UNIX AND SHELL PROGRAMMING

Prof. Anoop Chaturvedi
Prof. B. L. Rai

UNIX AND SHELL PROGRAMMING

UNIX AND SHELL PROGRAMMING

By

ANOOP CHATURVEDI
MCA, M.Tech(CSE) (Hons)
Associate Professor
Department of Computer Science,
L.N.C.T., Bhopal

B.L. RAI
MCA, M.Tech(CSE)
Head of The Department
Department of Computer Science,
J.N.C.T., Bhopal

UNIVERSITY SCIENCE PRESS
(An Imprint of Laxmi Publications Pvt. Ltd.)
BANGALORE • CHENNAI • COCHIN • GUWAHATI • HYDERABAD
JALANDHAR • KOLKATA • LUCKNOW • MUMBAI • PATNA
RANCHI • NEW DELHI

Published by :

UNIVERSITY SCIENCE PRESS

(*An Imprint of Laxmi Publications Pvt. Ltd.*)
113, Golden House, Daryaganj,
New Delhi-110002

Phone : 011-43 53 25 00
Fax : 011-43 53 25 28

www.laxmipublications.com
info@laxmipublications.com

First Edition : 2011

OFFICES

✆ **Bangalore**	080-26 75 69 30	✆ **Chennai**	044-24 34 47 26
✆ **Cochin**	0484-237 70 04, 405 13 03	✆ **Guwahati**	0361-251 36 69, 251 38 81
✆ **Hyderabad**	040-24 65 23 33	✆ **Jalandhar**	0181-222 12 72
✆ **Kolkata**	033-22 27 43 84	✆ **Lucknow**	0522-220 99 16
✆ **Mumbai**	022-24 91 54 15, 24 92 78 69	✆ **Patna**	0612-230 00 97
✆ **Ranchi**	0651-221 47 64, 220 44 64		

UUS-9626-145-UNIX AND SHELL PROG-CHA **C —**
Typeset at : Kalyani Computer Services, New Delhi.

Dedicated to
Lord Ganesha

CONTENTS

PREFACE

The UNIX system was first described in a 1974 paper in the Communication of the ACM by Ken Thompson and Dennis Ritchie. Since that time, it has become increasingly widespread and popular throughout the computer industry where more and more vendors are offering support for it on their machines. It is especially popular in universities where its frequently used for operating systems research and case studies.

This book describes the internal algorithm and structures that form the basis of the operating system and their relationship to the programmer interface. This book also cover the topic Shell Programming, AWK, SED and PERL. It is thus applicable to several environments.

First it can be used as a text book for an operating system course. Second, system programmers can use the book as a reference to gain better understanding of how the kernel works and to compare algorithms used in the UNIX system to algorithms used in other operating systems. Finally Application Programmer can use the book to implement **Shell Script**, Programming with **awk,** Programming with **perl** and Programming with **sed.**

Many exercise originally prepared for the course have been included at the end of the book, and they are a key part of the book. Some exercises are straightforward, designed to illustrate concepts brought out in the text.

The book not only covers the entire scope of the subject but explain the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

—Authors

ACKNOWLEDGEMENT

It is my pleasure to acknowledge the assistance of many friends and colleagues who encouraged me while I wrote this book and provided constructive criticism of the manuscript. My deepest appreciation goes to Mr. Ashok Rai (Professor in LNCT, Bhopal) and Dr. Priyanka Tripathi (Professor in NIT Raipur), who suggested that I write this book, gave me early encouragement. Ashok Rai taught me many tricks of the trade, and I will always be indebted to him. Manish Khemaria and Prashant Baredar also had a hand in encouraging me from the very beginning, and we will always appreciate his kindness and thoughtfulness. We would like to thank my management for their continued support throughout this project and my colleagues, for providing such a stimulating atmosphere and wonderful work environment at LNCT Laboratories.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

—Authors

Chapter 1

INTRODUCTION TO UNIX

1.1 DEVELOPMENT OF UNIX

UNICS (Uniplexed Information and Computing System) 1970. And then UNIX—1973 in 'C'.

1.1.1 "Bill Joy" (AT&T Bell Laboratory) is a Student Who Wrote *Vi* Editor

Microsoft was the first to run UNIX on a PC with 640 KB of memory. They called their product XENIX that was bared on earlier edition of AT&T. But with some BSD borrowed utilities. XENIX was later sold off to SCO (The Santa, cruz operation) who today markets the most popular commercial brand of UNIX for the desktop—SCO UNIX. It now offers two major flavors—SCO open server release 5 and SCO UNIX ware 7: the later is SVR 4—compliant,

1.1.2 Structure of UNIX System

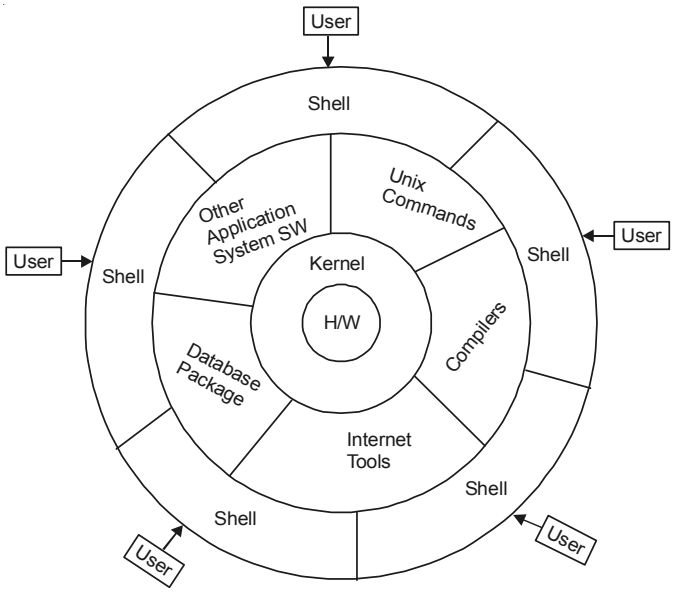


Fig. 1.1

Kernel → This is the name of UNIX Operating System.

Shell → This is the command interpreter used in unix.

Application programs and utilities—Utilities are UNIX commands. Application programs such as word processor, spreadsheets and Database Management Systems may be installed along side the linux (or UNIX) commands. A user may run a utility or Application Programs through shell.

User

(*i*) Ordinary user → Work only their own working directory.

(*ii*) Super user → Super user have the command on entire system. They can add user. Remove user etc.

1.2 TYPES OF SHELL

(*i*) **Bourne Shell**—It was created by Steve Bourne probably that's why its bounded with every UNIX system. Because it was bounded with every system it became popular. Whatever the cause and the effect, the fact remains that this is the shell used by many UNIX users.

(*ii*) **C Shell**—It was created by "Bill Joy" then pursuing his graduation at the University of California at Berkeley. This shell is a hit with those who are seriously into UNIX programming. It has two advantage over Bourne Shell.

(*a*) It allows aliaring of commands.

(*b*) C Shell has a command history feature.

(*iii*) **Korn Shell**—It was designed by "David Korn" of AT&T Bell Labs. The not so widely used Korn shell is very powerful and is a superset of Bourne shell. It offers a lot more capabilities and is decidedly more efficient than the other.

(*iv*) **Available to Linux** r-eh (Restricted Shell)—This restricts the area of memory the user may access to his or her own directory, thus limiting access to all other users files. Its not available on many machines.

There are other shells that have been developed since most generally for Linux—ash, tcsh, and zsh.

However the most widely used, Linux based shell is the Bourne again shell (bash) based on the original Bourne shell, it has similar extensions as the Korn shell, plus its own further extension.

Linux also offers a windows based shell interfaces commonly known as X-windows or simply as X.

Kernel is represented by the file Island/Unix or/Unix in SCO Unix, /vm linuz (Linux) and the shell by/bin/sh (/bin/bash in linux).

Note: The command prompt is a $ if you are operating in "Bourne Shell" or a % if in "C Shell".

1.2.1 The Reasons for Popularity and Success of the UNIX System

(*i*) The system is written in high level language, making it easy to read, understand change and move to other machines.

(*ii*) It has a simple user interface that has the power to provide the services that users want.

(*iii*) It provides primitives that permits complex programs to be built from simpler programs.

(*iv*) It user a hierarchical file system that allows easy maintenance and efficient implementation.

(*v*) It uses consistent format for files, the byte stream, making application programs easier to write.

(*vi*) It provides a simple, consistent interface to peripheral devices.

(*vii*) Its a multiuser, multiprocess system. Each user can execute several process simultaneously.

(*viii*) It hides the *m/c* architecture from the user, making it easier to write programs that run on different *h/w* implementations.

1.3 FEATURES OF UNIX

There are ten features of UNIX:

1. Multiuser system
2. Multitasking capabilities
3. Communication
4. Security
5. Portability
6. Pattern matching
7. Programming facility
8. Windowing system
9. Documentation
10. System calls and libraries.

1. Multiuser System

In a multiuser system, the same computer resources–hard disk, memory etc., are accessible to many users. The users don't flock together at the same computer but are given different terminal to operate from. All terminals are connected to the main computer whose resources are available by all users. The following figure shows a typical UNIX setup.

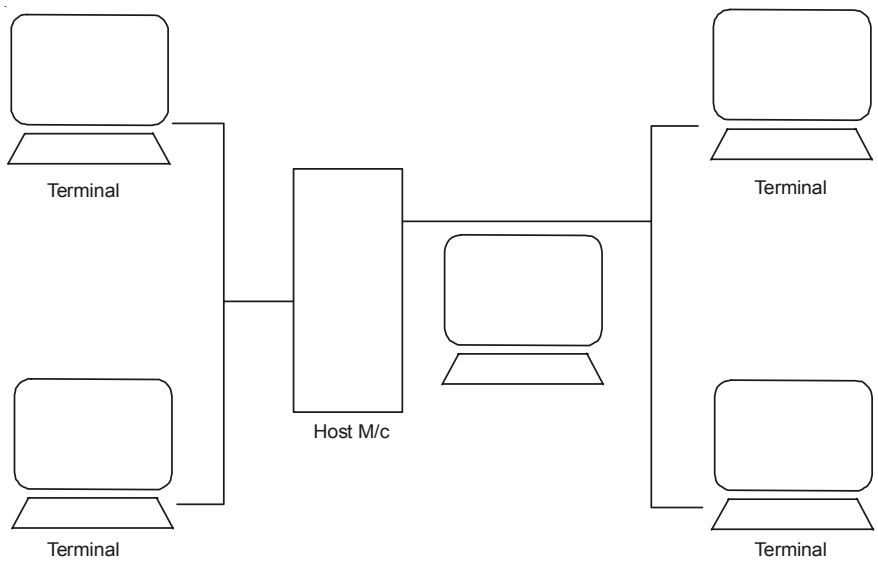


Fig. 1.2

Host M/C also known as server or a console. The number of terminal connected to the Host M/C depends on the number of ports that are present in its controller card. There are several type of terminal that can be attached to host M/C.

(*i*) **Dumb Terminal**—These terminal consist of a keyboard and a display unit with no memory or disk of its own. These can never act as independent machine.

(*ii*) **Terminal Emulation**—A PC has its own microprocessor, memory and disk driver. By attaching this to host M/C through a cable and running a S/W from this PC. We can emulate it to work as if it is a dumb terminal. At such times the memory and disk are not in use and the PC can't carry out any processing its own. The S/W that makes the PC work like a dumb terminal is called terminal emulation S/W. VTERM and XTALK are two such popular S/W.

(*iii*) **Dial-In-Terminal**—These terminal used telephones lines to connect with host M/C. To communicate over telephone lines we attach a modem.

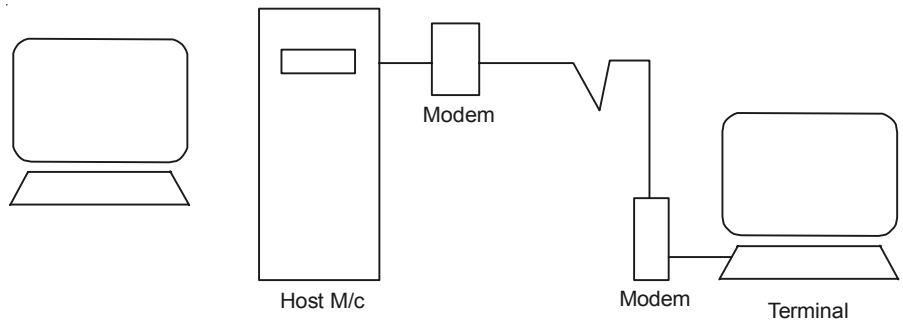


Fig. 1.3

2. Multitasking Capabilities

Its capable of carrying out that a single user can run more than one job at the same time. In UNIX this is done by running one job normally and other job in background. This is managed by dividing the CPU time b/w all processes.

"The multitasking provide by MS-DOS is known as serial multitasking".

3. Communication

The communication may be within the n/w of a single main computer or between two or more such computer n/w. The users can easily exchange mail, data, programs through such n/w. Distance poses no barrier to passing information or message to and from.

4. Security

UNIX has three inherent provision for protecting data.

(*i*) By assigning passwords and login names to individual users ensuring that not anybody can come and have access to your work.

(*ii*) At the file level there are read, write, and execute permissions to each file decide who can access a particular file.

(*iii*) File encryption.

5. Portability

It can be ported to almost any computer system with only the bare minimum of adaptations to suit the given computer architecture.

1.4 HIERARCHICAL STRUCTURE OF UNIX O.S. OR TREE STRUCTURE OF UNIX O.S.

(*i*) / - This is the rout directory of file system. The main memory of entire file system and HOME directory for super user.

(*ii*) /sbin - This contains programs used in boating the system and in recovery.

(*iii*) /bin - This is used to hold useful utilities programs.

(*iv*) /dev - This contains special device files that includes terminals, printers and storage device. There files contains device number that identify device to the O.S.

(*v*) /etc - This contains system administrators and configuration database.

(*vi*) /HOME - This contains home directory and files of all users, if your logname is "Anoop" your default HOME directory is/home/Anoop.

(*vii*) /tmp - This contains all temporary files used by UNIX system.

(*viii*) /var - This contains the directory of all files that vary among system. Files in this directory includes.

/var/adm — Contains system logging and accounting files.

/var/mail — Contains user mail files.

/var/news — Contains message of common interest.

/var/temp — Is a directory for temporary files.

/var/UUCP — Contains log and status files for UUCP system.

(*ix*) /mnt - Contains entries for removable (mountable) media such as CD-ROM, DLT taps.

(*x*) /proc - Contains process used in system.

(*xi*) /usr - This contains other accessible directory such as /usr/bin/usr/lib.

/usr/bin — Contains many executable program and unix system utilities.

/usr/sbin — Contain executable program for S.A.

/usr/games — Contains Binaries for games programmes and data for games.

/usr/lib — Libraries for programs and programming language.

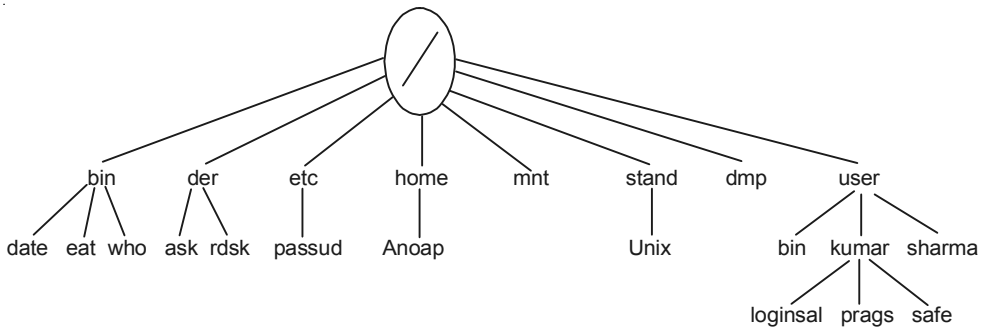


Fig. 1.4

In UNIX everything is treated as file. Its still necessary to divide there files into three categories:

(*i*) **Ordinary files**—Contains only data. This includes all data, source programs, object and executable code all UNIX commands as well as any files created by the user. The most common type or ordinary file is the text file.

(*ii*) **Directory files**—A directory contains no external data but keeps some details of the files and sub-directories that it contains. A directory file contains two field for each file. The **name of the file** and its identification number.

(*iii*) **Device files**—Consider physical device as files. This definition includes printers, tapes, floppy driver, CD-ROMs, hard disk and terminal.

(*a*) *Full Path Name*

List each directory, starting from/, down to the file itself. Each directory and the file name must be separated by a /.

(*b*) *Partial Path Name*

If the file is in or in a sub, directory below the working directory, the names of higher directories may be omitted.

(*c*) *Relative Path Name*

If the file is in a directory near the working directory, a relative path may be used.

If the working directory is /home/Ion, the file *al.c* in that directory may be reffered by:

Full Path Name — /home/Ion/al.c

Partial Path Name — al.c

If the working directory is /home/Ion, the file **temp** under directory /home/nic may be reffered by:

Full Path Name : /home/nic/temp

Relative Path Name : ../nic/temp

Chapter 2

OPERATING SYSTEM SERVICES

The Kernel performs various primitive operations on behalf of user process to support the user interface. Among the services provided by the Kernel are:

(*i*) Controlling the execution of process by allowing their creation, termination or suspension and communication.

(*ii*) Scheduling process fairly for execution on the CPU. Processes share the CPU in a time shared manner. The CPU executes a process, the Kernel suspend it when its time quantum elapses and the Kernel schedules another process to execute. The Kernel later reschedules the suspended process.

(*iii*) Allocating main memory for an executing process.

(*iv*) Allocating secondary memory for efficient storage and retrieval of user data. This service constitutes the file system.

(*v*) Allowing processes controlled access to peripheral devices such as terminals, tape driver, disk driver and N/W devices.

2.1 ASSUMPTION ABOUT H/W—LEVEL OF UNIX SYSTEM

The execution of user processes on UNIX systems is divided into two levels:

(*i*) User and (*ii*) Kernel level

When a process executes a system call the execution mode of the process changes from user mode to Kernel mode. The O.S. executer and attempts to service. The user request, returning an error code if it fails. Difference b/w, there modes are:

Kernel Mode	K			K
User Mode		U	U	

2.1.1 Differences

(*i*) Process in *user mode* can access their own instructions and data but not Kernel instructions and data. Process in *Kernel mode* however can access Kernel and user addresses. For example, the virtual address space of a process may be divided b/w

addresses that are accessible only in Kernel mode and addresses that are accessible in either mode.

(*ii*) Some *m/c* instructions are privileged and result in an error when executed in user mode. For example a *m/c* may contain an instruction that manipulates the processor status register. Process executed in user mode should not have this capabilities.

Although the system executes in one of two modes, the Kernel runs on behalf of a user process. The Kernel is not a seperate set of processes that run in parallel to user-process but its part of each user process.

2.1.2 Interrupts and Exceptions

The UNIX system allows devices such as I/O peripherals or the system clock to interrupt the CPU asynchronously. On, receipt of interrupt, the Kernel saver its current context determines the cause of the interrupt and services the interrupt. After the Kernel services the interrupt it restore its interrupted context and proceeds as if nothing had happened. The h/w usually prioritizes devices according to the order that interrupts should be handled. When the Kernel services an interrupt, it blocks out lower priority interrupts but services higher priority interrupt.

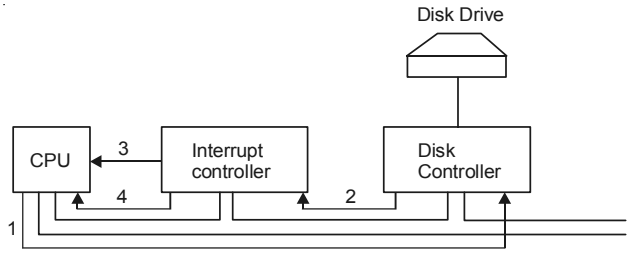


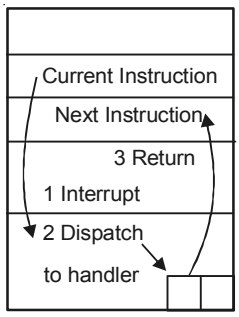
Fig. 2.1

Steps in starting an I/O devices and getting interrupt.

In step 1. The driver tells the controller what to do by writing into its device Register. The controller then starts the device. When the controller has finished reading or writing the number of bytes it has been told to transfer, it signals the interrupt controller chip using certain bus lines in **step 2.**

If the interrupt controller is prepare to accept the interrupt it assert a pin on CPU chip informing it, in **step 3.**

In step 4. The interrupt controller puts the number of the device on the bus so the CPU can read it and know which device has just finished.



Interrupt processing involves taking the interrupt, running, the interrupt handler, and returning to the user program.

Exception

An exception condition refers to unexpected events caused by a process such as addressing illegal memory, executing privileged instructions, dividing by zero and soon. OR "Exception are run time error".

Difference

Exception	*Interrupt*
1. Internal to process	1. External to process
2. Occur within a program	2. Occur between two programs
3. System attempt to restart the instruction after handling the exceptions.	3. The system continues with the next instruction after servicing the interrupt.

2.2 PROCESSOR EXECUTION LEVEL

The Kernel must sometimes prevent the occurence of interrupts during critical activity, which could result in corrupt data if interrupts were allowed. For instance the Kernel may not want to receive a disk interrupt while manipulating linked lists because handling the interrupt could corrupt the pointers. Setting the processor execution level to certain values masks off interrupts from that level and lower levels, allowing only higher level interrupts.

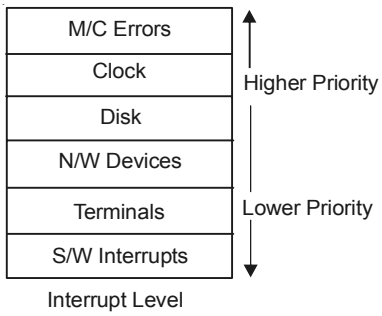


Fig. 2.2

Memory Management

The Kernel permanently resides in main memory as does the currently executing process.

2.3 ARCHITECTURE OF UNIX O.S.

2.3.1 System Calls and Libraries

System calls and libraries represent the border between user program and the Kernel. "Instruct the Kernel to do some specific task".

System calls look like ordinary function calls in C programs and libraries map these function calls to the primitives needed to enter the O.S. Assembly language programs may invoke system called directly without a system call library. The libraries are linked with the programs at compile time and are thus part of the user program for purpose of this discussion.

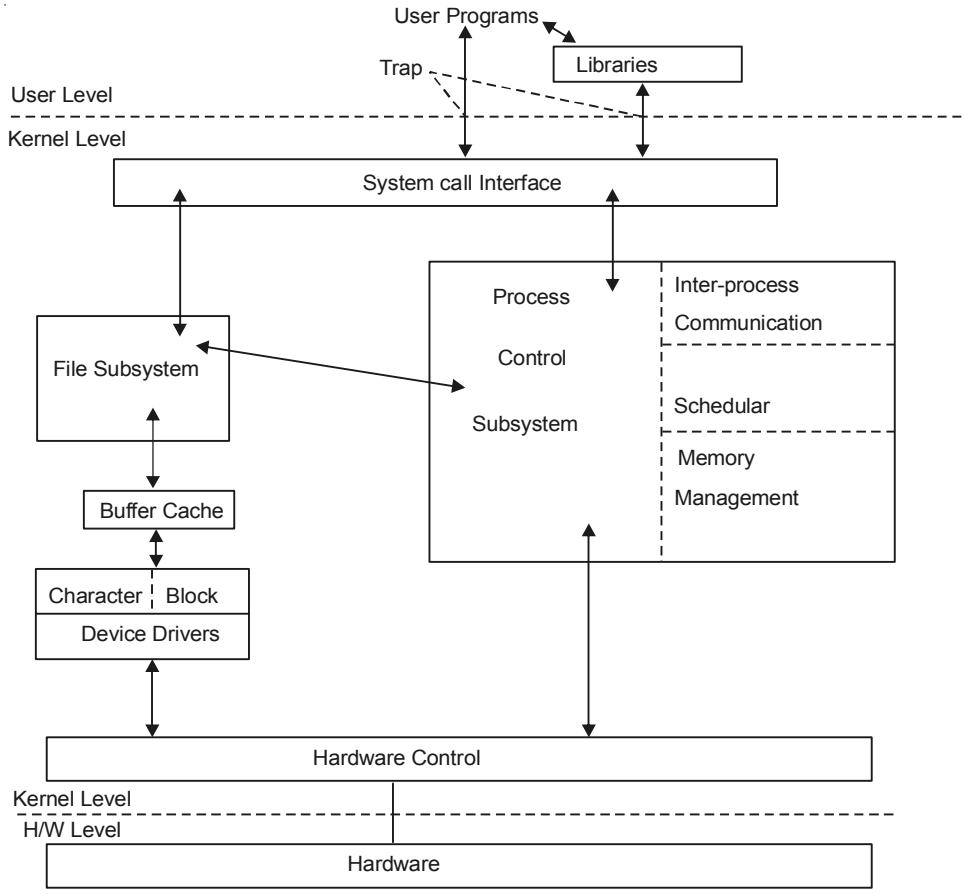


Fig. 2.3 Block diagram of system kernel

Trap

A trap (or exception) is a S.W. generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an 0.5 service be performed.

File Subsystem

The file subsystem manages files, allocating file space administering free space controlling access to files and retrieving data for users. Process interact with the file subsystem via a specific set of system calls such as open, read, write, shown, stat, chmod etc.

2.3.2 Buffer

The file subsystem accesses file data using a buffering mechanism that regulates data flow b/w the Kernel and secondary storage devices. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from and Kernel Block I/O device are random access storage device to the rest of the system. It also interacts directly with raw I/O devices (character device that are not block device) without the intervention of buffering mechanism.

2.3.3 Device Drivers

Device drivers are the Kernel modules that control the operation of peripheral devices Block I/O device drivers make them appear to be random access storage device to the rest of system. A tape driver may allow the Kernel to treat a tape unit as a random access storage device.

2.3.4 Process Control Subsystem

The file subsystem and process control subsystem interact when loading a file into memory for execution. Its responsible for:

(*i*) **Memory Management**—This module control the allocation of memory. If at any time the system does not have enough physical memory for all process, the Kernel moves them b/w main memory and secondary memory so that all process get a fair choice to execute two policies for managing memory swapping and demand paging.

(*ii*) **Schedular**—This modules allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting are source or until the Kernel prompts them when their recent run time exceeds a time quantum.

(*iii*) **Interprocess Communication**—This mechanism allow arbitrary-processes to exchange of data and synchronize execution. There are several form of TPC, pipes, signals, messages semaphore.

2.3.5 H/W Control

This is responsible for handling interrupts and for communicating with *m/c*. Devices such as disks or terminals may interrupt the CPU while a process is executing.

Chapter 3

FILE SYSTEM

After the disk has been partitioned its still not ready for use. A file system has to be created in each partition. There usually are multiple file systems in one *m/c* each one having its own directory tree headed by root.

Every file system is organized in a sequence of blocks of 512 bytes each. (1024 in Linux) and will have there four components:

Boot Block	Super Block	Inode Blocks		Data Block

3.1 FILE SYSTEM LAYOUT

1. **Boot Block**—This block contains a small boot strap program. This is the Master Boot Record (MBR) that DOS users would like to call it. This is loaded into memory when the system is booted. It may in turn load another program from the disk but eventually it will load the Kernel (the file/stand/unix or/vm linuz) into memory.
2. **Super Block**—It describe the state of file system (balance sheet of every UNIX file system). The Kernel first reads this area before allocating disk blocks and inodes for new files. The super block contains global file information about disk usage and availability of data blocks and inodes. It contains:

   —The size of the file system.

   —The length of a disk block.

   —Last time of updation.

   —The number of free data blocks available.

   —A partial list of immediately allocable free data blocks.

   —Number of free inodes available.

   —A partial list of immediately usable inodes.

   When a file is created it looks up the list available in the superblock. The Kernel reads and writes the copy of the super block in memory when controlling allocation of inode and data blocks.

3. **Inodes**—Since the block of a file are scattered throughout the disk, its obvious that the addresses of all its blocks have to be stored and not just the starting and ending ones. These address are available in the term of link list in the inode a table maintained individually for each file. All inode are stored in inode blocks distinctly separate from the data block and are arranged contiguously in a user in accessible area of file system.

3.1.1 Contents of an Inode

Every file or directory has an inode, which contains all that you could possibly need to know about a file except its name and contents. Each inode contain following attributes of a file:

—File type (regular, directory etc)

—Number of links (the number of alias the file has)

—Owner (The user-id number of the owner)

—Group (The group-id number)

—File Mode (The traid of the three permission)

—Number of bytes in the file

—Date and time of last modification

—Date and time of last access

—Date and time of last change of inode

—An array of 13 pointers to the file.

UNIX system also maintain a inode-table in memory for a file which is being used by it.

4. **Data Blocks**—The data block start at the end of the inode list and contain file data and administrative data. An allocated data can belong to one and only one file in the file system.

3.1.2 Logical and Physical Blocks

(Hard disk, floppy and tapes handle data in chunka or blocks).

$1024 * 10 + (256) * 1024 + (256 * 256) * 1024 + (256 * 256 + 256) * 1024$

$\approx$ 17 GB

So the size of a file in UNIX system is approximately equivalent to 17 GB.

3.2 BLOCK ADDRESSING SCHEME

There are 13 entries (or addresses) in the inode table containing the addresses of 13 disk block.

The first 10 addresses are simple. They contain the disk addresses of the first 10 blocks of file when files exceeds 10 blocks. We use single indirect (11th block) then double indirect (12th block) and then tripple indirect (13th block). Thus the maximum size a UNIX file system can support becomes 17 GB.

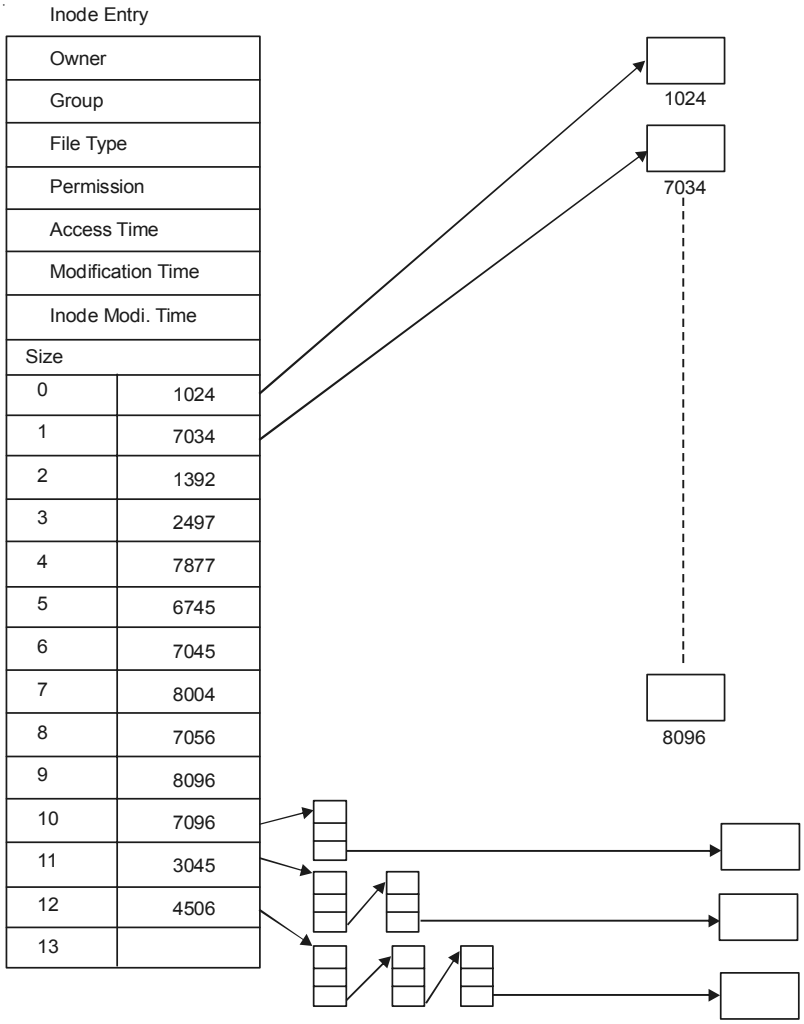


Fig. 3.1 Inode structure

The internal representation of a file is given by an inode which contains a description of the disk layout.

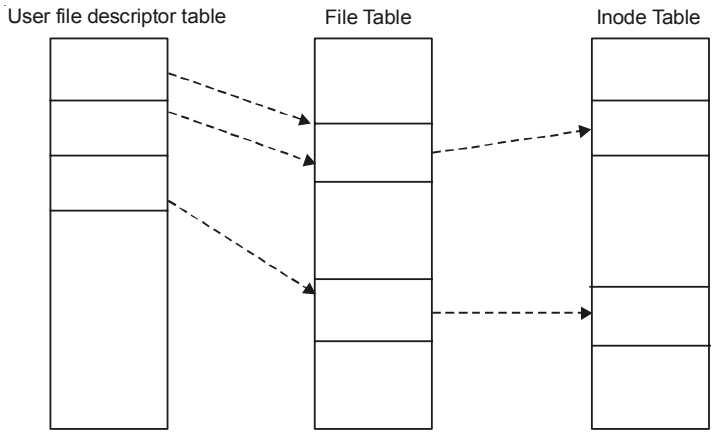


Fig. 3.2 File descriptor, file table, inode table

3.2.1 File Table

The file table is a global Kernel structure.

User File Descriptor Table

Its allocated per process and identifies all open file for a process.

When a process open or creates a file, the Kernel allocates an entry from each table, corresponding to the file inode. Entries in these three structure maintain the state of the file and the user access to it. The file table keeps track of the byte offset in the file where the users next read or write will start and the access rights allowed to the opening process.

For example, if a process calls.

```
"open ("/dev/anoop/fl.C",  1);
```

The Kernel retrives the inode for "/devl/anoop/fl.C". Then file table contains the entries for this file. And user file descriptor table contains a number which is a integer type call file descriptor.

The Kernel returns a file descriptor for open (), and create system call which is an index into the user file descriptor table. And then this file descriptor is used by the system call such as read(), write () etc.

3.3 PROCESS

A process is a program in execution and consists of a pattern of bytes that the CPU interprets as *m/c* instruction (called "text"), "data" and "stack". Many process appears to execute simultaneously as the Kernel schedules them for execution and several processes may be instance of one program. A process executes by following strict sequence of instruction that is self contained and does not jump to another process, it reads and writes its data and stack sections, but it can't read the stack and data sections of other process. Processes communicate with other processes via system calls.

A process on UNIX system is the entity that is created by "fork" system calls. Every processes except process O (sched) is created when other processes invokes the "fork" system call. The process that invokes the "fork" system call is called parent process and newly created process is called child process. The Kernel identifies each process by its process number called PID.

A user compile the source code of a program to create an executable file which consist of several parts:

(*i*) A set of "headers" that describes the attributes of the file.

(*ii*) The program text.

(*iii*) A *m/c* language representation of data that has initial values when the program starts execution and an indication of how much space the Karnel should allocate for uninitialized data, called brs[2] (the Kernel initilize it to O at run time).

(*iv*) Other section such as symbol table information. bss (block started by symbol) comes from an assembly *pseudo* operator on IBM 7090 *m/c*.

The Kernel loads an executable file into memory during an exec. system call and the loaded process consist of atleast three parts called regions; text, data and stack. The text and data regions

correspond to the text and data brs sections of the executable file but the stack region is automatically created and its size is dynamically adjusted by the Kernel at run-time.

The stack consists of logical stack frames that are pushed when calling a function and poped when returning; a special register called the stack pointer indicates the current stack depth.

We know that in UNIX system a process can execute either in (*i*) User or (*ii*) Kernel modes. It uses a separate stack for each mode.

The user stack contains the arguments, local variables and other data for functions executing in user mode.

The Kernel stack contains the stack frames for functions executing in Kernel mode. The functions and data entries on the Kernel stack refer to functions and data in Kernel not the user program but its construction is the same as that of user stack.

The Kernel stack of a process is null when the process execute in user mode.

