System Programming

Assignment

LINUX

&

OPERATING SYSTEM

****

Abhishek Kumar

11501172

3CE4

**INDEX**

TITLE Page No.

INTRODUCTION 2

FEATURES 9

HISTORY 20

PLATFORM 22

COMPILER WORKING 26

OPERATING SYSTEM 36

**INTRODUCTION TO LINUX**

Linux is quite possibly the most important free software achievement since the original Space War, or, more recently, Emacs. It has developed into an operating system for business, education, and personal productivity. Linux is no longer only for UNIX wizards who sit for hours in front of a glowing console (although we assure you that many users fall into this category). This book will help you get the most from Linux.

Linux (pronounced with a short i, as in LIH-nucks) is a UNIX operating system clone which runs on a variety of platforms, especially personal computers with Intel 80386 or better processors. It supports a wide range of software, from TeX, to the X Window System, to the GNU C/C++ compiler, to TCP/IP. It's a versatile, bona fide implementation of UNIX, freely distributed under the terms of the GNU General Public License .

Linux can turn any 80386 or better personal computer into a workstation that puts the full power of UNIX at your fingertips. Businesses install Linux on entire networks of machines, and use the operating system to manage financial and hospital records, distributed computing environments, and telecommunications. Universities worldwide use Linux to teach courses on operating system programming and design. Computing enthusiasts everywhere use Linux at home for programming, productivity, and all-around hacking.

What makes Linux so different is that it is a free implementation of UNIX. It was and still is developed cooperatively by a group of volunteers, primarily on the Internet, who exchange code, report bugs, and fix problems in an open-ended environment. Anyone is welcome to join the Linux development effort. All it takes is interest in hacking a free UNIX clone, and some programming know-how.

# Copyright issues.

Linux is covered by what is known as the GNU General Public License, or **GPL**. The GPL was developed for the GNU project by the Free Software Foundation and specifies several provisions for the distribution and modification of free software. Free, in this sense, refers to distribution, not cost. The GPL has always been subject to misinterpretation. We hope that this summary will help you understand the extent and goals of the GPL and its effect on Linux.

Originally, Linus Torvalds released Linux under a license more restrictive than the GPL, which allowed the software to be freely distributed and modified, but prevented any money from changing hands for its distribution and use. On the other hand, the GPL allows people to sell and profit from free software, but does not allow them to restrict another's right to distribute the software in any way.

First, it should be explained that free software that is covered by the GPL is not in the public domain. Public domain software by definition is not copyrighted and is literally owned by the public. Software covered by the GPL, on the other hand, is copyrighted by the author. The software is protected by standard international copyright laws, and the author is legally defined. The GPL provides for software which may be freely distributed but is not in the public domain.

GPL-licensed software is also not shareware. Generally, shareware is owned and copyrighted by an author who requires users to send in money for its use. Software covered by the GPL may be distributed and used free of charge.

The GPL also lets people take, modify, and distribute their own versions of the software. However, any derived works of GPL software must also be covered by the GPL. In other words, a company may not take Linux, modify it, and sell it under a restrictive license. If the software is derived from Linux, that software must be covered under the GPL also.

The GPL allows free software to be distributed and used free of charge. It also lets a person or organization distribute GPL software for a fee, and even make a profit from its sale and distribution. However, a distributor of GPL software cannot take those rights away from a purchaser. If you purchase GPL software from a third-party source, you may distribute the software for free, and sell it yourself as well.

This may sound like a contradiction. Why sell software when the GPL allows you to get it for free? Let's say that a company decided to bundle a large amount of free software on a CD-ROM and distribute it. That company would need to charge for the overhead of producing and distributing the CD-ROM, and may even decide to profit from the sales of the software. This is allowed by the GPL.

Organizations that sell free software must follow certain restrictions set forth in the GPL. They cannot restrict the rights of users who purchase the software. If you buy a CD-ROM that contains GPL software, you can copy and distribute the CD-ROM free of charge, or resell it yourself. Distributors must make obvious to users that the software is covered by the GPL. Distributors must also provide, free of charge, the complete source code to the software distributed. This permits anyone who purchases GPL software to make modifications to that software.

Allowing a company to distribute and sell free software is a good thing. Not everyone has access to the Internet and the ability to download software for free. Many organizations sell Linux on diskette, tape, or CD-ROM via mail order, and profit from the sales. Linux developers may never see any of this profit; that is the understanding reached between the developer and the distributor when software is licensed by the GPL. In other words, Linus Torvalds knew that companies may wish to sell Linux, and that he might not see a penny of the profits.

In the free software world, the important issue is not money. The goal of free software is always to develop and distribute fantastic software and allow anyone to obtain and use it. In the next section, we'll discuss how this applies to the development of Linux.

# The design and philosophy of Linux.

New users often have a few misconceptions and false expectations about Linux. It is important to understand the philosophy and design of Linux in order to use it effectively. We'll start by describing how Linux is not designed.

In commercial UNIX development houses, the entire system is developed under a rigorous quality assurance policy that utilizes source and revision control systems, documentation, and procedures to report and resolve bugs. Developers may not add features or change key sections of code on a whim. They must validate the change as a response to a bug report and subsequently ``check in'' all changes to the source control system, so that the changes may be reversed if necessary. Each developer is assigned one or more parts of the system code, and only that developer can alter those sections of the code while it is ``checked out'' (that is, while the code is under his or her control).

Organizationally, a quality assurance department runs rigorous tests on each new version of the operating system and reports any bugs. The developers fix these bugs as reported. A complex system of statistical analysis is used to ensure that a certain percentage of bugs are fixed before the next release, and that the operating system as a whole passes certain release criteria.

The software company, quite reasonably, must have quantitative proof that the next revision of the operating system is ready to be shipped; hence, the gathering and analysis of statistics about the performance of the operating system. It is a big job to develop a commercial UNIX system, often large enough to employ hundreds, if not thousands, of programmers, testers, documenters, and administrative personnel. Of course, no two commercial UNIX vendors are alike, but that is the general picture.

The Linux model of software development discards the entire concept of organized development, source code control systems, structured bug reporting, and statistical quality control. Linux is, and likely always will be, a hacker's operating system. (By hacker, I mean a feverishly dedicated programmer who enjoys exploiting computers and does interesting things with them. This is the original definition of the term, in contrast to the connotation of hacker as a computer wrongdoer, or outlaw.)

There is no single organization responsible for developing Linux. Anyone with enough know-how has the opportunity to help develop and debug the kernel, port new software, write documentation, and help new users. For the most part, the Linux community communicates via mailing lists and Usenet newsgroups. Several conventions have sprung up around the development effort. Anyone who wishes to have their code included in the ``official'' kernel, mails it to Linus Torvalds. He will test and include the code in the kernel as long as it doesn't break things or go against the overall design of the system.

The system itself is designed using an open-ended, feature-minded approach. The number of new features and critical changes to the system has recently diminished, and the general rule is that a new version of the kernel will be released every few weeks. Of course, this is a rough figure. New release criteria include the number of bugs to be fixed, feedback from users testing pre-release versions of the code, and the amount of sleep Linus Torvalds has had this week.

Suffice it to say that not every bug is fixed, nor is every problem ironed out between releases. As long as the revision appears to be free of critical or recurring bugs, it is considered to be stable, and the new version is released. The thrust behind Linux development is not to release perfect, bug-free code: it is to develop a free UNIX implementation. Linux is for the developers, more than anyone else.

Anyone who has a new feature or software application generally makes it available in an **alpha version**--that is, a test version, for those brave users who want to hash out problems in the initial code. Because the Linux community is largely based on the Internet, alpha software is usually uploaded to one or more Linux FTP sites, and a message is posted to one of the Linux Usenet newsgroups about how to obtain and test the code. Users who download and test alpha software can then mail results, bug fixes, and questions to the author.

After the initial bugs have been fixed, the code enters a **beta test** stage, in which it is usually considered stable but not complete. It works, but not all of the features may be present. The software may also go directly to a final stage, in which the software is considered complete and usable.

Keep in mind that these are only conventions--not rules. Some developers may feel so confident of their software that they decide it isn't necessary to release alpha or test versions. It is always up to the developer to make these decisions.

You might be amazed at how such an unstructured system of volunteers who program and debug a complete UNIX system gets anything done at all. As it turns out, this is one of the most efficient and motivated development efforts ever employed. The entire Linux kernel is written from scratch, without code from proprietary sources. It takes a huge amount of work to port all the free software under the sun to Linux. Libraries are written and ported, file systems are developed, and hardware drivers are written for many popular devices--all due to the work of volunteers.

Linux software is generally released as a **distribution**, a set of prepackaged software which comprises an entire system. It would be difficult for most users to build a complete system from the ground up, starting with the kernel, adding utilities, and installing all of the necessary software by hand. Instead, many software distributions are available which include everything necessary to install and run a complete system. There is no single, standard distribution--there are many, and each has its own advantages and disadvantages. We describe installation of the various Linux distributions starting on page .

# Differences between Linux and other operating systems.

It is important to understand the differences between Linux and other operating systems, like MS-DOS, OS/2, and the other implementations of UNIX for personal computers. First of all, Linux coexists happily with other operating systems on the same machine: you can run MS-DOS and OS/2 along with Linux on the same system without problems. There are even ways to interact between various operating systems, as we'll see.

##### Why use Linux?

Why use Linux, instead of a well known, well tested, and well documented commercial operating system? We could give you a thousand reasons. One of the most important, however, is that Linux is an excellent choice for personal UNIX computing. If you're a UNIX software developer, why use MS-DOS at home? Linux allows you to develop and test UNIX software on your PC, including database and X Window System applications. If you're a student, chances are that your university computing systems run UNIX. You can run your own UNIX system and tailor it to your needs. Installing and running Linux is also an excellent way to learn UNIX if you don't have access to other UNIX machines.

But let's not lose sight. Linux isn't only for personal UNIX users. It is robust and complete enough to handle large tasks, as well as distributed computing needs. Many businesses--especially small ones--have moved their systems to Linux in lieu of other UNIX based, workstation environments. Universities have found that Linux is perfect for teaching courses in operating systems design. Large, commercial software vendors have started to realize the opportunities which a free operating system can provide.

##### Linux vs. MS-DOS.

It's not uncommon to run both Linux and MS-DOS on the same system. Many Linux users rely on MS-DOS for applications like word processing. Linux provides its own analogs for these applications, but you might have a good reason to run MS-DOS as well as Linux. If your dissertation is written using WordPerfect for MS-DOS, you may not be able to convert it easily to TeX or some other format. Many commercial applications for MS-DOS aren't available for Linux yet, but there's no reason that you can't use both.

