








Document Information

Analyzed document	Unix and Shell Programming.pdf (D165842163)
Submitted	2023-05-04 08:47:00
Submitted by	Mumtaz B
Submitter email	mumtaz@code.dbuniversity.ac.in
Similarity	2%
Analysis address	mumtaz.dbuni@analysis.arkund.com

Sources included in the report

SA	BCA 303 LINUX.docx Document BCA 303 LINUX.docx (D53176565)	 7
SA	Linux_system_administration_block_2.pdf Document Linux_system_administration_block_2.pdf (D149208459)	 11
W	URL: https://usermanual.wiki/Document/PracticalGuidetoLinuxCommandsEditorsandShel.2146662400/help Fetched: 2021-11-14 20:51:00	 8
W	URL: https://dokumen.pub/a-practical-guide-to-unix-for-mac-os-x-users-1733062564-0131863339-9780131... Fetched: 2022-05-17 22:36:04	 2
SA	unixassignment-1(14691A0532).docx Document unixassignment-1(14691A0532).docx (D25928407)	 1
W	URL: https://mamchenkov.net/wordpress/wp-content/uploads/2017/05/Prentice_Hall_A_Practical_Guide_to... Fetched: 2022-07-25 11:13:41	 2
SA	DECAP448_LINUX_AND_SHELL_SCRIPTING.pdf Document DECAP448_LINUX_AND_SHELL_SCRIPTING.pdf (D142327428)	 2

Entire Document

UNIX and Shell Programming

Before you Begin Many is the time when have to walk into a classroom filled with expectant questions; a group of eager minds, waiting to absorb all the pearls of wisdom that you may deign to drop. At such times, where do you turn to? Agreed the Net is always around for you, so are half a dozen or more books, but all these tools are more of over skill. There is so much information in there that it becomes difficult to stay on track of what you are looking for and sometimes to even find that small bit of elusive knowledge that you know is out there, somewhere. To the doubting Thomases we can only say, get onto a search engine, any one, and perform a search for Unix. The result will give you an idea of what we are referring to here. Through the book, we hope to solve this problem. It is meant to be a reference material for you, a place to turn to when you need that little bit of ammunition. Apart from additional information on the topic at hand, we have also provided answers to frequently asked questions used in each session. All of these have been painstakingly developed by our group of experts drawing from their vast experience in imparting quality education to a wide cross-section of students. However, if there may be anything more you would like to see added to this material, you are more than welcome to get in touch with us and we will be delighted to accommodate you. Why? Because this book is meant for you and only you can tell us what more you would like within its covers.

Objectives and Conventions Objectives of this Book This book is intended to add to your existing knowledge of the subject at hand, Unix System V. A lot of the information contained herein may be new to you, some of it, things you already knew.

Irrespective of that, you will find that the book's inherent strength lies in its ability to let you put the knowledge you gained and the knowledge you possessed within the framework of the session that you are about to handle. Conventions Used in this Book ? Subheadings appear in bold without underlining ? Note is shown inside a box. It is information, which is necessary to understand the subject discussed but does not come as a part of the text flow ? Figure numbers and descriptions are given with each picture, diagram or screen for proper navigation ? The code in the sessions have been referred with a background

Table of Contents 1. A Preview of Unix 1.1 Operating System - An Overview 1.2 Operating System Concepts 1.3 Memory Management 1.4 Unix - An Overview 1.5 Commands In Unix 1.6 Vi Editor 2. Files and Nodes 2.1 The Unix File 2.2 Implementation of Unix File 2.3 About Unix Files 2.4 Directories 2.5 Telnet 2.6 Networking 2.7 Sockets 3. Standard Devices & Vi 3.1 Introduction 3.2 vi Editor 3.3 Search Patterns 3.4 List of Commands in vi Editor 3.5 Line Editing in vi 3.6 Shell Escape Commands 3.7 Options in vi 3.8 Limitations 4. Unix Threading 4.1 Unix Proceses 4.2 Process States 4.3 Process Creation 4.4 Process Handling 4.5 UNIX Mail System 5. Shell Programming 5.1 Introduction to Shell 5.2 Bourne Shell 5.3 Bourne Shell 5.4 Functions in Bourne Shell 5.5 Restricted Shell 5.6 C Shell 5.7 C Shell Builtins

Table of Contents 6. Shell Programming 6.1 System Administration 6.2 Security Issues 6.3 Disk Management 6.4 File System Mounting 6.5 Checking the Network I. Appendix II. The Zero Hour

Chapter 1- A Preview of UNIX OPERATING SYSTEM – AN OVERVIEW The most fundamental of all system programs is the operating system, which controls all the computer resources and provides the base upon which the application programs are written. The operating system has gradually grown to a stage where a layer of software is put on top of the bare hardware, to manage all parts of the system and present the user with an interface or virtual machine that is easier to understand and program. A computer system consists of hardware, system programs and application programs. The hardware consists of physical devices such as integrated circuit chips, wires, power supplies, cathode ray tubes and other such devices. The system programs layer is the primitive software that directly controls these devices and provides a cleaner interface to the next layer. It is called the microprogram and is usually read only. It is an interpreter which interprets machine language and contains a set of primitive instructions such as ADD, MOVE, SUB, etc. On top of the operating system is the rest of the system software. Here we find the command interpreter (shell), compilers, editors and similar application independent programs. The operating system is that part of the software that runs in the kernel mode or supervisor mode. It ensures that the user does not tamper with the hardware. Compilers and editors run in the user mode. Users are free to write their own compilers. Fig. 1.1 Layers in Operating System Above the system programs are the application programs written by the users to solve specific problems such as commercial data processing, engineering calculations or game playing.

The Operating System as a Resource Manager: The operating system's primary role is to present users with a convenient interface in a top-down view. A modern computer consists of processors, memories, timers, disks, terminals, magnetic tape drives, network interfaces, laser printers and a wide variety of other devices. An operating system should provide for an orderly controlled allocation of the above resources among the users. History of Operating Systems: The First Generation (1945 - 1955): Vaccum Tubes and Plugboards Between 1945 and 1955 a single group of people designed, built, programmed, operated and maintained the machines. All programming was done in absolute machine language by wiring plugboards. The usual mode of operation was for the programmer to sign up for a block of time on the signup sheet, then come down to the machine room insert his or her plugboard and do the work. All the work was numeric in nature. The Second Generation (1955-1965): Transistors and Batch Systems The introduction of transistors in the mid 1950.s changed the picture radically. There was a clear distinction between the programmers, operators, designers, builders, etc. These machines were locked in special air-conditioned rooms and costed millions of dollars. To run a job, a programmer would first write a program on paper and then punch it on cards. The punched card deck was then bought into the input room and handed over to one of the operators. When the job was completed the operator would get the printout and the programmer would use it. Batch Systems were used to speed up work. The process involved collecting a tray full of jobs in the input room and then reading them on to magnetic tapes using inexpensive computers such as the IBM-1401. The Third Generation (1965-1980): ICs and Multiprogramming The IBM-360 was the first major computer to use Integrated Circuits (ICs) and this provided a major price/performance advantage over the second generation computers which were built up from individual transistors. Multiprogramming involved partitioning memory into several pieces and assigning a different task to each partition. While one task waits for I/O to complete, another task could use the CPU. If sufficient tasks are held in the main memory at all times, the CPU could always be kept busy. Fig. 1.2 A Multiprogramming System with three tasks in memory

Another major feature of the third generation operating system was its ability to read the tasks from cards onto the disk as soon as they are brought to the computer room. Once a task was completed, the operating system loaded a new task from the disk into the empty partitions. This technique is called spooling (Simultaneous Peripheral Operation On Line). The Fourth Generation (1980-1990): Personal Computers With the development of Large Scale Integration (LSI) circuits, the microprocessor chip made it possible for a single individual to have his or her own personal computer. The most powerful personal computers were called Workstations. Much of the software written for these were user friendly. Two operating systems dominated the personal computer and workstation market scene. These were Microsoft's MS-DOS and UNIX. MS-DOS is widely used on the IBM PC and other machines using the Intel 8088 CPU and its successors 80286, 80386 and 80486. UNIX on the other hand is predominantly used on non-Intel computers and workstations.

Network Operating System: Here the users are aware of the existence of multiple computers and can log into remote machines and copy files from one machine to another. Each machine runs its own local operating system. These machines need a network interface controller and low level software to drive it and programs to achieve remote login and remote file access.

Distributed Operating System: Distributed Operating System is a type of operating system, which appears to the user as a uniprocessor system even though it is actually composed of multiple processors. In a true distributed system the users should not be aware of where their programs are being run or where their files are located. All these should be handled by the operating system. Distributed Operating Systems' require more complex processor scheduling algorithms in order to optimize the amount of parallelism achieved as these operating systems allow programs to run on several processors at the same time.

OPERATING SYSTEM CONCEPTS System Calls: User programs communicate with the operating system and request services from it by making System Calls. Corresponding to each system call is a library procedure that user programs can call. This procedure puts the parameters of the system call in a specified machine registers and then issues a TRAP instruction to start the operating system. When operating system gets control after the TRAP it checks the parameters and performs the work requested. It puts a status code in a register indicating whether the call is successful or not.

A Shell: A shell is the primary interface between a user at his terminal and the operating system. It makes heavy use of the features of the operating system. When a user logs in, a shell starts up. The shell has a terminal as standard input and output. It starts out by typing the prompt, a character akin to the dollar sign, which informs the user that the shell is waiting to accept a command. Commands may take arguments, which are passed, to the called program as character strings. To make it easy to specify multiple file names the shell accepts magic characters called wildcards. `ls *.C` This command lists all the files with .C extension. A program like shell does not have to open the terminal in order to read from it or write to it. Instead when it starts up it automatically has access to a file called the standard input (for reading), a file called the standard output (for writing normal output) and standard error (for writing error messages). All three are default to the terminal. It is possible to put a list of commands in a file and then start a shell with this file as standard input. Files containing shell commands are called shell scripts. Shell scripts may assign values to shell commands and read them later. They may also have parameters and use if, for, while and case constructs. Berkeley C shell is an alternative shell that has been designed to make shell scripts look like C programs in many respects.

Operating System Structures: There are various operating system structures. They are

- Monolithic Systems:** Here the operating system is written as a collection of procedures, each of which can invoke any of the other procedures whenever the need arises. Each procedure in the system has a well-defined interface in terms of parameters and results. The services or the system calls are requested by putting the system calls in registers or the stack and then executing them in special trap instructions known as the kernel call or supervisor call. The instruction switches from user mode to the kernel mode and transfers control to the operating system.
- Layered Systems:** Here the operating system is organized in a hierarchy of layers each one constructed upon the one below it. The first system built in this manner is THE system built at the Technische Hogeschool Eindhoven in the Netherlands by E.W.Dijkstra. THE has six layers. Fig. 1.3 Structure of the Operating System
- MULTICS** was also organized as a series of concentric rings with the inner ones being more privileged than the outer ones.
- Windows-NT** also uses the layered model. There is a Hardware Abstraction Layer (HAL) above the hardware which frees the various managers from the complexities of hardware. The advantage of using the layered operating system structure is that better security policies can be enforced as the top most layers can access the resources of the layer immediately below it.
- Virtual Machines:** Virtual machines run on bare hardware and do multi-programming, providing not one but several virtual machines to the next layer. They are exact copies of the bare hardware including the kernel /user mode, I/O interrupts, etc.
- Client-Server Model:** This is a new trend of moving the code up into higher layers and removing as much as possible from the operating system leaving a minimal kernel. To request a service such as reading block of file, a user process (client process) sends the request to a server process which then does the work and sends back the answer. Fig. 1.3 Client – Structure of the Operating System Here the kernel only handles communication between clients and servers. By splitting the operating system up into parts each of which handles one facet of the system such as the file service, process service, terminal service etc. The client – server model is so adaptable that it can be used in the distributed systems. The kernel handles only the communication.

Processes: A process is basically a program in execution. It consists of the executable program, its data and stack, its program counter, stack pointer and other registers and other information needed to run the program. In an operating system all the information about each process other than the contents of its own address space is stored in an operating system table called the process table, which is an array (or linkedlist) of structures, one for each process currently in existence. The key process management calls are those that deal with creation and termination of processes. A process called the command interpreter or shell reads the commands

from an input process, creates a child process which in turn can create a new child process and this constitutes a process tree. Other process system calls are available to request for more memory, wait for a child process to terminate, and overlay its program with a different one. When the specified number of seconds have elapsed the operating system sends a signal to the process. This signal causes the process to temporarily suspend whatever it is doing, save its registers on the stack and start running a special signal handling procedure to retransmit a presumably lost message. When signal handler is done the running process is restarted. Signals are also used for process-to-process communication. The Process Model: In this model all the runnable software on the computer including the operating system is organized into a number of sequential processes. In reality the real CPU switches back and forth from one process to another. This rapid switching back and forth is called multiprogramming. When this switching happens between processes the rate at which the process performs its computation will not be uniform. So processes must be programmed with built in assumptions about timing. Process Hierarchies: In UNIX, processes are created by the fork system call, which creates an identical copy of the calling process. After the call the parent continues running in parallel with the child. The parent can then fork off more children. Process States: Every process will be in any one of the following three states. They are ? Running (using the CPU at that instant) ? Ready (runnable; temporarily stopped to let another process run) ? Blocked (unable to run until some external event happens) Fig. 1.4 UNIX Directory Structure Four transitions are possible among these three states as shown. When a process discovers that it cannot continue it then goes into a blocked state. In some systems a process must execute a BLOCK system call to get to a blocked state. A process goes from a running state to a ready state when the scheduler picks up another process. A process goes from a ready state to a running state when the scheduler picks up the

process for execution. A process shifts from a blocked state to a ready state if an external event happens. Process Table: The operating system maintains a table called the process table with one entry per process. This contains information about program counter, stack pointer, memory allocation, etc. Everything about the process must be saved when the process is switched from the running to ready state so that it can be restarted later. PROCESS TABLE INFORMATION PROCESS MANAGEMENT MEMORY MANAGEMENT FILE MANAGEMENT Registers Exit status Root directory Program counter Signal status Working directory Program status word Parent process File descriptors Process stats Process group Effective uid Process id Effective uid Effective gid Various flag bits Real gid System call parameters CPU time used Effective grid Various flag bits An operating system does the following when a software or hardware interrupt occurs. ? Hardware stacks program counters, etc. ? Hardware loads new program counter from interrupt vector ? Assembly language procedure saves registers ? Assembly language procedure sets up a new stack ? C procedure marks service process as ready ? Scheduler decides which process to run next ? C-procedure returns to the assembly code ? Assembly language procedure starts up current process MEMORY MANAGEMENT Memory management systems can be divided into two classes. One class moves the processes back and forth between main memory and the disk and the other class does not. The methods that come under the first category are compaction and overlays. Swapping and paging belong to the second class. The main memory breaks up into pieces as data is transferred to and from the disk. This process is called fragmentation. Compaction is a method of memory management where in all the broken pieces of memory are rejoined into a single large chunk of memory so that the process can be executed. Overlays involve relaying the previous data brought into the memory with the new data that is required to run the new process.

Swapping is a memory management scheme in which the processes are brought from the disk to the main memory. This movement of processes is called Swapping. Swapping can be done using variable sized partitions or fixed size partitions. Sufficient swap space should be provided for moving the process from the disk to the memory. Memory can also be managed by bit maps and linked lists. Paging is a memory management algorithm in which the virtual address space is divided into pages. The logical address space is divided into page frames. The pages and the page frames are of equal size. Each page is numbered and this number is used to reference the page. The page table contains all the information about the page. There are various level of paging such as single-level paging, two-level paging and multi-level paging. Various paging algorithms are used to decide which page is to be kept in the memory and which needs to be removed. Input/Output: One of the most important goals of an operating system is to control all the computer's input and output devices. It must issue commands to the devices, catch interrupts and handle errors. Input and output can be differentiated into hardware and software. I/O Hardware: This constitutes the commands the hardware accepts, the functions it carries out and the errors the system reports back. The various hardware involved in I/O are I/O Devices: They are categorized into block devices and character devices. Disks are block devices as they store the information in fixed size blocks. Any program can read and write into any of the blocks. Device Controllers: The electronic component in the I/O device is called as the device controllers or adapter. The controller's job is to convert the serial input bit stream into a block of bytes and perform any necessary error correction. Each controller has a few registers that are used for communicating with the CPU. Direct Memory Access (DMA): It was invented to free the CPU from the low level work of reading the disk block from the controller's buffer area in bytes or words at a time. CPU will give the control to a controller which will perform the reading of the information from the control area and transfer the disk block to the memory. I/O Software: The goal of I/O software is to organize the software as a series of layers with the lower ones concerned with the peculiarities of the hardware. The key concepts are device independence and uniform naming. This is structured into four layers namely Interrupt handlers, Device Drivers, Device-independent operating system software and user level software.

Device Drivers: These handle one type of device and all the device independent code goes into these drivers. Device-independent operating system software: It performs various functions such as error reporting, allocating and releasing dedicated devices, buffering, device protection, device naming, etc. UNIX – AN OVERVIEW UNIX is a powerful computer operating system originally developed at AT&T Bell Laboratories. It is very popular among the scientific, engineering, and academic communities due to its multiuser and multi-tasking environment, flexibility, portability, electronic mail and networking capabilities, and the numerous programming, text processing and scientific utilities available. It has also gained widespread acceptance in government and business circles. Over the years, two major forms (with several vendor.s variants of each) of UNIX have evolved: AT&T UNIX System V and the Berkeley Software Distribution (BSD) from University of California at Berkeley.s. Sun.s Solaris 2.5.1, which is an implementation of System V is the primary version of UNIX available at Rice. Irix is also available as a System Unix-based version used by Silicon Graphics machines. UNIX Layers When you use UNIX, several layers of interaction occur between you and the computer hardware. The first layer is the kernel, which runs on the actual machine hardware and manages all the interaction with the hardware. All applications and commands in UNIX interact with the kernel, rather than the hardware directly, and they make up the second layer. On top of the applications and commands is the command-interpretor program, the shell, which manages the interaction between you, your applications, and the available UNIX commands. Most UNIX commands are separate programs, distinct from the kernel. A final layer, which may or may not be present on your system, is a windowing system such as X. The windowing system usually interacts with the shell, but it can also interact directly with applications. The final layer is the user. You will interact with the entire operating system through just the shell, or through a combination of the shell and the window system. The figure below gives a visual representation of the layers of UNIX. Fig. 1.5 Layers in UNIX

The six basic elements of UNIX are: commands, files, directories, users environment, processes, and tasks. Commands are the instructions you give the system to tell it what to do. Files are collections of data that have been given filenames. A file is analogous to a container in which you can store documents, raw data, or programs. A single file might contain the text of a research project, statistical data, or an equation processing formula. Files are stored in directories. A directory is similar to a file cabinet drawer that contains many files. A directory can also contain other directories. Every directory has a name. Your environment is a collection of items that describe or modify how your computing session will be carried out. It contains things such as where the commands are located and which printer to send your output to. A command or application running on the computer is called a process. The sequence of instructions given to the computer from the time you initiate a particular task until it ends it is called a task. A task may have one or more processes in it. When you pick up your account information, you are given a userid and a password. This combination of information allows you to access your account. Type your userid using lower-case letters, then enter your password and press the RETURN key. On X terminals, you will get a window containing system information. After reading it, use the left mouse button and click on either the Help or Go Away button, depending on what you want. Help puts you into a help system, Go Away allows you to begin your work. A new account is provided with a set of command procedures, which are executed each time you log in. You can change part of the UNIX environment by changing these setup files (accounts on systems are set up to produce a default environment). The system will then display the command prompt. The prompt signals that the system is ready to enter your next command. The name of the workstation followed by a percent sign (%) forms the command prompt (e.g. %). Once you finish typing a command, press RETURN to execute it. Changing Your Password You can change your password at any time. You should change it the first time that you log in, and it is recommended that you change it on a regular basis. Do the following steps: 1. At the command prompt, type passwd. 2. You will be prompted to enter your old password and be asked twice to enter your new password. Neither your old nor new password will appear on the screen as you type. In order to be accepted, your password must meet the following conditions: ? It must be at least seven characters long

? It must not match anything in your UNIX account information, such as your login name, or an item from your account information data entry ? It must not be found in the system's dictionary unless a character other than the first is capitalized ? It must not have three or more consecutively repeated characters or words For example, changing your password from unixuser to newunixuser will be as follows, except that the keystrokes for your old and new password will not be echoed on the screen. passwd Current password: unixuser New password (? for help): newunixuser New password (again): newunixuser Password changed for userid Remember to choose your own passwords. On many systems, the password is not changed immediately. It can take more than an hour for the system to install the new password due to the scheduling of the password changing process. Thus you should be prepared to use your old password to login again. COMMANDS IN UNIX The UNIX Shell Once you have logged in, you are ready to start using UNIX. As mentioned earlier, you interact with the system through a command interpreter program called the shell. Most UNIX systems have two different shells, although you will only use one or the other almost all of the time. The shell you will find on Information Systems supported networks is the C shell. It is called the C shell because it has syntax and constructs similar to those in the C programming language. The C shell command prompt often includes the name of the computer that you are using and usually ends with a special character, most often the percent sign (%). Another common shell is the Bourne shell, named after its author. The default prompt for the Bourne shell is the dollar sign (\$). (If the prompt is neither one of these, a quick way to check which shell you are using is to type the C shell command alias; if a list appears, then you are using the C shell; if the message, Command not found appears, then you are using the Bourne shell). Modified versions of these shells are also available. TC shell (tcsh) is C shell with file name completion and command line editing (default prompt: <);. The GNU Bourne-Again shell (bash) is basically the Bourne shell with the same features added (default prompt: bash\$). In addition to processing your command requests, UNIX shells have their own syntax and control constructs. You can use these shell commands to make your processing more efficient, or to automate repetitive tasks. You can even store a sequence of shell commands in a file, called a shell script, and run it just like an ordinary program.

About UNIX Commands UNIX has a wide range of commands that allow you to manipulate not only your files and data, but also your environment. This section explains the general syntax of UNIX commands to get you started. A UNIX command line consists of the name of the UNIX command followed by its arguments (options, filenames and/or other expressions) and ends with a RETURN. In function, UNIX commands are similar to verbs in English. The option flags act like adverbs by modifying the action of the command, and filenames and expressions act like objects of the verb. The general syntax for a UNIX command is: command [-flag options] file/expression The brackets around the flags and options indicate that they are often optional, and only need to be invoked when you want to use that option. Also, flags need not always be specified separately, each with their own preceding dash. Many times, the flags can be listed one after the other after a single dash. You should follow several rules with UNIX commands: ? UNIX commands are case-sensitive, but most are lowercase ? UNIX commands can only be entered at the shell prompt ? UNIX command lines must end with a RETURN ? UNIX options often begin with a .- (minus sign) ? More than one option can be included with many commands Getting On-Line Help with Commands: The standard on-line help facility available with UNIX is electronic reference manuals, known as the man pages; and you access them with the man command. Syntax man command-name The man pages provide an in-depth description of command-name, with an explanation of its options, examples, and further references. The information is an electronic duplicate of the paper reference manual pages. Use the man command for explicit information about how to use a particular command. Use the -k option to search for a keyword among the one-line descriptions in the help files. Syntax man -k keyword

The command apropos serves exactly the same function as man -k and is used in the same way. You can read about the man command using man. Type man man at the prompt. The UNIX reference manual is divided into eight numbered sections: 1. General User Commands 2. System Calls 3. User-level Library Functions 4. Device Drivers, Protocols 5. File Formats 6. Games (rarely available) 7. Document Preparation 8. System Administration You can see the command summary for each section by typing: man # intro where # is one of the eight section numbers. In addition, other applications that reside on your system may have man pages. These pages can often be invoked in the same manner as the operating system man pages. Case Sensitivity UNIX is a case sensitive operating system. It treats lower-case characters differently from uppercase characters. For example, the files readme, Readme, and README would be treated as three different files. Most command names and files are entirely in lower-case. Therefore, you should generally plan to type in lower-case for most commands, command line arguments, and option letters. Special Keys and Control Characters UNIX recognizes special keys and control-character key strokes and assigns them special functions. A special key such as the DELETE key is usually mapped to the ERASE function, which erases the most recent character that you typed on the current line. A control-keystroke such as CTRL-C is invoked .The notation for control characters is usually ^C or CTRL-C. Some standard special keys and control characters are summarized below.

SPECIAL KEYS AND CONTROL CHARACTERS SPECIAL KEY FUNCTION/DESCRIPTION DELETE Acts as a rubout or erase key. Pressing DELETE once will backup and erase one character, allowing you to correct and retype mistakes. BACKSPACE This key is sometimes used as the rubout key instead of the DELETE key. Otherwise, it is mapped as a backspace key, which generates a ^H on the display. CTRL-U U erases the entire command line. It is also called the line kill character. CTRL-W W erases the last word on the command line. CTRL-S S stops the flow of output on the display. CTRL-Q Q resumes the flow of output stopped by CTRL-Q CTRL-C C interrupts a command or process in progress and returns to the command line. This will usually work; if it doesn't, try typing several ^C.s in a row. If it still doesn't work, try typing ^\, q (for quit), exit, ^D, or ^Z. CTRL-Z Z suspends a command or process in progress. CTRL-D D generates an end-of-file character. It can be used to terminate input to a program, or to end a session with a shell. CTRL-\ ^\ quits a program and saves an image of the program in a file called core for later debugging.