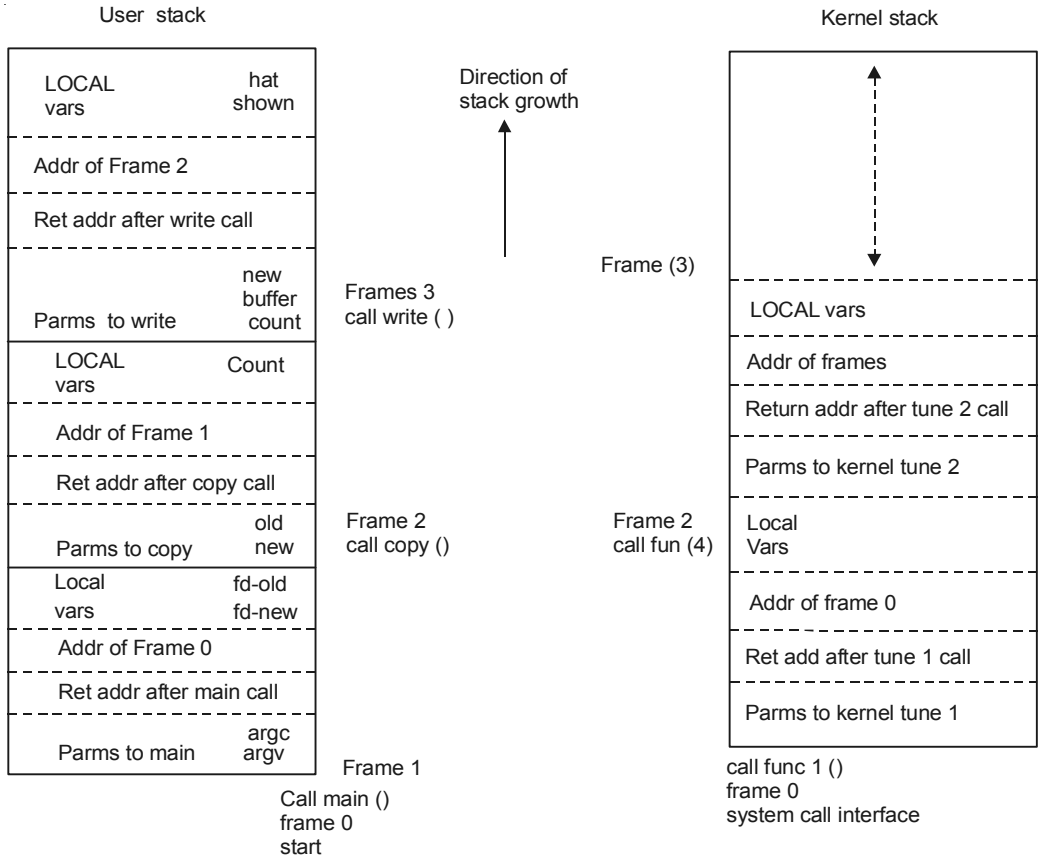


Fig. 3.3 User and kernel stack for copy program

3.3.1 Process Data Structure

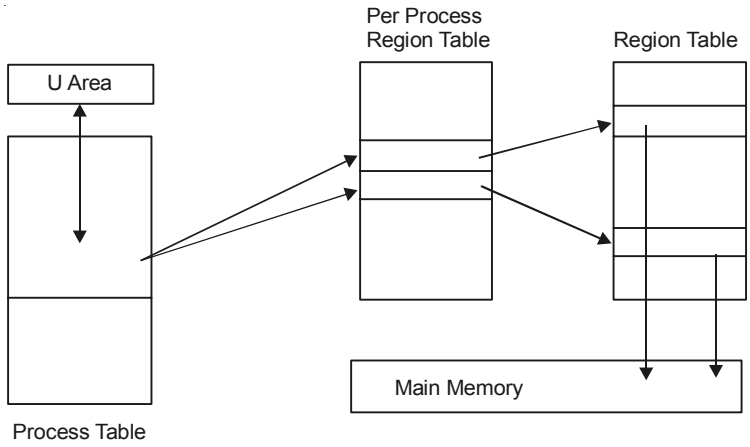


Fig. 3.4

U-Area (User-area)

U-area contains information describing the process that needs to be accessible only when the process is executing. The important fields are:

(*i*) A pointer to the process table stot of the currently executing process.

(*ii*) Parameters of the current system call return values and error codes.

(*iii*) File descriptors for all open files.

(*iv*) Current directory current root.

(*v*) Process and file size limit.

The Kernel can directly access fields of the u-area of executing process but not of the u-area of other-processes.

Process Table

The process table is resident all the time and contains information needed for all processes even those that not currently present in memory. The user structure is swapped or page-out when its associated process is not in memory. The process table entry and u-area contain control and status information about the process fields in the process tables are:

(*i*) A state field.

(*ii*) Identifiers indicating the user who owns the process (UID).

(*iii*) An event descriptor set when a process suspended.

(*iv*) Memory Image—Pointers to the text, data and stack segment or if paging is used to their page tables.

Region Table

Region (called segment in some early version and Berkeley releases) is a contiguous area of virtual address space that can be treated as a distinct object to be shared or protected. Region is a data area which may be accessed in mutual exclusive manner. In the region page table actual page descriptors are stored.

A process in UNIX terminology consist of three separate section, Text or Code, data and stack. Each occupies a contiguous area of virtual memory separate section belonging to a single task may be placed in non-contiguous areas of virtual memory. For example when a UNIX task creates a child process via tork() system calls the two share a single copy of the text region but the child obtains a fresh private copy of the parent's data regions.

The region table entries describes the attributes of the region, such as whether its contains text or data, whether its shared or private and where the "data" of the region is located in memory. When a process invokes the exec. system call the Kernel allocates region for its text, data and stack after freeing the old regions the process had been using when a process invokes tork () system call Kernel duplicate the address space of old process allowing process to share, regions when possible and making a physical copy otherwise when a process invokes exit () system call the Kernel trees the region *th* process had used.

Per Process Region Table

The entries of per process region table point to entries in a region table. In this way a shared region may be described by a single copy of the second level page, tables but by separate per-task first-level region pointers to it. For example a parent and a child process sharing a region need only maintain private first-level pointers to a single shared copy of the second level page table in which the page description are stored.

3.3.2 Context of a Process

The context of a process is its state as defined by its text, the values of its global user variables and data structures, the values of *m/c* registers it user. The values stored in its process table slot and *u* area and the contents of its user and Kernel stacks.

When executing a process, the system is said to be executing in the content of the process. When the Kernel decides that it should execute another process it does a context switch, so that the system executes in the context of the other process. The Kernel allows a context switch only under specific conditions. When doing a context switch, the Kernel saves enough information so that it can later switch back to the first process and resume its execution. Similarly when moving from user to Kernel mode, the Kernel saves enough information so that it can later return to user mode and continue execution from where it left off. Moving between user and Kernel mode is a change in mode not a context switch.

	A	B	C	D
Kernel	K			K
User		U	U	

In this figure, the Kernel does a context switch when it changes context from process A to process B.

3.4 PROCESS STATE AND TRANSITIONS

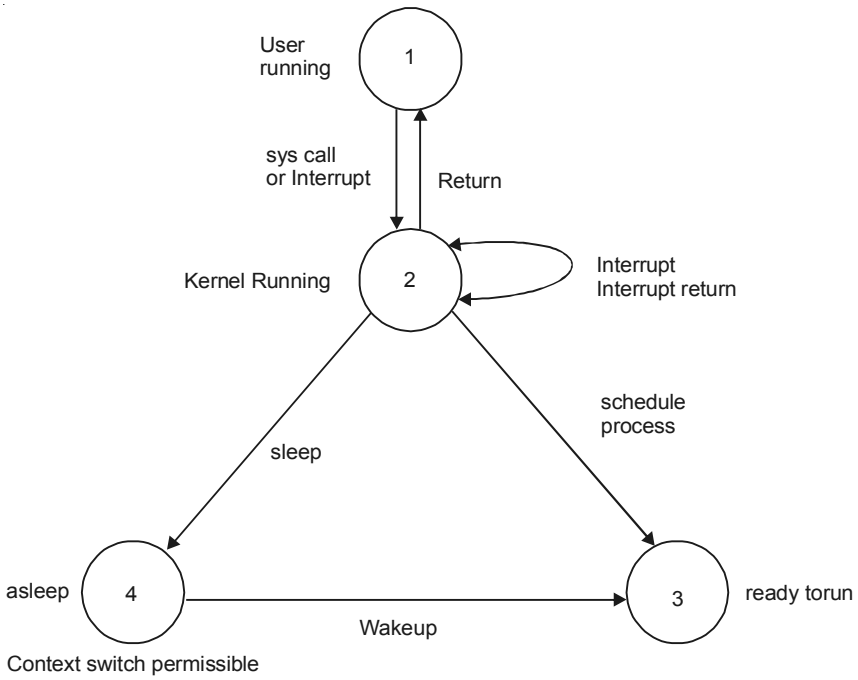
Some States are:

(*i*) The process is currently executing in user mode.

(*ii*) Processer currently executing in Kernel mode.

(*iii*) The process is not executing, but its ready to run as soon as the schedular chooses it. Many processes may be in this state and the scheduling algorithm determine which one will execute next.

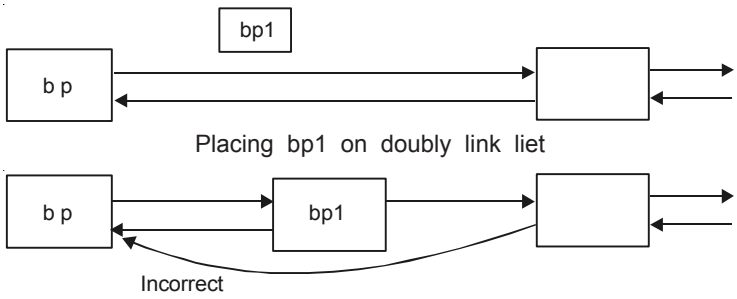
(*iv*) The processes is sleeping. A process puts itself to sleep when it can no longer continue executing such as when its waiting far I/O to complete.



The Kernel allows a context switch only when a process moves from the state "Kernel running" to the state "asleep in memory". Process running in Kernel mode cannot be preempted by other process; therefore the Kernel is sometimes said to be non-preemptive, although the system does preempt processes that are in user mode.

Consider the code:

```
Struct Queue { } * bP, * bP1
bP1 → torP = bP → tarp
bP1 → bakP = bP
bP → tarp = bP1
1 * context switch */
bP1 → torP → backP = bP*
```

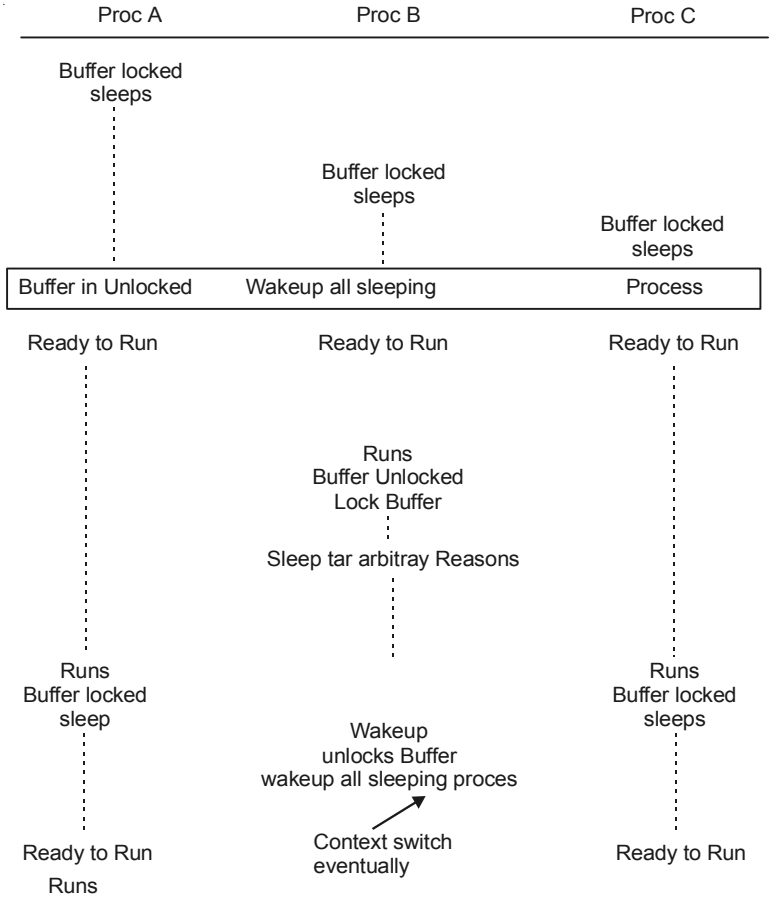


To solve this problem, the system could prevent all interrupts while executing in Kernel mode, but that would delay servicing of the interrupt. Instead the Kernel raises the processor execution level to prevent interrupts when entering critical regions of code. A section of code is critical if execution of arbitrary interrupts handlers could result in consistency problems.

The Kernel protects its consistency by allowing a content switch only when a process puts itself to sleep and by preventing one process from changing the state of another process. It also raises the processor execution level around critical regions of code to prevent interrupts that could otherwise cause inconsistency.

3.5 SLEEP AND WAKEUP

Processes go to sleep because they are awaiting the occurence of some event such as waiting for I/O completion from a peripheral device waiting for a process to exit, waiting for system resources to become available and soon. Processes are said to be sleep on an event meaning that they are in the sleep state until the event occurs, at which time they wakeup and enter the state ready to run. Many processes can simultaneously sleep on an event; when an event occurs, all processes sleeping on the event wakeup because the even condition no longer true. When a process wakeup, it follows the state transition from the "sleep" state to the "ready-to-run" state where its eligible for later scheduling; it does not execute immediately. Sleeping process do not consume CPU resources. The Kernel does not constantly check to see that a-process is still sleeping but waits for the event to occur and awakens the process then.



For example, a process executing in Kernel mode must sometimes lock a data structure in case it goes to sleep at a later stage, process attempting to manipulate the locked structure must check the lock and sleep if another process owns the lock. The Kernel implement such lock in following manner,

```
while (condition is true)
sleep (event: the condition becomes false); set condition true;
```

It unlocks the lock and awakens all processes sleep on the lock in following manner:

```
set condition true;
wakeup (event: The condition is false);
```

Multiple process Sleeping on a lock.

3.6 KERNEL DATA STRUCTURE

Most Kernel data structures occupy fixed sizes tables rather than dynamically allocated space. The advantage of this approach is that the Kernel code is simple but it limits the number of entries for a data structure to the number that was originally configured when generating the system. If during operation of system the Kernel should run out of entries for a data structure, it can not allocate space for new entries dynamically but must report an error to the requesting user. If the Kernel is configured so that its unlikely to run out of table space, the extra table space may be wasted because it can not be used for other purpose.

Example: *Since there are many different version of UNIX so showing Kernel structure is slightly tricky. But here we give the structure of 4.4 BSD.*

<table>
<tr><td colspan="5">System Calls</td><td colspan="2">Interrupts and Trap</td></tr>
<tr><td colspan="2">Terminal Handling</td><td>Sockets</td><td>File Naming</td><td>Map- Page ping fault</td><td>Signal Handling</td><td>Process creation and termination</td></tr>
<tr><td rowspan="2">Raw tty</td><td>Cooked tty</td><td>N/W protocols</td><td>Fill systems</td><td>Virtual memory</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>Line discipline</td><td></td><td>Buffer Cache</td><td>Page Cache</td><td colspan="2">Process Scheduling</td></tr>
<tr><td colspan="2">Character devices</td><td>N/W device drivers</td><td>Disk device drivers</td><td></td><td colspan="2">Process dispatching</td></tr>
<tr><td colspan="2"></td><td>Hardware</td><td></td><td></td><td colspan="2"></td></tr>
</table>

Structure of 4.4 BSD Kernel

3.7 SYSTEM ADMINISTRATION

Administrative process are those process that do various functions for the general welfare of the user community. Such functions include disk formatting creation of new file systems, repair of damaged file system. Kernel debugging and others. There is no difference between administrative process and user processes. They use the same set of system calls available to the general

community. They are distinguish from general user process only in the rights and privileges they allowed. The Kernel does not recognize a separate class of administrative process.

Following figure shows some of the common fields present in UNIX system.

Process Mge	*Memory Mge*	*File Mgt*
Register	Pointer to text segment	UMASK mark
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to *bss* (bare) segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective vid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real vid	
Message queue pointers	Real gid	
Pending signal bits	Effective vid	
Process id	Effective gid	
Various flage bits	Bit map for signals	
	Various flag bits	

Chapter 4

BUFFER CACHE

When a process want to access data from a file, the Kernel brings the data into main memory where the process can examine it alter it and request that the data be saved in the file system again. The super block of a file system describes the free space available on the file system. The Kernel reads the super block into memory to access its data and writes it back to the file system when it wishes to save its data. Similarly the inode describes the layout of file. The Kernel reads an inode into memory when it want to update the file layout.

The Kernel could read and write directly to and from the disk for all file system access, but system response time and throught put would be poor because of the slow disk transfer rate. The Kernel therefore minimize the frequency of disk access by keeping a pool of internal data buffers, called the buffer cache which contains the data inrecently used disk blocks.

The position of buffer cache module in Kernel architecture between the file subsystem and (block) device drivers.

When reading data from disk. The Kernal attempts to read the data from buffer cache. If the data is already in cache the Kernel does not have to read from disk. If the data is not in cache, the Kernel reads the data from disk and caches it using an algorithm that tries to save as much good data in the cache as possible. "*Similar for write*".

4.1 BUFFER HEADERS

During system initialization, the Kernel allocates space for a number of buffers configurable according to memory size and system performance constraints. A buffer consist of two parts.

(*i*) **A Memory Array**—It contains data from the disk.

(*ii*) **Buffer Header**—That identifies the buffer. The buffer header also contains a pointer to a data array for the buffer whose size must be atleast as big as the size of a disk block and a status field that summarizes the current status of the buffer.

The data in a buffer corrosponds to the data in a logical disk block on a file system and the Kernel identifies the buffer contents by examining identifier fields in the buffer header.

The buffer is the in-memory copy of the disk-block; the contents of the disk block map into the buffers but the mapping is temporary until the Kernel decides to map another disk block into

the buffer. A disk block can never map into more than one buffer at a time. If two buffers were contain data for one disk block, the Kernel would not know which buffer contained the current data and could write incorrect data back to disk. "A $\vec{\theta}$ B" $\rightarrow$ current.

The buffer Header contains a:

(*i*) **Device number**—Specific file system. Its the logical file system number, not a physical device (disk) unit number.

(*ii*) **Block number**—It specify the block number of the data on disk and uniquely identify the buffer.

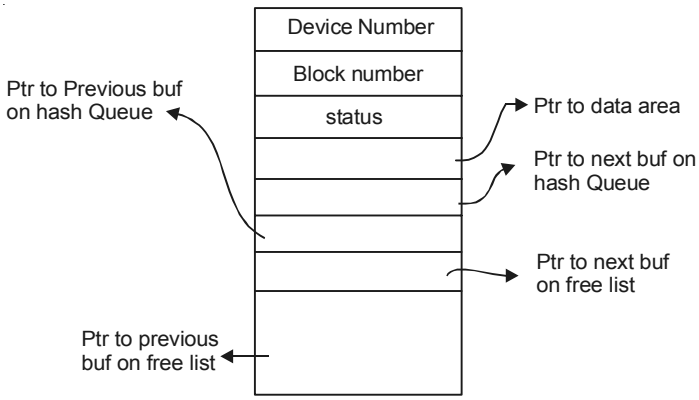


Fig. 4.1. Buffer Header

The status of a buffer is a combination of the following conditions:

(*i*) The buffer is currently locked.

(*ii*) The buffer contains valid data.

(*iii*) The Kernel must write the buffer contents to disk before reassigning the buffer. (delay write).

(*iv*) The Kernel is currently reading or writing the contents of the buffer to disk.

(*v*) A process is currently waiting for the buffer to become tree.

4.1.1 Structure of Buffer Pool

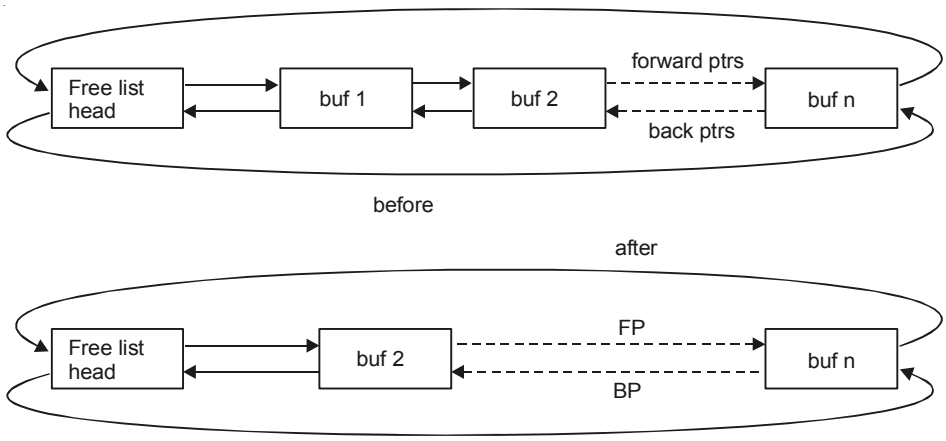


Fig. 4.2 Free list of buffers

The Kernel caches data in the buffer pool according to a LRU algorithm: after it allocates a buffer to a disk blocks, it cannot use the buffer for another block until all other buffers have been used more recently. The Kernel maintains a free list of buffers that preserves the least recently used order. The free list is a doubly linked circular list of buffers with a dummy buffer header that marks its begining and end when the Kernel returns a buffer to the buffer pool it usually attaches the buffer to the tail of the free list, occasionally to the head of the free list (for error case), but never to the middle.

When the Kernel access a disk block, it searches for a buffer with the appropriate device-block number combination. Rather than search the entire buffer pool, it organizes the buffers into separate queues hashed as a function of the device and block number the Kernel links the buffers on ahash queue into a circular, doubly linked list. The number of buffers on a hash queue varies during the lifetime of the system. The Kernel must use a hashing function that distributes the buffers uniformally accross the set of hash queues. Yet the hash function must be simple so that performance does not suffer. System administrators configure the number of hash queues when generating the O.S.

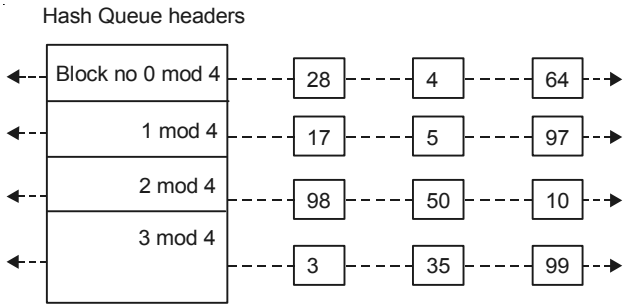


Fig. 4.3 Buffers on the hash queue

Each buffer always exists on a hash queue. But there is no significance to its position on the queue. No two buffers may simultaneously contain the contents of the same disk block therefore every disk block in the buffer pool exists on one and only one hash queue and only once in that queue. However, a buffer may be on the free list as well if its status is free. The Kernel has two ways to find a buffer on a hash queue. It searcher the hash queue, if its looking for a particular buffer and it remover a buffer from the free list if its looking for any free buffer.

4.2 SCENARIOS FOR RETRIEVAL OF A BUFFER

High level Kernel algorithms in the file subsystem invoke the algorithms for managing the buffer cache. The high level algorithm determine the logical device number and block number that they wish to access when they attempt to retrieve a block. The algorithms for reading and writing disk blocks use “getblk” algorithm to allocate buffers from the pool.

There are five scenarios the Kernel may follow in getblk to allocate a buffer for a disk block.

1. The Kernel finds the block on its hash queue and its buffer is free.

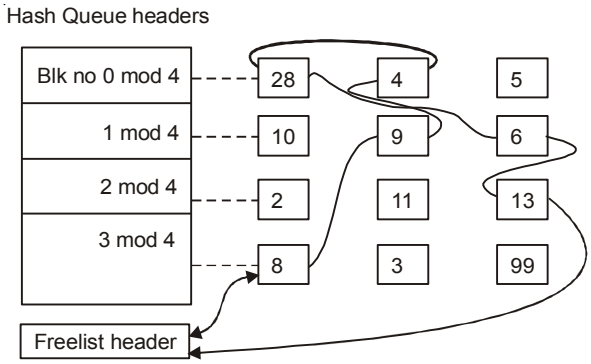


Fig. 4.4 Search for block 4 on first hash queue

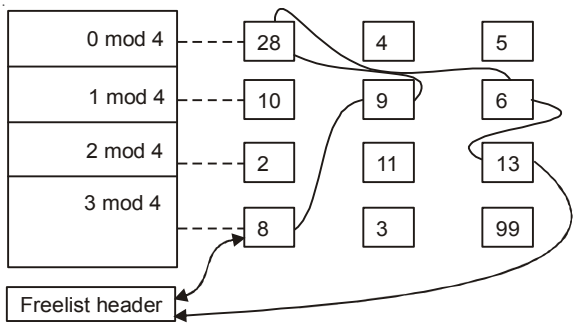


Fig. 4.5 Remove block 4 from free list

The Kernel may read data from the disk to the buffer and manipulate it and write data to the buffer and possibly to disk. The Kernel leaves the buffer marked busy. When the Kernel finishes using the buffer it release the buffer according to algorithm "brelese". It wakeup processes that had fallen asleep because the buffer was busy and process that had fallen asleep because no buffers remained on free list. The Kernel places the buffer at the end of the free list, unless an I/O error occured or unless it marked the buffer "old". This is now free for another process to claim it.

4.2.1 Algorithm for Releasing a Buffer

```
algorithm brelse:
input: locked buffer
output : none
{wakeup all process: event, waiting for any buffer to become free;
wakeup all proc: event, waiting for this buffer to become free; raise
processor execution level to block interrupts;
it (buffer contents valid and buffer not old)
enqueue buffer at end of free list lower processor execution level
to allow interrupts;
unlock (buffer);}
```

Algorithm for buffer allocation

```
algorithm getblk
input : file system number
output : locked buffer that can now be used for block
{while (buffer not found)
       { if (block in hash queue)
            {
              {if (buffer busy) / * 5 */
              }
              {sleep (event: buffer becomes free):
               continue;
            }
            mark buffer busy; / * 1  * /
            remove buffer from free list;
            return buffer;
       }
else / * block not in hash queue * /
} if (there are no buffer on free list) / * 4 * /
            { sleep(event; any buffer becomes free);
            continue;
            }
            remove buffer from free list:
            if (buffer marked for delay write) / * 3 * /
            } asynchronous write buffer to disk;
            continue;}
       / * 2 * /
            remove buffer from old hash queue;
            put buffer onto new hash queue;
            return buffer;
            }
         }
       }
```

2. Kernel cannot find the block on the hash. Queue, so it allocates a buffer from the free list.

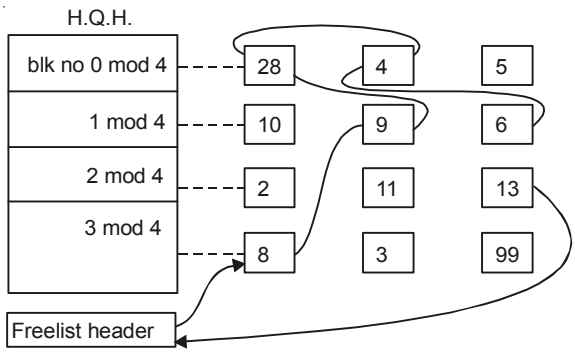


Fig. 4.6 Search for block 20–not in cache

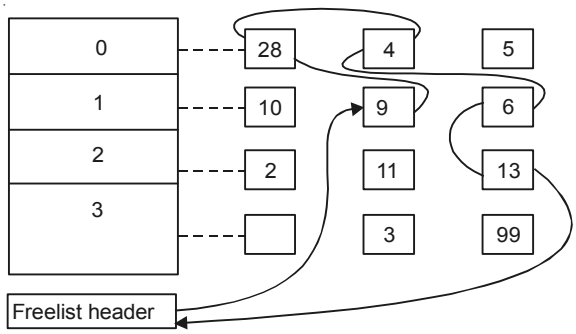


Fig. 4.7 Remove first block from free list assign to 20

3. The Kernel cannot find the block on the hash queue and in attempting to allocate a buffer from free list, finds a buffer on the free list that has been marked "delayed write". The Kernel must write the "delayed write" buffer to disk and allocate another buffer.

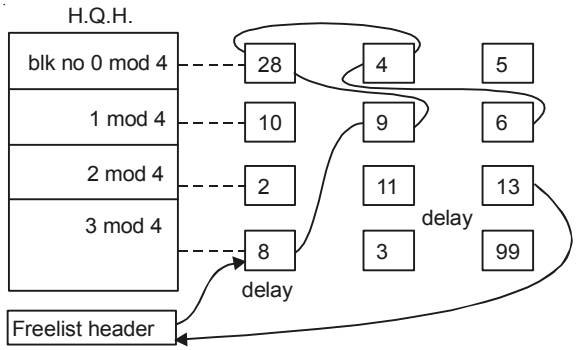


Fig. 4.8. Search for Block 20. Delayed write blocks on free list.

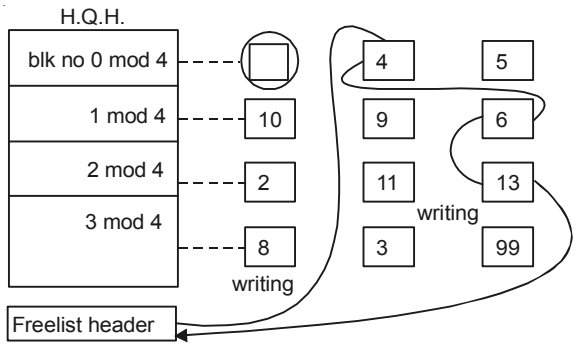


Fig. 4.9 Writing blocks 3, 5, Reassign 28 to 20

4. The Kernel cannot find the block on the hash queue and the free list of buffers is empty.

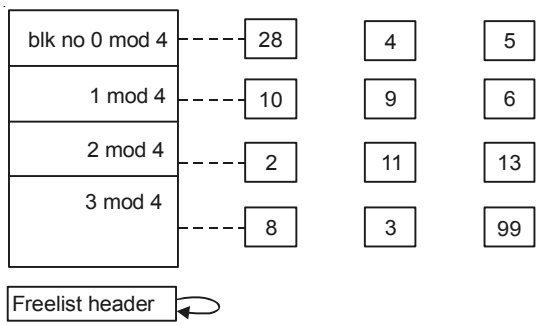


Fig. 4.10 Search for block 20 freelist empty

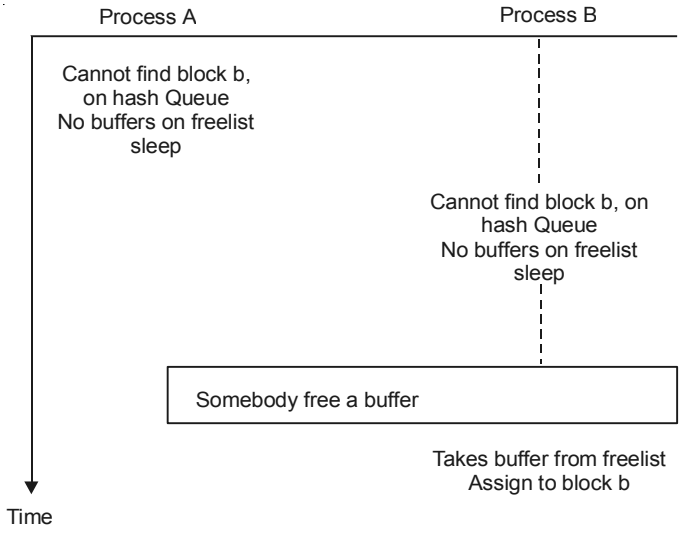


Fig. 4.11 Race for free buffer

5. Kernel finds the block on the hash queue but its buffer is currently busy.

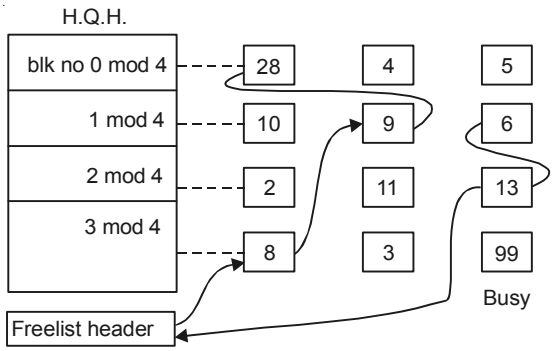


Fig. 4.12 Search for block 99. Block busy.

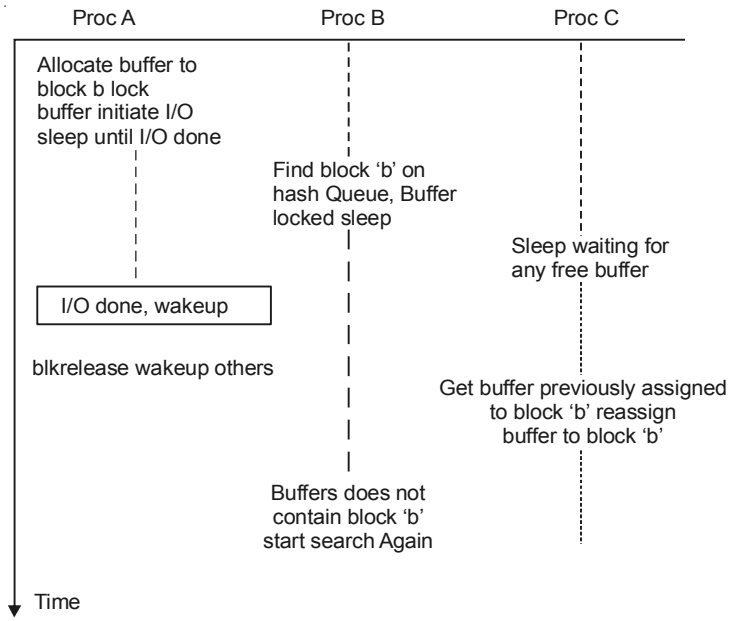


Fig. 4.13 Race for a locked buffer

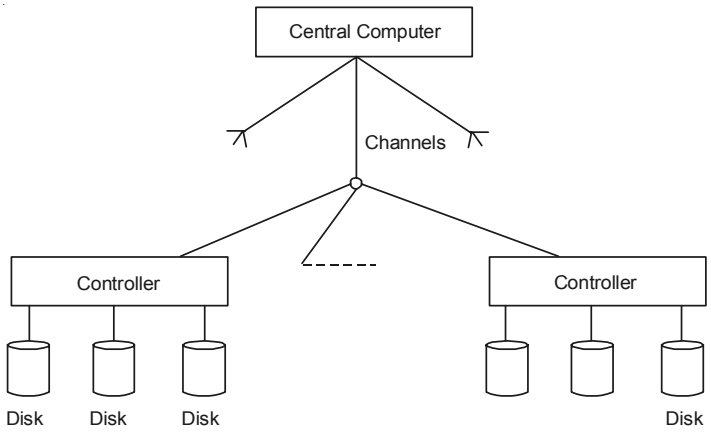
Chapter 5

READING AND WRITING DISK BLOCKS

To read a disk block a process uses an algorithm to get a Block search for it in the buffer cache. If its in the cache, the Kernel can return it immediately without physically reading the block from disk. If its not in cache, the Kernel calls the disk driver to schedule a request and goes to sleep awaiting the event that the I/O completes. The disk drivers notifies the *disk controller h/w* that it want to read data and the disk controller later transmits the data to buffer. Finally the disk controller interrupts the processor when the I/O is complete and the disk interrupt handler awakens the sleeping process: the contents of the disk block now in the buffer.

5.1 DISK CONTROLLER

Several physical disk units are managed by a single disk controller device. The controller in turn is connected to an I/O channel that transmits information from disk unit to the central computer one channel might support several disk. Controller each of which might, in turn, support several disk drives.



In this scheme I/O channels are not connected directly to the disk unit they service. This fact cause the designer to investigate a bottleneck, further deciding to incorporate disk scheduling.

If a controller becomes saturated the designer may wish to reducing the number of disk on that controller. Thus h/w reconfiguration needed to eliminate certain bottlenecks.

5.2 ALGORITHM: FOR READING A DISK BLOCK

```
algorithm : bread
     input : File system block number
   output  : buffer containing data
{
get buffer for block (by using some technique)
if (buffer data valid)
     return buffer:
initiate disk read;
sleep (event : disk read complete):
return buffer:
}
```

5.3 BLOCK READ AHEAD

Sometimes higher-level Kernel modules (such as file subsystem) may anticipate the need for a second disk block when a process reads a file sequentially. The modules request the second I/O asynchronously in the hope that the data will be in memory when needed improving performance. To do this the Kernel executes the block read-ahead algorithm. The Kernel checks if the first block is in the cache and it its not there invokes the disk driver to read that block. If the second blocks is not in the buffer cache, the Kernel instruct the disk driver to read it asynchronously. Then the process goes to sleep awaiting the event that the I/O is complete on the first block. When it awakens it returns the buffer for the first block and does not care when the I/O for the second block completes. When the I/O for second block does completes the disk controller interrupts the systems the interrupt handler recognizes that the I/O was asynchronous and release the buffer.

```
Algorithm : breada / * block read and read ahead * /
input :
  (1) File system block number for immediate read.
  (2) File system block number for asynchronous read;
output; Buffer containing data for immediate read;
{ if (first block not in cache)
{ get buffer for first block / * by some technique * / if (buffer
data not valid)
        initiate disk read;
}
```

```
if (second block not in cache)
{ get buffer for second block;
if (buffer data valid)
release buffer // by some technique
else
initiate disk read;
}
if (first block was originally in cache)
{ read first block by some technique
return buffer;
}
sleep (event: first buffer contains valid data);
return buffer;
}
```

Writing the contents of a buffer to a disk block is similar.

