the data via buffer cache & finally accesses driver strategy interface.

But when kernel accesses the disk via "***raw device interface***", it does not make conversion mentioned above. Instead it accesses process's u-area, takes byte offset saved there and immediately passes it to the driver routine as its parameter. The actual conversion is done further by either "***driver read()***" or "***driver write()***" routines. These two routines (which-ever is used) thus convert the passed "***byte offset***" to "***block offset***", accesses data and directly pass this data to user address space (i.e. to local buffer used by process) bypassing all buffer cache mechanisms.

Hence if one process writes via block interface and second process reads from same address via raw interface, the second and process may not read the data written by process one, because data may not read the data written by process one, because data may be still in "***buffer cache***" and not yet written to the required disk address. But if second process also reads the data using block interface then it will be able to read data not from the disk but from the buffer cache, as it exists there.

Using "***raw interfaces***" also can show some strange behavior. For example, a process if reads or writes a raw device with small unit than the block size, results are totally driver dependant.

Then a question may arise, why to use raw interface? The answer is speed. Obviously we cannot take advantage of caching data (executing same application immediately, which had previously taken longer time to execute. This becomes possible because at 2nd time app's data is read from the cache and not from the disk, as done at first time). Processes accessing block devices go through buffer cache mechanism and hence have constraint of "**block size**". Means if a file system has logical block size of 1k, then processes reading or writing block devices can use at the most 1k data during each I/O. on the contrary, processes which are accessing disk via raw interface do not have such constraint and thus can read or write multiple data blocks during each I/O. For them capability of hardware disk controller chip is the only constraint.

Conceptually processes reading or writing data, whether by "***block interface***" or by "***raw interface***" find same results but using "***raw interface***" makes it much faster to get result.

If we take same previous program code and assume that a process reads 4096 bytes (4kb) using block interface for a file system whose logical block size is 1kb, then kernel has to loop for 4 times. Because during each read it can at the most read 1kb and the data actually is 4kb. But when using raw interface, this whole operation can be done at once. Additionally block interface has additional copy of data in cache while raw interface has not, as it copies data directly to user address space.

Terminal Driver

As like other drivers, Terminal driver also has the same function of "***controlling the transmission of data to and from terminals***". But terminals are special drivers because they are the user's interfaces to the system.

Internally Terminal drivers have a group of modules called as "***line discipline***". These modules actually translate terminal inputs & outputs.

In canonical mode (i.e. the mode which deals with "***what really user meant***"), the line discipline modules convert raw data sequences typed at keyboard to a canonical form (in other words,

"***formatted form***") before sending the data to receiving process. Also line discipline modules convert raw output sequences, written by a process, to a formal that actually user expects.

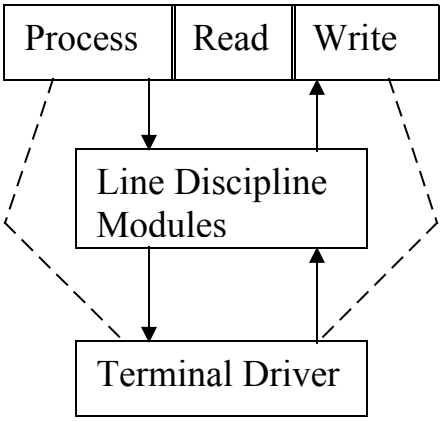
In raw mode line discipline modules just pass data between processes & terminals, but without any conversions. So data is not formatted as per user's expectations.

For example, due to fast typing by an user, some unwanted characters get entered and user, after realizing this, can use "***erase***" key to remove unwanted part and then makes required corrections. The terminal, in any case sends the entire sequence to the machine along with those erased characters (we are assuming "***dump terminals***" here). Now if the terminal driver is working in canonical mode, the line discipline modules buffers all the data into lines (means either "***enter key***" is pressed or a newline character appears) & then internally deals with erased characters before sending "***revised***"(or say "***corrected***") sequence to the reading process. But if the terminal driver right now working in raw mode, then no such buffering is done, means there is "***no waiting***" for enter key or new line. And thus "***receiving process***" can read characters, as they are type by "***sending process***". Means in raw mode terminal does not wait and hence works in asynchronous form.

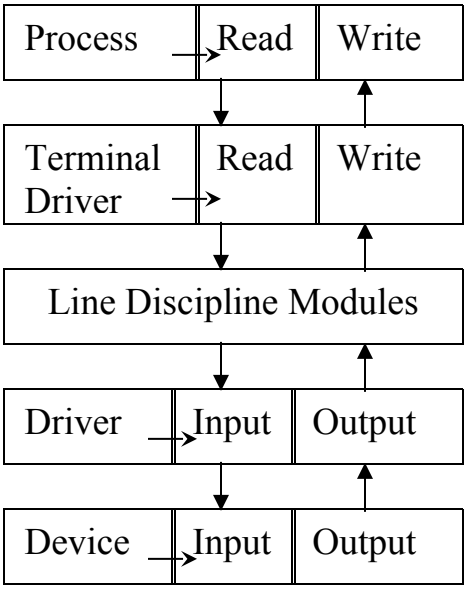
Dennis Ritchie noted that, "***the line disciplines used originally during system development were only in shells and editors, but not in the kernel itself***".

But realizing its need for many programs, later they are added in kernel itself. Logically speaking line discipline modules appear between terminal drive and rest of the kernel. In other words, kernel does not directly invoke the line discipline modules, but the terminal driver invokes them internally.

Data Flow



Control Flow



Previous figure shows the logical flow of data through the terminal driver and the line discipline modules. Another figure shows the flow of control through the terminal driver.

User can specify, "***what line discipline are wanted***", by using ioctl() system call. But for a device it is still difficult to implement use of multiple line disciplines for single device simultaneously, where each specified line discipline module successively could call next line discipline module to process the data in turn.

Clists

The line discipline module use a data structure called as clists to manipulate the data. A clist(or say **c**haracter **l**ist) is a "***variable length***"(not fixed) linked list having 2 fields...

1) A cblock (character block) and

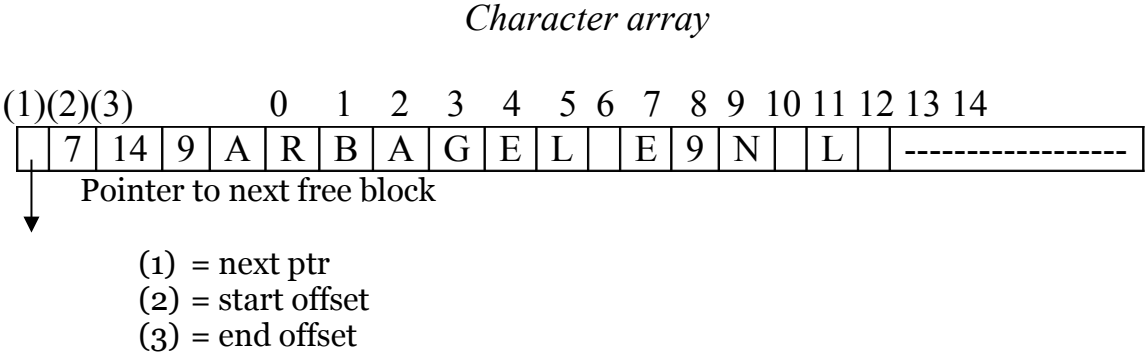
2) A count of number of character on the whole list.

A cblock, in turn, contains 4 fields…

1) Pointer to the next cblock.
2) A character array to hold actual character data.
3) Start offset, indicating the first location of valid data.
4) End offset, indicating the first location of non-valid data.

Kernel internally maintains a linked list of "***free cblocks***" and does 6 operations on clists and cblocks.

1. Assign a free buffer from the kernel list of "***free cblocks***" to the terminal driver.
2. When a cblock becomes free, return it back to the linked list of "***free cblock***".
3. Retrieve the first character from a clist – it does this by removing first character from the first cblock on clist and adjusts the clist's character count field accordingly. It also adjusts the indices of retrieved characters in cblock, in such a way, that next retrieval should not retrieve the same character again. If this retrieval operation retrieves the last character from a cblock, the kernel places the empty cblock (obviously after removal of last character, block will become empty) back to the linked list of "***free cblocks***", and adjusts the clist pointers accordingly. While doing character retrieval operation, if a clist contains "***no characters***", then kernel returns the null character.
4. By finding the last cblock on clist, kernel can put a character into it. After doing this it adjusts the offset values accordingly. If cblock becomes full, kernel can allocate a new, free cblock (from linked list of "***free cblocks***"), links it at the end of present clist and then places character into this new cblock.
5. The kernel can remove group of characters (recall that in operation 3, we removed a single character) from beginning at clist, from one cblock at a time. This operation is equivalent to removing all characters from one cblock but one character at a time.
6. The kernel can place a cblock (containing characters) at the end of clist.



Above figure of cblock shows all its 4 fields. Such cblocks are linked to form clist.

Functions of Line Discipline modules:

1) Parse input character strings into different lines.
2) Process "***erase characters***".
3) Process a "***kill character***" which invalidates all input characters typed uptill now on the current line.
4) Echo (or write) received character on terminal console.
5) Expand the output string's blank spaces into tables.

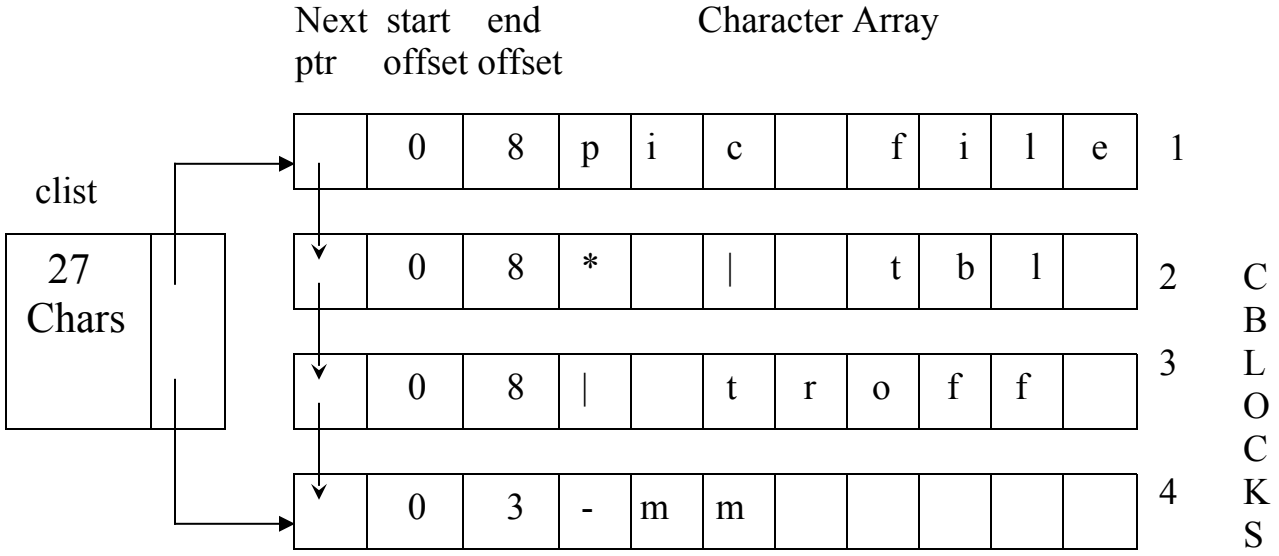
6) Generate signals and send them to respective processes about terminal hangup, line breaks, del key hitting.
7) Allow "***raw mode***", such that there is no processing of erase, kill or enter keys.