MS-DOS does not fully utilize the functionality of 80386 and 80486 processors. On the other hand, Linux runs completely in the processor's protected mode, and utilizes all of its features. You can directly access all of your available memory (and beyond, with virtual RAM). Linux provides a complete UNIX interface which is not available under MS-DOS. You can easily develop and port UNIX applications to Linux, but under MS-DOS you are limited to a subset of UNIX functionality.

Linux and MS-DOS are different entities. MS-DOS is inexpensive compared to other commercial operating systems and has a strong foothold in the personal computer world. No other operating system for the personal computer has reached the level of popularity of MS-DOS, because justifying spending $1,000 for other operating systems alone is unrealistic for many users. Linux, however, is free, and you may finally have the chance to decide for yourself.

You can judge Linux vs. MS-DOS based on your expectations and needs. Linux is not for everybody. If you always wanted to run a complete UNIX system at home, without the high cost of other UNIX implementations for personal computers, Linux may be what you're looking for.

##### Linux vs. The Other Guys.

A number of other advanced operating systems have become popular in the PC world. Specifically, IBM's OS/2 and Microsoft Windows have become popular for users upgrading from MS-DOS.

Both OS/2 and Windows NT are full featured multitasking operating systems, like Linux. OS/2, Windows NT, and Linux support roughly the same user interface, networking, and security features. However, the real difference between Linux and The Other Guys is the fact that Linux is a version of UNIX, and benefits from contributions of the UNIX community at large.

What makes UNIX so important? Not only is it the most popular operating system for multiuser machines, it is a foundation of the free software world. Much of the free software available on the Internet is written specifically for UNIX systems.

There are many implementations of UNIX from many vendors. No single organization is responsible for its distribution. There is a large push in the UNIX community for standardization in the form of open systems, but no single group controls this design. Any vendor (or, as it turns out, any hacker) may develop a standard implementation of UNIX.

OS/2 and Microsoft operating systems, on the other hand, are proprietary. The interface and design are controlled by a single corporation, which develops the operating system code. In one sense, this kind of organization is beneficial because it sets strict standards for programming and user interface design, unlike those found even in the open systems community.

Several organizations have attempted the difficult task of standardizing the UNIX programming interface. Linux, in particular, is mostly compliant with the POSIX.1 standard. As time goes by, it is expected that the Linux system will adhere to other standards, but standardization is not the primary goal of Linux development.

##### Linux vs. other implementations of UNIX.

Several other implementations of UNIX exist for 80386 or better personal computers. The 80386 architecture lends itself to UNIX, and vendors have taken advantage of this.

Oher implementations of UNIX for the personal computer are similar to Linux. Almost all commercial versions of UNIX support roughly the same software, programming environment, and networking features. However, there are differences between Linux and commercial versions of UNIX.

Linux supports a different range of hardware than commercial implementations. In general, Linux supports most well-known hardware devices, but support is still limited to hardware which the developers own. Commercial UNIX vendors tend to support more hardware at the outset, but the list of hardware devices which Linux supports is expanding continuously. We'll cover the hardware requirements for Linux in Section .

Many users report that Linux is at least as stable as commercial UNIX systems. Linux is still under development, but the two-pronged release philosophy has made stable versions available without impeding development.

The most important factor for many users is price. Linux software is free if you can download it from the Internet or another computer network. If you do not have Internet access, you can still purchase Linux inexpensively via mail order on diskette, tape, or CD-ROM.

Of course, you may copy Linux from a friend who already has the software, or share the purchase cost with someone else. If you plan to install Linux on a large number of machines, you need only purchase a single copy of the software--Linux is not distributed with a ``single machine'' license.

The value of commercial UNIX implementations should not be demeaned. In addition to the price of the software itself, one often pays for documentation, support, and quality assurance. These are very important factors for large institutions, but personal computer users may not require these benefits. In any case, many businesses and universities have found that running Linux in a lab of inexpensive personal computers is preferable to running a commercial version of UNIX in a lab of workstations. Linux can provide workstation functionality on a personal computer at a fraction of the cost.

Linux systems have travelled the high seas of the North Pacific, and manage telecommunications and data analysis for an oceanographic research vessel. Linux systems are used at research stations in Antarctica. Several hospitals maintain patient records on Linux systems.

Other free or inexpensive implementations of UNIX are available for the 80386 and 80486. One of the best known is 386BSD, an implementation of BSD UNIX for the 80386. The 386BSD package is comparable to Linux in many ways, but which one is better depends on your needs and expectations. The only strong distinction we can make is that Linux is developed openly, and any volunteer can aid in the development process, while 386BSD is developed by a closed team of programmers. Because of this, serious philosophical and design differences exist between the two projects. The goal of Linux is to develop a complete UNIX system from scratch (and have a lot of fun in the process), and the goal of 386BSD is in part to modify the existing BSD code for use on the 80386.

NetBSD is another port of the BSD NET/2 distribution to several machines, including the 80386. NetBSD has a slightly more open development structure, and is comparable to 386BSD in many respects.

Another project of note is HURD, an effort by the Free Software Foundation to develop and distribute a free version of UNIX for many platforms. Contact the Free Software Foundation for more information about this project. At the time of this writing, HURD is still under development.

Other inexpensive versions of UNIX exist as well, like Minix, an academic but useful UNIX clone upon which early development of Linux was based. Some of these implementations are mostly of academic interest, while others are full fledged systems.

**FEATURES OF LINUX**

# 

# System features.

Linux supports features found in other implementations of UNIX, and many which aren't found elsewhere. In this section, we'll take a nickel tour of the features of the Linux kernel.

Linux is a complete multitasking, multiuser operating system, as are all other versions of UNIX. This means that many users can log into and run programs on the same machine simultaneously.

The Linux system is mostly compatible with several UNIX standards (inasmuch as UNIX has standards) at the source level, including IEEE POSIX.1, UNIX System V, and Berkely System Distribution UNIX. Linux was developed with source code portability in mind, and it's easy to find commonly used features that are shared by more than one platform. Much of the free UNIX software available on the Internet and elsewhere compiles under Linux ``right out of the box.'' In addition, all of the source code for the Linux system, including the kernel, device drivers, libraries, user programs, and development tools, is freely distributable.

Other specific internal features of Linux include POSIX job control (used by shells like csh and bash), pseudoterminals ( pty devices), and support for dynamically loadable national or customized keyboard drivers. Linux supports **virtual consoles** that let you switch between login sessions on the same system console. Users of the screen program will find the Linux virtual console implementation familiar.

The kernel can emulate 387-FPU instructions, and systems without a math coprocessor can run programs that require floating-point math capability.

Linux supports various file systems for storing data, like the ext2 file system, which was developed specifically for Linux. The Xenix and UNIX System V file systems are also supported, as well as the Microsoft MS-DOS and Windows 95 VFAT file systems on a hard drive or floppy. The ISO 9660 CD-ROM file system is also supported.

Linux provides a complete implementation of TCP/IP networking software. This includes device drivers for many popular Ethernet cards, SLIP (Serial Line Internet Protocol) and PPP (Point-to-Point Protocol), which provide access to a TCP/IP network via a serial connection, PLIP (Parallel Line Internet Protocol), and NFS (Network File System). The complete range of TCP/IP clients and services is also supported, which includes FTP, telnet, NNTP, and SMTP.

The Linux kernel is developed to use protected-mode features of Intel 80386 and better processors. In particular, Linux uses the protected-mode, descriptor based, memory-management paradigm, and other advanced features. Anyone familiar with 80386 protected-mode programming knows that this chip was designed for multitasking systems like UNIX. Linux exploits this functionality.

The kernel supports demand-paged, loaded executables. Only those segments of a program which are actually in use are read into memory from disk. Also, copy-on-write pages are shared among executables. If several instances of a program are running at once, they share physical memory, which reduces overall usage.

In order to increase the amount of available memory, Linux also implements disk paging. Up to one gigabyte of **swap space** may be allocated on disk (upt to 8 partitions of 128 megabytes each). When the system requires more physical memory, it swaps inactive pages to disk, letting you run larger applications and support more users. However, swapping data to disk is no substitute for physical RAM, which is much faster.

The Linux kernel also implements a unified memory pool for user programs and disk cache. All free memory is used by the cache, which is reduced when running large programs.

Executables use dynamically linked, shared libraries: code from a single library on disk. This is not unlike the SunOS shared library mechanism. Executable files occupy less disk space, especially those which use many library functions. There are also statically linked libraries for object debugging and maintaining ``complete'' binary files when shared libraries are not installed. The libraries are dynamically linked at run time, and the programmer can use his or her own routines in place of the standard library routines.

To facilitate debugging, the kernel generates core dumps for post-mortem analysis. A core dump and an executable linked with debugging support allows a developer to determine what caused a program to crash.

# Software features.

Virtually every utility one would expect of a standard UNIX implementation has been ported to Linux, including basic commands like ls, awk, tr, sed, bc, and more. The familiar working environment of other UNIX systems is duplicated on Linux. All standard commands and utilities are included.

Many text editors are available, including vi, ex, pico, jove, and GNU emacs, and variants like Lucid emacs, which incorporates extensions of the X Window System, and joe. The text editor you're accustomed to using has more than likely been ported to Linux.

The choice of a text editor is an interesting one. Many UNIX users prefer ``simple'' editors like vi. (The original author wrote this book with vi.) But vi has many limitations due to its age, and modern editors like emacs have gained popularity. emacs supports a complete, Lisp based macro language and interpreter, powerful command syntax, and other extensions. There are emacs macro packages which let you read electronic mail and news, edit directory contents, and even engage in artificially intelligent psychotherapy sessions (indispensible for stressed-out Linux hackers).

Most of the basic Linux utilities are GNU software. GNU utilities support advanced features that are not found in the standard versions of BSD and UNIX System Vprograms. For example, the GNU vi clone, elvis, includes a structured macro language that differs from the original implementation. However, GNU utilities are intended to remain compatible with their BSD and System V counterparts. Many people consider the GNU versions to be superior to the originals.

A **shell** is a program which reads and executes commands from the user. In addition, many shells provide features like **job control,** managing several processes at once, input and output redirection, and a command language for writing **shell scripts**. A shell script is a program in the shell's command language and is analogous to a MS-DOS batch file.

Many types of shells are available for Linux. The most important difference between shells is the command language. For example, the C Shell (csh) uses a command language similar to the C programming language. The classic Bourne Shell sh uses another command language. The choice of a shell is often based on the command language it provides, and determines, to a large extent, the qualities of your working environment under Linux.