The Kernel informs the disk driver that it has a buffer whose contents should be output and the disk driver schedules the block for I/O. If the write is synchronous, the calling process goes to sleep awaiting I/O completion and releases the buffer when it awakens. If the write is asynchronous, the Kernel starts the disk write but does not wait for the write to complete. The Kernel will release the buffer when the I/O completes.

When the Kernel does not write data immediately to disk. If it does a "delayed write", it marks the buffer accordingly release the buffer and continues without scheduling I/O. The Kernel writes the block to disk before another process can reallocate the buffer to another block.

A "delayed write" is different from asynchronous write:

(*i*) When doing an asynchronous write the Kernel starts the disk operation immediately but does not wait for its completion.

(*ii*) For a "delayed-write" the Kernel puts off the physical write to disk as long as possible, then write the buffer 'old' and write back to disk asynchronously.

```
algorithm : bwrite / * Block write * /
input     : buffer
output    : none
{
    initiate disk write :
      if (I/O synchronous)
{
sleep (event : I/O complete);
                release buffer
}
else if (buffer marked for delayed write);
mark buffer to put head of free list;
}
```

5.4 ADVANTAGE OF DISK BLOCK

1. **System design is simpler.** The use of buffers allows uniform disk access because the Kernel does not need to know the reason for the I/O. Instead it copies data to and from buffers regardless of whether the data is part of a file, an inode or a super block. The buffering of disk I/O makes the code more modular, since the parts of the Kernel that do the I/O with the disk have one interface for all purposes.
2. The system places no data alignment restrictions on user processes doing I/O, because the Kernel aligns data internally. H/W implementations frequently require a particular alignment of data for disk I/O such as aligning the data on a two byte boundry or on a four-byte boundary in memory. Without a buffer mechanism, programmers would have to make sure that their data buffers were correctly aligned.

   Many programmers error would result and programs would not be portable to UNIX systems running on machines with stricter address alignment properties. By copying data from user to system (or vice versa) buffers, the Kernel eliminates the need for special alignment of user buffers, making user program simpler and more portable.
3. *Use of buffer cache can reduce the amount of disk traffic*, thereby increasing overall system through put and decreasing response time.

   Process reading from the file system may find data blocks in the cache and avoid the need for disk I/O. The Kernel frequently user "delayed write" to avoid unncessary disk writes, leaving the block in the buffer cache and hoping for a cache hit on the block.
4. The buffer algorithms help insure file system integrity because they maintain a common single image of disk blocks contained in the cache.

   If two processes simultaneously attempt to manipulate one disk block, the buffer algorithm serialize their access preventing data corruption.

5.4.1 Disadvantage of Buffer Cache

1. Since the Kernel does not immediately write data to the disk for a "delayed write" the system is vulnerable to crashes that leave disk data in an incorrect manner.

   Although recent system implementations have reduced the damage caused by catastrophic events, the basic problem remains. A user issuing a write system call is never sure when the data finally makes its way to disk.
2. Use of buffer cache requires an extra copy when reading and writing to and from user processes.

   When transmitting large amounts of data the extra copy shows down-performance.
3. Obviously, the chances of cache hit are greater for systems with many buffers. However, the number of buffers a system can profitably configure is constrained by the amount of memory that should be kept available for executing processes; if too much memory is used for buffers the system may slow down because of excessive process swaping or paging.

5.4.2 Internal Representation of Files

Two types of Inodes:

(*i*) **Disk Inodes:**

Owner: group: type: Perms: accessed: modified: inode: size: disk address: (1.3 Block)	Changing the contents of a file automatically implies a change to the inode, but changing the inode does not imply that the contents of the file change.

(*ii*) **Incore Copy of inode:** The active copy of a disk inode.

The incore copy of the inode contains following field:

(*i*) The status of incore inode; indicating whether

– The inode is locked.

– A process is waiting for the inode to become unlocked.

– The incore representation of inode differs from the disk copy as a result of a change to the data in the inode.

– The incore representation of the file differs from disk copy as a result of a change to the file data.

– The file is a mount point.

(*ii*) The logical device number of the file system that contains the file.

(*iii*) The inode number. Since inodes are stored in a linear array on disk, the Kernel identifies the number of a disk inode by its position in the array. The disk inode does not need this field.

(*iv*) *Pointers to other in-core inodes.* The Kernel links inodes on hash queues and on a free list. A hash queue is identified according to the inodes logical device number and inode number. The Kernel can contain atmost one in-core copy of a disk inode, but inodes can be simultaneously on a hash queue and on the free list.

(*v*) A reference count, indicating the number of instances of the file that are active.

5.5 ACCESSING INODES

The Kernel identifies particular inodes by their file system and inode number and allocates in-core inodes at the request of higher-level algorithms. The Kernel maps the device number and inode number into a hash queue and searches the queue for inode. If it cannot find the inode, it allocates one from the free list and locks it. The Kernel then prepares to read the disk copy of the newly accessed inode into the incore copy. We can compute:

block num = ((inode number – 1)/no. of inodes per block) + start block of inode list

Ex. start block = 2, inode no = 8, no. of inodes per block = 8

block num = ((8 – 1) / 8) + 2 = (7/8) + 2 = 0 + 2 = 2

When the Kernel knows the device and disk block number it reads the block then uses the following formula to compute the byte offset of the inode in the block.

= (inode no. – 1) modulo (no. of inodes per block) * size of diskinode.

If each disk inode occupies 64 bytes (size of disk inode) and there are 8 inodes to disk block, then inode no 8 starts at byte offset 448 in disk block.

= ((8 – 1) % 8) * 64 = (7 % 8) * 64 = 7 * 64 = 448

The Kernel removes the in-core inodes from free list, places it on the correct hash queue and sets its in-core reference count to 1. It copies the file type, owner fields permission settings, link count file size, and the table of contents from the disk inode to the in-core inode and returns a locked inode.

The Kernel manipulates the inode lock and reference count independently. The lock is set during execution of a system call to prevent other processes from accessing the inode while its in use. The Kernel release the lock at the conclusion of the system call an inode is never locked across system calls. The Kernel increment the reference count for every active reference to a file. It decrements the reference count only when the reference becomes inactive. The reference count thus remains set across multiple system calls. The lock is free between system calls to allow processes to share simultaneous access to a file; the reference count remains set between system calls to prevent the Kernel from real locating an active in-core inode. Thus the Kernel can lock and unlock an allocated inode independent of the value of reference count.

Processes have control over the allocation of inodes at user level via execution of open () and close () system calls and consequently the Kernel cannot guarantee when an inode become available. Therefore a process that goes to sleep waiting for a free inode to become available may never wakeup. Rather than leave such a process “hanging” the Kernel fails the system call.

5.5.1 Buffer

However process do not have such control over buffers. Because a process cannot keep a buffer locked across system calls, the Kernel can guarantee that a buffer will become free soon and a process therefore sleeps until one is available.

If the inode is in the cache, the process (A) would find it on its hash queue and check if the inode was currently locked by another process (B). Process (A) sleeps setting a flag in the in-core inode to indicate that its waiting for the inode to become free. When process (B) unlocks the inode, it awakens all processes waiting for the inode to become free when process a finally able to use the inodes it locks the inode so that other process cannot allocate it.

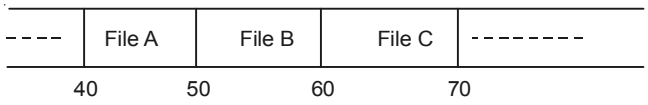
5.6 ALGO: RELEASING INODE (IN-CORE)

```
algorithm input
input : pointer to incore inode
```

```
output : none
    { lock inode if not already locked;
    decrement inode reference count;
    if (reference count = = 0)
    { if (inode link count = = 0)
    { free disk block for file;
       set file type to 0;
         free inode;
    }
    if (tile accessed or inode changed or file changed)
            update disk inode;
      put inode on free list;
    } release inode lock;
    }
```

5.7 STRUCTURE OF A REGULAR FILE

We know that the inode contains the table of contents to locate a files data on disk. Since each block on a disk is addressable by number, the table of contents consists of a set of disk block numbers. If the data in a file were stored in a contiguous section of the disk then storing the start block address and the file size in the inode would suffice to access all the data in the file.



Block Address

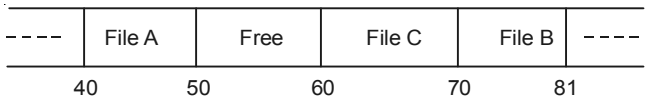


Fig. 5.1 Allocation of contiguous files and fragmentation of free space

Example: Suppose a user creates three files A, B, C each consisting of 10 disk blocks of storage and suppose the system allocated storage for the three files contiguous- of data to the middle file, B. The Kernel would have to copy file B to a place in the file system that had room for 15 blocks of storage. Aside from the expense of such an operation, the disk blocks previously occupied by file B's data would be unusable except for files smaller than 10 blocks. The Kernel could minimize fragmentation of storage space by periodically running garbage collection procedures to compact available storage, but that would place an added drain on processing power.

The Kernel allocates file space one block at a time and allows the data in a file to be spread throughout the file system. The table of contents could consist of a list of block numbers such that the block contain the data belonging to the file, but simple calculation show that a linear list of

file blocks in the inode is difficult to manage. If a logical block contain 1K bytes. Then a file consisting of 10K bytes would require an index of 10 block numbers. Either the size of inode would vary according to the size of the file, or a relatively low limit would have to be placed on the size of a file.

5.7.1 To Keep Inode Structure Small (Direct and Indirect block)

Assume that a logical block on the file system holds 1K byttes and that a block number is addressable by a 32 bit (4 byte). Then a block can hold upto 256 block numbers.

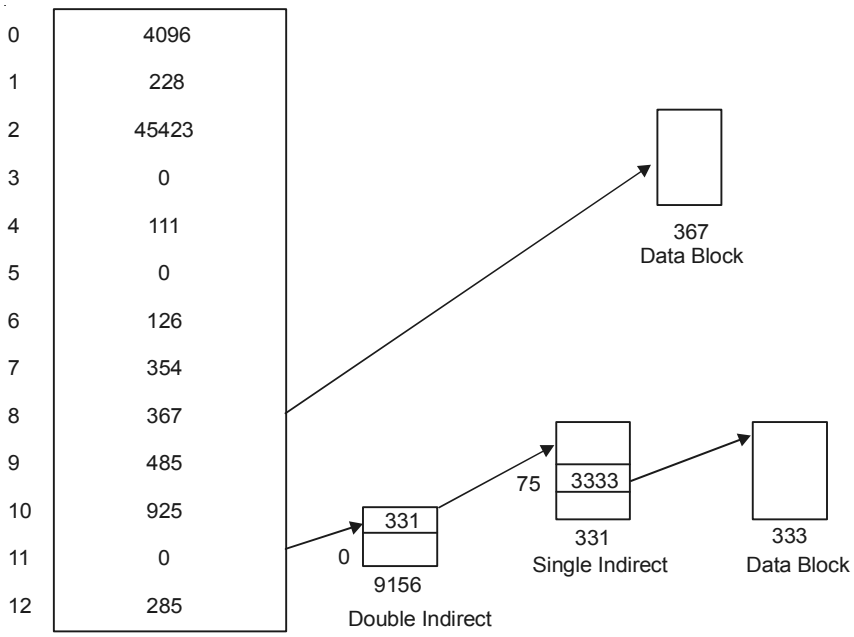


Fig. 5.2 Block layout of a simple file and its inode

Process access data in a file by byte offset. They work in terms of byte counts and view a file as a stream of bytes starting at byte address O and going up to the size of the file. The Kernel converts the user view of bytes into a view of blocks. The file starts at logical block O and continues to a logical block number corresponding to the file size. The Kernel access the inode and converts the logical file block into the appropriate disk block.

Consider the block layout in following figure and assume that a disk block contains 1024 bytes. If a process wants to access byte offset 9000. The Kernel find that the byte is in direct block 8 in the file (counting from 0).

Since in 8th block (trom 1) total bytes can be 8192 and in 9th block (9216) so the 9000 byte will be locate in 9th block (block number 8).

1024 = 216 = 808

It then access block number 367. The 808th byte in that block is byte 9000 in the file.

If a process wants to access byte offset 3,50,000 in a file. It must access a double indirect block. Since

256K + 10K = 272,384

So

$$(256K + 256K) + 1024 * 10 = 534528$$
$$\Rightarrow \quad 534{,}528 - 350{,}000 = 184528$$
$$\Rightarrow \quad 256K - 184528 = 262144 - 184528$$
$$= 77616$$

So byte number 77,616 of a single indirect block is in the 75th direct block in the single indirect block, block number 3333.

Since for single 256K + 10K = 272384 < 350,000

So Double 256K + 256K + 10K = 534528

534528 – 350,000 = 184528

So in single 256K – 184528 = 77616

So the bytes is in the 77616/1024 = 75.7 blocks

which is in the 76th block and block number 75 counted from O.

1024 * 76 – 77616 = 208

So the byte number 1024 – 208 = 816

Several block entries in the inode are a meaning that the logical block entries contain no data. This happens if no process ever wrote data into the file at any byte offsetes corresponding to those blocks and hence the block number remains at their initial value O. No disk space is wasted for such disk blocks.

```
/ * Block map of logical file byte offset to file system block * /
algorithm : b map
input     : (1) node
            (2) byte offset
Output    : (1) Block number in file system
            (2) byte offset into block
            (3) byte of I/O in block
            (4) Read ahead block number
{
Calculate logical block number in file/from byte offset;
Calculate start byte in block for I/O;
Calculate number of bytes to copy to user;
Check if read-ahead applicable, mark inode;
determine level of indirection:
while (not at necessary level of indirection)
{calculate index into inode or indirect
block-from logical block number in file;
get disk block number from inode or indirect block:
release buffer from previous disk read, if any;
```

```
if (no more level of indirection)
return (block number):
read indirect disk block;
adjust logical block number in file
according to level of indirection;
   }
}
```

5.8 DIRECTORIES

Directories are the files that give the file system its hierarchical structure; they play an important role in conversion of a file name to *an inode number. A directory is a file whose data is a sequence of entries each consisting of an inode number and the name* of a file contained in the directory. A path name is a null terminated character string divided into separate components by the "/" character. Each component except the last must be the name of a directory but the last component may be a non directory file. UNIX system V restricts component names to maximum of 14 characters; with a two byte entry for the inode number, the size of a directory entry is 16 bytes.

byte offset in directory	Inode number (2 bytes)	File names
0	83	.
16	2	..
32	1798	init
48	1276	trik
224	0	crash
208	1432	getty
240	95	mkts
	→ directory entry may be empty	

The Kernel stores data for a directory just as it stores data for an ordinary file using the inode structure and level of direct and indirect block.

5.9 CONVERSION OF A PATHNAME TO AN INODE NUMBER

```
algorithm : namei
input     : path name
output    : locked inode
{   if (pathname starts from root)
working inode = rootinode; // get inode
else
      working inode = current directory inode ; // get inode
while (there is more path name)
```

```
    { read next path name component from input;
    verify that working inode is of directory, access permissions OK;
    if (working inode is of root and component is "..")
         continue;
    read directory (working inode) by using blockmap,
    block read and block release for reading the content if (component
matches an entry in directory (working inode))
    {
    get inode number for matched component;
    release working inode;
    working inode = inode of matched component;
    }
    else
    return (no inode);
    }
         return (working inode);
}
```

The Kernel does a linear search of the directory file associated with the working inode, trying to match the pathname component to a directory entry name. Starting at byte offset O, it converts the byte offset in the directory to the appropriate disk block and reads the block. It searches the block for the pathname component, treating the contents of the block as a sequence of directory entry release the block and the old working inode and allocates the inode of the matched component. The new inode becomes the working inode. If the Kernel does not match the pathname with any names in the block, it release the block, adjust the byte offset by the number of bytes in a block, converts the new offset to a disk block number and reads the next block. The Kernel repeats the procedure until it matches the pathname component with a directory entry name or until it reaches the end of the directory.

Chapter 6

Inode Assignment to a New File

The file system contains a linear list of inodes. An inode is free if its type field is zero. When a process needs a new inode, the Kernel could theoretically search the inode list for a free node. However, such a search would be expensive, requiring atleast one read operation for every inode. To improve performance, the file system super block contains an array to cache the numbers of free inodes in the file system.

For assigning new inodes. The Kernel first verifies that no other processes have locked access to the super block free inode list. If the list of inode numbers in the super block is not empty the Kernel assigns the next inode number, allocates a free incore inode for the newly assigned disk inode copies the disk inode to the incore copy initilizes the fields in the inode and returns the locked inode. It updates the disk inode to indicate that the inode is now in use. A non zero file type field indicates that the disk inode is assigned.

Allocate inode (Assigning New Inodes)

```
algorithm  ialloc
input  :  File  system
output  :  locked  inode
{
        while  (not  done)
             {
if  (super  block  locked)
{  sleep  (event  :  super  block  becomes  free):
   continue;
}
if  (inode  list  in  super  block  is  empty)
{   lock  super  block:
get  remembered  inode  for  free  inode  search
search  disk  for  free  inodes  until  super  block  or  no  more  free
inodes;
```

```
unlock super block;
wakeup (event: super block becomes free)
if (no free inodes found on disk)
return (no inode);
set remembered inode for next free inode
}
get inode number from super block inode get inode;
if (inode not free after all) {write inode release inode;
continue;
}
        initialize inode;
        write inode to disk:
        decrement till system free inode count;
        return (inode);
    }
}
```

6.1 REMEMBERED INODE

If the super block list of free inodes is empty, the Kernel searches the disk and places as many free inode number as possible into the super block. The Kernel reads the inode-list on disk block-by-block and fills the super block list of inode numbers to capacity, remembering the high numbered inode that it finds call that inode remembered inode its the one saved in super block. The next time the Kernel searches the disk for free inode it user the remembered inode as its starting point, thereby assuring that it was no time reading disk blocks where no free inode should exist.

Whenever the Kernel assigns a disk inode, it decrements the free inode count recorded in the super block.

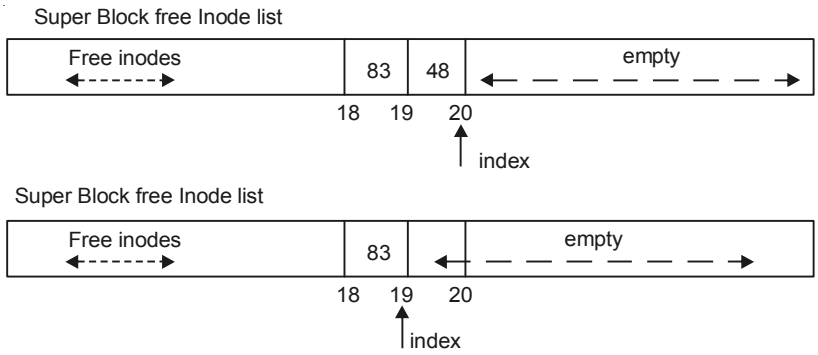


Fig. 6.1 Super block free inode list

(*a*) Assigning Free Inode-from Middle of List

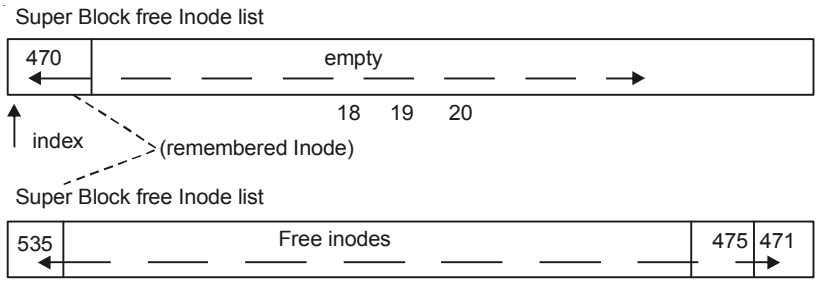


Fig. 6.2 Super block free inode list

(*b*) Assignment free Inode-Super Block list empty

If the list of free inodes in the super block available when the Kernel assigns a inode, it decrements the index for next value inode number (in Figure 6.2) and takes inode number.

If the list of free inodes in the super block not available, it will notice that the O is empty and search the disk for free inode starting from 470, the remembered inode. When the Kernel fills the super block free list to capacity it remembers the last inode as the start point for the next search of the disk. The Kernel assigns an inode, it just took from the disk (number 471 in figure) and continues whatever it was doing.

Freeing An Inode

```
Algorithm : ifree
input     : file system inode number
output    : none
{
increment file system free inode count;
if (super block locked)
          return;
if (inode list full)
{
if (inode number less than remembered inode for search)
set remembered inode for search = input inode number;
}
else
store inode number in inode list;
return;
}
```

For freeing an inode increment the total number of available inodes in the file system, the Kernel checks the lock on the super block. If locked it avoids race condition by returning immediately. The inode number is not put into the super block, but it can be found on disk and is available for reassignment. If the list is not locked the Kernel checks if it has room for more inode

numbers and if it does places the inode number in the list and returns. If the list is full the Kernel may not save the newly freed inode there. It compares the number of the freed inode with that of the remembered inode. If the freed inode number is less than the remembered inode number, it "remembers" the newly freed inode number, discarding the old remembered inode number from the super block. The inode is not last because the Kernel can find it by searching the inode list on disk. The Kernel maintains the super block list such that the last inode it dispenses from the list is the remembered inode. There should never be free inodes whose inode number is less than the remembered inode number, but exceptions are possible.

Process A	Process B	Process C
Assigns inode I	:	:
from super block	:	:
Sleeps while	:	:
reading inode (a)	:	:
:	Tries to assign inode	:
:	from super block	:
:	super block empty (b)	:
:	Search for free inode	:
:	on disk, puts inode I	:
:	in super block (c)	:
Inode I incore	:	:
does usual activity	:	:
:	completes search,	:
:	assign another mode (d)	:
:	:	Assign inode I from
:	:	super block
:	:	I is in use
:	:	Assign Another inode (e)

Fig. 6.3 Race condition in assigning inodes

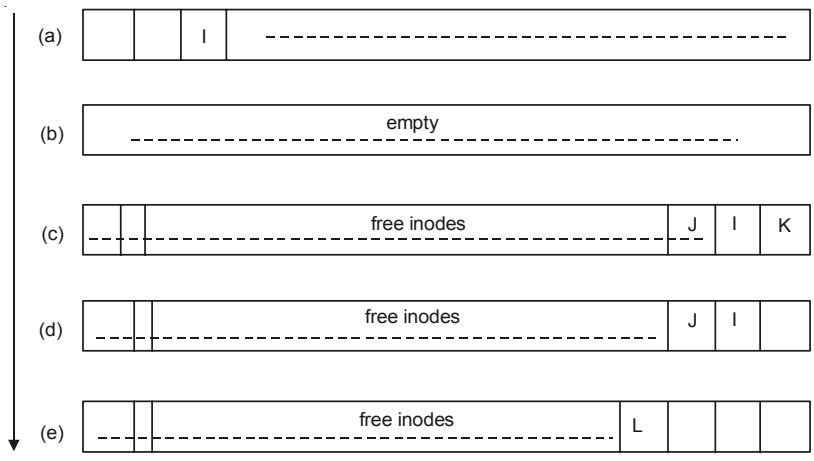
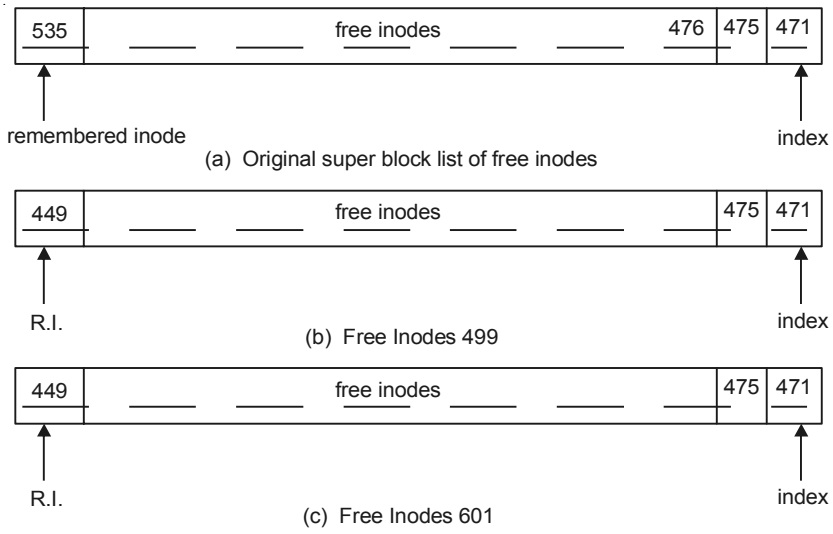
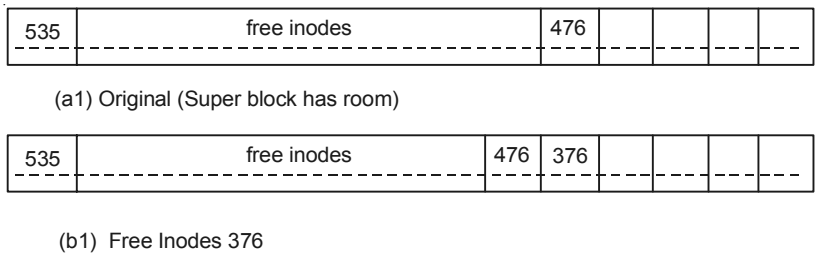


Fig. 6.4 Placing free inode number into the super block



Another list



6.2 ALLOCATION OF DISK BLOCK

When a process writes data to a file, the Kernel must allocate disk blocks from the file system for direct data blocks and sometimes for indirect data blocks. The file system super block contains an array that is used to cache the numbers of free disk blocks in the file system. The utility program *mkfs* organizes the data blocks of a file system in a linked list, such that each link of the list is a disk block that contains an array of free disk block numbers and one array entry is the number of the next block of the linked list.

When the Kernel wants to allocate a block from a file system, it allocates the next available block in the super block list. Once allocated, the block cannot be reallocate until it becomes free. If the allocated block is the last available block in the super block cache, the Kernel treats it as a pointer to a block that contains a list of free blocks. It reads the block, populate the super block array with the new list of block numbers and then proceeds to use the original block number. It allocates a buffer for the block and clears the buffers data zero. The disk block has now been assigned and the Kernel has a buffer to work with. If the file system contains no free blocks the calling process receives an error.

If a process *writes* a lot of data to a file, it repeatedly asks, the system for blocks to store the data, but the Kernel assigns only one block at a time. The program *mkfs* tries to organize the original linked list of free block numbers so that block numbers dispensed to a file are near each other. This helps performance because it reduces disk seek time and latency when a process reads a file sequentially as given in Figure 6.5.

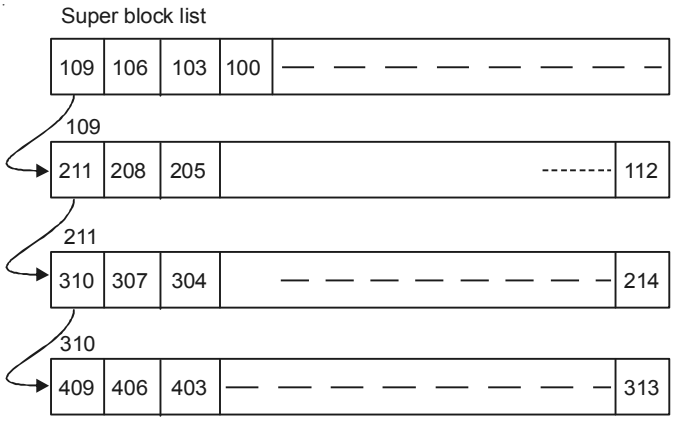


Fig. 6.5 Linked list of free disk block number

Unfortunately, the order of block numbers on the free block linked list break down with heavy use as process *write* files and remove them, because block numbers enter and leave the free list at random. The Kernel makes no attempt to sort block numbers on the free list.

6.2.1 Freeing a Block

If the super block list is not full the block number of the newly freed block is placed on the super block list. If the super block list is full, the newly freed block becomes a link block, the Kernel writes the super block list into the block and writes the block to disk. It then places the block number of the newly freed block in the super block list. That block is the only member of the list.

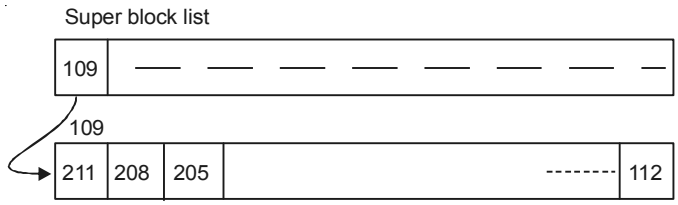
File System Block Allocation

```
algorithm : alloc
input     : file system number
output    : buffer for new block
{ while (super block locked)
sleep (event : super block not locked)
remove block from super block free list;
if (removed last block from free list)
{ lock super block;
read block just taken from free list;
copy block numbers in block into super block;
release block buffer;
unlock super block;
wake up process (event: super block not locked);
}
get buffer for block removed from super block list;
```

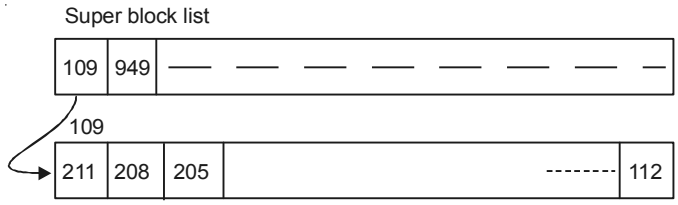
```
zero buffer contents;
decrement total count of free blocks;
mark super block modified;
return buffer;
}
```

6.3 DIFFERENT TREATMENT OF DISK BLOCK AND INODE

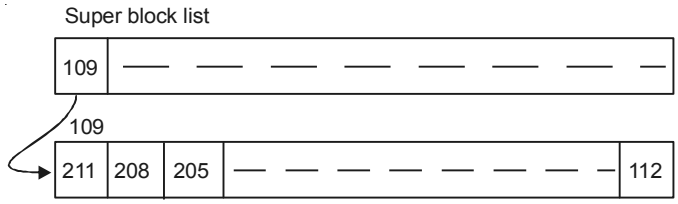
Block	*Inode*
1. It cannot determine whether a block is free just by looking at it. It could not distinguish between a bit pattern that indicates the block is and data that happened to have that bit pattern.	1. The Kernel can determine whether an inode is free if *the type field is clear.*
2. Disk blocks lend themselves to the use of linked list: A disk blocks hold large lists of free block numbers.	2. Inodes have no convenient place for bulk storage of large lists of free inode numbers.
3. Users tend to consume disk block resources more quickly, so searching free disk block is not critical.	3. Users tend to consume inode is less quickly so searching inode is very critical.



(a) Original Configuration



(b) After freeing block number (949)



(c) After assigning block number (949)

Chapter 7

SYSTEM CALLS

System calls are standard function which instruct the kernel to do some specific task.

7.1 TYPES OF SYSTEM CALLS

7.1.1 Open

This system call is the first step a process must take to access the data in a file. The syntax for this system call is:

fd = open (Pathname, flags, modes)

Pathname—is a file name. It may be absolute or relative.

Flags—Indicate the type of open such as for reading or writing.

Mode—Modes gives the file permission if the file is being created.

File descriptor—The open system calls return an integer called user file descriptor.

7.1.2 Algorithm Open

```
inputs : file name
         type of open
         file permission (for creation type of open)
output : File descriptor
{ convert file name to inode: // Pathname
if file does not exist or not permitted access
return (error):
allocate file table entry for inode, initilize count, offset;
allocate user file descriptor entry, set pointer to file table entry;
if (type of open specifies truncate file)
free all file blocks;
```

```
unlock  (inode);
return  (user  file  descriptor);
}
```

The Kernel searches the file system for the file name parameter. It checks permission for opening the file after it finds the incore inode and allocates an entry in the file table for the open file. The file table entry contains a pointer to the inode of the open file and a field that indicates the byte offset in the file where the Kernel expects the next read or write to begin. The Kernel initializes the offset to O during the open call, meaning that the initial read or write starts at the begining of a file by default. The entry in the user file table points to the entry in the global file table.

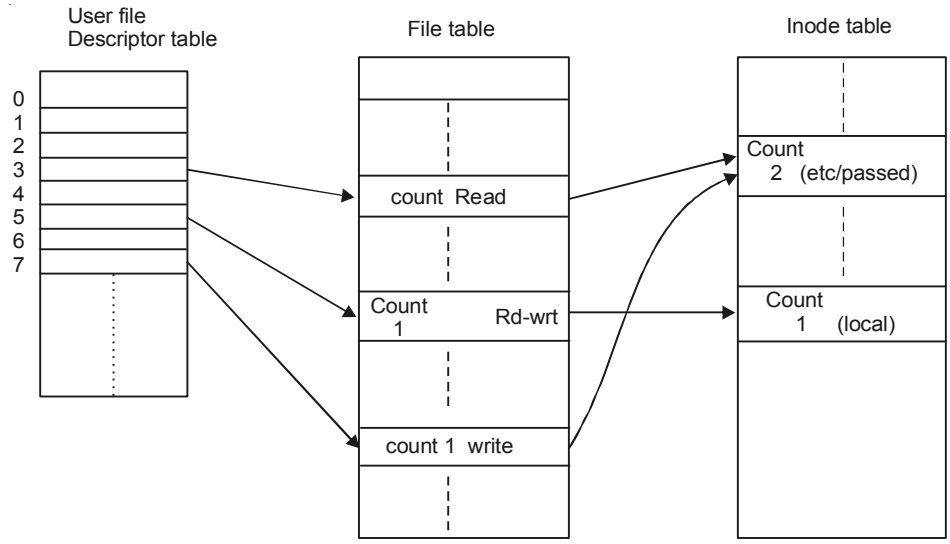


Fig. 7.1 Data structures after open

fd1 = Open ("/etc/passwd", O – RDONLY);

fd2 = Open ("local", O – RDWR);

fd3 = Open ("/etc/passwd", O – WRONLY);

The first three user file description (0, 1, 2) are called the standard *i/p* standard *o/p* and standard error file descriptors.

7.1.3 Read

The syntax of *read* system call is number = read (fd, buffer, count)

fd—fd is the file descriptor returned by the open system call.

buffer—buffer is the address of a data structure in the user process that will contain the read data on successful completion of the call.

count—count is the number of bytes the user wants to read.

number—is the number of bytes actually read.

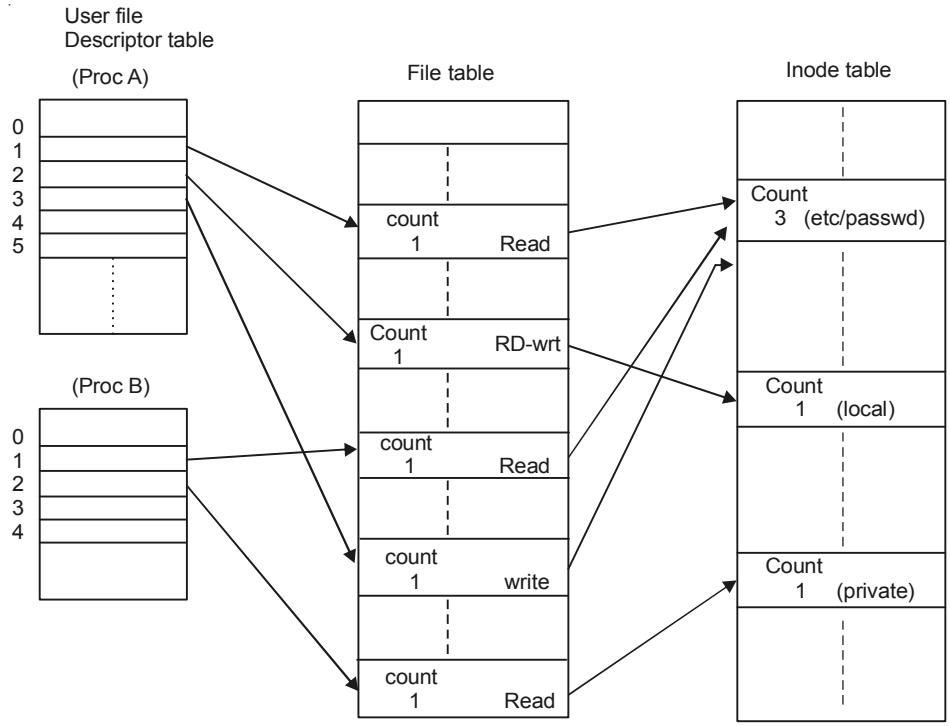


Fig. 7.2 Data structures after two process open files

7.2 ALGORITHM : READ

```
input : user file descriptor
      : address of buffer in user process
      : number of bytes to read
Output : count of bytes copied into user space.
{
get file table entry from user file descriptor;
check file accessibility;
set parameters in u-area for user address,
byte count, I/O to user;
get inode from file table;
lock inode;
set byte offset in u-area from file table offset;
while (count not satisfied)
{ convert file offset to disk block;
calculate offset into block, number of bytes to read;
```

```
if (number of bytes to read is 0)
break;
read block;
copy data from system buffer to user address;
update u-area fields for file byte offset,
read count, address to write into user space;
release buffer;
}
unlock inode;
update file table offset for next read;
return (total number of bytes read);
}
```

I/O Parameters Saved in u-Area

mode : Indicates read or write

count : count of bytes to read or write

offset : byte offset in file

address : target address to copy data, in user or Kernel memory

flag : indicates if address is in user or Kernel memory.