Manipulation Of Clist

clist data structure has a simple buffering mechanism which is more useful for small amount of data transmission from terminal like slow devices. Manipulation of data takes place either one character at a time or in groups of cblocks.

How character data is removed from a clist?

For example consider a situation where data is removed from a clist by "***one character at a time***" method.



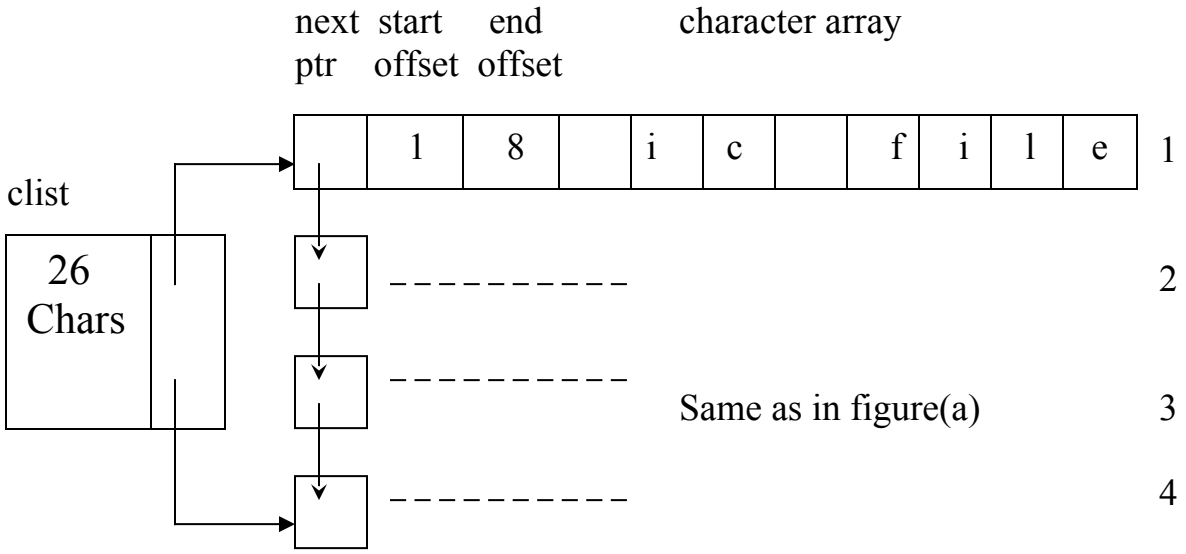
(a)

All upcoming figures from (a) to (h) assume the limit of character array field in a cblock is 8. The real picture might differ.

Figure (a) on last page shows a clist having 4 cblocks with a 27-character command....
pic file* | tbl | troff –mm (27 characters)

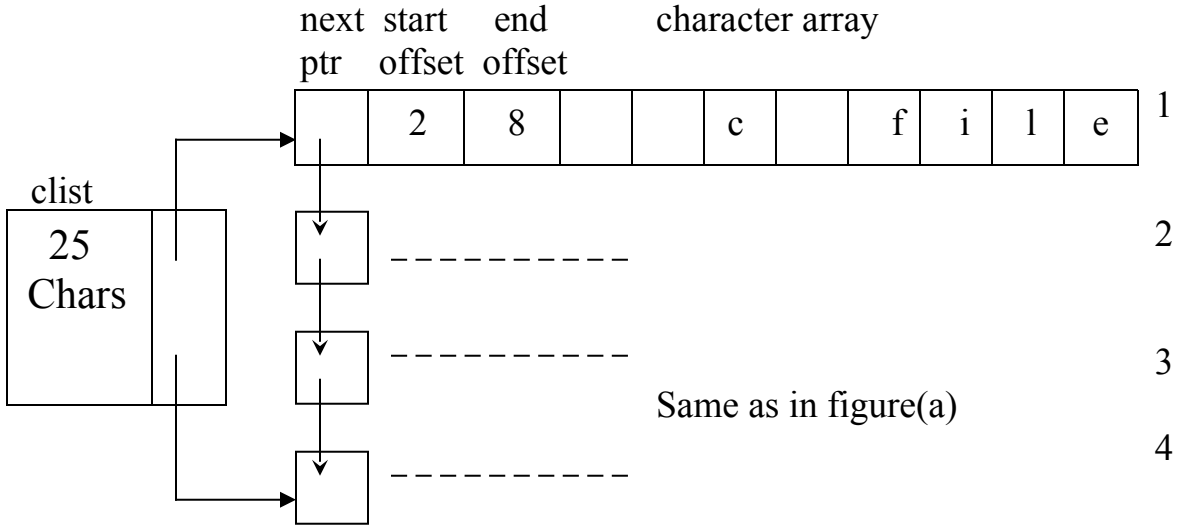
The block on left side of figure shows number of characters on clist, which are 27. Next field in clist is pointer to the first cblock (out of 4). The right side of figure shows 4 cblocks. As stated before the first field of cblock is pointer to the next cblock. Thus all 4 cblocks are linked together to their corresponding first field i.e. *Next ptr*. Along with "***next ptr***" field, each cblock has "***start offset***" field, which is index 0 in all 4 cblocks, indicating occurrence of 1st valid character of their each character array. The next field is "***end offset***", which should contain last index of character array. Thus 1st, 2nd, &3rd cblocks, as each contains 8 characters, the last index is 7, but as "***end offset***" means occurrence of 1st invalid character, the entry in this filed is 8, because of 8th index, there is no character, i.e. invalid. The last filed in each cblock is a character array of 8 characters each. Thus 1st cblock's character array field contains '*pic file'*(8 characters), the 2nd cblcok's character array field contains '* | tbl '(8 characters), the 3rd cblock's character array field contains '| troff ', and the last, 4th

cblock's character array field contains '–mm', so 4th cblock's "***end offset***" field has entry 3, because at 0th index(valid offset) there is '-' character, followed by two 'm' characters at 1st & 2nd indices. The third index is first invalid character's occurrence offset and hence "***end offset***" field is 3. Remaining 5 places of character array of 4th cblock are empty because command ends at last 'm' of "-mm".



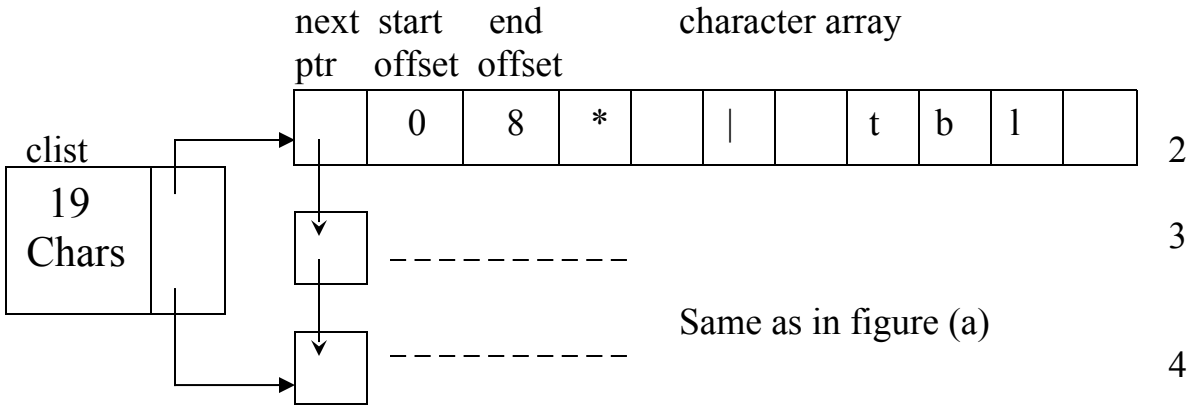
(b)

This figure b) shows the situation when one character is removed from the clist. The removal of characters begins from 1st cblock's first valid character, which is '*p*'. As '*p*' is removed, its partition becomes empty and hence its block's entry is shown empty. Now next valid character is at index 1, which is '*i*', and hence the "***start offset***" field becomes '*1*'(from 0). As one character is removed from the clist, now total number of characters in this clist becomes *26*(from *27*).



(c)

Above figure(c) shows the situation, when next character to '*p*', i.e. '*i*' is removed from the clist. Notice that, after removal of '*i*', "***start offset***" field changes to 2(from 1), first two element of character array become empty & total number of characters in clist now becomes *25*.