The GNU Bourne Again Shell (bash) is a variation of the Bourne Shell which includes many advanced features like job control, command history, command and filename completion, an emacs-like interface for editing command lines, and other powerful extensions to the standard Bourne Shell language. Another popular shell is tcsh, a version of the C Shell with advanced functionality similar to that found in bash. Other shells include zsh, a small Bourne-like shell; the Korn Shell (ksh); BSD's ash; and rc, the Plan 9 shell.

If you're the only person using the system and refer to use vi and bash exclusively as your editor and shell, there's no reason to install other editors or shells. This ``do it yourself'' attitude is prevalent among Linux hackers and users.

## Text processing and word processing.

Almost every computer user needs a method of preparing documents. In the world of personal computers, **word processing** is the norm: editing and manipulating text in a ``What-You-See-Is-What-You-Get'' (WYSIWYG) environment and producing printed copies of the text, complete with graphics, tables, and ornamentation.

Commercial word processors from Corel, Applix, and Star Division are available in the UNIX world, but **text processing,** which is quite different conceptually, is more common. In text processing systems, text is entered in a **page-description language,** which describes how the text should be formatted. Rather than enter text within a special word processing environment, you can modify text with any editor, like vi or emacs. Once you finish entering the source text (in the typesetting language), a separate program converts the source to a format suitable for printing. This is somewhat analogous to programming in a language like C, and ``compiling'' the document into printable form.

Many text processing systems are available for Linux. One is groff, the GNU version of the classic troff text formatter originally developed by Bell Labs and still used on many UNIX systems worldwide. Another modern text processing system is TeX, developed by Donald Knuth of computer science fame. Dialects of TeX, like LaTeX, are also available.

Text processors like TeX and groff differ mostly in the syntax of their formatting languages. The choice of one formatting system over another is based upon what utilities are available to satisfy your needs, as well as personal taste.

Many people consider groff's formatting language to be a bit obscure and use find TeX more readable. However, groff produces ASCII output which can be viewed on a terminal more easily, while TeX is intended primarily for output to a printing device. Various add-on programs are required to produce ASCII output from TeX formatted documents, or convert TeX input to groff format.

Another program is texinfo, an extension to TeX which is used for software documentation developed by the Free Software Foundation. texinfo can produce printed output, or an online-browsable hypertext ``Info'' document from a single source file. Info files are the main format of documentation used in GNU software like emacs.

Text processors are used widely in the computing community for producing papers, theses, magazine articles, and books. (This book is produced using LaTeX.) The ability to process source language as a text file opens the door to many extensions of the text processor itself. Because a source document is not stored in an obscure format that only one word processor can read, programmers can write parsers and translators for the formatting language, and thus extend the system.

What does a formatting language look like? In general, a formatted source file consists mostly of the text itself, with **control codes** to produce effects like font and margin changes, and list formatting.

Consider the following text:

Mr. Torvalds:

We are very upset with your current plans to implement post-hypnotic suggestions in the **Linux** terminal driver code. We feel this way for three reasons:

1. Planting subliminal messages in the terminal driver is not only immoral, it is a waste of time;
2. It has been proven that ``post-hypnotic suggestions'' are ineffective when used upon unsuspecting UNIX hackers;
3. We have already implemented high-voltage electric shocks, as a security measure, in the code for login.

We hope you will reconsider.

This text might appear in the LaTeX formatting language as the following:

\begin{quote}

Mr. Torvalds:

We are very upset with your current plans to implement

{\em post-hypnotic suggestions\/} in the {\bf Linux} terminal

driver code. We feel this way for three reasons:

\begin{enumerate}

\item Planting subliminal messages in the kernel driver is not only

immoral, it is a waste of time;

\item It has been proven that ``post-hypnotic suggestions''

are ineffective when used upon unsuspecting UNIX hackers;

\item We have already implemented high-voltage electric shocks, as

a security measure, in the code for {\tt login}.

\end{enumerate}

We hope you will reconsider.

\end{quote}

The author enters the text using any text editor and generates formatted output by processing the source with LaTeX. At first glance, the typesetting language may appear to be obscure, but it's actually quite easy to understand. Using a text processing system enforces typographical standards when writing. All the enumerated lists within a document will look the same, unless the author modifies the definition of an enumerated list. The goal is to allow the author to concentrate on the text, not typesetting conventions.

When writing with a text editor, one generally does not think about how the printed text will appear. The writer learns to visualize the finished text's appearance from the formatting commands in the source.

WYSIWYG word processors are attractive for many reasons. They provide an easy-to-use visual interface for editing documents. But this interface is limited to aspects of text layout which are accessible to the user. For example, many word processors still provide a special format language for producing complicated expressions like mathematical formulae. This is text processing, albeit on a much smaller scale.

A not-so-subtle benefit of text processing is that you specify exactly which format you need. In many cases, the text processing system requires a format specification. Text processing systems also allow source text to be edited with any text editor, instead of relying on format codes which are hidden beneath a word processor's opaque user interface. Further, the source text is easily converted to other formats. The tradeoff for this flexibility and power is the lack of WYSIWYG formatting.

Some programs let you preview the formatted document on a graphics display device before printing. The xdvi program displays a ``device independent'' file generated by the TeX system under X. Applications like xfig and gimp provide WYSIWYG graphics interfaces for drawing figures and diagrams, which are subsequently converted to text processing language for inclusion in your document.

Text processors like troff were around long before WYSIWYG word processing was available. Many people still prefer their versatility and independence from a graphics environment.

Many text-processing-related utilities are available. The powerful METAFONT system, which is used to design fonts for TeX, is included in the Linux port of TeX. Other programs include ispell, an interactive spelling checker and corrector; makeindex, which generates indices in LaTeX documents; and many other groff and TeXbased macro packages which format many types of technical and mathematical texts. Conversion programs that translate between TeX or groff source to a myriad of other formats are also available.

A newcomer to text formatting is YODL, written by Karel Kubat. YODL is an easy-to-learn language with filters to produce various output formats, like LaTeX, SGML, and HTML.

## Programming languages and utilities.

Linux provides a complete UNIX programming environment which includes all of the standard libraries, programming tools, compilers, and debuggers which you would expect of other UNIX systems.

Standards like POSIX.1 are supported, which allows software written for Linux to be easily ported to other systems. Professional UNIX programmers and system administrators use Linux to develop software at home, then transfer the software to UNIX systems at work. This not only saves a great deal of time and money, but also lets you work in the comfort of your own home. (One of the authors uses his system to develop and test X Window System applications at home, which can be directly compiled on workstations elsewhere.) Computer Science students learn UNIX programming and explore other aspects of the system, like kernel architecture.

With Linux, you have access to the complete set of libraries and programming utilities and the complete kernel and library source code.

Within the UNIX software world, systems and applications are often programmed in C or C++. The standard C and C++ compiler for Linux is GNU gcc, which is an advanced, modern compiler that supports C++, including AT&T 3.0 features, as well as Objective-C, another object-oriented dialect of C.

Besides C and C++, other compiled and interpreted programming languages have been ported to Linux, like Smalltalk, FORTRAN, Java, Pascal, LISP, Scheme, and Ada (if you're masochistic enough to program in Ada, we aren't going to stop you). In addition, various assemblers for writing protected-mode 80386 code are available, as are UNIX hacking favorites like Perl (the script language to end all script languages) and Tcl/Tk (a shell-like command processing system which has support for developing simple X Window System applications).

The advanced gdb debugger can step through a program one line of source code at a time, or examine a core dump to find the cause of a crash. The gprof profiling utility provides performance statistics for your program, telling you where your program spends most of its execution time. As mentioned above, the emacs text editor provides interactive editing and compilation environments for various programming languages. Other tools include GNU make and imake, which manage compilation of large applications, and RCS, a system for source code locking and revision control.

Finally, Linux supports dynamically linked, shared libraries (DLLs), which result in much smaller binaries. The common subroutine code is linked at run-time. These DLLs let you override function definitions with your own code. For example, if you wish to write your own version of the malloc() library routine, the linker will use your new routine instead of the one in the libraries.

## 

## Introduction to the X Window System.

The X Window System, or simply X, is a standard graphical user interface (GUI) for UNIX machines and is a powerful environment which supports many applications. Using the X Window System, you can have multiple terminal windows on the screen at once, each having a different login session. A pointing device like a mouse is often used with X, although it isn't required.

Many X-specific applications have been written, including games, graphics and programming utilities, and documentation tools. Linux and X make your system a bona fide workstation. With TCP/IP networking, your Linux machine can display X applications running on other machines.

The X Window System was originally developed at the Massachusetts Institute of Technology and is freely distributable. Many commercial vendors have distributed proprietary enhancements to the original X Window System as well. The version of X for Linux is XFree86, a port of X11R6 which is freely distributable. XFree86 supports a wide range of video hardware, including VGA, Super VGA, and accelerated video adaptors. XFree86 is a complete distribution of the X Windows System software, and contains the X server itself, many applications and utilities, programming libraries, and documents.

Standard X applications include xterm, a terminal emulator used for most text-based applications within a window, xdm, which handles logins, xclock, a simple clock display, xman, a X-based manual page reader, and xmore. The many X applications available for Linux are too numerous to mention here, but their number includes spreadsheets, word processors, graphics programs, and web browsers like the Netscape Navigator. Many other applications are available separately. Theoretically, any application written for X should compile cleanly under Linux.

The interface of the X Window System is controlled largely by the **window manager**. This user-friendly program is in charge of the placement of windows, the user interface for resizing and moving them, changing windows to icons, and the appearance of window frames, among other tasks. XFree86 includes twm, the classic MIT window manager, and advanced window managers like the Open Look Virtual Window Manager (olvwm) are available. Popular among Linux users is fvwm--a small window manager that requires less than half the memory of twm. It provides a 3-dimensional appearance for windows and a virtual desktop. The user moves the mouse to the edge of the screen, and the desktop shifts as though the display were much larger than it really is. fvwm is greatly customizable and allows access to functions from the keyboard as well as mouse. Many Linux distributions use fvwm as the standard window manager. A version of fvwm called fvwm95-2 offers Microsoft Windows 95-like look and feel.

The XFree86 distribution includes programming libraries for wily programmers who wish to develop X applications. Widget sets like Athena, Open Look, and Xaw3D are supported. All of the standard fonts, bitmaps, manual pages, and documentation are included. PEX (a programming interface for 3-dimensional graphics) is also supported.