After reading the block into buffer, it copies the data from the block to the target address in the user process. It updates the I/O parameters in the *u*-area according to the number of bytes it read, incrementing the file where the next data should be delivered and decrementing the count of bytes it needs to read to satisfy the user read request. If the user request is not satisfied, the Kernel repeats the entire cycle converting the file byte offset to a block number, reading the block from disk to a system buffer, copying data from the buffer to the user process releasing the buffer and updating I/O parameters in the *u*-area. The cycle completes either when the Kernel completely satisfies the user request, when the file contains no more data, or if the Kernel encounters an error in reading the data from disk or in copying the data to user space. The Kernel updates the offset in the file table according to the number of bytes it actually read.

Example: For Reading a File

```
main ( )
{ int fdi char lilbuf [20], bigbuf [1024];
fd = open ("/etc/passwd," O_RDONLY);
read (fd, lilbuf, 20);
read (fd, bigbuf, 1024);
read (fd, lilbuf, 20);
}
```

In this example open ( ) return the file descriptor (fd), the Kernel verifies that the file descriptor parameter is legal and that the process had previously opened the file for reading. It stores the values *lilbuf* 20 and O in the *u*-area corresponding to the address of the user buffer, the

byte count and the starting byte offset in the file. It calculates that the byte offset O is in Oth block of the file and retrieves the entry for the Oth block in the inode. Assuming such a block exists the Kernel reads the entire block of 1024 bytes into a buffer but copies only 20 bytes to the user address lilbuf. It increments the *u*-area byte offset to 20 and decrement the count of data to read to O. Since the read has been satisfied, the Kernel resets the file table offset to 20, so that subsequent reads on the file descriptor will begin at byte 20 in the file and the system call returns the number of bytes actually read 20.

For second read call determine that '*fd*' is legal. It stares in the *u*-area the user address bigbuf; the number of bytes the process wants to read, 1024 and the starting offset in the file 20, taken from the file table. It converts the file offset to the correct disk block as above and reads the block. The Kernel cannot satisfy the *read* request entirely from the buffer, because only 1004 *out of the 1024 bytes for this request are in the buffer. So it copies the last 1004 bytes* from the buffer into the user data structure *bigbuf* and updates the parameters in the *u*-area to indicate *that the next iteration* of the read loop starts at byte 1024 in the file, that the data should be copied to byte position 1004 in bigbuf and that the number of bytes to satisfy the read request is 20.

Then Kernel looks up the second direct block number in the inode and find the correct disk block to read. It copies 20 bytes from the buffer to the correct address in the user process. Before leaving the system call, the Kernel sets the offset field in the file table entry to 1044, the byte offset that should be accessed next. For *last* system call start the reading at byte 1044 in file.

A Reader and A Writer Process

```
/ * Process A * /
main ()
{ int fd ; charbuf [512];
fd = open ("/etc/-passwd", O-RDONLY);
read (fd, buf, size of (buf)); / * read 1 * /
read (fd, buf, size of (buf)); /* read 2 * /
}
/ * process B * /
main ()
{ int fd, i
char buf [512];
for (i = 0; i < size of (buf); i + t)
buf (i) = 'a';
fd = open ("/etc/passwd", O_WRONLY);
write (fd, buf, size of (buf)); / * write 1 * /
write (fd, buf, size of (buf)); / * write 2 * /
}
```

When a process invokes the read system call the Kernel locks the inode for the duration of call. Afterwards, it could go to sleep reading a buffer associated with data or with indirect blocks of the inode. If another process were allowed to change file while the first process was sleeping,

read could return inconsistent data. Hence the inode is left locked for the duration of the read call, affording the process a consistent view of the file as it existed at the start of the call.

The Kernel can preempt a reading process between system calls in user mode and schedule other process to run. Since the inode is unlocked at the end of a system call, nothing prevents other processes from accessing the file and changing its contents. It would be unfair for the system to keep an inode locked from the time a process opened the file until it closed the file because one process could keep a file open and thus prevent other processes from ever accessing it. To avoid such problem the Kernel unlocks the inode at the end of each system call that user it. If another process changes the file between the two read system calls by the first process, the first process may read unexpected data, but the Kernel data structures are consistent.

Write → Syntax

Number = Write (fd, buffer, count)

For writing a regular file. If the file does not contain a block that corresponds to the byte offset to be written, the Kernel allocates a new block and assigns the blocks number to the correct position in the inode table of contents. If the byte offset is that of an indirect block, the Kernel may have to allocate several blocks for use as indirect blocks and data blocks. The inode is locked for duration of **write**, because the Kernel may change the inode when allocating new blocks allowing other processes access to the file corrupted the inode if several process allocates block simultaneously for the same byte offset. When the write is complete, the Kernel updates the file size entry in the inode if the file has grown larger.

7.3 ADJUSTING THE POSITION OF FILE I/O—LSEEK

lseek() system calls is use to position the I/O and allow random access to a file. The syntax

position = lseek (fd, offset, reference)

fd → file descriptor

offset → is a byte offset

reference → Indicates whether offset should be considered from the beginning of the file, from the current position of read/write offset, or from the end-of-file.

position → is the byte offset where the next read or write will start.

Program with lseek call

```
main (int argc, char * argv[])
{ int fd, skval; Char C;
if (argc 1 = 2)
exit ();
Fd = open (argv[1], O_RDONLY);
if (fd = = -1)
exit (1);
while ((skval = read (fd, 8C, 1)) = = 1)
{ print f ("Char % C   ", C);
o for beginning of file
```

```
1 for 1024th by of file
2 for beyond the eof
skval = lseek (fd, 1023L, 1);
print f ("new seekval%d\ n", skval); }
```

7.4 CLOSE

A process closes an open file when it no longer wants to access it. Syntax.

close (Fd);

Tables after closing a file

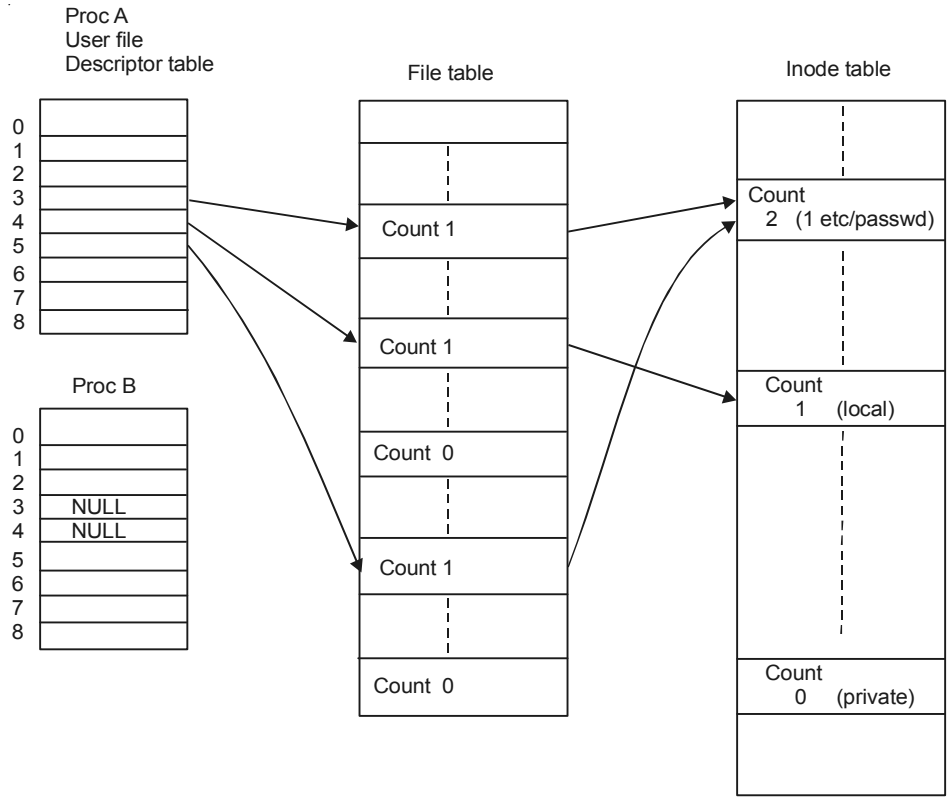


Fig. 7.3

The Kernel does the close operation by manipulating the file descriptor and the corresponding file table and inode table entries. If the reference count of file table entries is greater than I because of dup or tork calls then other user file descriptors reference the file table entry, the Kernel decrements the count and the close completes. If the file table reference count is 1, the Kernel frees the entry and releases the in-core inode.

If other processes still reference the inode, the Kernel decrements the inode reference count but leaves it allocated, otherwise the inode is free for reallocation because its reference count is 0.

7.5 FILE CREATION

The open system call gives a process access to an existing file, but the create system call creates a new file in system. Syntax

Fd = create (pathname, modes);

If no such file existed previously, the Kernel creates a new file with the specified name and permission modes; if the file already existed the Kernel truncates the file subject to suitable file access permission.

```
algorithm : creat
input     : file name
          : permission settings (mode)
output    : file descriptor
{ get inode for file name;
if (file already exist)
{ if (not permitted access)
{ release inode;
return error;
}
}
else
{ assign free inode from file system;
create new directory entry in parent directory;
include new file name and newly assigned inode number;
}
allocate file table entry for inode, initilize count;
if (file did exist at time of create)
free all file blocks;
unlock (inode);
return (user file descriptor);
}
```

7.6 CREATION OF SPECIAL FILES

The system call mknod() creates special files in the system including named pipes, device files and directories. Syntax

mknod (pathname, type and permissions, dev)

pathname → is the name of the node to be created.

Type and permissions → Give the node type (for example Directory)

and access permissions for the new file to be created.

dev → dev specifies the major and minor device number for block and character special files.

```
algorithm : make new node
inputs : node (file name)
file type
permissions
major, minor device number
output : none
{ if (new node not named pipe and user not super user)
return (error);
get inode of parent of new node;
if (new node already exists)
{ release parent inode;
return (error); }
assign free inode from file system for new node;
create new directory entry in parent directory;
include new node name and newly assigned inode number;
release parent directory inode;
if (new node is block or character special files)
write major, minor numbers into inode structure;
release new node inode;
}
```

Change Directory → Chdir (pathname);
Change Root → Chroot (pathname);
Change Owner → Chown (pathname, owner, group);
Change Mode → Chmod (pathname, mode)

7.7 STAT AND FSTAT

There system call allow processes to query the status of files, returning information such as the file type, file owner, access permission file size, number of links, inode number and file access times. Syntax.

stat (pathname, statbuffer);

fstat (Fd, statbuffer);

pathname → file name

fd → file descriptor

Statbuffer → is the address of a data structure in the user process that will contain the status information of the file on completion of the call. The system calls simply write the fields of the inode into statbuffer.

7.8 PIPES

Pipes allow transfer of data between processes in FIFO manner and they also allow synchronization of process execution. Their implementation allows processes to communicate even though they do not know what processes are on the other end of pipe. The traditional implementation of pipes uses the file system for data storage. There are two kinds of pipes.

(*i*) Named pipe → Process use open system call for this pipe.

(*ii*) Unnamed pipe → Pipe() system call to create an unnamed pipe.

Only related processes, descendants of a process that issued the pipe call can share access to unnamed pipe.

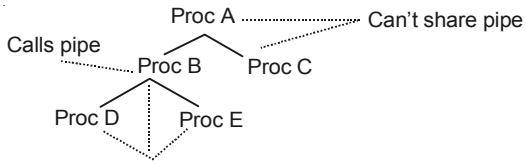


Fig. 7.4 Share pipe

7.8.1 Process Tree and Sharing Pipes

However all process can access a named pipe regardless of their relationship, subject to the usual file permissions.

(*i*) The pipe system call → Syntax

pipe (fdptr);

fdptr → The pointer to an integer array that will contain the two file descriptors for reading and writing the pipe. Because the Kernel implements pipes in the file system and because a pipe does not exist before its use, the Kernel must assign an inode for it on creation. It use the file table so that the interface for the read, write and other system calls is consistent with the interface for regular files. Processes do not have to know whether they are reading or writing a regular file or a pipe.

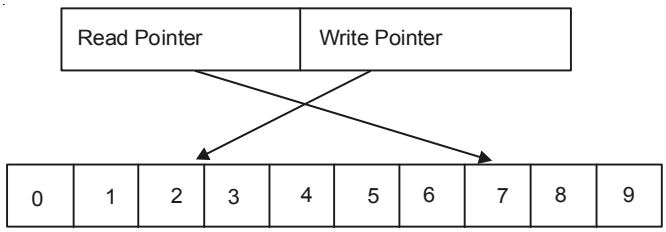
```
algorithm : pipe (unnamed pipe)
input     : none
output    : read file descriptor
          : write file descriptor
{ assign new inode from pipe device;
allocate file table entry for reading other for writing;
initialize file table entry to point to new node;
allocate user FD for reading another for writing;
initialize to point to respective file table entries;
set inode reference count to 2;
initialize count of inode readers, writers to 1;
}
```

(*ii*) Opening a named pipe → A named pipe is a file whose semantics are the same as those of an unnamed pipe, except that it has a directory entry and is accessed by a pathname. Processes open named pipes in the same way that they open regular files and hence processes that are not closely related to communicate. Named pipes permanently exist in the file system hierarchy but unnamed pipes are transient. When all process finish using the pipe, the Kernel reclaims its inode.

A process that opens the named pipe for reading will sleep until another process opens the named pipe for writing and vice versa.

(*iii*) Reading and writing pipes → A pipe should be viewed as if processes write into one end of the pipe and read from other end. The number of processes reading from a pipe do not necessarily equal the number of processes writing the pipe, if the number of readers or writers is greater than 1. They must coordinate use of the pipe with other mechanisms.

The difference between storage allocation for a pipe and a regular file is that a pipe uses only the direct blocks of the inode for greater efficiency although this places a limit on how much data a pipe can hold at a time. The Kernel manipulates the direct blocks of the inode as a circular queue.



7.9 FOUR CASES FOR READING AND WRITING PIPES

(*a*) Writing a pipe that has room for the data being written.

(*b*) Reading from a pipe that contains enough data to satisfy the read.

(*c*) Reading from a pipe that does not contain enough data to satisfy the read.

(*d*) Writing a pipe that does not have room for the data being written.

Closing pipes → The Kernel decrements the number of pipe readers or writers, according to the type of the file descriptor. If the count of writer processes drops to O and there are processes asleep waiting to read data from the pipe the Kernel awakens them and they return from their read calls without reading any data. If the count of reader process drops to O and there are processes asleep waiting to write data to the pipe, the Kernel awakens them and send them a signal to indicate an error condition.

In both cases, it makes no sense to allow the processes to continue sleeping when there is no hope that the state of the pipe will ever change.

```
Char.string = "Hello";
main ()
{ Char buf [1024]; * CP1, * CP2;
```

```
int fds (2)                → writing and reading a pipe
CP1 = string;
CP2 = buf;
while (*CP1)
*CP2 + + = * CP1 ++;
pipe (fds);
for (;;) {write (fds(1), buf, 6}; read (fds (0), buf, 6)} }
```

7.9.1 Reading and Writing a Named Pipe

```
Char string [] = "Hello"
main (int argc, Char * argv[])
{int Fd ; Char buf [256];
/ * create Named Pipe with read/write permission/or all users * /
mknod ("Fifo", 777, 0);
if (argc = = 2)
Fd = open ("Fifo"O_WONLY);
else
Fd = open ("Fito", O. RDONLY);
For (;;)
if argc = = 2
write (Fd, string 6);
else
read (Fd, buf, 6);
}
```

7.10 DUP

The dup() system calls copies a file descriptor into the first free slot of the user file descriptor table, returning the new file descriptor to user. It works for all file types. Syntax.

```
newfd = dup (fd)
main ()
{ int i, j; Char buf [512], buf 2 [512];
i = open ("/etc/password," O_RDONLY);
j =  dup (i);
read (i, buf1, size of (buf1));
read (), buf2, size of (buf2));
close (i);
read (i, buf2, size of (buf2));
}
```

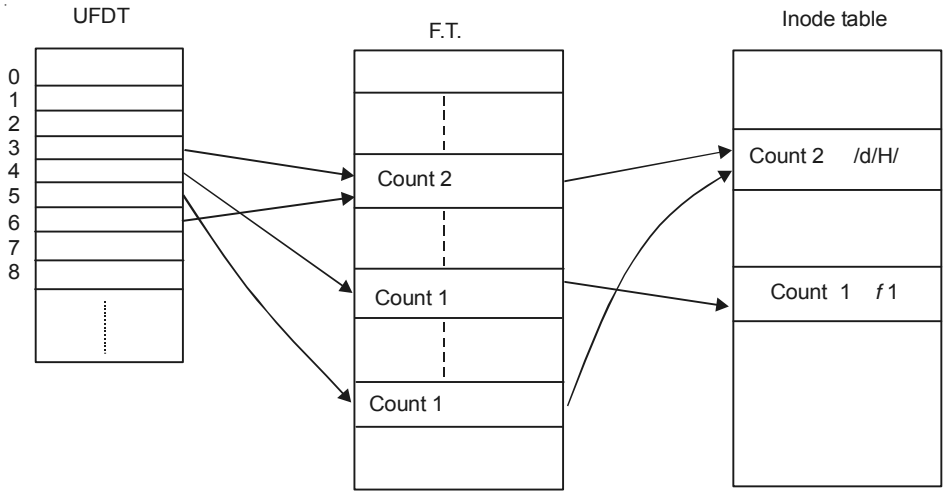


Fig. 7.5 Data structure after dup

7.11 MOUNTING AND UNMOUNTING FILE SYSTEMS

A physical disk unit consists of several logical sections, partitioned by the disk driver and each section has a device file name. Processes can access data in a section by opening the appropriate device file name and then reading and writing the file, treating it as a sequence of disk blocks. A section of a disk may contain a logical file system, consisting of a boat block super block, inode list and data blocks. The mount system call connects the file system in a specified section of a disk to the existing file system hierarchy and the unmount system call disconnects a file system from the hierarchy. The mount system call thus allows users to access data in a disk section as a file system instead of a sequence of disk blocks. Syntax.

7.12 MOUNT (SPECIAL PATHNAME, DIRECTORY PATHNAME, OPTIONS)

1. Special pathname → is the name of the device special file of the disk section containing the file system to be mounted.
2. Directory pathname → is the directory in the existing hierarchy where the file system will be mounted (mount point)
3. Options → indicate whether the file system should be mounted "read-only".

7.13 MOUNT FILE

The Kernel has a mount table with entries for every mounted file system. Each mount table entry contains:

(*i*) A device number that identifies the mounted file system.

(*ii*) A pointer to a buffer containing the file system super block.

(*iii*) A pointer to root inode of mounted file system.

(*iv*) A pointer to the inode of the directory that is the mount-point.

Association of the mount point inode and the root inode of the mounted file system set up during the mount system call, allows the Kernel to traverse the file system hierarchy gracefully without special user knowledge.

7.14 ALGORITHM FOR MOUNTING A FILE SYSTEM

```
inputs : file name of block special file
       : directory name of mount point
       : options (read only)
output : none
{   if (not super user)
       return (error);
get inode for block special file;
make legality check;
get inode for "mounted on" directory name
if (not directory or reference count > 1)
{
release (inode);
return (error);
}
find empty slot in mount table;
invoke block device driver open routine;
get free buffer from buffer cache;
read super block into free buffer;
initialize super block fields;
get root inode of mounted device, save in mount table;
mark inode of "mounted on" directory as mount point;
release special file inode;
unlock inode of mount point directory;
}
```

The Kernel finds the inode of the special file that represents the file system to be mounted, extracts the major and minor numbers that identify the appropriate disk section and finds the inode of directory on which the file system will be mounted. The reference count of the directory inode must not be greater than 1 because of potentially dangerous side effects. The Kernel then allocates a free slot in the mount table marks the slot in use and assigns the device number field in the mount table.

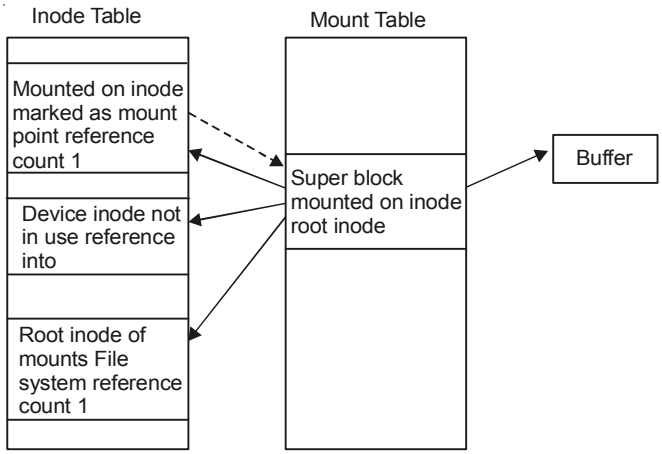


Fig. 7.6 Data structures after mount

7.15 CROSSING MOUNT POINTS IN FILE PATHNAMES

The two cases for crossing a mount point are:

(*i*) Crossing from the mounted on file system to the mounted file system (in the direction from the global system root towards a leaf node).

(*ii*) Crossing from the mounted file system to the mounted-on file system.

Example: mount / dev / dskl / usr ↵

(*i*) Case Cd / usr/ src/ uts

(*ii*) Case Cd ../ .. / ..

7.16 REVISED ALGORITHM FOR ACCESSING AN INODE

```
input  : file system inode number
output : locked inode
{ while (not done)
{ if (inode in inode cache)
{ if (inode locked)
{ sleep (event inode becomes unlocked);
   continue;
}
if (inode a mount point)
{ find mount table entry for mount point;
get new file system number from mount table;
use root inode number in search;
continue; }
```

```
if (inode on inode free list)
remove from free list;
increment inode reference count;
return (inode);
}
remove new inode from free list;
reset inode number and file system;
remove inode from old hash queue, place on new one;
read inode from disk;
initialize inode;
return inode;
}
}
```

Revised Algorithm for parsing a file name

```
algo  : namei
input : pathname
output: locked inode
{ if (pathname start from root)
 working inode = root inode;
else
working inode = current directory inode;
while (there is more pathname)
{ read next pathname component from input;
 verify that inode is of directory, permissions;
 if (inode is of changed root and component is ",,")
 continue;
Component Search;
 read inode (directory);
 if (component matches a directory entry)
 { get inode number for matched component;
if (found inode of root and working inode is root and component
name is ",,")
{
get mount table entry for working inode;
release working inode;
working inode = mounted on inode
lock mounted on inode;
increment reference count of working inode;
get O component search (for “,,”);
```

```
}
release working inode;
working inode = inode for new inode number;
}
else
return (no inode);
} return (working inode);
}
```

7.17 UNMOUNTING A FILE SYSTEM → SYNTAX

```
umount (special file name);
algorithm : umount
input     : special file name of file system to be unmounted
output    : none
{ if (not super user)
return (error);
get inode of special file
extract major, minor number of device being unmounted;
get mount table entry, based on major, minor number
for unmounting file system;
release inode of special file;
remove shared text entries from region table
for files belonging to file system;
update super block inodes, flush buffers;
if (files from file system still in use)
return (error);
get root inode of mounted file system from mount table;
lock inode;
release inode;
invoke close routine for special device;
invalidate buffers in pool from unmounted file system;
get inode of mount point - point from mount table;
lock inode;
clear flag marking it was mount point;
release inode;
free buffer used for super block;
free mount table slot;
}
```

7.18 LINK

Link system call links a file to a new name in the file system directory structure, creating a new directory entry for an existing inode. Syntax.

link (source file name, target file name);

Source file name → is the name of an existing file

Target file name → is the new (additional) name the file will have after completion of link call.

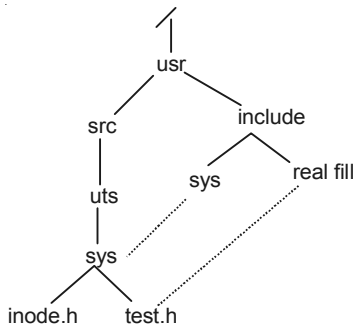


Fig. 7.7 Linked files in file system tree

```
algorithm : link
input     : existing file name
          : new file name
output    : none
{ get inode for existing file name;
if (too many links on file of linking directory without super user
permission)
{ release (inode);
return (error);
}
increment link count on inode;
update disk copy of inode;
unlock inode;
get parent inode for directory to contain
new file name;
if (new file name already exist or existing file, news file on
different file systems)
{ undo update done above;
return (error);
}
create new directory entry in parent directory of new file name :
include new file name,
inode number of existing file name;
```

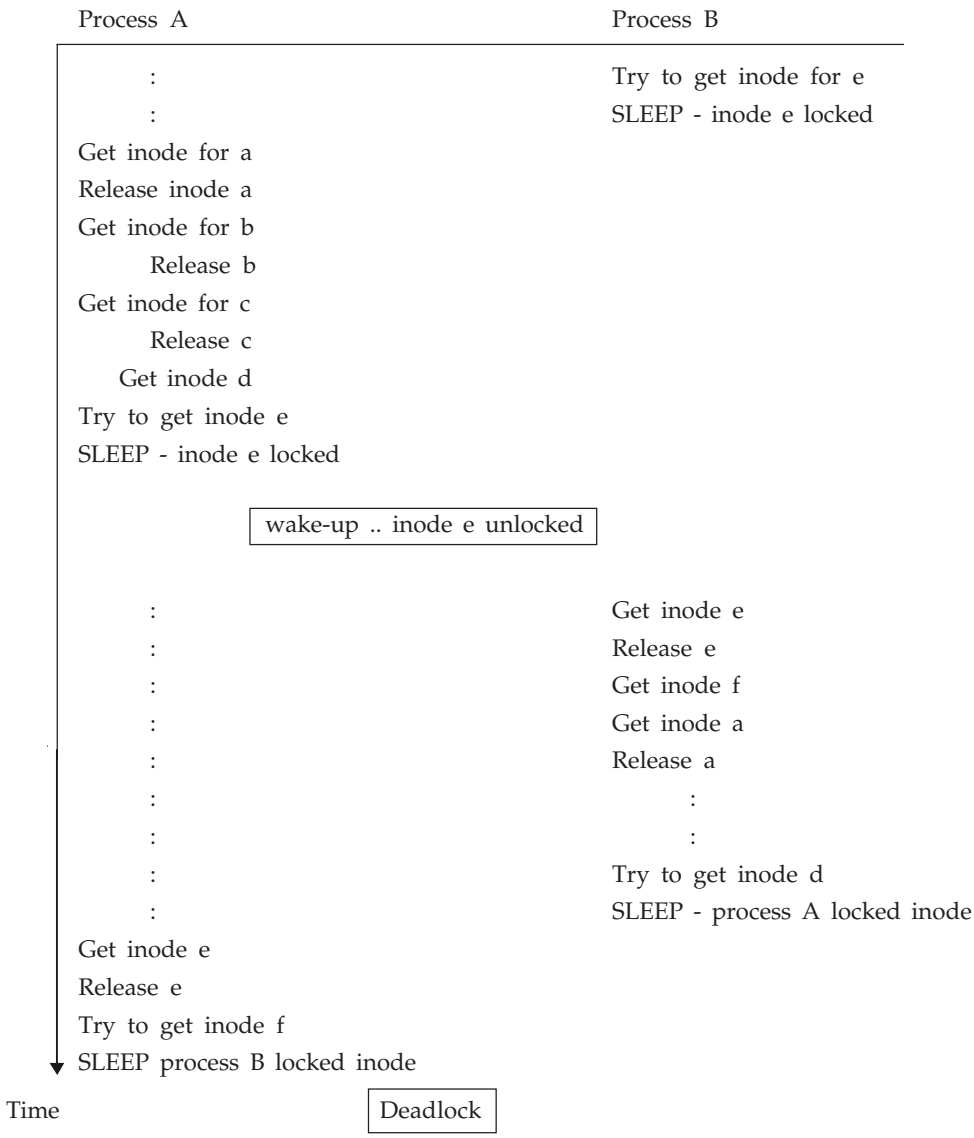
```
release parent directory inode;
release inode of existing file;
}
```

7.18.1 Deadlock in Link Call

Two deadlock possibilities both concerning the reason the process unlocks the source file inode after incrementing its link count. If the Kernel did not unlock the inode two process could deadlock by executing the following system calls simultaneously.

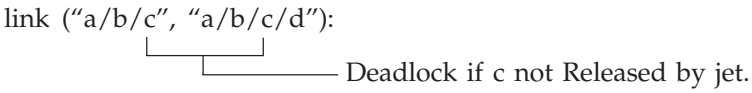
Process A : link ("a/b/c/d", "e/f/g");

Process B : link ("e/f", "a/b/c/d/ee");



In this example process A would be holding a locked inode that process B wants and process B would be holding a locked inode that process A wants. The Kernel avoids this deadlock condition by releasing the source file inode after incrementing its link count. Since the first resource (inode) is free when accessing the next resource, no deadlock can occur.

This example showed how two process could deadlock each other if the inode lock were not released. A single process could also deadlock itself. If it executed.



If two process or even one process, could not continue executing because of deadlock. So since inodes are finitely allocatable resources, receipt of a signal cannot awaken the process from its sleep. Hence, the system could not break the deadlock without rebooting. If no other processes accessed the files over which the processes deadlock, no other process in the system would be effected.

7.19 UNLINK

The unlink() system call removes a directory entry for a file. Syntax.

unlink (pathname);

If the file being unlinked is the last link of the file the Kernel eventually free its data blocks

```
algorithm : unlink
input     : file  name
output    : none
{ get  parent  inode  of  file  to  be  unlinked:
if  (last  component  of  file  name  is  ".");
increment  inode  reference  count;
else
get  inode  of  file  to  be  unlinked  :
if  (file  is  directory  but  user  is  not  super  user)
{  release  inodes;
return  (error);
}
if  (shared  text  file  and  link  count  currently  1)
remove  from  reqion  table;
write  parent  directory  :  zero  inode  number  of  unlinked  file;
release  inode  parent  directory;
decrement  file  link  count;
release  file  inode;
}
```

I File System Consistency

The Kernel orders its writes to disk to minimize file system corruption in event of system failure. For instance when it removes a file name from its parent directory,

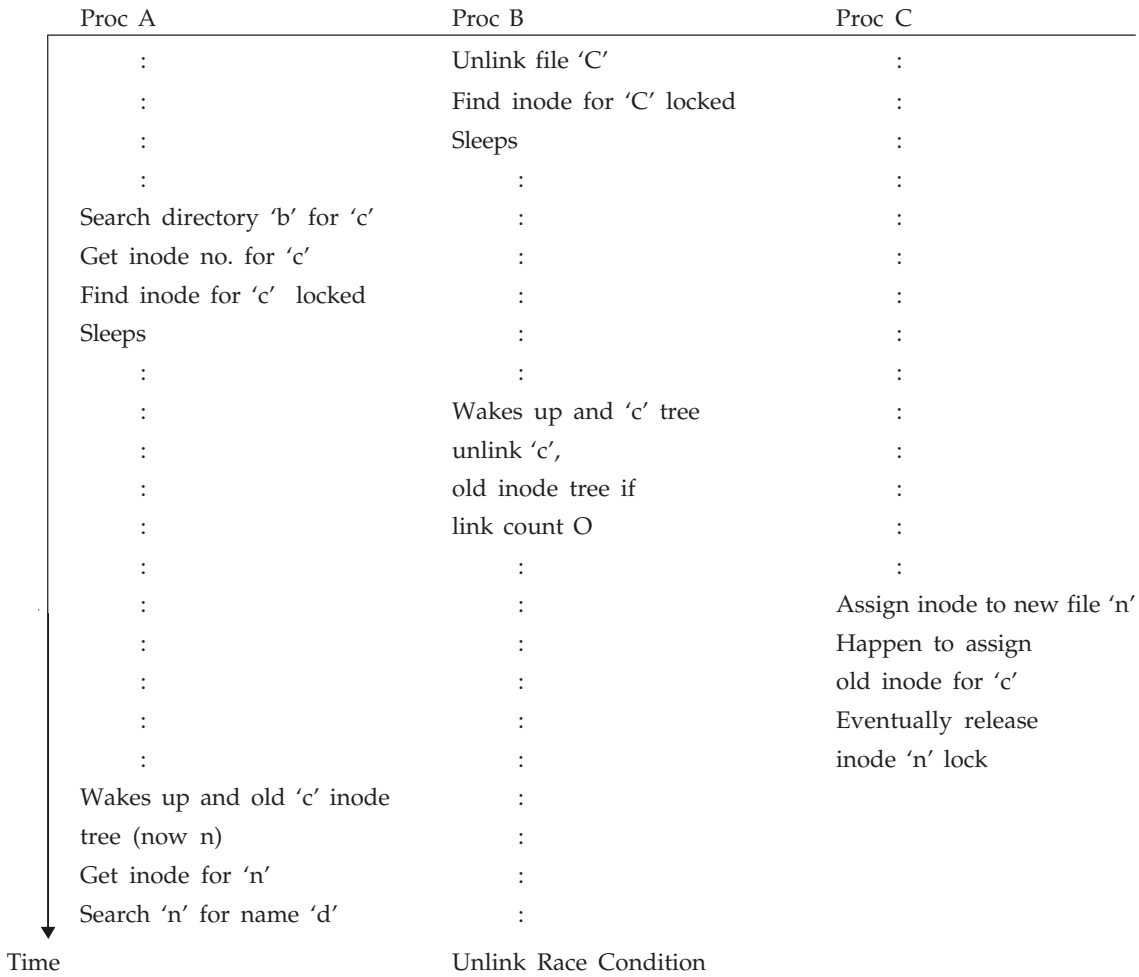
(*i*) It writes the directory synchronously to the disk before it destroys the contents of the file and frees the inode.

If the system were to crash before the file contents were removed, damage to the file system would be minimal. There would be an inode that would have a link count '1 greater than the number of directory entries that access it, but all other paths to the file would still be legal.

(*ii*) If the directory write were not synchronous it would be possible for the directory entry on disk to point to a free (or reallocated) inode after a system crash. Thus there would be more directory entries in the file system that refer to the inode than the inode would have link counts.

II Race Condition

Race conditions abound in the unlink system call, particularly when unlinking directories.



Chapter 8

STRUCTURE OF A PROCESS

The Kernel contains a process table with an entry that describes the state of every active process in the system. The *u*-area contains additional information that controls the operation of a process. The process table entry and the *u*-area are part of the context of a process. The aspect of the process context that most visibly distinguishes it from the context of another process is of course, the contents of its address space.

8.1 PROCESS STATES AND TRANSITIONS

The following list contains the complete set of process states.

(*i*) The process is executing in user mode.

(*ii*) The process is executing in Kernel mode.

(*iii*) The process is not executing but is ready to run as soon as the Kernel schedules it.

(*iv*) The process is sleeping and resides in main memory.

(*v*) The process is ready to run, but the swapper (schedule process O) must swap the process into main memory before the Kernel can schedule it to execute.

(*vi*) The process is sleeping and the swapper has swapped the process to secondary storage to make room for other processes in main memory.

(*vii*) The process is returning from the Kernel to user mode, but the Kernel preempts it and does a context switch to schedule another process.

(*viii*) The process is newly created and is in a transition state; the process exists but its not ready to run, nor is it sleeping. This state is the start state for all processes except process O.

(*ix*) The process executed the exit system call and is in the zombie state. The process no longer exists, but it leaves a record containing an exit code and some timing statistics for its parent process to collect. The zombie state is the final state of a process.

The process enters the state model in the "created" state when the parent process executes the fork () system calls. The process schedular will eventually pick the process to execute and the process enters the state "Kernel running" where it completes its part of the fork() system call.

When the process completes the system call it may move to the state "user running" where it executes in user mode. After a period of time, the system clock may interrupt the processor and the process enters state "preempted" and the other process executes.

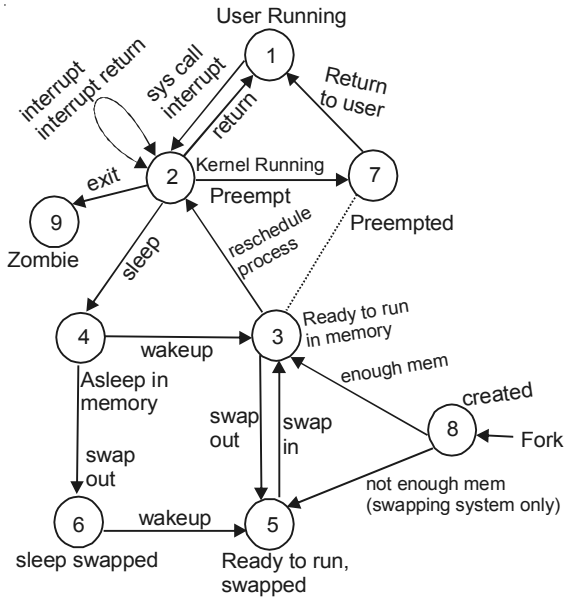


Fig. 8.1 Process state transition diagram

When a process executes a system call; it leaves the state "user running" and enters the state "Kernel running". Suppose the system call requires I/O from the disk and the process must wait for the I/O to complete. It enters the state "asleep in memory" putting itself to sleep until its notified that the I/O has completed. When the I/O later completes, the H/W interrupts the CPU, and the interrupt handler awakens the process causing it to enter the state "ready to run in memory".

