A SunCam online continuing education course

Fundamentals of Linux©

For the Professional Engineer

By

Raymond L. Barrett, Jr., PhD, PE

CEO, American Research and Development, LLC

Abstract & Preamble

Engineering Practices - With the recognition of the Professional Engineer (PE) status for the practice of Computer Engineering in April of 2009, the practice of Control Systems Engineering in October of 2011, and the practice of Software Engineering in April of 2013, there has been the need for specialized continuing education courses related to these practices.

Computer Engineering majors may have taken a course with some elements of Linux, or a predecessor Unix course. Software engineers have probably used Linux in coursework. Other disciplines, including Electrical Engineers and Control Systems engineers will find Linux most applicable in their practice. In some curricula, the operating system employed is assumed to be learned by the student without a formal course. We have chosen to examine Linux because it is a Free Open-Source Software (FOSS) system and as such we can delve into any and all of its components to draw examples.

Also, as software is increasingly included as an integral part of the practice of all disciplines of Professional Engineering practices and it is the legal responsibility of the PE that all results are correct, all disciplines face the need for understanding the tools they employ.

In recent years, the computer community has become acutely aware of the possibility of malicious attacks on software; most often through internet connections of the operating system. This course offers an overview of the Linux operating system discussing its operation and structure and why it is perceived to be less vulnerable to cyber-attacks.

This course introduces the origins of Linux along with the Free-Software and Open-Source developments leading to today's distributions. We discuss the relation of the Linux kernel to popular distributions and two examples from the diverse suite of distributions. Using the bootstrap process involved in loading the Linux kernel, we discuss the memory management, I/O bus hardware interface and file system loading. In a set of appendices, we show how the VirtualBox application is used to support virtual machines and demonstrate concurrent installations of two Linux distributions. We contrast those distributions into the arena of the shell interface, shell programming, process management, communications and applications support.

We introduce many of the topics germane to the study of operating system software in the context of a Free Open-Source Software environment and equip the student with sufficient knowledge to answer basic why and how questions about operating systems software.

1.0 Introduction –

Electronic computer hardware that we would recognize had its origins in early machines by Atanasoff in 1942.
http://www4.ncsu.edu/~belail/The_Introduction_of_Electronic_Computing/Atanasoff-Berry_Computer.html

Hardware development was accelerated by the needs for gunnery tables and the Manhattan project during World War II. The Ballistics Research Laboratory envisioned a machine to calculate gunnery tables and sponsored Mauchly and Eckert to develop the vacuum-tube ENIAC.
http://inventors.about.com/od/estartinventions/a/Eniac.htm

The Eniac Computer U.S. Army Photo

Early machines we very much a hands-on affair with programming performed with plug-in wiring connections. Access to a very expensive machine, loading each program manually was a major bottleneck to widespread dispersal of the technology. As computer technology advanced, particularly in terms of the peripheral printers, paper-tape, and punched-card devices added, it became apparent that each programmer need not re-invent the software for these devices for every program. The growing complexity was a major force in the evolution of the software "operating system." https://en.wikipedia.org/wiki/History_of_operating_systems

The software evolved with the inclusion of computer languages and the assemblers and compilers required for translating symbolic programs to executable machine instructions. Collections of programs in libraries became common-place and programs that combined high-level job-control instructions supported the early functions of the operating system.

Computer hardware evolved from large centralized main-frame systems through smaller mini-computer architectures, and on to the micro-computer, primarily in response to economic forces. Today, we find processors embedded in many applications that could not be implemented a few decades ago.

The marketplace for computers was serviced by large companies like IBM, Univac, Sperry Rand, etc., but evolved through the generations with start-up companies like Digital Equipment Corporation and Data General growing and declining again with the mini-computer to be followed by other ventures like Apple and Compaq introducing microcomputers. A few of the older companies survive and have a presence today.

In a similar fashion, software concepts followed the introduction of new hardware technologies and built on the fundamental concepts introduced with the early machines.

Fundamental to the concept of the operating system is the attachment of software, libraries, peripherals, and other computing hardware to a capability supporting the programmer and eventually a more novice general user community.

One important capability that drove the evolution of operating systems was the inclusion of remote access to the computer, the use of the computer and its operating system in managing communications capabilities, and eventually computer-to-computer communication. Central to this development was the research arm of ATT through Bell Laboratories and the university community. The Department of Defense showed interest and support through the (D)ARPA organization.

Joint research at the University of California, Berkeley resulted in the development of the Unix operating system. https://en.wikipedia.org/wiki/History_of_Unix

When ATT decided to make the System-V version of Unix proprietary to ATT, Berkeley continued with another BSD version that was offered on a more liberal license. https://www.freebsd.org/doc/en_US.ISO8859-1/articles/explaining-bsd/what-a-real-unix.html

Issues with the commercialization, copyright, and proprietary competition of software tools prompted Richard Stallman to create a completely free operating system, found the Free Software Foundation (FSF), https://en.wikipedia.org/wiki/Free_Software_Foundation and provide GNU (GNU's Not Unix). In that sense, Stallman is a key founder of the Free Open-Source Software (FOSS) practices that Linux is noted for. Linus Torvalds wrote a kernel program that was released under the GNU license and is the fundamental building-block under

all versions of Linux. Linux is that kernel and its successors with layers that support many processors, file systems, and other components to make up the distributions of Linux.

Many Linux distributions https://en.wikipedia.org/wiki/Comparison_of_Linux_distributions are available that employ the common kernel architecture and we shall use two of them, Xubuntu and Kali Linux. To support these distributions, we use a virtual machine interface VirtualBox running on the student's machine but not replacing the student's host operating system.

This course installs two comparable instances of Linux distributions and follows from the bootstrap installation for sequencing the installation of resources. The bootstrap process builds component subsystems as they are required and provides a natural progression for discussion of the component subsystems from that construction.

2.0 General Operating System (OS) Objectives –

Fundamentally, the objectives of any OS are to support the interface between the user community and the underlying computer hardware in a relatively uniform manner. At the computer hardware CPU interface, Linux supports over twenty computer architectures: https://en.wikipedia.org/wiki/List_of_Linux-supported_computer_architectures

For Intel alone, https://en.wikipedia.org/wiki/Intel_Core, we can identify eight families of core architecture, each with multiple variants.

Despite the wide variation of hardware we have a common OS that supports them all and presents a common user interface across that variability. This does not mean that a 64-bit Intel machine and a Nokia cell-phone have the same speed and storage capability, but they do share a similar user interface to the OS.

The development tools, including many text-editors, compilers, interpreters, and many applications packages are compatible. It is this compatibility at the user interface that makes Linux popular with developers.

The OS provides the hardware/user interface for memory management, file-system structures, I/O attachments, and communications protocol support as well. Regardless of the hardware system, the Linux instructions do the same functions in nearly the same manner across all platforms. Clearly, the user cannot directly access a disk-drive on system without one attached, but there are some provisions for remote mounting that provide the appearance of such a drive where one is not locally attached.

Many Linux systems support a user interface using a keyboard and monitor, but may be embedded in some applications without such support connected.

The Linux OS supports time management through multi-tasking, multi-threading, multi-processing (i.e., SMP), context management, windowing, and queueing as required.

3.0 Installing Linux (OS) Environments –

The Linux OS can be installed directly on the host hardware as the host operating system on most laptop and desktop computers, as well as a large number of other computing devices. For the purposes of this course, we choose instead to employ a virtual machine on your host machine's OS and install Linux as a guest in a virtual machine.

The virtual machine software is VirtualBox and is installed first on the host. You are directed to the VirtualBox Installation Addendum for a detailed illustrated installation procedure.

Following the installation of VirtualBox, you are directed to a Xubuntu Installation Addendum and a Kali Installation Addendum for detailed illustrated installation procedures to include both distributions as guest OS examples on your host machine.

Using two guest installations supports side-by-side comparisons of the distributions for comparison and contrast. If you should happen to have a catastrophic "crash" of one of the guest OS installations, the components should be readily available on your host machine to rebuild your installation with little difficulty.

Further, the Kali distribution was developed to support a suite of tools for evaluations and investigations into cyber-security issues associated with intrusion detection, and digital forensics. We compare Kali to Xubuntu to show similarities and differences between the two distributions.

The virtual machine architecture is often employed in commercial client-server installations to support multiple client users in a multi-OS service environment as well as for disparate demonstration platforms. The links below are to course addenda that take you through download and installation of 1) VirtualBox, 2) Xubuntu Linux, and 3) Kali Linux. Installation of these software tools are not required, but will certainly add value to this course.

http://www.suncam.com/authors/112Barrett/VirtualBox.pdf
http://www.suncam.com/authors/112Barrett/Xubuntu.pdf
http://www.suncam.com/authors/112Barrett/Kali.pdf

4.0 Linux (OS) Bootstrap Architecture –
The installation of the Linux distribution that you have just completed in the appendices hides much of the detailed functionality from the user. The bootstrap "boot" process expands what you have just seen demonstrated.

4.1 **BIOS**: The first part of the **boot** process is loading the Basic Input/Output System or **BIOS**;
4.2 that enables loading the Master Boot Record (**MBR**);
4.3 which in turn loads the Grand Unified Bootloader (**GRUB**);
4.4 that actually loads the **kernel** code.

The **BIOS** was initially stored on a computer motherboard in a battery-backup "CMOS RAM" device. Today, the "CMOS RAM" has typically been replaced by a Non-Volatile RAM (NVRAM) that retains memory without power.

Philosophically, the NVRAM technology has improved sufficiently that large (more than 1TByte) peripheral devices also replace traditional rotating magnetic disk memory with fast, NVRAM, SSD technology. Slowly, the boot process is evolving, too. The **BIOS** has sufficient code to identify some selections to tell it where to find the remainder of the OS code, but as technology improves, some functions may accompany the **BIOS** into NVRAM spaces on newer motherboards. Usually, during the start-up sequence, the user is given the choice of altering some **BIOS** settings in accordance with the available hardware.

Using our Linux installations on VirtualBox, we modify settings in VirtualBox to indicate some of the **BIOS** functions, but the Linux **BIOS** is also accessible from the guest installation environment and such functions as boot-order (for example) can easily be changed from the client side of the installation. In many installations, you can access the underlying hardware using the virtual machine with essentially the same quality of service as the host system, but you generally cannot access hardware that does not exist on the host. For instance, if you indicate that the guest machine should boot from a non-existent floppy-disk, and no alternate boot device exists, the Linux will fail to boot load.

The **MBR** program historically occupied a location such as "Track 0" on a hard-drive designated as the boot device. Older systems may have employed a floppy disk, and newer devices employed CD/DVD drives with a similar default location as appropriate.

In the Installation Addendum for each Linux distribution, you were directed to download a copy of the distribution as an **.iso** file. As such, the file is an image of a CD/DVD and is read as if it were that media. We choose to place copies of that file in an easily accessed location in the file-system of the host machine. VirtualBox reads the **.iso** file image as if it were reading an actual hardware device of the type specified and listed in its boot-order list. On installation, the boot order directs the virtual machine to use the **.iso** image. Afterward, when Linux has bootstrapped itself to copy files onto its virtual disk drive, we re-direct the process to bootstrap from that virtual disk drive instead. Changing the boot order is easily done from the virtual machine description on the host machine.

The initial bootstrap operation from the **.iso** image provides the function of placing MBR file copies on the virtual disk drive that you defined in your virtual machines. The next time the bootstrap refers to that copy.

The remaining bootstrap software is in the Grand Unified Bootloader (**GRUB**) with a little more interaction supported. This multi-level bootstrap process is an artifact of legacy systems with very small peripheral memory resources that needed the progressive loading of portions of the software. We can say legacy is no longer needed, but the resurgence of very small systems, especially in the Internet of Things (IoT), makes small Linux systems widespread there, too.

The **GRUB** loads the Linux kernel and **initrd** (Initial RAM Disk) image and establishes the root file system. That file system is established in the RAM memory because the rest of the peripherals and drivers are not yet loaded and the peripheral hardware is not yet accessible, but the system cannot function at all without RAM, so that resource must be present. We will say much more about memory and its management in Linux.

