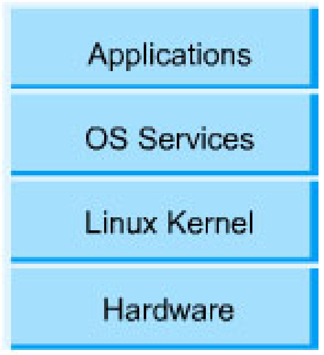
# Module 20: More on LINUX

## Linux Kernel Architecture

**The Big Picture:**

It is a good idea to look at the Linux kernel within the overall system's overall context..



**Applications and OS services:**

These are the user application running on the Linux system. These applications are not fixed but typically include applications like email clients, text processors etc. OS services include utilities and services that are traditionally considered part of an OS like the windowing system, shells, programming interface to the kernel, the libraries and compilers etc.

**Linux Kernel:**

Kernel abstracts the hardware to the upper layers. The kernel presents the same view of the hardware even if the underlying hardware is ifferent. It mediates and controls access to system resources.

**Hardware:**

This layer consists of the physical resources of the system that finally do the actual work. This includes the CPU, the hard disk, the parallel port controllers, the system RAM etc.

**The Linux Kernel:**

After looking at the big picture we should zoom into the Linux kernel to get a closer look.

**Purpose of the Kernel:**

The Linux kernel presents a virtual machine interface to user processes. Processes are written without needing any knowledge (most of the time) of the type of the physical hardware that constitutes the computer. The Linux kernel abstracts all hardware into a consistent interface.

In addition, Linux Kernel supports multi-tasking in a manner that is transparent to user processes: each process can act as though it is the only process on the computer, with exclusive use of main memory and other hardware resources. The kernel actually runs several processes concurrently, and mediates access to hardware resources so that each process has fair access while inter-process security is maintained.

The kernel code executes in privileged mode called kernel mode. Any code that does not need to run in privileged mode is put in the system library. The interesting thing about Linux kernel is that it has a modular architecture – even with binary codes: Linux kernel can load (and unload) modules dynamically (at run time) just as it can load or unload the system library modules.

Here we shall explore the conceptual view of the kernel without really bothering about the implementation issues (which keep on constantly changing any way). Kernel code provides for arbitrations and for protected access to HW resources. Kernel supports services for the applications through the system libraries. System calls within applications (may be written in C) may also use system library. For instance, the buffered file handling is operated and managed by Linux kernel through system libraries. Programs like utilities that are needed to initialize the system and configure network devices are classed as user mode programs and do not run with kernel privileges (unlike in Unix). Programs like those that handle login requests are run as system utilities and also do not require kernel privileges (unlike in Unix).

**The Linux Kernel Structure Overview:**

The "loadable" kernel modules execute in the privileged kernel mode – and therefore have the capabilities to communicate with all of HW.

Linux kernel source code is free. People may develop their own kernel modules. However, this requires recompiling, linking and loading. Such a code can be distributed under GPL. More often the modality is:

Start with the standard minimal basic kernel module. Then enrich the environment by the addition of customized drivers.

This is the route presently most people in the embedded system area are adopting world-wide.

The commonly loaded Linux system kernel can be thought of comprising of the following main components:

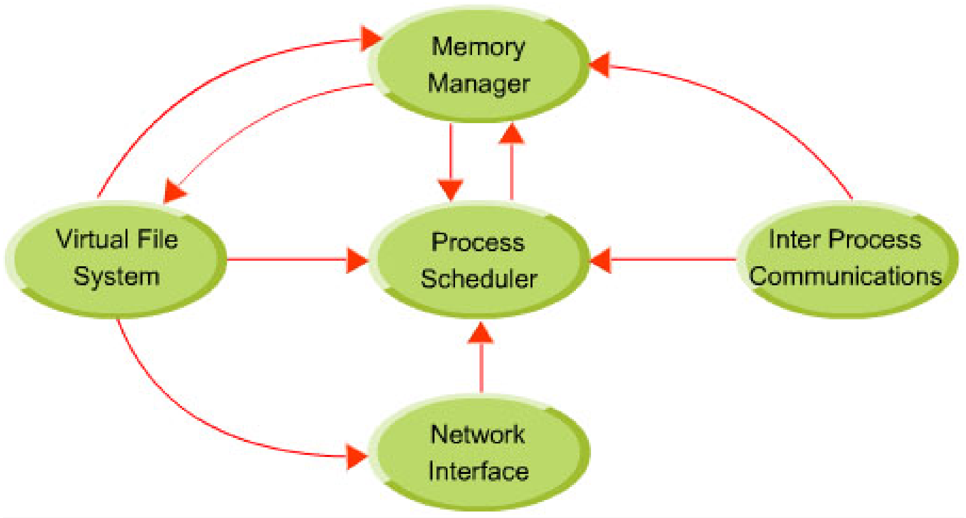
**Process Management**: User process as also the kernel processes seek the cpu and other services. Usually a fork system call results in creating a new process. System call *execve* results in execution of a newly forked process. Processes, have an id (PID) and also have a user id (UID) like in Unix. Linux additionally has a personality associated with a process. Personality of a process is used by emulation libraries to be able to cater to a range of implementations. Usually a forked process inherits parent's environment.

In Linux Two vectors define a process: these are argument vector and environment vector. The environment vector essentially has a (name, value) value list wherein different environment variable values are specified. The argument vector has the command line arguments used by the process. Usually the environment is inherited however, upon execution of *execve* the process body may be redefined with a new set of environment variables. This helps in the customization of a process's operational environment. Usually a process also has some indication on its scheduling context. Typically a process context includes information on scheduling, accounting, file tables, capability on signal handling and virtual memory context.

In Linux, internally, both processes and threads have the same kind of representation. Linux processes and threads are POSIX compliant and are supported by a threads library package which provides for two kinds of threads: user and kernel. User-controlled scheduling can be used for user threads. The kernel threads are scheduled by the kernel. While in a single processor environment there can be only one kernel thread scheduled. In a multiprocessor environment one can use the kernel supported library and clone system call to have multiple kernel threads created and scheduled.

**Scheduler:**

Schedulers control the access to CPU by implementing some policy such that the CPU is shared in a way that is fair and also the system stability is maintained. In Linux scheduling is required for the user processes and the kernel tasks. Kernel tasks may be internal tasks on behalf of the drivers or initiated by user processes requiring specific OS services. Examples are: a page fault (induced by a user process) or because some device driver raises an interrupt. In Linux, normally, the kernel mode of operation can not be pre-empted. Kernel code runs to completion - unless it results in a page fault, or an interrupt of some kind or kernel code it self calls the scheduler. Linux is a time sharing system. So a timer interrupt happens and rescheduling may be initiated at that time. Linux uses a credit based scheduling algorithm. The process with the highest credits gets scheduled. The credits are revised after every run. If all run-able processes exhaust all the credits a priority based fresh credit allocation takes place. The crediting system usually gives higher credits to interactive or IO bound processes – as these require immediate responses from a user. Linux also implements Unix like nice process characterization.



**The Memory Manager:**

Memory manager manages the allocation and de-allocation of system memory amongst the processes that may be executing concurrently at any time on the system. The memory manager ensures that these processes do not end up corrupting each other's memory area. Also, this module is responsible for implementing virtual memory and the paging mechanism within it. The loadable kernel modules are managed in two stages:

First the loader seeks memory allocation from the kernel. Next the kernel returns the address of the area for loading the new module.

- The linking for symbols is handled by the compiler because whenever a new module is loaded recompilation is imperative.

**The Virtual File System (VFS):**

Presents a consistent file system interface to the kernel. This allows the kernel code to be independent of the various file systems that may be supported (details on virtual file system VFS follow under the files system).

**The Network Interface:**

Provides kernel access to various network hardware and protocols.

**Inter Process Communication (IPC):**

The IPC primitives for processes also reside on the same system. With the explanation above we should think of the typical loadable kernel module in Linux to have three main components:

- Module management,
- Driver registration and
- Conflict resolution mechanism.

**Module Management:**

For new modules this is done at two levels – the management of kernel referenced symbols and the management of the code in kernel memory. The Linux kernel maintains a symbol table and symbols defined here can be exported (that is these definitions can be used elsewhere) explicitly. The new module must seek these symbols. In fact this is like having an external definition in C and then getting the definition at the kernel compile time. The module management system also defines all the required communications interfaces for this newly inserted module. With this done, processes can request the services (may be of a device driver) from this module.

**Driver registration:**

The kernel maintains a dynamic table which gets modified once a new module is added – some times one may wish to delete also. In writing these modules care is taken to ensure that initializations and cleaning up operations are defined for the driver. A module may register one or more drivers of one or more types of drivers. Usually the registration of drivers is maintained in a registration table of the module.