SETUP AND STATUS COMMANDS COMMAND PURPOSE logout Ends your UNIX session passwd Changes password by prompting for old and new passwords stty Sets terminal options date Displays or set the date finger Displays information about users ps Displays information about processes env Displays or changes current environment set C shell command to set shell variables alias C shell command to define command abbreviations history C shell command to display recent commands

FILE AND DIRECTORY COMMANDS COMMAND PURPOSE cat concatenates and displays file(s) more paginator - allows you to browse through a text file less more versatile paginator than more mv moves or renames files cp copies files rm removes files ls lists contents of directory mkdir makes a directory rmdir removes a directory cd changes a working directory pwd prints a working directory name du summarizes disk usage chmod changes mode (access permissions) of a file or directory file determines the type of file quota -v displays current disk usage for this account

EDITING TOOLS COMMAND PURPOSE pico Simple text editor diff Show differences between the contents of files grep Searches a file for a pattern sort Sorts and collates lines of a file (only works on one file at a time) wc Counts lines, words, and characters in a file look Looks up specified words in the system dictionary awk Pattern scanning and processing language gnuemacs Advanced text editor vi Screen oriented (visual) display editor lprloc Locations, names of printers; printout costs pacinfo Current billing info for this account

PROGRAM CONTROLS, PIPES, AND FILTERS COMMAND PURPOSE CTRL-C Interrupts current process or command CTRL-D Generates end-of-file character CTRL-S Stops flow of output to screen CTRL-Q Resumes flow of output to screen CTRL-Z Suspends current process or command jobs Lists background jobs sleep Suspends execution for an interval kill Terminates a process nice Runs a command at low priority renice Alters priority of running process and runs process in background when placed at end of the command line. < Redirects the output of a command into a file << Redirects and appends the output of a command to the end of a file. > Redirects a file to the input of a command <& Redirects standard output and standard error of a command into a file (C shell only) | Pipes the output of one command into another The following commands are not available with the Bourne Shell. BOURNE SHELL NOT SUPPORTED COMMANDS COMMAND PURPOSE bg Runs a current or specified job in the background fg Brings the current or specified job to the foreground !! Repeats the entire last command line !\$ Repeats the last word of the last command line OTHER TOOLS AND APPLICATIONS COMMAND PURPOSE pine electronic mail bc desk calculator man print UNIX manual page to screen elm another electronic mail program

ALIASES- ALTERNATE NAMES TO COMMANDS The alias command makes it possible to launch any command or group of commands (inclusive of any options, arguments and redirection) by entering a pre-set string (i.e., sequence of characters). That is, it allows a user to create simple names or abbreviations (even consisting of just a single character) for commands regardless of how complex the original commands are and then use them in the same way that ordinary commands are used. The alias command is built into a number of shells including ash, bash (the default shell on most UNIX systems), csh and ksh. It is one of several ways to customize the shell (another is setting environmental variables). Aliases are recognized only by the shell in which they are created, and they apply only for the user that creates them, unless that user is the root (i.e., administrative) user, which can create aliases for any user. Listing and Creating Aliases The general syntax for the alias command varies somewhat according to the shell. In the case of the bash shell it is alias [-p] [name="value"] When used with no arguments and with or without the -p option, alias provides a list of aliases that are in effect for the current user, i.e. alias Some of the aliases listed are likely to be system-wide aliases that apply to all users and are created automatically for each new user for a particular shell. Aliases for any other shell can be seen by first switching to that shell and then using the alias command as above. name is the name of the new alias and value is the command(s) which it initiates. The alias name and the replacement text can contain any valid shell input except for the equals sign (=). The commands, including any options, arguments and redirection operators, are all enclosed within a single pair of quotation marks, which can be single quotes or double quotes. No spaces are permitted before or after the equals sign. Any number of aliases can be created simultaneously by enclosing the name in each name-value pair in quotes. As a trivial example of alias creation, the alias p could be created for the commonly used pwd command, which shows the current location of the user in the directory structure (and which is an abbreviation for present working directory), by typing the following command and then pressing the ENTER key:

alias p="pwd" Then, to show the current location, instead of typing pwd, the user would only have to type the letter p and press the ENTER key, i.e. p An alias can be created with the same name as the core name of a command (i.e., a command without any options or arguments). In such case, it is the alias that is called (i.e., activated) first when the name is used, rather than the command with the same name. For example, an alias named ls could be created for the command ls -al as follows: alias ls="ls -al" ls is a commonly used command that by default lists the names of the files and directories within the current directory (i.e., the directory in which the user is currently working). The -a option instructs ls to also show any hidden files and directories, and the -l option tells it to provide detailed information about each file and subdirectory. Such an alias can be disabled temporarily and the core command called by preceding it directly (i.e. with no spaces in between) with a backslash, i.e. \ls It makes no difference whether double or single quotes are used when creating an alias. It can be slightly easier to use single quotes because this obviates the need to simultaneously use the shift key. Thus, the above example could have been written as alias ls='ls -al' This example could be simplified even further, again with no adverse effect on performance, by using a single character instead of two characters for the alias name, for example: alias l='ls -al' In addition to options, arguments can also be included in alias values. For example, to have the ls alias always display the contents of the /etc directory, it could be rewritten as: alias l='ls -al /etc' Multiple commands can be included in the same alias by inserting them within the same pair of quotation marks and separating them with semicolons. For example, the alias pl could be created to first launch pwd and then immediately launch ls:

```
alias pl='pwd; ls'
```

Aliases can even be created to call other aliases. For example, if the alias ls as shown earlier had already been created, then pl will launch it in the above example, otherwise it will launch the conventional ls command. If the aliases p and l shown in earlier examples have already been created, then the above example could alternatively be written as alias pl='p; l' The following is an example of creating two separate aliases simultaneously, as contrasted with creating a single alias that launches two separate commands: alias p="pwd"; l="ls -al" As an example of an alias for a series of commands linked by a pipe (represented by a vertical line), the alias dir can be created to generate a list of the names of and information about all of the subdirectories in the current directory: alias dir="ls -al | grep ^d" Here ls -al obtains a listing of all files and directories in the current directory. Its output is sent by the pipe to the filter grep, which then searches for lines beginning with the letter d (as all directories have a line returned by ls -al that begins with d). The caret (i.e. upward-pointing angular character) before d tells grep to search only for lines beginning with that letter. Not only can options and arguments be used in the command(s) that an alias can substitute for, but they can also be used with an alias that has already been created. As a trivial example, supposing the alias l is created for the ls command: alias l="ls -a" Then, the alias l could be used with any argument that the command ls could be used with. For example, to list the files and directories in the the /etc directory: l /etc The alias l could also be used with any option that the command ls could be used with. For example: l -l /etc The alias command is unusual in that it only has a single option. That option, -p, tells it to display a list of the aliases for the current user on the current shell. This might be helpful if used when creating an alias, but it is, of course, redundant when the alias command is used without arguments.

Uses for Aliases There are several types of uses for aliases. They include: 1. Reducing the amount of typing that is necessary for commands or groups of commands that are long and/or tedious to type. These commands could include opening a file that is frequently used for studying or editing 2. Correcting common misspellings of commands 3. Increasing the safety of the system by making commands interactive. This forces the user to confirm that it is desired to perform a specific action and thereby reduces the risk from accidental or impulsive abuse of powerful commands. 4. Standardizing the name of a command across multiple operating systems. For people accustomed to MS-DOS commands, the following aliases can be defined so that a Unix-like operating system appears to behave more like MS-DOS: alias dir="ls" alias copy="cp" alias rename="mv" alias md="mkdir" alias rd="rmdir" alias del="rm -i" VI EDITOR Vi invokes a screen oriented display editor and offers a powerful set of text editing operations based on a set of mnemonic commands. Most of the commands are single keystrokes that perform simple editing vi, view, vedit all invoke a screen- oriented display editor. Syntax: vi [-option ...] [command ...] [filename ...] view [-option ...] [command ...] [filename ...] vedit [-option ...] [command ...] [filename ...] Description: vi displays a full screen window into the file you are editing. The contents of this window can be changed quickly and easily within vi. While editing, visual feedback is provided (the name vi itself is short for —visuall). The view command is the same as vi except that the readonly option (-R) is set automatically. The file cannot be changed with view. The vedit command is the same as vi except for differences in the option settings. vedit uses novice mode, turns off the magic option, sets the option report=1 and turns on the options showmode and redraw. The showmode option informs the vedit user, in a message in the lower right corner of the screen, which mode is being used. For instance

When the `vi` command is used, the message reads `INSERT MODE vi` and the line editor `ex` are one and the same editor: the names `vi` and `ex` identify a particular user interface rather than any underlying functional differences. In user interface, however they are strikingly different. `ex` is a powerful line-oriented editor, similar to the editor `ed`. However, in both `ex` and `ed`, visual updating of the terminal screen is limited, and commands are entered on a command line. `vi`, on the other hand, is a screen-oriented editor designed so that what you see on the screen corresponds exactly and immediately to the contents of the file you are editing. In the following discussion, `vi` commands and options are printed in bold. Options available on the `vi` command line include: `-x` Encryption option; when used, the file is encrypted as it is written and requires an encryption key to be read. `vi` makes an educated guess to determine if a file is encrypted or not. Refer to the `crypt` page for information about restrictions on the availability of encryption options. `-C` Encryption option; the same as `-x` except that `vi` assumes that the files are encrypted. Refer to the `crypt` page for information about restrictions on the availability of encryption options. `-c` command `+command` Begins editing by executing the specified editor command (usually a search or positioning command). `-t` tag Equivalent to an initial tag command; edits the file containing tag and positions the editor at its definition. `-r` file Used in recovering after an editor or system crash; retrieves the last saved version of the named file. `-L` Lists the names of all files saved as a result of an editor or system crash. Files may be recovered with the `-r` option. `-wn` Sets the default window size to `n`; useful on dialups to start in small windows. `-R` Sets a read-only option so that files can be viewed but not edited.

Starting and exiting `vi` To enter `vi`, enter `-R` Sets a read-only option so that files can be viewed but not edited. `vi` Edits empty editing buffer `vi` file Edits named file `vi +123` file Goes to line 123 `vi +45` file Goes to line 45 `vi +/word` file Finds first occurrence of word
 Ways to exit the editor: There are several ways to exit `vi`. Some abort the editing session, some write out the editing buffer before exiting, and some warn you if you decide to exit without writing out the buffer. All of these ways of exiting are elaborated upon: `q` Exits `vi`. No automatic write of the editor buffer to a file is performed. `q!` Quits from the editor, discarding changes to the buffer without complaint. `wq` name Like a `w` and then a `q` command. `wq!` Name Overrides checking normally made before execution of the `w` command to any file. For example, if you own a file but do not have write permission turned on, `wq!` allows you to update the file anyway. `x` name If any changes have been made and not written, writes the buffer out and then quits. Otherwise, it just quits. `:ZZ` Writes the editing buffer to the file only if any changes are made. `:q!` Cancels an editing session. The exclamation mark (!) tells `vi` to quit unconditionally. In this case, the editing buffer is not written out.

Chapter 2- Files and Nodes THE UNIX FILE SYSTEM A UNIX file is a sequence of 0 or more bytes containing arbitrary information about the file system. There is no distinction between ASCII files and binary files or any other kind of files. File names are 14 characters long but for some versions of UNIX such as the Berkeley UNIX the length is 255 characters. Assigning a 9-bit mode called the right bits can protect files. The first three bits refer to the owner's access to the file. The next three bits refer to the access of other members of the owner's group. The last three bits refer to everyone else. Thus a mode of 640 (octal) implies that the owner can read and write the file. Other members of the owner's group can also read the file but outsiders have no access at all. Files can be grouped as directories and to a large extent can be treated as files. The character `_/` is used to separate directory names. When a process wants to read or write a file it must first open the file. It is opened using the `OPEN` system call, whose first argument gives the path name of the file to be opened and the second argument specifies if the file is to be read, written or both. If the access is permitted then the system returns a small positive integer called the file descriptor. If the access is prohibited then `.1` is returned to indicate an error. When a process starts it has three available descriptors; 0 for standard input, 1 for standard output and 2 for standard error. When a file descriptor is closed its file descriptor can be allocated on a subsequent open.

IMPLEMENTATION OF UNIX FILE SYSTEM All the disks that contain the UNIX file systems have the various blocks. Block 0 is not used by the UNIX and contains the code to boot the computer. Super block contains critical information about the layout of the file system like the number of I-nodes, number of disk blocks and start of the list of free disk blocks. Destruction of the super block will render the file system unreadable. I-nodes come next and are numbered from 1. Each I-node is 64 bytes long and describes exactly one file. An I- node contains accounting information such as the name of the owner, protection bits, information to locate all the disk blocks that hold the file data. Following these are the data blocks. All the files and directories are stored here. All the data blocks are contiguous. I-nodes are generally put in a kernel data structure called the I-node table. We put an open file descriptor table between the file descriptor table and the I- node table. The open file descriptor allows a parent and a child to share a file position and provides unrelated processes with their own values. ABOUT UNIX FILES All files have a filename, and UNIX imposes few restrictions on filenames. It allows you to name your files in such a way that you can easily recognize their contents. You

will find it useful to adopt names and classes of names that indicate how important each file is and what connection it has with other files. For example, temporary files used to test commands and options could all begin with a `tmp`. A filename can be up to 256 characters long, consisting of any alphanumeric character on the keyboard except the `/`. In general, you should keep your filenames relatively short and use lower-case characters such as letters, numbers, periods and underscores. For instance, if your program calculates employee paychecks, you might call it `payroll`, or if your file is a research paper on Frank Wright, you might call it `wright`. Do not include blanks in your filenames as they will make it difficult for you to work with the file. If you do wish to separate letters in a filename, use the underscore (`_`) character (as in `wright_paper`) or the hyphen (`-`) character.

Remember that UNIX is case sensitive, which means it recognizes the difference between upper case and lower case letters. For instance, `Wright` and `wright` would refer to two different files. When you place a single period in the middle of a filename, the part after the period is commonly referred to as an extension or suffix and usually indicates what type of information is stored in the file. You may use any extension desired; a text file might have the extension `.txt` or `.text`; a note may have the extension `.note`, and so forth. UNIX does not require extensions, but they can be used to identify similar types of files. Since some UNIX programs (especially compilers) look for certain standard extensions, it is common practice to use the following conventions: `.h` for header files, `.c` for C source files, `.f` for FORTRAN, `.p` for Pascal, and `.s` for assembler source files. So the file `wright.txt` indicates a text file whereas the file `payroll.c` indicates a C program called `payroll`. Some UNIX files begin with a period, for example, `.cshrc` or `.login`. Files that begin with a period will not appear in a normal directory listing and are usually UNIX environment and application setup files. The files that begin with a period are most often hidden files and are invisible. A large grouping of files and directories is referred to as a file system. File systems are related to the disk size and structure, and to the internal structure of UNIX. What you should remember is that user's files and directories are usually on a different file system than the system's files and directories. If the number of users is large, as on OwlNet, the user files and directories may be on more than one file system.

Creating Files
Many files are created using a text editor. A text editor is a program that allows you to enter and save text. You can also use a text editor to manipulate saved text through corrections, deletions, or insertions. The main text editors on Information Systems managed networks are `vi`, GNU Emacs, Pico, and `aXe`. You can create a file without a text editor by using the `cat` command (short for concatenate) and the `>` (redirect output) symbol. To create a file using the `cat` command, type: `vi` is included with every UNIX system, but GNU Emacs is installed separately by system managers. `aXe` is only available if you are using the X Window system.

Syntax `cat filename` `cat < filename`; `filename` is the name you wish to give the file. The command `cat` generally reads in a file and displays it to standard output. When there is no filename directly following the command, `cat` treats standard input as a file. The `>` symbol will redirect the output from `cat` into the new filename you specify. `cat` will keep reading and writing each line you type until it encounters an end-of-file character. By typing `CTRL-D` on a line by itself, you generate an end-of-file character. It will stop when it sees this character. Try it, using this example as a guide: `cat < example` When you reach the end of each line; press the RETURN key. You can only correct mistakes on the line you are currently typing. Use the DELETE key to move the cursor back to the mistake and then retype the rest of the line correctly. When you have completed the last line, press RETURN and type `CTRL-D`.
Displaying Files After creating a file, you can display it in one of several ways. You could use the `cat` command. Just type `cat` followed by the name of the file that you want to see. `cat example` Sometimes the files you want to view may be very long. When using the `cat` command, the text will scroll by very quickly. You can control the flow of text by using `CTRL-S` and `CTRL-Q`. `CTRL-S` stops the flow of text and `CTRL-Q` restarts it. If you use `CTRL-S`, to stop the flow of text, you must remember to type `CTRL-Q` or the computer will not display any output, including anything that you type. `more` is a program that displays only one screen of information at a time. Type `more` followed by a filename. `more example` The computer will display one screen of text and then wait for you to press the space bar before it displays the next page of text, until you reach the end of the file. Pressing the `?` character will show help for `more`. A utility of greater power called `less` is available on many systems; it allows reverse scrolling of files and other enhancements. It is invoked the same way as `more`.
Listing Files The `ls` command will list

58%

MATCHING BLOCK 3/33

W

the files in the current directory that do not begin with a period. Below is a list of

options you can tack on to `ls`:

`ls -a` Lists all the contents of the current directory, including files with initial periods, which are not usually listed. `ls -l` Lists the contents of the current directory in long format, including file permissions, size, and date information. `ls -s` Lists contents and file sizes in kilobytes of the current directory. If you have many files, your directory list might be longer than one screen. You can use the programs `more` or `most` with the `|` (vertical bar or pipe) symbol to pipe the directory list generated as output by the `ls` command into the `more` program. `more` or `less` will display the output from `ls` one page at a time. `ls | more`
Copying Files To make a

54%

MATCHING BLOCK 1/33

SA

BCA 303 LINUX.docx (D53176565)

copy of a file, use the `cp` (copy) command. `cp source destination` where `source` is the file you wish to copy and `destination` is the

file you are creating. `cp example example1` `ls example example1` The example has created a new file called `example1` that has the same contents as `example`. If `example1` already exists, the `cp` command will overwrite the previous contents. New accounts are often set up so that `cp` will prompt for confirmation before it overwrites an existing file. If your account is not set up in this manner, use the `-i` option (`cp -i`) to get the confirmation prompt, like: `cp -i example example1`

Renaming Files To rename one of your files, use the `mv` (move) command. `mv oldfilename newfilename` where `oldfilename` is the original filename and `newfilename` is the new filename. For instance, to rename `example` as `workfile` type: `mv example workfile` `ls example workfile` This moves the contents of `example` into the new file `workfile`. New accounts are often set up so that `mv` will prompt for confirmation before doing this. If your account is not set up in this manner, use the `-i` option (`mv -i`) to get the confirmation prompt.

Deleting Files To delete files, use the `rm` (remove) command. For instance, to delete `workfile`, type: `rm workfile` `ls example` Important: `rm` can be very dangerous. Once a file has been removed you cannot retrieve it, except possibly, from system backups (which may or may not contain the file). It may take the system administrators several days to recover your deleted file, so use a great deal of caution when deleting files. New accounts are often set up so that `rm` will prompt for confirmation. If your account is not set up in this manner, use `-i` option to get the confirmation prompt.

Creating Links between Files You can refer to one particular file by different names in different directories. The `ln` command creates a link, which points to the file. Note that links are simply alternative names for a single file; `ln` does not rename the file (as does `mv`) nor does it make a copy of the file (as does `cp`). It allows you to access the file from multiple directories. Since only one copy of the file actually exists, any changes that you make through one of its links will be reflected when you access it through another of its links. However, if you delete the link, you do not delete what it points to. Links are useful for cross-referencing files. If you know that you will need to access a file from different directories, creating links is a better alternative than making a copy of the file for each directory (and then having to alter each, every time a change is made to the original). It is also more convenient than having to use the file's full pathname every time you need to access it. Another use for linking a file is to allow another user to access that particular file without also allowing entry into the directory that actually contains the file. The kind of link you will want to create is called a symbolic link. A symbolic link contains

87%

MATCHING BLOCK 5/33

W

the pathname of the file you wish to create a link to.

Symbolic links can tie into any file in the file structure; they are not Moving a file into an existing file overwrites the data in the existing file.

limited to files within a file system. Symbolic links may also refer to directories as well as individual files. To create a symbolic link to a file within the same directory, type: `ln -s originalFile linkName` Where `originalFile` is the file that you want to link to and `linkName` is the link to that file. To create a link in a directory other than that of the original file, type: `ln -s originalFile differentDirectoryName/linkName` If you create a link within the same directory as the original file, you cannot give it the same name as the original file. There is no restriction on a file's additional names outside of its own directory. Links do not change anything about a file, no matter what the link is named. If someone makes a link to one of your files, and you then delete that file, that link will no longer point to anything and may cause problems for the other user.

Printing Files To print a file, use the `lpr` command: `lpr filename` or `lpr [-Pprintername] filename` (for laser printers only) To get a list of the printers available to your machine, type: `lprloc` `lprloc` lists all of the printers that your system knows about, by name, along with their type and location. To get some status information on the printers, use the command `lpstat -p` Note: Line printers are used for text-only files. Laser printers are needed to handle graphics or PostScript files. PostScript is a page-description language developed by Adobe Systems, Inc. and was specially designed for creating graphics and typography on a printed page. The option flag `-P printername` specifies which laser printer to use and is optional (as indicated by the brackets). When no printer is given, the print command uses the system default printer. For more information on printing commands, use the `man` command to consult the manual pages on `lpq`, `lpr`, and `lprm`. Printing accounting information is available by running this command: You should always use symbolic links when linking to files owned by others.

grep This command stands for —get regular expression printll. It is a search command that will allow you to search files for particular expressions which it will then print to the screen. The normal format for grep is: \$ grep (options) expression filename (filename2 filename3) It should also be noted that it can be more than one filename, because you can search more than one file at a time. As an example of how to use grep, assume that you have a file named —textll, which has the following contents: Java, a new object-oriented programming language developed by Sun Microsystems, Inc., can be used to develop Java —appletsll, mini-applications that are embedded into the HyperText Markup Language HTML) code currently used to create World Wide Web pages. Applets add a revolutionary new level of interactivity to the Web that goes far beyond the simple forms and buttons currently available on most Web sites. Features of applets written in Java range from simple text manipulation and 3-D animated graphics to systems or network management tasks and best of all, users can instantaneously access and run these applications just by clicking on them from their Web browsers. If you want to find all the lines that the word —Javall features in, you should type: \$ grep Java text and these lines would appear on the screen. Java, a new object-oriented programming language developed by Sun Microsystems, Inc., can be used to develop Java applets, Features of applets written in Java range from simple text manipulation. There is also a list of options that you can put in the command to get other information. Some of these options are: - v All lines but those matching are printed. - x (Exact) only lines matched in their entirety are printed (fgrep only). - c Only a count of matching lines is printed. - n Each line is preceded by its relative line number in the file. - i The case of letters is ignored in making comparisons - that is, upper and lower case characters are same. This applies only to grep and fgrep only.

If you want to know the line numbers of those lines with the word Java in them, you should type: \$ grep -n Java text and the result would be: Java, a new object-oriented programming language developed by Sun Microsystems, Inc., can be used to develop Java applets, Features of applets written in Java range from simple text manipulation. One specific option that should be noticed is the -i option. If you don.t use it, then grep is case sensitive. For example, if you want to find all uses of the word applets in the file text, and you typed: \$ grep applets text then your output would be: Microsystems, Inc., can be used to develop Java .applets,. Features of applets written in Java range from simple text manipulation However, If you type: \$ grep -i applets text then the result would be: Microsystems, Inc., can be used to develop Java —appletsll, (HTML) code currently used to create World Wide Web pages. Applets add a features of applets written in Java range from simple text manipulation mknod The mknod command makes a directory entry and a corresponding inode for a special file. The first argument is the name of the entry. In the first case, the second argument is b if the special file is block-type (disks, tape) or c if it is character-type (other devices). The last two arguments are numbers specifying the major device type and the minor device (for example, unit, drive, or line number), which may be either decimal or octal. Minor numbers must be in the range 0 to 255. Syntax /etc/mknod name [b | c] major minor /etc/mknod name p /etc/mknod name s /etc/mknod name m

The assignment of major device numbers is specific to each system. Major device numbers can be found in the system source file /etc/conf/cf.d/mdevice mknod can also be used to create named pipes with the p option, semaphores with the s option, and shared data (memory) with the m option. Only the superuser can use the first form of the syntax. mknod does not understand extended minor device numbers. It will impose an upper limit of 255 on the minor de creat – create a new file or rewrite an existing one. The creat system call creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by the path. If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file owner’s ID is set to

62%

MATCHING BLOCK 2/33

SA Linux_system_administration_block_2.pdf (D149208459)

the effective user ID of the process; the group ID of the process is set to the effective group ID

of the process; and the low-order 12 bits of the file mode are set to the value of mode and changed as o. All bits set in the process’s file mode creation mask are cleared. The save text image after execution bit of the mode is cleared. Upon successful completion, a write only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across exec system calls. No process may have more than FOPEN_MAX files open simultaneously. A new file may be created with a mode that forbids writing. Symbolic constants defining the access permission bits are specified in the >sys/stat.h< header file and should be used to construct mode The call creat (path, mode) is equivalent to the following open (path, O_WRONLY | O_CREAT | O_TRUNC, mode) The creat system call fails if one or more of the following is true: ? Search permission is denied on a component of the path prefix ? The file does not exist and the directory in which the file is to be created does not permit writing ? The file exists and write permission is denied ? The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned, and errno is set to indicate the error.vice number parameter.

DIRECTORIES About UNIX Directories UNIX directories are similar to regular files; they both have names and both contain information. Directories, however, contain other files and directories. A directory consists of an unsorted collection of 16 byte entries. Each entry contains a file name (14 characters), and number of file I-nodes. To open a file in the working directory, the system reads the directory, compares the name to be looked up with each entry until it finds the file. If the file is present then the system extracts its I-node number and uses it as an index into the I-node table to locate the corresponding I-node number and brings it into the memory. Many of the same rules and commands that apply to files also apply to directories. All files and directories in the UNIX system are stored in a hierarchical tree structure. Envision it as an upside-down tree, as in the figure below. Fig. 2.1 UNIX Directory Structure At the top of the tree is the root directory. Its directory name is a slash /. Below the root directory is a set of major subdirectories that usually include bin, dev, etc, lib, pub, tmp, and usr. For example, the /bin directory is a subdirectory, or —childll of / the root directory. The root directory, in this case, is also the parent directory of the bin directory. Each path leading down, away from the root, ends in a file or directory. Other paths can branch out from directories, but not from files. Many directories on a UNIX system have traditional names and contents. For example, directories named bin contain binary files, which are the executable command and application files. A lib directory contains library files, which are often collections of routines that can be included in programs by a compiler. dev contains device files, which are the software components of terminals, printers, disks, etc. tmp directories are for temporary storage, such as when a program creates a file for something and then deletes it once it is done. The etc directory is used for miscellaneous administrative files and commands. pub is for public files that anyone can use, and usr has traditionally been reserved for user directories, but on large systems it usually contains other bin, tmp, and lib directories.

VARIOUS DIRECTORIES IN UNIX DIRECTORY DESCRIPTION /usr/include System header files /usr/spool Spooling directories for printer and other daemons /usr/src System source code /usr/adm System accounting /usr/man Online manuals Your home directory is the directory that you start out from when you first login. It is the top-level directory of your account. Your home directory name is almost always the same as your userid. Every directory and file on the system has a path by which it is accessed, starting from the root directory. The path to the directory is called its pathname. You can refer to any point in the directory hierarchy in two different ways: using its full (or absolute) pathname or its relative pathname. The full pathname traces the absolute position of a file or directory back to the root directory, using slashes (/) to connect every point in the path. For example, in the figure above, the full pathname of file2 would be /usr/bin/file2. Relative pathnames begin with the current directory (also called the working directory, the one you are in). If /usr is your current directory, then the relative pathname for file2 would be bin/file2. If you use a C shell, TC shell, or the Bourne-Again shell, UNIX provides some abbreviations for a few special directories. The character —ll (tilde) refers to your home directory. The home directory of any user (including you, if you want it) can be abbreviated from /parent-directories/userid to ~userid. Likewise, you can abbreviate /parent-directories/youruserid/file to ~/file. The current directory has the abbreviation .(period). The parent of the current directory uses .. (two consecutive periods) as it's abbreviation. **Displaying Directories** When you initially log in, the UNIX system places you in your home directory. The pwd command will display the full pathname of the current directory you are in. pwd /home/userid By typing the ls -a command, you can see every file and directory in the current directory, regardless of whether it is in your home directory. To display the contents of your home directory when it is not your current directory, enter the ls command followed by the full pathname of your home directory. ls /home/userid

If you use a shell other than the Bourne shell use the tilde symbol (~) with the ls command to display the contents of your home directory, instead of typing the full pathname. ls ~ To help you distinguish between files and directories in a listing, the ls command has a -F option, which appends a distinguishing mark to the entry name showing the kind of data it contains: no mark for regular files; —ll for directories; —@ll for links; —*ll for executable programs ls -F ~ **Changing Directories** To change your current directory to another directory in the directory tree, use the cd command. For example, to move from your home directory to your projects directory, type: cd projects (relative pathname from home directory) or, cd ~/projects (full pathname using ~) or, cd /home/userid /projects (full pathname) Using pwd will show you your new current directory. pwd /home/userid/projects To get back to the parent directory of projects, use the special —..ll directory abbreviation. cd .. pwd /home/userid If you get lost, issuing the cd command without any arguments will place you in your home directory. It is equivalent to cd ~, but also works in the Bourne shell. **Moving Files between Directories** You can move a file into another directory using the following syntax for the mv command: mv source-filename destination-directory

EXAMPLE `mv example.txt ~/projects` moves the file `example.txt` into the `projects` directory. Since the `mv` command is capable of overwriting files, it would be prudent to use the `-i` option (confirmation prompt). You can also move a file into another directory and rename it at the same time by merely specifying the new name after the directory path, as follows: `mv sample.txt ~/projects/newsample.txt` Copying Files to Other Directories As with the `mv` command, you can copy files to other directories: `cp sample.txt ~/projects` Like `mv`, the new file will have the same name as the old one unless you change it while copying it `cp sample.txt ~/projects/newsample.txt` Renaming Directories You can rename an existing directory with the `mv` command: `mv oldDirectory newDirectory` The new directory name must not exist before you use the command. The new directory need not be in the current directory. You can move a directory anywhere within a file system. Removing Directories To remove a directory, first ensure that you are in the parent of that directory. Then use the command `rmdir` along with the directory's name. You cannot remove a directory with `rmdir` unless all the files and subdirectories contained in it have been erased. This prevents you from accidentally erasing important subdirectories. You can erase all the files in a directory by first going to that directory (use `cd`) and then using `rm` to remove all the files in that directory. The quickest way to remove a directory and all of its files and subdirectories (and their contents) is to use the `rm -r` (for recursive) command along with the directory's name. For example, to empty and remove your `projects` directory, move to that directory's parent, then type: `rm -r projects` (removes directory and its contents) File and Directory Permissions It is important to protect your UNIX files against accidental (or intentional) removal or alteration. The UNIX operating system maintains information, known as permissions, for every file and directory on the system. This section describes how to inspect and change these permissions.