The kernel performs the **mount** of the file as specified in the **GRUB** as part of the configuration. The kernel then executes the **init** program as process ID 1. As we shall see later in the case of Xubuntu, we can examine the code at **/sbin/init**, then **/etc/init**, and **/bin/init** that creates the first user space process after the kernel **init**. You may look at the source code listed as **init/main.c** in the Linux kernel code in the Xubuntu installation. The same code is available for the Kali distribution but you must get it from the repository as it is not installed by the Kali kernel. The **init** contains sufficient peripheral driver software to access some hardware and decide required run level to direct it to load and run programs associated with that run-level. The run-level supports 0=halt, and 6=reboot that may confuse you if these are set by mistake later in your investigations. Other run-level settings include: 1- single user mode, 2 – multilevel without NFS, 3 – full multi-user, and 5- X11. Run-level 4 is unused. As we will see, Kali Linux does not

support multi-user because it is intended for use by cyber-security personnel and is usually restricted to personnel with administration privileges. Kali Linux is restricted to support single user mode only.

Using file system commands, we can examine the bootstrap operation in detail. We defer the detail until after we introduce the use of the terminal window and the concepts of the shell.

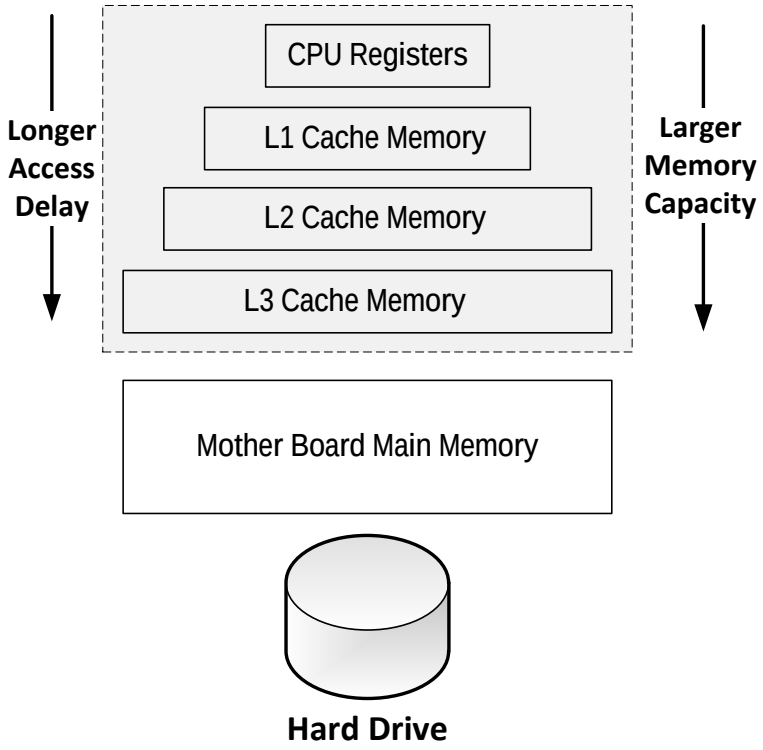
As we shall see, the kernel has and operates only in the restricted kernel mode. In that mode it has full access to all physical computer resources. It is kept in memory in a pre-defined memory space, but can be modified appropriately. Optional kernel modules can be added for driving hardware devices, for altering components of the file systems, and for modifying communications protocols, for examples. The module management includes the loading and unloading of modules, as well as making them available dynamically. The kernel also manages drivers to the extent of loading and unloading and indicating status of the drivers. Formatting services are provided, protocol selections are provided, and file system alternatives are supported. In addition, the kernel resolves conflicts in hardware access and possible multiple driver versions.

The kernel shares in managing windows and sub-systems. Services include process management, inter-process communication, memory management, as well as the already mentioned file-system, I/O, and network management. The rest of the Linux distribution builds on these essential kernel capabilities.

5.0 Linux (OS) Memory Management –

Mostly memory resources are implemented in physical hardware; however, Linux uses a "virtual" memory approach to augment the available physical memory. We illustrated the bootstrap loading process, establishing a file system in RAM memory. Limitations of early systems mandated hierarchical memory architectures based on the cost/performance tradeoff of each memory resource.

Each hardware CPU has a relatively small set of registers closely associated with the computing resources. These registers are typically the fastest memory anywhere in the system, but are costly in terms of area in the CPU, wiring access, and supporting logic requirements. Also within the CPU there may be some quantity of specialized Cache RAM memory connected to high-speed wiring that does not require much logic support for selection.



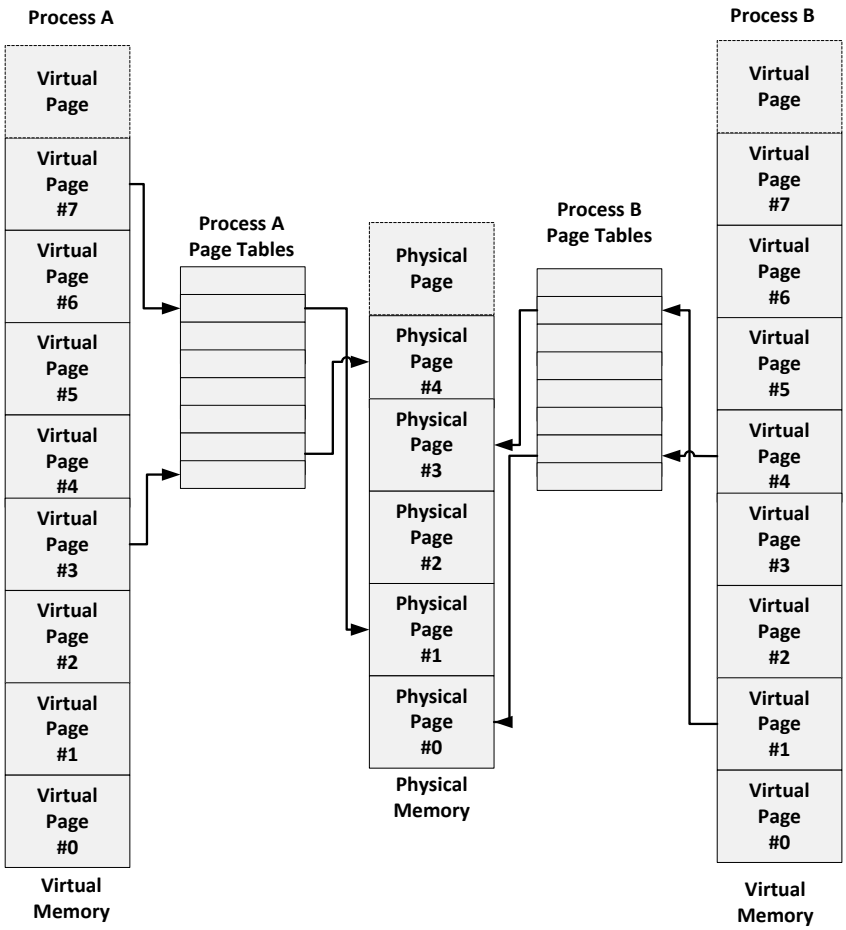
Selection of a memory location is achieved by assigning an address for that location. The relatively few registers, but more is generally better, can be achieved by assigning a register number to each. With 32 bit and 64 bit processors, the register number is often made part of the instruction acting on the contents. As we get further from the register file, though, it becomes impractical to assign a fixed address in the same way that the registers are accessed. Instead, we use a bus mechanism with a control bus, a data bus, and an address bus.

A 32 bit machine supports single-words of 32 bit and can span 2^{32} = 4, 294,967,296 (over 4 billion) distinct memory locations. As such, this imposes a 4G limit on memory address space. Newer 64 bit machines span over 16 x 10^{18} locations. A 4G limit can be finessed by extra levels of address decoding to determine which of several spaces are addressed.

For memory that is within the CPU, the logic decoding for such spans is prohibitive. To control the addressing spans, Linux employs a page scheme with a size assigned to pages and an address to locate the page. No fixed page size is imposed, but 12-bits (4k locations) is typical.

Local memory pages that are copied within the CPU need only be managed by page number. In today's CPU hardware, there is a Memory Manager Unit (MMU) that keeps track of pages and where they are located. Some amount of RAM is co-located on the CPU in one or more cache memory storage spaces and controlled by the MMU. Cache memory, as well as much of the rest of the RAM memory is controlled by the MMU. However, each CPU manufacturer has their own ideas of memory management. Some older machines do not use pages, but rather use a "Segmented" memory approach. Linux imposes a paged approach, regardless of the underlying MMU, but does support variations for page address control.

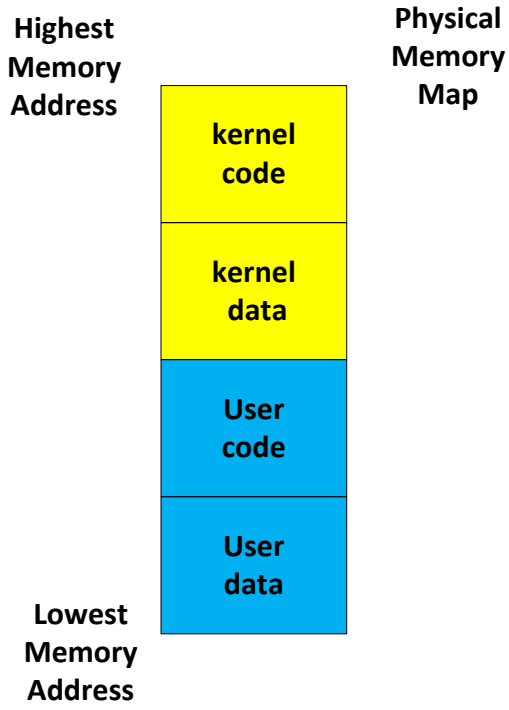
We illustrate the management of mapping virtual memory onto physical memory in the context of the memory assigned to Linux process entities. We discuss the dynamics of the assignment and de-assignment later in a discussion of the function of these processes.



The MMU serves to map a page physical address to an equivalent “virtual” page address. Tables are maintained within the CPU for the spaces managed by the MMU, and extended virtual memory page addresses tables are maintained by Linux. Pages may be located in the on-board RAM, in disk memory, as well as other memory resources available.

In addition to identifying the relationship between physical and virtual memory, additional information is kept to show how recently memory was used. Less often used memory is usually moved out of the CPU areas of physical memory giving priority to more often needed access.

Likewise, CPU memory is tracked for changes. If a location is to be sent back to a remote location from the CPU but is unchanged, there is no need to re-write unchanged content. Memory management is one of several fundamental sub-system tasks that Linux supports.



Within the memory address space are four ranges of accessible memory addresses. Two are assigned to kernel code and kernel data, and another two are assigned to user code and user data. Within each page address, there are five possible fields maintained in a Page Global Directory, a Page Upper Directory, a Page Middle Directory, a Page Table, and an Offset (the typical 4 k address space).

Navigating the directory structure to load/store a required page is the major function of the memory management subsystem, In addition, not all memory is occupied and assigning available memory to a task, as well as recovering unused or released memory is another important memory management subsystem task.

During the bootstrap loading, a filesystem, sufficient driver software, and initialization programs are copied into the available RAM spaces under the control of Linux memory management as well as the MMU of the particular CPU in the installation. Soon after the initial phases are complete, the kernel identifies all the devices connected to the I/O bus. Key among these are devices like keyboard, monitor, and peripheral memory devices like disk drives and SSD memory devices. Once those are identified, the kernel finds and loads the appropriate driver software for each device connected to the I/O bus.