The registration of drives entails the following:

1. Driver context identification: as a character or bulk device or a network driver.
2. File system context: essentially the routines employed to store files in Linux virtual file system or network file system like NFS.
3. Network protocols and packet filtering rules.
4. File formats for executable and other files.

**Conflict Resolution:**

The PC hardware configuration is supported by a large number of chip set configurations and with a large range of drivers for SCSI devices, video display devices and adapters, network cards. This results in the situation where we have module device drivers which vary over a very wide range of capabilities and options. This necessitates a conflict resolution mechanism to resolve accesses in a variety of conflicting concurrent accesses. The conflict resolution mechanisms help in preventing modules from having an access conflict to the HW – for example an access to a printer. Modules usually identify the HW resources it needs at the time of loading and the kernel makes these available by using a reservation table. The kernel usually maintains information on the address to be used for accessing HW - be it DMA channel or an interrupt line. The drivers avail kernel services to access HW resources.

**System Calls:**

Let us explore how system calls are handled. A user space process enters the kernel.

From this point the mechanism is some what CPU architecture dependent. Most common examples of system calls are: - open, close, read, write, exit, fork, exec, kill, socket calls etc.

The Linux Kernel 2.4 is non preemptable. Implying once a system call is executing it will run till it is finished or it relinquishes control of the CPU. However, Linux kernel 2.6 has been made partly preemptable. This has improved the responsiveness considerably and the system behavior is less 'jerky'.

**Systems Call Interface in Linux:**

System call is the interface with which a program in user space program accesses kernel functionality. At a very high level it can be thought of as a user process calling a function in the Linux Kernel. Even though this would seem like a normal C function call, it is in fact handled differently. The user process does not issue a system call directly - in stead, it is internally invoked by the C library.

Linux has a fixed number of system calls that are reconciled at compile time. A user process can access only these finite set of services via the system call interface. Each system call has a unique identifying number. The exact mechanism of a system call implementation is platform dependent. Below we discuss how it is done in the x86 architecture.

To invoke a system call in x86 architecture, the following needs to be done. First, a system call number is put into the EAX hardware register. Arguments to the system call are put into other hardware registers. Then the int0x80 software interrupt is issued which then invokes the kernel service.

Adding one's own system call is a pretty straight forward (almost) in Linux. Let us try to implement our own simple system call which we will call 'simple' and whose source we will put in simple.c.

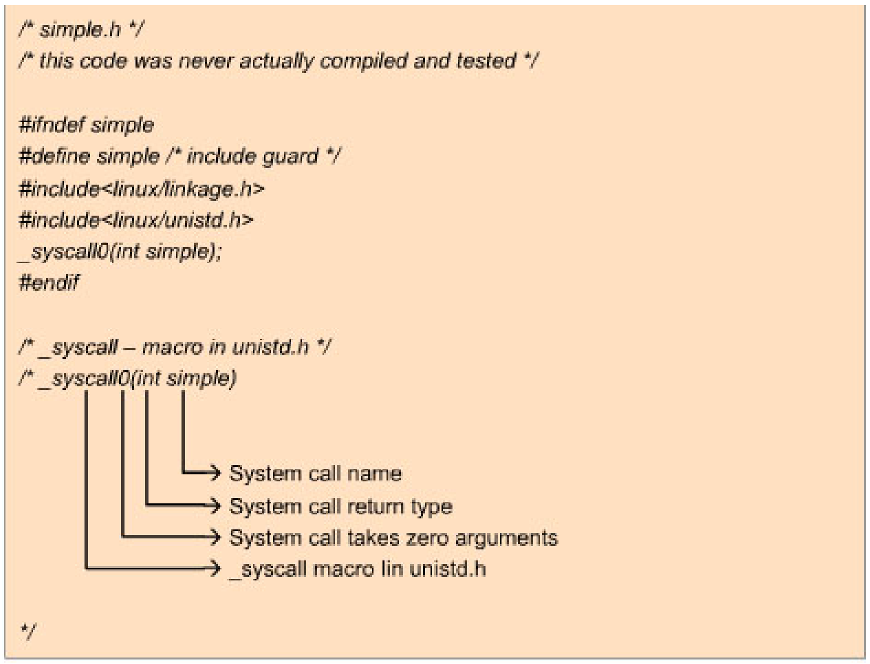
```
/* simple.c */
/* this code was never actually compiled and tested */
#include<linux/simple.h>
asmlinkage int sys_simple(void)
{
return 99;
}
```

As can be seen that this a very dumb system call that does nothing but return 99. But that is enough for our purpose of understanding the basics.

This file now has to be added to the Linux source tree for compilation by executing: /usr/src/linux.*.*/simple.c

Those who are not familiar with kernel programming might wonder what "asmlinkage" stands for in the system call. 'C' language does not allow access hardware directly. So, some assembly code is required to access the EAX register etc. The asmlinkage macro does the dirty work fortunately.

The asmlinkage macro is defined in *XXXX/linkage.h.* It initiates another macro_syscall in *XXXXX/unistd.h.* The header file for a typical system call will contain the following.



```
/* simple.h */
/* this code was never actually compiled and tested */

#ifndef simple
#define simple /* include guard */
#include<linux/linkage.h>
#include<linux/unistd.h>
_syscall0(int simple);
#endif

/* _syscall – macro in unistd.h */
/* _syscall0(int simple)

*/
```

After defining the system call we need to assign a system call number. This can be done by adding a line to the file unistd.h . unistd.h has a series of #defines of the form:

#define _NR_sys_exit 1

Now if the last system call number is 223 then we enter the following line at the bottom

#define _NR_sys_simple 224

After assigning a number to the system call it is entered into system call table. The system call number is the index into a table that contains a pointer to the actual routine. This table is defined in the kernel file 'entry.S' .We add the following line to the file :

* this code was never actually compiled and tested

*/.long SYSMBOL_NAME(sys_simple)

Finally, we need to modify the makefile so that our system call is added to the kernel when it is compiled. If we look at the file /usr/src/linux.*.*/kernel/Makefile we get a line of the following format.

obj_y= sched.o + dn.o …….etc we add: obj_y += simple.o

Now we need to recompile the kernel. Note that there is no need to change the config file. With the source code of the Linux freely available, it is possible for users to make their own versions of the kernel. A user can take the source code select only the parts of the kernel that are relevant to him and leave out the rest. It is possible to get a working Linux kernel in single 1.44 MB floppy disk. A user can modify the source for the kernel so that the kernel suits a targeted application better. This is one of the reasons why Linux is the successful (and preferred) platform for developing embedded systems In fact, Linux has reopened the world of system programming.

**The Memory Management Issues**

The two major components in Linux memory management are:

- The page management
- The virtual memory management

1. The page management: Pages are usually of a size which is a power of 2. Given the main memory Linux allocates a group of pages using a buddy system. The allocation is the responsibility of a software called "page allocator". Page allocator software is responsible for both allocation, as well as, freeing the memory. The basic memory allocator uses a buddy heap which allocates a contiguous area of size $2^n$ > the required memory with minimum $n$ obtained by successive generation of "buddies" of equal size. We explain the buddy allocation using an example.

   **An Example:** Suppose we need memory of size 1556 words. Starting with a memory size 16K we would proceed as follows:

   1. First create 2 buddies of size 8k from the given memory size ie. 16K
   2. From one of the 8K buddy create two buddies of size 4K each
   3. From one of the 4k buddy create two buddies of size 2K each.
   4. Use one of the most recently generated buddies to accommodate the 1556 size memory requirement.

Note that for a requirement of 1556 words, memory chunk of size 2K words satisfies the property of being the smallest chunk larger than the required size.

Possibly some more concepts on page replacement, page aging, page flushing and the changes done in Linux 2.4 and 2.6 in these areas.

2. **Virtual memory Management:** The basic idea of a virtual memory system is to expose address space to a process. A process should have the entire address space exposed to it to make an allocation or deallocation. Linux makes a conscious effort to allocate logically, "page aligned" contiguous address space. Such page aligned logical spaces are called regions in the memory.

Linux organizes these regions to form a binary tree structure for fast access. In addition to the above logical view the Linux kernel maintains the physical view ie maps the hardware page table entries that determine the location of the logical page in the exact location on a disk. The process address space may have private or shared pages. Changes made to a page require that locality is preserved for a process by maintaining a copy-on-write when the pages are private to the process where as these have to be visible when they are shared.

A process, when first created following a fork system call, finds its allocation with a new entry in the page table – with inherited entries from the parent. For any page which is shared amongst the processes (like parent and child), a reference count is maintained.