(d)

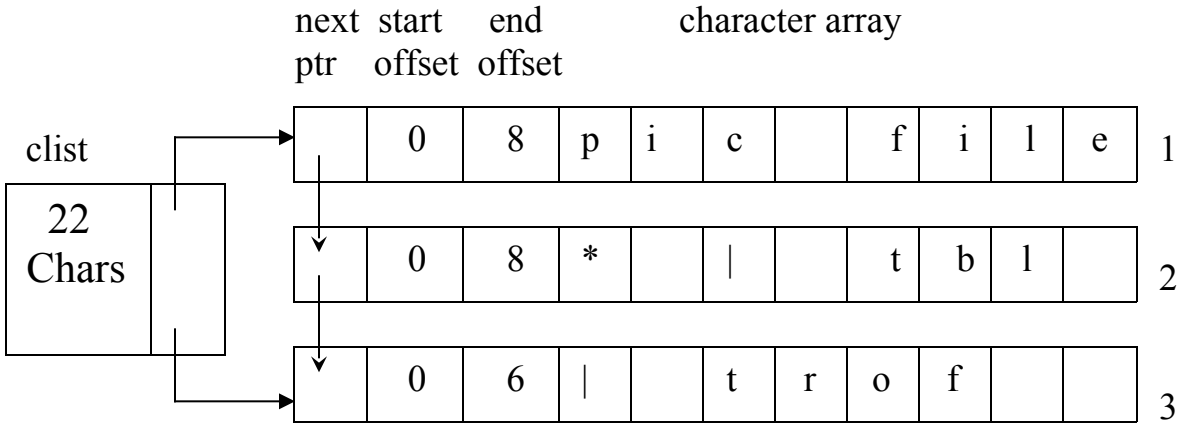
Now consider the situation in figure (d), where all 8 characters in 1st cblock are removed. As 1st cblock becomes empty, it is returned to "***free cblocks list***", and thus it disappears from the current clist. So the "***pointer to cblock***" field in clist now starts pointing to 2nd cblock. So if next characters are to be removed, them process of removed will starts from the 2nd cblock henceforth. Also notice that, removal of 8 characters from first cblocks, marks total number of characters field in clist to 19(i.e. 27 - 8 = 19). Now 2nd cblock in this clist becomes the 1st cblock and total number of cblocks now becomes 3(as 1st is returned).

How character data is added to a clist?

Suppose user uptill now typed a command like follows...

pic file * | tbl | trof (22 chars)

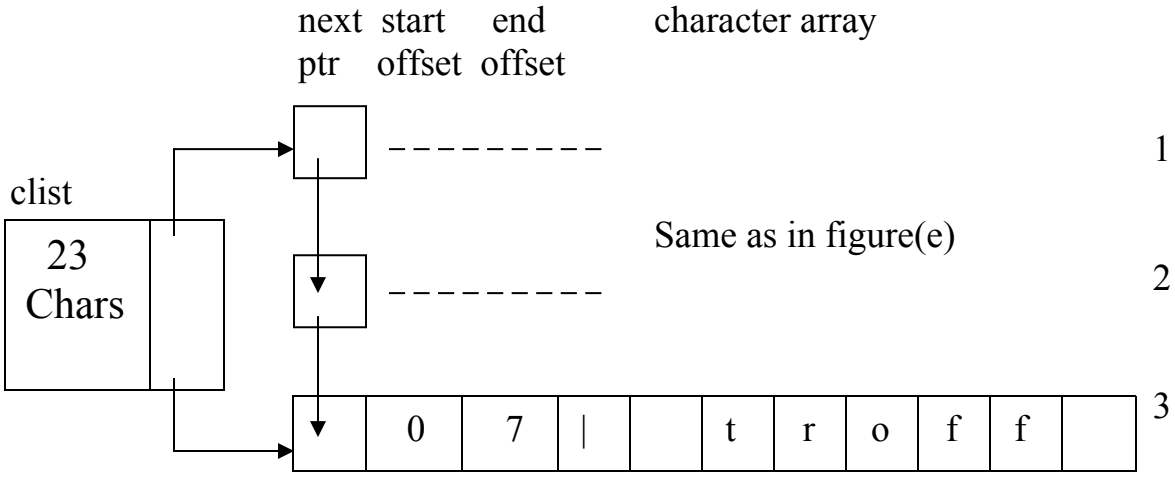
more characters are expected. Thus a clist now has 22 characters and yet more to be entered by user. So the situation in likes....



(e)

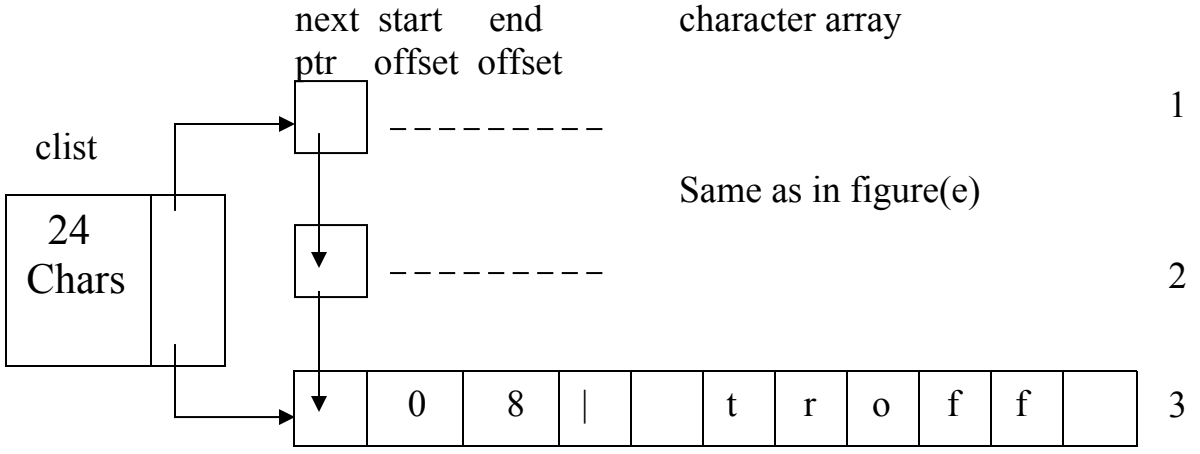
As shown fin figure (e), the 22 character command had already filled two cblocks completely and third cblock partially. The third cblock right now has 6 characters indexed from 0 to 5. Obviously the 6th index is right now empty and thus the first invalid character. Therefore the "***end offset***" field in third cblock shows entry of 6.

Now if user types another "*f*", it will occupy 6th index of 3rd cblock. So "***end offset***" field shows position of first invalid character, which is obviously 7, and total number of characters in current clist becomes 23.



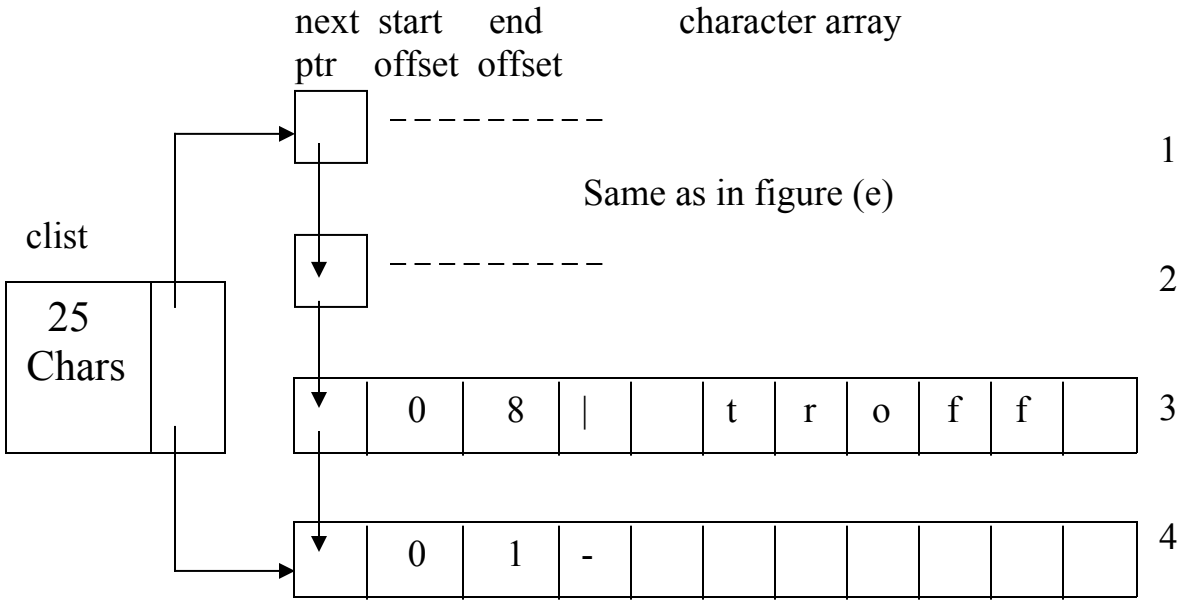
(f)

Suppose next to '*f* user types space bar key(i.e. blank) obviously it will occupy 7th index of 3rd block and the "***end offset***" field of 3rd cblock shows position of first invalid character, which is now obviously 8. Total number of characters in current clist is now 24 and importantly notices that, now third cblock is full.



(g)

Now suppose use next wants to type a hyphen character (i.e. '_'). As third cblock is now full, kernel demands a "***free cblock***" from "***free cblock list***" and adds this new cblock as forth cblock to the clist. Situation now looks like…



(h)

When a new cblock (here 4th) is added, kernel establishes its link with the third cblock and adds the typed hyphen at the 0th index of 4th cblock's character array field. As this is the first valid character, its "***start offset***" field shows 0 and as there is no character at index 1, means "***invalid***", the "***end offset***" field shows 1. Total number of character on this current clist is now 25.

Concluding we can say that, while removing character from a clist, "***start offset***" field at cblock gets affected and while adding character to a clist "***end offset***" field at cblock get affected. Also, during "***removal***", if 1st cblock becomes empty, it is removed from the clist and returned to the "***free cblock list***", and clist now pints to the next cblock as first cblock in the clist. During "***adding***", if a cblock becomes full, a new "***free cblock***" is borrowed from "***free cblock list***", then it is added to the clist at end of clist and linked with previous cblock.

Terminal Driver in Canonical Mode

The terminal driver contains 3 clists as its data structures. One clist to store "***output data***"(i.e. data to be shown on terminal), second clist to store "***raw input data***" typed by the user as it is(this data is given to the driver by "***terminal interrupt handler***") and third clist to store "***cooked data***"(i.e. "***disciplined***" by line discipline modules). Obviously this "***cooked data***" is the converted/formatted "***raw data***" by line discipline modules by taking care of tabs, erase/kill keys etc.

The algorithm of writing data to the terminal is as follows...

```
01 : algorithm : terminal_write()
02 : input : none
03 : {
04 :    while(more data is to be copied from user address space)
05 :    {
06 :            if(tty is already flooded with output data)
07 :            {
08 :                    start writing operation to the tty hardware by using data on "output clist";
09 :                    sleep(event : up till tty can accept more data);
10 :                    continue; /* back to while loop */
11 :            }
12 :            copy data of amount "size of cblock", from user address space to "output clist" and allow
                 this data to get converted properly by "line discipline modules" to take care of tables,
                 erase/kill keys etc;
13 :    }
14 :    start writing operation to tty hardware by using data on "output clist";
15 : }
16 : output : none
```

The canonical mode means "***consideration of users will to see the data in formatted form as he/she wants***". So "***line discipline modules***" have major role with this type of mode in terminal's driver and thus are invoked first.