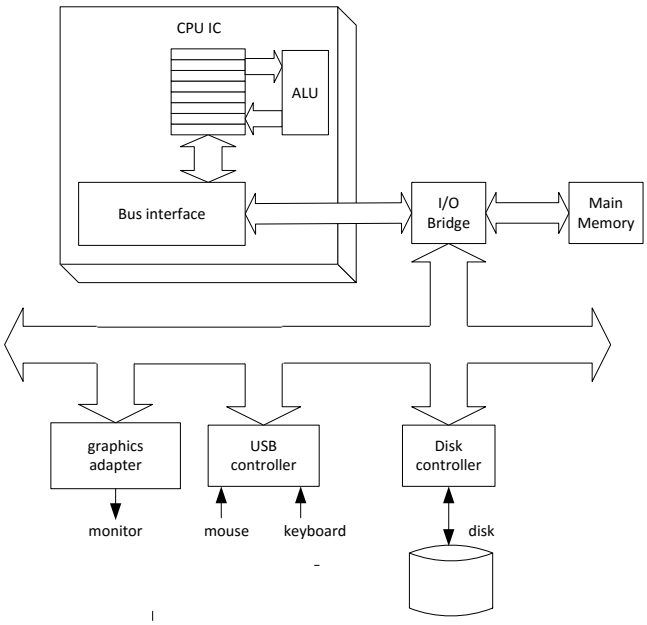
Variations of Memory Management under Linux can fill a course unto itself. More detail can be found at: **http://www.win.tue.nl/~aeb/linux/lk/lk-9.html**

6.0 Linux (OS) I/O Bus Management –

Insofar as the Memory Management Subsystem is concerned with the resolution of virtual memory addresses of pages across hierarchical levels of memory, the I/O bus has similar subsystem management issues. I/O management is complicated by the various bus standards including IDE, PCI, SATA, USB, and others. For the most part, I/O devices attached to these different bus structures each has a software device driver that must match the device attached.

For one such bus, the PCI bus, the kernel generally interfaces with a PCI controller that is a hardware equivalent to the kernel because it mates many CPU designs to a standard bus, while mating many I/O devices to the controller. During the boot, the kernel communicates with the PCI controller to identify which of 32 possible ports are populated and which of 8 functions is associated with that port. Devices are identified by a vendor-device pair to identify the necessary driver. It is possible for the device to be identified by the kernel as part of its conflict resolution function, but the driver selected in the process of elimination may be a poor match and performance impaired. The attached device signifies the memory requirements and the I/O MMU function (either in the controller or the kernel) finds space in the memory map. Memory access to the PCI bus is recommended to avoid conflicts with use of proprietary I/O bus

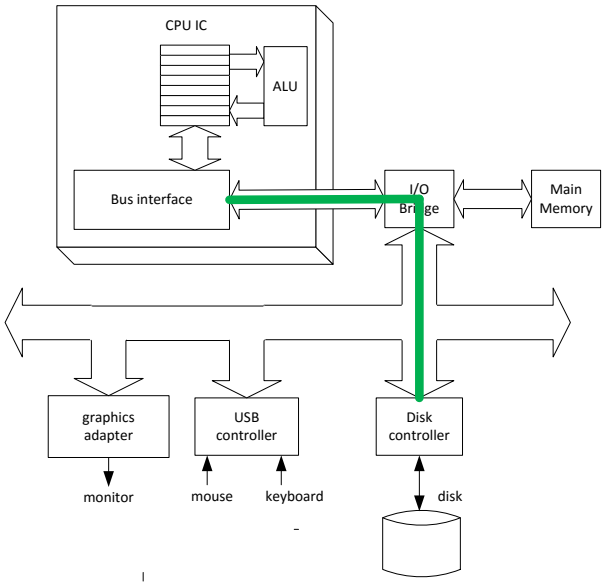
protocols (such as in the older x86 hardware). One use for an I/O MMU is to use the main RAM as video memory, with the contents mapped to a display.



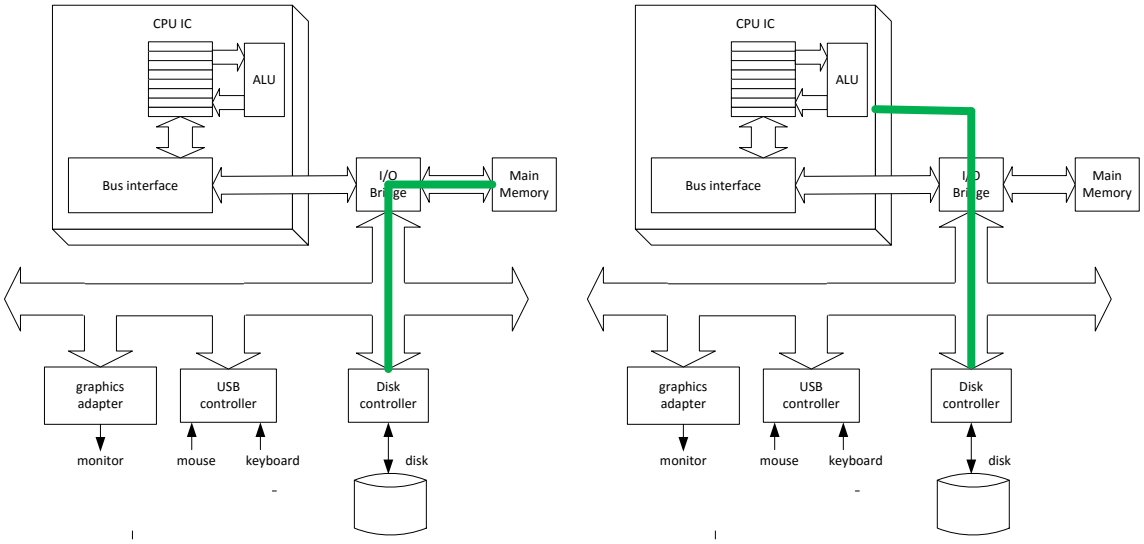
The system components on the I/O bus may include, but are not limited to:

1) A USB Controller - Proprietary controller "driver" for Manufacturer/Model
   Generic Universal Serial Bus (USB) controller interface to kernel
   Device USB software loaded to kernel from device manufacturer
2) Graphics Adapter - Proprietary controller "driver" for Manufacturer/Model
   Specialized controller interface to kernel for Manufacturer/Model
   Monitor software loaded to kernel from device for Manufacturer/Model
   Video Memory mapped into OS
3) Disk Controller - Proprietary controller "driver" from Manufacturer/Model
   Specialized controller interface to kernel for Manufacturer/Model
   Specialized Disk interface to kernel for Manufacturer/Model
   Both Read and Write Operation similar except data directions

If the system (most usual) supports Direct Memory Access (DMA), that function may also utilize the PCI bus, but the DMA controller is set up and a range of RAM contents are transferred by that controller. The system architecture below with an I/O bridge supports a DMA disk read/write as shown in the following:



A program sets up transfers by sending information about whether READ or WRITE is intended, the data location on the disk, the data location in main memory, and the amount of the data to be transferred. The actual transfer occurs between the disk controller and the main memory directly without further CPU participation in the transfer as we see in the next illustration.



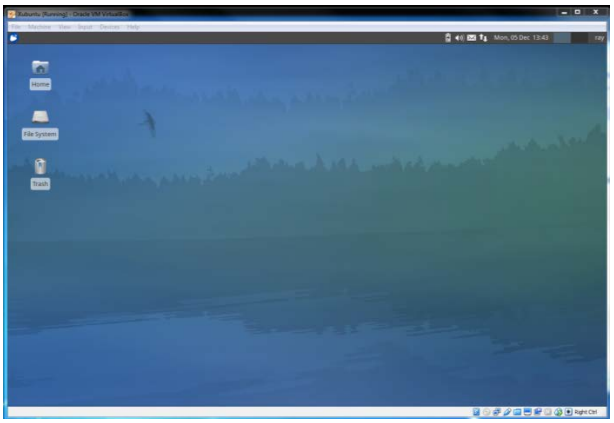
On completion of the transaction, the disk DMA controller signal completion directly to the CPU.

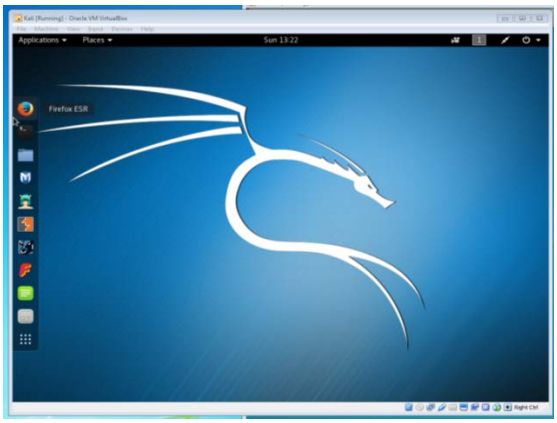
7.0 Linux (OS) User Interface Subsystem –

The bootstrap process .as outlined so far, continues typically with drivers for keyboard, pointing device, and graphics interface to support a user interface. Some interaction with the user is supported prior to continuation of the bootstrap and we digress to discuss the user interface before continuation with file system, task, process, and thread dynamics topics. The digression into the user interface is a suitable point to introduce the windows system and the shell interface that is largely the discriminating factor set between Linux distributions.

As you have encountered already in the installation of VirtualBox and the two Linux distributions in virtual machines, you were required to identify yourself and the machine to the kernel as well as supply a login password. The initial login is reserved for the "System Administrator" or "superuser" and accompanies higher privileges. It is assumed that only a superuser (identified in some operations as **su**) would be installing an operating system. Any time you wish to enjoy kernel privileges, you will be required to identify yourself as **su** and supply this password, identified as the "**root**" password. Later, after a successful login, you can change your password using the **passwd** command. You can also enjoy temporary privileges using the **sudo** command, but must supply the root password if the operation requires kernel privileges. You can also use the **su** command to login with a different user ID and password.

At the installation of the Xubuntu distribution and the Kali distribution, you certainly should notice the similarities and differences in appearance of the two distributions. Linux distributions have the choice of eight Desktop Environments. http://www.howtogeek.com/163154/linux-users-have-a-choice-8-linux-desktop-environments We the xfce default for Xubuntu and KDE default for Kali in a side-by-side illustration for comparison and contrast.



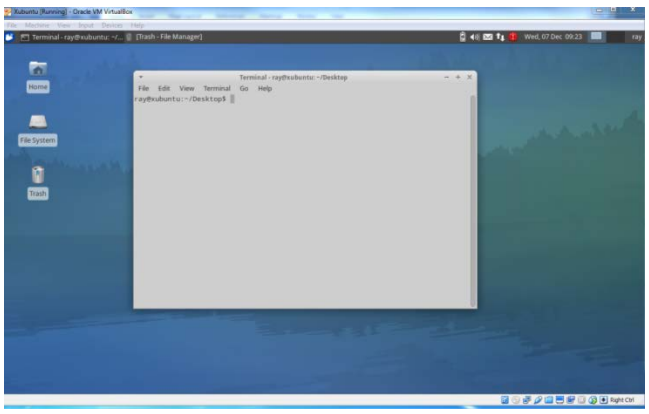


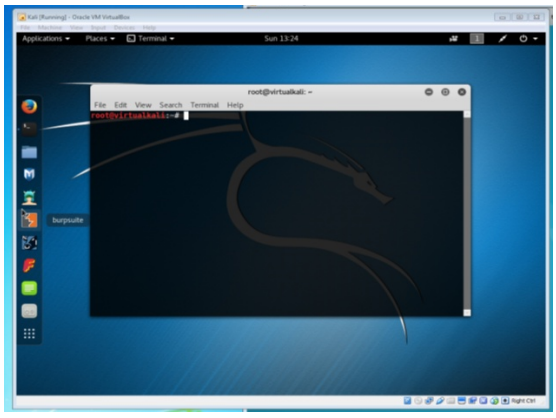
7.1 Linux Shell Introduction – All access to the hardware through the kernel is using a shell that can be accessed using a terminal sub-window in the user-interface "desktop" windows of the distributions contrasted above.

In early computer systems, the user interface may have been an electromechanical "Teletype" machine, soon replaced by a combination of an electromechanical keyboard and Cathode-Ray Tube (CRT) pair to provide an alphanumeric user interface. As a first-level interface, the pair has evolved with technology somewhat but maintains the "command console" or "terminal" primitive I/O function. Today, the "Terminal Window" replaces the hardware with a text-only window in the Graphical User Interface (GUI) represented as the "Desktop" Window. http://www.linfo.org/terminal_window.html

During the first time the Linux distribution is loaded, an identification name (ID) for login and a password is established. The first-time login ID is identified as the system administrator and that ID is granted special access to the kernel. Be careful that you do not forget the credentials.