Linux has a far more efficient page swapping algorithm than Unix – it uses a second chance algorithm dependent on the usage pattern. The manner it manifests it self is that a page gets a few chances of survival before it is considered to be no longer useful. Frequently used pages get a higher age value and a reduction in usage brings the age closer to zero – finally leading to its exit.

**The Kernel Virtual Memory:** Kernel also maintains for each process a certain amount of "kernel virtual memory" – the page table entries for these are marked "protected". The kernel virtual memory is split into two regions. First there is a static region which has the core of the kernel and page table references for all the normally allocated pages that can not be modified. The second region is dynamic - page table entries created here may point anywhere and can be modified.

**Loading, Linking and Execution:** For a process the execution mode is entered following an *exec* system call. This may result in completely rewriting the previous execution context – this, however, requires that the calling process is entitled an access to the called code. Once the check is through the loading of the code is initiated. Older versions of Linux used to load binary files in the a.out format. The current version also loads binary files in ELF format. The ELF format is flexible as it permits adding additional information for debugging etc. A process can be executed when all the needed library routines have also been linked to form an executable module. Linux supports dynamic linking. The dynamic linking is achieved in two stages:

1. First the linking process downloads a very small statically linked function – whose task is to read the list of library functions which are to be dynamically linked.
2. Next the dynamic linking follows - resolving all symbolic references to get a loadable executable.

## Linux File Systems

**Introduction:**

Linux retains most fundamentals of the Unix file systems. While most Linux systems retain Minix file systems as well, the more commonly used file systems are VFS and ext2FS which stand for virtual file system and extended file systems. We shall also examine some details of proc file system and motivation for its presence in Linux file systems.

As in other UNIXES in Linux the files are mounted in one huge tree rooted at /. The file may actually be on different drives on the same or on remotely networked machines. Unlike windows, and like unixes, Linux does not have drive numbers like A: B: C: etc.

**The *mount* operation**: The unixes have a notion of *mount* operation. The mount operation is used to attach a filesystem to an existing filesystem on a hard disk or any other block oriented device. The idea is to attach the filesystem within the file hierarchy at a specified mount point. The mount point is defined by the path name for an identified directory. If that mount point has contents before the mount operation they are hidden till the file system is un-mounted. The un-mount requires issuance of *umount* command.

Linux supports multiple filesystems. These include ext, ext2, xia, minix, umsdos, msdos, vfat, proc, smb, ncp, iso9660,sysv, hpfs, affs and ufs etc. More file systems will be supported in future versions of LINUX. All block capable devices like floppy drives, IDE hard disks etc. can run as a filesystem. The “look and feel” of the files is the same regardless of the type of underlying block media. The Linux filesystems treat nearly all media as if they are linear collection of blocks. It is the task of the device driver to translate the file system calls into appropriate cylinder head number etc. if needed. A single disk partition or the entire disk (if there are no partitions) can have only one filesystem. That is, you cannot have a half the file partition running EXT2 and the remaining half running FAT32. The minimum granularity of a file system is a hard disk partition.

On the whole the EXT2 filesystem is the most successful file system. It is also now a part of the more popular Linux Distributions. Linux originally came with the Minix filesystem which was quite primitive and 'academic' in nature. To improve the situation a new file system was designed for Linux in 1992 called the Exteneded File System or the EXT file system. Mr Remy Card (**Rémy Card, Laboratoire MASI--Institut Blaise Pascal, E-Mail: card@masi.ibp.fr**) further improved the system to offer the Extended File System -2 or the ext-2 file system. This was an important addition to Linux that was added along with the virtual file system which permitted Linux to interoperate with different filesystems.

**Description:**

Basic File Systems concepts:

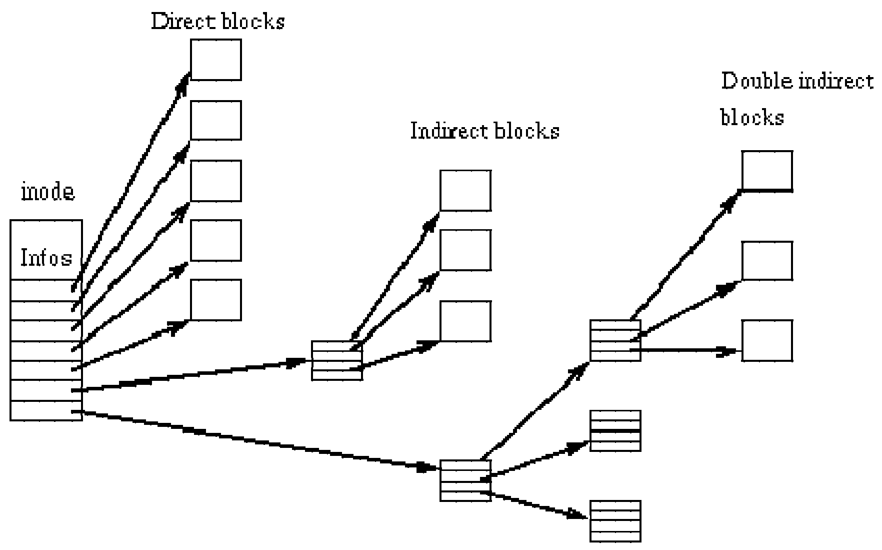
Every Linux file system implements the basic set of concepts that have been a part of the Unix filesystem along the lines described in “The Design of the Unix” Book by Maurice Bach. Basically, these concepts are that every file is represented by an inode. Directories are nothing but special files with a list of entries. I/O to devices can be handled by simply reading or writing into special files (Example: To read data from the serial port we can do cat /dev/ttyS0).

Superblock:

Super block contains the meta-data for the entire filesystem.

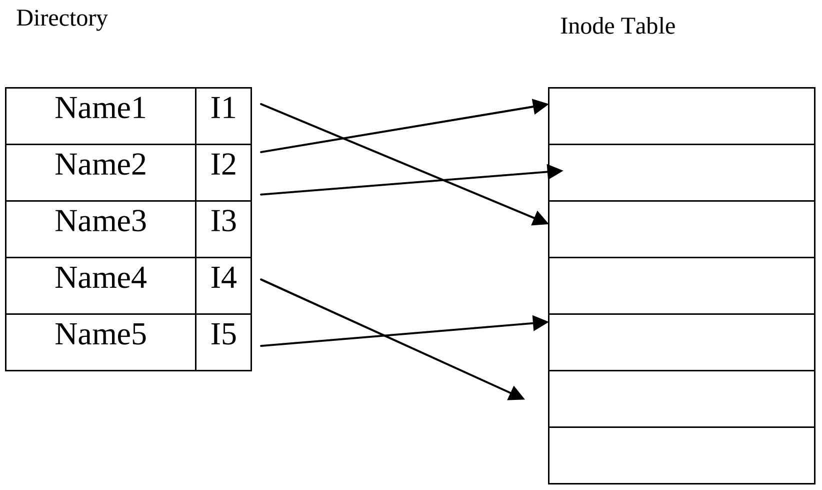
Inodes:

Each file is associated with a structure called an inode. Inode stores the attributes of the file which include File type, owner time stamp, size pointers to data blocks etc. Whenever a file is accessed the kernel translates the offset into a block number and then uses the inode to figure out the actual address of the block. This address is then used to read/write to the actual physical block on the disk. The structure of an inode is as shown below in the figure.



Directories:

Directories are implemented as special files. Actually, a directory is nothing but a file containing a list of entries. Each entry contains a file name and a corresponding inode number. Whenever a path is resolved by the kernel it looks up these entries for the corresponding inode number. If the inode number is found it is loaded in the memory and used for further file access.



Links:

UNIX operating systems implement the concept of links. Basically there are two types of links: Hard links and soft links. Hard link is just another entry in directory structure pointing to the same inode number as the file name it is linked to. The link count on the pointed inode is incremented. If a hard link is deleted the link count is decremented. If the link count becomes zero the inode is deallocated if the linkcount becoms zero. It is impossible to have cross file systems hard links.

Soft links are just files which contain the name of the file they are pointing to. Whenever the kernel encounters a soft link in a path it replaces the soft-link with it contents and restarts the path resolution. With soft links it is possible to have cross file system links. Softlinks that are not linked to absolute paths can lead to havoc in some cases. Softlinks also degrade system performance.