UNIX was designed and implemented by computer scientists working on operating system research. A low concern for security is one of the hallmarks of the UNIX operating systems. Therefore, unless you act to restrict access to your files, chances are high that other users can read them. Every file or directory in a UNIX file system has three types of permissions (or protections) that define whether certain actions can be carried out. The permissions are: read (r) A user who has read permission for a file may look at its contents or make a copy of it. For a directory, read permission enables a user to find out what files are in that directory. write (w) A user who has write permission for a file can alter or remove the contents of that file. For a directory, the user can create and delete files in that directory. execute (x) A user who has execute permission for a file can cause the contents of that file to be executed (provided that it is executable). For a directory, execute permission allows a user to change to that directory. For each file and directory, the read, write, and execute permissions may be set separately for each of the following classes of users: User(u) The user who owns the file or directory. Group(g) Several users purposely lumped together so that they can share each other's files. Others(o) The remainder of the authorized users of the system. The primary command that displays information about files and directories is `ls`. The `-l` option will display the information in a long format. You can get information about a single UNIX file by using `ls -l filename`. Each file or subdirectory entry in a directory listing obtained with the `-l` option consists of seven fields: permission mode, link count, owner name, group name, file size in bytes, time of last modification, and the filename (the group name appears only if the `—gll` flag is also specified, as in `ls -lg`). The first 10 characters make up the mode field. If the first character is a `—dll` then the item listed is a directory; if it is a `—--` then the item is a file; if it is an `—lll` then it is a link to another file. Characters 2 through 4 refer to the owner's permissions, characters 5 through 7 to the groups' permissions (groups are defined by the system administrator), and the last three to the general public's permissions. (You can type `id` to verify your userid and group membership.) If a particular permission is set, the appropriate letter appears in the corresponding position; otherwise, a dash indicates that the permission is not given. The second field in the output from `ls -l` is the number of links to the file. In most cases it is one, but other users may make links to your files, thus increasing the link count.while using links your copies of their files can be counted against them by the file quota system available on certain UNIX variants. This is why making links other than symbolic links with other people's files is strongly discouraged. The third field gives the userid of the owner of the file. The group name follows in the fourth field (if the `-g` option is used in conjunction with `-l`). The next two fields give

55%

MATCHING BLOCK 6/33

W

the size of the file (in bytes) and the date and time at which the file was last modified. The last field gives the name of the file.

`ls -l myfile -rw-r—r— 1 owner 588 Jul 15 14:39 myfile` A file's owner can change any or all of the permissions with the `chmod` (change mode) command. The `chmod` command allows you to dictate the type of access permission that you want each file to have. In the above example the current permissions for `myfile` are read for everybody, write for the owner, and execute by no one. The arguments supplied to `chmod` are a symbolic specification of the changes required, followed by one or more filenames. The specification consists of those permissions that are to be changed. Changes are for `u` for user (owner), `g` for (group), `o` for (others), or some combination thereof (a (all) has the same effect as `ugo`). Permissions can be changed by specifying `+` (adds a permission), `-` (removes a permission), and `=` (sets the specified permissions) and which permission to add or remove (

100%

MATCHING BLOCK 4/33

SA

BCA 303 LINUX.docx (D53176565)

r for read, w for write, and x for execute).

For example, to remove all the permissions from myfile: `chmod a-rwx myfile` `ls -l myfile` `----- 1 owner 588 Jul 15 14:41 myfile` To allow

61%

MATCHING BLOCK 8/33

W

read and write permissions for all users: `chmod ugoa+rw myfile` `ls -l myfile` `-rw-rw-rw- 1`

owner 588 Jul 15 14:42 myfile To remove write permission for your groups and other users: `chmod a= myfile` achieves the same effect.

`chmod go-w myfile` `ls -l myfile` `-rw-r--r-- 1 owner 588 Jul 15 14:42 myfile` Finally, to allow only read permission to all users: `chmod a=r myfile` `ls -l myfile` `-r--r--r-- 1 owner 588 Jul 15 14:43 myfile` Now allowing read only access protects the file; it cannot be written to or executed by anyone, including you. Protecting a file against writing by its owner is a safeguard against accidental overwriting, although not against accidental deletion. `chmod` will also accept a permission setting expressed as a 3-digit octal number. To determine this octal number, you first write a 1 if the permission is to be set and a 0 otherwise. This produces a binary number that can be converted into octal by grouping the digits in threes and replacing each group by the corresponding octal digit according to the table below. SYMBOLIC TO OCTAL CONVERSIONS SYMBOLIC BINARY OCTAL — 000 0 —x 001 1 - w- 010 2 -wx 011 3 r- 100 4 r-x 101 5 rw- 110 6 rwx 111 7 Thus, if the setting you want is `rw-r--r--`, determine the octal number with the following method: Fig. 2.2 Binary and Octal values for various read write and execute access

This shows that the octal equivalent of `rw-r--r--` is 644. The following example illustrates that the permissions for myfile have been reset to the values with which we began. `chmod 644 myfile` `ls -l myfile` `-rw-r--r-- 1 owner 588 Jul 15 14:44 myfile` To change the permissions back to read only, you can execute `chmod` as follows: `chmod 444 myfile` `ls -l myfile` `-r--r--r-- 1 owner 588 Jul 15 14:45 myfile` As with files, directories may also have permissions assigned. When listing directories, you may use the `-d` option to keep from descending into the directories you list. Otherwise, the contents of the directories will be displayed as well as their names. Given below is an example of permissions assigned to a directory: `ls -l` `gd home drwxrwxr-x 1 owner caam223 588 Jul 15 9:45 home` The files and directories under the directory home, may be read and executed by anyone, but written to only by the owner and users in the caam223 group. Assuming you are the owner of this directory, you may decide to change the permission to allow only yourself and the caam223 group to read and execute files in the home directory. You would set the permissions accordingly: `chmod o-rx home` `ls -l` `gd home drwxrwx-- 1 owner caam223 588 Jul 15 9:46 home` You may decide that only you can alter the contents of the directory for which you may want to remove the write permission for the group. `chmod 750 home` `ls -l` `gd home drwxr-x-- 1 owner caam223 588 Jul 15 9:48 home` An alternative to the previous command is `chmod g-w`. When you create a file the system gives it a default set of permissions. These are controlled by the system administrator and will vary from one installation to another. If you would like to change the default permissions available to you, choose your own with the `umask` command. Note that the permission specified by the `umask` setting will be applied to the file, unlike that specified in the `chmod` command, which normally adds or deletes (few people use the `=` operator to `chmod`).

First issue the command without arguments to cause the current settings to be echoed as an octal number: `Umask 022` If you convert these digits to binary, you will obtain a bit pattern of 1.s and 0.s. A 1 indicates that the corresponding permission is to be turned off, and 0, which means it is to be turned on. (Note that the bit patterns for `chmod` and `umask` are reversed.) Hence the mask output above is 000010010, which produces a permission setting of `rwr-r-` (i.e., write permission is turned off for group and other). Newly created files always have the execution bit turned off. Suppose you decide that the default settings you prefer are `rwr--`. This corresponds to the masking bit pattern 000010111, and so the required mask is 026: `umask 26` Now, if you create a new file during this session, the permissions assigned to the file will be the ones allowed by the mask value. Wildcard Characters Using wildcard characters that allow you to copy, list, move, remove, etc. items with similar names is a great help in manipulating files and directories. The symbol `—?` will match any single character in that position in the file name. The symbol `—*` will match zero or more characters in the name. Characters enclosed in brackets `[]` will match any one of the given characters in the given position in the name. A consecutive sequence of characters can be designated by `[char char]`. `?ab2` will match a name that starts with any single character and ends with `ab2`. `?ab?` will match all names that begin and end with any character and have `ab` in between. `ab*` would match all names that start with `ab`, including `ab` itself. `a*b` would match all names that start with `a` and end with `b`, including `ab`. `s[aqz]` would match `sa`, `sq`, and `sz`. `s[2-7]` would match `s2`, `s3`, `s4`, `s5`, `s6` and `s7`. These wildcard symbols help in dealing with groups of files, but you should remember that the instruction: `rm *`

would erase all files in your current directory (although by default, you would be prompted to okay each deletion). The wildcard `*.*` should be used carefully. FORMATTING AND PRINTING COMMANDS COMMAND PURPOSE `lpq` to view printer queue `lpr` to send file to printer queue to be printed `lprm` to remove job from printer spooling queue `enscript` to convert text files to POSTSCRIPT format for printing TELNET Telnet programs are a type of terminal emulation program, just like modem-based communications programs. The difference is that they allow you to access other computers through the Internet, rather than dial other computers directly. Telnet programs talk to other Internet-connected computers using what's called the telnet protocol for communication. This protocol specifies how telnet programs should send data back and forth, allowing for interactive text sessions. The following is an example of an interactive session, with the remote computer displaying text, then waiting for the user to type some text. Terminal Emulation In order for more than one person to use the computer, terminals were connected to the computer. At first these terminals were teletypes – just fancy typewriters called TTYs. TTY's were made to print characters, do line feeds, carriage returns, and read input. Then CRT (video) terminals were introduced, and ways of handling characters displayed on a screen had to be developed. These early terminals included the DEC family of VTs (Video Terminals) like the VT-52 and the VT-100. They allowed users to do full screen editing by moving a cursor around the screen and deleting characters at will. They also allowed the computers to print special characters like lines and dashes, display text in bold, or clear the screen entirely. We still use this emulation for remote login sessions like Telnet. The emulation of old terminal types, like TTY or VT- 100, are good ways of controlling a computer through another computer because they were character based and were designed to transmit and receive data quickly and efficiently. There are various arguments that can be used with the TELNET and this is tabulated below.

VARIOUS OPTIONS WITH TELNET ARGUMENTS DESCRIPTION `-8` Uses a eight bit data path. Negotiates BINARY on both input and output `-E` Stops any character from being recognized as a escape character `-K` Disables automatic login. `-L` Uses a eight bit data path. Negotiates BINARY on output. `-X atype` Disable authentication type atype. `-a` Automatic login into remote system. `-d` Toggles socket level debugging and useful only to the root host Indicates the host's official name or internet address of remote host. `port` Indicates the port number or the default TELNET port used. Once a connection has been opened, TELNET will enter the input mode. TELNET will attempt to enable the TELNET linemode option. If this fails, then TELNET will revert to one of the two input modes: either the line by line mode or the character at a time mode, depending on what the remote system supports. When linemode is enabled, character processing will be done on the local system while under the control of the remote system. When input editing or character echoing is to be disabled, the remote system will relay that information. The remote system will also relay changes to any special characters that happen on the remote system, so that they can take effect on the local system. If the linemode option is enabled or if the localchars toggle is TRUE (the default value for the old line by line mode), the user's quit, intr, and flush characters will be trapped locally and sent as TELNET protocol sequences to the remote machine. If LINEMODE had been enabled at any earlier time, then the user's susp and eof characters will also be sent as TELNET protocol sequences. quit will be sent as a TELNET ABORT instead of `>Break<`. The options toggle autoflush and toggle autosynch which cause this action to flush any subsequent output to the terminal (until the remote host acknowledges the TELNET sequence) and to flush previous terminal input (in the case of quit and intr). In the line by line mode, all text will be echoed locally, but (normally) only completed lines will be sent to the remote host. The local echo character (initially `^E`) may be used to enable and disable the local echo mode; normally, this would be used only for entering passwords so that they are not echoed. In the character at a time mode, most entered text will be sent immediately to the remote host for processing.

NETWORKING When personal computers were created, the only way to transfer files from one computer to another was to save the file from one computer to a data storage, such as floppy disk, and copy it from the disk, to the destination computer. The computer network was created to reduce this burden and make file transfer easy. There are 4 main layers in a TCP/IP network structure. The first one is the application layer, which applies to all the applications in the OS that uses the network system. The second is the TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) layer, also called as the Transport Layer, which breaks up the file or information to be sent into small packets, and reassembles them depending on whether the file is to be sent or received. The next layer, the IP layer, or the Data-Link Layer, is the main network protocol. This layer makes sure that the information broken up into packages in the Transport Layer gets to the destination, and also controls how the packaged information should get sent to the destination. And finally, there is the Physical Layer, which deals with mechanical, electrical and procedural interfacing of the raw bit data. The content of the application layer is up to the individual user. Any programs that use the network to receive or send information constitute this Layer. Fig. 2.3 Layers in TCP/IP Some of the most commonly used applications in the application level are: ? TELNET ? FTP ? WEB BROWSERS (Netscape Navigator, IE 5.0) ? PING ? TRACEROUTE ? FINGER The Transport Layer The function of this Layer is to break up the information to be sent through the net and put together the information received from other sources through the Internet. When sending an information, or file, the file is broken up into several small packages. The Layer then adds a header into each of the packages to be sent. The header contains Information about packet number, Number of the first elementary

fragment in the packet and the End of packet bit. By this line, the package length should not be more than 1008 bits, including the header. Fig. 2.4 Splitting of the information into headers and formation Another function of this layer is to choose a secure transmission of the packages (TCP), or a non-reliable transmission of the package (UDP). The difference between these two protocols lies in its implementation. To put it simply, the TCP protocol is a 100% reliable protocol that makes sure the packages gets to its destination, unlike UDP, which sends the packages and finishes its job. The TCP protocol is a connection- oriented protocol, meaning that the sender and receiver setup a connection between them and sends/receives the files. Each fragment has a header that contains the packet number, Number of the first elementary fragment in the current packet, and the End of packet bit, as described above. The packet number is the number assigned to each packet (0-?). The second part of the header tells you the first byte-number for each packet. If one packet is broken into 2 parts, the second part's header will contain the first byte-number that the fragment contains: like in the image above. The third part of the header file is constituted of 1 byte, and it tells whether the packet is the last packet. If the bit is put to 1, then it means it is the end of the file, and if it's 0 is not. The Data-Link Layer The IP layer mainly controls the routing of the packages to be sent. In the Internet, there are many gateways to chose from .Choosing the best path or route is very important, since if one of the paths has too much traffic (the server or the gateway might be down as well), the source computer should send the packages through a different route. To accomplish this, the IP layer puts another header into the packages passed by the Transport Layer, with the source and destination addresses, and the routing information. Once the headers are put into the packages and it is ready to send them then the layer sends signals to the destination computer. These signals (SYN, DLE, STX) get the destination computer ready to receive the packages that will be sent by the source. Once the last package from the source computer is sent, then the source computer sends other signals to end the transmission (DLE, ETX, CRC). Fig. 2.5 Information split up in the DataLink Layer

Physical Layer This layer mainly deals with the sending of raw data over the network. The main issue that this Layer deals with is the physics of sending data. Datas are a combination of bytes, and 1 byte contains 8 bits. Each of the bits uses the binary presentation: either 0 or 1. The physical layer represents these two bits with a high and low frequency of the electrical current.

SOCKETS A Socket creates an endpoint for communication and returns a descriptor .s is a file descriptor returned by the socket system call. The domain parameter specifies a communications domain within which communication takes place and this in turn selects the protocol family, which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the sockets. These families are defined in the include file `<sys/socket.h>`. The currently supported domains are AF_INET (Internet protocols) and AF_UNIX. The socket has the indicated type, which specifies the semantics of communication. Currently defined types are: SOCK_STREAM A SOCK_STREAM type provides sequenced, reliable, two-way connection-based byte streams with an out-of-band data transmission mechanism SOCK_DGRAM A SOCK_DGRAM socket supports datagrams, which are connectionless, unreliable messages of a fixed maximum length. SOCK_RAW SOCK_RAW sockets provide access to internal network protocols and interfaces. This type is available only to the superuser. The AF_INET domain supports all of the above listed types. The AF_UNIX domain supports only the SOCK_STREAM and SOCK_DGRAM types. Note that all protocol families do not support all types. The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified. The protocol number to use is particular to the communication domain in which communication is to take place. Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect call and if the connect call is unsuccessful, a second connect call to the same socket is not allowed. Once connected, data may be transferred using read and write calls or some variant of the send and recv calls. When a session has been completed, a close may be performed. If the connection fails at any time during the session (because of a network failure, for example), the socket cannot be re-used. The current socket must be closed and a new socket created. Out-of-band data may also be transmitted over a SOCK_STREAM. The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space and cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable errno. Some protocols optionally keep sockets warm by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period (for example, 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit. SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send calls. Datagrams are generally received with recv, which returns the next datagram with its return address. An ioctl call can be used to specify a process group to receive a SIGURG (SIGUSR1) signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events with SIGIO (SIGPOLL) signals. The operation of sockets is controlled by socket level options. These options are defined in the file `<sys/socket.h>`. setsockopt and getsockopt. They are used to set and get options, respectively. Return values A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

Chapter 3- Standard Devices & Vi INTRODUCTION TO VI The vi editor is a commonly used editor for Unix systems in that it makes use of a regular keyboard with an escape key. Complete documentation is available by typing the following at the Unix prompt. `man vi` It has two editors, one for line editing and another for visual mode editing. Visual mode is a better full screen editor than most, and it runs faster than its rivals that have a larger bag of screen-editing commands. Line editing mode dwarfs the global search and replaces facilities found in word processors and simple screen editors. Its only rivals are non-visual editors like `ed` where you must know in advance exactly what you want to do. But in the Vi/Ex editor, the two sides are very closely linked, giving the editor a combination that no other editor can rival. However, it does not format text and cannot be used as a word processor. It does not have built-in facilities for editing binaries, graphics, tables, outlines, or any programming language except Lisp. A large part of the power of vi requires custom programming. The editor vi performs no editing operations on the file that you name during invocation. Instead, it works on a copy of the file in an editing buffer. When vi is invoked with a single filename argument, the named file is copied to a temporary editing buffer. The editor remembers the name of the file specified at invocation, so that it can later copy the editing buffer back to the named file. The contents of the named file are not affected until the changes are copied back to the original file.

Emacs Tutorial Emacs is a Text Only word processor. You cannot use the mouse with this software. Also, it does not support different fonts, font styles, symbols, embedded graphics, etc. It only handles Text. The advantage of programs like vi and Emacs is that they are fast. Emacs can open up a several megabyte text file, edit it and save the results in the time it takes to start up Microsoft Word. If you want a X-Window graphical user interface to EMACS users can use PMGM. Emacs commands generally involve the control key (sometimes labeled CTRL or CTL) or the META key (sometimes labeled EDIT). You can also prefix a character rather than write out Meta or Control each time.

VI EDITOR Vi is always in one of three modes. Command mode, ex escape mode or insert mode. Command mode is the start-up mode of vi. While the ex escape mode begins when users enter a command that starts with a colon the insert mode starts once you have pressed i, a, o (or the uppercase versions) for insert, append, or open line mode, or one of the change text keys. Once you are in insert mode, what you type on the keyboard is added to your editing buffer until you press Escape. These are the changes you see displayed on the screen as you type. The usual process of editing involves moving around in the document while in command mode and using arrow keys and page keys, and then switching into one of the insert modes to make the changes. There are various clones of vi editor available in the market some of which are available absolutely free. The most important of the clones are vim and winvi which works on DOS and Windows.

Command Mode In command mode, the alternatives to the page keys are Control-U and Control-D (hold down the control key and press U or D simultaneously). Control-D will produce a half-page up, and CONTROL-D will produce a half-page down of the screen. Control-B and Control-F will move you backward or forward one screen. These four keys eliminate the need for page up and page down keys (previous and next on some keyboards). The arrow keys can be replaced by the single characters h, j, k and l which will produce single cursor position movements as follows: h moves the cursor one position to the left l moves the cursor one position to the right j moves the cursor one line down k moves the cursor one line up These keys are not terminal dependent and give user a fall back for the arrow keys when the arrow keys fail.

Starting vi Different methods of starting vi allow for some options that may speed up editing. The most useful option is to open a file and immediately position a word in the file. `vi +/hello example.txt` This command opens the file, `example.txt`, and positions to the first line containing the word `hello`. User can also include a phrase of more than one word by enclosing it in quotes as in the following example: `vi +/llwelcome to vi editorll example.txt` Alternative Once a file is open, anything that can be done with vi can be done on a standard alphanumeric keyboard that includes a Control key and an Escape key. There are keys and control sequences that can replace the arrow keys or page up and down keys. This is a handy feature of vi that overcomes some problems with remote logins. There are several different configurations in which users end up with the arrow keys but not the page up, page down keys, or vice versa. This can make editing difficult unless alternatives are known. A user can correct all this by defining a TERM variable. While using communications software on a PC, users can usually re-map their keyboard to send the correct character sequences to vi when the arrow and other keys are pressed. Either of these two options is necessary for frequent logins. When this is only an occasional problem, there is no need to struggle through `termcap`, `terminfo`, or user PC telecommunications configuration and software manual. Instead user can mimic the behavior of keys when vi is in command mode.

The Use of Numbers When using arrow keys or letters, the single step movement keys can be usefully modified by adding a number prior to the key. `6j` or `6` down arrow will move the cursor down 6 lines, and `11h` or `11` left arrow will move the cursor 11 characters to the left. These keys only work in command mode. Typing `15h` in insert mode will place the characters `—15hll` in the document. There are many additional commands for moving around the document in the command mode. `$` moves the cursor to the end of the current line. The `$` can be preceded by a count. The `$` command assumes a count of 1, so `$` and `1$` are equivalent. `5$` will move the cursor

53%

MATCHING BLOCK 9/33

SA

unixassignment-1(14691A0532).docx (D25928407)

from the current line to the end of the fifth line. `^` moves the cursor to the beginning of the current line. The `^` actually moves the cursor

to the

first non-space character in the line. A 0 (zero) will move the cursor all the way to the left. Counts do not work with ^ and 0. Users can also skip around in larger chunks. The w character moves the cursor to the beginning of the next word. w works with counts, 6w will move you forward 6 words and so on. The b character moves

63%

MATCHING BLOCK 7/33

W

the cursor to the beginning of the current word if it is within a word, or the previous word

if it not. This works as counts too. Moving

50%

MATCHING BLOCK 11/33

W

By Sentences and Paragraphs Sentences The left parenthesis —(— will move the cursor to the beginning of a sentence, and the

right parenthesis —) to the end. A sentence is defined as one or more words ending with a period (.), exclamation point (!) or question mark (?) and followed by two spaces.

Paragraphs There are also paragraph movement keys. Paragraphs are groups of text delimited by blank lines. a{ takes

55%

MATCHING BLOCK 12/33

SA

DECAP448_LINUX_AND_SHELL_SCRIPTING.pdf
(D142327428)

the cursor to the beginning of a paragraph, and a} moves it to the end of a paragraph.

These movement keys alone are quite powerful, but they can additionally be modified with counts. Entering 3{ will move the cursor forward to the beginning of the third paragraph from the current cursor position. Escape Mode The ex escape mode provides some other useful movement commands. Users can move directly to a line they want to in a text file by typing a colon followed by the line number of the line. EXAMPLE The command: 16 will move the cursor to line 16. The command: +5 will move you down five lines in the text. The command: -2 will move you back two lines. The command: :1 will move you to the top of the file. While \$ moves you to the end of the line, :\$ moves you to the end of the file. It is also possible to move back to a position you have marked. Mark a position in your document in command mode by placing the cursor at the position to mark. Type m followed by a lower case letter a-z. The letters represent 26 marks that can be placed in the document. For example, entering mb will mark the current cursor position as b. The mark does not show up in the users edited text, but is hidden in the edit buffer for later use. However the mark is lost when users close the file and exit vi. Users can return to the mark by typing a single quote (.) followed by a letter, or a single quote turned backwards (') followed again by the letter. The _b version returns you to the beginning of the line containing the mark b. The `b version will move the cursor to the exact position of the b mark. When users are reviewing two or more areas of a text file, simply mark them and flip back and forth using the single quote. SEARCH PATTERNS In quite a few functions of the editor, users can use the string-pattern search to indicate where something is to be done or how far some effect is to extend. These search patterns are a good example of an editor function that is very much in the Unix style, but is not exactly the same in detail as search patterns in any other Unix utility. Search patterns function in both line editing and visual editing modes, and they work the same way, with just a few exceptions and search patterns vary with the circumstances.

By default the magic is set to be on. If `nomagic` is set, the number of regular expression meta characters is greatly reduced, with only caret (^) and dollar sign (\$) having special effects. In addition, the meta characters tilde (~) and ampersand (&) in replacement patterns are treated as normal characters. All the normal meta characters may be made magic when `nomagic` is set by preceding them with a backslash (\). Table of Search Pattern Meta Characters . A period in a search pattern that matches any single character, whether a letter of the alphabet (upper or lower case), a digit, a punctuation mark, in fact, any ASCII character except the newline. So to find `—default value`ll when it might be spelled `—default-value`ll or `—default/valu`ll or `—default_valu`ell etcetera, use `/default.value/` as your search pattern. When the editor variable `magic` is turned off, you must backslash the period to give it its meta value. * An asterisk, plus the character that precedes it, matches any length string (even zero length) of the character that precedes the asterisk. So the search string `/ab*c/` would match `—ac`ll or `—abc`ll or `—abbc`ll or `—abbbc`ll and so on. (To find a string with at least one `—b`ll in it, use `/abb*c/` as your search string). When the asterisk follows another meta character, the two match any length string of characters that the meta character matches. That means that `/a.*b/` will find `—all` followed by `—b`ll with anything (or nothing) between them. When the editor variable `magic` is turned off, you must backslash the asterisk to give it its meta value. ^ A circumflex as the first character in a search pattern means that a match will be found only if the matching string occurs at the start of a line of text. It does not represent any character at the start of the line, and a circumflex anywhere in a search pattern except as the first character will have no meta value. So `/^cat/` will find `—cat`ll, but only at the start of a line, while `/cat^/` will find `—cat`ll anywhere in a line. \$ A dollar sign as the last character in a search pattern means that the match must occur at the end of a line of text. Otherwise it is the same as circumflex, above. \&t; At the start of a search pattern, a backslashed left-angle bracket means that the match can only occur at the start of a simple word; at any other position in a search pattern it is not a meta character. (In this editor, a `—simple`ll word is either a string of one or more alphanumeric character(s) or a string of one or more non-alphanumeric, non-whitespace character(s), so `—shouldn't`ll contains three simple words). Thus `/\&t;cat/` will find the last three characters in `—the cat`ll or in `—tom-cat`ll, but not in `—tomcat`ll. To remove the meta value from the left angle bracket, remove the preceding backslash: `/&t;cat/` will find `—&t;cat`ll regardless of what precedes it. \&t; At the end of a search pattern, a backslashed right angle bracket means that the match can occur only at the end of a simple word. Otherwise it is the same as the left angle bracket, above. ~ The tilde represents the last string you put into a line by means of a line mode substitute command, regardless of whether you were in line mode then or ran it from visual mode by preceding it with a colon (:~). For instance, if your last line mode substitution command was `s/dog/cat/` then a `/the ~/` search pattern will find `—the cat`ll. But the input string of a substitute command can use meta characters of its own, and if your last use involved any of those meta characters then a tilde in your search pattern will give you either an unexpected error message or a match that is not what you expected. When the editor variable `magic` is turned off, you must backslash the tilde to give it its meta value.