Many X application programmers use the proprietary Motif widget set for development. Several vendors sell single and multiple user licenses for binary versions of Motif. Because Motif itself is relatively expensive, not many Linux users own it. However, binaries statically linked with Motif routines can be freely distributed. If you write a program using Motif, you may provide a binary so users without the Motif libraries can use the program.

A major caveat to using the X Window System is its hardware requirements. A 80386-based CPU with 4 megabytes of RAM is capable of running X, but 16 megabytes or more of physical RAM is needed for comfortable use. A faster processor is nice to have as well, but having enough physical RAM is much more important. In addition, to achieve really slick video performance, we recommend getting an accelerated video card, like a VESA Local Bus (VLB) S3 chipset card. Performance ratings in excess of 300,000 xstones have been achieved with Linux and XFree86. Using adequate hardware, you'll find that running X and Linux is as fast, or faster, than running X on other UNIX workstations.

1. **Introduction to Networking.**

Would you like to communicate with the world? Linux supports two primary UNIX networking protocols: TCP/IP and UUCP. TCP/IP (Transmission Control Protocol/Internet Protocol) is the networking paradigm which allows systems all over the world to communicate on a single network, the **Internet.** With Linux, TCP/IP, and a connection to the Internet, you can communicate with users and machines via electronic mail, Usenet news, and FTP file transfer.

Most TCP/IP networks use Ethernet as the physical network transport. Linux supports many popular Ethernet cards and interfaces for personal computers, including pocket and PCMCIA Ethernet adaptors.

However, because not everyone has an Ethernet connection at home, Linux also supports **SLIP** (Serial Line Internet Protocol) and **PPP** (Point-to-Point Protocol), which provide Internet access via modem. Many businesses and universities provide SLIP and PPP servers. In fact, if your Linux system has an Ethernet connection to the Internet and a modem, your system can become a SLIP or PPP server for other hosts.

NFS (Network File System) lets your system seamlessly share file systems with other machines on the network. FTP (File Transfer Protocol) lets you transfer files with other machines. sendmail sends and receives electronic mail via the SMTP protocol; C-News and INN are NNTP based new systems; and telnet, rlogin, and rsh let you log in and execute commands on other machines on the network. finger lets you get information about other Internet users.

Linux also supports Microsoft Windows connectivity via Samba[gif](http://www.tldp.org/LDP/gs/footnode.html#319), and Macintosh connectivity with AppleTalk and LocalTalk. Support for Novell's IPX protocol is also included.

The full range of mail and news readers is available for Linux, including elm, pine, rn, nn, and tin. Whatever your preference, you can configure a Linux system to send and receive electronic mail and news from all over the world.

The system provides a standard UNIX socket programming interface. Virtually any program that uses TCP/IP can be ported to Linux. The Linux X server also supports TCP/IP, and applications running on other systems may use the display of your local system.

UUCP (UNIX-to-UNIX Copy) is an older mechanism to transfer files, electronic mail, and electronic news between UNIX machines. Historically, UUCP machines are connected over telephone lines via modem, but UUCP is able to transfer data over a TCP/IP network as well. If you do not have access to a TCP/IP network or a SLIP or PPP server, you can configure your system to send and receive files and electronic mail using UUCP.

## Telecommunications and BBS software.

If you have a modem, you'll be able to communicate with other machines via telecommunications packages available for Linux. Many people use telecommunications software to access bulletin board systems (BBS's) as well as commercial, online services like Prodigy, CompuServe, and America Online. People use modems to connect to UNIX systems at work or school. Modems can send and receive faxes.

A popular communications package for Linux is seyon, which provides a customizable, ergonomic interface undex X and has built-in support for the Kermit and ZModem file transfer protocols. Other telecommunications programs include C-Kermit, pcomm, and minicom. These are similar to communications programs found on other operating systems, and are quite easy to use.

If you do not have access to a SLIP or PPP server (see the previous section), you can use term to multiplex your serial line. The term program allows you to open more than one login session over a modem connection. It lets you redirect X client connections to your local X server via a serial line. Another software package, KA9Q, implements a similar, SLIP-like interface.

Operating a Bulletin Board System (BBS) is a favorite hobby and means of income for many people. Linux supports a wide range of BBS software, most of which is more powerful than that available for other operating systems. With a phone line, modem, and Linux, you can turn your system into a BBS and provide dial-in access for users worldwide. BBS software for Linux includes XBBS and UniBoard BBS packages.

Most BBS software locks the user into a menu based system where only certain functions and applications are available. An alternative to BBS access is full UNIX access, which lets users dial into your system and log in normally. This requires a fair amount of maintenance by the system administrator, but providing public UNIX access is not difficult. In addition to TCP/IP networking, you can make electronic mail and news access available on your system.

If you do not have access to a TCP/IP network or UUCP feed, Linux lets you communicate with BBS networks like FidoNet, which let you exchange electronic news and mail over a telephone line.

## World Wide Web.

It is worth noting that Linux includes web server software as well as web browsers. The most common server is Apache. Thousands of Linux systems run Apache on the Internet today, including the Linux Resources site, www.linuxresources.com.

Linux distributions include different web browsers, and other browsers can be downloaded from the Internet. Available browsers include Lynx, Mosaic, Netscape, Arena, and Amaya.

Linux provides complete support for Java and CGI applets, and Perl is a standard tool in the Linux programming environment.

## Interfacing and MS-DOS.

Various utilities exist to interface with MS-DOS. The most well-known application is the Linux MS-DOS Emulator, which lets you run MS-DOS applications directly from Linux. Although Linux and MS-DOS are completely different operating systems, the 80386 protected-mode environment allows MS-DOS applications to behave as if they were running in their native 8086 environment.

The MS-DOS emulator is still under development, but many popular applications run under it. Understandably, MS-DOS applications that use bizarre or esoteric features of the system may never be supported, because of the limitations inherent in any emulator. For example, you shouldn't expect to run programs that use 80386 protected-mode features, like Microsoft Windows (in 386 enhanced mode, that is).

Standard MS-DOS commands and utilities like PKZIP.EXE work under the emulators, as do 4DOS, a COMMAND.COM replacement, FoxPro 2.0, Harvard Graphics, MathCad, Stacker 3.1, Turbo Assembler, Turbo C/C++, Turbo Pascal, Microsoft Windows 3.0 (in real mode), and WordPerfect 5.1.

The MS-DOS Emulator is meant mostly as an ad-hoc solution for those who need MS-DOS for only a few applications and use Linux for everything else. It's not meant to be a complete implementation of MS-DOS. Of course, if the Emulator doesn't satisfy your needs, you can always run MS-DOS as well as Linux on the same system. Using the LILO boot loader, you can specify at boot time which operating system to start. Linux can also coexist with other operating systems, like OS/2.

Linux provides a seamless interface to transfer files between Linux and MS-DOS. You can mount a MS-DOS partition or floppy under Linux, and directly access MS-DOS files as you would any file.

Currently under development is **WINE**--a Microsoft Windows emulator for the X Window System under Linux. Once WINE is complete, users will be able to run MS-Windows applications directly from Linux. This is similar to the commercial WABI Windows emulator from Sun Microsystems, which is also available for Linux.

## Other applications.

A host of miscellaneous programs and utilities exist for Linux, as one would expect of such a hodgepodge operating system. Linux's primary focus is UNIX personal computing, but this is not the only field where it excels. The selection of business and scientific software is expanding, and commercial software vendors have begun to contribute to the growing pool Linux applications.

Several relational databases are available for Linux, including Postgres, Ingres, and Mbase. These are full-featured, professional, client/server database applications, similar to those found on other UNIX platforms. Many commercial database systems are available as well.

Scientific computing applications include FELT (finite element analysis); gnuplot (data plotting and analysis); Octave (a symbolic mathematics package similar to MATLAB); xspread (a spreadsheet calculator); xfractint (an X-based port of the popular Fractint fractal generator); and xlispstat (statistics). Other applications include SPICE (circuit design and analysis) and Khoros (image and digital signal processing and visualization). Commercial packages like Maple and MathLab are available.

Many more applications have been ported to Linux. If you absolutely cannot find what you need, you can attempt to port the application from another platform to Linux yourself. Whatever your field, porting standard UNIX applications to Linux is straightforward. Linux's complete UNIX programming environment is sufficient to serve as the base for any scientific application.

Linux also has its share of games. These include classic text based dungeon games like Nethack and Moria; **MUDs** (multi-user dungeons, which allow many users to interact in a text-based adventure) like DikuMUD and TinyMUD; and a slew of X games like xtetris, netrek, and xboard, the X11 version of gnuchess. The popular shoot-em-up, arcade-style game, Doom, has also been ported to Linux.

For audiophiles, Linux supports various sound cards and related software, like CDplayer, which makes a CD-ROM drive into an audio CD player, MIDI sequencers and editors, which let you compose music for playback through a synthesizer or other MIDI controlled instrument, and sound editors for digitized sounds.

Can't find the application you're looking for? The Linux Software Map, lists software packages which have been written or ported to Linux. Another way to find Linux applications is to look at the INDEX files found on Linux FTP sites, if you have Internet access.

Most freely-distributable, UNIX based software will compile on Linux with little difficulty. If all else fails, you can write the application yourself. If you're looking for a commercial application, there may be a free ``clone'' available. Or, you can encourage the software company to consider releasing a binary version for Linux. Several individuals have contacted software companies and asked them to port their applications to Linux, with various degrees of success.

# **A brief history of Linux**

UNIX is one of the most popular operating systems worldwide because of its large support base and distribution. It was originally developed at AT&T as a multitasking system for minicomputers and mainframes in the 1970's, but has since grown to become one of the most widely-used operating systems anywhere, despite its sometimes confusing interface and lack of central standardization.

Many hackers feel that UNIX is the Right Thing--the One True Operating System. Hence, the development of Linux by an expanding group of UNIX hackers who want to get their hands dirty with their own system.

Versions of UNIX exist for many systems, from personal computers to supercomputers like the Cray Y-MP. Most versions of UNIX for personal computers are expensive and cumbersome. At the time of this writing, a one-machine version of UNIX System V for the 386 runs about US$1500.

Linux is a free version of UNIX developed primarily by Linus Torvalds at the University of Helsinki in Finland, with the help of many UNIX programmers and wizards across the Internet. Anyone with enough know-how and gumption can develop and change the system. The Linux kernel uses no code from AT&T or any other proprietary source, and much of the software available for Linux was developed by the GNU project of the Free Software Foundation in Cambridge, Massachusetts, U.S.A. However, programmers from all over the world have contributed to the growing pool of Linux software.