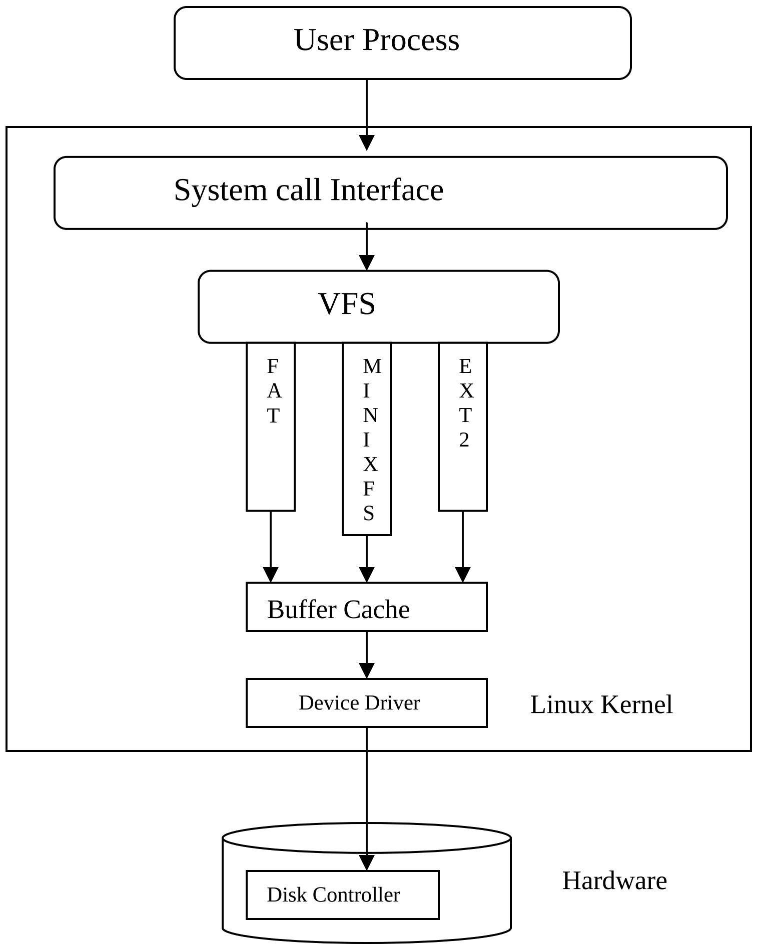
***Device specific files:***

UNIX operating systems enable access to devices using special files. These file do not take up any space but are actually used to connect the device to the correct device driver. The device driver is located based on the major number associated with the device file. The minor number is passed to the device driver as an argument. Linux kernel 2.4 introduced a new file system for accessing device files called as the device file system. (Look at the section on device drivers)

**The Virtual File system:**

When the Linux Kernel has to access a filesystem it uses a filesystem type independent interface, which allows the system to carry out operations on a File System without knowing its construction or type. Since the kernel is independent of File System type or construction, it is flexible enough to accommodate future File Systems as and when they become available.

Virtual File System is an interface providing a clearly defined link between the operating system kernel and the different File Systems.



**The VFS Structure and file management in VFS**:

For management of files, VFS employs an underlying definition for three kinds of objects:

1. inode object
2. file object
3. file system object

Associated with each type of object is a function table which contains the operations that can be performed. The function table basically maintains the addresses of the operational routines. The file objects and inode objects maintain all the access mechanism for each file's access. To access an inode object the process must obtain a pointer to it from the corresponding file object. The file object maintains from where a certain file is currently being read or written to ensure sequential IO. File objects usually belong to a single process. The inode object maintains such information as the owner, time of file creation and modification.

The VFS knows about file-system types supported in the kernel. It uses a table defined during the kernel configuration. Each entry in this table describes filesystem type: it contains the name of the filesystem type and a pointer to a function called during the mount operation. When a file-system is to be mounted, the appropriate mount function is called. This function is responsible for reading the super-block from the disk, initializing its internal variables, and returning a mounted file-system descriptor to the VFS. The VFS functions can use this descriptor to access the physical file-system routines subsequently. A mounted file-system descriptor contains several kinds of data: information that is common to every file-system type, pointers to functions provided by the physical file-system kernel code, and private data maintained by the physical file-system code. The function pointers contained in the file-system descriptors allow the VFS to access the file-system internal routines. Two other types of descriptors are used by the VFS: an inode descriptor and an open file descriptor. Each descriptor contains information related to files in use and a set of operations provided by the physical file-system code. While the inode descriptor contains pointers to functions that can be used to act on any file (e.g. create, unlink), the file descriptors contains pointer to functions which can only act on open files (e.g. read, write).

**The Second Extended File System (EXT2FS)**

**Standard Ext2fs features:**

This is the most commonly used file system in Linux. In fact, it extends the original Minix FS which had several restrictions – such as file name length being limited to 14 characters and the file system size limited to 64 K etc. The ext2FS permits three levels of indirections to store really large files (as in BSD fast file system). Small files and fragments are stored in 1KB (kilo bytes) blocks. It is possible to support 2KB or 4KB blocks sizes. 1KB is the default size. The Ext2fs supports standard *nix file types: regular files, directories, device special files and symbolic links. Ext2fs is able to manage file systems created on really big partitions. While the original kernel code restricted the maximal file-system size to 2 GB, recent work in the VFS layer have raised this limit to 4 TB. Thus, it is now possible to use big disks without the need of creating many partitions.

Not only does Ext2fs provide long file names it also uses variable length directory entries. The maximal file name size is 255 characters. This limit could be extended to 1012, if needed. Ext2fs reserves some blocks for the super user (root). Normally, 5% of the blocks are reserved. This allows the administrator to recover easily from situations where user processes fill up file systems.

As we had earlier mentioned physical block allocation policy attempts to place logically related blocks physically close so that IO is expedited. This is achieved by having two forms of groups:

1. Block group
2. Cylinder group.

Usually the file allocation is attempted with the block group with the inode of the file in the same block group. Also within a block group physical proximity is attempted. As for the cylinder group, the distribution depends on the way head movement can be optimized.

**Advanced Ext2fs features**

In addition to the standard features of the *NIX file systems ext2fs supports several advanced features.

File attributes allow the users to modify the kernel behavior when acting on a set of files. One can set attributes on a file or on a directory. In the later case, new files created in the directory inherit these attributes. (Examples: Compression Immutability etc)

BSD or System V Release 4 semantics can be selected at mount time. A mount option allows the administrator to choose the file creation semantics. On a file-system mounted with BSD semantics, files are created with the same group id as their parent directory. System V semantics are a bit more complex: if a directory has the setgid bit set, new files inherit the group id of the directory and subdirectories inherit the group id and the setgid bit; in the other case, files and subdirectories are created with the primary group id of the calling process.

BSD-like synchronous updates can be used in Ext2fs. A mount option allows the administrator to request that metadata (inodes, bitmap blocks, indirect blocks and directory blocks) be written synchronously on the disk when they are modified. This can be useful to maintain a strict metadata consistency but this leads to poor performances.

Ext2fs allows the administrator to choose the logical block size when creating the file-system. Block sizes can typically be 1024, 2048 and 4096 bytes.
Ext2fs implements fast symbolic links. A fast symbolic link does not use any data block on the file-system. The target name is not stored in a data block but in the inode itself.
Ext2fs keeps track of the file-system state. A special field in the superblock is used by the kernel code to indicate the status of the file system. When a file-system is mounted in read or write mode, its state is set to ``Not Clean''. Whenever filesystem is unmounted, or re-mounted in read-only mode, its state is reset to: ``Clean''. At boot time, the file-system checker uses this information to decide if a file-system must be checked. The kernel code also records errors in this field. When an inconsistency is detected by the kernel code, the file-system is marked as ``Erroneous''. The file-system checker tests this to force the check of the file-system regardless of its apparently clean state.
Always skipping filesystem checks may sometimes be dangerous, so Ext2fs provides two ways to force checks at regular intervals. A mount counter is maintained in the superblock. Each time the filesystem is mounted in read/write mode, this counter is incremented. When it reaches a maximal value (also recorded in the superblock), the filesystem checker forces the check even if the filesystem is ``Clean''. A last check time and a maximal check interval are also maintained in the superblock. These two fields allow the administrator to request periodical checks. When the maximal check interval has been reached, the checker ignores the filesystem state and forces a filesystem check.
Ext2fs offers tools to tune the filesystem behavior like tune2fs

**Physical Structure:**

The physical structure of Ext2 filesystems has been strongly influenced by the layout of the BSD filesystem .A filesystem is made up of block groups. The physical structure of a filesystem is represented in this table:

| | | | | |
|---|---|---|---|---|
| **Boot Sector** | Block Grp 1 | Block Grp2 | …….. | Block Grp N |

Each block group contains a redundant copy of crucial filesystem control informations (superblock and the filesystem descriptors) and also contains a part of the filesystem (a block bitmap, an inode bitmap, a piece of the inode table, and data blocks). The structure of a block group is represented in this table:

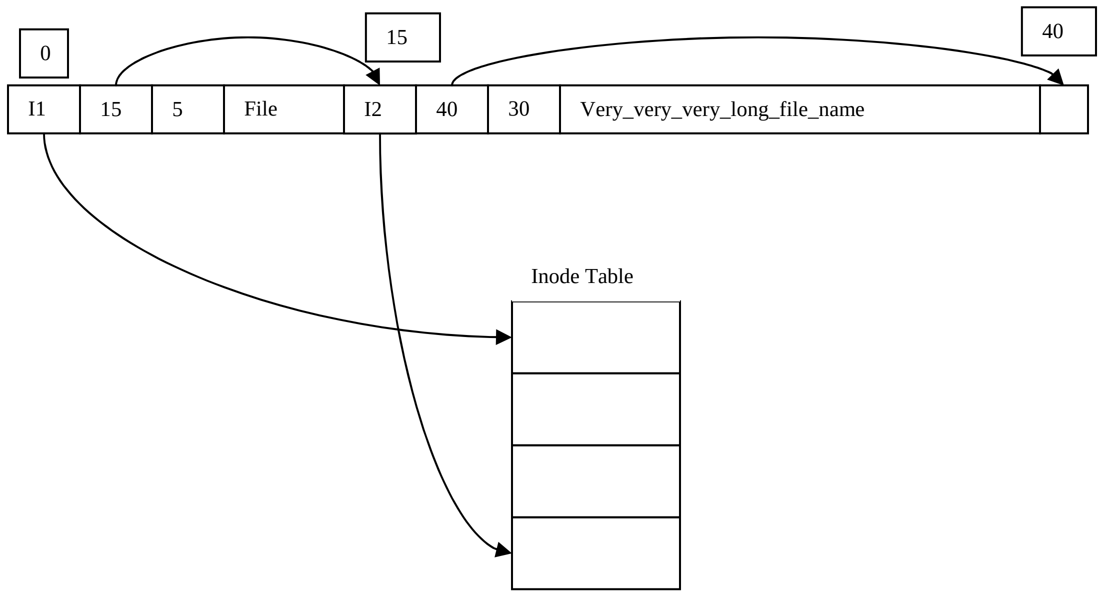
| | | | | | |
|---|---|---|---|---|---|
| Super Block | FS descriptors | Block Bitmap | Inode Bitmap | Inode Table | Data Blocks |

Using block groups is a big factor contributing to the reliability of the file system: since the control structures are replicated in each block group, it is easy to recover from a filesystem where the superblock has been corrupted. This structure also helps to get good performances: by reducing the distance between the inode table and the data blocks, it is possible to reduce the disk head seeks during I/O on files.

In Ext2fs, directories are managed as linked lists of variable length entries. Each entry contains the inode number, the entry length, the file name and its length. By using variable length entries, it is possible to implement long file names without wasting disk space in directories.

As an example, the next table represents the structure of a directory containing three files: File, Very_long_name, and F2. The first entry in the table is inode number; the second entry is the entire entry length: the third field indicates the length of the file name and the last entry is the name of the file itself

| | | | |
|---|---|---|---|
| I1 | 15 | 05 | File |
| I2 | 40 | 30 | Very_very_very_long_file_name |
| I3 | 12 | 03 | |
| | | | |



**The EXT3 file system:** The ext2 file system is in fact a robust and well tested system. Even so some problem areas have been identified with ext2fs. These are mostly with the shutdown fsck (for filesystem health check at the time of shutdown). It takes unduly long to set it right using e2fsck . The solution was to add journaling to the filesystem. One more line about journaling. Another issue with the ext2 file system is its poor capability to scale to very large drives and files. The EXT3 file system which is in some sense an extension of the ext2 filesystem will try to address these shortcomings and also offer many other enhancements.

**THE PROC FILE SYSTEM:**

Proc file system shows the power of the Linux virtual file system. The Proc file system is a special file system which actually displays the present state of the system. In fact we can call it a 'pretend' file system. If one explores the /proc directory one notices that all the files have zero bytes as the file size. Many commands like *ps* actually parse the /proc files to generate their output. Interestingly enough Linux does not have any system call to get process information. It can only be accessed by reading the proc file system. The proc file system has a wealth of information. For example the file /proc/cpuinfo gives a lot of things about the host processor.

A sample output could be as shown below:

```
processor  : 0
vendor_id   : AuthenticAMD
cpu family  : 5
model       : 9
model name  : AMD-K6(tm) 3D+ Processor
stepping    : 1
cpu MHz     : 400.919
cache size  : 256 KB
fdiv_bug    : no
hlt_bug     : no
f00f_bug    : no
coma_bug    : no
fpu     : yes
fpu_exception   : yes
```

```
cpuid level : 1
wp      : yes
flags       : fpu vme de pse tsc msr mce cx8 pge mmx syscall 3dnow k6_mtrr
bogomips    : 799.53
```

/proc also contains, apart from other things, properties of all the processes running on the system at that moment. Each property is grouped together into a directory with a name equal to the PID of the process. Some of the information that can be obtained is shown as follows.

/proc/PID/cmdline
    Command line arguments.

/proc/PID/cpu
    Current and last cpu in which it was executed.

/proc/PID/cwd
    Link to the current working directory.

/proc/PID/environ
    Values of environment variables.

/proc/PID/exe
    Link to the executable of this process.

/proc/PID/fd
    Directory, which contains all file descriptors.

/proc/PID/maps
    Memory maps to executables and library files.

/proc/PID/mem
    Memory held by this process.

/proc/PID/root
    Link to the root directory of this process.

/proc/PID/stat
    Process status.

/proc/PID/statm
    Process memory status information.

/proc/PID/status

Process status in human readable form

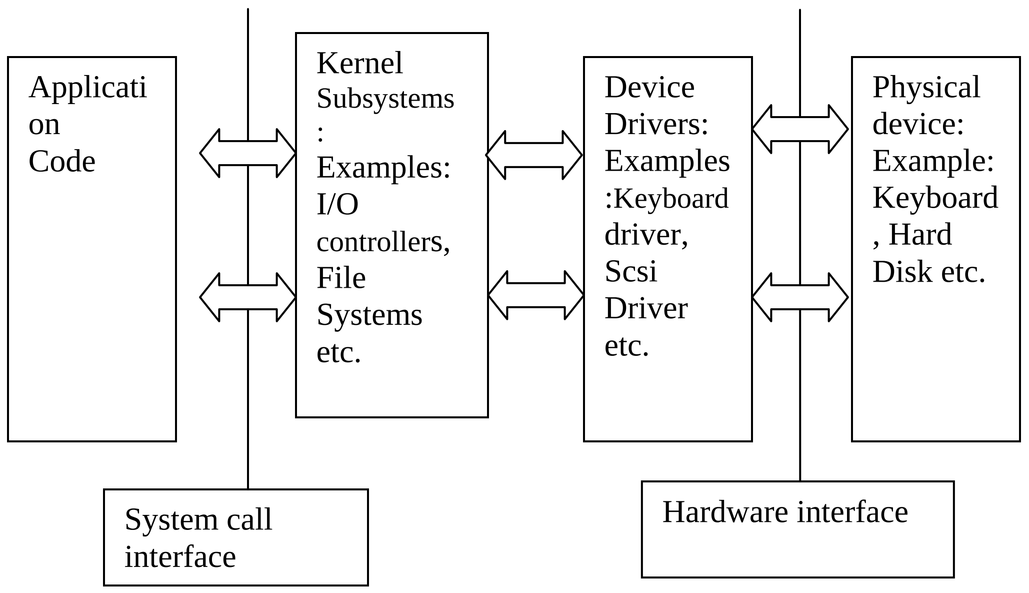
## DEVICE DRIVERS ON LINUX

### Introduction:

Most of the Linux code is independent of the hardware it runs on. Applications are often agnostic to the internals of a hardware device they interact with. They interact with the devices as a black box using operating system defined interfaces. As far as applications are concerned, inside the black box sits a program that exercises a protocol to interact with the device completely. This program interacts with the device at a very low level and abstracts away all the oddities and peculiarities of the underlying hardware to the invoking application. Obviously every device has a different device driver. The demand for device drivers is increasing as more and more devices are being introduced and the old ones become obsolete.

In the context of Linux as an open source OS, device drivers are in great demand. There are two principal drivers behind this. Firstly, many hardware manufacturers do not ship a Linux driver so it is left for someone from the open source community to implement a driver. Second reason is the large proliferation of Linux in the embedded system market. Some believe that Linux today is number one choice for embedded system development work. Embedded devices have special devices attached to them that require specialized drivers. An example could be a microwave oven running Linux and having a special device driver to control its turntable motor.