When a process completes, it invokes the exit system call, thus entering the states "Kernel running" and finally the "zombie" state.

8.2 KERNEL DATA STRUCTURES

There are two Kernel data structures that describes the state of a process:

8.2.1 Process Table

The process table contains the filed that must always be accessible to the Kernel. The fields in process table are:

(*i*) The state field identifies process state.

(*ii*) The process table entry contains fields that allow the Kernel to locate the process and its *u*-area in main memory or secondary storage. The process table entry also contains a field that gives the process size, so that the Kernel knows how much space to allocate for the process.

(*iii*) User identifiers (UID) determine various process privileges.

(*iv*) Process identifiers (PID) specifies the relationship of processes to each other.

(*v*) The process table entry contains an event descriptor when the process is in the "sleep" state.

(*vi*) Scheduling parameters allow the Kernel to determine the order in which processes move to the states "Kernel running" and "User Running".

(*vii*) A signal field enumerates the signals sent to a process but not yet handled.

(*viii*) Various timers gives process execution time and Kernel resource utilization, used for process accounting and for the calculation of process scheduling priority. One field is a user-set timer used to send an alarm signal to a process.

8.2.2 *u*-Area

The *u*-area contains fields that need to be accessible only to the running process. Therefore, the Kernel allocates space for *u*-area only when creating a process. It does not need *u*-areas for process table entries that do not have process. The fields in *u*-area are:

(*i*) A pointer to the process table identifies the entry that corresponds to *u*-area.

(*ii*) The real and effective user—ID determine various privileges allowed the process such as file access rights.

(*iii*) Timer fields record the time the process spent executing in user mode and in Kernel mode.

(*iv*) An array indicates how the process wishes to react to signals.

(*v*) The control terminal field identifies the "login terminal" associated with process if exists.

(*vi*) An error field records error encountered during a system call.

(*vii*) A return value field contains result of system calls.

(*viii*) I/O parameters describe the amount of data to transfer, the address of the source (or target) data array in user space, file offset for I/O and so on.

(*ix*) The current directory and current root describe file system environment of the process.

(*x*) The user-file-descriptor table record the file the process has open.

(*xi*) Limit field restrict the size of a process and the size of a file it can write.

(*xii*) A permission mode field masks mode setting on files the process creates.

8.3 LAYOUT OF SYSTEM MEMORY

A process on UNIX System consist of three logical sections: text, data and stack.

The text section contains the set of instructions the machines executes for the process; address in the text section includes text address (for branch instructions or subroutine calls), data addresses (for access to global data variables), or stack address (for access to data structures local to a subroutine).

If the *m/c* were to treat the generated addresses as address locations in physical memory it would be impossible for two processes to execute concurrently. If their set of generated address overlapped. The compiler could generate addresses that did not overlap between programs, but such a procedure impractical for general purpose computers because the amount of memory on a *m/c* is finite and the set of all programs that could be compiled is finite.

The compiler therefore generates addresses for virtual address space with a given address range, and the machines memory management unit translates the virtual address generated by the compiler into address locations in physical memory. The compiler does not know where in memory the Kernel will later load the program for execution. In fact several copies of a program can coexist in memory: All executes using the same virtual addresses but reference different physical addresses.

(*i*) *Region:* The Kernel divides the virtual address space of a process into logical regions. A region is a contiguous area of the virtual address space of a process that can be treated as a distinct object to be shared or protected. Thus text, data and stack usually forms seperate regions of a process. Several process can share a region.

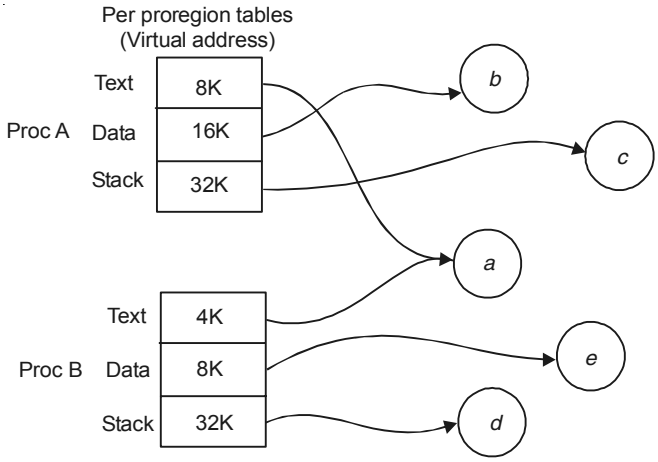


Fig. 8.2 Processes and regions

(*ii*) *Pages and Pages tables*: In a memory management architecture based on pages, the memory management h/w divides physical memory into a set of equal-sized blocks called pages. Typical page sizes range from 512 bytes to 4K bytes and are defined by the h/w. Every addressable location in memory is contained in a page and consequently every memory location can be addressed by a (page number, byte offset in page) pair.

Addressing Physical Memory as Pages

Hexadecimal Address	5 8 432
Binary	0101 10000100 01100010
Page Number, Page Offset	01 0110 0001 000011 0010
In Hexadecimal	161 32

In this example a *m/c* has 2^{20} bytes of physical memory and a page size of 1K bytes it has 2^{10} pages of physical memory; every 320 bit address can be treated as a pair of consisting of a 10 bit page number and 10 bit offset into the page.

If address is 2^{32} bytes of physical memory with page size 1K bytes then

22-bit for page number. And

K-bit for affect into pages.

Mapping of logical to physical page number.

Logical Page Number	Physical Page Number
0	177
1	54
2	209
3	17

Since a region is a contiguous range of virtual addresses in a program, the logical page number is the index into an array of physical page numbers. The region table entry contains a pointer to a table of physical page numbers called a page table. Page table entries may also contain *m/c*—dependent information such as permission bits to allow reading or writing of the page. The Kernel stores-page tables in memory and access them like all other Kernel data structures.

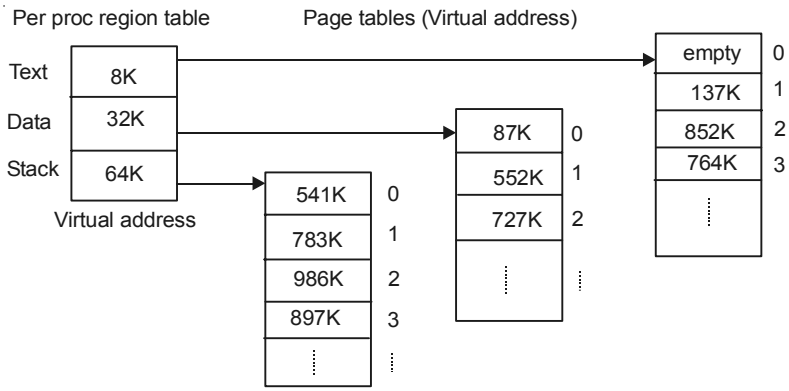


Fig. 8.3 Mapping virtual addresses to physical addresses

Assume page size is 1K bytes and process want to access V.M.A. (Virtual Memory Address) 68432. The P region entries shows that the address is in stack region starting from virtual address 64K (65,536).

$$68432 - 6536 = 2896$$

Since each page is 1K so the address is contained at byte offset 848 in page 2 of region located at physical address 986K.

We use following memory model in discussing memory management. The system contains a set of memory management register triples.

(*i*) First Register in triples contains the address of a page table in physical memory.

(*ii*) Second Register contains the first virtual address mapped via the triple.

(*iii*) Third Register contains control information such as the number of pages in the page table and page access permissions (read, write, execute).

(*iv*) Layout of the Kernel → The virtual memory mapping associated with the Kernel is independent of all processes. The code and data for the Kernel resides in the system permanently, and all processes share it. In many machines the virtual address space of a process is divided into several classes, including system and user and each class has its own page tables. When executing in Kernel mode the system permits access to Kernel addresses but it prohibits such access when executing in user mode.

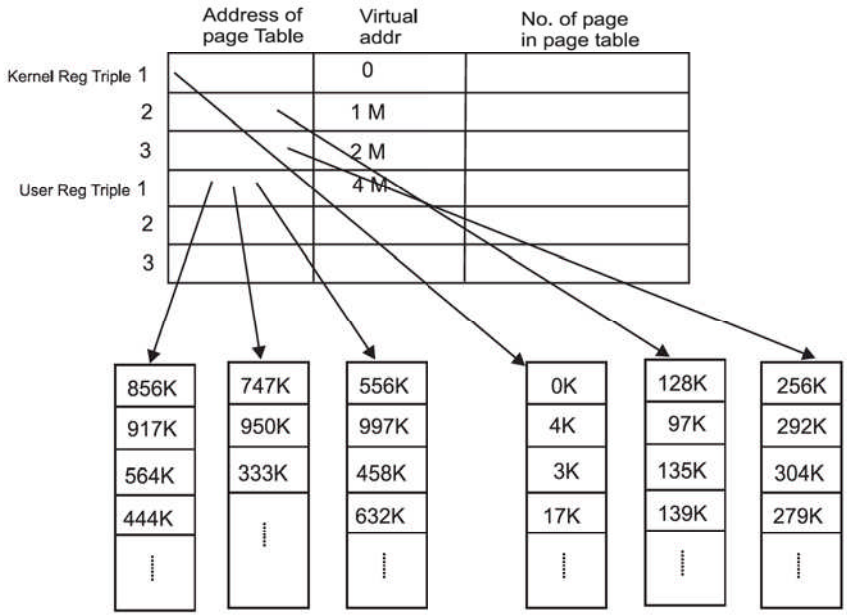


Fig. 8.4 Changing mode from user to kernel

(*iii*) The *u*-Area: Every process has a private *u*-area, yet the Kernel access it as if there were only one *u*-area in the system that of the running process. The Kernel changes its, the value of the *u*-area virtual address is known to other parts of the Kernel, in particular the module that does the context switch. The Kernel knows where in its memory management tables, the virtual address translation for *u*-area is done and it can dynamically change the address mapping of the *u*-area to another physical address.

A process can access its *u*-area when it executes in Kernel mode but not when it executes in user mode. Because the Kernel can access only one *u*-area at a time by its virtual address, the *u*-area partially defines the context of the process that is running on the system.

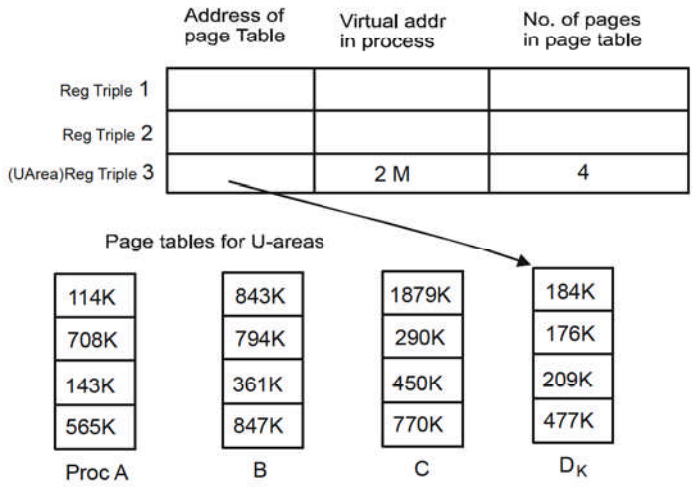


Fig. 8.5 Memory map of *u*-area in the kernel

8.4 THE CONTEXT OF A PROCESS

The context of a process consist of the contents of h/w Registers the contents of its (user) address space and Kernel data structures that relate to the process. We can say context of a process is the union of its Register context, user level context and system, level context.

(*a*) *User level context* → It consist of the process, text, data, user stack, and shared memory that occupy the virtual address space of the process.

Parts of the virtual address space of a process that periodically do not reside in main memory because of swapping or paging still constitute a part of the user level context.

(*b*) *Register Context* → It consist of:

(*i*) The Program counter specifies the address of the next instruction the CPU will execute.

(*ii*) The Process or Status Register (PS) specifies the h/w status of the machine as it Relates to process.

(*iii*) The stack pointer contains the current address of the next entry in the Kernel or user stack, determined by the mode of execution.

(*iv*) The general purpose register contain data generated by the process during its execution.

(*c*) *System level context* → The system level context of a process has a "static part" and a "dynamic part". A process has one static part of the system level context throughout its lifetime, but it can have a variable number of dynamic parts. The dynamic part of a system level context should be viewed as a stack of context layers that the Kernel pushes and paps on occurrence of various events. It consist of following components:

(*i*) The process table entry of a process defines the state of a process and contains control information that is always accessible to the Kernel.

(*ii*) The *u*-area of a process contains process control information that need be accessed only in the context of the process.

(*iii*) P region entry, region tables and page tables, define the mapping from virtual to physical addresses and therefore define the text, data, stack and other regions of a process. If several process share common regions, the regions are considered part of the context of each process, because each process manipulates the regions independently.

(*iv*) The Kernel stack contains the stack frames of Kernel procedures as a process executes in Kernel mode. Although all processes executes the identical Kernel code, they have a private copy of the Kernel stack that specifies their particular invocation of the Kernel functions. The Kernel stack is empty when the process executes in user mode.

(*v*) The dynamic part of the system, level context of a process consists of a set of layers, visualized as a last-in-first-out stack. Each system level context layer contains the necessary information to recover the previous layer including the register context of the previous level.

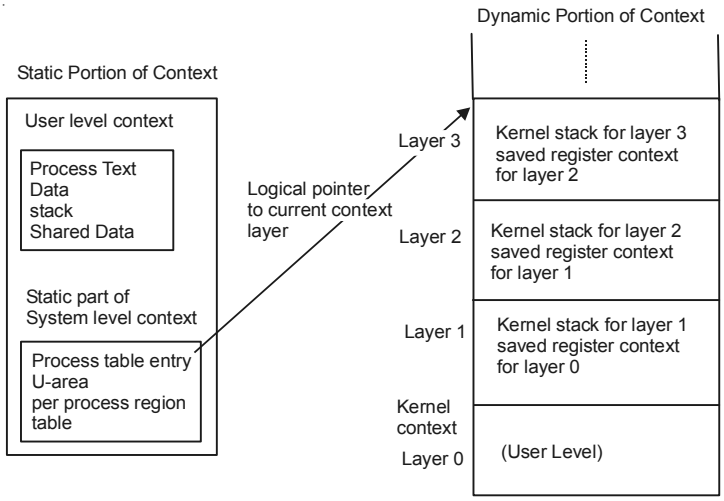


Fig. 8.6 Components of context of a process

The Kernel pushes a context layer when an interrupt occurs, when a process makes a system call or when a process does a context switch. It pops context layer when the Kernel returns from handling an interrupt, when a process returns to user mode after the Kernel completes execution of a system call, or when a process does a context switch. The Kernel pushes the context layer of the old process and pops the context layer of new process. The process table entry stores the necessary information to recover the current context layer.

A process runs within its context layer. The number of context layers is bounded by the number of interrupt levels the *m/c* supports.

8.4.1 Sleep

When a process goes to sleep, it typically does so during execution of a system call. The process enters the Kernel (context layer 1) when it executes on operating system trap and goes to sleep awaiting a resource. When the process goes to sleep, it does a context switch, pushing its current context layer and executing in Kernel context layer 2. Process also goes to sleep they incur page faults as a result of accessing virtual addresses that are not physically loaded; they sleep while the Kernel reads, the contents of the pages.

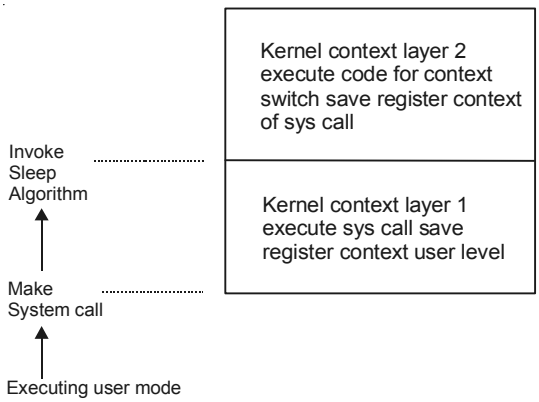


Fig. 8.7

8.5 TYPICAL CONTEXT LAYERS OF A SLEEPING PROCESS

(*i*) *Sleep Events and Address:* Processes are said to sleep on an event if they are in the sleep state until the event occurs at which time they wakeup and enter a "ready-to-run" state (in memory or swapped out). Although the system uses the abstraction of sleeping on an event, the implementation maps the set of events into a set of (Kernel) virtual address. The abstraction of the event does not distinguish how many processes are waiting the event nor does the implementation. As a result two anomalies arise:

(*a*) When an event occurs and a wakeup call is issued for processes that are sleeping on an event; they all wakeup and move from a sleep state to a ready to run state.

(*b*) Several events may map into one address:

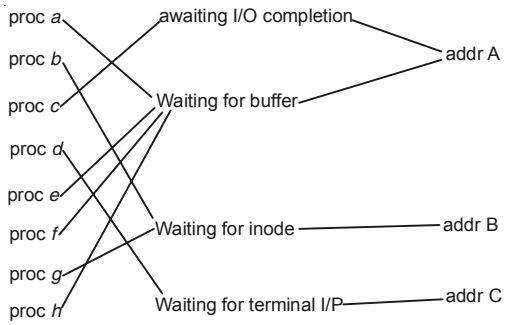


Fig. 8.8 Processes sleeping on events and events mapping into address

```
algorithm : sleep
input     : (1) Sleep address (2) priority.
output    : 1 if process awakened as a result of a signal that
process catches, jump algorithm if process awakened as a result of a
signal that it does not catch, 0 otherwise;
{ raise processor execution level to block all interrupt;
set process state to sleep;
put process on sleep hash queue, based on sleep address;
save sleep address in process table slot;
set process priority level to input priority;
if (process sleep is NOT interruptible)
{ do context switch;
reset processor priority level to allow interrupts
as when process went to sleep;
return (0); }
if (no signal pending against process)
{ do context switch;
if (no signal pending against process)
{ reset processor priority level to what it was
```

```
when process went to sleep;
return (0); } }
remove process from sleep hash queue, if still there;
reset processor priority level to what it was
when process went to sleep;
if (process sleep priority set to catch signals)
return (1)
do longjmp. algorithm; }
```

(*ii*) Algorithm for Sleep and Wakeup → To wakeup sleeping process, the Kernel executes the wakeup algorithm either during the usual system call algorithm or when handling an interrupt. The Kernel raises the processor execution level in wakeup to block out interrupts. Then for every process sleeping on the input sleep address, it marks the process state field "ready to run", removes the process from the linked list of sleeping processes, places it on a linked list of processes eligible for scheduling, and clears the field in the process table that marked its sleep addresss.

```
algorithm : wakeup
input     : sleep address
output    : none
{ raise processor execution level to block all interrupts;
find sleep hash queue for sleep address;
for (every process asleep on sleep address)
{ remove process from hash queue;
mark process state  "ready to run";
put process on schedular list of processes
ready to run;
clear field in process table entry for sleep address;
if (process not loaded in memory)
wakeup swapper process (0);
else if (awakened process is more eligible to run than currently
running process) set schedular flag;
}
restore processor execution level to original level;
}
```

8.6 MANIPULATION OF THE PROCESS ADDRESS SPACE

However, various system calls manipulate the virtual address space of a process doing so according to well defined operations on regions.

The region table entry contains the information necessary to describe a region. It contains:

(*i*) A pointer to the inode of the file whose contents were originally loaded into the region.

(*ii*) The region type (text, shared Memory, Private data or stack)

(*iii*) The size of the region.

(*iv*) The location of the region in physical memory.

(*v*) The status of a region, which may be a combination of
- locked, - in demand.
- in the process of being loaded into memory.
- valid, loaded into memroy.

(*vi*) The reference count, giving the number of processes that reference the region.

The operations that manipulate regions are:

I. *Locking and Unlocking a Region:* The Kernel has operations to lock and unlock a region, independent of the operations to allocate and free region. Thus the Kernel can lock and allocate a region and later unlock it without having to free the region.

II. *Allocating a Region:* The Kernel allocates a new region during fork, exec, and shmgt (shared memory) system calls. The Kernel contains a region table whose entries appear either on a free linked list or on an active linked list. When it allocates a region table entry, the Kernel removes the first available entry from the free list, places it on the active list, locks the region, and marks its type (shared or private).

```
algorithm : allocreg
input     : (i) inode pointer (ii) region type
output    : locked region
{ remove regions from linked list of free regions;
assign region type;
assign region inode pointer;
if (inode pointer not null)
increment inode reference count;
place region on linked list of active regions;
return (locked region);
}
```

III. *Attaching a region to a process:* The Kernel attach a region during the Fork, exec, and shmat system calls to connect it to the address space of a process. The region may be a newly allocated region or an existing region that the process will share with other processes. The Kernel allocates a free p region entry, sets its type field to text, data, shared memory, or stack, and records the virtual address where the region will exist in the process address space. The process must not exceed the system-imposed limit for the highest virtual address and the virtual addresses of the new region must not overlap the addresses of existing regions.

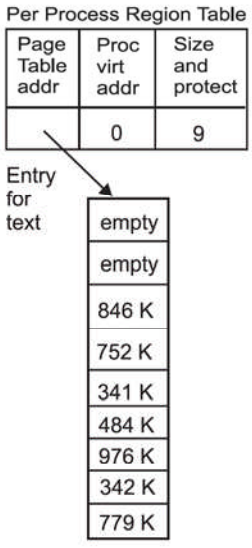


Fig. 8.9 Example of attaching to an existing text region

IV. *Changing the size of a region:* A process may expand or contract its virtual address space with the sbrk system call. Similarly the stack of a process automatically expands according to the depth of nested procedure calls. Internally, the Kernel invokes the algorithm growreg to change the size of a region. When a region expands, the Kernel makes sure that the virtual addresses of the expanded region do not overlap those of another region and that the growth of the region does not cause the process size to become greater than the maximum allowed virtual memory space. The Kernel never invokes growreg to increase the size of a shared region that is already to several processes, therefore, it does not have to worry about increasing the size of a region for one process and causing another process to grow beyond the system limit for process size.

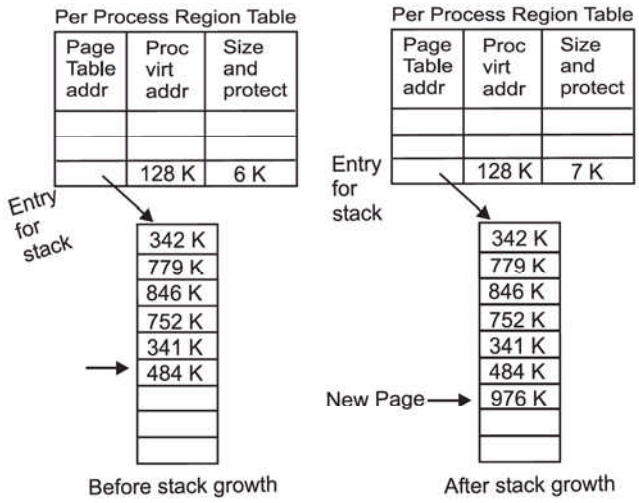


Fig. 8.10 Growing the stack region by 1K bytes

V. *Loading a region:*

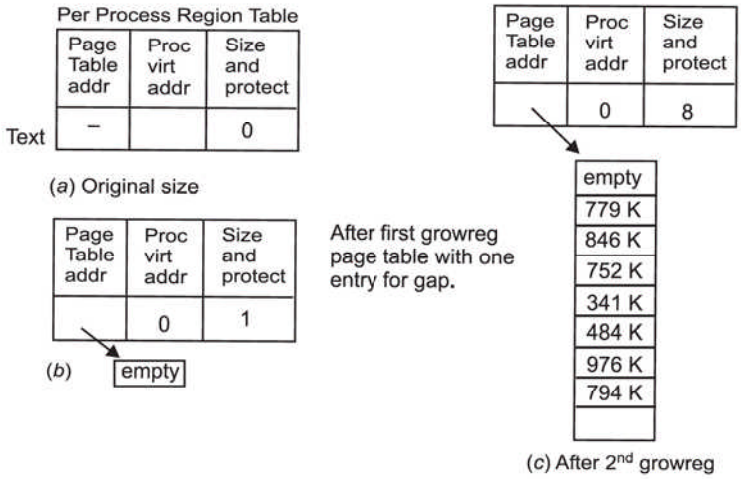


Fig. 8.11

VI. *Freeing a region:* When a region is no longer attached to any processes, the Kernel can free the region and return it to the list of free regions. The Kernel release physical resources associated with the region, such as page tables and memory pages.

VII. *Detaching a region from a process:* The Kernel detaches regions in the exec, exit and shmdt (detach shared memory) system calls. It updates the pregion entry and revers the connection to physical memory by invalidating the associated memory management register triple. The Kernel decrements the region reference count and the size field in the process table entry according to the size of the region.

Chapter 9

PROCESS CONTROL

The fork system call creates a new process, the exit call terminates process, execution and the wait call allows a parent process to synchronize its execution with the exit of a child process. Signals inform processes of a synchronous events. Because the Kernel synchronizes execution of exit and wait via signals.

9.1 PROCESS CREATION

To create a new process in the UNIX operating system is to invoke the fork system call. The process that invokes fork is called the parent process, and the newly created process is called the child process. The syntax for the fork system call is:

pid = fork ();

On return from the fork system call the two processes have identical copies of their user level context except for the return value pid. In the parent process pid is the child process ID; in the child process, pid is O. Process O created internally by the Kernel when the system is boated, is the only process not created via fork.

The Kernel does the following sequence of operations for fork:

(*i*) It allocate a slot in the process table for the new process.

(*ii*) It assigns a unique ID number to child process.

(*iii*) It makes a logical copy of the context of the parent process.

(*iv*) It increments file and inode table counters for files associated with the process.

(*v*) It returns the ID number of the child to the parent process, and a value to the child process.

```
algorithm : fork
input     : none
output    : to parent process, child PID no. to child process, O.
{ check for available Kernel Resources;
```

```
    get free proc table slot, unique PID number;
    check that user not running too many processes;
    mark child stat "being created;"
    copy data from parent proc table slot to new child slot;
    increment counts on current directory inode
    and changed root (if applicable);
    increment open file counts in file table;
    make copy of parent context (u-area, text, data, stack) in memory;
    push dummy system level context layer onto
    child system level context;
    dummy context contains data allowing child process to recognize
itself and start running from here when scheduled;
    if (executing process is parent process)
    { change child state to "ready to run"
    return (child ID);
    }
    else
    { initialize u-area timing fields;
    return (0);
    }
    }
```

The system imposes a (configurable) limit on the number of processes a user can simultaneously execute so that no user can steal many process table slots, thereby preventing other users from creating new users.

After the state of process is "being created". The Kernel adjusts reference counts for files with which the child process is automatically associated.

(*i*) The child process resides in the current directory of the parent process. The number of processes that currently access the directory increase by 1 and accordingly, the Kernel increments its inode reference count.

(*ii*) If the parent process or one of its ancestors had ever executed the chroot system call to change its root, the child process inherits the changed root and increments its inode reference count.

(*iii*) The Kernel searches the parent's user file descriptor table for open files known to the process and increments the global file table reference count associated with each open file.

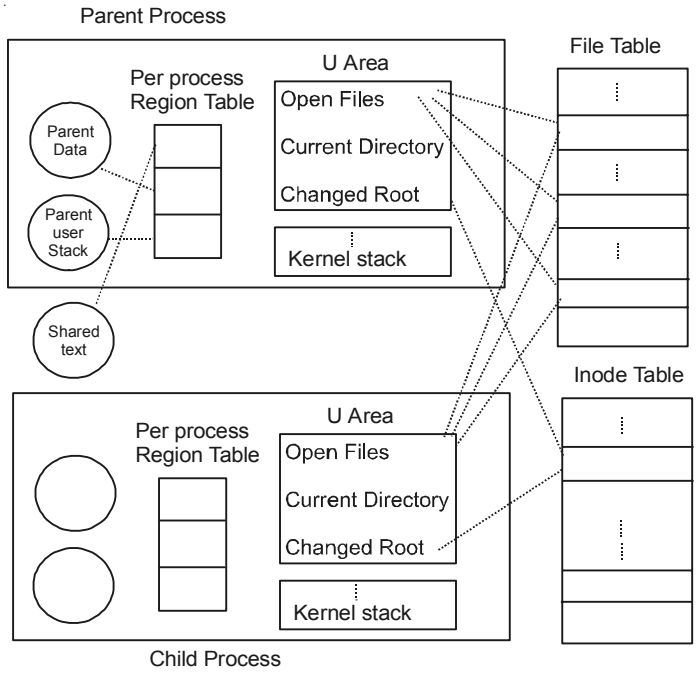


Fig. 9.1 Fork creating a new process context

9.2 AWAITING PROCESS TERMINATION

A process can synchronize its execution with the termination of a child process by executing the wait system call. Syntax.

pid = wait (stat – addr)

pid: is the process ID of the zombie child.

stat addr: is the address in user space of an integer that will contain the exit status of the child.

```
algorithm : wait
input     : address of variable to store status of existing process
output    : child ID, child exit code
{
if (waiting process has no child processes)
return (error);
for (;;)
{ if (waiting process has zombie child)
{ pick arbitrary zombie child;
add child CPU usage to parent;
free child process table entry;
```

```
return (child ID, child exit code);
}
if (process has no children)
return (error);
sleep at interruptible priority (event child process exists);
}
}
```

The Kernel searches for a zombie child of the process and if there are no children, returns an error. If it find a zombie child it extracts the PID number and the parameter supplied to the child exit call and returns those value from the system call. An exiting process can thus specify various return codes to give the reason it exited. The Kernel adds the accumulated time the child process executed in user and in Kernel mode to the appropriate fields in the parent process *u*-area and finally release the process table slot formerly occupied by the zombie process. The slot is now available for a new process.

If the process executing wait has child processes but none are zombie, it sleeps at an interruptible priority until the arrival of a signal. If the signal is "death of child" the process may respond differently.

9.3 THE USER-ID OF A PROCESS

The Kernel associates two user-IDs with a process, independent of the process ID:

(*i*) Real user-ID: It identifies the user who is responsible for the running process.

(*ii*) Effective user-ID: Its used to assign ownership of newly created files to check file access permissions and to check permission to send signals to process via the kill system call.

The Kernel allows a process to change its effective user-ID when it excess a setuid program or when it invoke setuid system call explicitly. A setuid program is an executable file that has the setuid bit set in its permission mode field. When a process excess a setuid program, the Kernel sets the effective user-ID fields in the process table and *u*-area to the owner ID of the file. Syntax.

setuid (uid);

uid → uid is the new user-ID and its result depends on the current value of the effective user-ID. If the effective user-ID of the calling process is not super user the Kernel reset the effective user-ID in the *u*-area to uid if uid has the value of the real user-ID or if it has the value of the saved user-ID. If the effective user-ID of a process is super user the Kernel resets the real and effective user-ID fields in the process table and *u*-area to uid. Otherwise the system call return error.

Generally a process inherits its real and effective user id from its parent during the fork system call and maintains their values across exec system calls.

9.4 EXAMPLE EXECUTION OF SETUID PROGRAM

```
main ()
{
int, uid, euid, fdmJb, fdmJb1;
uid = getuid () ;
```

```
    euid = geteuid () ;
    Print f ("uid % d euid % d", uid, euid);
    fdmjb = open ("mjb", O_RDONLY);
    fdmjb1 = open ("mjb1", O_RDONLY);
    print f ("fdmjb = % d, fdmjb1 = % d", fdmjb, fdmbj1);
    setujd (uid);
    print f ("After setuid(%d): uid % d euid % d", uid, getuid(),
geteuid());
    fdmjb = open ("mjb", O_RDONLY);
    fdmjb1 = open ("mjb1", O_RDONLY);
    print f ("fdmjb % d fdmjb1 % d", fdmjb, fdmjb1);
    setuid (euid);
    print f ("after setuid (%d): uid% d euid % d", euid, getuid(),
geteuid()
    }
```

Suppose the executable file produced by compiling the program has owner "mjb1" (ID-8319) its setuid bit on, and all users have permission to execute it. Assumes that user "mjb" (ID-5088) and "mjb1" own the files of their respective names and that both files have read-only permission for their owners. User "mjb" sees following output when executing the program:

uid 5088 euid 8319

fdmjb – 1 fdmjb1 3

after setuid (5088) : uid 5088 euid 5088

fdmjb 4 fdmjbl – 1

after setuid (8319) : uid 5088 euid 8319

user "mjb1" sees the following output:

uid 8319 euid 8319

fdmjb – 1 fdmjb1 3

after setuid (8319) : uid 8319 euid 8319

fdmjb – 1 fdmjb1 4

after setuid (8319) : uid 8319 euid 8319

9.5 CHANGING THE SIZE OF A PROCESS

A process may increase or decrease the size of its data region by using the brk system call. Syntax.

brk (endds);

endds: endds be comes the value of the highest virtual address of the data region of the process (called its break value). A user can call;

oldendds = sbrk (increment);

increment: increment changes the current break value by the specified number of bytes.

oddendds: is the break value before the call.

Sbrk is a C library routine that calls brk.

```
algorithm : brk
input     : new break value;
output    : old break value;
{ lock process data region;
if (region size increasing)
if (new region size is illegal)
{ unlock data region;
return (error);
}
change region size;
zero out addresses in new data space;
unlock process data region;
}
```

9.5.1 The Shell

The shell reads a command line from its standard input and interprets it according to a fixed set of rules. The standard input and standard output file descriptors for the login shell usually the terminal on which user logged in. If the shell recognizes the input string as a built in command, it executes the command internally without creating new processes; otherwise it assumes the command is the name of executable file.

who

grep include * C

ls – l

etc

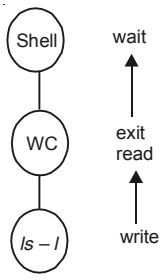


Fig. 9.2 Relationship of processes for ls – l/wc

9.5.2 Process Termination

Processes on a UNIX system terminate by executing the exit system call. An exiting process enters the zombie state relinquishes its resources and dismantles its context except for its slot in the process table. Syntax.

exit (status) :

Status: Status is returned to the parent process for its examination. Process may call exit explicitly or implicitly at the end of a program. The startup routine linked with all C programs calls exit when the program returns from the main function the entry point of all programs. Alternatively the Kernel may invoke exit internally for a process on receipt of uncaught signals. If so the value of status is signal number.

```
algorithm : exit
input     : return code for parent process
output    : none
{ ignore all signals;
if (process group loader with associated control terminal)
{
send hangup signal to all members of process group;
reset process group for all members too.
}
Close all open files;
release current directory;
release current (changed) root, if exist;
free regions, memory associated with process;
write accounting record;
make process state zombie;
assign parent process ID of all child processes to be init process (1);
if any children were zombie, send death
of child signal to init;
send death of child signal to parent process;
context switch;
}
```

The system imposes no time limit on the execution of a process and processes frequently exist for long time. For instance, process 0 (swapper) and 1 (init) exist throughout the life time of a system. And getty processes which monitor a terminal line, waiting for a user to log in and special purpose administrative processes.

Chapter 10

INTER-PROCESS COMMUNICATION

IPC mechanism allow arbitrary process to exchange data and synchronize execution.

We have already considered several forms of interprocess communication, such as pipes, named pipes and signals. Pipes suffer from the drawback that they are known only to processes which are descendants of the process that invoke the pipe system call: unrelated process cannot communicate via pipes. Although named pipe allow unrelated process to communicate, they cannot generally be used across a network nor do they readily lend themselves to setting up multiple communications paths for different sets of communicating process: its impossible to multiplex a named pipe to provide private channels for pairs of communicating process. Arbitrary process can also communicate by sending signals via the kill system call, but the message consist only of the signal number.

10.1 PROCESS TRACING

One process traces and controls the execution of another process. Tracing processes, useful for debugging. A debugger process such as sdb, spawns a process to be traced and controls its execution with the ptrace system call, setting and clearing break points, and reading and writing data in its virtual address space.

Thus process tracing consist of synchronization of the debugger process and the traced process and controlling the execution of traced process. Syntax for ptrace.

ptrace (cmd, pid, addr, data);

cmd → Specifies various commands such as reading data, writing data, resuming execution and so on.

pid → is the process-ID of traced process.

addr → is the virtual address to be read or written in the child process.

data → is an integer value to be written.

When executing the ptrace system call, the Kernel verifies that the debugger has a child whose ID is pid and that the child is in the traced state and then uses a global trace data structure to transfer data between the two processes. It locks the trace data structure to prevent other tracing processes from overwriting it, copies cmd, addr and data into the data structure, wakes up the

child process and puts it into the "ready-to-run" state then sleeps until the child responds. When the child resumes execution (in Kernel mode), it does the appropriate trace command, writes it reply into the trace data structure, then awakens the debugger. Depending on the command type, the child may reenter the trace state and wait for a new command or return from handling signals and resume execution. When the debugger resumes execution, the Kernel saves the "return value" supplied by the traced process, unlocks the traced data structure, and returns to the user.