Algorithm: A process can have its output data in one of the two ways,

(a) Either process itself s writing data to terminal by using printf() like standard output function *or*

(b) It is accepting "***output able*** data" from user's typing. In both above cases, process stores this data in its local buffer. This local buffer is part of user's address space.

When the process invokes driver, it first looks at this local buffer (i.e. user's address space) for data, which is to be output. Thus driver establishes a loop to extract this data from user's address space. Then driver copies "***cblock size***" data from user's address space to its "***output clist***", then line discipline modules convert tabs, erase/ kill keys (if any) to "***user desired***" format. When this "***output clist***" gets full or say exhausted, then the line discipline modules internally call procedures to output this data (in "***output clist***") to the terminal hardware.

Sometimes it may happen that, there is yet more data still in user's address space, but terminal is already flooded with output data. In such situation, driver, like above, outputs the data from "***output clist***" to the terminal hardware and puts this writing process to sleep, waiting for an event, which will occur when terminal becomes capable of accepting more data. So when terminal becomes capable of accepting more data, all processes (including this one) waiting for this event will wake up. This awakening is done by the terminal interrupt handler, which realizes that now the terminal is capable of accepting more data.

This loop goes on uptill user address space has more data to output. Note that "***flooding of terminal***" is predetermined by "***High***" & "***Low***" water level marks. So when the data in "***output clist***" crosses high water level mark, terminal is said to be "***flooded***" and thus cannot accept more data. When data on "***output clist***" drops below low water level mark, terminal is said to be "***not flooded***" and thus capable of accepting more data. The terminal interrupt handler answers this determination of terminal's condition.

If multiple processes try to write to a terminal, they all follow above algorithm independently & asynchronously. Obviously output on terminal may look garbled. There are two reasons for this garbage like output...

- A process may write to a terminal using multiple write() system calls. Now between two write() calls, kernel may do context switch to some other process and this other process may use its own write() calls to write to the same terminal. Thus output of previous process in followed by output of next process, which will be followed, again by output of previous process when it will be scheduled. Obviously terminal's output looks garbled.
- Secondly a writing process may sleep within write() system call, because terminal is flooded and thus this writing process is waiting for "***data drainage***" from the system. Again same above story may occur if some other process gets scheduled for writing its own data to the terminal.
  That is why kernel cannot give guarantee of output on terminal by write system call for its contiguity.

```
/* Global variable declarations */
char form[] = "this is a sample output string from child";
void main(void)
{
        /* variable declarations */
        char output[128];
        int I;
        /* code */
        for(I=0; I<18; I++)
        {
                switch(fork())
                {
                        case –1 : exit(); /* error when touch max proc count */
                        default : /* parent process */
                        break;
                        case 0 : /* child process */
                        /* format the output string in variable form */
                        sprintf(output, "%s%d\n%s%d\n", form, I, form, I);
                        for(; ;)
                                write(1, output, sizeof(output));
                }
        }
}
```

In above program code, parent process, inside its loop creates 18 children processes. Each child process formats a string by using sprintf() library function into an array named "***output***".

This string which will be now called as "***output***", contains message at 41 characters (i.e. the "***form***" string) then an integers (2 bytes), then a new line character (1 byte), then again the same message string at 41 characters, then again an integer (2 bytes) and finally again a newline character (1 byte). So the total of 88 characters (or say bytes) are there in "***output***" string.

Then this child process enters in an infinite loop & inside the loop writes this "***output***" string on terminal (i.e. standard output – stdout) during each iteration. As we assume here that the standard output is terminal (means default) the terminal driver works for this data transfer to the terminal.

Now in Unix System V implementation, the cblock is 64 bytes long and the output string is of 88 bytes, means too long to fit in the cblock. Obviously terminal driver needs cause output on the terminal become gabled. E.g.: when above program ran on an AT&T 3B20 computer, the output was like follows...

This is a sample output string from child1
This is a sample out**this is a sample output string from childo**

This demonstrates the fact of writing on a terminal by multiple processes asynchronously cause garbage like output. The 2 reasons are explained before the above program.

Reading data from a terminal in "***canonical mode***" is more complex. The read() system call, in its third parameter gives expected number of bytes to read in each operation. But as canonical mode involves "***line discipline***", which formats the input and thus if user process "***enter key***"(carriage return) the user's request of read gets satisfied even if the character count is not satisfied.(I.e. even if the number in third parameter to read() system call is yet not satisfied).

This adjustment is very logical because a process cannot anticipate how many characters the user will enter at the keyboard. And it is also not possible for the system(or as terminal driver) to wait for the user to type a large number of characters. E.g.: suppose we wants to type a command on shell and expecting that shell should respond him after pressing enter key, then for terminal driver & line discipline code, it makes no difference whether the given command is small like "***date***" or "***who***" or is very big & complex like "***pic file *|tbl|eqn|troff –mm –Taps | apsend***".

Here terminal driver & line discipline do not care because they don't know shell's syntax. And this is good to know nothing, because shells, editors in system have different syntax & yet more editors will have their own different syntax. So to be transparent to all, terminal driver and line discipline do not care, and thus satisfy read call on receiving enter key press event though expected character count (i.e. third parameter of read() system call) is not satisfied.

Algorithm for reading a terminal is like follows…

```
01 : algorithm : terminal_read()
02 : {
03 :    if(no data on canonical clist)
04 :    {
05 :            while(no data on raw clist)
06 :            {
07 :                    if(tty is opened with "no delay" option)
08 :                            return;
09 :                    if(tty in raw mode is based on timer, and timer inactive)
10 :                            arrangement for timer wakeup(callout table);
11 :                    sleep(event : data arrives from terminal);
12 :            } /* end of while loop */
13 :            /* there is data on raw list */
14 :            if(tty is in raw mode)
15 :                    copy all data from raw clist to canonical list;
16 :            else /* means if tty is in canonical mode */
17 :            {
18 :                    while(characters are there on raw clist)
19 :                    {
20 :                            copy one character at a time from raw clist to canonical
21 :                            clist; do erase or kill processing;
22 :                            if(typed character is "enter key" or EOF)
23 :                                    break;
24 :                    }
25 :            }
26 :    }
27 :    while(characters on canonical list are there and read count is not satisfied)
28 :    {
29 :            copy from cblocks on canonical list to user address space;
```

```
30 :   }
31 : }
32 : output : nothing
```

For understanding this algorithm we assume that terminal driver is in canonical mode(for raw mode see next section).

If there is "***no data***" in canonical clist, we enter in "***if block***". Now if data is neither on raw clist, we enter in a "***while loop***", (means there is no data on both input clists), and then process sleeps until a line of data arrives on terminal.

But if there is no data on both input clists (as above) and terminal (tty) is opened with "***no delay***" option, then like we had seen in "***pipe***", there is no point in waiting for data to be kept on terminal by the user. So due to this "***no delay***" option simple return from the algorithm.

There is yet more condition if the tty is opened in raw mode and if it based on "***specific time-up***" wake situation, but the timer is inactive, then first kernel goes to "***call-out table***"(see chapter (8) for clock & callouts) and makes arrangements for restarting the clock (i.e. timer wakeup) and then process sleeps for waiting for data arrival on terminal.

When data arrives on terminal (i.e. user enter data) the "***terminal interrupt handler***" executes the "***line discipline interrupt handler***", which intern places this entered data onto the raw clist as input for reading process. It also places the data on output clist for echoing back to the terminal (to allow the user to see what he had typed).

As this is the task of 2 interrupt handlers, it is not in algorithm.

Coming back to algorithm, now we are out of the first while loop (either because we never entered in it ***or*** because terminal interrupt handler awaken us because now there is data on tty). So now we have data in raw clist(which was put onto it by interrupt handlers as seen in previous paragraph). If the tty is in raw mode, copy all data from raw clist to canonical clist at once. But if tty is already in canonical mode, then this copying will take place, but not at once but character by character i.e. one character at a time inside 2nd while loop. Then as canonical mode is there, line discipline will do erase key & kill key like processing before putting the data on canonical clist. That is why character-by-character copying is necessary for such type of "***formatted***" processing.

When the character being copied is found to be "***enter key (i.e. carriage return)***" or "***EOF character***", then character putting on canonical clist is stopped and now canonical clist is ready of further processing. Thus we come out of this 2nd while loop.

When the character being copied is found to be "***enter key (i.e. carriage return)***" or "***EOF character***", then yet one more thing happens. The terminal line discipline interrupt handler awakens all processes, which were sleeping for reading from a terminal (i.e. due to "***no data***" on terminal). This may include other processes or may be this process. [*This part is also not included in algorithm*]

Algorithmically now we are out of first if block and now there is data in canonical clist. So we enter in third while loop (either after coming out of above if block, or after awaking). Now the data in canonical clist is copied to the user's address space (i.e. to the buffer whose addresses is given as 2nd parameter of read() system call) until either copied character is found to be "***enter key***" or until the character count(i.e. 3rd parameter of read() system call) is satisfied. Out of these two choices, the smaller one is chosen.

Sometimes awakened reading process (may be other or may be this) may find that due to other process's reading, the data in raw & canonical clist is finished (because it is like pipe), and thus this newly scheduled awakened process may face "***no data***" situation, similar to reading from a pipe by multiple processes simultaneously.

There is one surprising thing in this algorithm. The character processing in "***input***" and in "***output***" mechanisms are asymmetric. Means there are 2 input clists(i.e. raw & canonical) but there is only one output clist(only canonical). Though, it is not mentioned in algorithm, when data from input clists is copied to user address space (i.e. process's local buffer), the line discipline modules then

process it and put on output clist. If this has to be symmetric, there should be 1 input clist and 1 output clist. But this is not. Because if this has to happen, data from raw input clist has to be processed by interrupt handler which is more complex & time consuming and other interrupts may get blocked at critical time. Using 2 input clists makes this easy because interrupt handler will simply put data in raw clist and wakes up reading process, which then by using above terminal-read() algorithm do the processing by line discipline modules. As the processing of raw data is done by reading process (via line discipline) now there is no burden on interrupt handler and thus it can put input characters quickly on output clist and thus user experiences very minimal delay in viewing typed characters on the terminal.