In Linux the device driver can be linked into the kernel at compile time. This implies that the driver is now a part of the kernel and it is always loaded. The device driver can also be linked into the kernel dynamically at runtime as a pluggable module.


Application Code
Kernel Subsystems: Examples: I/O controllers, File Systems etc.
Device Drivers: Examples :Keyboard driver, Scsi Driver etc.
Physical device: Example: Keyboard, Hard Disk etc.
System call interface
Hardware interface

Almost every system call eventually maps to a physical device. With the exception of the processor, memory and a few other entities, all device control operations are performed by code that is specific to the device. This code as we know is called the device driver. Kernel must have device drivers for all the peripherals that are present in the system right from the keyboard to the hard disk etc.

**Device classes**:

**Char devices**:

These devices have a stream oriented nature where data is accessed as a stream of bytes example serial ports. The drivers that are written for these devices are usually called "char device drivers". These devices are accessed using the normal file system. Usually they are mounted in the /dev directory. If ls –al command is typed on the command prompt in the /dev directory these devices appear with a 'c' in the first column.

Example:

```
crw-rw-rw-    1 root     tty       2, 176 Apr 11  2002 ptya0
crw-rw-rw-    1 root     tty       2, 177 Apr 11  2002 ptya1
crw-rw-rw-    1 root     tty       2, 178 Apr 11  2002 ptya2
crw-rw-rw-    1 root     tty       2, 179 Apr 11  2002 ptya3
```

**Block devices:**

These devices have a ‘block’ oriented nature where data is provided by the devices in blocks. The drivers that are written for these devices are usually called as block device drivers. Classic example of a block device is the hard disk. These devices are accessed using the normal file system. Usually they are mounted in the /dev directory. If a ls –al command is typed on the command prompt in the /dev directory these devices appear with a ‘b’ in the first column.

Example:

```
brw-rw----    1 root     disk      29,   0 Apr 11  2002 aztcd
brw-rw----    1 root     disk      41,   0 Apr 11  2002 bpcd
brw-rw----    1 root     floppy     2,   0 Apr 11  2002 fd0
```

**Network devices**:

These devices handle the network interface to the system. These devices are not accessed via the file system. Usually the kernel handles these devices by providing special names to the network interfaces e.g. eth0 etc.

Note that Linux permits a lot of experimentation with regards to checking out new device drivers. One need to learn to load, unload and recompile to check out the efficacy of any newly introduced device driver. The cycle of testing is beyond the scope of discussion here.

**Major/minor numbers:**

Most devices are accessed through nodes in the file system. These nodes are called special files or device files or simply nodes of the file system tree. These names are usually mounted in the /dev/ directory.

If a ls –al command is issued in this directory we can see two comma separated numbers that appear where usually the file size is mentioned. The first number (from left side) is called the device major number and the second number is called the device minor number.

Example: `crw-rw-rw-    1 root     tty        2, 176 Apr 11 2002 ptya0`

Here the major number is 2 and the minor number is 176

The major number is used by the kernel to locate the device driver for that device. It is an index into a static array of the device driver entry points (function pointers). The minor number is passed to the driver as an argument and the kernel does not bother about it. The minor number may be used by the device driver to distinguish between the different types of devices of the same type it supports. It is left to the device driver, what it does with the minor numbers. For the Linux kernel 2.4 the major and minor numbers are eight bit quantities. So at a given time you can have utmost 256 drivers of a particular type and 256 different types of devices loaded in a system. This value is likely to increase in future releases of the kernel.

Kernel 2.4 has introduced a new (optional) file system to handle device. This file system is called the device file system . In this file system the management of devices is much more simplified. Although it has lot of user visible incompatibilities with the previous file system, at present device file system is not a standard part of most Linux distributions.

In future, things might change in favour of the device file system. Here it must be mentioned that the following discussion is far from complete. There is no substitute for looking at the actual source code. The following section will mainly help the reader to know what to *grep* for in the source code.

We will now discuss each of the device class drivers that is block, character and network drivers in more detail.

## Character Drivers:

**Driver Registeration/Uregisteration:**

We register a device driver with the Linux kernel by invoking a routine (<Linux/fs.h>)

int register_chrdev(unsigned int major, const char * name, struct file_operations * fops);

Here the major argument is the major number associated with the device. Name signifies the device driver as it will appear in the /proc/devices once it is successfully registered. The fops is a pointer to the structure containing function pointers to the devices' functionalities. We will discuss fops in detail later.

Now the question arises: how do we assign a major number to our driver:

**Assigning major numbers:**

Some numbers are permanently allocated to some common devices. The reader may like to explore: /Documentation/devices.txt in the source tree. So if we are writing device drivers for these devices we simply use these major numbers.

If that is not the case then we can use major numbers that are allocated for experimental usage. Major numbers in the range 60-63, 120-127, 240-254 are for experimental usage.

But how do we know that a major number is not already used especially when we are shipping a driver to some other computer.

By far the best approach is to dynamically assign the major number. The idea is to get a free major number by looking at the present state of the system and then assigning it to our driver. If the register_chrdev function is invoked with a zero in the major number field, the function, if it registers the driver successfully, returns the major number allocated to it. What it does is that it searches the system for an unused major number, assigns it to the driver and then returns it. The story does not end here. To access our device we need to add our device to the file system tree. That is, we need to do *mknod* for the device into the tree. For that we need to know the major number for the driver. For a statically assigned major number that is not a problem. Just use that major number you assigned to the device. But for a dynamically assigned number how do we get the major number? The answer is: parse the /proc/devices file and find out the major number assigned to our device. A script can also be written to do the job.

Removing a driver from the system is easy. We invoke the unregister_chrdev(unsigned int major, const char * name);

**Important Data Structure:**

**The file Structure<linux/fs.h>:**

Every Linux open file has a corresponding file structure associated with it. Whenever a method of the device driver is invoked the kernel will pass the associated file structure to the method. The method can then use the contents of this structure to do its job. We list down some important fields of this structure.

mode_t f_mode;

This field indicates the mode of the file i.e for read write or both etc.

loff_t f_pos;

The current offset in the file.

unsigned int f_flags;

This fields contains the flags for driver access for example synchronous access (blocking) or asynchronus(non blocking) access etc.

struct file_operations * fops;

This structure contains the entry points for the methods that device driver supports. This is an important structure we will look at it in more detail in the later sections.

void * private_data;

This pointer can be allocated memory by the device driver for its own personal use. Like for maintaining states of the driver across different function calls.

sruct dentry * f_dentry;

The directory entry associated with the file.

Etc.

**The file operations structure(fops):<linux/fs.h>**

This is the most important structure as far as device driver writer are concerned. It contains pointers to the driver functions. The file structure discussed in the previous section contains a pointer to the fops structure. The file (device) is the object and fops contains the methods that act on this object. We can see here object oriented approach in the Linux Kernel.

Before we look at the members of the fops structure it will be useful if we look at taggd structure initialization:

**Tagged structure initializations:**

The fops structure has been expanding with every kernel release. This can lead to compatibility problems of the driver across different kernel versions.

This problem is solved by using tagged structure initialization. Tagged structure initialization is an extension of ANSI C by GNU. It allows initialization of structure by name tags rather than positional initialization as in standard C.

Example:

```
struct fops myfops={
……………………..
………………….
open : myopen;
close : myclose:
…………..
…………
}
```

The intilization can now be oblivious of the change in the structure (Provided obviously that the fields have not been removed).

Pointers to functions that are implemented by the driver are stored in the fops structure. Methods that are not implemented are made NULL.

Now we look at some of the members of the fops structure:

loff_t (*llseek) (struct file *,loff_t);

/* This method can be used to change the present offset in a file. */

ssize_t (*read) (struct file*,char *,size_t,loff_t *);

/* Read data from a device.*/

ssize_t(*write) (struct file *,const char *,size_t,loff_t *);

/* Write data to the device. */

int (* readdir) (struct file *,void *,fill_dir_t);

/* Reading directories. Useful for file systems.*/

unsigned int (* poll) (struct file *,struct poll_table_struct *);

/* Used to check the state of the device. */

int (*ioctl)(struct inode *,struct file *,unsigned int,unsigned long);

/* The ioctl is used to issue device specific calls(example setting the baud rate of the serial port). */

int (*mmap) (struct file *,struct vm_area_struct *);

/* Map to primary memory */

int (* open) (struct inode *,struct file *);

/ * Open device.*/

int ( *flush) (struct file *) ;

/* flush the device*/

int (*release) (struct inode *,struct file *);

/* Release the file structure */

int(*fsync) (struct inode *,struct dentry *);

/* Flush any pending data to the device. */

Etc.

**Advance Char Driver Operations:**

Although most of the following discussion is valid to character as well as network and block drivers, the actual implementation of these features is explained with respect to char drivers.