One way to instantiate a shell is to use the pointing device (mouse) to locate an open area on the desktop window and hold the right button down to obtain a selection list. Selecting the "Open Terminal Here" from the list produces the Terminal or Shell windows as shown.





All access to the hardware is through the kernel and mediated using a shell; shell commands can be executed indirectly and not require direct operator intervention. As we see above, Linux distributions provide a distinctive Graphical User Interface (GUI) represented as their desktop

window. We illustrate showing Xubuntu and Kali desktop windows and shell windows for each GUI. Both Xubuntu and Kali are derived from a common Debian distribution but serve different purposes or specializations.

The shell window provides a line of text known as the shell prompt line. The last character in the line is the prompt character and indicates the type of shell that is installed. Normally, that character is the **"$"** and indicates a "**bash**" shell. You have some alternates to choose from, based on the historical development of the shell used, including a **"%"** character that indicates a "**C**" shell. You are expected to supply a command at the prompt regardless of the shell employed.

The Bourne, Korn, and Bash shells all use the **"$"** character but are similar enough that you should have no difficulty with shell differences at this point. Generally, if you see the **"#"** character as the shell prompt, it signifies that you are granted "**root**," **su**, or administrator privileges regardless of the shell.

7.2 Shell Command Interface – In response to the shell prompt, you are expected to respond with a shell command. We will introduce many of these commands as required. A complete reference manual for Linux commands can be found on sourceforge at: https://sourceforge.net/projects/linuxcommand/

You should download "The Linux Command Line" file and save it for reference but be aware that is well over 500 pages but covers mostly just the command line interface.

Built into Linux you will find a reference manual in the form of "**man**" pages. The man pages tend to be short and sometimes difficult to interpret, but nearly exhaustive and complete about every command and its options.

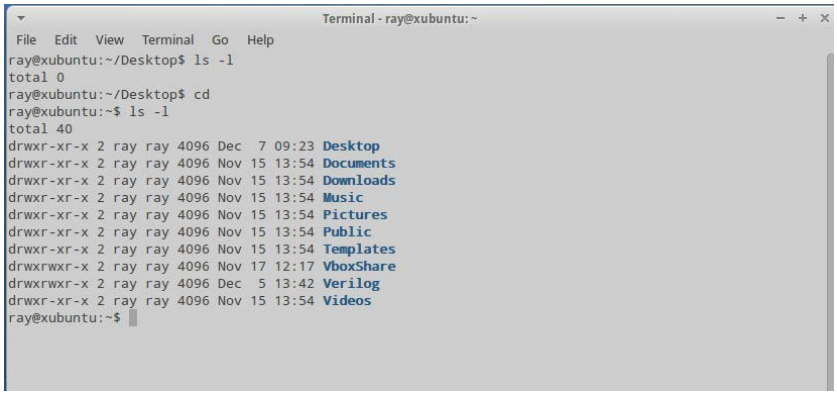
As a first experience, type the shell command **pwd**, followed by the enter key. The system responds with the "Path to the Working Directory" result that should look like "**/home/**<*your_name_here*>**/Desktop**" and indicates that your shell is on the Desktop window, and the system identifies the login as the "<*your_name_here*>**"** representation.

Because the text-based shell interface is alphanumeric and the available command repertoire has expanded, it has been necessary to provide online reference documentation describing the

available commands and their options. The online manual pages are accessed using the command **man**, followed by the command. You can try **man pwd** as an example. By itself, **man** describes the online manual function.

For familiarization, you should be able to try many of the file system commands including the **ls** command that lists the files at the **pwd** location. You should be able to copy a file using the **cp** command (if the permission settings for that file permit the copy operation). We recommend that you try the **man** descriptions for each of these examples before and after you try them to familiarize yourself with the commands and how they are described.

Using the "**ls**" command with the option "**–l**" at the command prompt, provides a file descriptor with identification fields that are explicit about that file, typically like the Xubuntu example below:



In the Xubuntu screenshot shown above, we see the result of the command sequence in a terminal shell window on the desktop. Our first attempt shows no files on the Desktop, but using a **cd** command without a file name specifier takes us to the root directory and the result with several files listed there. The first character in the field indicates the type of file, followed by file permissions.

type	owner	group	world
d - l	rwx	rwx	rwx

Type: "**d**" is for directory, "**–**" is for a regular file, and "**l**" is for a symbolic link; all files at the root directory above are type "**d**," indicating they are all directories. The three following three-character groups are for "owner," "group," and "world" and are formed in positioned patterns of three characters consisting of "**r**," "**w**," "**x**," or "**-**" in a specific "**rwx**" order.

The "**r**" character in the "**r**" position of a field indicates that field has a "**read**" permission. Similarly, the "**w**" character in the "**w**" position of a field indicates that field has a "**write**" permission. Also, the "**x**" character in the "**x**" position of a field indicates that field has "**execute**" permission. The permissions are granted distinctly to the file's owner, group, and the rest of the world.

Following the permissions, we find the file owner and group identification fields that default to the login names, followed by the file size, the date, the time, and the file name.
https://wiki.archlinux.org/index.php/File_permissions_and_attributes

The file identification and permissions are important parts of the Linux security provisions and will be discussed more deeply in that context.

You can follow reference manual for Linux commands for trying other Linux commands such as the **cp** command. After a copy **cp**, you may want to try a remove **rm** to delete that test case. If you try the move **mv** command, you must realize that it essentially renames the file and/or changes its location, so be careful with **mv** commands.

You can make new sub-directories using the **mkdir** command and later remove them using the **rmdir** command. There are often restrictions that you must empty any directory that you wish to remove prior to its removal so that data is not accidentally deleted. We have discussed commands that are limited to the file system at the introductory phase of this discussion to minimize the probability of accidents.

There are sufficient numbers of command-line shell commands be need an entire course to assure mastery. Meanwhile, there are excellent online learning resources for command-line reference and tutorial purposes:
https://sourceforge.net/projects/linuxcommand/
http://tldp.org/LDP/GNU-Linux-Tools-Summary/GNU-Linux-Tools-Summary.pdf
http://scc.qibebt.cas.cn/docs/linux/script/LinuxCommand.pdf

7.3 Shell Programming – Linux, including the kernel, is programmed in a high-level language. Necessarily, alphanumeric entry is supported both directly and indirectly. Commands can be executed directly as typed into a shell. Command execution can also be executed indirectly by constructing an alphanumeric named shell "script" file simply by typing that name.

Similarly, alphanumeric text files can be submitted to a supporting compiler or interpreter for translation to machine code so that kernel-specific programs can be developed.

The development of Unix (predecessor to Linux) is closely tied with the development of the "C" programming language. Unix was one of the first operating systems implemented in a higher-level language than the assembly language of the underlying hardware. The original project was based on a General Electric "Multics" operating system as part of a porting effort to a much smaller Digital Equipment PDP-7 machine. The effort cut so much of the Multics OS capability that the new OS was called Unix as a sound-alike for the result of a surgical operation. The co-development of Unix and "C" has kept the "C" language as a natural choice for Unix and its derivatives like Linux. Usually, a "C" compiler is ported to a new architecture and the kernel is compiled for that target. The "C" language is a natural choice for shell programming, but there are other choices available, too.

Some of the original tools delivered with Unix were **sed** & **awk** that offered alphanumeric string processing at the Unix command prompt. Eventually, the attempt was made to combine the two tools into the replacement **perl** scripting language. Most recently, the interpreted **python** language has been added to support command-line scripting.

Typically **sed** & **awk** are part of the Linux distribution. Usually, **perl** and **python** can be acquired from open source locations as well as the GNU "C" compiler.

https://directory.fsf.org/wiki/Sed
https://www.gnu.org/software/gawk/
https://www.gnu.org/software/gawk/manual/gawk.html
http://docs.python-guide.org/en/latest/starting/install/linux/
https://gcc.gnu.org/wiki/InstallingGCC

8.0 Linux (OS) File System –

The Memory Management Subsystem supports the hierarchy of memory resources and provides a unified virtual addressing space across those resources. The File System provides access to the most remote parts of the memory, organizing that access in a hierarchical system. The user sees a seamless memory over that entire space.

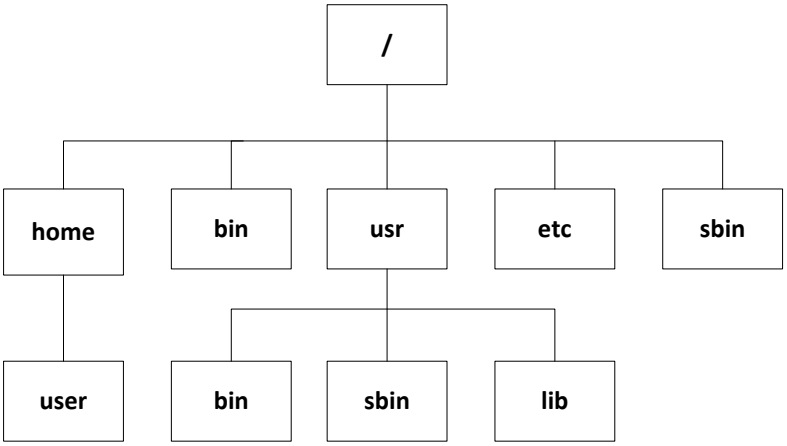
The bootstrap loader function of the kernel loads a micro-kernel into system memory and executes that to load the remainder of a kernel image into main memory. This stage is necessary because the peripheral memory like disk storage is unknown to the kernel initially. During the bootstrap process, the peripheral memory resources are identified as well as the I/O bus driver software for that set of peripherals followed by an image copy of the kernel that is loaded to the peripheral memory.

Today's processors manage their register resources, cache memory, and have a Memory Management Unit (MMU) in hardware that is identified and used during phases of the bootstrap process. Once the kernel is loaded sufficiently, the bootstrap process signals the user to re-start the machine from the installed image rather than from the installation medium. It is during a restart that the rest of the Linux kernel, as well as the distribution packages are installed. The Linux OS manages the drives, SSD, and installs both the file-system and the remainder of the system data.

During the bootstrap process, essential files are loaded and named as follows:

- **/bin** Essential user command binaries needed to be available also in single user mode.
- **/sbin** Essential system binaries (e.g. init, insmod, ifup)
- **/lib** Libraries for the binaries in **/bin** and **/sbin**
- **/usr/bin** Non-essential user command binaries that are not needed in single user mode
- **/usr/sbin** Non-essential system binaries (e.g. daemons for network-services)
- **/usr/lib** Libraries for the binaries in **/usr/bin** and **/usr/sbin**
- **/etc** Host-specific system-wide configuration files
- **/dev** Device files
- **/home** User home directories (optional)
- **/proc** Virtual file system documenting kernel and process status as text files

The basic kernel file structure is organized as a hierarchical "tree" built as follows:

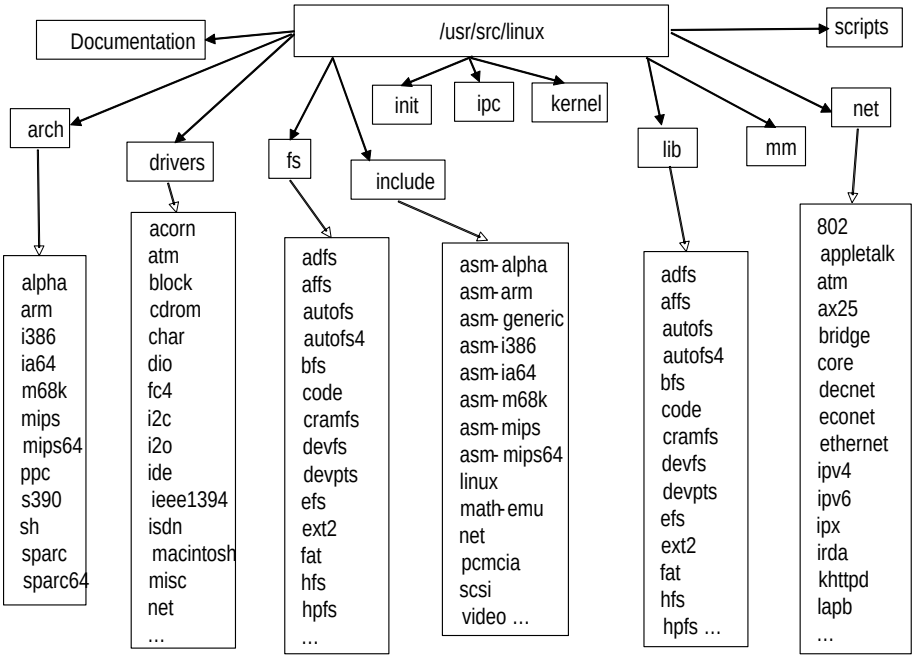


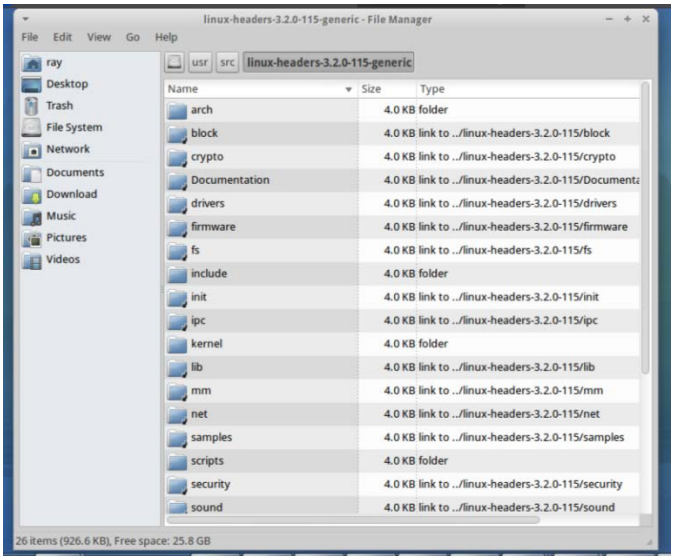
In a general sense, the user encounters a seamless invisible “memory” resource unless particular resources are specifically examined. One such specific case is performance monitoring as well as cases of diagnosis and debugging (usually during program development).

In some specialized cases, particularly with modern co-processor devices including a Graphical Processor Units (GPU) and the closely related General Purpose Graphical Processor Units (GPGPU) there are additional memory spaces that are mapped into both the file system and the memory map by specialized interfaces.

Likewise, Virtual Machines that are specialized application programs like VirtualBox are mapped into both the file system and the memory map by specialized interfaces.to provide host CPU file system access and specifically host memory access to map the virtual memory and virtual disk drives of the guest virtual machines.

One distinctive feature of Linux, associated with its FOSS nature is the availability and sometimes inclusion of the source files in the system installation. Generally, we can find the kernel source files as well as some higher-level inclusions. The Open-Source files for Xubuntu Linux are also loaded in a hierarchical “tree” as follows:





Using the Graphical File System viewer from the Xubuntu Desktop, we see the same structures in the illustration above. For Kali, though, sources are not included in the distribution and must be obtained from the repository: http://docs.kali.org/general-use/kali-linux-sources-list-repositories

9.0 Linux (OS) Software Subsystems Architecture –

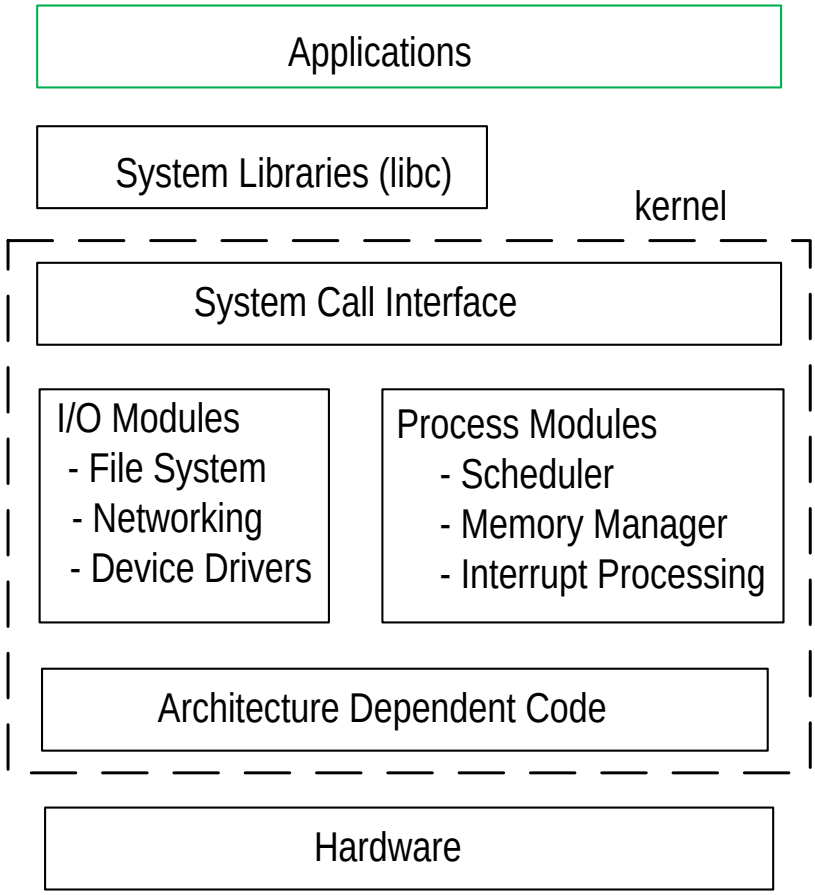
The Linux distribution, including and supported by the kernel, is organized in major software subsystems that include:

- Memory management
- I/O management
- File system management
- Process management
- Inter-process communication
- Networking

The Linux kernel is responsible for providing services in support of these major subsystems; to illustrate we followed the bootstrap sequence of loading the kernel into main memory as an easy introduction to the basic services including memory management of the internal memory hierarchy. We followed with a discussion of the I/O subsystem to support the extension of the memory map onto the disk drive and an introduction for the need for a virtual-memory approach for large memory spaces. Then we used the storage of the Linux kernel onto the extended memory of the disk drive to introduce the file system. We introduced many of the services of the shell in the context of the file system, as well as added shell programming topics to that discussion.

In some sense, we have introduced topics that describe the resources available, but within the context of static components available for manipulation. We now introduce topics that are more concerned with the dynamics of data movement within the Linux operating system. We introduce processes, tasks, and threads that are descriptors of system dynamics. With these dynamic entities executing within the Linux context, we introduce the Inter-Process Communications (IPC) and build extensions into remote communications support. Each of these topic areas are Linux subsystems in their own right.

The process management subsystem has the responsibility of performing any and all services on the available resources within the context shown in the illustration below:

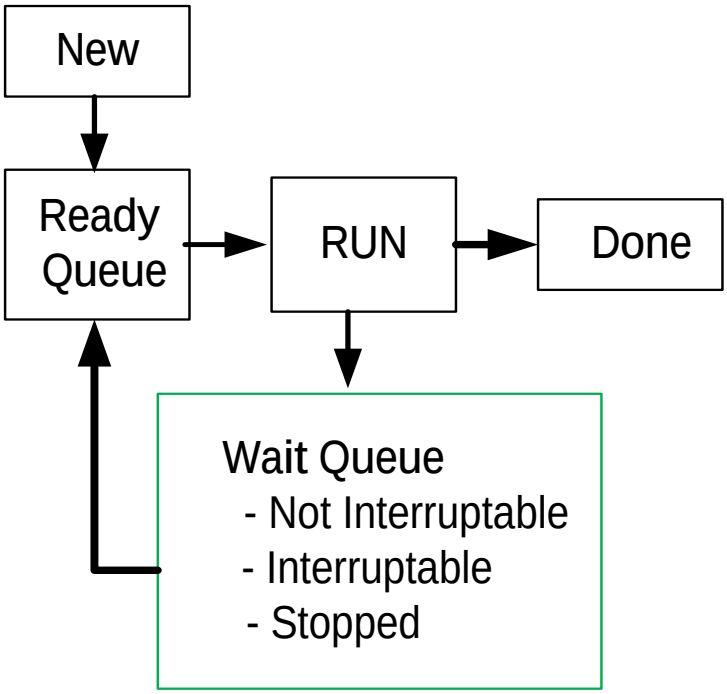


10.0 Linux (OS) Software Data Flow (Dynamics) –

Linux supports concurrent services using the atomic-level process unit of operation. The process is fundamentally an activity that performs a function on the resources. Some process activities are more or less obvious like moving data from one location to another. Other process activities manage time resources in sharing access and accounting for the resources used as well as accounting for the time used in accessing those resources.

Fundamental to process management is the initiation of a new process. Each new process is generated by a system **fork** call. The call to **fork()** creates a child process from a parent process. All Linux/UNIX processes are children of process ID 1, the **init** process. The generation of a process includes establishment of its properties. The new process is included into the process management queue system using the **execv** system call. Each process is assigned a process

identity (PID) with a process credential that is a group ID for resource access and a process personality that is an ID for legacy system compatibility.



Further, at the **fork**, the process inherits an argument vector list that contains the program name and command-line arguments, as well as an environment vector list used for parameter passing so that such data is passed at the process level rather than needing global assignments.

Each process is taken in a context with scheduling rules to suspend and restart the process, accounting context that supports resource usage rates and totals useful for billing and performance monitoring, a file system context defining the root directory, default directories, and allowable file openings. In addition the context includes a signal handler table to relate signals to addresses and a virtual memory context to support a private address space.

Linux also supports spawning threads using the **fork** system call that are identical to processes except they are clones of their parent and share the parent's data structures.

Linux processes are spawned by **fork()** but threads means something specific and different than a process. Once a parent process spawns a child with **fork()**, the child inherits the parent's memory

space. The child memory space is identical to the parent's unless the child attempts to modify it (copy-on-write). https://en.wikipedia.org/wiki/Copy-on-write

The most common threads library in Linux is **pthreads**, defined as a set of "C" language programming types and procedure calls originated from a POSIX environment through UNIX. Pthreads are spawned by a call to **pthread_create()**. There is no copy-on-write if a thread modifies memory. Threads share the same process ID & memory space as the process that created them. https://en.wikipedia.org/wiki/POSIX_Threads

Processes are arranged in a set of queues that support priority scheduling, time-slice scheduling and are used to support the kernel, I/O and other drivers, and user spawned processes. Thread scheduling is similar and supports a fair round-robin time-slice access as well as an interruptible real-time FIFO protocol.

The **wait** queue is associated with processes that are not currently eligible for execution. Device drivers and network functions typically spend time in the **wait** queue. Completion of the process that is waiting is generally signaled by a hardware interrupt that wakes all processes in the wait queue to determine any needed changes of status. Inter-Process Communication (**IPC**) is achieved using a shared-memory space.

Symmetrical Multi-Processor (**SMP**) operation for multi-core structures is achieved by allowing only single processor kernel access but multi-process threads per processor.

11.0 Linux (OS) Communications Network Architecture –

The Linux Communications Network Architecture is arranged in layers. One early architecture definition is due to the International Standards Organization (**ISO**) and is known as the Open System Interconnect (**OSI**) Architecture model, It is well known and relatively well enough understood that the concepts are widely dispersed and clear to all. At about the same time as the introduction of the **ISO** model, the **ARPANET** was being developed and the similar **TCP/IP** layered protocol stack came out of that effort.

For long-distance communications, the **Internet** is basically a **TCP/IP** protocol approach, but there are numerous Local Area Networks (**LAN**) that support variants and alternate architecture.