Searching from Current Location The more common use for search patterns is to go to a new location in the file, or make some editing changes that will extend from your present position to the place the pattern search finds. (In line editing mode it's also possible to search from one pattern's location to where another pattern is found, but both searches still start from your present location). If you want to search forward in the file from your present location (toward the end of the file), precede the search pattern with a slash (/) character, and type another to end the pattern. So if you want to move forward to the next instance of the string `—j++`ll in your file, type: `/j++/ (ret)` will do it. When there is nothing between the pattern and the RETURN key, the RETURN itself will indicate the end of the search pattern, so the second slash is not necessary. In visual mode, the ESCAPE key works as well as RETURN does for ending search inputs, so `/j++ (esc)` is yet another way to make the same request from visual mode. To search backwards (toward the start of the file), begins and ends with a question mark instead of a slash. The same rules of abbreviation apply to backward searches. Hence `?j++?` (ret) `?j++ (esc)` are all ways to head backwards in the file to the same pattern. Either way, request for a pattern search and the direction the search is to take in just one keystroke. If the search is backwards, any matching pattern the editor finds will be above the users present position in the file, and vice versa if you search forward. The editor looks there first, certainly, but if it gets to the top or bottom line of the file and hasn't found a match yet, it wraps around to the other end of the file and continues the search in the same direction. That is, if you use a question mark to order a backward search and the editor searches all the way through the top line of the file without finding a match, it will go on to search the last line followed by second last line, and so on until (if necessary) it gets back to the point where the search started. On the other hand if you were searching forward and the editor found no match up through the very last line of the file, it would search the first line, and then the second line, etc. If you don't want searches to go past either end of the file, you'll need to type in a line mode command: `:set nowrapscan (ret)` This will disable the wraparound searching during the present session in the editor. If you want to restore the wraparound searching mechanism before you leave the editor, type `:set wrapscan (ret)` You can turn this on and off as often as you like.

Global Search (Find them all search) The earlier search was about finding just one instance of the pattern; the one closest to your current location in the file, in the direction you chose for the search. But there is another style of search, used primarily by certain line editing mode commands, such as `global` and `substitute`. This search finds every line in the file (or in a selected part of the file) that contains the pattern and operates on them all. Both styles of search pattern are used in one command line. But the `find-one-instance` pattern or patterns will go before the command name or abbreviation, while the `find-them-all` pattern will come just after it. For example, in the command:

`:/Chapter 10?./The End/substitute/cat/dog/g (ret)` The first two patterns refer to the preceding line closest to the current line that contains the string `—Chapter 10` and the closest following line containing the string `—The End`. Note that each address finds only one line. Combined with the intervening comma, they indicate that the substitute command is to operate on those two lines and all the lines in between them. But the patterns immediately after the substitute command itself tell the command to find every instance of the string `—cat` within that range of lines and replace it with the string `—dog`. Aside from the difference in meaning, the two styles also have different standards for the delimiters that mark pattern beginnings and (sometimes) endings. With a `find-` them-all pattern, there's no need to indicate whether to search forward or backward. Thus, you are not limited to slash and question mark as your pattern delimiters. Almost any punctuation mark will do, because the editor takes note of the first punctuation mark to appear after the command name, and regards it as the delimiter in that instance. So `:/Chapter 10?./The End/substitute;cat;dog;g (ret)` `:/Chapter 10?./The End/substitute+cat+dog+g (ret)` `:/Chapter 10?./The End/substitute{cat{dog{g (ret)` are all equivalent to the substitution command above. (Avoid using punctuation marks that might have a meaning in the command, such as an exclamation point, which often appears as a switch at the end of a command name). The slash mark appears as itself in the search pattern. For example, suppose our substitution command above was to find each pair of consecutive slash marks in the text, separated by a hyphen - that is, change `//` to `/-`. Obviously, `:/Chapter 10?./The End/substitute/////-/g (ret)` Won't work; the command will only regard the first three slashes as delimiters, and everything after that as extraneous characters at the end of the command. This can be solved by backslashing: `:/Chapter 10?./The End/substitute/\/\/-/g (ret)` but this is even harder to type correctly than the first attempt was. But with another punctuation mark as the separator

`:/Chapter 10?./The End/substitute;/-/g (ret)` the typing is easy and the final command is readable. Simple search patterns The simplest search pattern is just a string of characters you want the editor to find, exactly as you've typed them in. For instance: `—the cat`. But, already there are several caveats: This search finds a string of characters, which may or may not be words by themselves. That is, it may find its target in the middle of the phrase. Whether the search calls `—The Cat` a match or not depends on how you've set an editor variable named `ignorecase`. If you've left that variable in its default setting, the capitalized version will not match. If you want a capital letter to match its lower-case equivalent, and vice versa, type in the line mode command. `:set ignorecase (ret)` To resume letting caps match only caps and vice versa, type `:set noignorecase (ret)` The search will not find a match when `—thell` occurs at the end of one line and `—cat` is at the start of the next line. It makes no difference whether if there is or isn't a space character between one of the words and the linebreak. Finding a pattern that may break across a line ending is a practically impossible task with this line-oriented editor. Where the search starts depends on which editor mode you are using. A search in visual mode starts with the character next to the cursor. In line mode, the search starts with the line adjacent to the current line. Meta Characters There are search meta characters or wild card characters that represent something other than themselves in the search. As an example, the meta characters `.` and `*` in `/Then .ed paid me $50*!/ (ret)` can cause the pattern to match any of the following

`Then Ted paid me $5!` `Then Red paid me $5000!` `Then Ned paid me $50!` or a myriad of other strings. Meta characters are what give search patterns their real power. Users must know the varied uses of the backslash (`\`) meta character in turning the wild card value of meta characters on and off. In many cases, the meta value of the meta character is on whenever the character appears in a search pattern unless it is preceded by a backslash; when the backslash is ahead of it the meta value is turned off and the character simply represents itself. As an example, the backslash is a meta character by itself, even if it precedes a character that does not have a meta value. The only way to put an actual backslash in your search pattern is to precede it with another backslash to remove its meta value. That is, to search for the pattern `a\b`, type `/a\\b/ (ret)` as your search pattern. If you type `/a\b/ (ret)` the backslash will be interpreted as a meta character without any effect (since the letter `b` is never a meta character) and your search pattern will find the string `ab`. Less-often-used meta characters are used in exactly the opposite way. This sort of character represents only itself when it appears by itself. You must use a preceding backslash to turn the meta value on. For example, in `/\>cat/` the left angle bracket (`>`) is a meta character; in `/>cat/` it only represents itself. These special meta characters are pointed out in the list below. There is a third class which is the most difficult to keep track of. Usually these meta characters have their meta values on in search patterns, and must be backslashed to make them represent just themselves: like our first example, the backslash character itself. But if you have changed the default value of an editor variable named `magic` to turn it off, they work oppositely. You must then backslash them to turn their meta value on: like our second example, the left angle bracket (not that you are likely to have any reason to turn `magic` off). These oddities are also noted in the list below.

The punctuation mark that initiates and ends your search pattern, be it a slash or a question mark or something else. Whatever it is, if it is also to appear as a character in the pattern you are searching for, you will have to backslash it there to prevent the editor viewing it as the end of the pattern. Character Classes There is one meta string form (a multi-character meta character) used in search patterns. When several characters are enclosed within a set of brackets ([]), the group matches any one of the characters inside the brackets. That is, /part [123]/ will match part 1, part 2 or part 3, whichever the search comes to first. One frequent use for this feature is in finding a string that may or may not be capitalized, when the editor variable ignorecase is turned off (as it is by default). Typing /[Cc]at/ will find either —Catll or —catll, and /[Cc][Aa][Tt]/ will find those or —CATll. In case there was a slip of the shift key when —CATll is typed in, the last pattern will even find CaT, CA, etcetera. Another feature of the metastring is that there can be meta characters inside it. Inside the brackets, a circumflex as the first character reverses the meaning. Now the meta string matches any one character that is NOT within the brackets. A /^[^]/ search pattern finds a line that does not begin with a space character. A circumflex that is not the first character inside the brackets represents an actual circumflex. A hyphen can be a meta character within the brackets, too. When it is between two characters, and the first of the two other characters has a lower ASCII value than the second, it's as if you have typed in all of the characters in the ASCII collating sequence from the first to the second one, inclusive. So /[0-9]%/ will find any numeral followed by the percent sign (%), just as /[0123456789]%/ would. A /[a-z]/ search pattern will match any lower-case letter, and /[a-zA-Z]/ matches any letter, capital or lower case. These two internal meta characters can be combined: /[^A-Z]/ will find any character except a capital letter. A hyphen that is either the first or the last character inside the brackets has no meta value. When a character-hyphencharacter string has a first character with a higher ASCII value than the last character, the hyphen and the two characters that surround it are all ignored by the pattern search and so /[ABz-a]/ is the same as /[AB]/. Backslashing character classes is complex. Within the brackets you must backslash a right bracket that's part of the class; otherwise the editor will mistake it for the bracket that closes the class. Of course you must backslash a backslash that is to be a part of the class, and you can backslash a circumflex at the start or a hyphen between two characters if you want them in the class literally and don't want to move them elsewhere in the construct. Elsewhere in a search pattern you will have to backslash a left bracket has to appear as itself, or else the editor will take it as an attempt to begin a character class. Finally, if magic is turned off, you will have to backslash a left bracket when you do want it to begin a character class. LIST OF COMMANDS IN VI EDITOR Starting an Editing Session vi

100%

MATCHING BLOCK 13/33

SA

Linux_system_administration_block_2.pdf (D149208459)

filename where filename is the name of the file to be

edited.

Undo Command u undo the last command. Screen Commands CTL/I Reprints current screen. CTL/L Exposes one more line at the top of screen. CTL/E Exposes one more line at the bottom of screen. CTL/F Pages forward one screen. CTL/B Pages back one screen. CTL/D Pages down half screen. CTL/U Pages up half screen. Cursor Positioning Commands j Moves cursor down one line, same column. k Moves cursor up one line, same column. h Moves cursor back one character. l Moves cursor forward one character. RET Moves cursor to beginning of next line. 0 Moves cursor to beginning of current line. \$ Moves cursor to end of current line. SPACE Moves cursor forward one character. nG Moves cursor to beginning of line n. Default is the last line of file. :n Moves cursor to beginning of line n. b

41%

MATCHING BLOCK 10/33

W

Moves the cursor backward to the beginning of the previous word. e Moves the cursor backward to the end of the previous word. w Moves the cursor forward to the next word. :pattern Moves

cursor forward to next occurrence of pattern. ?pattern Moves cursor backward to next occurrence of pattern. n Repeats last / or ? pattern search.

Text Insertion Commands a Appends text after cursor. Terminated by the escape key. A Appends text at the end of the line. Terminated the escape key. i Inserts text before cursor. Terminated by the escape key. I Inserts text at the beginning of the line. Terminated by the escape key. o Opens new line below the current line for text insertion. Terminated by the escape key. O Opens new line above the current line for text insertion. Terminated by the escape key. DEL Overwrites last character during text insertion. ESC Stops text insertion. The escape key on the DECstations is the F11 key. Text Deletion Commands x Deletes current character. dd Deletes current line. dw Deletes the current word. d) Deletes the rest of the current sentence. D, d\$ Deletes from cursor to end of line. P Puts back text from the previous delete. Changing Commands cw Changes characters of current word until stopped with escape key. c\$ Changes text up to the end of the line. C, cc Changes remaining text on current line until stopped by pressing the escape key. ~ Changes case of current character. xp Transposes current and following characters. J Joins current line with next line. s Deletes the current character and goes into the insertion mode. rx Replaces current character with x. R Replaces the following characters until terminated with the escape key.

Cut and Paste Commands `yy` Puts the current line in a buffer. Does not delete the line from its current position. `p` Places the line in the buffer after the current position of the cursor. Appending Files into Current File `:R games` Inserts the file named `games` where the cursor was before the `—:ll` was typed. Exiting vi `ZZ` Exits vi and saves changes. `:wq` Writes changes to current file and quits edit session. `:q!` Quits edit session (no changes made).

LINE EDITING IN VI EDITOR Whenever a user wants to give an editor command to delete some text, change lower-case letters to capitals, write to a file, etc the editor needs to know what part of the file to go to work on. A few commands have their addresses built in, and most line-mode commands have default addresses that the editor will use if you do not specify however in most cases users need to know how to give the editor an address and more importantly what address to give. Many line-mode commands are almost identical to corresponding commands in visual mode; many more do similar things in different ways. Most of the benefits of these duplicative command sets come from the totally-different addressing styles of line and visual modes. The differing address concepts mean that an edit that would be difficult or impossible to do with one mode's available addresses can be easily done with an address form found in the other mode. Line mode gives you a colon (`:`) prompt for your line mode commands, and prints only an occasional line from the file on your screen. The feel of this mode is very much like giving UNIX commands from your shell prompt. Few people work in line mode these days, largely because you can give most line-mode commands from visual mode, but users can't give any visual-mode commands while you are in line mode. Or perhaps they just prefer the comfortable WYSIWYG feeling of seeing the text on screen, with changes appearing as they are made. But at times users need to work temporarily in line mode. To get to line mode when you first launch the editor, invoke it by typing `—exll` instead of `—vill`. To go to line mode when you are already in the editor's visual mode, enter `—Qll`. To get back to visual mode, type `—vill` followed by a carriage return. You do not need to type that colon when you are giving a command from within the line mode. It may even be harmful. The rule is that if you type a colon at the start of a command from within the line mode, there must be nothing between the colon and the command name or abbreviation. Not an address, not even a space, nothing at all. All line-mode commands are without an initial colon, because if the colon is present users are working in visual mode. Users have to finish with a carriage return because you now realize that any line-mode command, given from either line or visual mode, has to end with a carriage return. `global /^/ move 0 g/^/m0` are identical in their effect. Line-Mode Addressing A single mode address is often all you need with a line-mode command. One address refers to just one line, which tells a command like delete or substitute to operate on that one line only. A command like insert or read, which places something immediately before or after a particular line, has no use for more than one address. A search pattern is always an acceptable line-mode address. Place the address at the start of the command line, before the command name (but after the initial colon if you are giving the command from visual mode), so: `?the [cC]at?` delete will erase the penultimate line that contains the string `—the catll` or `—the Catll`, while: `/^GLOSSARY$/ read gloss.book` puts the contents of the file `—gloss.bookll` right after the next line in the file you are editing that contains only the word `—GLOSSARYll`. There are two shorthand forms for reusing search patterns as addresses. Typing `—??ll` or `—//ll` prompts the editor to use the last search pattern used, and your choice of `—??ll` or `—//ll` will set the direction of the search, overriding the direction chosen the previous time for that search pattern. That is, if you type: `?the cat? yank // delete ?? print` The second command will search forward, to remove the penultimate line containing the string `—the catll`, even though original use of that pattern was in a backward search. The third command will search backward to find the line to print, which (by coincidence) is the direction of the original search. But the search pattern that those preceding abbreviations reuse may not be a pattern you have used to search for a line. If you ran a substitute command after any pattern searches for lines, then the pattern you gave the substitute command to tell it what text to take out of the line is the pattern that will be reused. This is so even if the substitute command began with a search pattern to specify the line on which the substitution was to be performed – the search to find the pattern to be replaced within the line was run after the first search pattern had found the line to operate on, so the search within the line was the last pattern search run. So if we type: `/the cat/ substitute /in the hat/on the mat ?? delete` the second command would, in this case, delete the penultimate line containing `—in the hatll`. To be sure that the pattern that gets reused is the last one used to find a line, use the abbreviations `\?` and `\/` to search backward and forward, respectively. In all other respects these work just as typing `—??ll` and `—//ll` do. Line Number This is also a valid line-mode address. The editor automatically numbers each line in the file consecutively, and this numbering is dynamic – that is, whenever you add or delete lines, the editor rennumbers all the lines following the insertion or deletion point. So if you change some text on line 46 in your file, and then delete lines 11 and 12, the line with the text you changed is now line 44. And if you then add ten new lines after line 17, the line with your changed text on it now automatically becomes line 54. There is never a gap or an overlap in the line number sequence, so the *n*th line in the file is always line number *n*; that is, the 7th line is always line number 7, and so on. To delete the 153rd line in your file, just type: `153 delete` Do not use any delimiters around a line number, or around any other address except a search pattern. There are two symbolic line numbers and one fictional one that can be used in line- mode addresses. As long as there are any lines in the buffer (that is, you have not specified a not-yet-existent file to edit and failed to enter any text so far), the editor regards you as being `_on'` one of them, usually the last line affected by your latest command. A period or dot (`.`) is the symbolic address for this line. The last line in the file also has a symbolic address: the dollar sign (`$$`). So if you should type: `. write << example $ delete` The first command would append a copy of just the line you are on now to a file named `example`, while the second would delete the last line in the file you are editing.

A few commands put text immediately after the line address you give: the append command is one of them. In order to let them put their text at the very start of a file (if that is where you want it), these commands can take the fictitious line number zero (0) as their address. So, if you want to type some text that will appear ahead of anything already in the file, you can do it with either of these command lines: `1 insert 0 append` However note, that insert and append are among the few line-mode commands that cannot be run from visual mode by starting with a colon, because they occupy more than one line including the text to be put in). Writing Own Line Addresses Users can attach lower-case letters to lines as line addresses, and change the attachments whenever they like. Users can even use a special address that is automatically attached to the last line they jumped off from. There are ways to mark a particular line with a lower-case letter of the alphabet, and those ways differ between line and visual modes. Once a line is marked, the line-mode address that refers to that line is just the single-quote character followed immediately by the lower-case letter with which the line was marked. So typing: `_'b print` Will display on the screen those lines users have previously marked with the letter b, no matter where the line is in relation to where you are when you give the command. There is no need to tell the editor whether to search forward or backward; there can be only one line at a time marked with any one letter, and the editor will find that line regardless. The editor does some line marking too. Whenever you move from one line to another by a non-relative address, the editor marks the line you just left. (A non-relative address is one that is not a known number of lines from where you are.) So: `$/the cat/ 358 ?glossary? +7 _'b` are all non-relative addresses, and if you give any one of them, the editor will mark the line you are leaving for future reference. Then you can return to that line by just typing two successive single quotes" as a line-mode address. Users can use this address with any line-mode command.

Modifying any of these addresses is possible, and there are two ways to do this. The simpler way is to offset the address a certain number of lines forward or backward with plus (+) or minus (-) signs. The rule is that each plus sign following an address tells the editor to go one line forward in the file than the basic address, while each minus sign means a line backward. So these three addresses all refer to the same line: `35 37 — 30 +++++` The count offsets will work with any line-mode addresses, and are most often used with search patterns. In any event, there is a shorthand for these counts. A plus or minus sign immediately followed by a number (single or multiple digits) is equivalent to a string of plus or minus signs equal to that number, so that these two addresses are the same: `/^register long/ +++++ /^register long/ +4` The `—4` in the second example does not mean `—line number 4`, as it would, if it appeared by itself as an address. After a plus or minus sign, a number is a count forward or backward from where the primary address lands (or if there is no primary address before the count, from the line you are on when you run the command). Note also that this is one of the few places in line-mode commands where you may not insert a blank space. The number must start in the very next character position after the plus or minus sign. If you violate this rule, the editor will uncomplainingly operate on some line that definitely is not the line you expect. The second style of address modifier is used where you want to do a search that's complex. Let's say you want to go forward in the file to delete a line that starts with `—WARNING!!!`, but not the first such line the editor would encounter; you want the second instance. Either of these command lines will do it: `/^WARNING!// ; /^WARNING!/ delete /^WARNING!// ; // delete` A semicolon (;) between two search patterns tells the editor to find the location of the first pattern in the usual way, then start searching from that location for the second pattern. In this case, the first search pattern turned up the first instance of a line starting with `—WARNING!!!`, and the second search pattern led the editor on to the second instance. A very significant point is that this combination of two search patterns, either of which could be a line address in itself, does not tell the editor to delete two lines. The semicolon means that the first pattern is merely a way station, and that the single line found by the second search pattern is the only line to be deleted. In brief, what looks like addresses for two lines is actually only an address for one.

Users are not restricted to just two addresses. They can use up to ten of them, all separated by semicolons, to reach one specific line. As an example: `?^Chapter 3$? ; /^Bibliography$/ ; /^Operating Systems/ ; /Unix/` will bring me to the title line of Operating Systems first work with `—Unix` in the title, in the bibliography for Chapter 3. Users are not limited to search pattern addresses when putting together a semicolon-separated address string. If you want to reach the first line following line 462 that contains the word `—process`, type: `462 ; /\>process\</` will bring you there. And any of the addresses can take numerical offsets, so: `462 +137 ; /register int/ --` is also a legitimate address string. So: `0 ; /Unix/ +++ ; /Kant/ delete` Which is a reasonable way to be sure your search will find the very first `—Unix` in your file, will actually fail with an error message about an illegal address. Each address in a semicolon-separated string must be farther down in the file than the one that precedes it. (This means the actual location found, after applying any plus-sign or minus-sign offset.) Users cannot move backward within the series of way points. The first address can be a backward search and a subsequent address can search backward if you are certain that the line it finds will actually be more forward in the file. For example, you may know that a certain backward search will wrap around to the bottom end of the file before it finds a match. A common example would be: `1 ; ?Unix? ; /dos/ yank` Beginning a backward search from the first line in the file means that the search must start with the last line in the file due to wraparound, which guarantees that the search will yank the `—dos` line that follows the vary last `—Unix` line in your file. Also, users can use a plus-sign offset after a backward search when they are certain that the line finally found after the offset is applied, will be farther down in the file than the preceding way point had been. Thus, if users want to find the first mention of dos in Chapter 8 that is at least 120 lines after the last mention of it in Chapter 7, you can type: `/^Chapter 8$/ ; ?dos? +119 ; //`

If a command with this address fails and gives an error message about a bad address, users know that the last mention of `dos` in Chapter 7 is more than 120 lines before the end of the chapter, so the very first mention of its name in Chapter 8 is what users are looking for. In that case, the command is as follows `/^Chapter 8$/ ; /dos/`. The situation with forward searches inside a semicolon-separated address string is a mirror image of what is above. A forward search can take a minus-sign offset if users know that the offset is small enough that the line found will be further down than the last way point. But a forward search will fail, even with no offset or a plus-sign offset, if wraparound makes it find a line earlier in the file than the way point from which it began. Addressing a Section of Text Two addresses can also stand for a range of lines. When two addresses are separated by a comma rather than a semicolon, the meaning changes radically. Often users want a line-mode command to act on a series of successive lines. For example, users may want to move a stretch of text from one place to another. To do this, users have to give the address of the first line they want the command to act on, followed by the last line it should act on, and separate the two addresses with a comma. So, the command: `14 , 17 delete` will delete lines 14 to 17. Putting more than two addresses in a comma-separated address string is pointless. The line mode of the editor is discreet if users ignore this and string together three or more addresses with comma separation: it uses the first two addresses and discards the rest. Any line-mode addresses may be used with a comma. All of the following combinations can be used: `_d , /^Unix/ 257 , . ?^Chapter 9$? , $`. The first address combination would cause the command that follows it, to operate on the section starting with the line you have previously marked `.d.` and ending with the next forward line that begins with `—Unixll`, inclusive. The second combination covers line 257 through the line you are on now. The third goes backward to include the previous line containing only `—Chapter 9ll`, and forward to include the very last line in your file; plus all the lines in between, of course.

There are limitations to this technique. The primary one is that the address after the comma (after any offsets, of course) cannot be earlier in the file than the address before the comma. That is, the range of lines must run forward from the first address to the second address. So the command: `57 , 188 delete` is just fine, while the similar-looking command: `188 , 57 delete` will produce an error message. (But if the two addresses happen to evaluate to the same line, there is no problem. The command will silently operate on the one line you have specified). That is, if: `642 , /in Table 23/ delete` has failed, giving an error message that the lines are in the wrong order, then: `/in Table 23/ , 642 delete` will solve that problem. The limitation is that when users use search patterns on both sides of a comma, the second search starts from the current line just as the first search did; it does not start from the line that the first search found. The solution involves using one or more semicolons along with a comma. A semicolon-separated address string can be used anywhere in line mode that you would use as a single address. The technique is to use these address strings on one or both sides of a comma, to indicate a range of lines to be affected. An address string separated by semicolons is the address of just one line, so this one line can be the start or the end of a range of lines. For example, in: `/^INDEX$/ ; /^Xerxes/ , $ write tailfile ?^PREFACE$? ; /^My 7th point/ , ?^PREFACE$? ; /^In summary/ — delete the first command would write the latter part of the index to a new file, while the second is used to remove a section of a book's preface. If users want the search after the comma to begin from the point the first search found, one should use the first search pattern followed by a semicolon as the start of the after-the-comma search string, as in either of: ?Welcome? , ?Welcome? ; /line editing/ ?Welcome? , ?? ; /line editing/`

The first of the way points in each semicolon-separated string must be in the forward direction, but the start of the second semicolon-separated string may be prior to any of the addresses in the first such string, that is, the one-way meter resets itself at the comma point. And using semicolon separated strings on both sides of a comma only requires that the final landing point of the second semicolon-separated string should not be earlier in the file than the final landing point of the first. The relative locations of the way points do not matter to the comma. The combination: `125 ; 176 ; 221 , 32 ; 67 ; 240` looks invalid due to the backward jump from line 221 to line 32, but is actually a good address. The back jump comes right after the comma, where it is all right. But: `125 ; 176 ; 221 , 32 ; 67 ; 218` will produce an error message, because the final landing point of the first semicolon-separated string, line 221, falls later in the file than the final landing point of the second semicolon-separated string, line 218. Most line-mode commands that can take an address have a `—defaultll` address built in, which tells the editor where to run the command if users don't give an address with it. Each command has its own default address, which may be the current line, the current line plus the one following, the last line of the file, or the entire file. The comma separator has default addresses of its own. The comma separator is the same regardless of what command is being used, and they override any command's own default address. If users put a comma before a command and not the address, by default the address there is the current line. In the same way, if users leave out the address after the comma, the default there is also the current line. Users can even leave out the address in both places and use the current-line default in both: that means the implied address is `—`from the current line to the current linell, which makes the current line the only line the command will operate on. So the following command lines: `. write << ; example1 . , . write << ; example1 . , . write << ; example1 . , write << ; example1 , write << ; example1` will do exactly the same thing: append a copy of just the current line in the file you are editing to another file named `—example1ll`.

There is one special symbol that represents a comma-separated address combination. The percent sign (%) has the same meaning as 1,\$ as a line-mode address combination. Both refer to the entire file. SHELL ESCAPE COMMANDS Users will often want to escape from the editor to execute normal UNIX commands. Users may also want to change the working directory so that editing can be done with respect to a different working directory. These operations are described below: cd directory Causes the specified directory to become the current directory. sh Creates a new shell. To return to vi, enter a >Ctrl<D to terminate the shell. !command Sends the remainder of the line after the exclamation (!) to a shell to be executed. Current line is unchanged by this command. If there has been —[No write]ll of the buffer contents since the last change to the editing buffer, a diagnostic is displayed before the command is executed, as a warning. A single exclamation (!) is displayed when the command completes. To avoid this problem, use the default prompt value as shown below if you are in c-shell. /usr/lib/mkuser/csh/cshrc if you are in c-shell. Other commands The abbr, map, and set commands can also be defined with the EXINIT environment variable, which is read by the editor each time it starts up. These commands can be placed in a .exrc file in users home directory, which the editor reads if EXINIT is not defined. abbr This command maps the first argument to the following string. Abbreviations can be turned off with the unabbreviate command. EXAMPLE :abbr rainbow yellow green blue red maps —rainbowll to —yellow green blue redll. magic (default: magic) If nomagic is set, the number of regular expression meta characters is greatly reduced, with only caret (^) and dollar sign (\$) having special effects. In addition, the meta characters tilde (~) and ampersand (&) in replacement patterns are treated as normal characters. All the normal meta characters may be made magic when nomagic is set by preceding them with a backslash (\).