Consider following program, in which a process creates many(18) child processes, which are reading data from terminal i.e. from standard input(stdin) having file description 0.

```
/* global data */
char input[256];
void main(void)
{
        /* local data*/
        register int I;
        /* code */
        for(I=0;I<18;I++)
        {
                switch(fork())
                {
                        case –1 : /* failure of fork() */
                                        printf("Error in fork()\n");
                                        exit();
                        case 0 : /* success of fork() */
                                        for(; ;) /* indefinite loop */
                                        {
                                                read(0, input, 256);
                                                printf("%d reads %s\n", I, input);
                                        }
                        default : /* means go to parent process */
                                break;
                }
        }
}
```

As input depends upon user's typing speed, the terminal input is usually too slow to satisfy the reading process's request. So in above code most of the reading processes will sleep in terminal-read() algorithm(which will be called inside read() system call) and wait for input data. When user enters a line of characters on terminal, the terminal interrupt handler awakens all reading processes (slept before). As they all are slept on same priority level, they all are eligible for running with same priority. So user can never anticipate & predict which reading process (out of above 18) will get scheduled, run and thus will read the data. Whichever successful process will print the number of it (i.e. 0 to 17) and will also echo the read line on the terminal. At sometime all other processes will also run, but some other processes had already taken the input data, they will not find any data on the terminal input and thus will again go to sleep. Due to the "***indefinite loop***", this whole procedure will be repeated again and again and it impossible to say all processes will get some of the input data. Rather one process can hog (acquire) all the input data, keeping all other processes in starving condition.

Thus it is stated that, though kernel tries to give equal justice to all processes, it is inherently dangerous and ambiguous to allow multiple processes to read the terminal as shown in above example. But this doesn't mean kernel avoids multiple processes to read the terminal. Rather it does allow, because the shell and the process spawned by the shell read input data by the same above way. If kernel avoids, they will never work. Thus it is very important that programmers (i.e. processes) must synchronize their terminal access on programmatic level,(means processes do this at user level), by using synchronization methods of IPC.

The Line Discipline detects "***end of file***"(i.e. EOF or ASCII's Ctrl + d combination) character and terminates the respective I/O. But when it is working for terminal read(), then it reads the data up to EOF and not including EOF. If user enters nothing and just presses Ctrl + d, then it returns "***no data***" to the calling process. In short EOF is identifiable but not countable. Now it's up to calling process to determine that it reached EOF and thus should stop reading the terminal. So the loop inside the shell algorithm (seen in chapter (7)) terminates on pressing Ctrl + d, then terminal read() refuses and shell exits.

All above discussion up till now assumes that the terminal hardware is dumb (so called "***dumb terminal***"). Dumb terminal transmits the data to machine as "***one character at a time***", as user types. But intelligent terminals cook (i.e. process) their input in peripheral devices. So that CPU will be free for doing other jobs. Intelligent terminal's driver is similar to dumb terminal's driver, but the line discipline module algorithms are different and depend on the capabilities of peripheral devices where they will the cook the data.

Terminal Driver In Raw Mode

By using ioctl() system call, user can set & get the terminal driver parameter settings like erase & kill characters. By above system call, terminal's baud rate (baud rate means the rate of bit transfer), input echoing, input flushing, output queue flushing and setting of "***start up***" & "***stop***" characters manually, also can be done. The terminal driver's data structure like *termio* saves these all settings and later line discipline takes parameters of ioctl() and then gets or sets fields in *termio* structure according to the parameters. It is very important to note that, when any process sets these terminal parameters, and as they are global kernel data structures (i.e. termio is global), these changes gets reflected to all processes which are using the terminal currently. Not only this, but if the process which changed these parameters, gets exited, then changes made by it are not lost but remain in effect until the "***session end***" or until come other process resets them to default.

Process if wish can put the terminal into raw mode. In such cases "***input processing***" is not done. Line discipline module, in raw mode, forget their job of input cooking and instead transmit characters exactly as user typed them. But still the kernel must know when to satisfy the user's read() request. This is because the "***enter key***"(i.e. carriage return key) is treated as ordinary input character in raw mode and not the special character as in the canonical mode.

Kernel's job of satisfying user's read() call is based on 2 criteria:

(1) Either when a minimum number of characters are input at the terminal.

(2) Or when waiting for a fixed time after typing is started.

In 2nd case, means when kernel waits for a fixed time (like setting a timer), the timer starts when first character appears on the terminal and thus kernel starts making entries into the callout table.

Both or any of the both criteria can be set using ioctl() system call. When given criteria is fulfilled, the line discipline interrupt handler awakens all sleeping processes. Then driver moves all characters from raw clist to the canonical clist(lines 14 and 15 in the algorithm of terminal_read()) and satisfies the process's read() request by following same further lines of canonical case in that algorithm.

Raw mode is particularly for screen-oriented applications like *vi* editor. Many commands of *vi* editor do not terminate with Enter key. For example one of the command called *dw*(stands for *delete word*) deletes word at current curser position.

Following program is an example of using ioctl() system call. Here the ioctl() system call saves the current terminal's setting of file descriptor 0(as we know the file descriptor 0 represents standard input). The ioctl() system call contains various commands/constants used as 2nd parameter. One of the commands is *TCGETA*, which instructs the terminal driver to retrieve the current terminal settings and then force them to get saved in kernel's data structure *termio*. This command is commonly used to determine whether the file descriptor used, is a terminal or not. If not, means if the ioctl() system call fails, the calling process determines that the given file descriptor is not a terminal. If succeeds, means if the given file descriptor is of terminal, then process turns of canonical mode (to allow raw mode), then also turns of character echo and tells to satisfy the read() call for minimum of 5 characters are received from the terminal ***or*** when any number of characters are received and 10 seconds time interval is elapsed since first character was received. When the interrupt signal is received, the process resets the original terminal flags (in signal catcher function) and then exits.

```
# include <signal.h>
# include <termio.h>
struct termio savetty;
void main(void)
{
        /* declarations */
        extern void sigcatch(void);
        struct termio newtty;
        int nrd;
        char buf[32];
        /* code */
        signal(SIGINT, sigcatch);
        if(ioctl(0, TCGETA, &savetty) == -1)
        {
                printf("ioctl() failed : file is not tty.\n");
                exit();
        }
        newtty = savetty;
        newtty.c_lfalg = newtty.c_lfalg & ~ICANON; /* turn off canonical mode */
        newtty.c_lflag = newtty.c_lflag & ~ECHO; /* turn off char echo*/
        newtty.c_cc[VMIN] = 5; /* minimum 5 chars */
        newtty.c_cc[VTIME] = 100; /* 10sec interval */
        if(ioctl(0, TCSETAF, &newtty) == -1)
        {
                printf("can not put tty in raw mode.\n");
                exit();
        }
        for(; ;)
        {
                nrd = read(0, buf, sizeof(buf));
                buf[nrd] = 0;
                printf("read %d chars '%s'\n", nrd, buf);
         }
}
void sigcatch(void)
{
        ioctl(0, TCSETAF, &savetty);
        /* reset terminal flags back to original */
        exit();
```

```
}
```

Terminal Polling

Polling means read only when data is there, else continue regular processing. So it is sometimes much better to poll a device. If we refer to terminal-read() algorithm, there is a case which can keep terminal device in polling state. Means if terminal is opened with "***no delay***" option and if there is no data on the raw clist, then read() call will not sleep the process but instead will return immediately. This method of device polling will also work when we want to monitories many devices. So it can open each device with "***no delay***" option and poll all of them for waiting if any input is arriving for them. But this method of polling wastes processing power. Polling is illustrated in following program….

```
# include <fcntl.h>
void main(void)
{
        /* declarations */
        register int I, n;
        int fd;
        char buf[256];
        /* code */
        if((fd = open("/dev/tty", O_RDONLY | O_NDELAY)) == -1)
                exit();
        n = 1;
        for(; ;) /* infinite loop */
        {
                for(I=0;I<n;I++);
                if(read(fd, buf, sizeof(buf)) > 0)
                {
                        printf("read at n %d\n", n);
                        n--;
                }
                else
                        n++;
         }
}
```

In *BSD Unix* system, device polling is done by ***select()*** system call, whose syntax is:

select(nfds, rfds, wfds, efds, timeout);

Where

- *nfds* means numbers of file descriptors chosen for device files.
- And *rfds*, *wfds* & *efds* are "***bit masks***" which will select opened file descriptors. Means 1 << fd(1 is shifted left by above value of the file descriptor) is set if user wants to select fd file descriptor.
- The *timeout* field indicates how long select() system call should sleep, waiting for data arrival. Now if data arrives for any file descriptor and yet timeout is not expired, select() returns, indicating which file descriptor is selected. This indication is given in bit masks. For example, if user wishes to sleep until receiving input on file descriptors 0, 1 & 2(i.e. stdin, stdout & stderr),

then *rfds* will point to the bit masks 7. So when select() returns bit mask gets overwritten with a mask of selected file descriptor having data "***ready***" on it. This is for *rfds* i.e. read file descriptor. Similar things will happen for write file descriptor *wfds* bit masks and if *efds* is set then some exceptional conditions has occurred for particular file descriptors. Thus in-all select() is very useful in networking related device polling situations.

Establishing A Control Terminal

By terminology the "***Control Terminal***" means a terminal from where user "***logs in***" into the system. Obviously after successful login this terminal becomes responsible for user process started from it. So when process opens a terminal, the terminal driver opens line discipline. Now if the running process is a "***process group leader***"(set by prior setpgrp() system call) and if it still not having a "***control terminal***", then line discipline makes the opened terminal(i.e. which proves had tried to open in first line of this page) as its "***control terminal***". It stores major number and minor number of this terminal device file in process's u-area and stores process's group number (as this is a group leader) of this process into the terminal driver data structure. In Unix system this process is generally "***login shell***".

This "***control terminal***" plays important role in handling signals. As shell is a group leader, when user process delete, break, quit like keys, the interrupt handler invokes line discipline which then sends the appropriate signal to all processes which belong to this group. Similarly when user "***hangs up***", the terminal driver's interrupt handler receives the hang up indication from the hardware and line discipline now sends the "***hang up***" signal to all processes in this group and thus all processes of this group receive hang up signal. By default most of these processes react this signal by exiting and the group leader process kills the process when user suddenly shuts off the terminal. Obviously after sending "***hang up***" signal now this terminal is no longer remains a "***control terminal***" and hence terminal interrupt handler disassociates this terminal from the process group, so that, "***still living process***" of this group will not receive any signals from this terminal again.

*[Note: The redirected file can be called as "**indirect terminal**" for understanding purpose.]*

Indirect Terminal Driver

As we know, many times the terminal is redirected to some file by redirecting stdin and stdout. So obviously reading and writing will now proceed via the redirected files and not via the terminal. In other words terminal is not "***control terminal***" for the process any more.