If the debugger process is not sleeping in the wait system call when the child enters the trace state, it will not discover its traced child until it calls wait.

10.1.1 Drawback of ptrace

(*i*) The Kernel must do context switches to transfer a word of data between a debugger and a traced process. The Kernel switches context in the debugger in ptrace call until the traced process replies to a query, switches context to and from the traced process, and switches context back to the debugger process with the answer to the ptrace call.

(*ii*) A debugger process can trace several child processes simultaneously.

(*iii*) A debugger cannot trace a process that is already executing if the debugged process had not called ptrace to let the Kernel know that it consents to be traced.

(*iv*) Its impossible to trace setuid programs, because users could violate security by writing their address space via ptrace and doing illegal operations.

10.2 SYSTEM V IPC

This package consist of three mechanism:

(*i*) Message (*ii*) Shared Memory (*iii*) Semaphores

They share common property:

I. Each mechanism contains a table whose entries describe all instances of the mechanism.

II. Each entry contains a numeric key, which is its user chosen name.

III. Each mechanism contains a "get" system call to create a new entry or to retrieve an existing one, and the parameters to the calls include a key and flag. The Kernel searches the proper table for an entry named by the key.

Process can call the "get" system calls with the key IPC PRIVATE.

They can set the IPC_CREAT bit in the flag field to create a new entry if one by given key does not already exist, and they can force an error notification by setting the IPC_EXCL and IPC_CREAT flags if an entry already exists for the key.

IV. For each IPC mechanism, the Kernel uses the following formula to find the index into the table of data structures from descriptor;

index = descriptor modulo (no. of entries in table)

For example if the table of message structures contains 100 entries, the descriptors for entry 1 are 1, 101, 201 and so on. When a process removes an entry the Kernel increment the descriptor associated with it by the number of entries in the table.

V. Each IPC entry has a permissions structure that includes the user-ID and group-ID of the process that created an entry.

VI. Each entry contains other status information such as the process-ID of the process to update the entry and the time of last access or update.

VII. Each mechanism contains a "control" system call to query status of an entry to set status information, or to remove the entry from the system.

10.2.1 Messages

Messages allow processes to send formatted data streams to arbitrary process. There are four system calls for messages:

(*a*) msgget—It returns a message descriptor that designates a message queue for use in other system calls. Syntax.

msgqid = msgget (Key, flag);

msgquid—is the descriptor returned by the call.

key—Its a numeric key, which is its user, chosen name. The Kernel searches the proper table for an entry named by the key.

flag—It specifies the action the Kernel should take.

msgqid as an index into an array of message queue headers. The queue structure contains the following fields.

—Pointers to the first and last messages on a linked list.

—The number of messages and the total number of data bytes on the linked list.

—The maximum number of bytes of data that can be on the linked list.

—The process IDs of the last processes to send and receive messages.

—Time stamps of the last msgind, msgrcvs and msgctl operations.

When a user calls msgget to create a new descriptor, the Kernel searches the array of message queues to see if one exists with the given key. If there is no entry for the specified key, the Kernel allocates a new queue structure, initialize it, and returns an identifier to user. Otherwise it checks permissions and returns.

msgsnd (msgquid, msg, count, flag);

msgquid—is the descriptor of message queue returned by a msgget call.

msg—is a pointer to a structure consisting of a user-chosen integer type and a character array.

count—count gives the size of data array.

flag—flag specifies the action the Kernel should take if it runs out of internal buffer space.

The Kernel checks that the sending process has write permission for the message descriptor, that the message length does not exceed the system limit that the message queue does not contain too many bytes, and that the message type is +*ve* integer. If all tests succeed, the Kernel allocates space for the message from a message *map* and copies the data from user space. The Kernel allocates a message header and puts it on the end of linked list of message headers for the message queue.

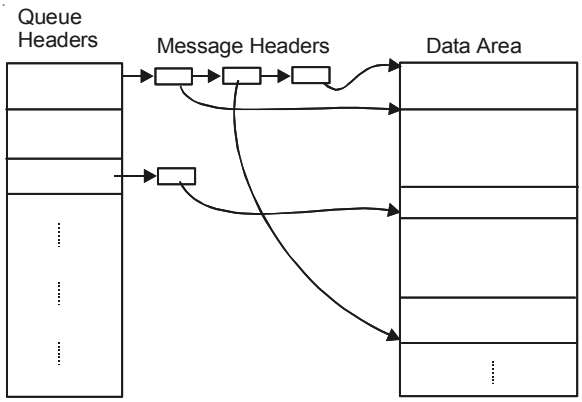


Fig. 10.1 Data structure for messages

A process calls msgget to get a descriptor for MSGKEY. It sets up a message of length 256 bytes, although it uses only the first integer, copies its process ID into the message text, assigns the message type value 1 then call msgsnd to send the message. A process receives messages by:

count = msgrcv (id, mrg, max count, type, flag);

id—Message descriptor

msg—is the address of a user structure to contain the received message.

maxcount—is the size of the data array in mrg

type—specifies the message type the user wants to read

flag—specifies what the Kernel should do if no messages are an queue

count—is the number of bytes returned to the user.

A message can query the status of a message descriptor, set its status and remove a message descriptor with the mrgctl system call. Syntax is:

mrgctl (id, cmd, metatbuf)

id— ...

cmd—type of command

metatbuf—is the address of a user data structure that will contain control parameters or the result of a query.

10.2.2 Shared Memory

Processes can communicate directly with each other by sharing parts of their virtual address space and then reading and writing the data stored in the shared memory. There are various system calls for manipulating shared memory.

(*a*) *shmgt*—This system call creates a new region of shared memory or returns an existing one. Syntax.

shmid = shmget (key, size, flag);

size—is the number of bytes in the region. The Kernel searches the shared memory table for the given key; if it finds an entry and the permissions modes are acceptable, it returns the descriptor for the entry. If it does not find an entry and the user had set the IPC_CREAT flag to create a new region, the Kernel verifies that the size is between system-wide minimum and maximum values and then allocates a region data structure. The Kernel saves the permission modes, size and a pointer to the region table

entry in the shared memory table and sets a flag there to indicate that no memory is associated with the region. It allocates memory for the region only when a process attaches the region to its address space.

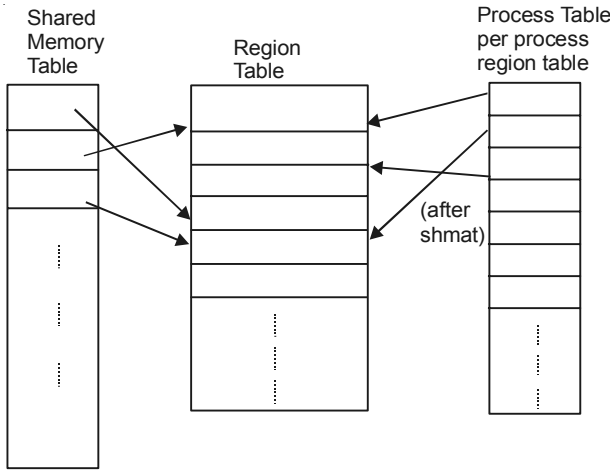


Fig. 10.2 Data structures for shared memory

(*b*) *shmat*—A process attaches a shared memory region to its virtual address space with this system call. Syntax.

vitaddr = shmat (id, addr, flags);

id—Returned by a previous shmget system call, identifies the shared memory region.

addr—is the virtual address where the user wants to attach the shared memory.

flags—specify whether the region is read only and whether the Kernel should round off the user specified address.

virtaddr—is the virtual address where the Kernel attached the region, not necessarily the value requested by the process.

(*c*) *shmdt*—A process detaches a shared memory region from its virtual address space by;

shmdt (addr)

addr—is the virtual address returned by a prior *shmat* call.

The Kernel searches for the process region attached at the indicated virtual address and detaches it from the process address space. Because the region tables have no back pointers to the shared memory table, the Kernel searches the shared memory table for the entry that points to the region and adjusts the field for the time the region was last detached.

(*d*) *shmctl*—A process uses this system call to query status and set parameters for the shared memory region:

shmctl (id, cmd, shmetatbuf);

id—identifies shared memory table entry.

cmd—specifies type of operation.

shmetatbuf—is the address of a user-level data structure that contains the status information of the shared memory table entry when quering or setting its status.

10.2.3 Semaphores

The semaphore system calls allow processes to synchronize execution by doing a set of operations automatically on a set of semaphores. Before the implementation of semaphores, a process would create a lock file with the creat system call if it wanted to lock a resource. The create fails if the file already exists, and the process would assume that another process had the resource locked.

Disadvantages—The process does not know when to try again and lock files may inadvertently be left behind when the system crashes or is reboated.

P : if (S > 0)

S – – ;

V: if (S < 0)

S + + :

S is a semaphore variable.

A semaphore in UNIX system V consists of the following elements.

(*i*) The value of semaphore.

(*ii*) The process ID of the last process to manipulate the semaphore.

(*iii*) The number of processes waiting for the semaphore value to increase.

(*iv*) The number of processes waiting for the semaphore value to equal 0.

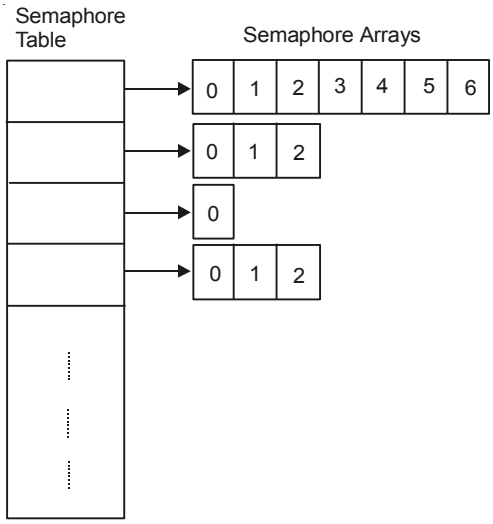


Fig. 10.3 Data structures for semaphores

semget—semget to create and gain access to a set of semaphores. It creates an array of semaphores. Syntax.

id = semget (Key, count, flag);

The Kernel allocates an entry that points to an array of semaphore structures with count elements. The entry also specifies the number of semaphores in the array, the time of last *semap* call and the time of last semctl call.

semop—Process manipulate semaphores with the semop system call.

oldval = semop (id, oplist, count);

id—Descriptor returned by semget.

oplist—is a pointer to an array of semaphore operations.

count—is the size of the array.

oldval—is the value of the last semaphore operated on in the set before the operation was done. The format of each element of oplist is:

(*i*) The semaphore number identifying the semaphore array being operated on.
(*ii*) The operation
(*iii*) Flags

semctl—It contains a myriad of control operations for semaphores:

semctl (id, number, arg):

Chapter 11

SOCKETS

Furthermore, the methods may not allow processes to communicate with other processes on the same *m/c* because they assume existence of a server process that sleeps in a driver open or read system call. To provide common methods for IPC and to allow use of sophisticated n/w protocols, the BSD system provides a mechanism known as sockets.

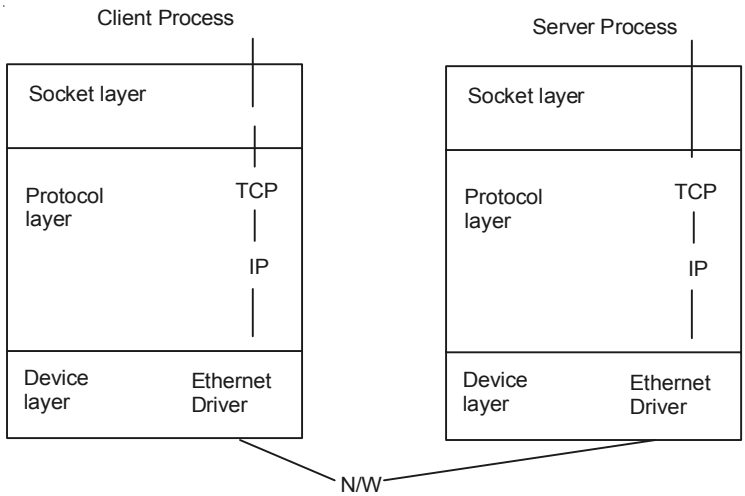


Fig. 11.1 Sockets model

The Kernel structure consists of three parts:

(*i*) *Socket Layer*—Provide the interface b/w the system calls and the lower layers.

(*ii*) *Protocol Layer*—Contains the protocol modules used for communication (*e.g.*, TCP/IP).

(*iii*) *Device Layer*—Contains the device driver that control the n/w devices.

Layer combinations of protocols and drivers are specified when configuring the system. Process communicate using the client server model; a server process listens to a socket one end point of a two way communication path, and client process communicate to the server process cover another socket, the other end point of communications path which may be on another *m/c*. The Kernel maintains internal connections and routes data from client to server.

Socket that share common communication property such as maming convention and protocol address format are grouped into domains. The 4.2.B.S.D system supports the "UNIX system Domain" for process communicating on one m/c. And "Internet Domain" for process communicating across a *n/w* using the DARPA (Defence Advanced Research Project Agency).

Each socket has a type:

(*i*) *Virtual circuit*—V.C. Allows sequenced reliable delivery of data. They are expensive.

(*ii*) *Datagram*—Datagrams do not guarantee sequenced, reliable or unduplicated delivery but they are less expensive because they do not require expensive setup operations.

The socket mechanism contains several system calls:

(*i*) **Socket** system calls establishes the end point of communication link. Syntax.

sd = Socket (format, type, protocol)

sd = Socket Descriptor

format = Specifies the communication domain.

type—Indicates type of communication over socket.

protocol—Indicates a particular protocols to control the communication

/ * close system call closes the socket * /

(*ii*) **bind** System calls associate a name with socket descriptor. Syntax

bind (sd, address, length)

address—*e.g*., file name in UNIX system domain point to a structure that specifies an identifier specific to the communication domain and protocol specified in the socket system call.

Length—Length is the length of the address structure, without this parameter the Kernel would not know how long the address is because it can vary across domains and protocols.

Server processes bind addresses to sockets and "advertise" their names to identify themselves to client process.

(*iii*) **Connect**—It request that the Kernel make a connection to an existing socket connect (sd, address, length)

address—Address of the target socket that will form the other end of the communications line.

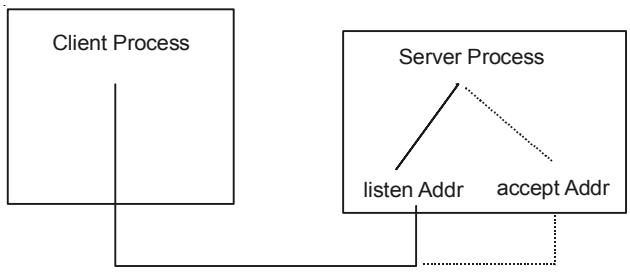


Fig. 11.2 A server accepting a call

When a server process arranges to accept connections over a virtual circuit, the Kernel must queue incoming request until it can service them.

(*iv*) The **listen** system call specify the maximum queue length.

listen (sd, q length)

q length—is the maximum number of outstanding request.

(*v*) **Accept**—It receives incoming request for a connection to a server process:

nsd = accept (sd, address, addrlen)

address—Points to a user data array that the Kernel fills with the return address of the connecting client.

addrlen—Indicates the size of user array.

(*vi*) **Send**—

Count = send (sd, msg, length, flags)

Count — Number of bytes actually sent

msg—Pointer to the data being sent

length—Length of message

flag—This parameter may be set to the value SOF_OOB to send data "out-of-band", meaning that data being set is not considered part of the regular sequence of data exchange b/w the communicating processes.

(*vii*) **recv**—

Count = recv (sd, buf, length, flags)

buf—buf is the data array for incoming data.

Flags—Flags can be set to peek at an incoming message and examine its contents without removing it from the queue or to receive "out of band" data.

(*viii*) **Shutdown**—It close a socket connection shutdown (sd, mode)

mode—Indicates whether the sending side, the receiving side or both sides allow no longer data transmission.

11.1 MULTIPROCESSOR SYSTEMS

A multiprocessor architecture contains two or more CPU that share common memory and peripherals potentially providing greater system throughput, because process can run concurrently on different processors. Each CPU executes independently but all of them execute one copy of the Kernel. Some multiprocessor systems are called attached processor systems, because the peripherals may not be accessible to all processors.

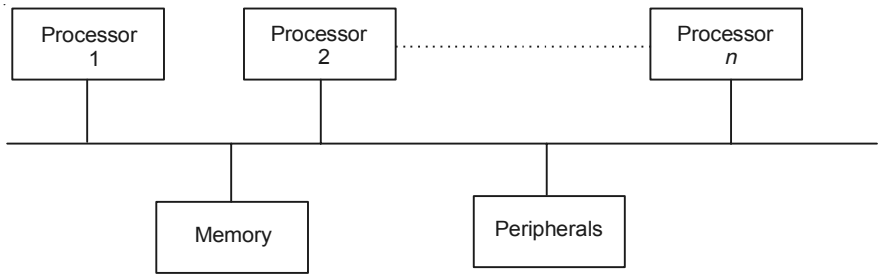


Fig. 11.3 Multiprocessor configuration

11.2 PROBLEM WITH MULTIPROCESSOR SYSTEM

```
struct queue {
} * bp, * bp1 ;
bp1 – torp = bp – torp
bp1 – backp = bp ;
bp – torp = bp1;
/ * consider a context switch here * /
bp1 – torp – balkp = bp1:
```

The Kernel cannot preempt a process and switch context to another process while executing in Kernel mode, and it masks out interrupts when executing a critical region of code if an interrupt handler could corrupt Kernel data structures. On a multiprocessor, however, if two or more process executes simultaneously in the Kernel on separate processor, the Kernel could become corrupt in spite of the protective measures that suffice for uniprocessor systems.

The Kernel must make sure that such corruption can never occur.

There are three methods for preventing such corruption:

I. Execute all critical activity on one processor, relying on standard uniprocessor methods for preventing corruption.

II. Serialize access to critical regions of code with locking primitives.

III. Redesign algorithms to avoid contention for data structures.

11.3 SOLUTION WITH MASTER SLAVE PROCESSORS

Master processor can execute in Kernel mode and slave executes only in user mode. The master processor is responsible for handling all system calls and interrupt. Slave processors execute processes in user mode and inform the master processor when a process makes a system call. /* System with one master and several slave */.

The schedular algorithm decides which processor should execute a process. When a process on a slave processor executes a system call, the slave Kernel sets the PID field in the process table, indicating that the process should run only on the master processor, and does a context switch to schedule other processes. The master Kernel schedule, the process of highest priority that must run on the master processor and executes it. When it finish the system call, it sets the processor ID field of the process to slave, allowing the process to run on slave processors again.

If processes must run on the master processor it is preferable that the master processor run them right away and not keep them waiting. If the master processor were executing a process in user mode when a slave processor requested service for a system call, the master process would continue executing until the next context switch. The master processor respond more quickly if the slave processor set a global flag that the master processor checked in the clock interrupt handler, the master processor would do a context switch in at most one clock tick. Alternatively the slave processor could interrupt the master processor and force it to do a context switch immediately but this assumes special h/w capability.

The clock interrupt handler on a slave processor makes sure that processes are periodically rescheduled so that no one process monopolizes the processor. A side from that the clock handler

"wakes up" a slave processor from an idle state once a second. The slave processor schedules the highest periority process that need not run one the master processor.

The only chance for corruption of Kernel data structures comes in the schedular algorithm, because it does not protect against having a process selected for execution on two processors.

For instance, if a configuration consists of a master processor and two slaves, it is possible that the two slave processor find one process in user mode ready for execution. If both processors were to schedule the process simultaneously, they would read, write and corrupt its address space.

The system can avoid this problem:

(*i*) The master can specify the slave processor on which the process should execute, permitting more than one process to be assigned to a processor. One processor may have lots of processes assigned to it whereas others are idle. The master Kernel would have to distribute the process load b/w the processors.

(*ii*) The Kernel can allow only one processor to execute the scheduling loop at a time.

11.4 SOLUTION WITH SEMAPHORES

Using this method for supporting UNIX systems on multiprocessor configuration is to partition the kernel into critical regions such that at most one processor can execute code in a critical regions at a time. There are two issues:

(*i*) How to implement Semaphores?

(*ii*) Where to define critical Region?

There are various algorithms in uniprocessor UNIX systems use a sleep lock to keep other processes out of a critical region in case the first process later goes to sleep inside the critical region. The mechanism for setting the lock is:

```
while (lock is set) /* Test Operation */
  sleep (condition until lock is free);
  set lock:
```

and the mechanism for unlocking the lock is free lock;

wake up (all process sleeping on condition lock set);

sleep-lock delineate some critical region

But they do not work on multiprocessor systems.

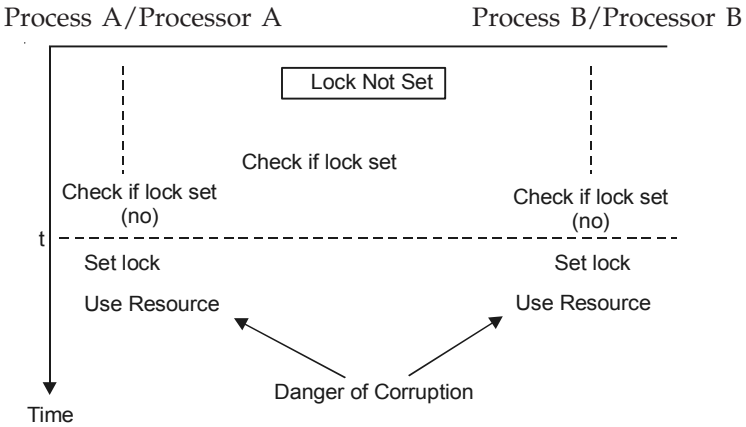


Fig. 11.4 Race condition in sleep-locks on multiprocessor

As illustrated in above figure. Suppose a lock is free and two processes on two processors simultaneously attempt to test and set it. They find that the lock is free at time t, set it, enter the critical region, and may corrupt kernel data structures. To prevent this situation the locking primitives must be atomic. i.e. The actions of testing the status of the lock and setting the lock must be done as a single, indivisible operation, such tha only one process can manipulate that lock at a time.

11.4.1 Definition of Semaphores

A semaphore is an integer valued object manipulated by the kernel that has the following atomic operations defined for it:

(*i*) Initialization of the semaphore to a nonnegative value.

(*ii*) A P operation that decrement the value of the semaphore. If the value of the semaphore is less than 0 after decrementing its value the process that did the P goes to sleep.

(*iii*) A V operation that increment the value of the semaphore. If the value of the semaphore becomes greater than or equal to 0 as a result, one process that had been sleeping as the result of a P operation wakes up.

(*iv*) A conditional P operation, abbreviated CP, that decrements the value of the semaphore and return an indication of true if its value is greater than 0. If the value of the semaphore is less than or equal to the value of the semaphore is unchanged and the return value is false.

As illustrated in Figure 11.4. Suppose a lock is free and two processes on two processors simultaneously attempt to test and set it. They find that the lock is free at time *t*, set it, enter the critical region and may corrupt kernel data structures. To prevent this situation the locking primitives must be atomic *i.e.*, the actions of testing the status of the lock and setting the lock must be done as a single, indivisible operation, such that only one process can manipulate the lock at a time.

Chapter 12

UNIX COMMAND

12.1 INTRODUCTION TO SHELL

The Shell is a program that provides an interpreter and interface between the user and the UNIX Operating System. It executes commands that are read either from a terminal or from a file. Files containing commands may be created, allowing users to build their own commands. In this manner, users may tailor UNIX to their individual requirements and style.

There are a number of different Shells. Each provides a slightly different interface between the user and the UNIX Operating System. There are three important types of shell in UNIX these are:

(*i*) Bourne Shell

(*ii*) C-Shell

(*iii*) Korn Shell

There are other shells that are less widely used and not available on many machines. A command issued by a user may be run in the present shell, or the shell itself may start up another copy of itself in which to run that command. In this way, a user may run several commands at the same time. A secondary shell is called a sub-shell.

When a user logs onto the system, a shell is automatically started. This will monitor the user's terminal, waiting for the issue of any commands. The type of shell used is stored in a file called passwd in the subdirectory etc. Any other shell may be run as a sub-shell by issuing it as a command. For example, /usr/bin/ksh will run a Korn shell. The original shell will still be running—in background mode—until the Korn shell is terminated.

There are three prompt in UNIX/Linux:

(*i*) # It is used by super user or System Administrator.

(*ii*) $ Its used by ordinary user.

(*iii*) % Its used by ordinary user. But when work with c-shell

Some super user related command.

1. Add a new user

Syntax:

useradd [OPTIONS] [USERNAME]

2. Change or assign password for a user

   #passwd <username>

3. ADD a group

   #groupadd <groupname>

4. DELETE

   (*i*) USER #userdel <username>

   (*ii*) #groupdel <groupname>

5. MODIFIED

   (*i*) GROUP #groupmod <groupname>

   (*ii*) USER #usermod <username>

OPTIONS with Useradd Command

-c Comment → The new user password comment field

-f Inactive days → The number of days password expire until the account is permanently disabled. A value 0 is disabled the account as soon as password has expire, and a value of –1 disabled the features. The default value is –1.

-m → The user HOME directory will be created if it does not exist. The files contained in skeleton_dir will be copied to the home directory if the –k option is used. Otherwise the files contained in /etc/skell will be used instead. Any directory contained in skeleton_dir or /etc/skell will be created in the users home directory.

-u uid → The numerical value of user-ID. This value must be unique unless the –o option is used. The value must be non negative. The default is to use smallest ID value greater then 99 and greater then other user-ID. Values between 0 and 99 are reserved for system accounts.

-M → The user HOME directory will not be created.

Example: # useradd –u 120 –g g1 –c "SQL" –d /home/anoop –s /bin/bash –m anoop

Password Administration

The following example sets the minimum and maximum number of weeks for change of password for user Anoop.

```
#passwd –n 12 ANOOP          /* Minimum 12 weeks*/
#passwd –x 12 ANOOP          /* Maximum 12 weeks */
```

Allowing User to Shutdown Only

First create a ordinary user account with useradd.

$ useradd –u 123 –g MCA –d /home/Anoop –m Anoop

Now you have to confer root status on this user by changing the UID in /etc/passwd from 123 to 0 the one used by root. The shutdown command must placed in the user .profile so that he/she can't do anything else.

Internal and External Command

Command or file having an independent existence in the /bin (or /usr/binj) directories an external command. Otherwise its an internal command. To know command type.

$type ls /*ls is /bin/ls*/

Command

ls – It will list directory and file. Syntax:

ls [options] filename.

Options

-l → Long listing file and directories. With this option the command will display following types of information.

File-type, Permission, Number-of-Link, Owner, Group-Owner, File-size, File-modification time, File-name.

-a → Show all files (Including hidden files) -i → Display inode number.

-A → All files but not **.** and **..**

$ ls ab* /*Start with ab*/

$ls ?ab /*Start with a single character*/

$ ls[abc]* /*First letter must any one of the letter given in []*/

$ ls[abc]* /*Not start with abc*/

$ls [a-d][c-m][2-9]?? /*List all files whose first letter a to d second letter c to m and third 2 to 9 and any two other character */

$ls **.txt** /*List all files with extension txt/

File Type	Meaning
-	Ordinary File
d	Directory File
b	Block Special File
c	Character Special File
p	Named Pipe
l	Symbolic Link
s	Semaphore
m	Shared Memory File

File permission: → These are three types of file permission

Permission	Weight
Read (r)	4
Write (w)	2
Execute (x)	1

A file consist of rwxrwxrwx in which first rwx for owner second for group owner and last three for other owner.

Change permission of a file → File permission can be change either by owner of the file or super user not by the other user.

Chmod (Change mode of a file) By this command we can change permission of a file. Syntax→

$ chmod [WHO] [+/−/=] [Permission] <File name>

WHO means **u** → For user, **g** → For group user, and **o** → for other user **+** → add permission, **—** → Remove permission , = → Instruct chmod to add the specified permission and take away all others, if present.

$ chmod 777<File name> /*It will change all the permission of a file*/

Create Directory → A directory created by mkdir command. SYNTAX: $ Mkdir <Directory name>

Option → -m → Set permission mode.
-v → Print a message for each created directory.
-p → Make parent directory as needed.

cat command → Create/display/append a file

$ cat >f1 /* Create the file f1 if f1 is an existing file then overwrite the contents of f1*/
$ cat >>f1 /* Append content in f1 if f1 does not exist then create it*/
$ cat f1 /* Open f1 if f1 not exist then report error*/
$cat f1>f2 /* Copy f1 to f2 if f2 exist then overwrite the contents*/
$ cat f1>>f2 /* Appends the content of f1 into f2 if f2 not exist then create it */
$ cat >a* /* Error (Try to create a file a*)*/
$cat >a* /* The file a* will be created*/

Remove file and directory

Ordinary file →

$rm <file name> /*Remove a ordinary file */
$rm a* /*Remove all file start with name a */
$rm a* /*Remove file name a* */

Directory File →

$rm –r <directory name> /*Remove a non-empty directory also*/

$rmdir <directory name> /*Remove only if directory is empty */

Date Command → date command print current date and time and date in a variety of formats. Its also used by system administrator to set system time.

$ date

Day_of_week Month Date Time IST Year /*By default printed */

Options →

Name	Description	Name	Description
%a	Local Abbreviated weekly name name(Sun,Mon,……)	%l	Month (01,…….23)
%A	Full Weekly name(Sunday….)	%m	Month (01……..12)
%b	Abbreviated month name(Jan,...)	%M	Minute (00…….59)
%B	Full month name	%n	A new line
%c	Local date and time	%N	Nanosecond
%C	Century(Year divided by 100)	%p	AM or PM indicator
%d	Day of month	%P	Am or pm indicator
%D	Date (DD/MM/YY)	%r	Time 12 hour(hh:mm:ss[AP]M)
%e	Day of month blank padded(1,..31)	%R	Time 24 hour (hh:mm)
%F	Same as %Y-%m-%d	%t	A horizontal tab
%g	Two digit year corresponding to %v week number	%U	Week number of year with Sunday as first day (00……..53)
%G	Four digit year corresponding to %v week number	%u	Day of week (1……7)
%W	Week number of year with Monday as first day of week(00,…53)	%V	Week number of year with Monday as first day.
%H	Hour (00,……23)	%T	Time 24 hour(hh:mm:ss)
%I	Hour (01,…12)	%w	Day of week (0…..6) 0 is sunday
%j	Day of year(001,….366)	%h	Abbreviated month name (Jan,..)
%k	Hour (0,…23)	%x	Local date (mm/dd/yy)
%X	Local time(%H%M%S)	%y	Last two digit of year
%Y	Year (2002)		

cal → Display a calendar

$cal /*Display calendar for current month of current year*/

Options: →

-l → Display single month

-3 → Display Prev/Current/Next month of current year

-s → Display Sunday as first day

-m → Display Monday as first day

-j → Display Julian date

-y → Display calendar for current year

Clear Command → Clear the terminal screen. It looks in the environment for the terminal type and then in the terminfo database to figure out to clear the screen.

Terminfo → /usr/share/terminfo/*/*

Terminfo is a database describing the terminal, used by the screen oriented programs. It describe terminals by giving a set of capabilities which they have, by specifying how to perform screen operations.

tput → tput initialize a terminal or a query terminfo database. The tput utility use the terminfo database to make the values of terminal dependent-capabilities and information available to shell.

```
$ tput 10 20          /*Send cursor at 10th row and 20th column */
$ tput clear          /*Echo the clear screen sequence for the current terminal*/
$ tput cols           /*Print number of column for the current terminal */
$ tput –T 450 cols    /*Print number of column for 450 terminal */
$ tput smso           /*Bold Character */
$ tput rmso           /*Off bold character */
$tput logname         /*Print logname from terminfo */
```

Tput process several capabilities. For example

```
$ tput –S <<!
>clear
>cup 10 20
>smso
>!
```

more → To see page wise output on screen

$ more –n /*Where n is a integer number and it's the screen size*/

–d → /*More will prompt the user with message (Press space to continue, q to quit)*/

+num → Start at line number (num)

–p → Clear hole screen and display text

wc → Print the number of bytes, word and line in a file.

Wc [options] [filename]

Options →

–c → Print the byte count, –m → Print the character count

–l → Print new line count –L → Print the length of longest line

–w → Print word count

banner → Print large banner on printer (Not use in Linux)

pwd → Print name of current working directory

who → Show who is logged on

$who [OPTIONS] [FILE | ARG1 ARG2]

Options →

–H → Print line of column headings

–i,–u → Add user idle time as HOUR: MINUTE :… or old

–m → Only hostname and user associated with stdin

–q → Count all login names and number of users log on.

finger → It will display following type of information about all users who is currently log on.

Login-name tty idle login-time office office-phone

echo → Display a line of text

Options →

–n → Do not output the trailing new line

–E → Disable interpretation of those sequence in string's.

–e → Enable those sequence of the backslash escaped character listed below:

\NNN → The character whose ASCII code is NNN (Octal)

\\ → Backlash, \v → Vertical tab, \r → carriage return \a → Alert

\b → Backspace, \n → New line, \t → Horizontal tab

cp → Copy source to destination.

cp [OPTIONS]source-pathname destination-pathname

Options →

–i → Interactive copy, –l → Link instead of copy, –v → Verbose

–R → Copy directory recursively, –H → Follow command line symbolic link

–d → Never follow symbolic link, –r → Copy recursively non directory as files

–u → Update copy only when source file is newer then destination file or destination file missing.

mv → Move (Rename) fil. Syntax is same as in cp command

Options →

–f → Never prompt before overwriting, –i → Interactive

file → Knowing file type. Syntax:

file [OPTIONS] [Name-of-file]

$ file * /*Display all file type in current directory*/

Options: –b → Do not prep end file name to output lines,

–z → Try to look compressed files

–v → print version of program and exit

bc → bc is a language that supports arbitrary precision with interactive execution of statements. bc starts by processing code from all the file listed on the command line in the order listed. After all files have been processed, bc reads from standard input.

Syntax: bc [OPTION] files –h → Print usage and exit, –i → Force interactive mode, –l → Define standard math library, –w → give warning for extensions to POSIX bc, –s → Process exactly the POSIX bc language, –q (quiet) → Do not print the normal be welcome, –v → print version number and copyright and quiet.

Variable →

ibase → Set input base default is 10,

obase → Set output base default is 10,

scale → Number after point,

last → (an extension) is a variable that has the value of last printed number.

/*bc comments */ # Single line comments

A simple variable is just a "name". An array variable is specified by name[expr].

–expr → result is negation of expression, ++var → Variable is incremented by one, – – var, var++, var– – , expr+expr → Sum of expression (*,/,^,-,%), var = expr,

Relation expression → <, >, ≤, ≥, =, = =, !=, !.

Boolean expression → &&, ||.

Precedence →

|| → Operator left associates

&& → Operator left associates .

Relation operator → Left associates,

Assignment operator → Right associates,

+,– → Operator left associates,

*,/,% → Operator left associates,

^ → Operator right associates,

Unary → Operator non associates,

++,— → Operator non associates,

Standard function →

length(expression) → Return length,

read() → read a number from standard input,

scale(expression) → The value of this function is the number of digits after decimal point in the expression.

sqrt(expression) → The square root of expression,

print → (list) Provide another method of output

print "\n ANOOP CHATURVEDI"

{Statement list} → It allows multiple statements to be grouped together for execution.

if (expression) stat1 [else stat2]

while(expression) statement,

for(exp1;exp2;exp3) {statement}, break →……., continue →……..

halt → This is an executed statement that cause the bc processor to quiet only when its executed.

return(expression)

Function → A function is defined as follows:

define name(parameter) {newline, auto_list, statement_list}

call → name(parameters)

define d(n){return(2*n);}

Array → Array are specified in parameter definition "name[]"

tty → Know your terminal type.

$tty // It print /dev/tty01,

–s → Print nothing only return on exit status.

uname → Print System information

$uname /* Print name of kernel*/

–a → Print all information,

–s → Print kernel name,

–v → Print kernel version,

–n → Print network node hostname,

–r → Print kernel release

lock → Lock your terminal (Not implemented in Linux 2.0 kernel)

$lock /* Remain locked for 30 minutes*/

password…….. Reenter password……

$lock –50/* Lock for 50 minutes(Lock not exceeding 60 minutes)*/

script → Record your login session in a file "typescript'

To exit from this session execute the following command

$exit /*Script is done file is typescript*/

$script –a /*Append activities to existing file typescript*/

$script anoop /*Activities to file anoop*/

spell and ispell → Check your spelling

$spell <filename> /*What is wrong */

$ispell <filename> /*Where is wrong correct it*/

factor → Factors a number(Print prime factor of each number)

$factor 30 /*2 3 5*/

Pattern Searching(in vi editor) →

/unix /*use / for search unix backward */

?unix /*use ? for search unix in forward*/

split: Splitting a file into multiple files → By default split 1000 lines.

$split –5 f1 /*Split each file into 5 lines, it creates a group of files xaa xab .. then xba ..*/

cmp: Comparing two files →

$cmp f1 f2/*f1, f2 differ byte 15 line 2.*/

–l →(list) Gives detail list of byte number and differing byte in octal for each character that differ in both files.

comm → Finding what is common.