Linux was originally developed as a hobby project by Linus Torvalds. It was inspired by Minix, a small UNIX system developed by Andy Tanenbaum. The first discussions about Linux were on the Usenet newsgroup, comp.os.minix. These discussions were concerned mostly with the development of a small, academic UNIX system for Minix users who wanted more.

The very early development of Linux mostly dealt with the task-switching features of the 80386 protected-mode interface, all written in assembly code. Linus writes,

``After that it was plain sailing: hairy coding still, but I had some devices, and debugging was easier. I started using C at this stage, and it certainly speeds up development. This is also when I started to get serious about my megalomaniac ideas to make `a better Minix than Minix.' I was hoping I'd be able to recompile gcc under Linux someday...

``Two months for basic setup, but then only slightly longer until I had a disk driver (seriously buggy, but it happened to work on my machine) and a small file system. That was about when I made 0.01 available (around late August of 1991): it wasn't pretty, it had no floppy driver, and it couldn't do much of anything. I don't think anybody ever compiled that version. But by then I was hooked, and didn't want to stop until I could chuck out Minix.''

No announcement was ever made for Linux version 0.01. The 0.01 sources weren't even executable. They contained only the bare rudiments of the kernel source and assumed that you had access to a Minix machine to compile and experiment with them.

On October 5, 1991, Linus announced the first ``official'' version of Linux, which was version 0.02. At that point, Linus was able to run bash (the GNU Bourne Again Shell) and gcc (the GNU C compiler), but not much else. Again, this was intended as a hacker's system. The primary focus was kernel development--user support, documentation, and distribution had not yet been addressed. Today, the Linux community still seems to treat these issues as secondary to ``real programming''--kernel development.

As Linus wrote in comp.os.minix,

``Do you pine for the nice days of Minix-1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on an OS you can try to modify for your needs? Are you finding it frustrating when everything works on Minix? No more all-nighters to get a nifty program working? Then this post might be just for you.

``As I mentioned a month ago, I'm working on a free version of a Minix-look-alike for AT-386 computers. It has finally reached the stage where it's even usable (though may not be, depending on what you want), and I am willing to put out the sources for wider distribution. It is just version 0.02...but I've successfully run bash, gcc, gnu-make, gnu-sed, compress, etc. under it.''

After version 0.03, Linus bumped up the version number to 0.10, as more people started to work on the system. After several further revisions, Linus increased the version number to 0.95 in March, 1992, to reflect his expectation that the system was ready for an ``official'' release soon. (Generally, software is not assigned the version number 1.0 until it is theoretically complete or bug-free.). Almost a year and a half later, in late December of 1993, the Linux kernel was still at version 0.99.pl14--asymptotically approaching 1.0. At the time of this writing, the current stable kernel version is 2.0 patchlevel 33, and version 2.1 is under development.

Most of the major, free UNIX software packages have been ported to Linux, and commercial software is also available. More hardware is supported than in the original kernel versions. Many people have executed benchmarks on 80486 Linux systems and found them comparable with mid-range workstations from Sun Microsystems and Digital Equipment Corporation. Who would have ever guessed that this ``little'' UNIX clone would have grown up to take on the entire world of personal computing?

# **PLATFORM OR Hardware requirements**

Keep in mind that Linux is developed by users. This means, for the most part, that the hardware supported by Linux is that which the users and developers have access to. As it turns out, most popular hardware and peripherals for personal computers are supported. Linux supports more hardware than some commercial implementations of UNIX. However, some obscure devices aren't supported yet.

Another drawback of hardware support under Linux is that many companies keep their hardware interfaces proprietary. Volunteer Linux developers can't write drivers for the devices because the manufacturer does not make the technical specifications public. Even if Linux developers could develop drivers for proprietary devices, they would be owned by the company which owns the device interface, which violates the GPL. Manufacturers that maintain proprietary interfaces write their own drivers for operating systems like MS-DOS and Microsoft Windows. Users and third-party developers never need to know the details of the interface.

In some cases, Linux programmers have attempted to write hackish device drivers based on assumptions about the interface. In other cases, developers work with the manufacturer and try to obtain information about the device interface, with varying degrees of success.

In the following sections, we attempt to summarize the hardware requirements for Linux. The Linux Hardware HOWTO contains a more complete listing of hardware supported by Linux.

**Disclaimer:** Much hardware support for Linux is in the development stage. Some distributions may or may not support experimental features. This section lists hardware which has been supported for some time and is known to be stable. When in doubt, consult the documentation of your Linux distribution.

Linux is available for many platforms in addition to Intel 80x86 systems. These include Macintosh, Amiga, Sun SparcStation, and Digital Equipment Corporation Alpha based systems. In this book, however, we focus on garden-variety Intel 80386, 80486, and Pentium processors, and clones by manufacturers like AMD, Cyrix, and IBM.

##### Motherboard and CPU requirements

Linux currently supports systems with the Intel 80386, 80486, or Pentium CPU, including all variations like the 80386SX, 80486SX, 80486DX, and 80486DX2. Non-Intel clones work with Linux as well. Linux has also been ported to the DEC Alpha and the Apple PowerMac.

If you have an 80386 or 80486SX, you may also wish to use a math coprocessor, although one isn't required. The Linux kernel can perform FPU emulation if the machine doesn't have a coprocessor. All standard FPU couplings are supported, including IIT, Cyrix FasMath, and Intel.

Most common PC motherboards are based on the PCI bus but also offer ISA slots. This configuration is supported by Linux, as are EISA and VESA-bus systems. IBM's MicroChannel (MCA) bus, found on most IBM PS/2 systems, is significantly different, and support has been recently added.

##### Memory requirements

Linux requires very little memory, compared to other advanced operating systems. You should have 4 megabytes of RAM at the very least, and 16 megabytes is strongly recommended. The more memory you have, the faster the system will run. Some distributions require more RAM for installation.

Linux supports the full 32-bit address range of the processor. In other words, it uses all of your RAM automatically.

Linux will run with only 4 megabytes of RAM, including bells and whistles like the X Window System and emacs. However, having more memory is almost as important as having a faster processor. For general use, 16 megabytes is enough, and 32 megabytes, or more, may be needed for systems with a heavy user load.

Most Linux users allocate a portion of their hard drive as swap space, which is used as **virtual RAM**. Even if your machine has more than 16 megabytes of physical RAM, you may wish to use swap space. It is no replacement for physical RAM, but it can let your system run larger applications by swapping inactive portions of code to disk.

##### Hard drive controller requirements

It is possible to run Linux from a floppy diskette, or, for some distributions, a live file system on CD-ROM, but for good performance you need hard disk space. Linux can co-exist with other operating systems--it only needs one or more disk partitions.

Linux supports all IDE and EIDE controllers as well as older MFM and RLL controllers. Most, but not all, ESDI controllers are supported. The general rule for non-SCSI hard drive and floppy controllers is that if you can access the drive from MS-DOS or another operating system, you should be able to access it from Linux.

Linux also supports a number of popular SCSI drive controllers. This includes most Adaptec and Buslogic cards as well as cards based on the NCR chip sets.

##### Hard drive space requirements

Of course, to install Linux, you need to have some amount of free space on your hard drive. Linux will support more than one hard drive on the same machine; you can allocate space for Linux across multiple drives if necessary.

How much hard drive space depends on your needs and the software you're installing. Linux is relatively small, as UNIX implementations go. You could run a system in 20 megabytes of disk space. However, for expansion and larger packages like X, you need more space. If you plan to let more than one person use the machine, you need to allocate storage for their files. Realistic space requirements range from 200 megabytes to one gigabyte or more.

Each Linux distribution comes with literature to help you gauge the precise amount of storage required for your software configuration.

##### Monitor and video adaptor requirements

Linux supports standard Hercules, CGA, EGA, VGA, IBM monochrome, Super VGA, and many accelerated video cards, and monitors for the default, text-based interface. In general, if the video card and monitor work under an operating system like MS-DOS, the combination should work fine under Linux. However, original IBM CGA cards suffer from ``snow'' under Linux, which is not pleasant to view.

Graphical environments like X have video hardware requirements of their own. Popular video cards are supported and new card support is added regularly.

##### Miscellaneous hardware

You may also have devices like a CD-ROM drive, mouse, or sound card, and may be interested in whether or not this hardware is supported by Linux.

##### Mice and other pointing devices

Typically, a mouse is used only in graphical environments like X. However, several Linux applications that are not associated with a graphical environment also use mice.

Linux supports standard serial mice like Logitech, MM series, Mouseman, Microsoft (2-button), and Mouse Systems (3-button). Linux also supports Microsoft, Logitech, and ATIXL bus mice, and the PS/2 mouse interface.

Pointing devices that emulate mice, like trackballs and touchpads, should work also.

##### CD-ROM drives

Many common CD-ROM drives attach to standard IDE controllers. Another common interface for CD-ROM is SCSI. SCSI support includes multiple logical units per device so you can use CD-ROM ``jukeboxes.'' Additionally, a few proprietary interfaces, like the NEC CDR-74, Sony CDU-541 and CDU-31a, Texel DM-3024, and Mitsumi are supported.

Linux supports the standard ISO 9660 file system for CD-ROMs, and the High Sierra file system extensions.

##### Tape drives

Any SCSI tape drive, including quarter inch, DAT, and 8MM are supported, if the SCSI controller is supported. Devices that connect to the floppy controller like floppy tape drives are supported as well, as are some other interfaces, like QIC-02.

##### Printers

Linux supports the complete range of parallel printers. If MS-DOS or some other operating system can access your printer from the parallel port, Linux should be able to access it, too. Linux printer software includes the UNIX standard lp and lpr software. This software allows you to print remotely via a network, if you have one. Linux also includes software that allows most printers to handle PostScript files.

##### Modems

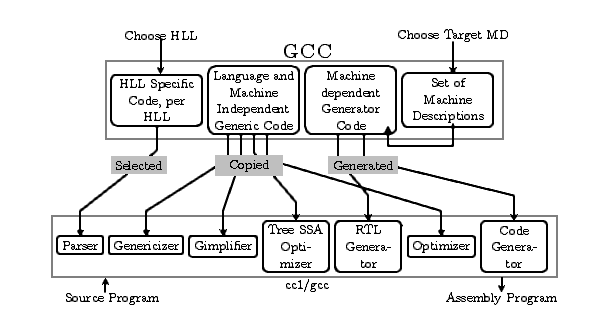
As with printer support, Linux supports the full range of serial modems, both internal and external. A great deal of telecommunications software is available for Linux, including Kermit, pcomm, minicom, and seyon. If your modem is accessible from another operating system on the same machine, you should be able to access it from Linux with no difficulty.