The well-defined protocol layers of the **OSI** Reference Model are:

Application Layer		**Application Layer**
Presentation layer		**Presentation layer**
Session Layer		**Session Layer**
Transport Layer		**Transport Layer**
Network Layer		**Network Layer**
Data-Link Layer		**Data-Link Layer**
Physical Layer	<=>	**Physical Layer**

The OSI Model Layer-to-Layer Communications

1. The **Application Layer** that describes how real work actually gets done. For example, this layer would implement file system operations.
2. The **Presentation Layer** that describes the syntax of data being transferred. For example, this layer could describe how floating point numbers are exchanged between hosts with different math formats.
3. The **Session Layer** that describes the organization of data sequences larger than the packets handled by lower layers. For example, this layer could describe how request and reply packets are paired in a remote procedure call.
4. The **Transport Layer** that describes the quality and nature of the data delivery. For example, this layer could define if and how retransmissions are used to ensure data delivery.
5. The **Network Layer** that describes how a series of exchanges over various data links can deliver data between any two nodes in a network. For example, this layer does define the addressing and routing structure of the Internet.
6. The **Data Link Layer** that describes the logical organization of data bits transmitted on a particular medium. For example, this layer defines framing, addressing and check-summing of Ethernet packets.
7. The **Physical Layer** that describes the physical properties of the various communications media, as well as the electrical properties and interpretation of the exchanged signals. For example, this layer defines the properties of Ethernet coaxial cable, the type of connectors used, and the termination method.

The Linux kernel needs/has a collection of protocol drivers to support the various protocol layers. The Linux kernel needs/has a collection of network device drivers to support the various hardware devices. At the Applications Programming Interface (**API**), the mechanism employed to make a connection to Network Communications services and Inter-Process Communications (**IPC**) is the socket interface. The original socket approach was developed at the University of

California at Berkeley, became part of the Unix kernel, and the concept was inherited as part of Linux with the name BSD sockets.

Linux implements the high-level kernel BSD socket interface that is similar to a file system's file interface. The pseudo-code shown below illustrates a "C" code file system interface in contrast/comparison to a socket interface:

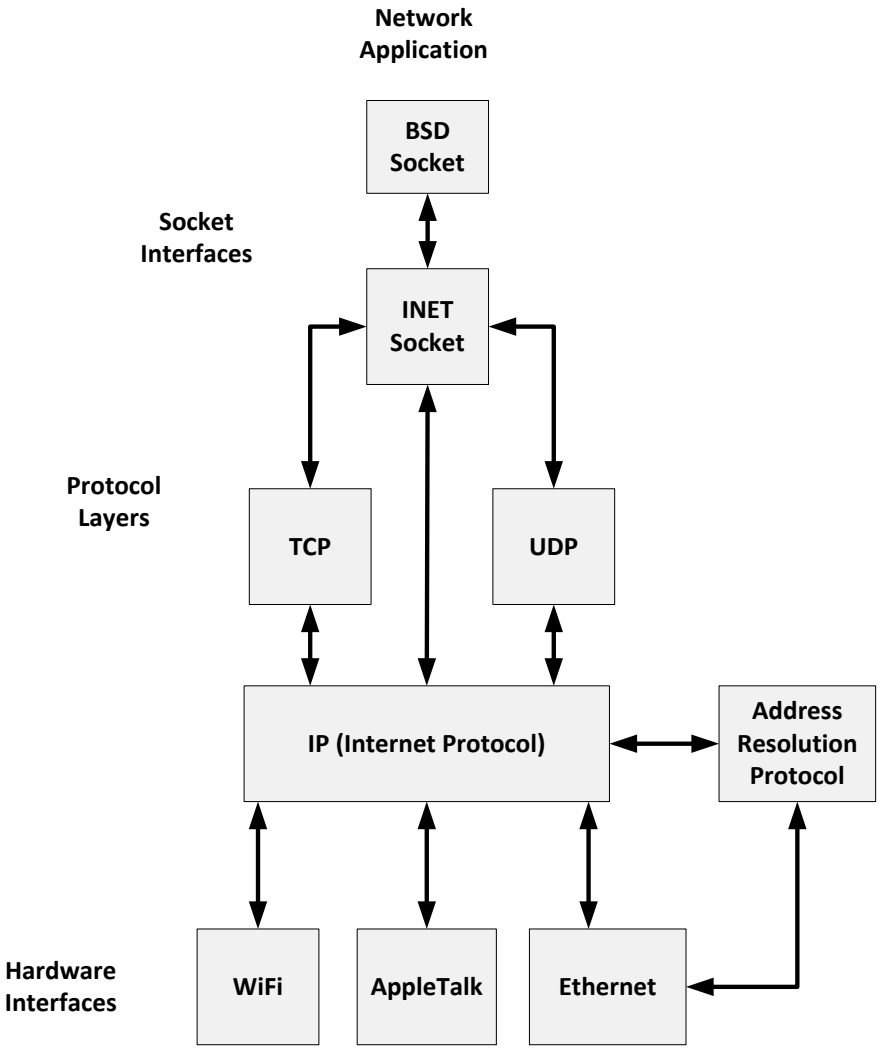
```
Main()
 {
   FILE *fd;
   fd = fopen(…);
   process(fd);
   fclose(fd);
 }
```

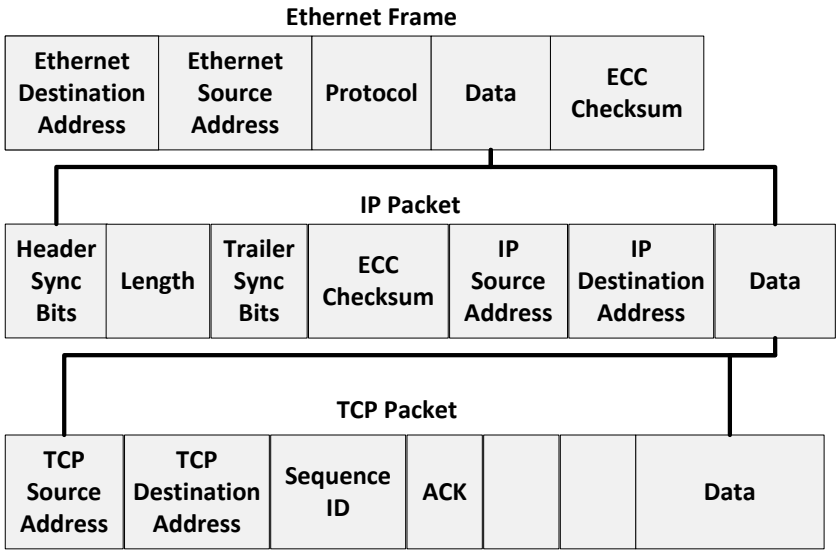
```
Main()
 {
   int sockfd;
   sockfd = socket(…);
   process(sockfd);
   close(sockfd);
 }
```

The socket interconnection initiates two basic end-to-end Network Communication channel approaches:

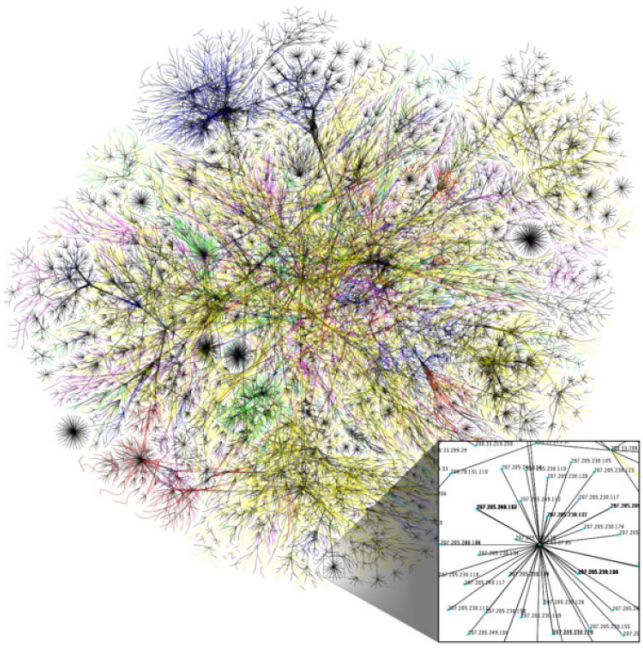
Virtual circuit **connection** (such as TCP) provides reliable two-way persistent channel route
Datagram socket provides a **connection-less** and unreliable connection (such as UDP)
Datagrams are often used to find a route for a virtual circuit and build routing tables.

The illustration below shows a kernel BSD socket interface to an application-level with an Internet Protocol employed to form a connection to a few possible hardware devices. We show an example using the formation of an Ethernet connection below:



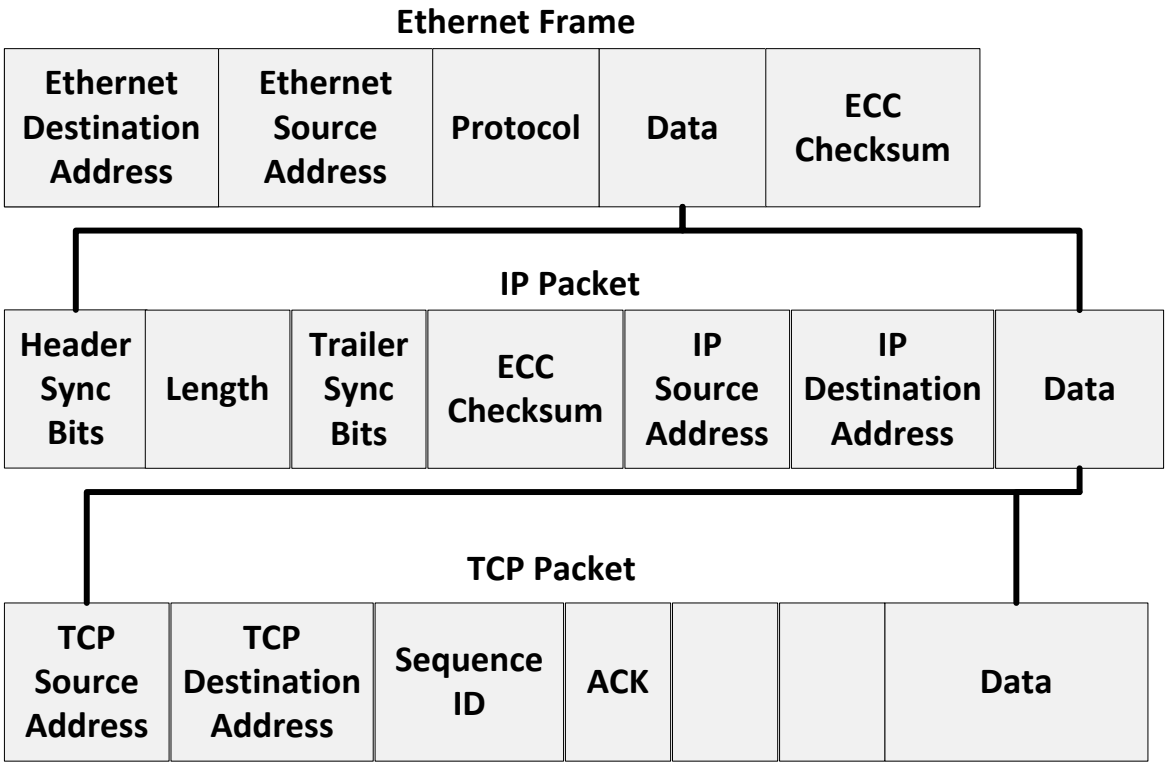


Today, the Internet represents the global topology of networks and inter-network communication world-wide.



Internet Map 2006

https://commons.wikimedia.org/wiki/File:Internet_map_1024_-_transparent,_inverted.png



Network debug is an entire course itself; a few commands you may want to know follow. We can use the **hostname** command to find and alter the machine's designated hostname; ***machine@network*** for example. To determine if a remote machine is reachable, we use the **ping** command with the hostname or the IP address for that machine. The **ip** command is used to obtain the IP address of specific devices and enable or disable specific network interfaces. The **ss** command lists all the TCP, UDP, and Unix socket connections to your machine. You can obtain the **traceroute** utility to identify the path for a packet to reach a destination, including the number of hops involved. The **dig** command gives access to Domain Name Server (DNS) lookup tables. The **route** command provides access to your route tables. Quite a few resources are available on-line to support your debug efforts: https://itsfoss.com/basic-linux-networking-commands/

The older **ipv4** packet address structures have IP addresses that are comprised of four basic fields each with a number from "0" to "255" coded as an 8-bit binary number for a total of a 32-bit address. It is that 32-bit address that supports 2^{32} or more than 4 billion IP addresses that is causing a shift to the newer **ipv6** packet addresses structures. As part of the newer **ipv6** scheme is the address length of 128bits. As well as the obvious address space increase available, the packet size is increased, the checksum is eliminated because network reliability is improved, and other features are added. Special fragmentation capabilities are added to **ipv6** that enable the simplification of routers while putting a somewhat higher complexity on the hosts.

