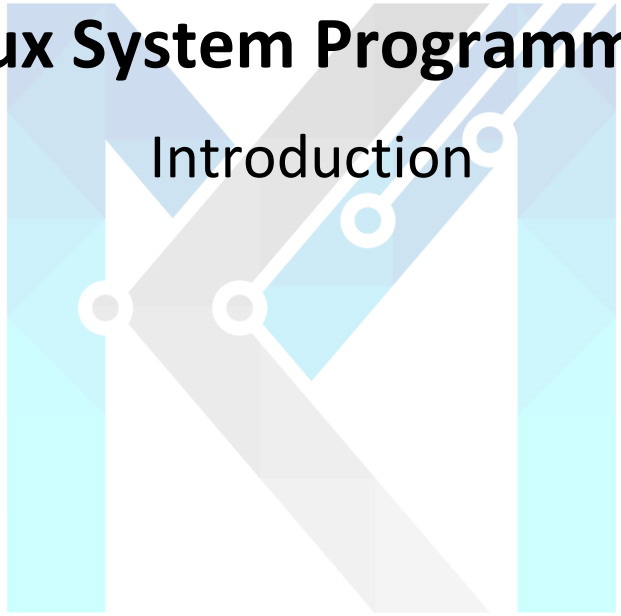
KERNEL MASTERS

Linux System Programming

Introduction



Authored and Compiled By: **Boddu Kishore Kumar**

Email: kishore@kernelmasters.org

Reach us online: www.kernelmasters.org

Contact: 9949062828

Important Notice

2012-2022 Kishore Kumar Boddu
Kernel Masters, Hyderabad - INDIA.

1. Introduction

The Linux operating system was written from the ground-up by Linus Torvalds sometime in 1991. Linus originally wrote the Linux O.S for the Intel 80386. Subsequently, a handful of volunteers worked and ported the Operating System to a wider range of hardware. Linus continues to manage the kernel development process and the CVS repository along with a dedicated team of highly skilled and talented developers such as Alan Cox, Russel Irving etc.

An "Operating System" is not just the kernel. It includes a collection of programs and utilities that make it "useful" to an end-user. Though we normally discuss the Linux kernel as if it's a separate entity, what makes it useful ultimately is the shell and other critical components such as a set of language compilers, debuggers, windowing managers etc. This is where GNU and the Free Software Foundation (FSF) come in. The FSF was founded by Richard Stallman and GNU stands for GNU's Not Unix! GNU provides hundreds of open source programs such as the GCC compiler collection, gdb the GNU debugger, bash etc. along with complete rewrites of all commonly used Unix commands such as ls, cat, more, ps, tar, gzip, dd, cpio etc.

This GNU collection available at no cost, along with the kernel, again available at no cost, is what makes the "package" very attractive. Linus has made the Linux Operating System as part of GNU and Richard Stallman has gone on record stating that one must never use the word "Linux", but "GNULinux" in order to promote GNU's popularity.

2. Features:

Linux comes under the category of what is known as a "monolithic" kernel, as opposed to what are termed "Microkernels". Though recent trends and investigations in Operating Systems design and architecture generally favour Microkernels, Linux was written from scratch as an easy-to-use, but fast alternative to Unix and the kernel is quite bulky and heavy. A non-compressed Linux kernel with "common" features to support commonly-used hardware, is often in the range of a few megabytes in size, which is deemed very bulky as compared to other tiny, compact operating systems. The linux kernel code however can be intelligently split into what is known as a static portion and a dynamically loadable portion. This wasy, the core linux kernel can be minimized to less than a megabyte. Only those features or components that are really required on the system on which the OS is required to run, need to be statically compiled in. The rest, such as file systems support, drivers for a myriad range of network devices etc. can be compiled as loadable kernel modules and loaded on demand.

It must be stressed that linux is not a derivative of any unix source such as BSD or System V. The source code has been completely rewritten from scratch. Linux is deemed to be POSIX compliant which means that the interface to the O.S and the commands used are POSIX compliant, but the actual implementation can be different.
Linux has a limited support for kernel threading. Context switches between kernel threads are less expensive than context switches between ordinary processes. However, unlike Solaris 2.x, kernel threads under linux do not execute user processes since the kernel thread is not a basic "Unit of execution" abstraction.

Linux supports multithreaded applications through its own mechanism of a system call known a clone(). Most other unix variants use kernel threads to implement Light Weight Processes or LWPs.

3. Kernel version-numbering scheme

Linux uses a Major Version, Minor Version and a Release Number to describe the kernel.
If a particular number is X.Y.Z then X is the Major Version, Y is the Minor Version and Z is the Release.
For example, Linux-2.4.20 describes it as Major Version 2, Minor Version 4, and Release 20.
All Odd Numbered Minor Versions are development versions such as 2.3, 2.5 etc. whereas all Even numbered Minor Versions are stable, production versions such as 2.4, 2.6 etc.