##### Ethernet cards

Many popular Ethernet cards and LAN adaptors are supported by Linux. Linux also supports some FDDI, frame relay, and token ring cards, and all Arcnet cards. A list of supported network cards is included in the kernel source of your distribution.

**Steps to Compile and Execute C Program in Linux Using Gcc:-**

Before talking of compiling and running C program in Linux let's see why C is so popular ever since it was created. He was the Dennis Ritchie who developed C language in 1969 to 1973. C was developed from the beginning as the system programming language for UNIX. Most of the UNIX kernel, and all of its supporting tools and libraries, were written in C. Initially, C was designed to implement the UNIX operating system. Later other folks found it useful for their programs without any hindrance, and they began using it. Even today, C is the first choice for system-level programming. This tutorial explains compilation and execution of C program is in detail.

The GCC Compiler Generation Framework (CGF) and it's use to generate the target specific compiler (cc1/gcc) components. Some components of the compiler (cc1/gcc) are *selected* from the CGF, some are *copied* from the CGF and some are *generated* from the framework.

**Compile C Program in Linux - The Classic Hello World!**

Kernighan and Ritchie (K & R) in their classic book on C programming language acquaint readers to C language by compiling and executing "Hello World!" C program as follows.

#include <stdio.h>

int main()

{

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("hello, world!**\n**");

}

*/\* helloworld.c \*/*

To compile and run this C program every part of the system has to perform in concert. In order to compile above C program in Linux, we will start right from the creation of the program. The 'Hello World!' program starts its life as a source file which is created with help of a text editor and saved as helloworld.c. The helloworld.c program code is stored in a file as a sequence of bytes. Each byte has a value corresponding to some character. The first byte has the value 35 that corresponds to the character '#', for example. Likewise, the second byte has the integer value 105, which corresponds to the character 'i', and so on. The idea illustrates that all information in a system is represented as a bunch of bits.

To compile and run the C program helloworld.c, all C statements must be translated individually into a sequence of instructions that a machine can understand. These instructions are then packaged in a form called executable object program. There are other programs which perform this task to get the program running. On a UNIX/Linux system, the translation from source code to object code (executable) is performed by a compiler driver. Here we will compile C program by gcc.

The following command (provided that gcc is installed on your Linux box) compiles C program helloworld.c and creates an executable file called helloworld. Don't forget to set appropriate permissions to helloworld.c, so that you won't get execute permission errors.

[root@host ~]# gcc helloworld.c -o helloworld

While compiling helloworld.c the *gcc* compiler reads the source file helloworld.c and translates it into an executable helloworld. The compilation is performed in four sequential phases by the compilation system (a collection of four programs - *preprocessor*, *compiler*, *assembler*, and *linker*).

Now, let's perform all four steps to compile and run C program one by one.

**Preprocessing**

During compilation of a C program the compilation is started off with preprocessing the directives (e.g., #include and #define). The preprocessor (*cpp* - c preprocessor) is a separate program in reality, but it is invoked automatically by the compiler. For example, the #include <stdio.h> command in line 1 of helloworld.c tells the preprocessor to read the contents of the system header file stdio.h and insert it directly into the program text. The result is another file typically with the .i suffix. In practice, the preprocessed file is not saved to disk unless the -save-temps option is used.

This is the first stage of compilation process where preprocessor directives (macros and header files are most common) are expanded. To perform this step gcc executes the following command internally.

[root@host ~]# cpp helloworld.c > helloworld.i

The result is a file helloworld.i that contains the source code with all macros expanded. If you execute the above command in isolation then the file helloworld.i will be saved to disk and you can see its content by vi or any other editor you have on your Linux box.

**Compilation**

In this phase compilation proper takes place. The compiler (*ccl*) translates helloworld.i into helloworld.s. File helloworld.s contains assembly code. You can explicitly tell *gcc* to translate helloworld.i to helloworld.s by executing the following command.

[root@host ~]# gcc -S helloworld.i

The command line option -S tells the compiler to convert the preprocessed code to assembly language without creating an object file. After having created helloworld.s you can see the content of this file. While looking at assembly code you may note that the assembly code contains a call to the external function printf.

**Assembly**

Here, the assembler (*as*) translates helloworld.s into machine language instructions, and generates an object file helloworld.o. You can invoke the assembler at your own by executing the following command.

[root@host ~]# as helloworld.s -o helloworld.o

The above command will generate helloworld.o as it is specified with -o option. And, the resulting file contains the machine instructions for the classic "Hello World!" program, with an undefined reference to printf.

**Linking**

This is the final stage in compilation of "Hello World!" program. This phase links object files to produce final executable file. An executable file requires many external resources (system functions, C run-time libraries etc.). Regarding our "Hello World!" program you have noticed that it calls the printf function to print the 'Hello World!' message on console. This function is contained in a separate pre compiled object file printf.o, which must somehow be merged with our helloworld.o file. The linker (*ld*) performs this task for you. Eventually, the resulting file helloworld is produced, which is an executable. This is now ready to be loaded into memory and executed by the system.

The actual link command executed by linker is rather complicated. But still, if you passionate enough you can execute the following command to produce the executable file helloworld by yourself.

[root@host ~]# ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib64/crt1.o /usr/lib64/crti.o /usr/lib64/crtn.o helloworld.o /usr/lib/gcc/x86\_64-redhat-linux/4.1.2/crtbegin.o -L /usr/lib/gcc/x86\_64-redhat-linux/4.1.2/ -lgcc -lgcc\_eh -lc -lgcc -lgcc\_eh /usr/lib/gcc/x86\_64-redhat-linux/4.1.2/crtend.o -o helloworld

And, you can greet the universe as follows:

[root@host ~]# ./helloworld

Output:

hello, world!

I executed the above command on an x86\_64 system having gcc 4.1.2. It may be the above command does not work on your system as it is. It all matters that where the libraries located?

For you, there is no need to type the complex ld command directly - the entire linking process is handled transparently by *gcc* when invoked, as follows.

[root@host ~]# gcc helloworld.c -o helloworld

During the whole compilation process there are other files also in role along with the source code file. If you see the very first statement of helloworld.c it is #include <stdio.h> (includes header file). Likewise, while compiling a C program you have to work with following types of files.

**Program Translation**

**Source code files:** These files contain high level program code which can be read and understood by programmers. Such files carry .c extension by convention.

**Header files:** These types of files contain function declarations (also known as function prototypes) and various preprocessor statements. They are used to allow source code files to access externally-defined functions. As a convention header files have .h extension.

**Object files:** These files are produced as an intermediate output by the gcc compiler during program compilation. They consist of function definitions in binary form, but they are not executable by themselves. Object files end with .o extension by convention (on UNIX like operating systems), although on some operating systems e.g., Windows, and MS-DOS they often end in .obj.

**Binary executables:** These are produced as the output of a program called a linker. During the process of compiling and running C program the linker links together a number of object files to produce a binary file which can be directly executed. Binary executables have no special suffix on UNIX like operating systems, while they generally have .exe on Windows.

Along with above four types of files, while compiling a C program you can come across .a and .so, static and shared libraries respectively, but you would not normally deal with them directly.

## Identifying files

When a source file has been compiled to an object file or executable the options used to compile it are no longer obvious. The file command looks at the contents of an object file or executable and determines some of its characteristics, such as whether it was compiled with dynamic or static linking.

For example, here is the result of the file command for a typical executable:

$ file a.out

a.out: ELF 32-bit LSB executable, Intel 80386,

version 1 (SYSV), dynamically linked (uses shared

libs), not stripped

The output shows that the executable file is dynamically linked, and compiled for the Intel 386 and compatible processors. A full explanation of the output is shown below:

ELF

The internal format of the executable file (ELF stands for "Executable and Linking Format", other formats such as COFF "Common Object File Format" are used on some older operating systems (e.g. MS-DOS)).

32-bit

The word size (for some platforms this would be 64-bit).

LSB

Compiled for a platform with *least significant byte* first word-ordering, such as Intel and AMD x86 processors (the alternative MSB *most significant byte* first is used by other processors, such as the Motorola 680x0). Some processors such as Itanium and MIPS support both LSB and MSB orderings.

Intel 80386

The processor the executable file was compiled for.

Version 1 (SYSV)

This is the version of the internal format of the file.

Dynamically linked

The executable uses shared libraries (statically linked indicates programs linked statically, for example using the -static option)

Not stripped

The executable contains a symbol table (this can be removed with the strip command).

The file command can also be used on object files, where it gives similar output. The POSIX standard for Unix systems defines the behavior of the file command

## Examining the symbol table

As described earlier in the discussion of debugging, executables and object files can contain a symbol table. This table stores the location of functions and variables by name, and can be displayed with the nm command:

$ nm a.out

08048334 t Letext

08049498 ? \_DYNAMIC

08049570 ? \_GLOBAL\_OFFSET\_TABLE\_

........

080483f0 T main

08049590 b object.11

0804948c d p.3

U printf@GLIBC\_2.0

Among the contents of the symbol table, the output shows that the start of the main function has the hexadecimal offset 080483f0. Most of the symbols are for internal use by the compiler and operating system. A ‘T’ in the second column indicates a function that is defined in the object file, while a ‘U’ indicates a function which is undefined (and should be resolved by linking against another object file). A complete explanation of the output of nm can be found in the GNU Binutils manual.

The most common use of the nm command is to check whether a library contains the definition of a specific function, by looking for a ‘T’ entry in the second column against the function name.

## Finding dynamically linked libraries

When a program has been compiled using shared libraries it needs to load those libraries dynamically at run-time in order to call external functions. The command ldd examines an executable and displays a list of the shared libraries that it needs. These libraries are referred to as the shared library *dependencies* of the executable.

For example, the following commands demonstrate how to find the shared library dependencies of the *Hello World* program:

$ gcc -Wall hello.c

$ ldd a.out

libc.so.6 => /lib/libc.so.6 (0x40020000)

/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)

The output shows that the *Hello World* program depends on the C library libc (shared library version 6) and the dynamic loader library ld-linux (shared library version 2).

If the program uses external libraries, such as the math library, these are also displayed. For example, the calc program (which uses the sqrt function) generates the following output:

$ gcc -Wall calc.c -lm -o calc

$ ldd calc

libm.so.6 => /lib/libm.so.6 (0x40020000)

libc.so.6 => /lib/libc.so.6 (0x40041000)

/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)

The first line shows that this program depends on the math library libm (shared library version 6), in addition to the C library and dynamic loader library.