Within a single network, there is a simple message format called Address Resolution Protocol (**ARP**) that is used in the Link Layer of the Internet Protocol Suite or equivalently in two layers of the OSI model that is used extensively in Ethernet to map Internet addresses to link layer addresses. The **ARP** is bound tightly to **ipv4** networks. In the newer **ipv6** networks the Network Discovery Protocol (**NDP**) replaces the ARP functions. Both sets of protocol software are supported by Linux distributions. https://en.wikipedia.org/wiki/Address_Resolution_Protocol

In support of network operations, a packet sniffing set of hardware or software in a network analyzer or protocol analyzer supports interception and logging of message traffic. Several of the specialized software tools offered in the Kali Linux distribution provide some parts of this capability.

12.0 Linux (OS) Security –

The Linux OS has a reputation for being much more secure than other proprietary OS products. Not all security issues are addressed by installing Linux, however.

As a first line of defense, the system administrator and each user needs a good password. No amount of communications security effort can prevent malicious breaches by personnel with the login names and passwords of legitimate users. Worst of these is that information for a system administrator.

Each file in the system needs a carefully thought-out set of access permissions.

If we assume that we do require a network connection, we must address issues of network security. A relatively secure approach used by military systems is the air-gap approach with no

electrical communications support and no wireless communications support. Once we have decided that we require a networked system, the levels of security needed become the issue.

There are multiple classes of intruders from the curious to the malicious. Some intruders may simply explore and others may pirate your resources to run a clandestine operation, disburse malicious software onward, or, as in the case of a competitor, copy, destroy, or replace proprietary data. You will need to decide policies about what you wish to protect, what the risk of loss may be, and to what extent you are willing to go to protect your data.

If you are in control of the data at each end of communications paths, the most effective means of protecting data on the network is end-to-end encryption, but while the data is protected, the service may still be vulnerable to such malicious practices as Distributed Denial of Services (DDoS) attacks. That flood channels with unwanted communications that overload your capacity to process and decide which are valid and which are malicious messages.

Occasionally, private wiring, RF links, satellite channels etc., are the only means to secure end-to-end service with disregarding actual physical attacks. Again, military models are meant to be of this level of security. If you are not willing to go to the expense of entirely private facilities, a Virtual Private Network (VPN) may suffice. The VPN is constructed using secured Gateway servers that provide end-to-end encryption between secured end-point networks.

An intermediate secure connection can be provided by a secure shell connection. A secure shell is established at each end of a communications service using the **ssh** command. That secure shell utilizes end-to-end encryption by employing public-key/private-key encryption service. A request is made to obtain both server and client **ssh** software from the key-server. A transaction with the key-server generates public and private keys for the transactions. The request to establish an **ssh** end-to end connection involves the retained private keys at each end and the retrieval of the public key from the server for each end. Both a public key and the private key are needed at each end for one-way traffic, but two such encrypted connections form the two-way **ssh** signaling service.

The **ssh** secure shell is not the only way to access remote servers and files. Older File Transfer Protocol (**ftp)** and Remote Desktop (**rdp**) protocols support file and desktop access. They are legacy tools mostly but useful for less secure access.

The US Department of Justice collects statistics on cyber-crimes of many sorts:
http://www.bjs.gov/index.cfm?ty=tp&tid=41

There is an open challenge for hacking Linux systems in a side-by-side comparison to Windows:
http://www.hackinglinuxexposed.com/about/challenge.html

There is an open compilation of vulnerabilities:
http://techtalk.gfi.com/most-vulnerable-operating-systems-and-applications-in-2014/

Do virus and malware programs exist for Linux:
http://www.linuxandubuntu.com/home/linux-security-how-can-your-linux-be-hacked-using-malware-trojans-worms-web-scripts-etc

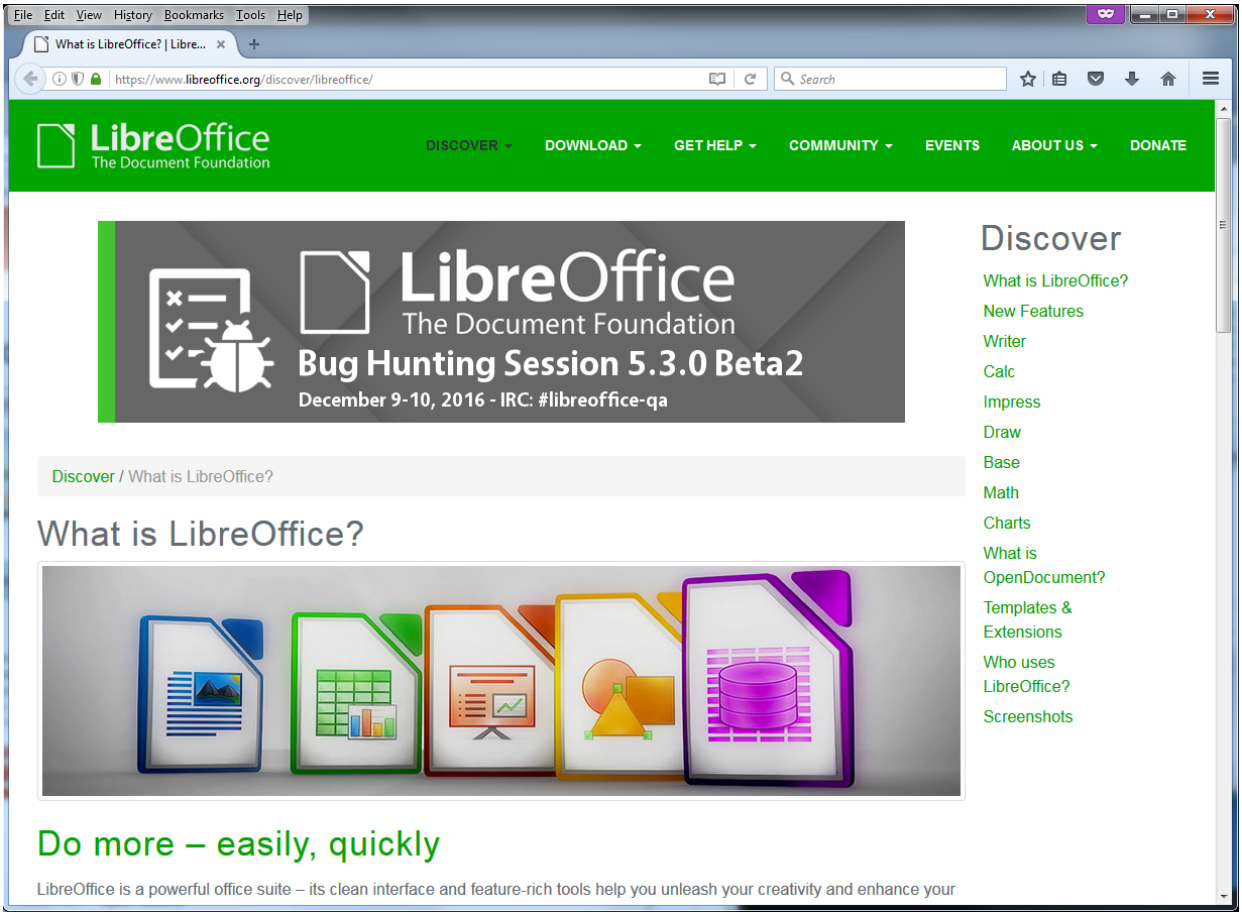
The Kali Linux distribution has an extensive collection of tools for system security evaluation. Kali Linux special tools include:

1) Fern Wifi Cracker puts a GUI on AirCrack to crack WEP and/or WPA passwords,
2) Burp Suite, included with Kali, tests web applications for security vulnerabilities,
3) Hydra is a free brute-force password cracking tool testing a broad range of services,
4) John the ripper on Kali with the Johnny GUI for testing weak passwords,
5) Maltego who and what links are connected,
6) Metasploit Framework simulates attacks to find security issues,
7) Nmap with the ZenMap GUI for network discovery and security auditing,
8) Zed Attack Proxy for finding web-based vulnerabilities,
9) Sqlmap for testing SQL database servers,
10) Wireshark is a network protocol analyzer for live capture and offline analysis.

13.0 Linux (OS) Applications Installations –

There are numerous FOSS applications programs that can be installed in Linux that will seem familiar to users of other proprietary Operating Systems. Listed below are just a few examples that can reside in the applications spaces of the Linux OS. Be aware that some of these applications require substantial disk space and the Linux installation may only support these particular installations with generous disk space allocations. Embedded and virtual installations are particular environments that require careful consideration for available installation resources.

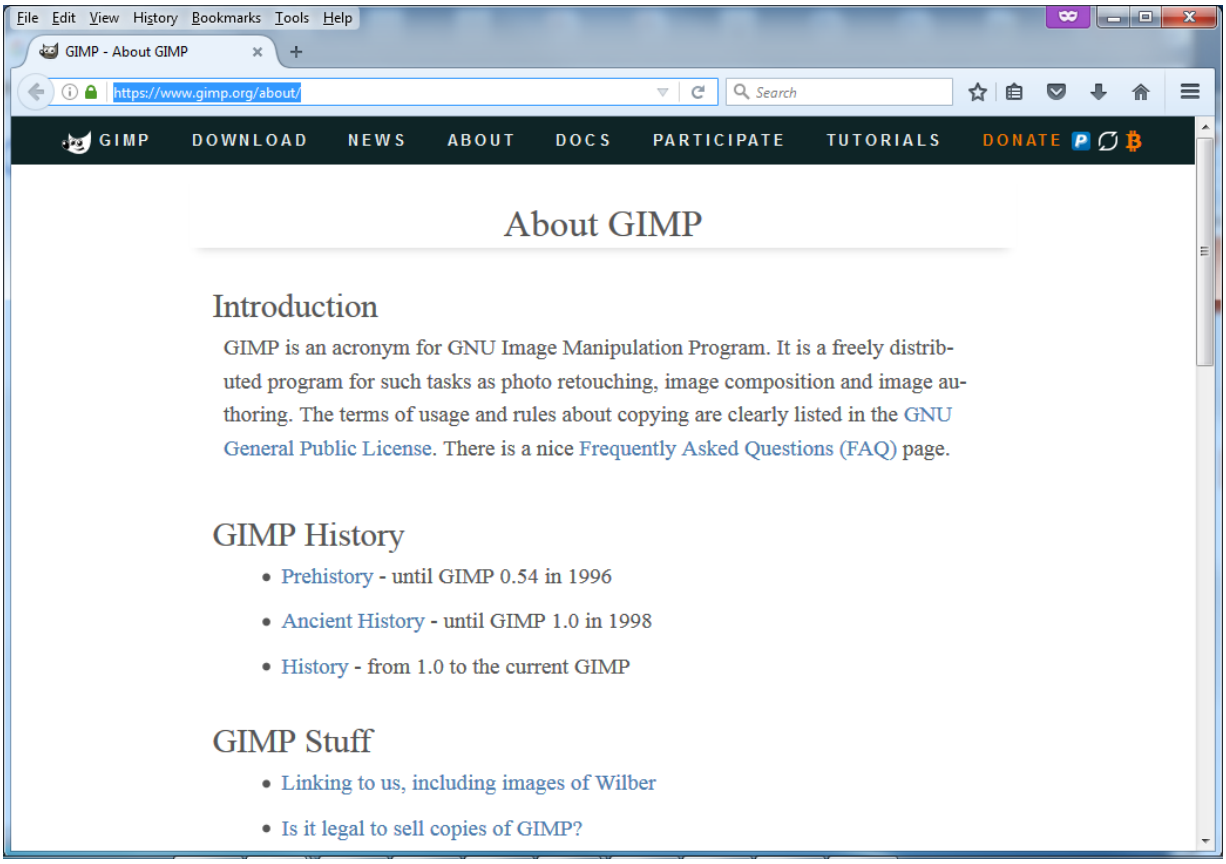
13.1 LibreOffice is a suite of compatible tools: https://www.libreoffice.org/discover/libreoffice/