Then a question may arise, what to do to send message to the "***actual control terminal***" even if it is redirected? E.g.: A shell script wants to send urgent messages directing to the control terminal, even if its (i.e. control terminal's) stdout & stderr are redirected somewhere else. UNIX system, for this purpose, provides "***indirect***" access to this control terminal via the device file */dev/tty*. Remember that */dev/tty* gives the control terminal to every process, which "***has***" a control terminal already. But users logged onto different terminal, through can access */dev/tty*, they actually access different terminals.

The Kernel can find the "*control terminal***" from the device file */dev/tty* by 2 ways:**

(1) Kernel can define a special device number for the "***indirect***" terminal file with a special entry in the character device switch table. So when wanted to invoke the indirect terminal, the indirect terminal gets the major number and minor number of the actual control terminal from u-area and invokes the actual control terminal's driver through the entry in character device switch table.

(2) Secondly, kernel checks the major number of the current indirect terminal before calling driver's open() routine. If it is of "***indirect terminal***", it releases the inode at */dev/tty*, allocates new inode for actual control terminal, resets its file table entries to point to this new inode and then calls driver's open() call routine of the actual control terminal's driver. So now the file descriptor of */dev/tty* opening is of real terminal (i.e. not redirected, indirect terminal) and the driver will be also of real terminal.

Logging In

In chapter 7, we saw that the process 1 i.e. init process has an infinite loop, in which it reads */etc/inittab* file to decide what to do on entering the "***single-user***" or "***multi-user***" state. In multi-user state, the most primary work of init process is to allow users to log into terminals. The algorithm of login() is as follows...

```
01 : algorithm : login()       /* procedure for logging in */
02 : {
03 :    execute getty process;
04 :    set process group by setpgrp() system call;
05 :    open tty line and sleep until gets opened;
06 :    if(opening of tty line is successful)
07 :    {
08 :            execute login program;
09 :            prompt for username;
10 :            turn off echo and prompt for password;
11 :            if(username and password are correct)
12 :            /* comparison is done with entry in /etc/passwd */
13 :            {
14 :                    put tty in canonical mode by ioctl();
15 :                    execute sheel;
16 :            }
17 :            else
18 :                    count login attempts and allow try again for login;
19 :    }
20 : }
```

The algorithm spawns a process called getty(stands for "***get terminal***" or for "***get tty***") and keeps track of which getty process opens which terminal. Each getty process resets its process group by setpgrp(), opens a particular tty line and usually sleeps until the machine senses a hardware connection for the terminal. When this opening of terminal returns successfully, the getty executes login program, which prompts for username & password for logging in. If logging is successful, login program executes shell and user starts working on this shell. This shell is thus called as "***login shell***". This shell has its pid same that of getty and thus it is a group leader (as getty previously called setpgrp()). If login is not successful (i.e. wrong username and password), login program exits after a time limit, closing the opened terminal line and init(due to infinite loop) now spawns another getty process for the terminal line. Init pauses until it receives a "***death of child***" signal. When "***death of child***" signal arrives, it wakes up, find out zombie process, looks whether it had any login shell and if it is has one, spawns anther getty process to open the terminal in place at one that died.

Streams

The device driver implementation in UNIX, though sufficient, had some drawbacks. Major drawbacks were two...

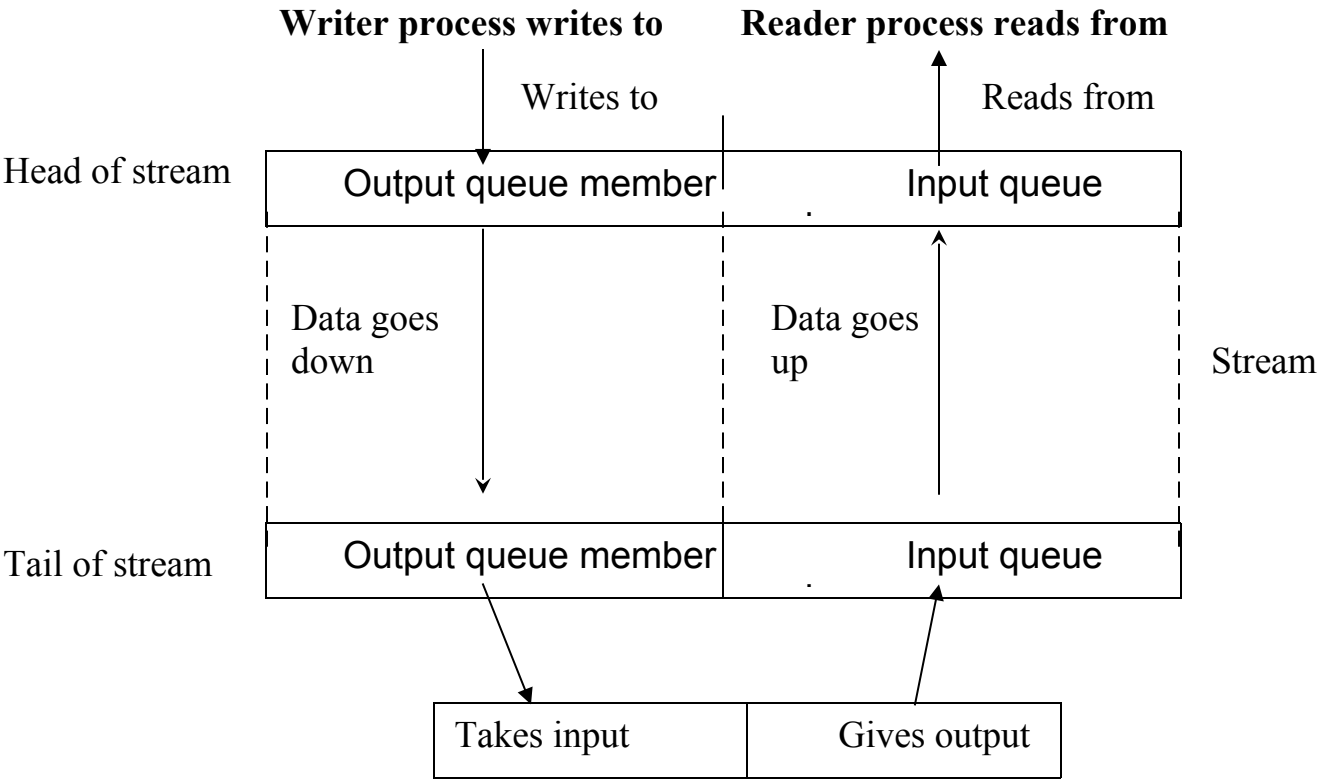
(1) Drivers, which implemented network protocols, tend to duplicate their functionality unnecessarily. These drivers have two portions. One the device control portion and second is the actual protocol portion. Actually the protocol portion should be common to all network devices but in practice, they are not. This is because kernel did not provide enough common mechanism for them. For example, clists are useful due to their internal buffering mechanism, but can be expensive due to their character-by-character manipulation. To avoid this character-by-character manipulation, if we bypass clists, the modularity of the I/O subsystem breaks down.

(2) The network protocols require line discipline like feature where each discipline should implement one part of the protocol and the component part then can be combined with flexibility. But practically, in Unix Kernel it is difficult to stack such line disciplines together.

Owing to above drawbacks, **Dennis Ritchie**, implemented a scheme called as streams. These give more modularity and flexibility for I/O subsystems. Theoretically a stream is a full-duplex(i.e. both sided) connection between a process and a device driver. Data structure wise a stream is a set of liner linked queue pairs, where one member of the pair (i.e. one queue of the two) is for input and other is for output.

So when a writing process writes data on a stream for sending it to device driver, though it is input for stream, the stream is ultimately going to output it to device driver which is obviously input for the device driver. In short process's writing is input to stream then stream's output is input to driver. Thus data written by the process is sent down to the output queue member (out of the pair), which then goes down the output queue member and finally given to the driver. Driver takes this data (which is now in output queue member) as its input and sends this data to input queue member of stream up words and finally the input queue member sends it to the reading process.

Now if we want to use this "***stream***" structure, we must make our I/O subsystem functions capable of using stream and its data structure. Thus each "***queue pair***" of stream is associated with kernel's I/O subsystem module like driver module or line discipline module or protocol module. In other words these modules can use "***queue pair***" as if their inbuilt parts.



Device Driver

Obviously "***queue pair***" in process or in driver or in line discipline or in protocol must be able to pass data to each other as per the requirement. The data sent in the form of a message via well-defined interfaces. These modules also can manipulate the data passing through queues as needed.

Each queue is a data structure having following elements…

1) Queue's open procedure, which is called inside the open() system call.
2) Queue's put procedure, which is called during message passing from queue to queue.
3) Queue's service procedure, which is called when a queue is scheduled to execute.
4) Queue's close procedure, which is called inside the close() system call.
5) Pointer to next Queue in the stream.
6) Pointer to a list of those messages, which are waiting for service of queue.
7) Pointer to a private data structure, which maintains the state of queue.
8) Flags & high-low water level marks, used for flow control, scheduling and maintaining queue states.

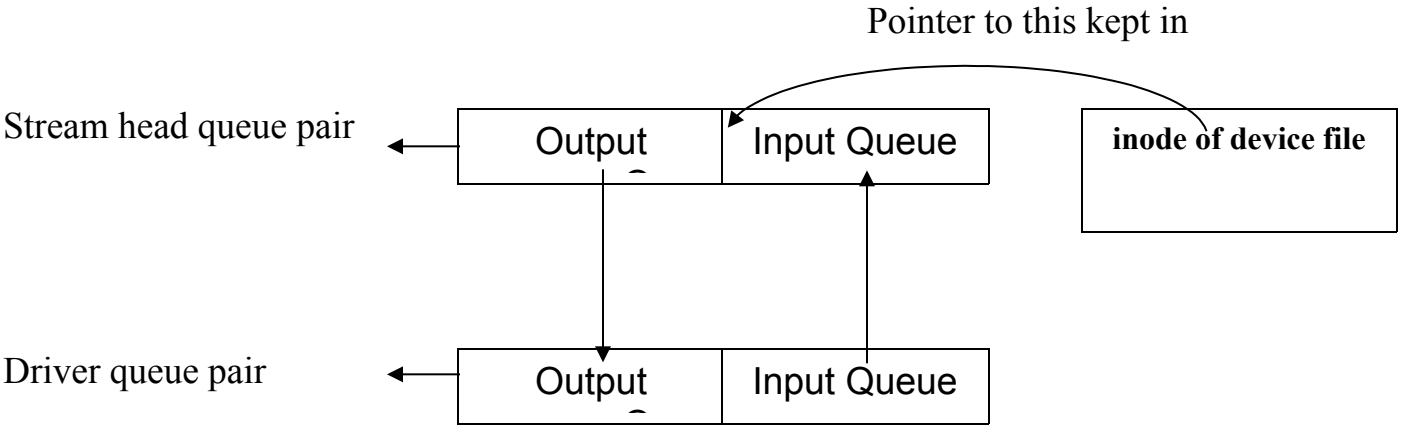
While memory allocation, kernel allocates queue pair "***adjacent memory locations***" so that one-member of a queue pair can easily find other member of the pair.