The ldd command can also be used to examine shared libraries themselves, in order to follow a chain of shared library dependencies.

# **Optimization in GCC**

The optimization levels provided by the GCC compiler toolchain, including the specific optimizations provided in each. We also identify optimizations that require explicit specifications, including some with architecture dependencies. This discussion focuses on the 3.2.2 version of gcc (released February 2003), but it also applies to the current release, 3.3.2.

**Levels of Optimization**

Let's first look at how GCC categorizes optimizations and how a developer can control which are used and, sometimes more important, which are not. A large variety of optimizations are provided by GCC. Most are categorized into one of three levels, but some are provided at multiple levels. Some optimizations reduce the size of the resulting machine code, while others try to create code that is faster, potentially increasing its size. For completeness, the default optimization level is zero, which provides no optimization at all. This can be explicitly specified with option -O or -O0.

**Level 1 (-O1)**

The purpose of the first level of optimization is to produce an optimized image in a short amount of time. These optimizations typically don't require significant amounts of compile time to complete. Level 1 also has two sometimes conflicting goals. These goals are to reduce the size of the compiled code while increasing its performance. The set of optimizations provided in -O1 support these goals, in most cases. These are shown in Table 1 in the column labeled -O1. The first level of optimization is enabled as:

gcc -O1 -o test test.c

[](http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/072/7269/7269t1.jpg)

Table :GCC optimizations and the levels at which they are enabled.

Any optimization can be enabled outside of any level simply by specifying its name with the -f prefix, as:

gcc -fdefer-pop -o test test.c

We also could enable level 1 optimization and then disable any particular optimization using the -fno- prefix, like this:

gcc -O1 -fno-defer-pop -o test test.c

This command would enable the first level of optimization and then specifically disable the defer-pop optimization.

**Level 2 (-O2)**

The second level of optimization performs all other supported optimizations within the given architecture that do not involve a space-speed trade-off, a balance between the two objectives. For example, loop unrolling and function inlining, which have the effect of increasing code size while also potentially making the code faster, are not performed. The second level is enabled as:

gcc -O2 -o test test.c

Table 1 shows the level -O2 optimizations. The level -O2 optimizations include all of the -O1 optimizations, plus a large number of others.

**Level 2.5 (-Os)**

The special optimization level (-Os or size) enables all -O2 optimizations that do not increase code size; it puts the emphasis on size over speed. This includes all second-level optimizations, except for the alignment optimizations. The alignment optimizations skip space to align functions, loops, jumps and labels to an address that is a multiple of a power of two, in an architecture-dependent manner. Skipping to these boundaries can increase performance as well as the size of the resulting code and data spaces; therefore, these particular optimizations are disabled. The size optimization level is enabled as:

gcc -Os -o test test.c

In gcc 3.2.2, reorder-blocks is enabled at -Os, but in gcc 3.3.2 reorder-blocks is disabled.

**Level 3 (-O3)**

The third and highest level enables even more optimizations (Table 1) by putting emphasis on speed over size. This includes optimizations enabled at -O2 and rename-register. The optimization inline-functions also is enabled here, which can increase performance but also can drastically increase the size of the object, depending upon the functions that are inlined. The third level is enabled as:

gcc -O3 -o test test.c

Although -O3 can produce fast code, the increase in the size of the image can have adverse effects on its speed. For example, if the size of the image exceeds the size of the available instruction cache, severe performance penalties can be observed. Therefore, it may be better simply to compile at -O2 to increase the chances that the image fits in the instruction cache.

**Alignment Optimizations**

In the second optimization level, we saw that a number of alignment optimizations were introduced that had the effect of increasing performance but also increasing the size of the resulting image. Three additional alignment optimizations specific to this architecture are available. The -malign-int option allows types to be aligned on 32-bit boundaries. If you're running on a 16-bit aligned target, -mno-align-int can be used. The -malign-double controls whether doubles, long doubles and long-longs are aligned on two-word boundaries (disabled with -mno-align-double). Aligning doubles provides better performance on Pentium architectures at the expense of additional memory.

Stacks also can be aligned by using the option -mpreferred-stack-boundary. The developer specifies a power of two for alignment. For example, if the developer specified -mpreferred-stack-boundary=4, the stack would be aligned on a 16-byte boundary (the default). On the Pentium and Pentium Pro targets, stack doubles should be aligned on 8-byte boundaries, but the Pentium III performs better with 16-byte alignment.

**Speed Optimizations**

For applications that utilize standard functions, such as memset, memcpy or strlen, the -minline-all-stringops option can increase performance by inlining string operations. This has the side effect of increasing the size of the image.

Loop unrolling occurs in the process of minimizing the number of loops by doing more work per iteration. This process increases the size of the image, but it also can increase its performance. This option can be enabled using the -funroll-loops option. For cases in which it's difficult to understand the number of loop iterations, a prerequisite for -funroll-loops, all loops can be unrolled using the -funroll-all-loops optimization.

A useful option that has the disadvantage of making an image difficult to debug is -momit-leaf-frame-pointer. This option keeps the frame pointer out of a register, which means less setup and restore of this value. In addition, it makes the register available for the code to use. The optimization -fomit-frame-pointer also can be useful.

When operating at level -O3 or having -finline-functions specified, the size limit of the functions that may be inlined can be specified through a special parameter interface. The following command illustrates capping the size of the functions to inline at 40 instructions:

gcc -o sort sort.c --param max-inline-insns=40

This can be useful to control the size by which an image is increased using -finline-functions.

**Code Size Optimizations**

The default stack alignment is 4, or 16 words. For space-constrained systems, the default can be minimized to 8 bytes by using the option -mpreferred-stack-boundary=2. When constants are defined, such as strings or floating-point values, these independent values commonly occupy unique locations in memory. Rather than allow each to be unique, identical constants can be merged together to reduce the space that's required to hold them. This particular optimization can be enabled with -fmerge-constants.

**Graphics Hardware Optimizations**

Depending on the specified target architecture, certain other extensions are enabled. These also can be enabled or disabled explicitly. Options such as -mmmx and -m3dnow are enabled automatically for architectures that support them.

**Other Possibilities**

We've discussed many optimizations and compiler options that can increase performance or decrease size. Let's now look at some fringe optimizations that may provide a benefit to your application.

The -ffast-math optimization provides transformations likely to result in correct code but it may not adhere strictly to the IEEE standard. Use it, but test carefully.

When global common sub-expression elimination is enabled (-fgcse, level -O2 and above), two other options may be used to minimize load and store motions. Optimizations -fgcse-lm and -fgcse-sm can migrate loads and stores outside of loops to reduce the number of instructions executed within the loop, therefore increasing the performance of the loop. Both -fgcse-lm (load-motion) and -fgcse-sm (store-motion) should be specified together.

The -fforce-addr optimization forces the compiler to move addresses into registers before performing any arithmetic on them. This is similar to the -fforce-mem option, which is enabled automatically in optimization levels -O2, -Os and -O3.

A final fringe optimization is -fsched-spec-load, which works with the -fschedule-insns optimization, enabled at -O2 and above. This optimization permits the speculative motion of some load instructions to minimize execution stalls due to data dependencies

**Testing for Improvements**

Earlier we used the time command to identify how much time was spent in a given command. This can be useful, but when we're profiling our application, we need more insight into the image. The gprof utility provided by GNU and the GCC compiler meets this need. Full coverage of gprof is outside the scope of this article, but Listing 3 illustrates its use.

**Listing :- Simple Example of gprof**

[mtj@camus]$ gcc -o sort sort.c -pg -O2 -march=pentium2

[mtj@camus]$ ./sort

[mtj@camus]$ gprof --no-graph -b ./sort gmon.out

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls ms/call ms/call name

100.00 0.79 0.79 1 790.00 790.00 bubbleSort

0.00 0.79 0.00 1 0.00 0.00 init\_list

[mtj@camus]$

The image is compiled with the -pg option to include profiling instructions in the image. Upon execution of the image, a gmon.out file results that can be used with the gprof utility to produce human-readable profiling data. In this use of gprof, we specify the -b and --no-graph options. For brief output (excludes the verbose field explanations), we specify -b. The --no-graph option disables the emission of the function call-graph; it identifies which functions call which others and the time spent on each.

Reading the example from Listing, we can see that bubbleSort was called once and took 790ms. The init\_list function also was called, but it took less than 10ms to complete (the resolution of the profile sampling), so its value was zero.

If we're more interested in changes in the size of the object than speed, we can use the size command. For more specific information, we can use the objdump utility. To see a list of the functions in our object, we can search for the .text sections, as in:

objdump -x sort | grep .text

From this short list, we can identify the particular function we're interested in understanding better.

**Examining Optimizations**

The GCC optimizer is essentially a black box. Options and optimization flags are specified, and the resulting code may or may not improve. When they do improve, what exactly happened within the resulting code? This question can be answered by looking at the resulting code.

To emit target instructions from the compiler, the -S option can be specified, such as:

gcc -c -S test.c

which tells gcc to compile the source only (-c) but also to emit assembly code for the source (-S). The resulting assembly output will be contained in the file test.s.

The disadvantage of the previous approach is you see only assembly code, no aspect of the size of the actual instructions is given. For this, we can use objdump to emit both assembly and native instructions, like so:

gcc -c -g test.c

objdump -d test.o

For gcc, we specify compile with only -c, but we also want to include debug information in the object (-g). Using objdump, we specify the -d option to disassemble the instructions in the object. Finally, we can get assembly-interspersed source listings with:

gcc -c -g -Wa,-ahl,-L test.c

This command uses the GNU assembler to emit the listing. The -Wa option is used to pass the -ahl and -L options to the assembler to emit a listing to standard-out that contains the high-level source and assembly. The -L option retains the local symbols in the symbol table.

**Conclusion**

All applications are different, so there's no magic configuration of optimization and option switches that yield the best result. The simplest way to achieve good performance is to rely on the -O2 optimization level; if you're not interested in portability, specify the target architecture using -march=. For space-constrained applications, the -Os optimization level should be considered first. If you're interested in squeezing the most performance out of your application, your best bet is to try out the different levels and then use the various utilities to check the resulting code. Enabling and/or disabling certain optimizations also may help exploit the optimizer to receive the best performance.

**Operating System**

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Some popular Operating Systems include Linux, Windows, OS X, VMS, OS/400, AIX, z/OS, etc.

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.



Following are some of important functions of an operating System.