LibreOffice provides compatible applications for:

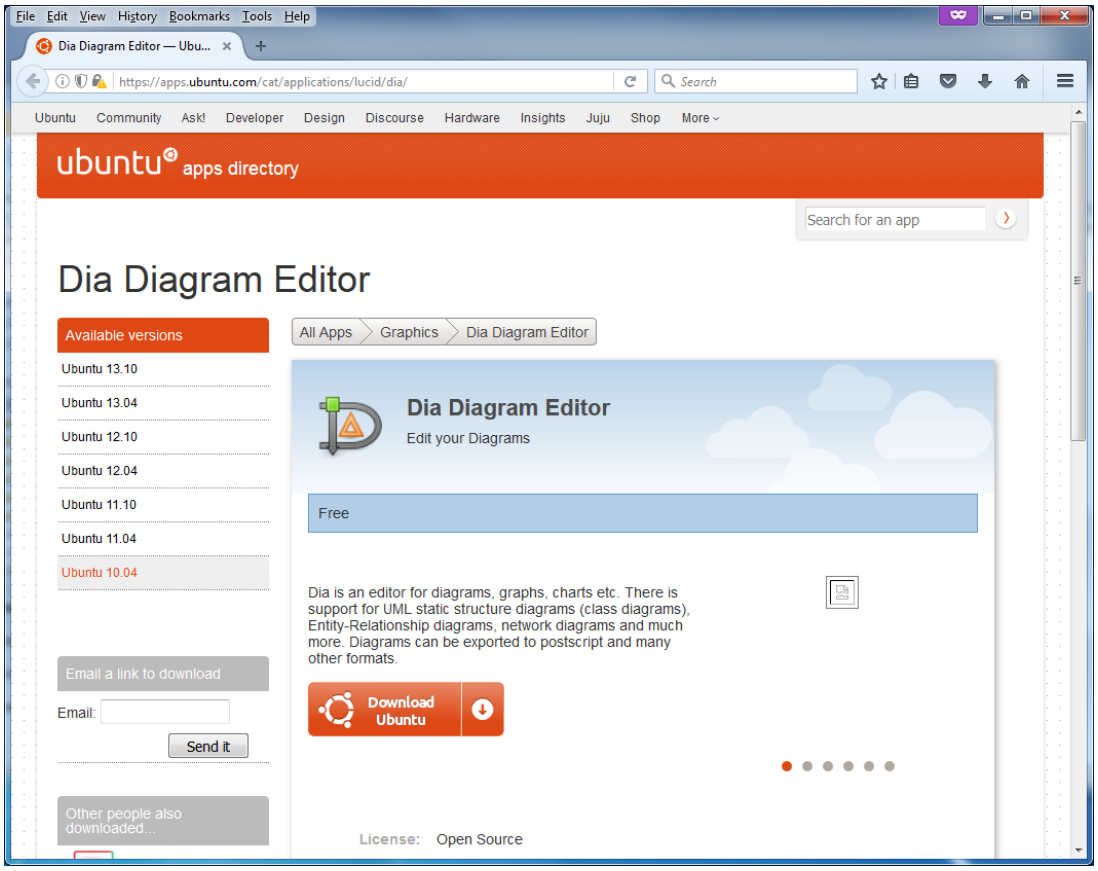
Word Processor - Writer
Spreadsheet - Calc
Presentation – Impress
Illustrations – Draw
Scientific Typesetting – Math
Database Manager – Base
http://libre-software.net/how-to-install-libreoffice-on-ubuntu-linux-mint/

13.2 Gimp is an acronym for GNU Image Manipulation Program and is an alternative to the commercial Photoshop software: https://www.gimp.org/about/



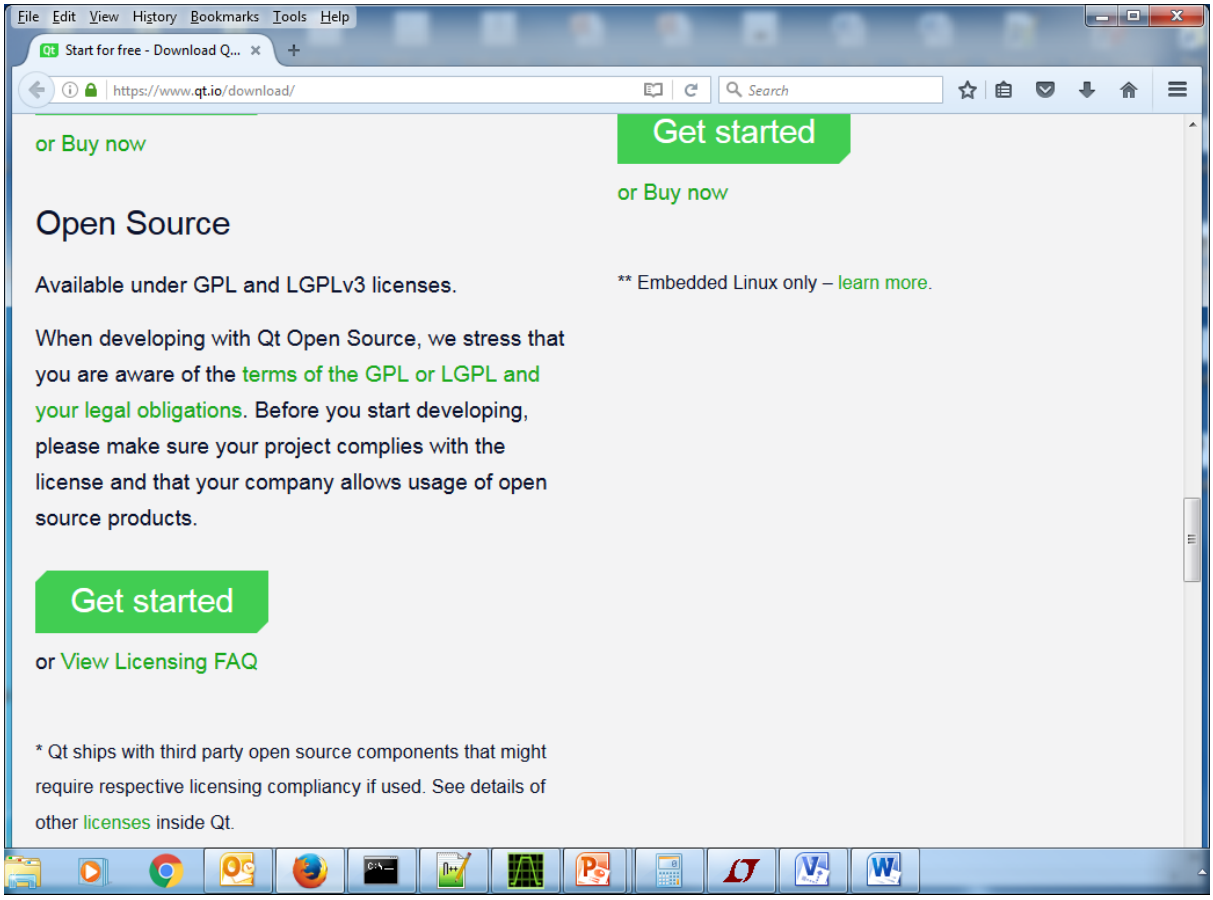
` https://docs.gimp.org/2.4/pdf/en.pdf
http://www.omgubuntu.co.uk/2014/08/whats-new-in-gimp-2-8-12-plus-install-ubuntu

13.3 The Dia applications is for drawing technical diagrams:
https://apps.ubuntu.com/cat/applications/lucid/dia/

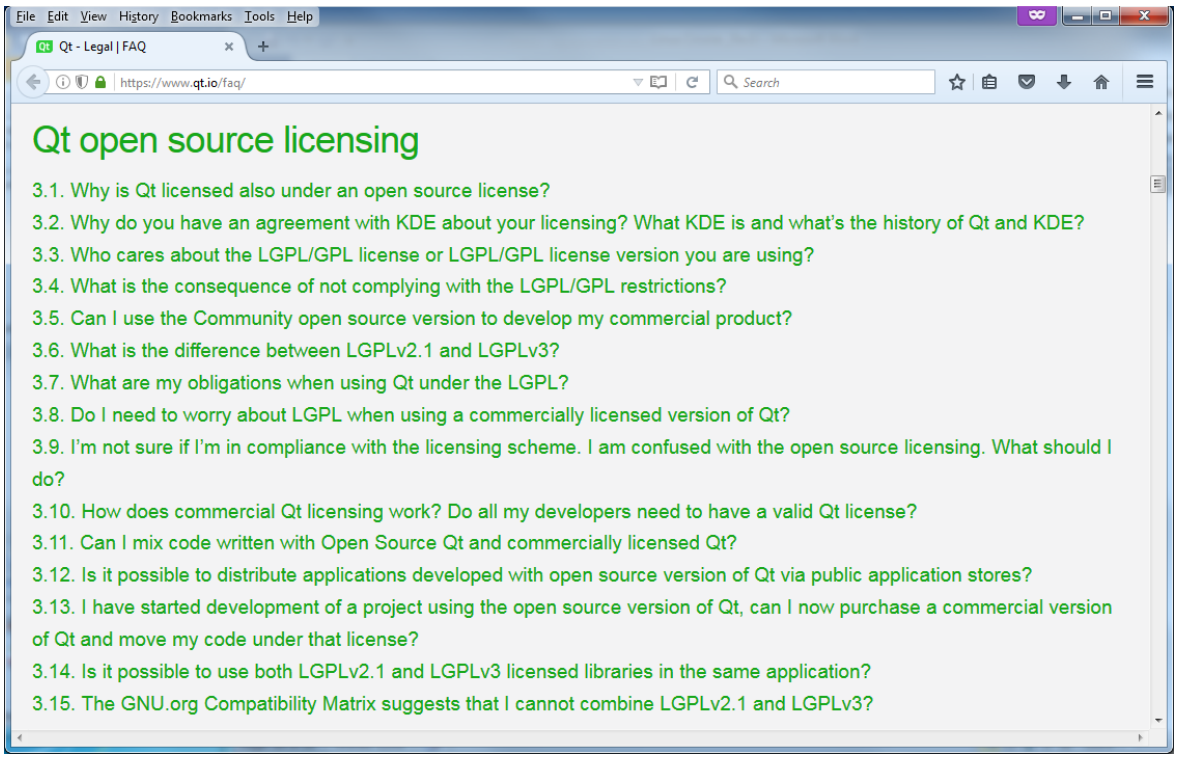


Because Dia is available as a Ubuntu family application, it can also be obtained directly using the command line: sudo apt-get install dia

13.4 Qt is a cross-platform, Integrated Development Environment (IDE) with capabilities for Graphical User Interface (GUI) development that is most closely tied to the KDE environment used for the Kubuntu distribution of Linux, but can easily be installed in the Xubuntu distribution we have been using. It also supports compilers for various languages including “C.” and “C++,” as well.



The installer determines the OS environment and warns about the licensing limitations. You may still find the IDE useful in an academic environment, or for development of Linux modules that are to be release in FOSS forms. https://www.qt.io/download/



13.5 For certain circumstances, there is an application that supports running Windows software under Linux. For those cases that require such capability under Linux, the reader is directed to the Wine Application: https://en.wikipedia.org/wiki/Wine_(software)

14.0 Linux (OS) Course Summary and Conclusions –

This course introduced the origins of Linux along with the Free-Software and Open-Source developments leading to today's distributions. We discussed the relation of the Linux kernel to popular distributions and two examples from the diverse suite of distributions. Using the bootstrap process involved in loading the Linux kernel, we discussed the memory management, I/O bus hardware interface, and file system loading. In a set of appendices, we showed how the VirtualBox application is used to support virtual machines and demonstrate concurrent installations of two Linux distributions. We contrasted those distributions into the arena of the shell interface, shell programming, process management, communications and applications support.

The open-source nature of the Linux kernel and the free distributions chosen for demonstration exemplified the value of FOSS for functionality and access to the underlying software without proprietary issues preventing the free exchange of information. The student should be able to support the why and how of operating systems in general and Linux in particular.