When a device driver has stream support, then it is sure that it is a character device. So such driver has a special field in character device switch table. This field is a pointer to "***streams initialization structure***". In turn this "***streams initialization structure***" contains all 8 fields explained on last page. So when kernel executes open() system call for such device and finds that this device is character special device, then it specifically looks for this stream concern field in character device switch table. If there is no entry of this field, it bypasses stream scheme and follows usual device I/O procedures for character devices. But if it finds this stream concern field, then in the stream's open(), it allocates memory for 2 pairs of queues. One pair is termed as "***stream head pair***" and other pair is termed as "***driver pair***"(for understanding we can term it as "***stream tail pair***"). For the sake of modularity & to avoid duplication, the "***stream head pair***" is made "***generic***" means the module of stream head pair is same for all instances of stream's open(). In other words, suppose since system starting, no device with stream support is yet invoked and if one started now, then the stream-head module started for this driver is not device specific. Means now stream head module will remain in memory and will serve for all instances of stream's open() for different devices. That's why we called it as "***generic***". Due to its generic ness, it has generic put() & generic service() procedures(contents 2 & 3 on last page). Recall that the stream's open() is called inside the process's open() system call. Naturally this shows that stream's head routines are interfaces to high level functions of devices to those processes call i.e. open(), read(), write(), ioctl() etc.

[The word stream's open() means the queue's open() explained on last page 1st content.]

Similar to "***stream-head queue pair***", kernel also initializes "***driver queue pair***". The important thing it does is that, it copies address of driver specific functions to the pointers in "***driver queue pair***" so that when stream's functions are called, indirectly driver functions get called. As devices are different, drivers are different and thus "***driver queue pair***" will be different for each driver. In other words it won't be generic like stream head pair. A question may arise, from where it finds addresses of driver specific functions? Each driver has its own per-driver initialization structure in memory. In this structure these are addresses of driver specific functions.

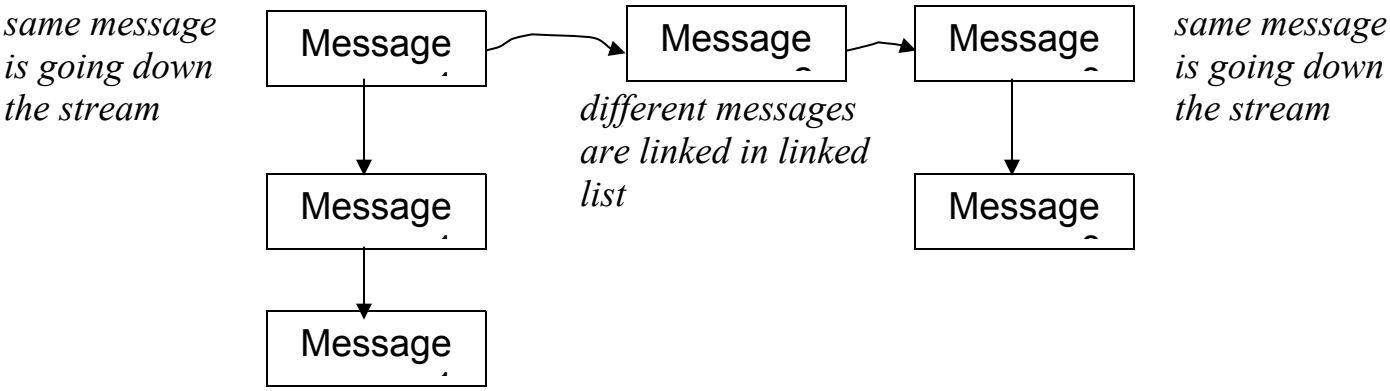
Now its time for the kernel, to call driver's open() routine(recall that from last page we are algorithmically in open() system call called by kernel on behalf of a process). In driver's open() routine we need device file's inode. And not only for this call, but whether we open same device we need same stream structure to get started. To allow this kernel puts a special pointer in device file's incore inode(not in disk inode of device file) and puts pointer to "***stream head pair***" in this special pointer. So when some other process opens the same device, it finds device file's inode as usual and by this special pointer it finds the associated stream with this device. Once "***stream head queue***" found, now it's easy to call open() calls of all modules by open() of queue pair.



Now we know that, queue data structure of a stream's queue pair contains routines. One queue's one routine communicates with the routine on the neighboring queue(i.e. other queue in the pair) by passing messages. So its time to see structure of this message. A message is made up of linked list of message block headers. Each of this message block header points to the start and end of location of block's data. There are 2 types of messages in stream. One is control message and other is data message. Message header has a type indicator to identify which type of this message is.

- Control message mat come as a result of ioctl() system call or as a result of special conditions like terminal's hang-up.
- Data message may come as a result of write() system call or as a result of arrival of data from a device like terminal.

Now its time to see how streams & messages work together. When a write() system call writes data on a device, whose stream is already opened, the kernel copies data from user address space(i.e. process's local buffer) to the message block allocated by the stream-head(contain no 6 of queue data structure). Now the stream head's put() generic routine calls put() of the next queue(down the line of the output queue, not the one of pair) in the line which in turn calls put routine of next queue and so on. Thus data flows down the stream via put() or if the data or say message needs further processing, put() may keep such processable message/data on the message-linked list. As message on the list are either passed directly to put() of next queue module or put in linked list for further processing, the message linked list gets the appearance of Two Way Linked List.



Then the put() module(of whatever queue in the stream) sets a flag in its queue data structure, which indicates that "***it has a data to be processed***", and importantly it schedules itself for servicing. Means it schedule itself to call service() procedure in it.

Now such scheduled queue is put into the linked list of such queues which request service() routine of them to be called by scheduler after time-up. So when time comes, the scheduler calls service() of each queue in this special linked list.

The scheduling is done by kernel on the basis of software interrupt (similar to that of calling scheduled functions in callout table as seen in the chapter of process scheduling) where the software interrupt handler calls the service() routine of scheduled queue on this special list.

Summing-up, we can say that stream is a common scheme for all devices to avoid duplication of functionality in their driver's code. One of the module in the driver when wants to be part of a stream, it allocates memory to stream's queue data structure and puts address of its own open(), close(), write() like routines into the function pointers of open(), close(), put() routines in stream's queue respectively and become part of the stream scheme. Many such modules by their own queue pairs join together to form a complete stream, right from process's high level system call to device driver's low level routines. Now data gets passed along the stream "***down***" the line while writing and "***up***" the line while reading during device I/O.

For example, suppose a device supports scheme. It is initialized, means its stream-head queue pair and its driver queue pairs are alive in memory. Some process had already worked with this initialization and thus stream is alive in memory and yet not freed. Now suppose a new process wants to use this stream, but it needs line discipline module and yet the line discipline module is not part at stream. So this new process wants "***line discipline***" to become part of stream and thus wants to push "***line discipline***" module on the stream. How this will happen?

Process can do this by using ioctl() system call. When process will open the device say */dev/ttyxy* by open() system call ***fd = open("/dev/ttyxy",O-RDWR);*** inside which it will become part at already opened stream.(if stream is not opened yet, open() will open the stream as explained before). Then it will tell the kernel that for this opened device's stream, it wants to push line discipline module by using ioctl() system call ***ioctl(fd, push, TTYLD);*** where *TTYLD* stands for **t**ele-**ty**pe terminal **l**ine **d**iscipline. Actually *TTYLD* is a macro indicating a number that identifies line discipline module.

Due to this ioctl() call with PUSH command, kernel interest the line discipline module immediately below the stream-head queue pair, and connects queue pointers of the stream-head pair with the queue pointers at the line discipline and then connects queue pointers at line discipline with the queue pointers of module lower to it down the line(in this example the lower module of the line discipline in driver). The queue painters are contents no. 5 of queue data structure. This connection keeps the doubly linked list nature of stream. The lower modules on the stream do not care for whether they are connected upwards with the stream-head or with the line discipline. The common point along all modules in the stream is put() procedures of all queues in the stream. So when stream-head becomes capable of calling put() of line discipline and when line discipline becomes capable of calling put() of lower module i.e. here driver's put(), we can say that new module is successfully pushed. There is no restriction on number of modules for pushing on the stream.