OPTIONS IN VI There are a number of options that can be set to affect the vi environment. These can be set with the ex set command while editing, with the EXINIT environment variable, or in the vi start-up file, .exrc. This file normally sets your preferred options so that they do not need to be set manually each time you invoke vi. The first thing that must be done before using vi, is to set the terminal type so that vi understands how to talk to the particular terminal you are using. There are only two kinds of options: switch options and string options. Switch Option A switch option is either on or off. A switch is turned off by prefixing the word no to the name of the switch within a set command. String Options String options are strings of characters that are assigned values with the syntax option=string. Multiple options may be specified on a line. vi options are listed below: autoindent, ai Default :No This Option eases the preparation of structured program text. For each line created vi looks at the preceding line to determine and insert an appropriate amount of indentation. Also processed in this mode are lines beginning with a caret (^) followed by a >Ctrl<D.. autoindent does not happen in global commands. autoprint ap Default : true This option causes the current line to be displayed after each ex copy, move, or substitute command. It is suppressed in globals, and only applies to the last command on a line. autowrite, aw (default: noaw). This option causes the contents of the buffer to be automatically written to the current file if you have modified it when you give a next, rewind, tag, or ! command, or a >Ctrl<^ (switch files) or >Ctrl<] (goto tag) command. directory, dir By default the directory is set to /tmp(default: dir=/tmp).This directory specifies the directory in which vi places the editing buffer file. If the directory does not have write permission, the editor exits abruptly when it fails to write to the buffer file.

errorbells, eb By default this option is set to no(default: noeb). This option precedes error messages by a bell. ignorecase, ic By default this option is set to no(default: noic). This option maps all uppercase characters in the text to lowercase in regular expression matching. In addition, all uppercase characters in regular expressions are mapped to lowercase except in character class specifications enclosed in brackets. list By default this option is set to no(default: nolist). This option displays all printed lines, showing tabs and end-of-lines. msg By default this option is set to no(default: nomsg). This option causes write permission to be turned off to the terminal while you are in visual mode, if nomsg is set. This prevents people writing to your screen with the UNIX write command and scrambling your screen as you edit. number, n By default this option is set to no(default: nonumber).This option causes all output lines to be printed with their line numbers. prompt By default this option is set to true(default: prompt).This option causes ex input to be prompted for with a colon (:). If noprompt is set, when ex command mode is entered with the Q command, no colon prompt is displayed on the status line. Remap By default this option is set to true (default: remap). If on, causes mapped characters to be repeatedly tried until they are unchanged. For example, if o is mapped to O and O is mapped to l, o will map to l if remap is set, and to O if noremap is set. Report By default this option is set to true(default: report=5). This option specifies a threshold for feedback from commands. Any command that modifies more than the specified number of lines provides feedback as to the scope of its changes.

shell By default the shell is in the bin directory(default: sh=/bin/sh). This option gives the pathname of the shell forked for the shell escape (!) command, and by the shell command. The default is taken from SHELL in the environment, if present. shiftwidth, sw By default this is set to eight(default:sw=8). This option gives the width of a software tab stop, used in reverse tabbing with >Ctrl<D when using auto-indent to append text, and by the shift commands. showmatch, sm By default this option is set to no(default: nosm). When a —|| or —|| is typed, moves the cursor to the matching —(— or —{— for one second if this matching character is on the screen. showmode By default this option is set to no(default:noshowmode). This option causes the message INPUT MODE to appear on the lower right corner of the screen when insert mode is activated. slowopen By default this option is set to no(default: noslowopen). This option postpones update of the display during inserts. tabstop, ts By default this option is set to eight(default: ts=8). This option causes the editor to expand tabs in the input file to be on n boundaries for the purposes of display. taglength, tl By default this option is set to zero(default: tl=0). This option causes the first n characters in a tag name to be significant, but all others to be ignored. A value of 0 (the default) means that all characters are significant. tags By default this is set to /usr/lib/tags(default: tags=tags /usr/lib/tags). This option specifies a path of files to be used as tag files for the tag command. A requested tag is searched for in the specified files, sequentially. By default, files named tags are searched for in the current directory and in /usr/lib.

term By default this value is set (default=value of shell TERM variable). This Specifies the terminal type of the output device. warn By default this is set to be true(default: warn). This will warns if there has been —[No write since last change]|| before a shell escape command (!). Window By default this option depends on the speed of cpu(default: window = speed dependent). This will specify the number of lines in a text window. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds. LIMITATIONS The /usr/lib/expreserve program can be used to restore vi buffer files that are lost as a result of a system crash. The program searches the /tmp directory for vi buffer files and places them in the directory/usr/ preserve. The owner can retrieve these files using the -r option. The /usr/lib/expreserve program must be placed in the system startup file,/etc/rc.d/3/recovery, before the command that cleans out the /tmp directory. A -r option that is not followed with an argument is replaced by -L, and +command is replaced by -c command. vi does not strip the high bit from 8-bit characters read in from text files, text insertion, and editing commands. vi does not look for —magic numbers|| of object files when reading in a text file. vi writes out text and displays text without stripping the high bit. vi uses the LC_CTYPE environment variable to determine if a character is printable, displaying the octal codes of non-printable 8-bit characters. vi uses LC_CTYPE and LANG to convert between upper and lowercase characters for the tilde command and for the ignorecase option. In vi when the percent sign (%) is used in a shell escape from vi via the exclamation mark (!), the —%|| is replaced with the name of the file being edited. Precautions Tampering with the entries in /usr/lib/terminfo/?/* (for example, changing or removing an entry) can affect programs such as vi that expect all entries to be present and correct. In particular, removing the —dumb|| terminal entry may cause unexpected problems. Software tabs using ^T work only immediately after the autoindent. Left and right shifts on intelligent terminals do not make use of insert and delete operations in the terminal.

Chapter 4-Unix Threading UNIX PROCESSES Before we discuss about processes, we need to understand exactly what a process is. A program or command, which is running, is called a process. A process exists for each Unix command or program that is currently running. It is also known as job or task. A Unix process is an entity that executes a given piece of code, has its own execution stack, set of memory pages, file descriptors table and a unique process ID. EXAMPLES After logging in to Unix, the shell starts running. The shell is a process. \$ vi While the vi program is running, there is a process associated with it. If ten people are simultaneously using vi, there will be ten processes associated with the vi program. \$ find ~ -name _*.txt' -print There is a process associated with the find command while it is running. When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program.

93%

MATCHING BLOCK 14/33

SA

Linux_system_administration_block_2.pdf (D149208459)

A process under UNIX consists of an address space and a set of data structures in the kernel to keep track of that process. The address space is a section of memory that contains the code to execute the process stack. The kernel must keep track of the following data for each process on the system: ? The address space map ? The current status of the process ? The execution priority of the process ? The resource usage of the process ? The current signal mask ? The owner of the process A process has certain attributes that directly affect execution. These include: ? PID - The PID stands for the process identification. This is a unique number that defines the process within the kernel ? PPID - This is the processes Parent PID, the creator of the process ? UID - The ID number of the user who owns this process ? EUID - The effective User ID of the process ? GID - The Group ID of the user that owns this process ? EGID - The effective Group User ID that owns this process ? Priority - The priority that this process runs at To view a process use the ps command. \$ ps -l F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME

CMD 20

R 211 2214 2213 2 47 24

fb11d218 60 - ttyp0 30:10 sh 20 O 211 2240 2214 1 48 24 fb11d370 168 - ttyp0 10:10 ps The ps command fields are discussed below. VARIOUS FIELDS IN PS COMMAND FIELDS MEANING

94%

MATCHING BLOCK 15/33**SA** Linux_system_administration_block_2.pdf (D149208459)

F This is the flag field. It uses hexadecimal values, which are added to show the value of the flag bits for the process. For a normal user process this will be 30, meaning it is loaded into memory. S The S field is the state of the process, the two most common values are S (for Sleeping) and R (

for

100%

MATCHING BLOCK 16/33**SA** Linux_system_administration_block_2.pdf (D149208459)

Running). An important value to look for is X, which means the process is waiting for memory to become available.

When you see this frequently on your system you are out of memory. UID The UID field shows the User ID (UID) of the process owner. For many processes this is 0 because they are run setuid.

98%

MATCHING BLOCK 17/33**SA** Linux_system_administration_block_2.pdf (D149208459)

PID The PID shows the Process ID of each process. This value should be unique. Generally PID are allocated lowest to highest, but wrap at some point. This value is necessary for you to send

a signal to a process such as the KILL signal. PPID This refers to the Parent Process ID. This identifies the parent process that started the process. Using this it allows you to trace the sequence of process creation that took place. PRI This stands for priority field. The lower the value the higher the value. This refers to the process NICE value. It will range from 0 to 39. The default is 20, as a process uses the CPU the system will raise the NICE value. The scheduler to compute the next process to get the CPU uses this value.

VARIOUS FIELDS IN PS COMMAND (CONTD.) FIELDS MEANING NI The NICE value of the process. ADDR The virtual address of the process entry in the process table.

97%

MATCHING BLOCK 18/33**SA** Linux_system_administration_block_2.pdf (D149208459)

SZ This refers to the SIZE field. This is the total number of pages in the process. Each page is 4096 bytes.

The sort command assists you while using the system. Use the sort command as the pipe output to sort by size or PID. For example to sort by SZ field use the command `ps -el | sort +9` (remember sort starts numbering fields with zero). WCHAN An address that uniquely identifies a process within the process table as sleeping until a particular resource becomes available; for example, until an I/O request has been completed, or in an SXBRK state until more pages of memory are available. TTY This is the terminal assigned to your process. On SGI based systems tty's with the letter —qll in them are psuedo, or network, tty's.

95%

MATCHING BLOCK 19/33**SA** Linux_system_administration_block_2.pdf (D149208459)

TIME The cumulative execution time of the process in minutes and seconds.

CMD The name of the command corresponding to the process. The `-f` option prints the full command name and its arguments. PROCESS STATES In order to know when to execute a program and when not to, it is convenient for the scheduler to label programs with a state variable. This is just an integer value that saves the scheduler time in deciding what to do with a process. Broadly speaking the state of a process may be one of the following. ? New ? Ready ? Running ? Waiting ? Terminated During time-sharing, the scheduler only needs to consider the processes, which are in the ready state. Changes of state are made by the system and follow the pattern in the diagram below.

Fig. 4.1 Process State The transitions between different states normally happen during interrupts. TRANSITION STATES OF PROCESS FROM STATE EVENT TO STATE New Accepted Ready Scheduled / Dispatch Running Running Need I/O Waiting Running Scheduler timeout Ready Running Completion / Error / Killed Terminated Waiting I/O completed or wakeup event 5 The Unix OS maintains information about each process in a process table. Entries in this table are often called process control blocks and must contain information about. ? Process state ? Memory state and ? Resource state for each process Process state: The process state must contain all the information needed so that the process can be loaded into memory and run. This includes. ? The value of each register ? The program counter ? The stack pointer Information about the state of the process needs to be stored. In addition it may contain extra fields such as: ? Process ID of itself, its parent, etc. ? Elapsed time ? Pending signals that have not yet been dealt with (e.g. they arrived while the process was asleep) Memory state: Pointers to the various memory areas used by the program need to be kept, so that they can be relocated when needed.

Resource state: The process will generally have files open, be in particular directory, have a certain user ID, etc. The information about these will need to be stored also. For example, in Unix each process has a file table. The first entry in this (file descriptor zero) is for the processes standard input, the second entry is for standard output and the third is for standard error. Additional entries are made when the process opens more files. Unix Process Startup When a system boots Unix, it creates a single process called init. This init acts as the root of the process tree. The init forks a set of copies of itself, one for each terminal line that is connected to it. Each one of these types the login message and then the terminal input. When the user name and password are typed, init checks whether it is valid, and if so changes to the user's home directory, resets the user ID of the process to the user, and executes a shell. At this point the login shell for that user has replaced the init for that terminal line. Logging out terminates the login shell. In the meantime, the init at the top of the process tree should wait for its children process to complete. The login shell is in fact a direct child of this toplevel init because it comes from executing a child. So when the shell terminates, the toplevel init wakes up. The toplevel init then forks a new init for that terminal line and starts a wait again, for another child to terminate. Context Switching: Switching from one running process to another is called context switching. While doing so the values of all the registers must be saved in the present state, the status of all open files must be recorded and the present position in the program must be recorded. Then the contents of the MMU (Memory management unit) must be stored for the process. These values must be validated for the next process, so that the state of the system is exactly as it was when the scheduler last interrupted the process. This is called context switch. Context switching is a system overhead. It costs real time and CPU cycles so remember not to context switch too often, or a lot of time will be wasted. The state of each process is saved to a data structure in the kernel called a process control block (PCB). The information stored in the PCB is listed below. ? Process ID (name, number) ? Process state ? Priority, owner, etc. ? Program counter ? CPU registers ? CPU scheduling information ? Memory-management information ? Accounting information ? I/O status information

InterProcess Communication: Processes do not run in isolation from each other. Generally they need to communicate with each other. Example #1: Any two processes in a pipeline communicate with each other. One sends a stream of bytes to the other. Example #2: A single process called lpd (the lineprinter daemon) controls access to the lineprinter. Each time a user runs lpr it has to communicate with lpd and send it the file to print. Example #3: Your home directories are stored on the machine workgroup. Each time you access a file the O/S has to make a connection to workgroup, request the file from a suitable process on workgroup and accept responses from it. There are many models of IPC, with different advantages and problems. They are enumerated below. Shared memory: If two processes share the same piece of memory then they can use this to communicate. For example, one may write information in this shared area and the other may read it. This is a rapid method of information transfer because RAM can be used. However synchronization is a major problem. (If the first process keeps writing data, how can it ensure that the second one reads it?). Pipeline: A pipe acts like a channel between two processes. When one process writes into the pipe the other process reads from it. A pipe can usually buffer information so that the writer can place a lot of information in the pipe before the child has to read it. When the pipe becomes full the writer has to suspend the process. Pipes can be unnamed. This is the norm in Unix where a process creates a pipe and then forks, so that the two processes share the pipe between them. If the processes do not come from a common ancestor then they can only share a pipe if they can both name it. Named pipes usually appear as though they are files in the file system. Streams: Pipes carry unstructured data - bytes put in one end are received at the other end. Streams are designed to carry record information - you put records in at one end and get the same records out the other. Each record must contain a field denoting how large it is. Sockets: Sockets are more like ports that you can send data to. A process listens at a port and accepts data sent to it. Signals: Signals are a fairly crude method of IPC. A process may send a signal to another such as wake up or die. The other process can respond to these signals in various ways.

PROCESS CREATION As you may already know, it is possible for a user to run a process in the system, suspend it (Ctrl- Z), and move it to the background (using the bg command). The creation of a process requires the following steps. The order in which they are carried out is not necessarily the same in all cases. ? Name: The name of the program, which is to run as the new process, must be known ? Process ID and Process Control Block: The system creates a new process control block, or locates an unused block in an array. This block is used to follow the execution of the program through its course, keeping track of its resources and priority. Each process control block is labeled by its PID or process identifier. ? Locate the program to be executed on disk and allocate memory for the code segment in RAM ? Load the program into the code segment and initialize the registers of the PCB with the start address of the program and appropriate starting values for resources ? Priority: A priority must be computed for the process, by using default for the type of process and any value which the user specified as a nice value. ? Schedule the process for execution The process creation is handled by the system calls. System calls are an interface between the operating system and its application programs. The actual mechanics of issuing a system call are highly machine dependent and are often expressed in assembly language. A procedure library is often provided to make it possible to make system calls from C programs. The system calls vary from one operating system to another even though the underlying concepts tend to be similar. The three important system calls are as follows: ? The fork() system call ? The exec() system call ? The wait() system call The fork() System Call The fork() system call is the basic way to create a new process. It is also a very unique system call, since it returns twice (!) to the caller. Sounds confusing doesn't it, this confusion stems from the attempt to define as few systems calls as possible. The fork() system call causes the current process to be split into two processes: ? A parent process ? A child process

All of the memory pages used by the original process get duplicated during the fork() call, so both parent and child process see the exact same image. The only distinction is when the call returns. When it returns in the parent process, its return value is the process ID (PID) of the child process. When it returns inside the child process, its return value is `_0`. If for some reason this call failed (not enough memory, too many processes, etc.), no new process is created, and the return value of the call is `-1`. In case the process was created successfully, both child process and parent process continue from the same place in the code where the fork() call was used. The exec() System Calls The exec() system call in all its forms transforms the calling process into a new process. The new process may be constructed from an ordinary executable file. Executable files consist of a header, a text segment and a data segment. The data segment contains an initialized portion and an uninitialized portion. There can be no return from a successful exec () because the calling process is overlaid by the new process. The exec() system call is different from fork in that no new sub process is created and run concurrently. The main difference between them is the way parameters can be passed. The wait() System Call The simple way of a process to acknowledge the death of a child process is by using the wait() system call. When wait() is called, the process is suspended until one of its child processes exit, and then the call returns with the exit status of the child process. If it has a zombie child process, the call returns immediately, with the exit status of that process. PROCESS HANDLING Before getting started with process handling, you should understand the concept of processes. Processes can also be described as either foreground or background processes. While the foreground process is active, no other process can access the I/O devices the foreground process controls. Background processes can run simultaneously with what is happening in the foreground. Most jobs will run in the foreground simply by entering the name of the program or command. A job may be started in the background by appending an ampersand (&) at the end of the command line. A job which writes to the screen or requires user input should generally not be run in the background. Normally, when you type a command, you do not get another prompt until that command has finished. However, if you end your command line with an ampersand (&), the command will start running in the background, and you will get another prompt right away. You can continue typing more commands while the background process is running.

For example, to run a command ps in the background, you have to type the ps command followed by ampersand sign (&) as follows: `$ ls& [1] 3119 $` The second line tells you that your background job number is 1 and the process ID is 3119. You may need these numbers to control the process later. To monitor the progress of the background process, type the ps command. Sometime you may need to kill a process you have started. This may be because the process takes too much process time, causing the system to slow down or perhaps because it is caught in a loop and will therefore never complete. To kill the current process, type: `>Ctrl< >Del< >Del< >Break< >Ctrl< >d<` Only the root user can kill processes belonging to another user or to the system. If the process you want to stop is a child of your shell, you can use command and the process job number to stop it, as in `kill %jobnumber`. If neither the interrupt generation keyboard sequences nor the kill command stop the process, try the following: ? Log in to another terminal ? Use `ps -u your_login` to find out the process ID of that process you want to stop ? Type `kill pid` to kill the process. For e.g., to force termination of a job whose process ID is 111, enter the command: `$ kill -9 111` Process Scheduling: You must know by now that Unix is a multi-tasking, multi-user Operating System. This means that many users can use the same machine at the same time and the same machine can run many processes at any given time. Some programs require a lot of computer resources and have little user interaction. These are known as batch jobs. Other programs that have people corresponding with them are called interactive processes. Interactive processes require little computer resources, but what they do need should be supplied instantly so that people are not made to wait. If you do run a CPU intensive program, please proceed with the nice command. It will cause it to run with less priority than interactive processes. Moreover, your job will only take a small amount of additional time to run and everybody's interactive programs will respond quickly. nice command - run command so that it is nice to everybody nice -10 command - run your command with the lowest priority The at command will allow you to run a job at a specific time. The output from the job is mailed to the invoking user. `$ date Thu Apr 5 15:47:33 IST 2001 $ at 15:48 echo Its time! *** CONTROL-D TO TERMINATE INPUT *** job 986465880.a-3209:0 at Thu Apr 5 15:48:00 2001` The process scheduler is responsible for changing the state of processes. A scheduler must compromise between certain desirable features. A scheduler - ? keeps the CPU busy 100% of the time ? makes sure each process gets a fair share of the CPU ? minimizes response delays Stopping processes There are two ways in which processes can stop: ? They voluntarily stop ? They can be forced to stop If there is a mechanism to stop processes, then the scheduling is preemptive, otherwise it is non-preemptive or co-operative. OS2 and Windows use non- preemptive schedulers. Windows NT and Unix use preemptive schedulers. To keep the CPU as busy as much as possible, a process should be replaced whenever it is blocked. Timer interrupts can be set to stop a process to allow other processes to run. When a process stops or is stopped, the registers have to be copied into the process control block. Any values in the PCB that have changed will need to be set. Round robin scheduling: This is one of the simplest scheduling algorithms. The O/S maintains a queue of runnable processes. When a running process is stopped, or a blocked process becomes runnable, it is placed on the end of the queue.

When the CPU becomes free, the process at the head of the queue is loaded and made the running process. Each process is allowed to run for a fixed maximum time before it is stopped. This fixed time must be carefully tuned. If it is too long, then response time for processes currently not running is too slow. If the time is too short, then the system will spend proportionally more time swapping processes and will just appear to be running slowly. Priority scheduling: In round robin, all processes have the same priority. In priority scheduling they are different. The O/S will maintain lists of processes, one for each priority. A runnable process from the highest priority list will be chosen. Priorities may be static or dynamic. In Unix, the longer a process runs, the lower its priority becomes. Within a queue, a process may be chosen say by round robin. Shortest job first: This is often used on batch systems. In these it is quite common to know in advance about how long a job will take. Then the list of jobs can be ordered as and when required. The one needing the shortest time can be scheduled first which in turn gives the best average time. This cannot easily be used in interactive systems because you do not know how long a command will take. Two level scheduling: The above schemes assume either that all processes are in memory, or that only one process (the one running currently) is in memory. Then the cost of context switching is nearly equal. The cost of context switching to a process that is in memory is that the registers have to be loaded, and a jump has to be made to the program counter. The extra cost of context switching to a process that is not in memory is that it first has to be loaded from disk. This will be significantly slower. A two-level scheduler will maintain two lists, one for processes in main memory, the other for those on disk. One scheduler will work on those in memory, switching between them. Periodically another scheduler will move processes between the lists so that those on disk have a chance to run. UNIX MAIL SYSTEM UNIX system has an in-built mail system, which allows you (as a sender) to compose a letter and send it across to a number of users. As a receiver, you can receive the messages and read it as and when you want to. In mail systems you are able to store messages and customize them. In UNIX, mail program has two modes. Each mode has its own set of commands. ? Send mode: You can compose and send a mail in this mode. It has two character commands that start with the tilde (~) character.

? Command mode: You can read and manage a mail in this mode. It has single character commands. Sending mail In fact to send a mail in UNIX is very easy. To do so, you have to follow these simple steps. ? Enter the command: \$ mail username@address ? You will then be prompted to enter the subject of your mail message. Enter the subject title and press the Return key. If you make a mistake use the Backspace or Delete key for corrections. For example: Subject: Meeting tomorrow at 10:00 ? Enter the text of your message. Use the Backspace or Delete key for corrections. Press the Return key whenever you want to start a new line. Note that you cannot move to preceding lines to make corrections. You will need to invoke an editor, such as vi to accomplish this. ? When you have finished typing the message, press the Return key to start a new line and type CTRL-D. You will then be prompted for addresses to Cc. You may either enter additional e-mail addresses or press the Return key if there are no addresses to copy. Your message will be sent and the shell prompt will be displayed. ? Send Mode Commands: These commands allow you to perform various functions while you are in the middle of composing your mail message. These commands consist of two characters, the first of which is the tilde (~) character. The tilde (~) character must be issued as the first characters on a line in order to be interpreted as commands. It should not be issued in the message text. A list of the send mode commands is given below. Note that this set is not universal. Different UNIX operating systems have different sets. The set below is for mail running under the UNIX V operating system. Control Commands: CTRL-D Sends message. ~q Quits editor without saving or sending message. ~p Displays the contents of the message buffer. ~? Displays help. ~:set Shows which mail options, are in effect. Add to Heading ~h Adds to lists for To: Subject: Cc: and Bcc:. ~t addrlist Adds user addresses in addrlist to the To: list. ~s subject Sets the Subject: line to the string specified by Subject. ~c addrlist Adds user addresses in addrlist to Cc: list. ~b addrlist Adds user addresses in addrlist to Bcc: list. Add to Message ~d Appends contents of dead.letter to message. ~r filename Appends contents of filename to the message. ~f numlist Appends contents of message numbers in numlist. ~m numlist Append/indents contents of message numbers in numlist. Change Message ~e Edits the message using an external editor (default is e). ~v Edits the message using an external editor (default is vi). ~w filename Writes the message to filename. ~! command Starts a shell, runs command, and returns to the editor. ~| command Pipes the message to standard input of command; Replaces the message with the standard output from that command. Invoking an editor, with either the ~e or ~v command makes creating your message much easier. If you use ~v, then the vi editor will be used. If you use ~e, you can use a different editor, such as pico: ~:set EDITOR=/local/bin/pico - first specify the editor of choice ~e - then invoke it In addition to the ~r command, you can send a file in a mail message as shown below. In this example, the file WRKGP will be sent to the address support@u18.edu.in. No editing of the mail message is possible – it will be sent directly. \$ mail support@u18.edu.in > WRKGP

Reading Mail To read your mail, simply enter the command: `$ mail` If you have any mail in your system mailbox, an indexed list of the messages will be displayed on your screen. For example: `< 1 support@u18.edu.in Mon March 26 14:49 94/3378`
`—Suggestionsll N 2 contactus@u18.edu.in Wed March 28 12:17 94/3308 —netfindll &` In the above example, `? <` indicates the current message `? N` indicates a new message `? Messages` are numbered and sorted by date/time `? Message sender` is displayed `? Message size in lines/characters` appears `? Subject` is displayed `? &` is the mail prompt, which informs you that you are in the command mode and that mail is ready to receive a command `? To read the current message`, just press the Return key `? To read a specific message other than the current message`, type the message number `? This message` will be displayed and becomes the current message `? To redisplay the message list`, use the `h` command `? To display the next message`, use the `n` command

Command Mode Commands: These commands enable you to perform tasks related to reading and managing your mail. They are always entered at the `—&ll` mail prompt. A summary of the command mode commands for UNIX V mail is tabulated below.

Control Commands `q` Quits - apply mailbox commands entered this session. `x` Quits - restores mailbox to original state. `!` `cmd` Starts a shell, run `cmd`, return to mailbox. `cd dir` Changes directory to `dir` or `$HOME`. `a` Displays list of aliases with addresses. `? Displays help`. `set` Shows, which mail options are in effect. `folder filename` Makes `filename` the current mail folder.

Display Commands Return key Displays current message. `- t` Displays current message. `t msg_list` Displays messages in `msg_list`. `n` Displays next message. `- f msg_list` Displays headings of messages in `msg_list`. `h` Displays headings or all messages.

Message Handling: `e num` Edits message `num` (default editor is `e`). `v num` Edits message `num` using `vi` editor. `d msg_list` Deletes messages in `msg_list` or current message. `u msg_list` Recalls deleted messages. `s msg_list file` Appends messages (with headings) to file. `w msg_list file` Appends messages (text only) to file. `pre msg_list` Keeps messages in system mailbox.

Creating New Mail `m addrlist` Creates/sends new message to addresses in `addrlist`. `r msg_list` Sends reply to senders and recipients of messages. `R msg_list` Sends reply only to senders of messages.