4. Kernel components

In describing any Operating System, one normally breaks it up into various components. The Linux Kernel can also be broken up into the following five components or sub-systems:

- Process Management
- Memory Management
- File System
- Device or I/O Management
- Networking

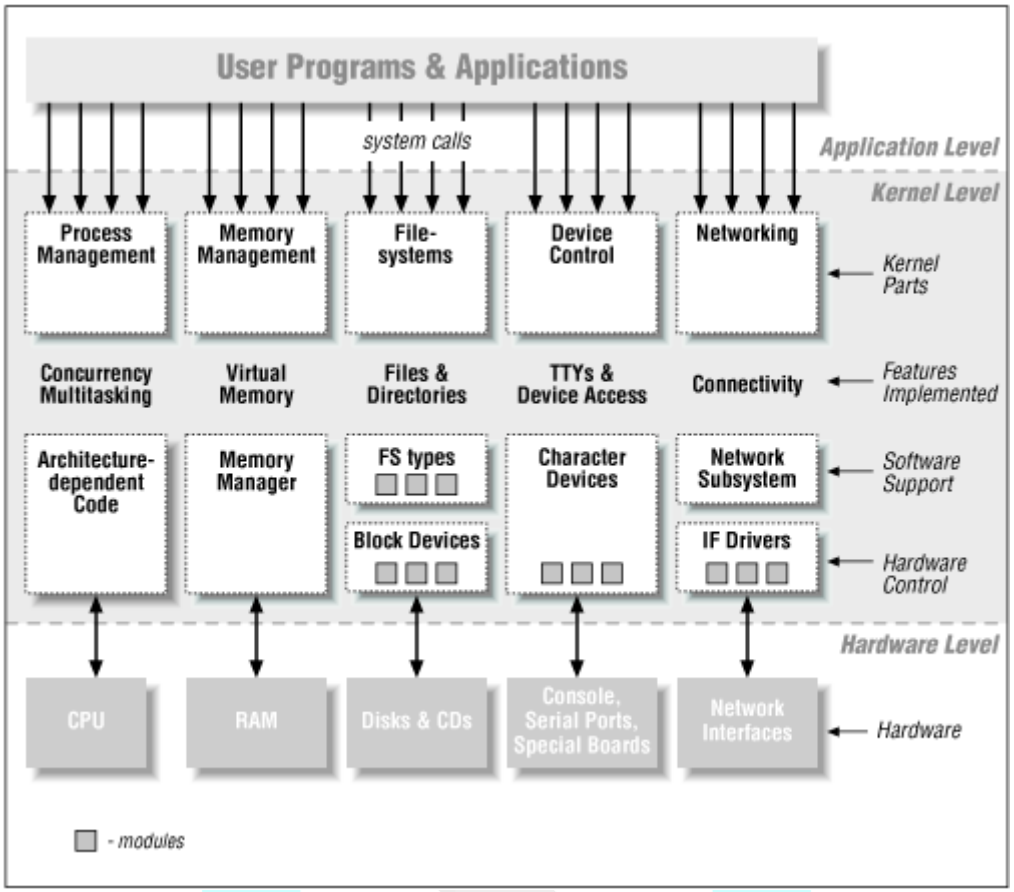
Normally, Networking is NOT a part of the Operating System core. However, its very important to know that under Linux, Networking is a part and parcel of the kernel and is maintained by the kernel developers and not by a third party group, such as for example Xwindows, ALSA, LiViD etc.

The **Process Management** sub-system contains code for multitasking, concurrency, scheduling etc. and also contains a lot of processor-dependent or architecture-dependent code written in assembly.

The **Memory Management** sub-system is quite complex. It handles cache-management and cachehierarchy, kernel memory allocators, heap management, algorithms for the cache/slab allocator and the buddy management system, page table handling and virtual store, page cache, buffer cache, swap cache etc.

The **File System** contains code and drivers for several dozens of supported file systems such as DOS, Windows (fat32 and vfat), NTFS, ReiserFS, ISO-9660, CramFS, ext2, ext3, hpfs, minix etc. A powerful abstration, known as the Virtual Filesystem Switch (VFS) is at the heart of the File System sub-system and provides a neat layer for developing new filesystem support.
I/O Management and Device Control is the layer that allows devices to interact with the kernel through a variety of peripheral buses. Linux has support for all the well-known bus systems such as ISA, PCI, SCSI, IrDA, Firewire (IEEE-1394), USB, PCMCIA, I2C, CAN, Network Devices, Telephony, Wireless devices, Radio, etc.

The **Networking subsystem** provides a complete protocol stack for well-known network protocols such as TCP, UDP, SLIP, PPP, IP, SNMP, Ethernet, TokenRing etc.
The kernel provides a very large range of drivers for well-known NICs (Network Interface Cards) such as 3-COM, Intel, Compaq, Realtek, Lotus, DEC etc.



5. User Versus Kernel Mode

This differentiation is at the heart of any modern operating system. An O.S needs to provide for a secure way to access critical resources such as memory and I/O devices, and the only way that this is possible is when the underlying CPU cooperates with the O.S to enforce a disciplined access.