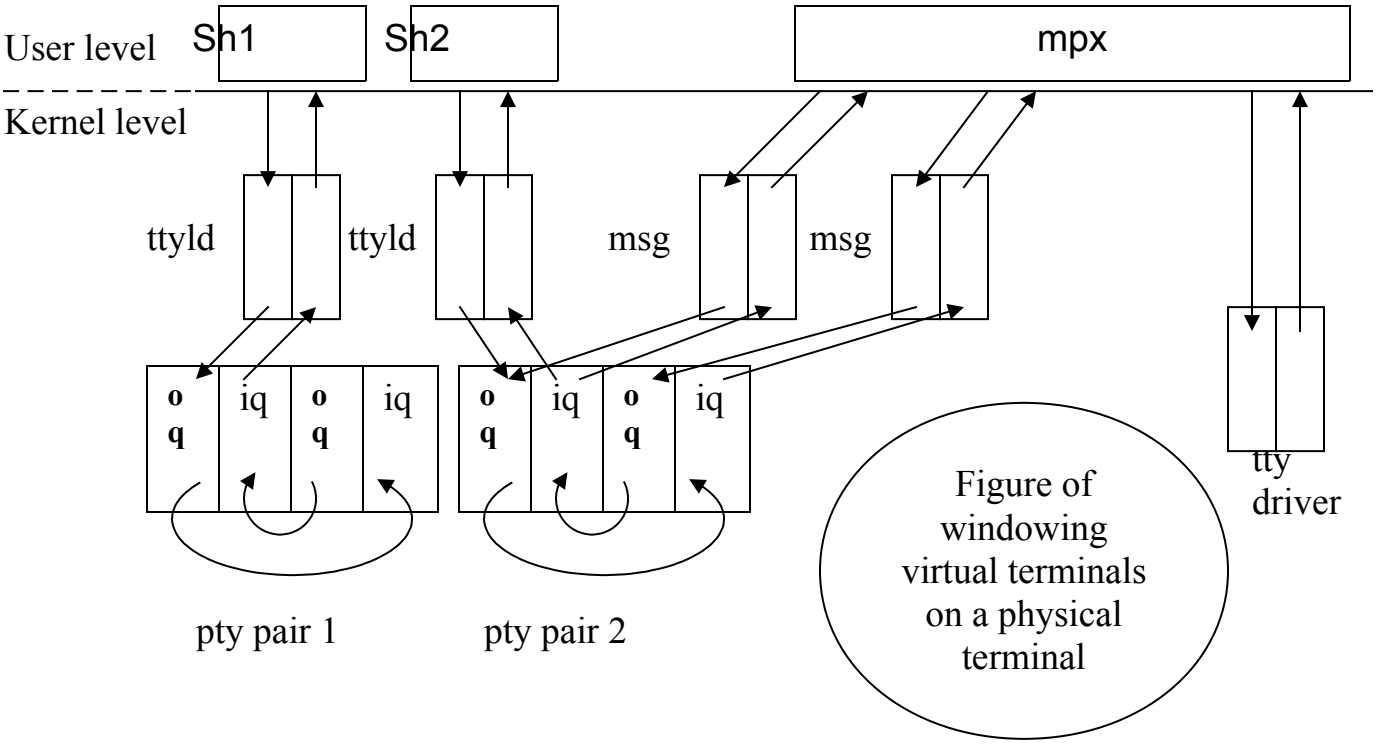
Similar to push, a process can "***pop***"(remove) a module from the stream, but in stack manner. Means most recently pushed module will be the first one to get popped.(i.e. LIFO). So ***ioctl(fd, pop, !);*** will do the popping of recently pushed module onto the stream. Repetitive calls will pop more & more. As line discipline module here, does not care about the modules lower to it, the driver(i.e. indirectly device) can be a simple terminal or can be a network device and polymorphism is achieved.

One last and important thing remained to explain is that, the line discipline explained before and the line discipline pushed on the stream, though the functionality identical, are not at all same. Because the line discipline which supports stream needs to allocate memory for queue pair of stream and needs to do its initialization so as to become part of the stream. So interfaces that connect with the driver are different for these two versions of line discipline. Same logic is applicable to all such modules along the stream and on the top also applicable to driver module too.

Before pushing line discipline, in above example, there will be no processing of written data and thus driver will receive un-processed data as was in raw mode. After pushing line discipline, the same stream & the same driver will now process written data for formatting as was in canonical mode. This also tells us that data arriving at stream-head is always un-processed and whether to process it further or not depends upon the modules below it.

Detailed Example Of Stream

Robe Pike (one of the author of "***programming In UNIX Environment***") described an implementation of "***Multiplexed Virtual Terminals***" using the idea of stream scheme. On the desktop, user looks several virtual terminals; each of them occupies a separate window (a GUI window) on a physical terminal. Pikes idea though worked for GUI terminal, it can work for a single, dumb terminal too (Here dumb terminal means "***DOS Prompt***" occupying entire screen when MS Window 98 is started in "***MS-DOS***" by selecting 3rd option in Shutdown dialog box). Applying this idea dumb terminal, each window (i.e. virtual terminal) will occupy the entire screen. For GUI based window terminals, minimizing one and maximizing other do switching between each window. While for dumb terminal, switching between multiple window can be done by control key combinations.



Above figure shows arrangement of processes (on User level) & kernel modules (kernel level). The user executes a process called *mpx*(stands for multiplex) to control the physical terminal(i.e. the actual terminal device). The *mpx* reads the physical terminal and writes for any notifications by events occurring on terminal, such as creating new window (i.e. virtual terminal), switching between multiple windows (i.e. switching between multiple virtual terminals) and so on.

When *mpx* receives a notification event, suppose creation of a new window, *mpx* spawns a new process to control this newly created window, and starts communicating with it by using a pseudo terminal (abbreviated as pty). This pty resembles a queue pair of stream idea. Output is directed to output member of the pair, which in turn is directed to input member of other pair.(Remember, not to input member of same pair). The input queue member then sends this input "***up***" the stream to whatever kernel module (i.e. like *ttyld* or *msg*) above it.

Following code is a pseudo-code to setup a window...

```
/* assume that 0 and 1 file descriptors already refer to stdin and stdout of physical device /dev/tty */

for(; ;) /* beginning of the loop */
{
```

```
        select(input);  /* and wait for some line of data */
        read input line;
        switch(line with input data)
        {
                case physical tty : /* input is on physical*/
                                        if(control command occurs) /* e.g. create new window */
                                        {
                                                open a free pseudo tty;
                                           /* means create/open one member of pty*/
                                                fork a new process;
                                                if(parent)
                                                {
                                                        push a msg discipline on mpx side;
                                                        continue;
                                                }
                                                /* child's code */
                                                close unnecessary file descriptors;
                                                open other member of pseudo tty pair and get stdin,
                                                stdout, stderr;
                                                push tty line discipline;
                                                execute shell; /* this now looks like virtual tty */
                                        }
                /* if "not" control command, but a regular data is coming for virtual tty */
                demuliplex data "read" from virtual tty, strip off headers and write to appropriate pty;
                continue;
                case logical tty : /*  i.e. virtual tty */
                                        encode headers indicating for which window data is coming;
                                        write header and data to physical tty;
                                        continue;
         }
}/* end of for loop*/
```

According to this pseudo-code of multiplexing windows, mpx allocate a pty queue pair and opens one member of it indicating the beginning of stream idea. Here the *tty* driver will check whether pty was previously allocated or not. If not, it will do as above and if yes, it will not allocate new, but will use the already allocated one.

Then *mpx* forks a new process and that new process will open other remaining member of pty queue pair (see the child's code). The *mpx*, in its code (i.e. parent's code) pushes a message module (i.e. msg) onto its pty stream connection. This message module is responsible for conversion of control messages to data messages. Where as the child process (see child's code) pushes a line discipline module onto its pty stream connection and then executes the shell (i.e. *sh1* or *sh2* in the figure). The shell, thus now running on a virtual terminal. Surprisingly, for the user it is indistinguishable from the physical terminal.

The *mpx* process, as its abbreviation states, is a multiplexer process (i.e. splitter process) which forwards output coming from virtual terminals (from user's point of view it is his input) to the physical terminal and then demultiplexes input from physical terminal (user does not see it although it does not matter to him) to the correct virtual terminal.

The *mpx* process can wait for "***arrival of data***" on any terminal line (whether logical pty or actual physical *tty*) by using select() system call(Recall about select() system call of BSD). When data arrives from physical terminal, *mpx* decides, whether data is a "***control message***"(such as creation of a new window) or whether data is a "***data message***" to be sent to the reading processes which are

reading virtual terminals (i.e. here shells). When data is in the form of a "***data message***", its message header indicates, to which virtual terminal, it is targeted for. Now *mpx* cuts this header part (in code, explained as "***strips off***") and separates actual data part and writes this actual data to the appropriate pty stream. The data then passes "***down***" the stream and ultimately driver routes the data to reading process "***up***" the stream. Reversal of this process occurs when data is written to the virtual terminal. Means mpx takes data from virtual terminal, attaches header to it(with information about from which terminal this data is coming) converts it into "***message***" form, sends it "***up***" the message module(msg) part of stream. Then this "***messaged***" data is sent by *mpx* to actual physical terminal "***down***" the line and *tty* driver gets the information to which window (i.e. *pty*) this terminal is to be printed.

Now if a process (viewing virtual terminal. E.g.: shell) calls ioctl() system call on virtual terminal, the line discipline modules set the required terminal settings(as indicated in ioctl()) for the virtual terminal line *pty*. As virtual terminals can be of n numbers, the settings may be different for each of them. But still as physical terminal is "***only one***", some information needs to be sent to it depending on type of device. The message module converts the control messages, generated by ioctl(), into data messages suitable for reading & writing by *mpx* process and then *mpx* transmits them to physical terminal device tty.

Analysis Of Streams

Dennis Ritchie mentions that he tried to implement streams either only with put() procedure or only with service() procedure and never both. But it seems that service() procedure is must in any circumstances, to control data flow in stream modules because sometimes or other, modules, instead of processing message may put message on message list temporarily. Similarly put() procedure is also necessary one, because sometimes data needs to be delivered directly to a neighboring module directly. For example a terminal time discipline module must echo input data back to the terminal as immediately as possible. Also it becomes possible for write() system call to call put() procedure of next queue which in turn will call put() of the next one and so on. This can be done without the need of scheduling mechanism.

A process may sleep if output queue gets congested but stream module cannot sleep on input side because interrupt handler invokes it. But as input, due to above reason, goes on, an innocent process on output side may sleep due to incoming crowding.

There is yet another issue concern with stream. The inter module communication cannot be symmetric in the input & output directions.

Some may think that the stream, instead of making with modules can be made by process. But stream may contain large number of modules and if all of them are converted to separate process and all of them get started when stream starts, then they all may overflow the process table.

As states before, modules in stream are functions and not processes. They also have scheduling mechanism. But this mechanism, as they are not processes, different from regular process scheduling mechanism. Their scheduling mechanism work on "***software interrupt***", but as process can go to sleep, modules cannot. Instead they tend the interrupted process to go to sleep. Obviously modules must keep their own "***state transition***" state internally as they can not get benefit of default scheduling mechanism. This makes their code more complicated than the code, if sleeping allowed.

Stream scheme suffers from several anomalies like.

- ✓ Process accounting & statistics become difficult, because modules in stream do not necessarily run in context of the process that started the stream. Some processes using stream, do complicated jobs like Networking while some processes using stream may do simple jobs like terminal line discipline. So we cannot say the process shares that stream uniformly.

- ✓ Users can put a terminal into raw mode, such that read call will return after a short time, if no data is available to read. This cannot be implemented by using stream easily. It has to be done in stream-head module, which as generic, become un-necessarily common to all.
- ✓ In multiplexing window's example the property of multiplexing is used on "***user level***" process(e.g.: *mpx*) because multiplexing in kernel level is difficult due to the linear connection between streams.

In-spite of these and some other anomalies, streams hold great promise for improving the design of device driver modules. One of the most popular implementation-using streams is Audio-Video streaming, which can be regardless of machine boundaries.