–1 → Suppress lines unique to left file,

–2 → Suppress lines unique to right file

–3 → Suppress lines that appear in both files.

diff → Find difference between two files.

–a → Treat all files as text and compare them line by line even if they do not seen to be text, –B → Ignore changes that just insert or delete blank lines,

–b → Ignore changes in amount of white space.

$diff f1 f2

0a1	/*Append after line 0 of first, */	>abc	// File this line
2c3,4	/*Change line two in first file */	<cde	//Replace this line with
>cde			
>dfg	// These two lines		
4d5	//Delete line 4 of first file,	<ceo	// Containing this line

Pattern Matching →

* → Zero or more character ? → A single character

When wild –card lose there meaning

'*' , '?' → Inside the class [*a*], '!' and – → Outside the class ![x-z]x-[y-z]

Redirection →

1. **Standard input:** The command consider its own input as a stream. This stream can come from
   (*i*) The keyword. This is the default source,
   (*ii*) A file (Using feature called redirection)
   (*iii*) Another program (Using pipeline concept).
2. **Standard output:** It also has three similar destination.
   (*i*) It can directed to the terminal,
   (*ii*) It can directed to a file.
   (*iii*) It can serve as input to other program.
3. **Standard Error:** It includes all error message written to the terminal. This output may be generated either by the command or by the shell, but in either case the default case is terminal like standard output it can also assigned to a file. 2 is used for standard error if a file 'aa' does not exist then

$ cat aa >f1 //Error $cat 2>f1 //Send error to file f1

$ cat aa 2>/dev/null //Store result in a file with size NULL.

$ cat aa 2>/dev/tty // Send error message to a particular terminal type.

< and 0< is used for standard input, > and 1> is used for standard output

2> is used for standard error.

Pipes → ' | ' 0< and 1> can be manipulated by shell. If that be so can't the shell connect these stream together, so that one command takes input from other. The pipes takes input form its left hand side command and gives output to its right hand side command.

Tees → 'tee' uses standard input and standard output which means that it can be placed anywhere in a pipeline. The additional feature it posses is that it breaks up the input into two components is saved in a file and the other is the connected to the standard output.

The Environment Variable →

1. System Variable →

(*i*) The set display a complete list of these variable.

(*ii*) PATH → PATH is variable that instruct the shell about the route it should follow to locate any executable commands.

(*iii*) HOME → When you login UNIX place you in a directory called home or login directory.

(*iv*) IFS → Ifs contains a string of characters that are used as word separators in the command line.

(*v*) MAIL → MAIL determine where all incoming mail addressed to this user to be stored.

(*vi*) PS1 & PS2 → The shell has two prompt in PS1 and PS2. The primary prompt string in PS1 as you see ($) and in PS2 (>) how a milt line command makes the shell respond with a >.

If $PS1="ANOOP", // So now prompt ANOOP in place of $sign.

(*vii*) SHELL → SHELL determine the type of shell that a user sees on logging in.

(*viii*) TERM → Indicate terminal type being used.

(*ix*) LOGNAME → Show your log name.

(*x*) .profile → The script executed during login time.

(*xi*) sty → Setting terminal characteristics. (–a → Display all current setting),

$sty –echo //Off $sty echo //On

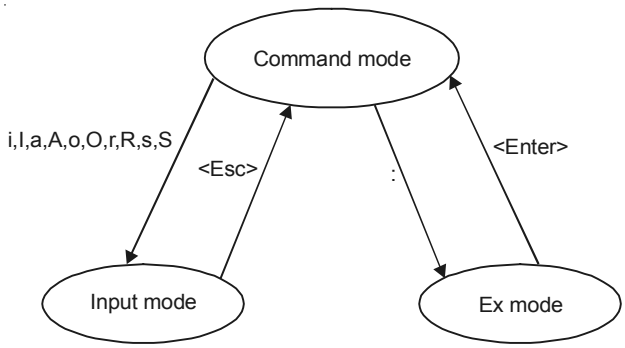
(*xii*) intr → Changing the interrupt key, $ stty \^c

(*xiii*) eof → Changing End-of-file key $ stty eof \^a

Aliases → Aliases are listed when alias is used without argument.

$ alias l='ls –l' and $unalias

Vi Editor → vi editor is an text editor. Modes of vi editor:



(*i*) **Input mode →** Where any key depressed is entered as text. Input mode to command mode press <ESC> key. The default mode is command mode.

i → Insert text to left of cursor, I → Insert text to beginning of line.

a → Append text right of cursor, A → Append text at end of line.

o → Open line below, O → Open line above,

S → Replace entire line, r 'ch' → Replace single character under cursor,

R → Replace text from cursor to right.

s → Replace single character under cursor with any number of character

(*ii*) **Command mode →** Where keys are used as commands to act on text. Command to ex mode press (:). The meaning of different keys work in command mode:

x → Delete a single character, dd → Delete a single line,

h → Moves cursor left(←), l → Right(→),

k → Up(↑), j → Down(↓),

b → Moves back to beginning of word,

e → Forward to end of word, w → Forward to beginning of word,

G → Moves cursor to a particular line number,

J → joining lines, . → Repeat last command.

(*iii*) **Ex mode →** Where ex mode command can be entered in the last line of screen to act on text. In this mode the operation is:

W	Saves file and remain in editing mode	x	Save file and quit editing mode
Wn2w.p	Like save as in Microsoft Windows	.w f1	Write current line to file f1
q!	Quits editing mode but after abandoning changes	q	Quits editing mode where no made to file
n1,n2wf1	Writes file n1 to n2 to file f1	$w f1	Write last line to file f1
sh	Escape to the UNIX shell	wq	Save files and quit editing mode

: set show mode → Show the particular mode in which you are working.

Search and replaces → :1,$s /anoop/ANOOP/g // All lines g(global search)

1→ For first, $→ For last

copying and moving text →

yw → Yank a word, y$ → Yank to end of line, y) → Yank to end of sentence,

y} → Yank to end of paragraph, y]] → Yank to end of section,

yy or Y → Yank to current line Y} → Yank line to the end of paragraphs.

Paste → p→ Puts the yank text to right of cursor,

P→ Puts the yank text to left of cursor,

Simple Filters → Some commands each of which accept some data as input performs some manipulation on it, and produce some output. Since they perform some filtering action on it, and produce some output. Since they performs some filtering action on the data they are appropriately called filter.

pr → Paginating output : → The pr command prepare a file for printing by adding suitable headers, footers and formatted text.

$ pr <filename>

july 31 10:30 2007 <filename> page1

Its often used as a preprocessor before printing with the lp command.

$pr f1 | lp //lpr in Linux

request id 112

By default page size with pr is 66 lines which can be changed with –1 option.

$ pr –1 50 f1 //Page set to 50 lines, $pr +20 f1 //Start printing from page 20.

–k(integer) → Output in more than one column,

–d → Double space the output,

–D → Use format for header date,

–n → Number lines (Counting)

–h → Use a centered header instead of filename in page header,

–h “”→ Print a blank line.

–N number → Start counting with numberat 1st line of 1st page printed,

–o Margin → Offset each line with margin(zero) spaces,

–t → Omit page header and trailer,

–r → Omit warning when a file can not be opened.

–w → Set page width to page_width(72) character for multiple text column o/p page only.

head → Display the beginning of a file. Syntax:

$ head [OPTION] <filename>

$ head f1 //Print first 10 lines of file f1.

–c SIZE → Print first SIZE bytes, –n → Print 1st n lines instead of 10,

–v → Always print header giving filename, –q → Never print header giving filename.

tail → Display end of file(By default 10 lines from end). Syntax:

$tail [OPTION] <filename>

–cN → O/P last N bytes, –n → O/P last n lines, –v → O/P header,

–q → Never O/P header, +n → O/P from n line to end of file,

–f → O/P appended data as files grows ($ prompt does not return).

cut → Slitting a file vertically.

Syntax: $cut [OPTION] <filename>

$cut –c (Represent column) –4,5,7,15- f1 //Cut column 1–4, 5, 7 and 15 to end of file.

$cut-d \| (or"|"(delimiter)) –f (field) 1,3- f1

–b → O/P only these bytes, –c O/P only these character, N– → From Nth byte,\

–n → with –b : Do not split multi byte character, –N → From 1 to Nth,

–s → Do not print line not containing delimiter, N–M → From N to M.

paste → Pasting files(Merge lines of files). Syntax : $paste [OPTION]<filename>.

–d(delimiter) → Reuse character, –s(Serial) → Paste one file at at6ime,

– → Reverse order

$paste –d \| f1 f2 //Paste delimiter between f1 and f2

sort → Ordering of file

–b → Ignore leading blank,–f(ignore case) → Fold lower case to upper case character,

–d(dictionary order) → Consider only blanks and alphanumeric character.

–r → Reverse the result of comparison, –c → Check whether i/p is sorted, do not sort,

–t → Field separator, write result to file instead of standard o/p,

–m → Merge already sorted files do not sort, +2 → Sort on 3rd field,

–3 → Stoppage of sorting after 3rd field (+4.5 –4.8 → Sorting 6th column of 5th field and terminate after 8th column of 5th field),–z → End lines with 0 bytes, not new line,

–u(unique) with –c → Check for strict ordering, Otherwise o/p only the 1st of an equal run.

Regular Expression and Grep Family

grep → Searching a pattern in a file grep [OPTIONS] pattern file

$ grep Anoop f1$grep "ANOOP CHATURVEDI" f1

$ grep ANOOP CHATURVEDI //Error

–v	Count non-matching lines(Invert)	–A N	Print N lines of trailing context after matching lines(---)
–n	Display line number	–R, –r	Read all files under each directory recursively.
–I	Ignore case	–C N	Print N lines of o/p context
–f	Obtain pattern from file one per line	–x	Select only those lies containing match whole lines

–H	Print file name for each match	–B N	Print N lines of leading context before matching lines(---)
–l	Display file name only	–q	Quiet: Do not write anything to standard output.
–e	Match more than one pattern only	–s	No message(File not exist)
–L	Files without match	–o	Show only part of a matching line that match pattern

A R.E. may be followed by one of several repletion operators

? The preceding item is optional and match at most once.

The preceding item is optional and match 0 or more times.

[pqr], [c1-c2],[^pqr]

^pat → Match pattern pat at beginning of line, And $ pat → At end of line.

$grep –e "ANOOP" –e "AMIT" –e "KUMAR" f1

$grep "[aA]gg*[ar][ar]wal" f1

Select those lines where salary lies between 7000 and 7999 $grep "7...$" f1

egrep → Extending grep(More than one pattern for search)

ch+ → Matches one or more occurrence of ch

ch? → Matches zero or more occurrence of ch

EXP1|EXP2 Matches expression EXP1 or EXP2

(x1|x2)x3 → Match x1x3 or x2x3

fgrep → Multiple string searching (Do not accept regular expression)

$ fgrep –f f1 f2

12.2 SHELL PROGRAMMING

The shell has a whole set of internal commands that can be stringed together as a language, with its own variables, conditionals and loops. External command can also be used as a control command for any of the shell constructs. Shell program run in interpretive mode *i.e.*, one statement at a time.

Shell Script: When a group of commands has to be executed regularly they are stored in a file. All such files are called script, or shell program. Or shell procedures.

Execution of shell script: Type the shell script on text (vi) editor. Then we can directly execute the shell script from line editor by the command "sh a1.sh" where a1.sh is the name of your shell script or direct type a1.sh in the ed (or line) editor but first change the mode of your shell script to execution mode.

1. Write a shell script to display the prev/current/next year calendar.

   $ cat a1.sh

   echo "The Calendar is"; cal-3

2. Write a shell script for search a pattern Anoop in afile f1

   $ cat a3.sh

   echo –e"\n Enter The pattern"; read pname

   echo –e "\n Enter The filename"; read flname

   grep "$pname" $flname

   echo –e "\n The selected pattern is given above"

   Parameter used by shell script in command line:

Parameter	Significance	Parameter	Significance
$1, $2 etc	Positional Parameter	$*	Complete set of Positional parameter as a single string.
$#	No. of arguments	$0	Name of executed command
$?	Exit status of last command	$!	PID of last background job

$@ - Same as $* except when enclosed in "" double quotes.

The logical operator- && (and) || (or)

Exit – Script termination. Used with argument 0 for true and other for false

Exit status of a command - $ grep "ANOOP" a1.txt >/dev/null; echo $?

If 0 – Pattern ANOOP found in a1.txt

If 1 – Pattern ANOOP does not exist

If 2 – File does not exist or permission denied

Variable in UNIX

(*i*) UNIX defined or system variable – These are standard variable which are always accessible. EX-PSI, PS2, TERM……

(*ii*) User defined variable – These are defined by us and used most extensively in shell programming.

IF Statement

If (condition); then; statement; else; statement if.

We can also use if…. elif

Numeric comparison with test:

–eq	Equal to	–ne	Not Equal	–gt	Greater than
–ge	Greater or equal	–lt	Less than	–le	Less or equal

3. Write a shell script for search a pattern in file by using command line argument $cat a1.sh

   If [$# –ne 3]; then ; echo –e "\n Not three argument "; exit2

   elif grep "$1" "$2">$3 2>/dev/null; then ; echo "Pattern found";

   Else ; echo "Pattern not found " ; rm $3;fi

Test	Exit Status	Test	Exit Status
–n stg	True if stg is not null	–z stg or !–n	
S1= s2	True if string s1=s2	Stg	True If stg not null
S1!=s1	True if string s1 not equal s2	–a (AND)	–o (Same as OR)

Test with File:

–e	True if file exist	–f	True if file exist and regular
–r	Readable	–w	Write able
–x	Executable	–d	Directory
–s	Size>0	–b	Block
–c	Character	–g	Exist and set group-ID
–h	Symbolic link	–1 stg	Evaluate the length of string stg
–p	And is named pipe	–O	And is owned by effective
–k	And Sticky bit is set	–L	Symbolic link (Same as –h)

Case: Evaluate one of several scripts, depending on a given value.

Case expression in; pat1) statement1; statement2;; pat2)-----;; esac

EX- case "$choice" in

[yY][eE]* ;;

[nN][oO] exit 1;;

*) echo "Invalid option";;

esac

While: Execute script repeatedly as long as condition meet.

while test condition; do; statement; done

Until: Compliment of while

For: Looping with a list

For variable in list; do; Execute statement; done

For file in *.c; do cc –o $ file {x} $file; done

$@- This is same as $* but if you use multiple argument in command line so $* read as a single argument and $@ read as multiple argument.

Expr – Evaluate expression. Expression may be

ARG1 | ARG2 – ARG1 if its neither null or 0 otherwise ARG2.

ARG1 and ARG2 –ARG1 if neither argument is null or 0 otherwise 0

ARG1 < ARG2, ARG1 > ARG2, ARG1 = ARG2, ARG1 ! ARG2, ARG1 + ARG2, - , * , / , %

Substr – substr STRING pos Length - Sub string of STRING, pos counted from 1.

Index – index STG Char - Index in STG Where any CHARS is found or 0.

Length – length STG - Length of STG.

12.3 SLEEP AND WAIT

$sleep 10; echo "10 second have elapsed" # Message will be print after 10 second

$wait – Wait for completion of all background process

wait 224 – Wait for completion of PID 224

basename: Changing file name extension

mv*.txt*.doc # Last argument must be a directory when moving multiple files when basename is used with second argument it strips off the string from first argument.

$basename a1.txt txt -a1. # txt stripped off

for file in*.txt; do

1= basename $file txt ; mv $file ${1}doc; done

chown: Change owner of a file, $chown user file – The owner for file has been changed

chgrp: Change group of a file

Listing by modification and access time:

ls – lt # The time of last modification 1s – lut # The time of last access

touch – changing time status and also create a empty file. (When used without option it change both time) -m → Modification -a → Access Time

touch 03171430 file (Change time) $ touch file → Create a empty file

Major and minor number in device:

#1s – 1 /dev

brwxrwxrwx 1 root root 2, 0 jul 1 23:14 fd0

The 5th column does not show the file size in bytes but rather pair of two numbers separated by a comma. The numbers are called major and minor number. The minor number indicates the special characteristics of device.

Directory

(*i*) **Read permission:** For a directory means that the list of file names stored in directory is accessible. If a directory has read permission you can use is to list out its contents.

(*ii*) **Write permission:** The presence of write permission for a directory implies that you are permitted to create or remove files in it.

(*iii*) **Execute permission:** Execution privilege of a directory means that a user can "Pass through " the directory in searching for sub_directory. For Ex- Cat/usr/anoop/chaturvedi/d1/a1.sh #You need to have execute permission for each of directories involved in the complete path name.

Example

1. *Write a shell script to print all the prime number from X1 to X2.*

```
cat  >prime.sh
echo  "Enter  lower  Limit"
read  x1
echo  "Enter  Higher  Limit"
read  x2
while  [  $x1  -le  $x2  ]
do
    i=2
while   [  $i  -le  $x1  ]
do
    if   [  `expr  $x1  %  $i'  -eq  0  ]
    then
      break
    else
      i=`expr  $i  +  1'
    fi
done
    if  [  $i  -eq  $x1  ]
    then
      echo  $x1
    fi
```

2. *Write a shell script which receives any year through keyboard and determine whether the year is leap or not. If no argument supply the current year should be assumed.*

```
cat  >leap.sh
year=0
echo  "Enter  Year:"
read  year
if  [  year  -eq  0  ]
then
    da=`date  "+%Y"'
    year=$da
fi
if  [  `expr  $year%400'  "eq  0  ]  -o  [  `expr  $year%4'  -eq  0  -a
`expr  $year%100'  -ne  0  ]
```

```
then
    echo "The Year $year is Leap Year"
else
    echo "The Year $year is Not Leap Year"
fi
```

3. *Write a shell script which receives two filename as arguments. It should check whether the two files contents are same or not. If they are same then second file should be deleted.*

```
cat >checkfile.sh
echo "Enter First File Name:"
read f1
echo "Enter Second File Name:"
read f2
if cmp $f1 $f2
then
    echo "The Files are Same"
    rm $f2
else
    echo "The Files Content are not Same"
fi
```

4. *Write a shell script to print all the Armstrong number from X1 to X2.*

```
cat >arm.sh
echo "Enter Lower Limit:"
read x1
echo "Enter Higher Limit:"
read x2
old=$x1
while [ $x1 -le $x2 ]
do
sum=0
while [ $x1 -gt 0 ]
do
    r=`expr $x1 % 10'
    sum=`expr $sum + $r \* $r \* $r'
    x1=`expr $x1 / 10'
done
    if [ $old -eq $sum ]
```

```
    then
      echo  $sum
    fi
      old=`expr  $old  +  1'
```

5. *Write a shell script to print sum of digit of a number. Number entered through keyboard.*

```
cat  >sumofdigit.sh
echo  "Enter  Number:"
read  n
no=$n
sum=0
while  [  $n  -gt  0  ]
do
    r=`expr  $n  %  10'
    sum=`expr  $sum  +  $r'
    n=`expr  $n  /  10'
done
    echo  "Sum  of  Digit  $no  is:  $sum"
```

6. *Write a shell script to print all number from 1 to 10 in same row.*

```
cat  >printno.sh
i=1
while  [  $i  -le  10  ]
do
    echo  -n  "$i"
    i=`expr  $i  +  1'
done
```

7. Write a shell script calculate the factorial of a number.

```
cat  >fact.sh
echo  "Enter  Number:"
read  n
no=$n
f=1
while  [  $n  -ge  1  ]
do
    f=`expr  $n  \*  $f'
    n=`expr  $n  -  1'
done
echo  "The  Factorial  of  $no  is:"  $f
```

8. *Ramesh basic salary is input through the keyboard. His dearness allowance is 40% of basic salary, and house rent allowance is 20% of basic salary. Write a shell script to calculate his gross salary.*

```
cat >grosspay.sh
echo "Calculation of Gross Salary of Ramesh"
echo "Salary of Ramesh is:"
read bs
d=`expr $bs \* 40'
h=`expr $bs \* 20'
da=`expr $d / 100'
hra=`expr $h / 100'
gs=`expr $bs + $da + $hra'
echo "Gross Salary of Ramesh Is:" $gs
```

9. *Write a shell script which will receive either the filename with its full path during execution. This script should obtain information about this file as given by is – l and display it in proper format.*

```
cat >fileinfo.sh
echo "Enter File Name:"
read file
if test -e $file
then
    echo "The Information of File $file is:"
    ls -l $file
else
    echo "File $file Is Not Exist"
fi
```

10. *Write a shell script to print all the even and odd number from 1 to 100.*

```
cat >evenodd.sh
i=1
j=1
echo "Even Number"
while [ $i -le 100 ]
do
    if [ `expr $i % 2' -eq 0 ]
    then
      echo $i
```

```
      i=`expr $i + 1'
    else
      i=`expr $i + 1'
    fi
done
echo "Odd Number"
while [ $j -le 100 ]
do
    if [ `expr $j % 2' -eq 0 ]
    then
      j=`expr $j + 1'
    else
      echo $j
      j=`expr $j + 1'
    fi
done
```

11. *Write a shell script which gets executed the moment the user logs in. It should display the message "Good Morning" / "Good Afternoon" / "Good Evening" depending upon the time at which the user logs in.*

```
cat >loginfo.sh
tim=`date "+%H"'
if [ $tim -lt 12 ]
then
    echo "Good Morning"
elif [ $tim -lt 17 ]
then
    echo "Good Afternoon"
else
    echo "Good Evening"
fi
```

12. *Write a menu driven program which has following option*:

    (*i*) *Contents of / etc/ passwd*

    (*ii*) *List of users who have currently logged in*

    (*iii*) *Present working directory*

    (*iv*) *Exit*

    Make use of case statement. The menu should be placed approximately in the center of the screen and should be displayed in bold.

```
cat >menu.sh
echo "\t\t\t\t Menu\n\n
    \t 1)\t Contents of /etc/passwd\n
    \t 2)\t List of Users Who have Currently Logged in\n
    \t 3)\t Present Working Directory\n
    \t 4)\t Exit\n\n\t\t Enter Your Option: \c"
read ch
case "$ch" in
    1) ls /etc /passwd ;;
    2) who ;;
    3) pwd ;;
    4) exit ;;
    *) echo "Invalid Option"
esac
```

13. *Write a shell script to count the number of lines and words supplied at standard input.*

```
cat >count.sh
echo "Enter a File Name:"
read fname
if test -e $fname
then
    echo "File Name IS: "$fname
    nol=`cat $fname | wc -l'
    now=`cat $fname | wc -w'
    echo "$fname file have $nol number of Lines"
    echo "$fname file have $now number of Words"
else
    echo "The File $fname is Not Exist"
fi
```

14. *Write a shell script which displays a list of all files in the current directory to which you have read, write and execute permissions.*

```
cat >filedisplay.sh
flag=1
for file in *.*
do
    if test -r $file
    then
```

```
        if test -w $file
        then
          if test -x $file
          then
            echo $file
          fi
        fi
    fi
done
```

15. *Write a shell script which will receive any number of filename as arguments. The shell script should check whether every argument supplied is a file or directory. If it's a directory it should be appropriately reported. If it's a filename then name of the file as well as the number of lines present in it should be reported.*

```
cat >filecheck.sh
for file in *
do
    if test -d $file
    then
      echo "$file is a Directory"
       elif test -f $file
       then
    echo "$file"
    nol='cat $file | wc -l'
    echo "The Number of Line is:$nol"
fi
done
```

Chapter 13

AWK AND PERL PROGRAMMING

13.1 INTRODUCTION TO AWK

awk is a simple and elegant pattern scanning and processing language. I would call it the first and last simple scripting language.

awk is a little programming language, with a syntax close to C in many aspects. It is an interpreted language and the **awk** interpreter processes the instructions.

About the syntax of the awk command interpreter itself:

awk is also the most portable scripting language in existence. It was created in late 70th of the last century almost simultaneously with Borne shell. The name was composed from the initial letters of three original authors **Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger**. It is commonly used as a command-line filter in pipes to reformat the output of other commands. It's the precursor and the main inspiration of Perl. Although originated in Unix it is available and widely used in Windows environment too.

awk takes two inputs: data file and command file. The command file can be absent and necessary commands can be passed as augments. As Ronald P. Loui aptly noted awk is very underappreciated language.

Most people are surprised when I tell them what language we use in our undergraduate AI programming class. That's understandable. We use GAWK. GAWK, Gnu's version of Aho, Weinberger, and Kernighan's old pattern scanning language isn't even viewed as a programming language by most people. Like PERL and TCL, most prefer to view it as a "scripting language".

There are three variations of awk:

AWK—the original from AT&T

NAWK—A newer, improved version from AT&T

GAWK—The Free Software foundation's version.

The main advantage of awk is that unlike Perl and other "scripting monsters" that it is very slim without feature creep so characteristic of Perl and thus it can be very efficiently used with pipes. Also it has rather simple, clean syntax and like much heavier TCL can be used with C for "dual-language" implementations.

In awk you can became productive in several hours. For instance, to print only the second and sixth fields of the date command—the month and year—with a space separating them, use:

```
date | awk '{print $2 " " $6}'
```

Syntactically, a rule consists of a pattern followed by an action. The action is enclosed in curly braces to separate it from the pattern. Newlines usually separate rules. Therefore, an awk program looks like this:

```
pattern {action }
```

When you run awk, you specify an awk *program* that tells awk what to do. The program consists of a series of *rules*. (It may also contain *function definitions*, an advanced feature that we will ignore for now. Each rule specifies one pattern to search for and one action to perform upon finding the pattern.

13.2 HOW TO RUN AWK PROGRAMS?

There are several ways to run an awk program. If the program is short, it is easiest to include it in the command that runs awk, like this:

```
awk 'program' input-file1 input-file 2 ...
```

When the program is long, it is usually more convenient to put it in a file and run it with a command like this:

```
awk -f program-file input-file1 input-file 2 ...
```

13.3 COMMENTS IN AWK PROGRAMS

A *comment* is some text that is included in a program for the sake of human readers; it is not really an executable part of the program. Comments can explain what the program does and how it works. Nearly all programming languages have provisions for comments, as programs are typically hard to understand without them.

In the awk language, a comment starts with the sharp sign character ('#') and continues to the end of the line. The '#' does not have to be the first character on the line. The awk language ignores the rest of a line following a sharp sign.

13.4 THE PRINTF STATEMENT

awk's printf statement is essentially the same as that in C except that the * format specifier is not supported. The printf statement has the general form

```
printf format, expr[1], expr[2], . . ., expr[n]
```

Where *format* is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list, as in "awk printf conversion characters". Each specification begins with a %, ends with a letter that determines the conversion, and may include:

Left-justify expression in its field.

13.4.1 width

Pad field to this width as needed; fields that begin with a leading 0 are padded with zeros.

13.4.2 .prec

Specify maximum string width or digits to right of decimal point.

"awk printf conversion characters" lists the **printf** conversion characters.

13.4.3 awk Printf Conversion Characters

Character	Prints expression as
c	single character
d	decimal number
e	[-]*d.dddddd***E**[+ –]*dd*
f	[-]*ddd.dddddd*
g	e or f conversion, whichever is shorter, with nonsignificant zeros suppressed
o	unsigned octal number
s	string
x	unsigned hexadecimal number
%	print a %; no argument is converted

13.5 CONDITIONAL STATEMENTS

awk also offers very nice C-like if statements. If you'd like, you could rewrite the previous script using an **if** statement:

```
{
   if ( $5 ~ /root/ ) {
      print $3
   }
}
```

Using if statements, we can also transform this code:

```
{
   if ( $0 !~ /matchme/ ) {
      print $1 $3 $4
   }
}
```

Both scripts will output only those lines that *don't* contain a **matchme** character sequence. Again, you can choose the method that works best for your code. They both do the same thing.

awk also allows the use of boolean operators "||" (for "logical or") and "&&"(for "logical and") to allow the creation of more complex boolean expressions:

```
( $1 == "foo" ) && ( $2 == "bar" ) { print }
```

13.6 LOOPS IN AWK

For loop and while loop are used for looping purpose in awk.

Syntax of for loop

Syntax:

```
for (expr1; condition; expr2)
{
    Statement 1
    Statement 2
    Statement N
}
```

Statement(s) are executed repeatedly UNTIL the condition is true. BEFORE the first iteration, expr1 is evaluated. This is usually used to initialize variables for the loop. AFTER each iteration of the loop, expr2 is evaluated. This is usually used to increment a loop counter.

```
$ cat > a1.awk
BEGIN{
printf "Press ENTER to continue with for loop example from LSST
v1.05r3\n"
}
{
sum = 0
i = 1
for (i=1; i<=10; i++)
{
  sum += i; # sum = sum + i
}
printf "Sum for 1 to 10 numbers = %d \nGoodbuy!\n\n", sum
exit 1
}
```

Run it as follows:

$ awk -f while01.awk

Press ENTER to continue with for loop example from LSST v1.05r3

Sum for 1 to 10 numbers = 55

Goodbuy

Above for loops prints the sum of all numbers between 1 to 10, it does use very simple for loop to achieve this. It take number from 1 to 10 using i variable and add it to sum variable as sum = previous sum + current number (*i.e., i*).

Consider the one more example of for loop:

```
$ cat > for_loop
BEGIN {
  printf "To test for loop\n"
  printf "Press CTRL + C to stop\n"
}
{
  for(i=0;i<NF;i++)
  {
    printf "Welcome %s, %d times.\n" ,ENVIRON["USER"], i
  }
}
```

Run it as (and give input as **Welcome to Linux!**)

```
$ awk -f for_loop
```

To test for loop

Press CTRL + C to Stop

Welcome to Linux!

Welcome Anoop, 0 times.

Welcome Anoop, 1 times.

Welcome Anoop, 2 times.

13.7 STARTUP AND CLEANUP ACTIONS (BEGIN & END)

A **BEGIN** rule is executed once only, before the first input record is read. Likewise, an **END** rule is executed once only, after all the input is read. For example:

```
$ awk '
> BEGIN { print "Analysis of \"foo\"" }
> /foo/ { ++n }
> END   { print "\"foo\" appears", n, "times." }' BBS-list
-| Analysis of "foo"
-| "foo" appears 4 times.
```

This program finds the number of records in the input file **BBS-list** that contain the string 'foo'. The **BEGIN** rule prints a title for the report. There is no need to use the **BEGIN** rule to

initialize the counter **n** to zero, since awk does this automatically (see Variables). The second rule increments the variable **n** every time a record containing the pattern 'foo' is read. The **END** rule prints the value of **n** at the end of the run.

The special patterns **BEGIN** and **END** cannot be used in ranges or with Boolean operators (indeed, they cannot be used with any operators). An awk program may have multiple **BEGIN** and/or **END** rules. They are executed in the order in which they appear: all the **BEGIN** rules at startup and all the **END** rules at termination. **BEGIN** and **END** rules may be intermixed with other rules. This feature was added in the 1987 version of awk and is included in the POSIX standard.

You can use while loop as follows:

Syntax:

```
while (condition)
{
     statement1
     statement2
     statementN
Continue as long as given condition is TRUE
}
```

While loop will continue as long as given condition is TRUE. To understand the while loop lets write one more awk script:

Example:

```
$ cat > while_loop
{
no = $1
remn = 0
while ( no > 1 )
   {
     remn = no % 10
     no /= 10
     printf "%d" ,remn
   }
   printf "\nNext number please (CTRL+D to stop):";
}
```

Run it as

```
$awk -f while_loop
654
456
Next number please(CTRL+D to stop):587
785
```

Next number please(CTRL+D to stop):

Here user enters the number 654 which is printed in reverse order *i.e.*, 456. Above program can be explained as follows:

awk support array and for loops. Let say I have IP logs that access to my servers time to time, and I wanna calculate various IP connecting to my servers, I can write a simple awk script uses array and for loops to do that.

The IP logs may looks as below:

```
190607 084849   202.178.23.4 ...
190607 084859   164.78.22.64 ...
190607 084909   202.188.3.2   ...
...
```

Column 1 is date, column 2 is time and column 3 is IP. Let say my query is at 19 June 2007, prints me all the IP access to my servers, and how many times they accessing my servers.

The awk scrips will look something as below:

```
$1=="190607"{IP[$3]++;}
   END {
     for (a in IP)
       print a "access" IP[a] "times.";
     }
```

If column 1 is equal to 190607, make column 3 (which is IP Address) as an item of the array (IP[]), and increase the value of array IP[] by one. After finish accessing all logs, awk will get into END state, and printout results using for loops. Make 'a' as index of array IP, and printout 'a' and its array's value. It may seems complicated at first, try to understand it by reading few times, or just try it out. Please take note that, the curly open brace must be place just after the keyword END.

Make the scripts as access.awk, and run the scripts with awk -f. Assume all logs with headings iplogs and with extension .txt

13.8 BUILT-IN VARIABLES

Most **awk** variables are available for you to use for your own purposes; they never change except when your program assigns values to them, and never affect anything except when your program examines them. However, a few variables in **awk** have special built-in meanings. Some of them **awk** examines automatically, so that they enable you to that they carry information from the internal workings of **awk** to your program. There atell **awk** how to do certain things. Others are set automatically by **awk**, so re following types of built-in variable :

- **User-modified:** Built-in variables that you change to control **awk**.
- **Auto-set:** Built-in variables where **awk** gives you information.
- **ARGC and ARGV:** Ways to use **ARGC** and **ARGV**.

13.8.1 Built-in Variables that Control awk

This is an alphabetical list of the variables which you can change to control how **awk** does certain things.

CONVFMT

This string controls conversion of numbers to strings. It works by being passed, in effect, as the first argument to the sprintf function. Its default value is **"%.6g"**.

FS

FS is the input field separator. The value is a single-character string or a multi-character regular expression that matches the separations between fields in an input record. If the value is the null string (" "), then each character in the record becomes a separate field. The default value is " ", a string consisting of a single space. You can set the value of FS on the command line using the '-F' option:

awk -F, '*program*' *input-files*

OFMT

This string controls conversion of numbers to strings for printing with the print statement. It works by being passed, in effect, as the first argument to the sprintf function. Its default value is **"%.6g"**.

OFS

This is the output field separator. It is output between the fields output by a print statement. Its default value is **""**, a string consisting of a single space.

ORS

This is the output record separator. It is output at the end of every print statement. Its default value is **"\n"**.

RS

This is awk's input record separator. Its default value is a string containing a single newline character, which means that an input record consists of a single line of text. It can also be the null string, in which case records are separated by runs of blank lines, or a regexp, in which case records are separated by matches of the regexp in the input text.

SUBSEP

SUBSEP is the subscript separator. It has the default value of **"\034"**, and is used to separate the parts of the indices of a multi-dimensional array. Thus, the expression foo["A", "B"] really accesses foo["A\034B"].

13.8.2 Built-in Variables that Convey Information

This is an alphabetical list of the variables that are set automatically by **awk** on certain occasions in order to provide information to your program.

ARGC

ARGV

The command-line arguments available to awk programs are stored in an array called ARGV.

ARGC is the number of command-line arguments present. Unlike most awk arrays, ARGV is indexed from zero to ARGC-1. For example:

```
$ awk 'BEGIN {
>   for (i = 0; i < ARGC; i++)
>       print ARGV[i]
>}' inventory-shipped BBS-list
-| awk
-| inventory-shipped
-| BBS-list
```

In this example, ARGV[0] contains "awk", ARGV[1] contains "inventory-shipped", and ARGV[2] contains "BBS-list". The value of ARGC is three, one more than the index of the last element in ARGV, since the elements are numbered from zero.

The following fragment processes ARGV in order to examine, and then remove, command line options.

```
BEGIN {
   for (i = 1; i < ARGC; i++) {
      if (ARGV[i] == "-v")
         verbose = 1
      else if (ARGV[i] == "-d")
         debug = 1
      else if (ARGV[i] ~ /^-?/) {
         e = sprintf("%s: unrecognized option — %c",
               ARGV[0], substr(ARGV[i], 1, ,1))
         print e > "/dev/stderr"
      } else
         break
      delete ARGV[i]
   }
}
```

ENVIRON

An associative array that contains the values of the environment. The array indices are the environment variable names; the values are the values of the particular environment variables. For example, ENVIRON["HOME"] might be `/home/arnold'.

FILENAME

This is the name of the file that awk is currently reading. When no data files are listed on the command line, awk reads from the standard input, and FILENAME is set to "-". FILENAME is changed each time a new file is read.

FNR

FNR is the current record number in the current file. FNR is incremented each time a new record is read. It is reinitialized to zero each time a new input file is started.

NF

NF is the number of fields in the current input record. NF is set each time a new record is read, when a new field is created, or when $0 changes.

NR

This is the number of input records awk has processed since the beginning of the program's execution. NR is set each time a new record is read.

RLENGTH

RLENGTH is the length of the substring matched by the match function. RLENGTH is set by invoking the match function. Its value is the length of the matched string, or –1 if no match was found.

RSTART

RSTART is the start-index in characters of the substring matched by the match function. RSTART is set by invoking the match function. Its value is the position of the string where the matched substring starts, or zero if no match was found.