Saving Mail and Using Folders One of the most useful features of mail is its ability to save mail messages and to organize it in the way you wish. By default, mail will save all messages that you have read a file in your home directory called `mbox`. You can choose to save messages in other files by using the `s` command. Whenever you save a mail message, it is appended to the file you specify. Files can also be called folders, since they allow multiple mail messages to be organized into a common location. Some examples of saving mail into folders are given in the next page:

`s jwilliams` Save current message to file `jwilliams` in current directory. `s 1-4 projects` Save messages 1 thru 4 to file `projects` in current directory. `s 2 4 5 Mail/old/williams` Save messages 2,4,5 to file `Mail/old/williams`. By default, when you start mail, it uses your system mailbox as its folder. Your system mailbox is usually located someplace like `/usr/spool/mail` or `/var/spool/mail`. It contains both new mail and unread mail. You can view other folders by starting mail with the `-f` option. For example, this command will start mail with the folder `projects`: `$ mail -f projects` While you are in the mail program, you can switch between different folders by using the `folder` command at the `—&ll` prompt. For example, the following command switches the current folder to `—Mail/jwilliamsll`: `$ folder jwilliams` If you prefer to save all of your folders in a certain subdirectory, and do not wish to type the full name of that directory every time you save a message or switch folders, mail provides a feature which makes it easier. Set an environment variable called `folder` to point to the directory where all of your mail folders should be kept. For example, the following line could be added to your `cshrc` file: `setenv folder Mail/folders` When saving or accessing a folder, precede the folder name with a plus (+) sign. This will prompt mail to use the location specified by your folder environment variable. For example: `$ s +projects $ mail -f +projects $ folder +projects`

Customizing Your Mail Environment Mail has many options. The default option settings for your system are specified in a file called: `/usr/lib/Mail.rc` Mail allows you to customize your mail environment and preferences. Your customizations are specified in a file in your home directory, called `.mailrc`. These specifications will override or augment the system defaults. Some useful mail options are listed below.

VARIOUS OPTIONS WITH THE MAIL COMMAND

VARIABLE	ACTION	alias	name	address
<code>Creates an alias</code>	<code>Creates an alias</code>	<code>alias</code>	<code>name</code>	<code>address</code>
<code>Prompts for the subject of each message sent</code>	<code>Prompts for the subject of each message sent</code>	<code>ask</code>		
<code>Prompts for carbon copy mail addresses</code>	<code>Prompts for carbon copy mail addresses</code>	<code>askcc</code>		
<code>Defines the number of lines of a mail message the Mail program displays before pausing for input. Works with PAGER.</code>	<code>Defines the number of lines of a mail message the Mail program displays before pausing for input. Works with PAGER.</code>	<code>crt=lines</code>		
<code>Gives the full path name of the editor to be started with the e mailbox subcommand or the ~e mail editor subcommand.</code>	<code>Gives the full path name of the editor to be started with the e mailbox subcommand or the ~e mail editor subcommand.</code>	<code>EDITOR=editor</code>		
<code>Directory to save mail files in.</code>	<code>Directory to save mail files in.</code>	<code>folder=directory</code>		
<code>Keeps messages that have been read in the system mailbox, not the mbox file.</code>	<code>Keeps messages that have been read in the system mailbox, not the mbox file.</code>	<code>hold</code>		
<code>Specifies which header fields to not display (ignore).</code>	<code>Specifies which header fields to not display (ignore).</code>	<code>ignore fieldlist</code>		
<code>Do not delete messages that have been saved. Keep them in the system mailboxes. Use with the hold option.</code>	<code>Do not delete messages that have been saved. Keep them in the system mailboxes. Use with the hold option.</code>	<code>keepsave</code>		
<code>Prevents retention of interrupted letters in the \$HOME/dead.letter file.</code>	<code>Prevents retention of interrupted letters in the \$HOME/dead.letter file.</code>	<code>nosave</code>		
<code>The pager is used to paginate output when a message is longer than the screen. Works with crt setting.</code>	<code>The pager is used to paginate output when a message is longer than the screen. Works with crt setting.</code>	<code>PAGER</code>		
<code>Defines a file in which to record outgoing mail.</code>	<code>Defines a file in which to record outgoing mail.</code>	<code>record=fileName</code>		
<code>Defines the number of lines of message headers displayed before pausing for input.</code>	<code>Defines the number of lines of message headers displayed before pausing for input.</code>	<code>screen=lines</code>		
<code>Defines the number of lines at the top of each message to print with the ~t command.</code>	<code>Defines the number of lines at the top of each message to print with the ~t command.</code>	<code>topline=number</code>		
<code>The name of the editor called by the ~v command. Editor is given as a full pathname. A simple .mailrc file appears below. Note that more than one option can be placed on a line.</code>	<code>The name of the editor called by the ~v command. Editor is given as a full pathname. A simple .mailrc file appears below. Note that more than one option can be placed on a line.</code>	<code>VISUAL=editor</code>		

Simple .mailrc file # `set ask askcc EDITOR=/source/local/bin/pico folder=Mail set PAGER=more crt=20`

Chapter 5- Shell Programming INTRODUCTION TO SHELL A shell is a command line interpreter. It takes commands and executes them. A shell is used to create shell scripts, i.e. programs that are interpreted /executed by the shell. Each input line is scanned and split into tokens; parameters are substituted subject to quoting, filenames are generated, input and output are optionally redirected and then the commands are executed.

1. Shell scripts are simple text files created with an editor
2. Shell scripts are marked as executable
3. They should be located in user search path and ~/bin should be in your search path
4. Users likely need to rehash if user is a Csh(tcsh) user
5. Arguments are passed from the command line and referenced. For example, as \$1.

BOURNE SHELL The Bourne shell is the traditional Unix shell originally written by Stephen Bourne. All Bourne Shell scripts should begin with the sequence `#!/bin/sh`. When you issue a command in the Bourne shell, it first evaluates the command and makes all indicated substitutions. It then runs the command provided that the command name is a Bourne shell special built-in command or the command name matches the name of a defined function. If this is the case, the shell sets the positional parameters to the parameters of the function. If the command name matches neither a built-in command nor the name of a defined function and the command names an executable file that is a compiled (binary) program, the shell (as parent) spawns a new (child) process that immediately runs the program. If the file is marked executable but is not a compiled program, the shell assumes that it is a shell procedure. In this case, the shell spawns another instance of itself (a subshell), to read the file and execute the commands included in it. The shell also runs a parenthesized command in a subshell. To the end user, a compiled program is run in exactly the same way as a shell procedure. The shell normally searches for commands in file system directories, in the order `/usr/bin, /etc, /usr/sbin, usr/ucb, $HOME/bin, /usr/bin/X11, /sbin`. The shell searches each directory, in turn, continuing with the next directory if it fails to find the command. The order in which the shell searches directories is determined by the `PATH` variable. You can change the particular sequence of directories searched by resetting the `PATH` variable. If you give a specific path name when you run a command (for example, `/usr/bin/sort`), the shell does not search any directories other than the one you specify. If the command name contains a / (slash), the shell does not use the search path. You can give a full path name that begins with the root directory (such as `/usr/bin/sort`). You can also specify a path name relative to the current directory. If you specify, for example `bin/myfile` the shell looks in the current directory named `bin` for the file `myfile`. The shell recollects the location in the search path of each executed command (to avoid unnecessary `exec` commands later). If it finds the command in a relative directory (one whose name does not begin with /), the shell must predetermine the command's location whenever the current directory changes. The shell forgets all locations each time you change the `PATH` variable or run the `hash -r` command.

Search Path All shell scripts should include a search path specification: `PATH=/usr/ucb:/usr/bin:/bin; export PATH`. A `PATH` specification is recommended any time, a script fails because of a different or incomplete search path. The Bourne Shell does not export environment variables to children unless explicitly instructed to do so by using the `export` command.

Looping Constructs

- `until test-commands; do consequent-commands; done` Execute consequent-commands as long as the final command in test-commands has an exit status which is not zero.
- `while test-commands; do consequent-commands; done` Execute consequent-commands as long as the final command in test-commands has an exit status of zero.
- `for name [in words ...]; do commands; done` Execute commands for each member in words, with name bound to the current member. If in words is not present, in `—$@` is assumed. The restricted shell does not run commands containing a / (slash).

Conditional Constructs

- `if test-commands; then consequent-commands; [elif more-test-commands; then more-consequents;] [else alternate-consequents;] fi` Execute consequent-commands only if the final command in test-commands has an exit status of zero. Otherwise, each `elif` list is executed in turn, and if its exit status is zero, the corresponding `more-consequents` is executed and the command completes. If `else alternate-consequents` is present, and the final command in the final `if` or `elif` clause has a non-zero exit status, then execute `alternate-consequents`.
- `case word in [pattern [| pattern]...) commands ;;]... esac` Selectively execute commands based upon word matching patterns. The `_|'` is used to separate multiple patterns.

echo -n —Enter the name of an animal: — read ANIMAL echo -n —The \$ANIMAL has — case \$ANIMAL in horse | dog | cat) echo -n —fourll;; man | kangaroo) echo -n —twoll;; *) echo -n —an unknown number ofll;; esac echo —legs.

BOURNE SHELL BUILT-INS

The following are the shell built in commands inherited from the Bourne shell. These commands are implemented as specified by the Posix 1003.2 standard.

SHELL BUILT-IN COMMANDS

COMMAND	DESCRIPTION
<code>.</code>	Do nothing beyond expanding any arguments and performing redirections.
<code>.</code>	Read and execute commands from the filename argument in the current shell context.
<code>break</code>	Exit from a for, while, or until loop.
<code>cd</code>	Change the current working directory.
<code>continue</code>	Resume the next iteration of an enclosing for, while, or until loop.
<code>echo</code>	Print the arguments, separated by spaces, to the standard output.
<code>eval</code>	The arguments are concatenated together into a single command, which is then read and executed.
<code>exec</code>	If a command argument is supplied, it replaces the shell. If no command is specified, redirections may be used to affect the current shell environment.
<code>exit</code>	Exit the shell.
<code>export</code>	Mark the arguments as variables to be passed to child processes in the environment.
<code>getopts</code>	Parse options to shell scripts or functions.
<code>hash</code>	Remember the full pathnames of commands specified as arguments, so they need not be searched on subsequent invocations.

55%

MATCHING BLOCK 20/33

SA

DECAP448_LINUX_AND_SHELL_SCRIPTING.pdf
(D142327428)

kill Send a signal to a process. pwd Print the current working directory. read Read a line from

the shell input and use it to set the values of specified variables. readonly Mark variables as unchangeable. return Cause a shell function to exit with a specified value. shift Shift positional parameters to the left. test Evaluate a conditional expression. times Print out the user and system times used by the shell and its children. trap Specify commands to be executed when the shell receives signals. umask Set the shell process's file creation mask. unset Cause shell variables to disappear. wait Wait until child processes exit and report their exit status. Bourne Shell Compound Commands A compound command is one of the following: ? Pipeline (one or more simple commands separated by the | (pipe) symbol) ? List of simple commands ? Command beginning with a reserved word ? Command beginning with the control operator ((left parenthesis) Unless otherwise stated, the value returned by a compound command is that of the last simple command executed.

Pipeline A pipeline is a sequence of one or more commands separated by a vertical bar (|). A pipe to the standard input of the next command connects the standard output of each command but the last. Each command is run as a separate process; the shell waits for the last command to terminate. A list is a sequence of one or more pipelines separated by ;, &, &&, or || and can be terminated by ; or &. Of these four symbols, ; and & have equal precedence, which is lower than that of && and ||. The symbols && and || also have equal precedence. A semicolon (;) causes sequential execution of the preceding pipeline; an ampersand (&) causes asynchronous execution of the preceding pipeline (that is, the shell does not wait for that pipeline to finish). The symbol && causes the list following it to be executed if the preceding pipeline succeeds (returns a 0 exit status). The symbol || causes the list following it to be executed only if the preceding pipeline fails (returns a non-zero exit status). An arbitrary number of newlines may appear in a list, instead of semicolons, to delimit commands. Because commands in pipelines are run as separate processes, variables set in a pipeline have no effect on the parent shell. The Bourne shell processes these commands as follows: : (colon) exec shift . (dot) exit times break export trap continue readonly wait eval return ? Keyword parameter assignment lists preceding the command remain in effect when the command completes ? I/O redirections are processed after parameter assignments ? Errors in a shell script cause the script to stop processing Allocating the Shell Resources ulimit [-HS] [-c | -d | -f | -m | -s | -t] [limit] Displays or adjusts allocate shell resources. There are two modes for displaying the shell resource settings, which can either be displayed individually or as a group. The default mode is to display resources set to the soft setting, or the lower bound, as a group. The setting of shell resources depends on the effective user ID of the current shell. The hard level of a resource can be set only if the effective user ID of the current shell is root. You will get an error if you are not logged onto the root and you are attempting to set the hard level of a resource. By default, the root user sets both the hard and soft limits of a particular resource. The root user should therefore be careful

in using the -S, -H, or default flag usage of limit settings. Unless you are a root user, you can only set the soft limit of a resource. Once a non-root user has decreased a limit, it cannot be increased, back to the original system limit. To set a resource limit, select the appropriate flag and the limit value of the new resource, which should be an integer. You can only set one resource limit at a time. If more than one resource flag is specified, you receive undefined results. By default, ulimit with only a new value on the command line sets the file size of the shell. Reserved Words The following reserved words are recognized only when they appear without quotes as the first word of a command: for do done case esac then fi elif else while until { } () Bourne Shell Variables IFS A list of characters that separate fields; used when the shell splits words as part of the expansion. PATH

96%

MATCHING BLOCK 21/33

SA Linux_system_administration_block_2.pdf (D149208459)

A colon-separated list of directories in which the shell looks for commands. HOME The

current user's home directory. CDPATH A colon-separated list of directories used as a search path for the cd command. MAILPATH A colon-separated list of files which the shell periodically checks for new mail. PS1 The primary prompt string. PS2 The secondary prompt string. OPTIND The index of the last option processed by the builtin getopts. OPTARG The value of the last option argument processed by the builtin getopts. Exit Status All Unix utilities should return an exit status. The header file /usr/include/sys/exits.h contains the various exit codes. Exit codes are important for those who use the code. Many constructs test the exit status of a command.

is the year out of range for me? if [\$year -lt 1901 -o \$year -gt 2099]; then echo 1<&2 Year \\\\$year\\ out of range exit 127 fi # All done, exit ok exit 0 A non-zero exit status indicates an error condition of some sort while a zero exit status indicates things worked as expected. Stdin, Stdout, Stderr Standard input, output, and error are file descriptors 0, 1, and 2. Each has a particular role and should be used accordingly. Error messages should appear on stderr and not on stdout. Output should appear on stdout. Variables Variables are sequence of letters, digits or underscore beginning with a letter or underscore. To get the contents of a variable the name is preceded with a \$. Numeric variables (e.g. like \$1, etc.) are positional variables for argument communication. Variable Assignment Assign a value to a variable by variable=value. For example: PATH=/usr/ucb:/usr/bin:/bin; export PATH or TODAY=_(set _date_; echo \$1)_ Conditional Reference \${variable-word} If the variable has been set, use its value else use word. \${variable:-word} If the variable has been set and is not null, use its value, else use word. These are useful constructions for honoring the user environment. \${variable:?word} If variable is set use its value, else print out word and exit. Special Variables \$\$ This will give the current process id and is very useful for constructing temporary files.

tmp=/tmp/cal0\$\$ trap —rm -f \$tmp /tmp/cal1\$\$ /tmp/cal2\$\$ll trap exit 1 2 13 15 /usr/lib/calprog <\$tmp \$? This gives the exit status of the last command. \$command # Run target file if no errors and ... if [\$? -eq 0] then fi Quotes Single Quotes Within single quotes all characters are quoted including the backslash. The result is one word. grep :\${gid}: /etc/group | awk -F: '{print \$1}' Double Quotes Within double quotes you have variable substitution (i.e. the dollar sign is interpreted) but no file name is generated (i.e. * and ? are quoted). The result is one word. if [! —\${parent}ll]; then parent=\${people}/\${group}/\${user} fi

FUNCTIONS IN BOURNE SHELL Functions are a powerful feature of the shell. Shell functions are a way to group commands for later execution using a single name for the group. They are executed just like a regular command. Shell functions are executed in the current shell context; no new process is created to interpret them. Functions are declared using this syntax: [function] name () {command-list;} This defines a function named name. The body of the function is the command-list which appears between {and}. This list is executed whenever name is specified as the name of a command.

79%

MATCHING BLOCK 22/33

W

The exit status of a function is the exit status of the last command executed in the

body.

When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter # that gives the number of positional parameters is updated to reflect the change. Positional parameter 0 is unchanged. If the builtin command return is executed in a function, the function completes and execution resumes with the next command after the function call. When a function is completed, the values of the positional parameters and the special parameter # are restored to the values they had prior to function execution. # Purge a directory _purge() { # there had better be a directory if [! -d \$1]; then echo \$1: No such directory 1<&2 return fi } Test The most powerful command is test. if test expression; then Useful expressions are: test { -w, -r, -x, -s, ... } filename is file writeable, readable, executable, empty, etc. test n1 { -eq, -ne, -gt, ... } n2 are numbers equal, not equal, greater than, etc. test s1 { =, != } s2 Are strings the same or different? test cond1 { -o, -a } cond2 Binary or; binary and; use! for unary negation. EXAMPLE if [\$year -lt 1901 -o \$year -gt 2099]; then echo 1<&2 Year \\\$year\\ll out of range exit 127 fi

Debugging The shell has a number of flags that make debugging easier: sh -n command Read the shell script but do not execute the commands. It checks the syntax. sh -x command Display commands and arguments as they are executed. **RESTRICTED SHELL (RSH)** rsh [flags] [name [arg1 ...]] rsh is a restricted version of the standard command interpreter. It is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of rsh are identical to those of sh, except that changing directory with cd, setting the value of \$PATH, using command names containing slashes, and redirecting output using < and << are all not allowed. When invoked with the name .rsh, rsh reads the user's .profile (from \$HOME/.profile). It acts as the standard sh while doing this, except that an interrupt causes an immediate exit, instead of causing a return to command level. The restrictions above are enforced after .profile is interpreted. When a command to be executed is found to be a shell procedure, rsh invokes sh to execute it. Thus, it is possible to provide shell procedures to the end user who has access to the full power of the standard shell, while restricting him/her to a limited menu of commands. This scheme assumes that the end user does not have write and execute permissions in the same directory. The net effect of these rules is that the writer of the .profile has complete control over user actions, by performing guaranteed setup actions, leaving the user in an appropriate directory (not the login directory). rsh is actually just a link to sh and any flags arguments are the same for sh. The system administrator often sets up a directory of commands that can be safely invoked by rsh. **C-SHELL** Bill Joy at UC Berkeley created the C-Shell (csh). It is generally considered to have better features for interactive use than the original Bourne shell. Some of the csh features present in Bash include job control, history expansion, protected redirection, and several variables for controlling the interactive behavior of the shell (e.g. IGNOREEOF).

csh is a command language interpreter. When it is first invoked, csh executes commands from the file .cshrc, located in the home directory of the user. If it is a login shell, it then executes commands from the file .login in the same directory. Subsequently, if it is running in an interactive mode, csh reads commands from the terminal, prompting the user for each new line by printing a —%ll. Arguments can be passed to the shell, and it can be used to process files containing command scripts. Users can modify the default C shell environment for all users on the system by editing /etc/cshrc. The shell reads a line of command input and breaks into words. This sequence of words is placed on the command history list and then parsed. Finally, each command in the current line is executed. When a login shell terminates, it executes commands from the file .logout in the user's home directory. The shell splits input lines into words, blanks and tabs with the following exceptions. The characters & | ; > < (and) are treated as separate words. Some of these characters can be paired up; the pairs &&, ||, >>, << are treated as single words. In order to use these meta characters within other words, their special meaning must be suppressed by preceding them with a backslash (\). A newline preceded by a —\ll is equivalent to a blank. Strings Strings are enclosed in matched pairs of quotations, (_ ' or —), and form parts of a word. Meta characters in these strings, including blanks and tabs, are not treated as separate words. The semantics of strings are as follows. 1. Quoted strings delimited by pairs of (_ ' or —) characters 2. A newline preceded by a —\ll gives a true newline character 3. If the shell reads the character —#ll in its input, it treats all the text to the right of the —#ll as a comment, and ignores it 4. The —#ll character loses this special meaning if it is preceded by a backslash character (\) or placed inside quotation marks (_ ' or —). Brace Expansion Brace expansion is a mechanism by which arbitrary strings may be generated. This mechanism is similar to pathname expansion, but the file names generated need not exist. They take the form of an optional preamble, followed by a series of comma- separated strings between a pair of braces, followed by an optional postamble. The preamble is put before each string contained within the braces, and the postamble is then appended to each resulting string, expanding from left to right. Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved. For example, a{d,c,b}e expands into ade ace abe. Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result. It is strictly textual. Bash does not apply any syntactic interpretation to the context of the expansion or the text between the braces. A correctly formed brace expansion must contain unquoted opening and closing braces, and at least one unquoted comma. Any incorrectly formed brace expansion is left unchanged. This construct is typically used as shorthand when the common prefix of the strings to be generated is longer than in the above example: mkdir /usr/local/src/bash/{old,new,dist,bugs} or chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}} Commands in csh A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by —|ll characters (pipes) forms a pipeline. The output from each command in a pipeline is used as the input to the next command. Sequences of pipelines may be separated by semi-colons (;); the elements of such a sequence are executed sequentially. A sequence of pipelines may be executed without waiting for it to terminate by ending the command line with an ampersand character (&). Such a sequence is protected from termination by hangup signals sent by the shell; the nohup command need not be used. The above commands may be placed in parentheses to form a new simple command (which in turn may be used as a component of a pipeline or some other more complex command). It is also possible to separate pipelines with the logical AND||&&|| or logical OR —||ll expressions. In csh these symbols take the opposite meaning to that assumed by the C programming language and other UNIX utilities. Use of these expressions make the execution of the second pipeline conditional upon the success (logical-AND) or failure (logical-OR) of the first. The various transformations the shell performs on the input in the order in which they are carried out are as follows: History substitutions History substitutions can be used to reintroduce sequences of words from previous commands, possibly altering them in the process. Thus, history substitutions provide a general redo facility. History substitutions begin with the character —!ll and may begin anywhere in the input stream unless a history substitution is already in progress. A —!ll is preceded by a backslash (\), or followed by a space, tab, newline, —=ll or —(—, is treated as a literal —!ll and its special meaning is suppressed. History substitutions may also occur when an input line begins with —^ll.

The text of any input line containing a history substitution is echoed in the terminal after the substitution has been carried out, so that the user can see the command that is being executed. Commands entered at the terminal and consisting of one or more words are saved on the history list, the size of which is controlled by the history variable. The previous command is always retained. Commands are assigned numbers incrementally, starting with 1 (the first command executed under the current csh). Quotations with ' and " Quoted (␣) or double quoted (␣) strings are exempt from some or all of the substitutions. Strings enclosed in single quotes are not subject to interpretation. Strings enclosed in double quotes are subject to variable and command expansion. Since history (!) substitution occurs within all quotes, you must escape ␣! with a backslash (\), and the resulting text becomes all or part of a single word.

Alias Substitution The shell maintains a list of aliases, which can be established, displayed and modified by the alias and unalias commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, is checked to see if it has an alias. If it has, then the text of the alias for that command is reread, and the history mechanism is applied to it as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged. Thus, if the alias for ls is ls -l, the command ls /usr maps to ls -l /usr. Similarly if the alias for lookup is grep \!^ /etc/passwd, then lookup bill maps to grep bill /etc/passwd. There are four csh aliases distributed. These are pushd, popd, swapd, and flipd. These aliases maintain a directory stack. pushd . pushd [dir | +n | -n] pushes the current directory onto the top of the directory stacks and then changes to the directory dir. pushd with no arguments, exchanges the top two directories. +n Brings the n th directory counting from the left of the list to top of the list. -n Brings the n th directory counting from the right of the list to the top of the list dir Makes the current working directory the top of the stack

50%

MATCHING BLOCK 23/33

W

popd Changes to the directory at the top of the stack, then removes (pops) the top directory from the stack,

and announces the current directory.

swapd Swap the top two directories on the stack. The directory on the top becomes second from the top, and the second from the top directory becomes the top directory. flipd It flips between two directories, the current directory and the top directory on the stack. If you are currently in dir1, and dir2 is on the top of the stack, when flipd is invoked you change to dir2, and dir1 is replaced as the top directory of the stack. When flipd is invoked again, you change to dir1, and dir2 is again the top directory of the stack.

Input/Output The standard input and output of a command may be redirected with the following syntax: > name Opens file name (after variable, command and filename expansion) as the standard input. >> word Reads the shell input up to a line, which is identical to a word. Each input line is compared to a word before any substitutions are done on this input line. &t; name, &t;! name, &t;& name, &t;&! name opens the file name as the standard output. If the file does not exist, then it is created; if the file exists, it is overwritten. &t;&t; name, &t;&t;& name, &t;&t;! name, &t;&t;&! name Uses file name as the standard output. This is like .&t;. but places output at the end of the file. The >> mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allow the shell to block read its input.

C Shell Variables IGNOREEOF If this variable is set, it represents the number of consecutive EOFs Bash will read before exiting. By default, Bash will exit upon reading a single EOF. ctable_vars If this variable is set, Bash treats arguments to the cd command which are not directories as names of variables whose values are the directories to change to.

C-SHELL BUILT-INS Built-in commands are executed within the shell. If a built-in command occurs as any component of a pipeline except the last, then it is executed in a subshell.

COMMANDS IN C-SHELL

COMMANDS	DESCRIPTION
alias	Prints all the aliases
alias name	Prints the aliases for name
alias name wordlist	Assigns the specified wordlist as the alias of name. Wordlist is the command; filename substitution may be applied to wordlist. name is not allowed to be alias or unalias.
break	Causes execution to resume after the end of the nearest enclosing for each or while statement.
breaksw	Causes a break from a switch, resuming after the ends.
case label:	This is part of the switch statement discussed below.
cd [-L -P] [name]	Changes the shell's working directory to directory name
chdir [-L -P][name]	Changes the shell's working directory to directory name. The -L and -P flags are relevant to systems with symbolic links:
COMMANDS IN C-SHELL	COMMANDS DESCRIPTION
-L	Preserve logical pathnames so that cd -L.
-P	Use a physical model for pathnames so that cd -L.
continue	Continues execution of the nearest enclosing while or foreach.
default:	Labels the default case in a switch statement. The default should come after all case labels.
echo wordlist	The specified words are written to the shell's standard output.
exec command	The specified command is executed in place of the current shell.
exit	The shell exits either with the value of the status variable exit (expr) Exits with the value of the specified expr.

COMMANDS IN C-SHELL (CONTD.)