**Blocking and non-blocking operations**:

Device drivers usually interact with hardware devices that are several orders of time slower than the processor. Typically if a modern PC processor takes a second to process a byte of data from a keyboard, the keyboard takes several thousand years to produce a single byte of data. It will be very foolish to keep the processor waiting for data to arrive from a hardware device. It could have severe impact on the overall system performance and throughput. Another cause that can lead to delays in accessing devices, which has nothing to do with the device characteristics, is the policy in accessing the device. There might be cases where device may be blocked by other drivers . For a device driver writer it is of paramount importance that the processor is freed to perform other tasks when the device is not ready.

We can achieve this by the following ways.

One way is blocking or the synchronous driver access. In this way of access we cause the invoking process to sleep till the data arrives. The CPU is then available for other processes in the system. The process is then awakened when the device is ready.

Another method is in which the driver returns immediately whether the device is ready or not allowing the application to poll the device.

Also the driver can be provided asynchronous methods for indicating to the application when the data is available.

Let us briefly look at the Linux kernel 2.4 mechanisms to achieve this.

There is a flag called O_NONBLOCK flag in filp->f_flags ( <linux/fcntl.h> ).

If this flag is set it implies that the driver is being used with non-blocking access. This flag is cleared by default. This flag is examined by the driver to implement the correct semantics.

**Blocking IO**:

There are several ways to cause a process to sleep in Linux 2.4. All of them will use the same basic data structure, the wait queue (wait_queue_head_t). This queue maintains a linked list of processes that are waiting for an event.

A wait queue is declared and initialized as follows:

```
wait_queue_head_t my_queue; /* declaration */
init_waitqueue_head(&my_queue) /* initialization */
```

/* 2.4 kernel requires you to intialize the wait queue, although some earlier versions of the kernel did not */

The process can be made to sleep by calling any of the following:

sleep_on(wait_queue_head_t * queue);

/* Puts the process to sleep on this queue. */

/* This routine puts the process into non-interruptible sleep */

/* this a dangerous sleep since the process may end up sleeping forever */

interruptible_sleep_on(wait_queue_head_t * queue)

/* same as sleep_on with the exception that the process can be awoken by a signal */

sleep_on_timeout(wait_queue_head_t * queue,long timeout)

/* same as sleep_on except that the process will be awakened when a timeout happens. The timeout parameter is measured in jiffies */

interruptible_sleep_on_timeout(wait_queue_head_t * queue,long timeout)

/* same as interruptible_sleep_on except that the process will be awakened when a timeout happens. The timeout parameter is measured in jiffies */

void wait_event(wait_queue_head_t * queue,int condition)

int wait_event_interruptible(wait_queue_head_t * queue, int condition)

/* sleep until the condition evaluates to true that is non-zero value */

/* preferred way to sleep */

If a driver puts a process to sleep there is usually some other part of the driver that awakens it, typically it is the interrupt service routine.

One more important point is that if a process is in interruptible sleep it might wake up even on a signal even if the event it was waiting on, has not occurred. The driver must in this case put a process in sleep in a loop checking for the event as a condition in the loop.

The kernel routines that are available to wake up a process are as follows:

wake_up(wait_queue_head_t * queue)

/* Wake proccess in the queue */

wake_up_interruptible(wait_queue_head_t * queue)

/* wake process in the queue that are sleeping on interruptible sleep in the queue rest of the procccess are left undisturbed */

wake_up_sync(wait_queue_head_t_ * queue)

wake_up_interruptible_sync(wait_queue_head_t_ * queue)

/* The normal wake up calls can cause an immediate reschedule of the processor */

/* these calls will only cause the process to go into runnable state without rescheduling the CPU */

**Non Blocking IO**:

If O_NONBLOCK flag is set then driver does not block even if data is not available for the call to complete. The normal semantics for a non-blocking IO is to return -EAGAIN which really tells the invoking application to try again. Usually devices that are using non-blocking access to devices will use the poll system call to find out if the device is ready with the data. This is also very useful for an application that is accessing multiple devices without blocking.

Polling methods: Linux provides the applications 'poll' and 'select' system calls to check if the device is ready without blocking. (There are two system calls offering the same functionality for historical reasons. These calls were implemented in UNIX at nearly same time by two different distributions: BSD Unix (select)   System 5(poll))

Both the calls have the following prototype:

unsigned int (*poll)(struct file * ,poll_table *);

The poll method returns a bit mask describing what operations can be performed on the device without blocking.

**Asynchronous Notification**:

Linux provides a mechanism by which a drive can asynchronously notify the application if data arrives. Basically a driver can signal a process when the data arrives. User processes have to execute two steps to enable asynchronous notification from a device.

1. The process invokes the F_SETOWN command using the fcntl system call, the process ID of the process is saved in filp->f_owner. This is the step needed basically for the kernel to route the signal to the correct process.
2. The asynchronous notification is then enabled by setting the FASYNC flag in the device by means of F_SETFEL fcntl command.

After these two steps have been successfully executed the user process can request the delivery of a SIGIO signal whenever data arrives.

**Interrupt Handling in LINUX 2.4**

The Linux kernel has a single entry point for all the interrupts. The number of interrupt lines is platform dependent. The earlier X86 processors had just 16 interrupt lines. But now this is no longer true. The current processors have much more than that. Moreover new hardware comes with programmable interrupt controllers that can be programmed among other things to distribute interrupts in an intelligent and a programmable way to different processors for a multi-processors system. Fortunately the device driver writer does not have to bother too much about the underlying hardware, since the Linux kernel nicely abstracts it. For the Intel x86 architecture the Linux kernel still uses only 16 lines.

The Linux kernel handles all the interrupts in the same manner. On the receipt of an interrupt the kernel first acknowledges the interrupt. Then it looks for registered handlers for that interrupt. If a handler is registered, it is invoked. The device driver has to register a handler for the interrupts caused by the device.

The following API is used to register an interrupt handler.

<linux/sched.h>

int request_irq(unsigned int irq, void ( * interruptHandler ) (int, void *,struct pt_regs *), unisgned long flags, const char * dev_name, void * dev_id);

/* irq -> The interrupt number being requested */

/* interruptHandler -> function pointer to the interrupt handler */

/* flags -> bitwise orable flags one of

SA_INTERRUPT implies 'fast handler' which basically means that the interrupt handler finishes its job quickly and can be run in the interrupt context with interrupts disabled.

SA_SHIRQ implies that the interrupt is shared

SA_SAMPLE_RANDOM implies that the interrupt can be used to increase the entropy of the system */

/* dev_name ->A pointer to a string which will appear in /proc/interrupts to signify the owner of the interrupt */

/* dev_id-> A unique identifier signifying which device is interrupting. Is mostly used when the interrupt line is shared. Otherwise kept NULL*/

/* the interrupt can be freeed implying that the handler associated with it can be removed */void free_irq(unsigned int irq,void * dev_id);

/* by calling the following function. Here the meaning of the parameters is the same as in request_irq

*/Now the question that arises: how do we know which interrupt line our device is going to use. Some device use predefined fixed interrupt lines. So they can be used. Some

devices have jumper settings on them that let you decide which interrupt line the device will use. There are devices (like device complying to the PCI standard) that can on request tell which interrupt line they are going to use. But there are devices for which we cannot tell before hand which interrupt number they are going to use. For such device we need the driver to probe the IRQ number. Basically what is done is the device is asked to interrupt and then we look at all the free interrupt lines to figure out which line got interrupted. This is not a clean method and ideally a device should itself announce which interrupt it wants to use. (Like PCI).

The kernel provides helper functions for probing of interrupts( <linux/interrupt.h> probe_irq_on , probe_irq_off) or the drive can do manual probing for interrupts.

**Top Half And Bottom Half Processing:**

One problem with interrupt processing is that some interrupt service routines are rather long and take a long time to process. These can then cause interrupts to be disabled for a long time degrading system responsiveness and performance. The method used in Linux (and in many other systems) to solve this problem is to split up the interrupt handler into two parts : The “top half” and the “bottom half”. The top half is what is actually invoked at the interrupt context. It will just do the minimum required processing and then wake up the bottom half. The top half is kept very short and fast. The bottom half then does the time consuming processing at a safer time.