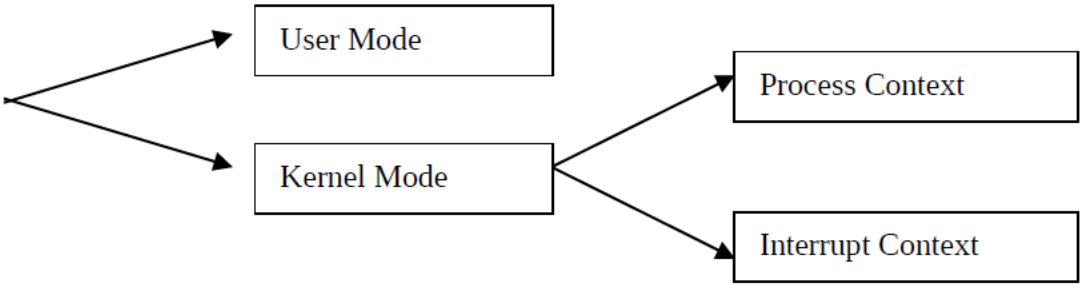
UNIX like systems (which include NT, which incidentally is POSIX compliant), permit the OS to operate in either of two modes – User Mode and Kernel Mode.

When operating under User Mode, the CPU is severely restricted in what it can do in terms of accessing memory, I/O ports etc. However, in Kernel Mode, the CPU is allowed access to critical resources.

Since access to critical resources can only be done when the CPU is in kernel mode, it is sufficient to control the transition between user mode to kernel mode. This transitioning can only be done through an Interrupt, either directly when a hardware device raises it or

through a software interrupt. In the case of an x86 based CPU this interrupt is 0x80 (decimal 128).

The processor therefore can run in User Mode or in Kernel Mode. Note however that in Kernel Mode, the CPU can either be in process context or in an interrupt context. Thus, there is a difference between a Mode and a Context and this distinction is crucial.



The CPU is in process context when the CPU is executing a system call or when a System Call is executing any other function inside the kernel. The CPU is in Interrupt Context when an external Interrupt has switched the CPU from whatever it was executing to attend to the Interrupt. Thus, when the CPU is executing an Interrupt Service Routine, the CPU has no access to any process'es data structures.

There are a few restrictions on what the CPU can do when running inside an Interrupt Context, which will be described later in the Module on Interrupts.

6. Pre-emptible, re-entrant, SMP-aware etc.

The linux kernel is a re-entrant kernel. What this means is that any function inside the kernel can be invoked at any time and the developer must therefore understand the implications carefully.

For example, if there are any global variables, then its likely that the values inside these variables can get "corrupted" when an Interrupt drags the CPU away from its current thread of execution, and the Interrupt Service Routine again invokes the same function the CPU was executing earlier.

The linux 2.4.x kernel is not pre-emptible. This kernel version guarantees that when an interrupt takes place while the kernel is executing kernel code, on return from the ISR, the CPU will continue executing the kernel code that it was earlier executing, without the scheduler getting invoked.

This guarantee has been removed through an optional feature in the 2.6.x series. Therefore, as of the latest kernel, you must be prepared for all kernel code to be pre-emptible and take adequate precautions.

The linux kernel is Symmetric Multi-processing aware. This means that the kernel takes advantage of more than 1 CPU being available to process tasks faster. Essentially, what this

means is that when a process switch takes place, all information about the running process is saved and theoretically, any processor, not necessarily the one that saved the process context, can start execution of the process at any time. This requires careful anaylsis and thinking, since on one processor, one can assume that the processor would be executing processes serially and to some extent that is “safe”, but in a situation where there are multiple processors, there are no guarantees ***when*** the process that was saved, will again start executing and hence critical regions need to be carefully analysed and tracked.

7. Useful web-links and references

http://www.kernel.org
http://www.kernelnewbies.org
http://www.linuxdevices.com
http://www.linuxjournal.com
http://www.freshmeat.net
http://www.tldp.org
http://www.lwn.net

Parameters	Linux System Programming	Linux Kernel Programming	Linux Device Drivers Programming
Purpose	System Programmers write daemons, utilities and other tools for automating common or difficult tasks.	Kernel Developers focus on interfaces, data structures, algorithms and optimization for the core of the operating system.	Device Drivers use the interfaces and data structures written by the kernel developers to implement device control and IO.
Programming	All the Standard C libraries are available at system programming level.	Kernel programming is done using Module programming technique. There are no standard libraries available. Have to use pure C programming	Device Drivers is done using Module programming technique. There are no standard libraries available. Have to use pure C programming.
Application	System Programmer should know about various low level functions (system calls) for testing device drivers.	A very good kernel programmer may not know a lot about interrupt latency and hardware determinism, but he will know a lot about how locks, queues and Kobjects work.	A device driver programmer will know how to use locks, queues and other kernel interfaces to get their hardware working properly and responsively, but he won't be as likely to fix a page allocation bug or write a new scheduler.