COMMANDS	DESCRIPTION
glob	wordlist This is like echo but no <code>—\ </code> escapes are recognized and words are delimited by null characters in the output
goto	word Filename and command expansion is applied to the specified word to yield a string of the form label:
history	This displays the history event list.
nice	The first form sets the nice for this shell to 4 nice +number Sets the nice to the given number
nice command	Runs the command at priority 4 nice +number command Runs the command at the specified number
nohup	Causes hangups to be ignored for the remainder of the Script
nohup command	Causes specified command to be run with hang- ups ignored.
nohuped	Terminates shell scripts or returns to the terminal
onintr -	Causes all interrupts to be ignored
onintr label	Causes the shell to execute a goto label
pwd[-L -P]	Prints the current working directory. The -L and -P flags are useful with symbolic links
rehash	Causes the internal hash table of the contents of the directories in the path variable to be computed again.
repeat count	command The specified command is executed count times.
: repeat count	command The specified command is executed count times.
set	Shows the value of all shells
set name	Sets name to the null string
setenv name value	This sets the value of the environment variable name to be value, which must be a single string
shift	Members of argv are shifted to the left
shift variable	Members with variable name are shifted to the left
source name	The shell reads commands from name.
time	Summary of CPU time used by this shell and its children are printed
time command	Specified simple command is timed and a time summary is printed.
Umask	The file creation mask is displayed

Non-built-in Command execution When a command to be executed is found not to be a built-in csh command, the shell attempts to execute the command via exec. Parenthesized commands are always executed in a subshell. Each word in the variable PATH names a directory from which the shell will attempt to execute the command. If it is given neither a -c nor a -t option, the shell will hash the names in these directories into an internal table so that it will only try an exec in a directory if there is a possibility that the command resides there. The shell concatenates each directory component of PATH with the given command name to form a pathname of a file which it then attempts to execute. This concatenation occurs if any of the following is true: 1. The non-built-in command execution mechanism is turned off via unhash. 2. The shell is given a -c or -t argument. 3. The directory component of PATH does not begin with a `—/|`. **LIMITATIONS OF C-SHELL** 1. Built-in control structure commands like foreach and while cannot be used with `—|`, `—&|` or `—;|` 2. Commands within loops, prompted for by `—?|`, are not placed in the history list 3. Cannot use the colon (:) modifiers on the output of command substitutions 4. The syntax of the C shell and Bourne shell commands differs 5. During intervals of heavy system load, pressing the delete key while at a C shell prompt (%) may cause the shell to exit. If csh is the login shell, the user is logged out 5. csh attempts to import and export the PATH environment variable for use with regular shell scripts 6. The `|` and `&&` operators are reversed in this implementation 7. Words can be no longer than 512 characters 6. The length of any argument of a command after filename expansion cannot exceed 159 characters 7. Command substitutions may substitute no more characters than are allowed in an argument list 8. The shell restricts the number of alias substitutions in a single line to 20

Chapter 6- The Superuser SYSTEM ADMINISTRATION The job of a system administrator is complex. It requires patience, understanding and a lot of knowledge and experience. A system administrator needs to be inventive in a crisis, and more importantly know a lot of facts and figures about the way computers work. System administration is not about installing operating systems. It is about planning and designing an efficient network of computers so that end users will be able to get their jobs done. The responsibilities of a system administrator are many and they include: ? Designing a network that is logical and efficient ? Deploying large numbers of machines that can be easily upgraded later ? Deciding what services are needed ? Planning and implementing adequate security measures ? Providing a comfortable environment for users ? Developing ways of fixing errors and problems as and when they occur ? Keeping track of and understanding how to use the enormous amount of knowledge, which increases every year As a system administrator, your first responsibility is to the greater network community and then to the users of your system. Some system administrators are responsible for both the hardware of the network and the computers, i.e. the cables as well as the computers. Some are only responsible for the computers. Either way, an understanding of how data flows from machine to machine is essential as well as an understanding of how each machine affects every other. Anyone who uses a system regularly must have a personal login account. A login account gives a person unique work areas on the system (a home directory) where the person can store files and customize the desktop environment. The system automatically labels the work area and all files that the person creates with the person's login name Each time the person begins a session on the system, he types his login name and, if necessary, an associated password. Root user/ Super user In Unix, a privileged account is called root. A root account is necessary, as many system administration files and programs need to be kept separate from the executables available to non-privileged users. Unix allows users to set permissions on the files they own. However on some occasions a system administrator may need to override these permissions. A superuser is a privileged user who has unrestricted access to the whole system. He/She can access all commands and files regardless of their permissions. By

convention, the username for the superuser account is root. A superuser account is a privileged account with unrestricted access to all files and commands. Many administrative tasks can only be performed by a superuser account. The Unix system grants superuser privileges to any account with a UID of 0 (zero). Typically, Unix systems have only one superuser account accessed with the username root. However, it is possible to have multiple superuser accounts using any valid usernames. A person with access to a superuser account is called a superuser. A super user has the following rights. ? Only the superuser can shutdown a Unix machine ? The superuser can read and delete your files regardless of the file permissions Access to the root account is restricted by a password. Because the superuser has the potential to affect the security of the entire system, it is recommended that this password be given only to people who absolutely need it, such as the system administrator. It is also a good idea to change the password often on this account. There are several ways to log into the root account. If a system comes up in single user mode whoever is logged in automatically has root privileges. When a system is already up in multi-user mode, a user can log in directly as root. This is not recommended, as it is easy to take superuser privileges for granted and perform mundane tasks in this mode. When a user is already logged in, issuing the su command without options will cause the system to prompt for the root password. Once it is given the user becomes root. The root account has its own shell and frequently displays a prompt that is different from the normal user prompt. If this is not the case, changing the default shell for the root account will change the prompt. Commands and programs that a system administrator will need as root are kept in /etc to decrease the chances of a user trying them by accident. For example, /etc contains /etc/passwd, which holds

87%

MATCHING BLOCK 24/33

W

a list of all users who have permissions to use the system.

Because the root account has extensive privileges it has an equal potential for destruction. This is amplified by the fact that safeguards built into some commands do not apply to this account. For example, the superuser may change another user's password without knowing the old password. The superuser can also mount and unmount file systems, remove any file or directory, and shut down the entire system. The root account should be used with caution and only when necessary. A misplaced keystroke in this mode may sometimes be disastrous. su Command The su command makes the user a superuser or another user. Its syntax is as shown below.

su [-] [name [arg ...]] The su command allows authorized users to change their user id to that of another user without logging off. The default user name is root. If a user has su authorization then they can grant su status to any account, providing they know the password for that account. If the user does not have su authorization, they can super use only to their own account or to another account that they own, or to an account that has the same owner as the current account. To use su, the appropriate password must be supplied (unless you are already the super user). If the password is correct, su will execute a new shell with the user ID, group ID, and supplemental group list set to those of the specified user. The new shell also has the kernel and subsystem authorizations of the specified user, although the LUID is not changed. The new shell is defined by the program field in /etc/passwd. /bin/sh is run by default if no program is specified. To restore normal user ID privileges, press EOF >Ctrl<d to exit the new shell. Any additional arguments given on the command line are passed to the program invoked as the shell. When using programs like sh, an arg of the form -c string executes string via the shell and an arg of -r gives the user a restricted shell. You must specify a username with the -c option; for example, su root -c scoadmin. When you exit the system administration shell, you will no longer be root. The following statements are true only if the optional program named in the shell field of the specified user's password file entry is like sh. If the first argument to su is a `__-` the environment is changed to what would be expected if the user actually logged in as the specified user. This is done by invoking the program used as the shell with an arg0 value whose first character is `__-` thus causing the system's profile (/etc/profile) and then the specified user's profile (.profile in the new \$HOME directory) to be executed. Otherwise, the environment is passed along with the possible exception of \$PATH, which is set to /bin:/etc:/usr/bin for root. The `__-` option should never be used in /etc/rc scripts. Note that if the optional program used as the shell is /bin/sh, the user's .profile can check arg0 for -sh or -su to determine if it was invoked by login(M) or su, respectively. If the user's program is other than /bin/sh, then .profile is invoked with an arg0 of -program by both login and su. The file /etc/default/su can be used to control several aspects of how su is used. Several entries can be placed in /etc/default/su:

VARIOUS ENTRIES IN /ETC/DEFAULT/SU LOG ENTRIES MEANING SULO Name of log file to record all attempts to use su. (Usually /usr/adm/sulog.) If this is not set, no logfile is kept. PATH The PATH environment variable is set for non- root users. If it is not, it defaults to: /bin:/usr/bin. The current PATH environment variable is ignored. SUPATH The PATH environment variable is set for root. If not set, it defaults to /bin:/etc:/usr/bin. The current PATH is ignored. CONSOLE Attempts to use su to change to the root account are logged to the named device, independently of SULO. For example, if you want to log all attempts by users to become root, edit the file /etc/default/su. In this file, place a string similar to: SULO=/usr/adm/sulog This causes all attempts by any user to switch user IDs to be recorded in the file /usr/adm/sulog. This filename is arbitrary. The su logfile records the original user, the UID of the su attempt, and the time of the attempt. If the attempt is successful, a plus sign (+) is placed on the line describing the attempt. A minus sign (-) indicates an unsuccessful attempt. EXAMPLES To become user bin while retaining your previously exported environment, enter: \$ su bin To become user bn but want to change the environment, to which the bin originally belonged, enter: \$ su - bin To execute command with the temporary environment and permissions of user bin, enter: \$ su - bin -c command args If you run su to change to a different user account, and that different user account does not have the permission to access the current directory, you will be unable to use commands such as pwd and ls.

SECURITY ISSUES When you think about security, the first thing that comes to your mind is password security. Password security is the first line of defense against intruders. Experience shows that many users have little or no idea about the importance of using a good password. Though Passwords are not visible to users, their encrypted form is. There are many publicly available programs, which can guess passwords and compare them with the encrypted forms. No one with an easy password is safe. Passwords should never be a word from a dictionary or a simple variation of such a word or name. It takes just a few seconds to guess these. Some new operating systems like FreeBSD, NetBSD and Solaris and GNU/Linux have shadow password files, which are not readable by users. The regular password file contains an 'x' instead of a password, and the encrypted password is kept in an unreadable file. This makes it much harder to scan the password file for weak passwords. Once a malicious user has gained access to an account, it is easy to exploit other weaknesses in security. Good passwords are the key to a safe system. On the network, it is possible to install programs such as crack which are designed to break passwords. It is not unusual to find a few accounts with trivial passwords in the space of a few seconds.

89%

MATCHING BLOCK 25/33

SA BCA 303 LINUX.docx (D53176565)

Backup Media: Backup media generally falls into two categories. One category is tape media and the other is removal cartridge. Tape media is generally cheaper and supports large sizes. However, tape media does not easily support random access to information. Removal cartridge drives, whether optical or magnetic, do support random access to information. But they have

a

100%

MATCHING BLOCK 27/33

SA BCA 303 LINUX.docx (D53176565)

lower capacity and a much higher cost per megabyte than tape media. Finally, optical drives generally retain information for a longer time period than tapes and may be used if a permanent archival is needed.

96%

MATCHING BLOCK 26/33

SA BCA 303 LINUX.docx (D53176565)

Among tape media, the two most common choices now are 4mm DAT (digital audio tape) and 8MM Video Tape. The 4mm tape supports up to 2GB of data on a single tape (with compression this can approach 8GB). The 8mm tape supports up to 7GB of data on a tape (with compression this can approach 25GB). Both of these technologies have been in existence since the late eighties and are relatively proven. However, while they changed the dynamics of backup at their introduction they are now having problems keeping up with the growth in data. One of the principal problems is speed. At their maximum they can backup about 300K bytes/sec or just

less than 1

95%

MATCHING BLOCK 28/33

SA BCA 303 LINUX.docx (D53176565)

GB in an hour. Among cartridge media, optical is the most commonly used. WORM (write once read many) is the primary choice. Worm drives can store between 600MB and 10GB on a platter and have a throughput similar to cartridge tapes for writing data. Optical storage will last longer and supports random access to data. These features make optical storage useful for

recording information. UNIX has many backup options. They are, The tar program: It is a tape archive program, easy to use and transferable. The syntax is as follows: tar [key] [files] The tar program saves and restores files to and from an archive medium; typically, this is a floppy disk, a tape, or a regular file. The key controls the actions of tar; a string of characters containing at most one function letter and possibly one or more function modifiers. The file specifies the ones to be backed up or restored. If a directory name is specified, this implies that the files and the entire subdirectory tree of the directory are to be backed up or restored. You can specify one of the following function letters in the tar program: VARIOUS OPTIONS IN TAR PROGRAM FUNCTION LETTER MEANING c Creates a new archive; writing begins at the beginning of the archive, instead of after the last file. C Creates a new archive as above, containing compressed files. The files to be archived should be first compressed using compress(C). When the archive is unpacked, tar will pipe the files through uncompress(C), expanding them (if necessary). Note that this function modifier has no effect on filenames. That is, files stored with a .Z suffix retain them when unpacked, even if they have been uncompressed. r The named files are written to the end of an existing archive. These files of the directory are to be backed up or restored. The function letter is only valid for appending files to disk archives. When specifying the absolute path of an archive device with the f function modifier, use the n function modifier to indicate that the device is not a magnetic tape. t The names of the specified files are listed each time that they occur on the archive. If no files argument is given, all the names on the archive are listed.

VARIOUS OPTIONS IN TAR PROGRAM (CONTD.) FUNCTION LETTER MEANING u The named files are added to the archive

76%

MATCHING BLOCK 29/33

W

if they are not already there, or if they have been modified since last written

on that archive. This function letter cannot be used with tape devices. x The named files are extracted from the archive. If a named file matches a directory whose contents had been written onto the archive, this directory is (recursively) extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, the entire contents of the archive are extracted. Note that if several files with the same name are on the archive, the last one overwrites all earlier ones. There is no way to ask for the nth occurrence of a file. The cpio command: The cpio (copy in/out archive program) is rarely used except on older System V type machines. They

100%

MATCHING BLOCK 30/33

SA

BCA 303 LINUX.docx (D53176565)

must be given a list of file names to archive.

The syntax is as follows: cpio -o [-aBLuvV] [-C bufsize] [-c | -H format] [-K volumesize] [[-O file [, file ...]] [-M message]] [-Pifd,ofd] cpio -i [-6AbBcdfkmnqrsStTuvV] [-C bufsize] [[-I file [, file ...]] [-M message]] [-Pifd,ofd] [pattern ...] cpio -p [-adLLmruvV] [-Pifd,ofd] directory The cpio command copies files to and from an archive file or from another directory hierarchy. There are three main options to cpio which determine its mode of operation. These modes are used to create an archive (cpio -o), extract files from an archive (cpio -i), and copy files from one filesystem to another (cpio -p). There are a number of secondary options, which are interpreted differently depending, on which mode of operation cpio is in.

VARIOUS OPTION IN CPIO OPTION MEANING -o (copy out) Reads a list of pathnames from the standard input, and copies those files onto the standard output together with pathname and status information. Output is padded to a 512- byte boundary by default. -i (copy in) Extracts files from the standard input, which is assumed to be the product of previous cpio -o. Only files with names that match the wildcard patterns, are selected. Extracted files are conditionally created and copied into the current directory tree in accordance with the secondary options. If cpio is used to copy files by a process without appropriate privileges, the access permissions are set in the same fashion that creat(S) would have set them when given the mode argument, matching the file permissions supplied by the c_mode field of the cpio format. The owner and group of the files will be that of the current user unless the user is root, which causes cpio to retain the owner and group of the files of the previous cpio -o. -p (pass) Reads the standard input to obtain a list of pathnames of files that are conditionally created and copied into the destination directory tree based upon the available options. Archives of text files created by cpio are portable between implementations of UNIX System V. The dump Command: The dump command is used to dump the selected parts of an object file. Its syntax is as given below: dump [options] files The dump command outputs selected parts of each of its input object files. The output can be directed via the standard redirection tools. The dump command accepts both object files and archives of object files, and can operate on both COFF and ELF object files. Dump scans each input file to determine its file format and executes the appropriate binary. The file format of the first module in an archive library is used as the file format for all modules in that library. Use the modifier -b to enforce dump to use a particular file format. Certain options and modifiers described here are applicable to either ELF or COFF file format, not both.

VARIOUS OPTIONS OF DUMP OPTION MEANING -a Dump the archive header of each member of each archive file argument. -c Dump the string table. -C Dump decoded C++ symbol table names. -D Dump debugging information. -f Dump each file header. -g Dump the global symbols in the symbol table of an archive -h Dump section headers. -L Dump dynamic linking information and static shared library information (the contents of the .lib sections), if available. -l Dump line number information. -o Dump each optional header. -r Dump relocation information. -s Dump section contents. -t Dump symbol table entries. -T index -T index1, index2 Dump only the indexed symbol table entry defined by index or a range of entries defined by index1, index2. This is an ELF only option. -V Print the dump version number. -z name Dump line number entries for the named function.

DISK MANAGEMENT One of the features that make Unix especially attractive is the structure of its file system. Unix makes use of an inverted branching-tree file structure. The tree trunk consists of a single large file, `—/ll` (called root), which contains all the files on the system. (In this structure, all files, which contain other files, are called directories). Any number of subdirectories may branch from this main trunk. Disk management involves finding the disk usage, free space available on the disk and effective utilization of the disk. If disk management is not properly administered you will run out of disk space on your system. UNIX supports many disk management commands. Some of them are discussed here. The du Command: This command summarizes the disk usage. The syntax is shown below.

`du [-afkrsuVx] [names]` The du command gives the number of blocks contained in all files and directories recursively within each directory and file specified by the list in the names argument. The block count includes the indirect blocks of the file. If names are not supplied, the current directory is used. The du command has the following options: VARIOUS DU COMMAND OPTIONS OPTION MEANING -a causes an entry to be generated for each file. Without the .a option, the default behavior is to output the block count for directories and those files explicitly named by the names argument. -f has the same effect as the -x option. -k causes du to report in units of 1024 bytes. The default is to report in units of 512 bytes. -r causes du to report directories that cannot be read, files that cannot be opened, and so on. This option is obsolete since this is now the default behavior of du. -s causes only the grand total (for each of the specified names) to be given. -u causes du to ignore files that have more than one link. -V causes du to display a three-column output reporting the space usage for versioned files. The first column displays the current space taken up by files in the directory. The second shows the space taken up by previous files, that is, files, which have been deleted, and the space shown is that used by the hidden versioned file(s). The third is the total of the first and second columns, providing usage information for versioned files. -x causes du to display the usage of files in the current filesystem only. Directories containing mounted filesystems will be ignored. A file with two or more links is only counted once. Symbolic links are not followed, but the disk space used to hold the actual symbolic link is counted. By default, this utility reports sizes in 512-byte blocks. The du utility interprets 1block from a 1024-byte block system as 2 of its own 512-byte blocks; thus, du would report

a 500-byte file as using 2 blocks rather than 1. The du command returns the following values: RETURN VALUES OF DU COMMAND VALUE MEANING 0 successful completion <0 an error occurred

The df command: This command reports the number of free disk blocks. The syntax is as follows: `df [-B | -P] [-k] [filesystem ...]` `df [-iv] [-flt] [-k] [filesystem ...]` `df [-l] [filesystem ...]` The df command reports the number of free blocks and free inodes available for on- line filesystems by examining the counts kept in the super-blocks. You can specify the filesystems to be examined using their device name (for example, `/dev/root`). If the filesystems are not specified, df writes the free space on all mounted filesystems to the standard output. The df command understands the following options: VARIOUS OPTIONS OF DF COMMANDS OPTIONS MEANING -B Uses portable XPG4/POSIX2 output formatting (as for option -P) but does not truncate the filesystem device name. -f Reports only an actual count of the blocks in the free list (free inodes are not reported). With this option, df reports on raw devices. -i Reports the percent of inodes used as well as the number of free inodes. -l Reports inode information using the same format as the -B option. -k Reports blocks as 1024-byte logical blocks instead of default 512-byte physical blocks. -l Reports local resources only. -P Uses portable XPG4/POSIX2 output formatting. The first line of the output is a header that includes the block size. Lines following the header show the following information for each filesystem: device name, total spaces, space used, free space, percentage of space used, and mount point.

VARIOUS OPTIONS OF DF COMMANDS (CONTD.) OPTIONS MEANING -t Reports the total number of allocated blocks as well as the number of free blocks. -v Reports the percent of used blocks as well as the numbers of used and free blocks. FILE SYSTEM MOUNTING All UNIX systems are provided with at least one permanent non-removable hard disk system. The root directory and the directories below it are stored on this disk. Its structure is known as the root file system. If an additional hard disk is added, UNIX creates a separate new filesystem on this disk. Before it can be used, it must be attached to the root file system. This is called mounting an additional filesystem. An empty directory on the root file system is selected as the mount point, and using the UNIX mount command, the new file system will appear under this directory. Most UNIX machines store their files on magnetic disk drives. A disk drive is a device that can store information by making electrical imprints on a magnetic surface. One or more heads skim close to the spinning magnetic plate, and can detect or change, the magnetic state of a given spot on the disk. The drives use disk controllers to position the head at the correct place at the correct time to read from, or write to the magnetic surface of the plate. It is often possible to partition a single disk drive into more than one logical storage area. Every item in a UNIX file system can be defined as belonging to one of the four possible types: ? Ordinary files: Ordinary files can contain text, data, or program information. An ordinary file cannot contain another file or directory. It can be thought of as a one dimensional array of bytes. ? Directories: A directory is actually implemented as a file that has one line for each item contained within the directory. Each line in a directory file contains only the name of the item, and a numerical reference to the location of the item. The reference is called an i-number, and is an index to a table known as the i-list. The i-list is a complete list of all the storage space available for the file system. ? Special files: Special files represent input/output (i/o) devices, like a tty (terminal), a disk drive, or a printer. Because UNIX treats such devices as files, a degree of compatibility can be achieved between device i/o, and ordinary file i/o, allowing for a more efficient use of software. Special files can be either character special files that deal with streams of characters or block special files that operate on larger blocks of data. Typical block sizes are 512 bytes, 1024 bytes, and 2048 bytes. ? Links: A link is a pointer to another file. Remember that a directory is nothing more than a list of names and i-numbers of files. A directory entry can be a hard link, in which the i-number points directly to another file.

95%

MATCHING BLOCK 31/33

W

A hard link to a file is indistinguishable from the file

itself. When a hard link is made, then the i-numbers of two different directory file entries point to the same inode. For that reason, hard links cannot span across file systems. A soft link (or symbolic link) provides an indirect pointer to a file. A soft link is implemented as a directory file entry containing a pathname. Soft links are distinguishable from files, and can span across file systems. Not all versions of UNIX support soft links. Mount Command: This command is used to mount a file system on to directories. Only the superuser may invoke it. It is intended for use only by the mount (ADM) utility. The mount command fails if one or more of the following is true: ? EBUSY o dir is currently mounted on, is someone's current working directory, or is otherwise busy o The device associated with spec is currently mounted o There are no more mount table entries ? EFAULT: spec or dir points outside the allocated address space of the process ? EINVAL: The super block has an invalid magic number or the fstyp is invalid or mflag is not valid ? EMULTIHOP: Components of path require hopping to multiple remote machines ? ENOENT: Any of the named files does not exist ? ENOLINK: path points to a remote machine and the link to that machine is no longer active ? ENOSPC: The filesystem state in the super block is not FsOKAY and mflag requests write permission ? ENOTBLK: spec is not a block special device ? ENOTDIR o A component of a path prefix is not a directory o dir is not a directory ? ENXIO : The device associated with spec does not exist ? EPERM: The effective user ID is not superuser ? EREMOTE: spec is remote and cannot be mounted ? EROFS: spec is write-protected and mflag requests write permission The mount command returns 0 upon successful completion. Otherwise, a value of -1 is returned, and errno is set to indicate the error.

Umount Command: The Umount command is used for unmounting the file system. After unmounting the directory upon which the file system was mounted reverts to its ordinary interpretation. Only the superuser may invoke the umount system call. The umount system call fails if one or more of the following is true: ? EBUSY: A file on file is busy ? EFAULT: file points to an illegal address ? EINVAL: file does not exist, or is not mounted ? EMULTIHOP: Components of the path pointed to by file require hopping to multiple remote machines ? ENOENT: The named file does not exist ? ENOLINK: file is on a remote machine, and the link to that machine is no longer active ? ENOTBLK: file is not a block special device ? ENOTDIR: A component of the path-prefix is not a directory ? EPERM: The process's effective user ID is not superuser ? EREMOTE: file is remote The umount command returns a value of 0 upon successful completion. Otherwise, a value of -1 is returned and errno is set to indicate the error.

CHECKING THE NETWORK Ping is a way of checking an Internet connection to see if a computer at the other end is online or not. This means of testing is called as ping. If you get the echo response from the machine that you ping, it means that the machine is online. In the case of sites, which should always be online, such as public providers, ftp sites, and the like, ping will let you know if there is a network problem preventing you from accessing that machine. Sometimes you may be able to ping a site, and still not be able to access the service, due to some temporary local problem at the site. Whenever you are online, other machines on the Internet can ping you. The ping (Packet InterNet Groper) is a program used to check whether destinations are reachable by sending an ICMP echo request and waiting for a reply. ICMP is an extension of Internet protocol. It allows for the generation of error messages, test packets and informational messages related to IP. The syntax of the ping command is shown below. ping [-dLmnrqtvr] [-c count] [-I ip-addr] [-i wait] [-l preload] [-p pattern] [-s packetsize] [-T ttl] host | source-route The ping program uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams (pings) have an IP and ICMP header, followed by a struct timeval and an arbitrary number of pad bytes used to fill out the packet. The options are as follows:

VARIOUS OPTIONS OF PING OPTIONS MEANING -c Count Stop after sending (and receiving) count ECHO_RESPONSE packets. -d Set the SO_DEBUG option on the socket being used. -f Flood ping: Outputs packets as fast as they come back or one hundred times per seconds, whichever is more. For every ECHO_REQUEST sent, a period (.) is printed, while for every ECHO_REPLY received a backspace is printed. This provides a rapid display of how many packets are being dropped. Only root may use this option. This can be very hard on a network and should be used with caution. -I ip-addr When sending multicast datagrams, use the interface with address ip-addr as the originating interface. -i wait Wait, wait seconds between sending each packet. The default is to wait for one second between each packet. This option is incompatible with the -f option. -L Causes loopback of multicast datagrams to be disabled. -l preload If preload is specified, ping sends that many packets as fast as possible before falling into its normal mode. -m Sends an ICMP subnet-mask request to the specified address. This is useful for checking the configuration of a particular host. -n Numeric output only: No attempt will be made to look up symbolic names for host addresses. -p pattern You may specify up to 16 pad bytes to fill out the packet you send. This is useful for diagnosing data-dependent problems in a network. For example, -p ff will cause the sent packet to be filled with all ones. q Quiet output: Nothing is displayed except the summary lines at startup and when finished. -R Records route. Includes the RECORD_ROUTE option in the ECHO_REQUEST packet and displays the route buffer on returned packets. Note that the IP header is only large enough for nine such routes. Many hosts ignore or discard this option.

VARIOUS OPTIONS OF PING (CONTD.) OPTIONS MEANING -r Bypasses the normal routing tables and sends directly to a host on an attached network. If the host is not on a directly attached network, an error is returned. This option can be used to ping a local host through an interface that has no route through it (for example, after the interface was dropped by routed (ADMN)). -s packetsize Specifies the number of data bytes to be sent. The default is 56, which translates into 64 ICMP data bytes when combined with the 8 bytes of ICMP header data. -T ttl Sets the IP time-to-live (TTL) in outgoing datagrams to ttl. -t Sends an ICMP time-stamp request to the specified address. This is useful for checking the configuration of a particular host. -v Verbose output. ICMP packet details: An IP header without options is 20 bytes. An ICMP ECHO_REQUEST packet contains an additional 8 bytes worth of ICMP header followed by an arbitrary amount of data. When a packet size is given, it indicates the size of this extra piece of data (the default is 56). Thus the amount of data received inside of an IP packet of type ICMP ECHO_REPLY will always be 8 bytes more than the requested data space (the ICMP header). If the data space is at least eight bytes large, ping uses the first 8 bytes of this space to include a timestamp, which it uses in the computation of round-trip times. If less than 8 bytes of pad are specified, no round-trip times are given. ping reports duplicate and damaged packets. Duplicate packets should never occur, and are caused by inappropriate link-level retransmissions. Duplicates may occur in many situations and are rarely a good sign, although the presence of low levels of duplicates may not always be cause for alarm. Damaged packets are obviously a cause for alarm and often indicate broken hardware somewhere in the ping packet's path (in the network or in the hosts). The (inter) network layer should never treat packets differently depending on the data contained in the data portion. Unfortunately, data-dependent problems have been known to sneak into networks and remain undetected for long periods of time. In many cases the particular pattern that will have problems is something that does not have sufficient transitions, such as all ones or all zeroes, or a pattern right at the edge, such as almost all zeroes. It is not necessary, to specify a data pattern of all zeroes (for example) on the command line because the pattern that is of interest is at the data

link level, and the relationship between what you type and what the controllers transmit can be complicated. This means that if you have a data-dependent problem you will probably have to do a lot of testing to find it. If you are lucky, you may manage to find a file that cannot be sent across your network or one that takes much longer to transfer than other similar length files. You can then examine this file for repeated patterns that you can test using the `-p` option of ping. The TTL value of an IP packet represents the maximum number of IP routers that the packet can go through before being thrown away. In current practice you can expect each router in the Internet to decrement the TTL field by exactly one. The TCP/IP specification states that the TTL field for TCP packets should be set to 60, but many systems use smaller values (4.3 BSD uses 30, 4.2 used 15). The maximum possible value of this field is 255, and most UNIX systems set the TTL field of ICMP ECHO_REQUEST packets to 255. This is why you will find you can ping some hosts but cannot reach them with telnet or ftp. In its normal mode ping prints the TTL value from the packet it receives. When a remote system receives a ping packet, it can do one of the three things with the TTL field. ? Not change it; this is what the Berkeley UNIX systems did before the 4.3 BSD tahoe release. In this case the TTL value in the received packet will be 255 minus the number of routers in the round-trip path ? Set it to 255; this is what Berkeley UNIX systems currently do. In this case the TTL value in the received packet will be 255 minus the number of routers in the path from the remote system to the pinging host ? Set it to some other value; some machines use the same value for ICMP packets that they use for TCP packets, for example either 30 or 60. Others may use completely wild values. Many hosts and gateways ignore the RECORD_ROUTE option. The maximum IP header length is too small for options like RECORD_ROUTE to be completely useful. Flood pingging is not recommended in general, and flood pingging the broadcast address should only be done under very controlled conditions.