* Memory Management
* Processor Management
* Device Management
* File Management
* Security
* Control over system performance
* Job accounting
* Error detecting aids
* Coordination between other software and users

It is the first software you see when you turn on the computer, and the last software you see when the computer is turned off. It is the software that enables all the programs you use.

## Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management −

* Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
* In multiprogramming, the OS decides which process will get memory when and how much.
* Allocates the memory when a process requests it to do so.
* De-allocates the memory when a process no longer needs it or has been terminated.

## Processor Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management −

* Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
* Allocates the processor (CPU) to a process.
* De-allocates processor when a process is no longer required.

## Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management −

* Keeps tracks of all devices. Program responsible for this task is known as the **I/O controller**.
* Decides which process gets the device when and for how much time.
* Allocates the device in the efficient way.
* De-allocates devices.

## File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

* Keeps track of information, location, uses, status etc. The collective facilities are often known as **file system**.
* Decides who gets the resources.
* Allocates the resources.
* De-allocates the resources.

## Other Important Activities

Following are some of the important activities that an Operating System performs −

* **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
* **Control over system performance** − Recording delays between request for a service and response from the system.
* **Job accounting** − Keeping track of time and resources used by various jobs and users.
* **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
* **Coordination between other softwares and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

## Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.**

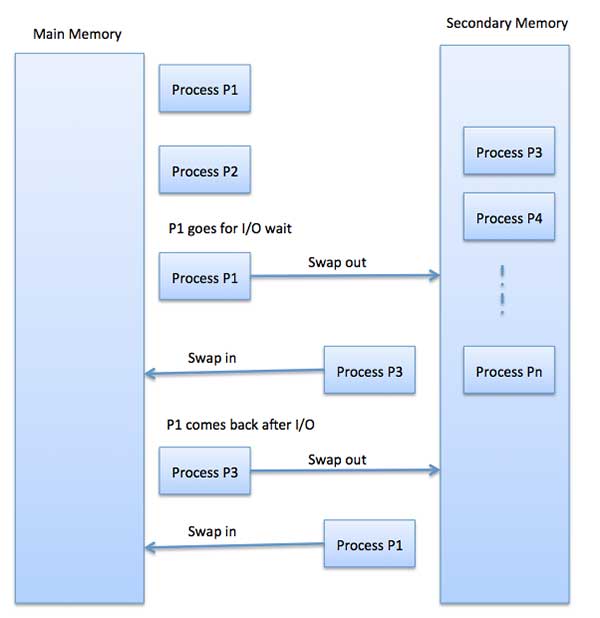
The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

* The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
* The user program deals with virtual addresses; it never sees the real physical addresses.

## Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**.



The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

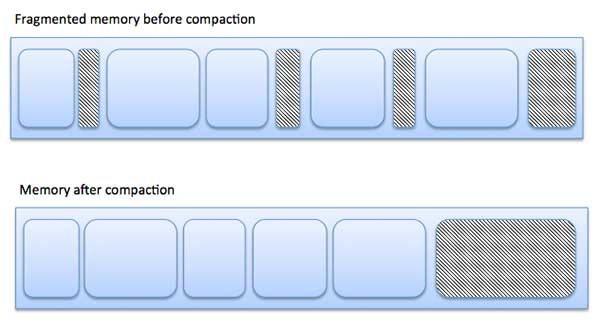
## Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types −

|  |  |
| --- | --- |
| 1 | **External fragmentation**  Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used. |
| 2 | **Internal fragmentation**  Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process. |

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory −



External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

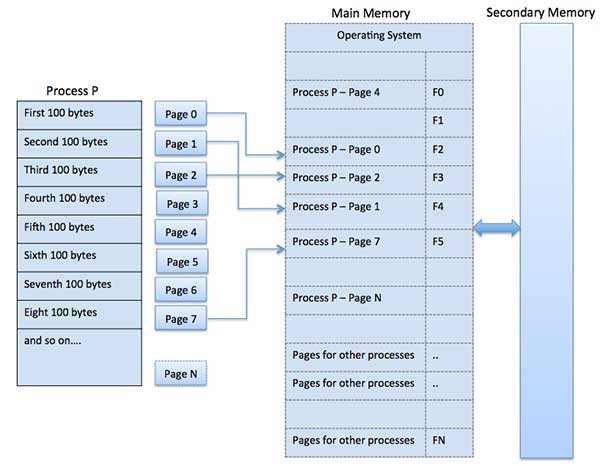
The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

## Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



### **Address Translation**

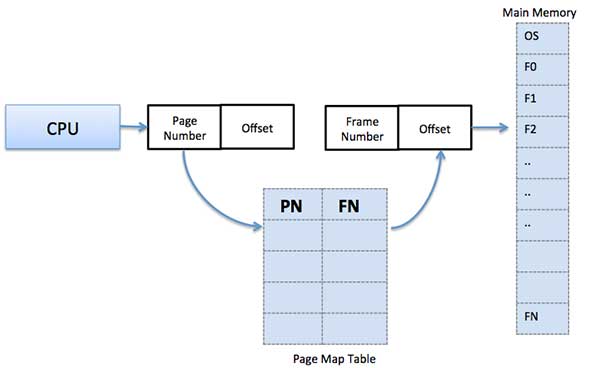
Page address is called **logical address** and represented by **page number**and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.



When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

### Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

* Paging reduces external fragmentation, but still suffer from internal fragmentation.
* Paging is simple to implement and assumed as an efficient memory management technique.
* Due to equal size of the pages and frames, swapping becomes very easy.
* Page table requires extra memory space, so may not be good for a system having small RAM.

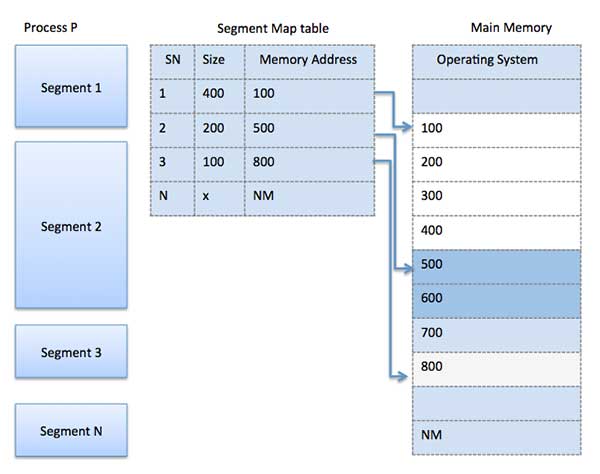
## Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Segmentation

**Virtual Memory**

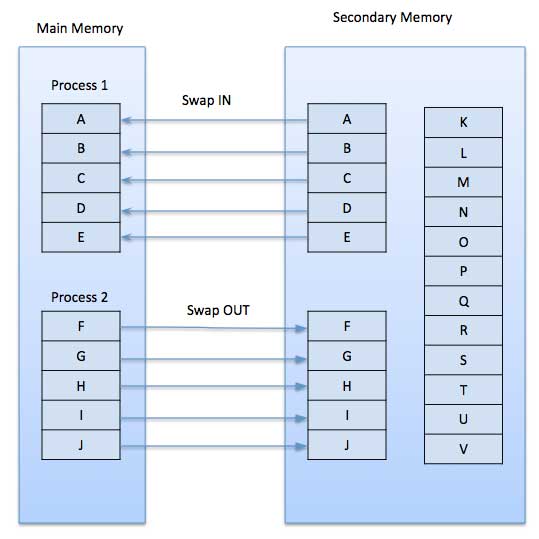
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

## Virtual memory is commonly implemented by demand paging.

## Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program’s pages out to the disk or any of the new program’s pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program’s pages as they are referenced.



While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

### Advantages

Following are the advantages of Demand Paging −

* Large virtual memory.
* More efficient use of memory.
* There is no limit on degree of multiprogramming.

### Disadvantages

* Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

## Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

Some of the algorithms include:

1.First In First Out Algorithm(FIFO)

2.Optimal Page Algorithm

3.Least Recently Used Algorithm(LRU)

4.Least Frequently Used Algorithm(LFU)

5.Most Frequently Used Algorithm(MFU)

**Processor Management**

## Process

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

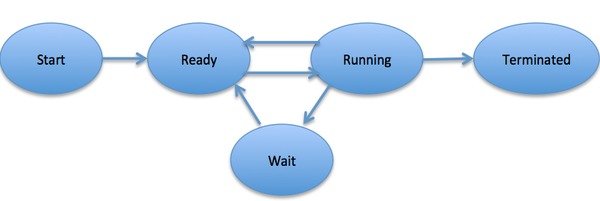
To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

## Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

|  |  |
| --- | --- |
| **S.N.** | **State & Description** |
| 1 | **Start**  This is the initial state when a process is first started/created. |
| 2 | **Ready**  The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process. |
| 3 | **Running**  Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions. |
| 4 | **Waiting**  Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available. |
| 5 | **Terminated or Exit**  Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory. |



## Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

|  |  |
| --- | --- |
| **S.N.** | **Information & Description** |
| 1 | **Process State**  The current state of the process i.e., whether it is ready, running, waiting, or whatever. |
| 2 | **Process privileges**  This is required to allow/disallow access to system resources. |
| 3 | **Process ID**  Unique identification for each of the process in the operating system. |
| 4 | **Pointer**  A pointer to parent process. |
| 5 | **Program Counter**  Program Counter is a pointer to the address of the next instruction to be executed for this process. |
| 6 | **CPU registers**  Various CPU registers where process need to be stored for execution for running state. |
| 7 | **CPU Scheduling Information**  Process priority and other scheduling information which is required to schedule the process. |
| 8 | **Memory management information**  This includes the information of page table, memory limits, Segment table depending on memory used by the operating system. |
| 9 | **Accounting information**  This includes the amount of CPU used for process execution, time limits, execution ID etc. |
| 10 | **IO status information**  This includes a list of I/O devices allocated to the process. |

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems.

The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

The Operating System maintains the following important process scheduling queues −

* **Job queue** − This queue keeps all the processes in the system.
* **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
* **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.



## Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

* Long-Term Scheduler
* Short-Term Scheduler
* Medium-Term Scheduler

## Long Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound.

## Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

## Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

## Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

* A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms-First-Come, First-Served (FCFS) Scheduling
* Shortest-Job-Next (SJN) Scheduling
* Priority Scheduling
* Shortest Remaining Time
* Round Robin(RR) Scheduling

**References**

1. Kernighan and Ritchie, The C Programming Language, 2nd Ed.
2. C: The Complete Reference
3. Randal E. Bryant, David R. O'Hallaron, Computer Systems: A Programmer's Perspective.
4. Brian J. Gough, An Introduction to GCC