Earlier Linux had a predefined fixed number of bottom halves (32 of them) for use by the driver. But now the (Kernel 2.3 and later) the kernel uses “tasklets” to do the bottom half processing. Tasklet is a special function that may be scheduled to run in interrupt context, at a system determined safe time. A tasklet may be scheduled to run multiple times, but it only runs once. An interesting consequence of this is that a top half may be executed several times before a bottom half gets a chance to execute. Now since only a single tasklet will be run, the tasklet should be able to handle such a situation. The top half should keep a count of the number of interrupts that have happened. The tasklet can use this count to figure out what to do.

```
/* Takelets are declared using the following macro */
DECLARE_TASKLET(taskLetName,Function,Data);
/* taskLetName -> Name of the tasklet */
/* the function to be run as a tasklet. The function has the following prototype */
/* void Function(usigned long ) */
/* Data is the argument to be passed to the function */
/* the tasklet can be scheduled using this function */
tasklet_schedule(&takletName)
```

Interprocess Communication in Linux:

Again there is considerable similarity with Unix. For example, in Linux, *signals* may be utilized for communication between parent and child processes. Processes may synchronize using *wait* instruction. Processes may communicate using *pipe* mechanism. Processes may use *shared memory mechanism* for communication.

Probably need some more points on this topic on IPC and the different mechanisms available. I found a good url "http://cne.gmu.edu/modules/ipc/map.html".

(Show these using animation)

Let us examine how the communication is done in the networked environment. The networking features in Linux are implemented in three layers:

1. Socket interface
2. Protocol drivers
3. Network drivers.

Typically a user applications' first I/F is the socket. The socket definition is similar to BSD 4.3 Unix which provides a general purpose interconnection framework. The protocol layer supports what is often referred to as protocol stack. The data may come from either an application or from a network driver. The protocol layer manages routing, error reporting, reliable retransmission of data

For networking the most important support is the IP suite which guides in routing of packets between hosts. On top of the routing are built higher layers like UDP or TCP. The routing is actually done by IP driver. The IP driver also helps in disassembly / assembly of the packets. The routing gets done in two ways:

1. By using recent cached routing decisions
2. By using a table which acts as a persistent forwarding base

Generally the packets are stored in a buffer and have a tag to identify the protocol that need to be used. After the selection of the appropriate protocol the IP driver then hands it over to the network device driver to manage the packet movement.

As for security, the firewall management maintains several chains – with each chain having its own set of rules of filtering the packets.

Real Time Linux:

Large number of projects both open source and commercial have been dedicated to get real time functionality from the Linux kernel. Some of the projects are listed below

**Commercial distributions:**

FSMLabs: RTLinuxPro

Lineo Solutions: uLinux

LynuxWorks: BlueCat RT

MontaVista Software: Real-Time Solutions for Linux

Concurrent: RedHawk

REDSonic: REDICE-Linux

**Open source distributions:**

ADEOS –

ART Linux

KURT -- The KU Real-Time Linux

Linux/RK

QLinuxRealTimeLinux.org

RED-Linux

RTAI

RTLinux

## Linux Installation

Amongst various flavors of UNIX, Linux is currently the most popular OS. Linux is also part of the GNU movement which believes in free software distribution. A large community of programmers subscribe to it. Linux came about mainly through the efforts of Linus Torvalds from Finland who wanted a UNIX environment on his PC while he was a university student. He drew inspiration from Prof. Andrew Tanenbaum of University of Amsterdam, who had earlier designed a small OS called Minix. Minix was primarily used as a teaching tool with its code made widely available and distributed. Minix code could be modified and its capability extended. Linus Torvalds not only designed a PC-based Unix for his personal use, but also freely distributed it. Presently, there is a very large Linux community worldwide. Every major university, or urban centre, has a Linux group. Linux found ready acceptance and the spirit of free distribution has attracted many willing voluntary contributors. Now a days Linux community regulates itself by having all contributions evaluated to ensure quality and to take care of compatibility. This helps in ensuring a certain level of acceptance. If you do a Google search you will get a lot of information on Linux. Our immediate concerns here are to help you have your own Linux installation so that you can practice with many of the tools available under the broad category of Unix-based OSs.

## 20.1 The Installation

Linux can be installed on a wide range of machines. The range may span from one's own PDA to a set of machines which cooperate like Google's 4000 node Linux cluster. For now we shall assume that we wish to install it on a PC. Most PCs have a bootable CD player and BIOS. This means in most cases we can use the CD boot and install procedure. Older PC's did not have these features. In that case one was required to use a set of floppies. The first part of this basic guide is about getting the installation program up and running: using either a CD or a set of floppies.

## 20.2 The Installation Program

In this section we describe the Linux installation. The main point in the installation is to select the correct configuration.

Typically Red Hat Linux is installed by booting to the install directory from a CD-ROM. The other options may include the following.

* Booting to install using a floppy disk.
* Using a hard drive partition to hold the installation software.
* Booting from a DOS Command line.
* Booting to an install and installing software using FTP or HTTP protocols.
* Booting to an install and installing software from an NFS-mounted hard drive.

Installing from CD-ROM : Most PCs support booting directly from a CD-ROM drive. Set your PC's BIOS (if required). Now insert the CD-ROM and reboot to the PC to install Red Hat Linux. You should see a boot screen that offers a variety of options for booting .The options typically would be as follows:

* <ENTER> - Start the installation using a Graphical interface
* text - Start the install using a text interface

* nofb - Start the install using a video frame buffer
* expert
* Linux rescue
* Linux dd

At this stage if you press key F2 then it provides a help screen for the text-based installation. Type the word text at the boot prompt and press Enter to continue.

You shall be asked to select a language. So select a language of your choice. Highlight OK button and press Enter. You will then be asked to select a keyboard for install. So highlight OK Button and press Enter after selecting a keyboard. You shall be next asked to select a pointing device, select a suitable mouse and press OK.

Next you will be asked: Select the type of installation from?

* Workstation
* Server
* Laptop
* Custom
* Upgrade an existing system

Select the suitable option, for example, select server install and press Enter. Next you will choose a partitioning scheme. The choices include the following:

* Auto Partition
* Disk Druid
* Fdisk

The *Auto Partition* will the format hard drive according to the type of selected installation. It will automatically configure the partitions for use with Linux. The Disk Druid will launch a graphical editor listing the free spaces available. The *Fdisk* option offers an ability to create nearly 60 different types of partitions.

On clicking Disk Druid, you will get an option of creating new partitions if you are using a new hard drive. If you are using an old hard disk the partitions are recognized. Create the appropriate partitions or use existing ones as the case may be. Finally, press OK to continue.

Red Hat Linux requires a minimum of two partitions. One is a swap partition and the other a root(/) partition. The swap partition should be more than twice as large as the installed amount of memory. Other partitions may be */remote* and */home*. These can be created after the installation as well.

You will now be as asked to select a boot-loader for booting Linux. The choice of not using a boot-loader is also available. The options available are GRUB and LILO. Select the appropriate boot loader and press OK. Grub and Lilo are typically installed in the MBR of the first IDE hard drive in the PC. You will now be asked for to choose kernel parameters for booting Linux. Enter the arguments in the dialog box or use the OK Button to continue.

If for some reason we cannot arrive at dual booting automatically, then add this code at the end of the file */etc/boot/grud/grub.conf* file

```
title Windows
        rootnoverify(hd0,0)
        chainloader +1
        makeactive
```

You can now configure a dual boot system, if required by configuring the boot-loader. When finished click OK and you will be asked to select a firewall configuration. Use a security level from

* High
* Medium
* None

After this you will have to set the incoming service requests followed by a time-zone selection dialog box. Select the appropriate settings and press OK to continue.

You will now be prompted to enter a user-id and password. The password will not be echoed onto the screen. Now is the time to create user accounts. Each has home directory home usually under */home/usr* directory.

Next you have to select packages you want to install. Use the spacebar to select the various groups of software packages. The size of the installed software will dynamically reflect the choices. Use the select individual package item to choose the individual software packages. The installer will now start installing the packages selected from the CD-ROM drive onto the new Linux partitions.

At the end of the installation you will get an option of creating a boot-disk for later use. You can create the boot disk later using the *mkbootdisk* command.

After this, your installation is done. Press OK and Red Hat Linux will eject the CD ROM and reboot. After rebooting you will be able to log onto a Linux session. To shutdown your computer use the *shutdown -h now* command.

Usually most distributions allow you to Test the set-up. It helps to see if it works. The auto detection (like in Red Hat) takes care of most of the cards and monitor types.

### 20.2.1 Finishing the installation

With the above steps, we should have installed a good working Linux machine. The install program will usually prompt to take out all boot-disks, etc. and the machine will be rebooted (sometimes you may have to reboot). You will see the Linux loader coming up. It is also known as LILO. Newer versions or distributions like Mandrake come up with their own LILO's. RedHat 7.X comes with a graphical screen and menu for startup. Anyway, one may see options like Linux and/or DOS or Windows . Normally we fill in these names during the installations. Another popular boot-loader called GRUB has become the default for RedHat.