Appendix UNIX INSTALLATION Installing UNIX on a machine requires more thought and planning than installing DOS or Microsoft Windows. You need to decide if this system will be stand-alone or dependent on a server on your network. You also have to pay careful attention to system resources such as hard drive space, processor speed, memory and so on. DOS and Windows are not designed to easily share large sections of the installation. UNIX (especially because of its disk needs) almost expects that some sharing will occur. The degree of disk space sharing leads to the definition of stand-alone, server and diskless machines. ? Stand-alone system: A stand-alone system means that this particular machine can function on its own. It doesn't require any assistance from any other machine on the LAN ? Server: A server is a machine that is connected to the LAN that runs daemons to give remote clients some functions such as mail or news. Technically, a server can be a stand-alone machine, but because of its tasks, it never is ? Diskless: If the client system has no disk drive at all, it is considered diskless. It depends on its server for booting, for the entire operating system and for swap space. Many people use such machines as dumb terminals or machines that just provide an interface to a remote machine. Type of Users UNIX users generally fall into one of the following categories: ?

Application users: These users run commercial or locally developed applications. They rarely interact with the shell directly and do not write their own applications. These users might be running a database application, a word processor or desktop publishing system, a spreadsheet, or some in-house-developed set of applications. They spend most of their time in think mode, where they are deciding what to do with the results the application has presented them, or in data entry mode, typing responses or data into the system. Their need for large amounts of local disk access is minimal, and they do not change applications frequently.

Also they won't be running many applications simultaneously. Although application users might put a large load on their database servers, they do not normally put large disk loads on their own systems. ? Power users: These users run applications, just like the application users, but they also run shell scripts and interact more closely with the system. They are likely to be running multiple applications at once, with all these applications processing in parallel. These users keep several applications busy and access the disk more frequently and use more CPU resources than do the normal

application users. ? Developers: Developers not only run applications, they also run compilers, access different applications than users, require access to the development libraries, and generally use more components of the operating system than do users.

Furthermore, they tend to use debugging tools that require more swap space and access to more disk resources than the application user generally needs. Partitions The following partitions are required on all UNIX installations: root and swap. It is recommended that you create partitions to hold `usr`, `var`, `home` and `tmp`. The root Partition The root partition is mounted at the top of the file system hierarchy. It is mounted automatically as the system boots, and it cannot be unmounted. All other file systems are mounted below the root. The root needs to be large enough to hold the following: ? The boot information and the bootable UNIX kernel, and a backup copy of the kernel in case the main one gets damaged ? Any local system configuration files, which are typically in the `/etc` directory ? Any stand-alone programs, such as diagnostics, that might be run instead of the OS This partition typically runs on between 10 and 20 MB. It is also usually placed on the first slice of the disk, often called slice 0. The swap Partition The default rule is that there is twice as much swap space as there is RAM installed on the system. If you have 16 MB of RAM, the swap space needs to be a minimum of 32 MB. If you have 256 MB of RAM, the recommended swap is 512 MB. If you are unsure as to the swap needs of your users, start with the rule of twice RAM. Monitor the amount of swap space used via the `pstat` or `swap` commands. If you did not allocate enough, most UNIX systems support adding additional swap at runtime via the `swapon` or `swap` commands. The `usr` Partition The `usr` slice holds the remainder of the UNIX operating system and utilities. It needs to be large enough to hold all the packages you chose to install when you made the list earlier. If you intend to install local applications or third-party applications in this partition, it needs to be large enough to hold them as well. The `var` Partition The `var` partition holds the spool directories used to queue printer files and electronic mail, as well as log files unique to this system. It also holds the `/var/tmp` directory, which is used for larger temporary files. Every system, even a diskless client, needs its own `var` file system. It cannot be shared with other systems.

The home Partition This is where the user's login directories are placed. Making home its own slice prevents users from hurting anything else on the system if they run this file system out of space. A good starting point for this slice is 5 MB per application user plus 10 MB per power user and 20 MB per developer you intend to support on this system. The tmp Partition Large temporary files are placed in /var/tmp but sufficient temporary files are placed in /tmp that you don't want it to run your root file system out of space. If your users are mostly application users, 5 to 10 MB is sufficient for this slice. If they are power users or developers, 10 to 20 MB is better. If there are more than 10 users on the system at once, consider doubling the size of this slice.

Assigning Partitions to Disk Drives If you have more than one disk drive, a second decision you have is on which drive to place the partitions. The goal is to balance the disk accesses between all of the drives. If you have two drives, consider the following partitioning scheme: Drive1 Drive2 Root usr Swap Home Var

Assigning IP Addresses If the system has a network connection, it must be assigned an IP address. An IP address is a set of four numbers separated by dots, called a dotted quad. Each network connection has its own IP address. Within a LAN segment, usually the first three octets of the dotted quad are the same. The fourth must be unique for each interface. The addresses 0 and 255 (all zeros and all ones) are reserved for broadcast addresses. The remaining 254 addresses may be assigned to any system.

Performing the Installation By now, if you've been following along, you should have an installation checklist. It should contain the following: ? The name of the system holding the drive for the installation, and its device name ? Diskless, stand-alone or server system ? The name of the host and domain ? The IP address ? The packages to install ? How to partition the disk ? Whether to use a network database ? Now you should be all set

Booting the Installation Media The first step in installing a UNIX system is to load the mini-root into RAM (the mini-root is basically a scaled down kernel that will give you the ability to run the UNIX installation programs). UNIX uses the UNIX operating system to perform its installation. It needs a version of UNIX it can run, and to do this the install loader uses RAM to hold a small version of the UNIX file system. When you boot the installation media, it builds a root file system and copies the files it needs to control the installation to this RAM-based file system. This is the reason it takes a while to boot the media.

Booting from CD/DVD: Take CD/DVD and place it in drive. Boot the system in the normal manner, by pressing the Ctrl+Alt+Del keys at the same time or by power cycling the machine. The system will load the boot loader off the CD/DVD and then use that to create the RAM-based file systems and load the UNIX image into RAM. It will ask for additional install media. Answer CD-ROM or tape, as appropriate, and the system will then load the remainder of the mini-root from the installation media.

Installing the Master System Once the mini-root is loaded, you are generally presented with the install options. Some systems leave you at a shell prompt. If this happens, enter install to start the installation procedure. Your distribution may be different, or it may be automatic. Follow the installation procedure located in your manual. UNIX contains a set of install procedures that walk you through the installation. They are almost identical to one another in concept, but they are slightly different in implementation.

Installing Optional or Additional Packages Once the system is installed and rebooted, you are running UNIX. Of course, you will still need to perform installations from time to time to add packages and applications. All UNIX packages and most standard applications for System V Release 4 use the pkgadd format. Installation of these packages and applications is automatic using the pkgadd utility. Using pkgadd and pkgrm Packages are added to System V Release 4 systems by using the pkgadd command. This command automatically installs the software from the release media and updates a database of what is currently installed on the system. Packages are deleted just as easily with the pkgrm command. To run pkgadd on the install media, place the media in the drive and enter the command pkgadd -d path-name-to-device pkg_name pkgadd will then prompt you for which packages to install and give you progress messages as it installs the package. Different packages may also ask you questions prior to installation. These questions usually relate to where to install the package and any other installation options.

THE ZERO HOUR Q: What three main categories can the operating system be divided into? Ans: It can be divided into Process management, Memory management, and File management. Q: What is multiprogramming? Ans: Multiprogramming is defined as the task of running many tasks at a particular time in the memory by partitioning the memory into blocks. Q: What is a shell? Ans: A Shell is a command interpreter and is in-between the user and the operating system. Q: What the various types of operating system structures available? Ans: The various operating system structures are Monolithic operating system, Layered operating system, Client-Server operating system, Virtual machines etc. Q: In what three states can a process be in? Ans: A process can be in running, ready, blocked states. Q: What is a process table? Ans: A Process table contains information about program counter, stack pointer, memory allocation etc and is unique for each process. Everything about the process must be saved when the process is switched from running to ready state so that it can be restarted later. Q: What are the basic elements of UNIX operating system? Ans: The six basic elements of UNIX are commands, files, directories, users' environment, processes, and tasks. Q: What does the command apropos do? Ans: Apropos finds manual pages that contain any of the given keywords in their short descriptions. Apropos considers each keyword separately and is insensitive to case. Apropos sort its output alphabetically by command name. Q: What is magic variable? Ans: Magic is a database of file type headers. The file /etc/magic is a database of magic numbers used by file (and utilities such as more) to help it to determine the probable use to which a regular file is put. Q: What does the command man do? Ans: Man command locates and prints the entry named title from the designated reference section. Page is often used as a synonym for entry in this context. It displays reference manual pages.

Q: What is a file in UNIX? Ans: A file in UNIX is treated as a sequence of one or more bytes containing arbitrary information about the file system and everything in UNIX is considered as a file. Q: What is a Super Block? Ans: A Super Block contains the critical information about the layout of the file system, the number of I-nodes, number of disk blocks and start of the list of free disk blocks etc. Q: What is the command used to change permissions for a file? Ans: The command that is used for changing the permissions of a file is `chmod`. Q: What is the `grep` command used for? Ans: The command stands for —get regular expression printll. It is a search command that will allow you to search files for particular expressions which it will then print to the screen. Q: What does the program `more` do? Ans: `More` is a program that displays only one screen of information at a time. It waits for user to tell it to continue. Q: What does the FDDI stand for? Ans: FDDI stands for Fiber Distributed Data Interface. Q: What does the `telnet` command? Ans: The `telnet` command communicates with another host using the TELNET protocol. If `telnet` is invoked without the host argument, it will enter command mode. In this mode, `telnet` will accept and execute the commands. If `telnet` is invoked with arguments, it will perform an open command with those arguments. Q: What is a Socket? Ans: Socket creates an endpoint for communication and returns a descriptor. Q: What are the various kinds of sockets that are available? Ans: The various kinds of sockets that are available are Stream sockets, Datagram sockets, Raw sockets. Q: What are the various directories in UNIX operating system? Ans: The various directories in UNIX are `bin`, `dev`, `etc`, `lib`, `pub`, `tmp`, and `usr`. Q: What are the three modes `vi` operates in? Ans: `vi` editor operates in command mode, insert mode and escape mode. Q: What are the various meta characters available in `vi` editor? Ans: The various meta characters available in `vi` editor or period(`.`), `asterisk`(`*`), `circumflex`(`^`), `dollar`(`$`), `/>`, `/<`.

Q: What is a character class? Ans: A character class is a multi-character meta character used in search patterns. When several characters are enclosed within a set of brackets (`[]`), the group matches any one of the characters inside the brackets. Q: What are the various screen commands? Ans: `CTL/I` Reprints current screen, `CTL/E` Exposes one more line at the bottom of screen, `CTL/F` Pages forward one screen etc. Q: What are the two types of editing that can be done by using `vi`? Ans: The two types of editing that `vi` can be used for are line editing, screen editing. Q: What are the various ways in which we can exit `vi`? Ans: `ZZ` exits `vi` and saves changes, `:wq` writes changes to current file and quits edit session, `:q!` quits edit session. Q: What are the various text deletion commands? Ans: `dd` deletes current line, `dw` deletes the current word, `d)` deletes the rest of the current sentence, `D`, `d$` deletes from cursor to end of line. Q: What does `vi` stand for? Ans: `vi` stands for visual editor. Q: What are search patterns used for? Ans: Search patterns are used for searching for a particular set of characters in the text. Q: What does the command `ignorecase` do? Ans: This option maps all uppercase characters in the text to lowercase in regular expression matching. Q: What is a process? Ans: A program or command, which is running, is called a process. Q: What does a UNIX process contain? Ans:

93%

MATCHING BLOCK 32/33

SA Linux_system_administration_block_2.pdf (D149208459)

A process under UNIX consists of an address space and a set of data structures in the kernel to keep track of that process.

Q: What does the PID field in `ps` command output mean? Ans:

95%

MATCHING BLOCK 33/33

SA Linux_system_administration_block_2.pdf (D149208459)

The PID shows the Process ID of each process. This value should be unique. Generally PID are allocated from lowest to highest, but wrap at some point. This value is necessary for you to send

a signal to a process such as the KILL signal. Q: What are the various process states? Ans: New, Ready, Running, Waiting, and Terminated

Q: What does context-switching mean? Ans: Switching from one running process to another is called context switching. Q: What are the two modes of mail program in UNIX? Ans: Send mode, Command mode Q: What is send mode? Ans: You can compose and send a mail in this mode. It has two character commands that start with the tilde (`~`) character. Q: What is command mode? Ans: You can read and manage mail in this mode. It has a single character command. Q: How many kinds of shell programs are there? Ans: There are various shells like C-Shell, Bourne shell, Bash shell, Restricted shell, SCO shell, Korn shell. Q: What is the default pattern for Bourne shell? Ans: `#!` Is the default pattern for the Bourne shell. Q: What are the commands which are processed in a different way? Ans: `:` (colon), `exec`, `shift`, `.` (dot), `.exit`, `times`, `break`, `export`, `trap`, `continue`, `read-only`, `wait`, `eval`, `return`. Q: How are quoted strings delimited? Ans: Quoted strings are delimited by `_` or `—`. Q: Where does the c-shell execute commands from when first started? Ans: C-shell executes commands from the `.cshrc` file and if login it executes the commands from `.login` file. Q: What does the command `nice` do? Ans: The command `nice` runs the process with a priority 4. Q: What command causes all the interrupts to be ignored? Ans: The command `ointr-` causes all the interrupts to be ignored. Q: What is the command used to repeat a command number of times? Ans: `repeat count` command or `:repeat count` command is used to repeat a command number of times. Q: What is a pipeline? Ans: A pipeline is a set of commands separated by a vertical bar (`|`).

Q: What is Rsh? Ans: Rsh is acronym for restricted shell and is used to setup login names and execution environments whose capabilities are controlled by the administrator. Q: What is root? Ans: In Unix, a privileged account is called root. Q: Which command makes you get the root privilege? Ans: The su command enables you to get the root privilege. Q: What are the two categories of backup media? Ans: Tape media and removal cartridge. Q: What is the function of the dump command? Ans: The dump command is used to dump the selected parts of an object file. Q: Give the function and syntax of the df command? Ans: The df command reports the number of free disk blocks. The syntax is as follows: df [-B | -P] [-k] [filesystem ...] df [-iv] [-flt] [-k] [filesystem ...] df [-l] [filesystem ...] Q: What is the function of ping command? Ans: The ping (Packet InterNet Groper) is a program used to check whether destinations are reachable by sending an ICMP echo request and waiting for a reply. Q: What is the function of the du command? Ans: The du command gives the number of blocks contained in all files and directories recursively within each directory and file specified by the list in the names argument.

Hit and source - focused comparison, Side by Side

Submitted text As student entered the text in the submitted document.

Matching text As the text appears in the source.

1/33	SUBMITTED TEXT	28 WORDS	54% MATCHING TEXT	28 WORDS
	copy of a file, use the cp (copy) command. cp source destination where source is the file you wish to copy and destination is the			
	SA BCA 303 LINUX.docx (D53176565)			
2/33	SUBMITTED TEXT	21 WORDS	62% MATCHING TEXT	21 WORDS
	the effective user ID of the process; the group ID of the process is set to the effective group ID			
	SA Linux_system_administration_block_2.pdf (D149208459)			
3/33	SUBMITTED TEXT	19 WORDS	58% MATCHING TEXT	19 WORDS
	the files in the current directory that do not begin with a period. Below is a list of		the visible files in the working directory (those files whose filenames do not begin with a period). The file-list is a list of	
	W https://usermanual.wiki/Document/PracticalGuidetoLinuxCommandsEditorsandShel.2146662400/help			
4/33	SUBMITTED TEXT	11 WORDS	100% MATCHING TEXT	11 WORDS
	r for read, w for write, and x for execute).			
	SA BCA 303 LINUX.docx (D53176565)			
5/33	SUBMITTED TEXT	13 WORDS	87% MATCHING TEXT	13 WORDS
	the pathname of the file you wish to create a link to.		the pathname of the file you want to create a link to.	
	W https://usermanual.wiki/Document/PracticalGuidetoLinuxCommandsEditorsandShel.2146662400/help			

6/33	SUBMITTED TEXT	30 WORDS	55% MATCHING TEXT	30 WORDS
<p>the size of the file (in bytes) and the date and time at which the file was last modified. The last field gives the name of the file.</p>		<p>The size of the file in characters (bytes) The date and time the file was created or last modified The name of the file</p>		
<p>W https://usermanual.wiki/Document/PracticalGuidetoLinuxCommandsEditorsandShel.2146662400/help</p>				
7/33	SUBMITTED TEXT	20 WORDS	63% MATCHING TEXT	20 WORDS
<p>the cursor to the beginning of the current word if it is within a word, or the previous word</p>		<p>the cursor moves left to the beginning of the current word (as you are entering a word) or the previous word (</p>		
<p>W https://usermanual.wiki/Document/PracticalGuidetoLinuxCommandsEditorsandShel.2146662400/help</p>				
8/33	SUBMITTED TEXT	17 WORDS	61% MATCHING TEXT	17 WORDS
<p>read and write permissions for all users: chmod ugoa+rw myfile ls -l myfile -rw-rw-rw- 1</p>		<p>read and write permissions (rw) for all (a) users: Access Permissions \$ chmod a+rw letter.0610 \$ ls -l letter.0610 -rw-rw-rw- 1</p>		
<p>W https://dokumen.pub/a-practical-guide-to-unix-for-mac-os-x-users-1733062564-0131863339-9780131863 ...</p>				
9/33	SUBMITTED TEXT	30 WORDS	53% MATCHING TEXT	30 WORDS
<p>from the current line to the end of the fifth line. ^ moves the cursor to the beginning of the current line. The ^ actually moves the cursor</p>				
<p>SA unixassignment-1(14691A0532).docx (D25928407)</p>				
10/33	SUBMITTED TEXT	35 WORDS	41% MATCHING TEXT	35 WORDS
<p>Moves the cursor backward to the beginning of the previous word. e Moves the cursor backward to the end of the previous word. w Moves the cursor forward to the next word. :pattern Moves</p>		<p>moves the cursor backward to the first letter of the previous word. The B key moves the cursor backward by blank-delimited words. Similarly the e key moves the cursor to the end of next word; E moves</p>		
<p>W https://dokumen.pub/a-practical-guide-to-unix-for-mac-os-x-users-1733062564-0131863339-9780131863 ...</p>				
11/33	SUBMITTED TEXT	23 WORDS	50% MATCHING TEXT	23 WORDS
<p>By Sentences and Paragraphs Sentences The left parenthesis —(— will move the cursor to the beginning of a sentence, and the</p>		<p>by Sentences and Paragraphs))({ The) and } keys move the cursor forward to the beginning of the next sentence or the</p>		
<p>W https://mamchenkov.net/wordpress/wp-content/uploads/2017/05/Prentice_Hall_A_Practical_Guide_to_Li ...</p>				

12/33	SUBMITTED TEXT	20 WORDS	55%	MATCHING TEXT	20 WORDS
<p>the cursor to the beginning of a paragraph, and a) moves it to the end of a paragraph.</p> <p>SA DECAP448_LINUX_AND_SHELL_SCRIPTING.pdf (D142327428)</p>					
13/33	SUBMITTED TEXT	12 WORDS	100%	MATCHING TEXT	12 WORDS
<p>filename where filename is the name of the file to be</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>					
14/33	SUBMITTED TEXT	232 WORDS	93%	MATCHING TEXT	232 WORDS
<p>A process under UNIX consists of an address space and a set of data structures in the kernel to keep track of that process. The address space is a section of memory that contains the code to execute the process stack. The kernel must keep track of the following data for each process on the system: ? The address space map ? The current status of the process ? The execution priority of the process ? The resource usage of the process ? The current signal mask ? The owner of the process A process has certain attributes that directly affect execution. These include: ? PID - The PID stands for the process identification. This is a unique number that defines the process within the kernel ? PPID - This is the processes Parent PID, the creator of the process ? UID - The ID number of the user who owns this process ? EUID - The effective User ID of the process ? GID - The Group ID of the user that owns this process ? EGID - The effective Group User ID that owns this process ? Priority - The priority that this process runs at To view a process use the ps command. \$ ps -l F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>					
15/33	SUBMITTED TEXT	61 WORDS	94%	MATCHING TEXT	61 WORDS
<p>F This is the flag field. It uses hexadecimal values, which are added to show the value of the flag bits for the process. For a normal user process this will be 30, meaning it is loaded into memory. S The S field is the state of the process, the two most common values are S (for Sleeping) and R (</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>					
16/33	SUBMITTED TEXT	21 WORDS	100%	MATCHING TEXT	21 WORDS
<p>Running). An important value to look for is X, which means the process is waiting for memory to become available.</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>					

17/33	SUBMITTED TEXT	36 WORDS	98% MATCHING TEXT	36 WORDS
<p>PID The PID shows the Process ID of each process. This value should be unique. Generally PID are allocated lowest to highest, but wrap at some point. This value is necessary for you to send</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>				
18/33	SUBMITTED TEXT	23 WORDS	97% MATCHING TEXT	23 WORDS
<p>SZ This refers to the SIZE field. This is the total number of pages in the process. Each page is 4096 bytes.</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>				
19/33	SUBMITTED TEXT	13 WORDS	95% MATCHING TEXT	13 WORDS
<p>TIME The cumulative execution time of the process in minutes and seconds.</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>				
20/33	SUBMITTED TEXT	19 WORDS	55% MATCHING TEXT	19 WORDS
<p>kill Send a signal to a process. pwd Print the current working directory. read Read a line from</p> <p>SA DECAP448_LINUX_AND_SHELL_SCRIPTING.pdf (D142327428)</p>				
21/33	SUBMITTED TEXT	15 WORDS	96% MATCHING TEXT	15 WORDS
<p>A colon-separated list of directories in which the shell looks for commands. HOME The</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>				
22/33	SUBMITTED TEXT	18 WORDS	79% MATCHING TEXT	18 WORDS
<p>The exit status of a function is the exit status of the last command executed in the</p> <p>The exit status of a command list is the exit status of the last command in the</p> <p>W https://usermanual.wiki/Document/PracticalGuidetoLinuxCommandsEditorsandShel.2146662400/help</p>				
23/33	SUBMITTED TEXT	21 WORDS	50% MATCHING TEXT	21 WORDS
<p>popd Changes to the directory at the top of the stack, then removes (pops) the top directory from the stack,</p> <p>popd Changes the working directory to the directory on the top of the directory stack and removes that directory from the directory stack (</p> <p>W https://mamchenkov.net/wordpress/wp-content/uploads/2017/05/Prentice_Hall_A_Practical_Guide_to_Li...</p>				

24/33	SUBMITTED TEXT	14 WORDS	87% MATCHING TEXT	14 WORDS
<p>a list of all users who have permissions to use the system. a list of all users who have permission to use the system. /</p> <p>W https://usermanual.wiki/Document/PracticalGuidetoLinuxCommandsEditorsandShel.2146662400/help</p>				
25/33	SUBMITTED TEXT	57 WORDS	89% MATCHING TEXT	57 WORDS
<p>Backup Media: Backup media generally falls into two categories. One category is tape media and the other is removal cartridge. Tape media is generally cheaper and supports large sizes. However, tape media does not easily support random access to information. Removal cartridge drives, whether optical or magnetic, do support random access to information. But they have</p> <p>SA BCA 303 LINUX.docx (D53176565)</p>				
26/33	SUBMITTED TEXT	114 WORDS	96% MATCHING TEXT	114 WORDS
<p>Among tape media, the two most common choices now are 4mm DAT (digital audio tape) and 8MM Video Tape. The 4mm tape supports up to 2GB of data on a single tape (with compression this can approach 8GB). The 8mm tape supports up to 7GB of data on a tape (with compression this can approach 25GB). Both of these technologies have been in existence since the late eighties and are relatively proven. However, while they changed the dynamics of backup at their introduction they are now having problems keeping up with the growth in data. One of the principal problems is speed. At their maximum they can backup about 300K bytes/sec or just</p> <p>SA BCA 303 LINUX.docx (D53176565)</p>				
27/33	SUBMITTED TEXT	37 WORDS	100% MATCHING TEXT	37 WORDS
<p>lower capacity and a much higher cost per megabyte than tape media. Finally, optical drives generally retain information for a longer time period than tapes and may be used if a permanent archival is needed.</p> <p>SA BCA 303 LINUX.docx (D53176565)</p>				
28/33	SUBMITTED TEXT	66 WORDS	95% MATCHING TEXT	66 WORDS
<p>GB in an hour. Among cartridge media, optical is the most commonly used. WORM (write once read many) is the primary choice. Worm drives can store between 600MB and 10GB on a platter and have a throughput similar to cartridge tapes for writing data. Optical storage will last longer and supports random access to data. These features make optical storage useful for</p> <p>SA BCA 303 LINUX.docx (D53176565)</p>				

29/33	SUBMITTED TEXT	16 WORDS	76% MATCHING TEXT	16 WORDS
<p>if they are not already there, or if they have been modified since last written</p> <p>W https://usermanual.wiki/Document/PracticalGuidetoLinuxCommandsEditorsandShel.2146662400/help</p>		<p>if they are not already in the archive or if they have been modified since they were last written</p>		
30/33	SUBMITTED TEXT	11 WORDS	100% MATCHING TEXT	11 WORDS
<p>must be given a list of file names to archive.</p> <p>SA BCA 303 LINUX.docx (D53176565)</p>				
31/33	SUBMITTED TEXT	12 WORDS	95% MATCHING TEXT	12 WORDS
<p>A hard link to a file is indistinguishable from the file</p> <p>W https://usermanual.wiki/Document/PracticalGuidetoLinuxCommandsEditorsandShel.2146662400/help</p>		<p>A hard link to a file is indistinguishable from the original file.</p>		
32/33	SUBMITTED TEXT	26 WORDS	93% MATCHING TEXT	26 WORDS
<p>A process under UNIX consists of an address space and a set of data structures in the kernel to keep track of that process.</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>				
33/33	SUBMITTED TEXT	36 WORDS	95% MATCHING TEXT	36 WORDS
<p>The PID shows the Process ID of each process. This value should be unique. Generally PID are allocated from lowest to highest, but wrap at some point. This value is necessary for you to send</p> <p>SA Linux_system_administration_block_2.pdf (D149208459)</p>				