A side note about NR and FNR. awk simply increments both of these variables each time it reads a record, instead of setting them to the absolute value of the number of records read. This means that your program can change these variables, and their new values will be incremented for each record. For example:

```
$ echo '1
> 2
> 3
> 4' | awk 'NR == 2 { NR = 17 }
> { print NR }'
-| 1
-| 17
-| 18
-| 19
```

13.9 INTRODUCTION TO GETLINE

This command is used in several different ways, and should *not* be used by beginners. It is covered here because this is the chapter on input. The examples that follow the explanation of the getline command include material that has not been covered yet. Therefore, come back and study the getline command *after* you have reviewed the rest of this book and have a good knowledge of how awk works.

getline returns one if it finds a record, and zero if the end of the file is encountered. If there is some error in getting a record, such as a file that cannot be opened, then getline returns –1.

13.9.1 Using getline with No Arguments

The `getline` command can be used without arguments to read input from the current input file. All it does in this case is read the next input record and split it up into fields. This is useful if you've finished processing the current record, but you want to do some special processing *right now* on the next record. Here's an example:

```
awk '{
   if ((t = index($0, "/*")) != 0) {
     # value will be "" if t is 1
     tmp = substr($0, 1, t - 1)
     u = index(substr($0, t + 2), "*/")
     while (u == 0) {
       if (getline <= 0) {
         m = "unexpected EOF or error"
         m = (m ": " ERRNO)
         print m > "/dev/stderr"
         exit
       }
       t = -1
       u = index($0, "*/")
     }
     # substr expression will be "" if */
     # occurred at end of line
     $0 = tmp substr($0, t + u + 3)
   }
   print $0
}'
```

This `awk` program deletes all C-style comments, `'/* ... */'`, from the input. By replacing the `'print $0'` with other statements, you could perform more complicated processing on the decommented input, like searching for matches of a regular expression. This program has a subtle problem—it does not work if one comment ends and another begins on the same line.

13.9.2 Using getline Into a Variable

You can use `'getline` *var*`'` to read the next record from `awk's` input into the variable *var*. No other processing is done.

For example, suppose the next line is a comment, or a special string, and you want to read it, without triggering any rules. This form of `getline` allows you to read that line and store it in a variable so that the main read-a-line-and-check-each-rule loop of `awk` never sees it.

Here's the program:

```
awk '{
  if ((getline tmp) > 0) {
    print tmp
    print $0
  } else
    print $0
}'
```

The `getline` command used in this way sets only the variables `NR` and `FNR` (and of course, *var*). The record is not split into fields, so the values of the fields (including `$0`) and the value of `NF` do not change.

13.9.3 Using getline from a File

Use '`getline` < *file*' to read the next record from the *file file*. Here *file* is a string-valued expression that specifies the file name. '< *file*' is called a **redirection** since it directs input to come from a different place.

For example, the following program reads its input record from the file `'secondary.input'` when it encounters a first field with a value equal to 10 in the current input file.

```
awk '{
  if ($1 == 10) {
    getline < "secondary.input"
    print
  } else
    print
}'
```

Since the main input stream is not used, the values of `NR` and `FNR` are not changed.

13.9.4 Using getline into a Variable from a File

Use '`getline` *var* < *file*' to read input the file *file* and put it in the variable *var*. As above, *file* is a string-valued expression that specifies the file from which to read.

In this version of `getline`, none of the built-in variables are changed, and the record is not split into fields. The only variable changed is *var*.

For example, the following program copies all the input files to the output, except for records that say '`@include` *filename*'. Such a record is replaced by the contents of the file *filename*.

```
awk '{
  if (NF == 2 && $1 == "@include") {
    while ((getline line < $2) > 0)
      print line
```

```
        close($2)
      } else
        print
  }'
```

Note here how the name of the extra input file is not built into the program; it is taken directly from the data, from the second field on the `'@include'` line.

The **close** function is called to ensure that if two identical `'@include'` lines appear in the input, the entire specified file is included twice.

13.9.5 Using getline from a Pipe

You can pipe the output of a command into `getline`, using '*command* | `getline`'. In this case, the string *command* is run as a shell command and its output is piped into **awk** to be used as input. This form of `getline` reads one record at a time from the pipe.

For example, the following program copies its input to its output, except for lines that begin with `'@execute'`, which are replaced by the output produced by running the rest of the line as a shell command:

```
awk '{
    if ($1 == "@execute") {
      tmp = substr($0, 10)
      while ((tmp | getline) > 0)
        print
      close(tmp)
    } else
      print
  }'
```

The **close** function is called to ensure that if two identical `'@execute'` lines appear in the input, the command is run for each one.

13.10 BUILT-IN FUNCTIONS

Built-in functions are functions that are always available for your awk program to call. This chapter defines all the built-in functions in awk.

- **Calling Built-in:** How to call built-in functions.
- **Numeric Functions:** Functions that work with numbers, including int, sin and rand.
- **String Functions:** Functions for string manipulation, such as split, match, and sprintf.
- **I/O Functions:** Functions for files and shell commands.
- **Time Functions:** Functions for dealing with time stamps.

13.10.1 Calling Built-in Functions

To call a built-in function, write the name of the function followed by arguments in parentheses. For example, 'atan2(y + z, 1)' is a call to the function atan2, with two arguments.

Each built-in function accepts a certain number of arguments. In some cases, arguments can be omitted. The defaults for omitted arguments vary from function to function and are described under the individual functions. In some awk implementations, extra arguments given to built-in functions are ignored.

When a function is called, expressions that create the function's actual parameters are evaluated completely before the function call is performed. For example, in the code fragment:

```
i = 4
j = sqrt(i++)
```

The variable i is set to five before sqrt is called with a value of four for its actual parameter.

The order of evaluation of the expressions used for the function's parameters is undefined. Thus, you should not write programs that assume that parameters are evaluated from left to right or from right to left. For example,

```
i = 5
j = atan2(i++, i *= 2)
```

If the order of evaluation is left to right, then i first becomes six, and then 12, and atan2 is called with the two arguments six and 12. But if the order of evaluation is right to left, i first becomes 10, and then 11, and atan2 is called with the two arguments 11 and 10.

13.10.2 Numeric Built-in Functions

Here is a full list of built-in functions that work with numbers. Optional parameters are enclosed in square brackets ("[" and "]").

`int(`*x*`)`

This produces the nearest integer to *x*, located between *x* and zero, truncated toward zero. For example, int(3) is three, int(–3.9) is three, int(–3.9) is –3, and int(–3) is –3 as well.

`sqrt(`*x*`)`

This gives you the positive square root of *x*. It reports an error if *x* is negative.

`exp(`*x*`)`

This gives you the exponential of *x* (e ^ *x*), or reports an error if *x* is out of range.

`log(`*x*`)`

This gives you the natural logarithm of *x*, if *x* is positive; otherwise.

`sin(`*x*`)`

This gives you the sine of *x*, with *x* in radians.

`cos(`*x*`)`

This gives you the cosine of *x*, with *x* in radians.

`atan2(`*y*`, `*x*`)`

This gives you the arctangent of y/x in radians.

`rand()`

This gives you a random number. The values of rand are uniformly-distributed between zero and one. The value is never zero and never one. Often you want random integers instead. Here is a user-defined function you can use to obtain a random non-negative integer less than *n*:

```
function randint(n) {
      return int(n * rand())
}
```

The multiplication produces a random real number greater than zero and less than n. We then make it an integer (using int) between zero and n – 1, inclusive.

`srand([`*x*`])`

The function srand sets the starting point, or seed, for generating random numbers to the value *x*. If you omit the argument *x*, as in srand(), then the current date and time of day are used for a seed. This is the way to get random numbers that are truly unpredictable. The return value of srand is the previous seed.

13.10.3 Built-in Functions for String Manipulation

The functions in this section look at or change the text of one or more strings. Optional parameters are enclosed in square brackets ("[" and "]").

`index(`*in*`, `*find*`)`

This searches the string *in* for the first occurrence of the string *find*, and returns the position in characters where that occurrence begins in the string *in*. For example:

```
$ awk 'BEGIN { print index("peanut", "an") }'
-| 3
```

If *find* is not found, index returns zero.

`length([`*string*`])`

This gives you the number of characters in *string*. If *string* is a number, the length of the digit string representing that number is returned. For example, length("abcde") is five. By contrast, length(15 * 35) works out to three. How? Well, 15 * 35 = 525, and 525 is then converted to the string "525", which has three characters. If no argument is supplied, length returns the length of $0.

`match(`*string*`, `*regexp*`)`

The match function searches the string, *string*, for the longest, leftmost substring matched by the regular expression, *regexp*. It returns the character position, or **index**, of where that substring begins (one, if it starts at the beginning of *string*). If no match is found, it returns zero. For example:

```
awk '{
  if ($1 == "FIND")
     regex = $2
  else {
```

```
        where = match($0, regex)
      if (where != 0)
        print "Match of", regex, "found at", \
              where, "in", $0
      }
    }'
```

This program looks for lines that match the regular expression stored in the variable regex.

`split(`*`string`*`, `*`array`*` [, `*`fieldsep`*`])`

This divides *string* into pieces separated by *fieldsep*, and stores the pieces in *array*. The first piece is stored in *array*[1], the second piece in *array*[2], and so forth. If the *fieldsep* is omitted, the value of FS is used. split returns the number of elements created. For example:

```
split("cul-de-sac", a, "-")
```

splits the string 'cul-de-sac' into three fields using '-' as the separator. It sets the contents of the array a as follows:

```
a[1] = "cul"
a[2] = "de"
a[3] = "sac"
```

The value returned by this call to split is three.

`sprintf(`*`format`*`, `*`expression1`*`,...)`

This returns (without printing) the string that printf would have printed out with the same arguments. For example:

```
sprintf("pi = %.2f (approx.)", 22/7)
```

returns the string `"pi = 3.14 (approx.)"`.

`sub(`*`regexp`*`, `*`replacement`*` [, `*`target`*`])`

The sub function alters the value of *target*. It searches this value, which is treated as a string, for the leftmost longest substring matched by the regular expression, *regexp*, extending this match as far as possible. For example:

```
str = "water, water, everywhere"
sub(/at/, "ith", str)
```

sets str to "wither, water, everywhere", by replacing the leftmost, longest occurrence of 'at' with 'ith'. The sub function returns the number of substitutions made . For example:

```
awk '{ sub(/candidate/, "& and his wife"); print }'
```

changes the first occurrence of 'candidate' to 'candidate and his wife' on each input line. Here is another example:

```
awk 'BEGIN {
    str = "daabaaa"
    sub(/a*/, "c&c", str)
```

```
        print str
    }'
    -| dcaacbaaa,
```

`gsub(`*`regexp`*`, `*`replacement`*` [, `*`target`*`])`

This is similar to the sub function, except gsub replaces *all* of the longest, leftmost, *non-overlapping* matching substrings it can find. The 'g' in gsub stands for "global," which means replace everywhere. For example:

```
awk '{ gsub(/Britain/, "United Kingdom"); print }'
```

replaces all occurrences of the string 'Britain' with 'United Kingdom' for all input records. The gsub function returns the number of substitutions made.

`substr(`*`string`*`, `*`start`*` [, `*`length`*`])`

This returns a *length*-character-long substring of *string*, starting at character number *start*. The first character of a string is character number one. For example, substr("washington", 5, 3) returns "ing". If *length* is not present, this function returns the whole suffix of *string* that begins at character number *start*. For example, substr("washington", 5) returns "ington". The whole suffix is also returned if *length* is greater than the number of characters remaining in the string,

`tolower(`*`string`*`)`

This returns a copy of *string*, with each upper-case character in the string replaced with its corresponding lower-case character. For example, tolower("MiXeD cAsE 123") returns "mixed case 123".

`toupper(`*`string`*`)`

This returns a copy of *string*, with each lower-case character in the string replaced with its corresponding upper-case character. For example, toupper("MiXeD cAsE 123") returns "MIXED CASE 123".

13.10.4 Built-in Functions for Input/Output

The following functions are related to Input/Output (I/O). Optional parameters are enclosed in square brackets ("[" and "]").

`close(`*`filename`*`)`

Close the file *filename*, for input or output. The argument may alternatively be a shell command that was used for redirecting to or from a pipe; then the pipe is closed.

`fflush([`*`filename`*`])`

Flush any buffered output associated *filename*, which is either a file opened for writing, or a shell command for redirecting output to a pipe. Many utility programs will **buffer** their output; they save information to be written to a disk file or terminal in memory, until there is enough for it to be worthwhile to send the data to the ouput device.

`system(`*`command`*`)`

The system function allows the user to execute operating system commands and then return to the awk program. The system function executes the command given by the string *command*. It returns, as its value, the status returned by the command that was

executed. For example:

```
END {
    system("date | mail -s 'awk run done' root")
}
```

In the above program the system administrator will be sent mail when the awk program finishes processing input and begins its end-of-input processing.

Controlling Output Buffering with System

The flush function provides explicit control over output buffering for individual files and pipes. However, its use is not portable to many other awk implementations. An alternative method to flush output buffers is by calling system with a null string as its argument:

```
system("")   # flush output
```

13.10.5 User-defined Functions

Complicated **awk** programs can often be simplified by defining your own functions. User-defined functions can be called just like built-in ones (see section Function Calls), but it is up to you to define them—to tell **awk** what they should do.

- **Definition Syntax:** How to write definitions and what they mean.
- **Function Example:** An example function definition and what it does.
- **Function Caveats:** Things to watch out for.
- **Return Statement:** Specifying the value a function returns.

Function Definition Syntax

Definitions of functions can appear anywhere between the rules of an **awk** program. Thus, the general form of an **awk** program is extended to include sequences of rules *and* user-defined function definitions. There is no need in **awk** to put the definition of a function before all uses of the function. This is because **awk** reads the entire program before starting to execute any of it.

The definition of a function named *name* looks like this:

```
function name(parameter-list)
  {
      body-of-function
  }
```

name is the name of the function to be defined. A valid function name is like a valid variable name: a sequence of letters, digits and underscores, not starting with a digit.

parameter-list is a list of the function's arguments and local variable names, separated by commas. The local variables are initialized to the empty string.

The *body-of-function* consists of **awk** statements. It is the most important part of the definition, because it says what the function should actually *do*.

During execution of the function body, the arguments and local variable values hide or **shadow** any variables of the same names used in the rest of the program. The shadowed variables

are not accessible in the function definition. The function body can contain expressions which call functions. They can even call this function, either directly or by way of another function. When this happens, we say the function is **recursive**.

In many **awk** implementations, including **gawk**, the keyword function may be abbreviated func. To ensure that your **awk** programs are portable, always use the keyword function when defining a function.

Function Definition Examples

Here is an example of a user-defined function, called `myprint`, that takes a number and prints it in a specific format.

```
function myprint(num)
  { printf %6.3g\n", num }
```

To illustrate, here is an `awk` rule which uses our `myprint` function:

```
  $3 > 0     { myprint($3) }
```

This program prints, in our special format, all the third fields that contain a positive number in our input. Therefore, when given:

```
  1.2      3.4      5.6      7.8
 9.10    11.12   –13.14    15.16
17.18    19.20    21.22    23.24
```

This program, using our function to format the results, prints:

```
 5.6
21.2
```

This function deletes all the elements in an array.

```
function delarray(a,    i)
{
  for (i in a)
    delete a[i]
}
```

When working with arrays, it is often necessary to delete all the elements in an array and start over with a new list of elements. Instead of having to repeat this loop everywhere in your program that you need to clear out an array, your program can just call **delarray**.

Here is an example of a recursive function. It takes a string as an input parameter, and returns the string in backwards order.

```
function rev(str, start)
{
    if (start == 0)
     return ""
    return (substr(str, start, 1) rev(str, start - 1))
}
```

If this function is in a file named 'rev.awk', we can test it this way:

```
$ echo "Don't Panic!" |
> awk —source '{ print rev($0, length($0)) }' -f rev.awk
-| !cinaP t'noD
```

Calling User-defined Functions

Calling a function means causing the function to run and do its job. A function call is an expression, and its value is the value returned by the function.

A function call consists of the function name followed by the arguments in parentheses. What you write in the call for the arguments are **awk** expressions; each time the call is executed, these expressions are evaluated, and the values are the actual arguments. For example, here is a call to **foo** with three arguments (the first being a string concatenation):

```
foo(x  y,  "lose",  4  *  z)
```

When a function is called, it is given a *copy* of the values of its arguments. This is known as **call by value**. The caller may use a variable as the expression for the argument, but the called function does not know this.

However, when arrays are the parameters to functions, they are *not* copied. Instead, the array itself is made available for direct manipulation by the function. This is usually called **call by reference**. Changes made to an array parameter inside the body of a function *are* visible outside that function. For example:

```
function changeit(array, ind, nvalue)
{
array[ind] = nvalue
}

BEGIN {
  a[1] = 1; a[2] = 2; a[3] = 3
  changeit(a, 2, "two")
  printf "a[1] = %s, a[2] = %s, a[3] = %s\n",
    a[1], a[2], a[3]
}
```

This program prints 'a[1] = 1, a[2] = two, a[3] = 3', because **changeit** stores **"two"** in the second element of **a**.

The return Statement

The body of a user-defined function can contain a `return` statement. This statement returns control to the rest of the `awk` program. It can also be used to return a value for use in the rest of the `awk` program. It looks like this:

```
return [expression]
```

The *expression* part is optional. If it is omitted, then the returned value is undefined and, therefore, unpredictable. A `return` statement with no value expression is assumed at the end of every function definition. So if control reaches the end of the function body, then the function returns an unpredictable value. `awk` will *not* warn you if you use the return value of such a function.

Here is an example of a user-defined function that returns a value for the largest number among the elements of an array:

```
function maxelt(vec,   i, ret)
{
    for (i in vec) {
      if (ret == "" || vec[i] > ret)
          ret = vec[i]
    }
    return ret
}
```

You call `maxelt` with one argument, which is an array name. The local variables **`i`** and **`ret`** are not intended to be arguments.

13.11 INTRODUCTION TO PERL

Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines some of the best features of C, *sed*, *awk*, and *sh*, so people familiar with those languages should have little difficulty with it. Expression syntax corresponds quite closely to C expression syntax. Unlike most Unix utilities, *perl* does not arbitrarily limit the size of your data—if you've got the memory, *perl* can slurp in your whole file as a single string. Recursion is of unlimited depth. And the hash tables used by associative arrays grow as necessary to prevent degraded performance. *Perl* uses sophisticated pattern matching techniques to scan large amounts of data very quickly. Although optimized for scanning text, *perl* can also deal with binary data, and can make dbm files look like associative arrays (where dbm is available). Setuid *perl* scripts are safer than C programs through a dataflow tracing mechanism which prevents many stupid security holes.

13.11.1 Running Perl

A Perl program is just a text file. You edit the text of your Perl program, and the Perl interpreter reads that text file directly to "run" it. This structure makes your edit-run-debug cycle nice and fast. On Unix, the Perl interpreter is called "perl" and you run a Perl program by running the Perl interpreter and telling it which file contains your Perl program ...

```
> perl myprog.pl
```

The interpreter makes one pass of the file to analyze it and if there are no syntax or other obvious errors, the interpreter runs the Perl code. There is no "main" function—the interpreter just executes the statements in the file starting at the top.

Following the Unix convention, the very first line in a Perl file usually looks like this...

```
#!/usr/bin/perl -w
```

This special line is a hint to Unix to use the Perl interpreter to execute the code in this file. The "-w" switch turns on warnings which is generally a good idea. In unix, use "chmod" to set the execute bit on a Perl file so it can be run right from the prompt...

```
> chmod u+x foo.pl  ## set the "execute" bit for the file once
>
> foo.pl     ## automatically uses the perl interpreter to "run" this
file
```

The second line in a Perl file is usually a "require" declaration that specifies what version of Perl the program expects ...

```
#!/usr/bin/perl -w
require 5.004;
```

Perl is available for every operating system imaginable, including of course Windows and MacOS, and it's part of the default install in Mac OSX.

13.11.2 Syntax and Variables

The simplest Perl variables are "scalar" variables which hold a single string or number. Scalar variable names begin with a dollar sign ($) such as `$sum` or `$greeting`. Scalar and other variables do not need to be pre-declared—using a variable automatically declares it as a global variable. Variable names and other identifiers are composed of letters, digits, and underscores (_) and are case sensitive. Comments begin with a "#" and extend to the end of the line.

```
$x = 2; ## scalar var $x set to the number 2
$greeting = "hello"; ## scalar var $greeting set to the string "hello"
```

A variable that has not been given a value has the special value "undef" which can be detected using the "defined" operator. Undef looks like 0 when used as a number, or the empty string "" when used as a string, although a well written program probably should not depend on undef in that way. When Perl is run with "warnings" enabled (the -w flag), using an undef variable prints a warning.

```
if(!defined($binky)){
 print"the variable 'binky' has not been given a value!\n";
}
```

13.12 STARTING A PERL SCRIPT

Upon startup, *perl* looks for your script in one of the following places:

1. Specified line by line via **-e** switches on the command line.

2. Contained in the file specified by the first filename on the command line.
3. Passed in implicitly via standard input. This only works if there are no filename arguments—to pass arguments to a *stdin* script you must explicitly specify a-for the script name.

After locating your script, *perl* compiles it to an internal form. If the script is syntactically correct, it is executed.

13.12.1 Strings

Strings constants are enclosed within double quotes (") or in single quotes (‘). Strings in double quotes are treated specially—special directives like `\n` (newline) and `\x20` (hex 20) are expanded. More importantly, a variable, such as $x, inside a double quoted string is evaluated at run-time and the result is pasted into the string. This evaluation of variables into strings is called "interpolation" and it's a great Perl feature. Single quoted (‘) strings suppress all the special evaluation—they do not evaluate `\n` or $x, and they may contain newlines.

```
$fname = "binky.txt";
$a = "Could not open the file $fname.";  ## $fname evaluated and pasted in -- neato!
$b = 'Could not open the file $fname.';  ## single quotes (') do no special evaluation
## $a is now "Could not open the file binky.txt."
## $b is now "Could not open the file $fname."
```

The characters ‘$’ and ‘@’ are used to trigger interpolation into strings, so those characters need to be escaped with a backslash (\) if you want them in a string. For example:

```
"nick\@stanford.edu found \$1".
```

The dot operator (.) concatenates two strings. If Perl has a number or other type when it wants a string, it just silently converts the value to a string and continues. It works the other way too—a string such as "42" will evaluate to the integer 42 in an integer context.

```
$num = 42;
$string = "The " . $num . " ultimate" . " answer";
## $string is now "The 42 ultimate answer"
```

13.12.2 Using the Perl chop() function

Sometimes you will find you want to unconditionally remove the last character from a string. While you can easily do this with regular expressions, **`chop`** is more efficient.

The **`chop()`** function will remove the last character of a string (or group of strings) regardless of what that character is.

Example 1. *Chopping a string*

The **`chop()`** function removes and returns the last character from the given string:

```
#!/usr/bin/perl
Use strict;
```

```
Use warnings;
my $string= 'Anoop';
my $chr =chop($string);
print "String : $ string\n";
print "Char : $chr\n";
This program gives you:
String: Anoo
Char : p
```

If the string is empty, **chop()** will return an empty string. If the string is undefined, **chop()** will return undefined.

Example 2. *Chopping strings in an array*

If you pass the **chop()** function an array, it will remove the last character from every element in the array.

Note that this will only work for a one-dimensional array. In other words, it is not valid to pass in an array reference, or an array that contains an array (or hash).

```
#!/usr/bin/perl
use strict;
use warnings;
my @array = ('fred', 'bob', 'jill', 'joan');
my $chr = chop(@array);
foreach my $str (@array) {
print "$str\n";}
print "Char: $chr\n";
```

This produces the output:

```
fre
bo
jil
joa
Char: n
```

Example 3. *Chopping strings in a hash*

If you pass a hash into **chop()**, it will remove the last character from the values (not the keys) in the hash. For example:

```
#!/usr/bin/perl
use strict;
use warnings;
my %hash = (
first => 'one',
```

```
second => 'two',
third => 'three',
);
my $chr = chop(%hash);
foreach my $k (keys %hash) {
print "$k: $hash{$k}\n";
}
print "Char: $chr\n";
This program outputs:
first: on
second: tw
third: thre
Char: e
```

Note that as with arrays, chop is not designed to process hash reference or hashes containing other hashes (or arrays).

13.13 PERL—ARITHMETIC OPERATORS

Arithmetic operators are symbols used to execute general arithmetic procedures including: addition (+), subtraction (–), multiplication (*), and division (/).

Arithmetic Operators:

Operator	Example	Result	Definition
+	7 + 7	= 14	Addition
–	7 – 7	= 0	Subtraction
*	7 * 7	= 49	Multiplication
/	7 / 7	= 1	Division
**	7 ** 7	= 823543	Exponents
%	7 % 7	= 0	Modulus

With these operators we can take a number and perform some simple math operations.

13.13.1 PERL Arithmetic

```
#!/usr/bin/perl
print "content-type: text/html \n\n";  #HTTP Header
#PICK A NUMBER
$x = 81;
```

```
$add = $x + 9;
$sub = $x - 9;
$mul = $x * 10;
$div = $x / 9;
$exp = $x ** 5;
$mod = $x % 79;
print "$x plus 9 is $add<br />";
print "$x minus 9 is $sub<br />";
print "$x times 10 is $mul<br />";
print "$x divided by 9 is $div<br />";
print "$x to the 5th is $exp<br />";
print "$x modulus 79 is $mod<br />";
```

Your browser should read:

arithmetic.pl:

81 plus 9 is 90

81 minus 9 is 72

81 times 10 is 810

81 divided by 9 is 9

81 to the 5th is 3486784401

81 modulus 79 is 2

13.14 PERL—ASSIGNMENT OPERATORS

Assignment operators perform an arithmetic operation and then assign the value to the existing variable. Using assignment operators we will replace that value with a new number after performing some type of mathematical operation.

Assignment Operators:

Operator	Definition	Example
+ =	Addition	($x + = 10)
– =	Subtraction	($x – = 10)
* =	Multiplication	($x * = 10)
/ =	Division	($x / = 10)
% =	Modulus	($x % = 10)
** =	Exponent	($x ** = 10)

13.14.1 PERL Assignment

```
#!/usr/bin/perl
print "content-type: text/html \n\n";  #HTTP HEADER
#START WITH A NUMBER
$x = 5;
print '$x plus 10 is '.($x += 10);
print "<br />x is now ".$x;      #ADD 10
print '<br />$x minus 3 is '.($x -= 3);
print "<br />x is now ".$x;      #SUBTRACT 3
print '<br />$x times 10 is '.($x *= 10);
print "<br />x is now ".$x.      #MULTIPLY BY 10
print '<br />$x divided by 10 is '.($x /= 10);
print "<br />x is now ".$x;      #DIVIDE BY 10
print '<br />Modulus of $x mod 10 is '.($x %= 10);
print "<br />x is now ".$x;      #MODULUS
print '<br />$x to the tenth power is '.($x **= 10);
print "<br />x is now ".$x;      #2 to the 10th
Display:
$x plus 10 is 15
x is now 15
$x minus 3 is 12
x is now 12
$x times 10 is 120
$x divided by 10 is 12
x is now 1201
x is now 12
Modulus of $x mod 10 is 2
x is now 2
$x to the tenth power is 1024
x is now 1024
```

Each time an operation is performed our variable ($x) is permanently changed to a new value of $x.

13.15 PERL—LOGICAL AND RELATIONAL OPERATORS

Relationship operators compare one variable to another. (5 < 12) They are used to compare equality or inequality of two or more variables, be it a string or numeric data.

Logical operators state *and/or* relationships. Meaning, you can take two variables and test an either or conditional. Logical operators are used later on in conditionals and loops. For now, just be able to recognize them in the upcoming examples.

Logical/Relational Operators:

Relational

Operator	Example	Defined	Result
==,eq	5 == 5 5 eq 5	Test: Is 5 equal to 5?	True
!=,ne	7 != 2 7 ne 2	Test: Is 7 not equal to 2?	True
<,lt	7 < 4 7 lt 4	Test: Is 7 less than 4?	False
>,gt	7 > 4 7 gt 4	Test: Is 7 greater than 4?	True
<=,le	7 <= 11 7 le 11	Test: Is 7 less than or equal to 11?	True
>=,ge	7 >= 11 7 ge 11	Test: Is 7 greater than or equal to 11?	False

13.15.1 Logical

Operator	Defined	Example
&&,and	Associates two variables using AND	if (($x && $y) == 5)...
||,or	Associates two variables using OR	if (($x || $y) == 5)...

13.15.2 PERL—Variables + Operators

Variables can be used with mathematical formulas using PERL Operators discussed in a previous lesson. Also, note that variables are case sensitive. "$myvariable," "$MYvariable," and "$Myvariable" can all be assigned different values due to case sensitivity. Numbers of course can be added, subtracted, or multiplied using operators. Strings as shown in the example below can also be used with operators.

PERL Code:

```
#!/usr/bin/perl
print "Content-type: text/html \n\n"; #HTTP HEADER
#TWO STRINGS TO BE ADDED
$myvariable = "Hello,";
```

```
$Myvariable = " World";
#ADD TWO STRINGS TOGETHER
$string3 = "$myvariable $Myvariable";
print $string3;
```

13.16 PERL—$_ AND @_

Perl's a great language for special variables—variables that are set up without the programmer having to intervene and providing information ranging from the number of lines read from the current input file ($.) through the current process ID ($$) and the operating system ($^O). Other special variables effect how certain operations are performed ($| controlling output buffering/flushing, for example), or are fundamental in the operation of certain facilities—no more so than $_ and @_.

Lets clear a misconception. **$_** and **@_** are *different variables*. In Perl, you can have a list and a scalar of the same name, and they refer to unrelated pieces of memory.

$_ is known as the "default input and pattern matching space". In other words, if you read in from a file handle at the top of a **while** loop, or run a **foreach** loop and don't name a loop variable, $_ is set up for you. Then any regular expression matches, **chop**s (and **lc**s and many more) without a parameter, and even **print**s assume you want to work on $_.

Thus:

```
while ($line = <FH>) {
  if ($line =~ /Perl/) {
    print FHO $line;
    }
  print uc $line;
  }
```

Shortens to:

```
while (<FH>) {
  /Perl/ and
    print FHO ;
  print uc;
  }
```

13.16.1 @_ Is the List of Incoming Parameters to a Sub

So if you write a sub, you refer to the first parameter in it as **$_[0]**, the second parameter as **$_[1]** and so on. And you can refer to **$#_** as the index number of the last parameter:

```
sub demo {
  print "Called with ",$#_+1," params\n";
  print "First param was $_[0]\n";
```

Note that the **English** module adds in the ability to refer to the special variables by other longer, but easier to remember, names such as @ARG for @_ and $PID for $$. But **`use English;`** can have a detrimental performance effect if you're matching regular expressions against long incoming strings.

13.17 ARRAYS—@

List arrays (also known simply as "arrays" for short) take the concept of scalar variables to the next level. Whereas scalar variables associate one value with one variable name, list arrays associate one array name with a "list" of values.

Array constants are specified using parenthesis ( ) and the elements are separated with commas. Perl arrays are like lists or collections in other languages since they can grow and shrink, but in Perl they are just called "arrays". Array variable names begin with the at-sign (@). Unlike C, the assignment operator (=) works for arrays—an independent copy of the array and its elements is made. Arrays may not contain other arrays as elements. Perl has sort of a "1-deep" mentality. Actually, it's possible to get around the 1-deep constraint using "references", but it's no fun. Arrays work best if they just contain scalars (strings and numbers). The elements in an array do not all need to be the same type. A list array is defined with the following syntax:

```
@array_name  =  ("element_1",  "element_2"..."element_n");
```

For example, consider the following list array definition:

```
@available_colors  =  ("red",  "green",  "blue","brown");
@array = (1, 2, "hello");       ## a 3 element array
@empty = ();                    ## the array with 0 elements
$x = 1;
$y = 2;
@nums = ($x + $y, $x - $y);
## @nums is now (3, -1)
```

Just as in C, square brackets [ ] are used to refer to elements, so $a[6] is the element at index 6 in the array @a. As in C, array indexes start at 0. Notice that the syntax to access an element begins with '$' not '@'—use '@' only when referring to the whole array (remember: all scalar expressions begin with $).

```
@array = (1, 2, "hello", "there");
$array[0] = $array[0] + $array[1];## $array[0] is now 3
```

Perl arrays are not bounds checked. If code attempts to read an element outside the array size, undef is returned. If code writes outside the array size, the array grows automatically to be big enough. Well written code probably should not rely on either of those features.

```
@array = (1, 2, "hello", "there");
$sum = $array[0] + $array[27];       ## $sum is now 1, since $array[27]
returned undef
$array[99] = "the end";              ## array grows to be size 100
```

When used in a scalar context, an array evaluates to its length. The "scalar" operator will force the evaluation of something in a scalar context, so you can use scalar() to get the length of an array. As an alternative to using scalar, the expression `$#array` is the index of the last element of the array which is always one less than the length.

```
@array = (1, 2, "hello", "there");
$len = @array;              ## $len is now 4 (the length of @array)
$len = scalar(@array)       ## same as above, since $len represented
                                                        a scalar
                            ## context anyway, but this is more
                                                        explicit
@letters = ("a", "b", "c");
$i = $#letters;             ## $i is now 2
```

That `scalar(@array)` is the way to refer to the length of an array is not a great moment in the history of readable code. At least I haven't showed you the even more vulgar forms such as `(0 + @a)`.

The sort operator `(sort @a)` returns a copy of the array sorted in ascending alphabetic order. Note that sort does not change the original array. Here are some common ways to sort...

```
(sort @array)                          ## sort alphabetically, with
                                                    uppercase first
(sort {$a <=> $b} @array)              ## sort numerically
(sort {$b cmp $a} @array)              ## sort reverse alphabetically
(sort {lc($a) cmp lc($b)} @array)      ## sort alphabetically, ignoring
                                       case (somewhat inefficient)
```

The sort expression above pass a comparator function { . . . } to the sort operator, where the special variables $a and $b are the two elements to compare –`cmp` is the built-in string compare, and <=> is the built-in numeric compare.

There's a variant of array assignment that is used sometimes to assign several variables at once. If an array on the left hand side of an assignment operation contains the names of variables, the variables are assigned the corresponding values from the right hand side.

```
($x, $y, $z) = (1, 2, "hello", 4);
## assigns $x=1, $y=2, $z="hello", and the 4 is discarded
```

This type of assignment only works with scalars. If one of the values is an array, the wrong thing happens (see "flattening" below).

13.17.1 Array Add/Remove/Splice Functions

These handy operators will add or remove an element from an array. These operators change the array they operate on ...

- Operating at the "front" ($array[0]) end of the array...
  - `shift(array)`—returns the frontmost element and removes it from the array. Can be used in a loop to gradually remove and examine all the elements in an array left to right. The foreach operator, below, is another way to examine all the elements.
  - `unshift(array, elem)`—inserts an element at the front of the array. Opposite of shift.
- Operating at the "back" ($array[$len-1]) end of the array...
  - `pop(array)`—returns the endmost element (right hand side) and removes it from the array.
  - `push(array, elem)`—adds a single element to the end of the array. Opposite of pop.
- `splice(array, index, length, array2)`—removes the section of the array defined by index and length, and replaces that section with the elements from array2. If array2 is omitted, splice() simply deletes. For example, to delete the element at index $i from an array, use splice(@array, $i, 1).

13.17.2 Hash Arrays—%

Hash arrays, also known as "associative" arrays, are a built-in key/value data structure. Hash arrays are optimized to find the value for a key very quickly. Hash array variables begin with a percent sign (%) and use curly braces { } to access the value for a particular key. If there is no such key in the array, the value returned is undef. The keys are case sensitive, so you may want to consistently uppercase or lowercase strings before using them as a key (use lc and uc).

```
$dict{"bart"}  = "I didn't do it";
$dict{"homer"} = "D'Oh";
$dict{"lisa"}  = "";
## %dict now contains the key/value pairs (("bart" => "I didn't do it"),
## ("homer" => "D'oh"), ("lisa" => ""))
$string = $dict{"bart"};       ## Lookup the key "bart" to get
                               ## the value "I didn't do it"
$string = $dict{"marge"};      ## Returns undef -- there is no entry for
"marge"
$dict{"homer"} = "Mmmm, scalars";     ## change the value for the key
                                      ## "homer" to "Mmmm, scalars"
```

A hash array may be converted back and forth to an array where each key is immediately followed by its value. Each key is adjacent to its value, but the order of the key/value pairs depends on the hashing of the keys and so appears random. The "keys" operator returns an array of the keys from an associative array. The "values" operator returns an array of all the values, in an order consistent with the keys operator.

```
@array = %dict;
## @array will look something like
##  ("homer", "D'oh", "lisa", "", "bart", "I didn't do it");
##
## (keys %dict) looks like ("homer", "lisa, "bart")
## or use (sort (keys %dict))
```

You can use => instead of comma and so write a hash array value this cute way...

```
%dict = (
    "bart" ⇒ "I didn't do it",
    "homer" ⇒ "D'Oh",
    "lisa" ⇒ "",
);
```

In Java or C you might create an object or struct to gather a few items together. In Perl you might just throw those things together in a hash array.

13.18 @ARGV AND %ENV

The built-in array @ARGV contains the command line arguments for a Perl program. The following run of the Perl program critic.pl will have the ARGV array ("-poetry", "poem.txt").

```
unix% perl critic.pl -poetry poem.txt
```

%ENV contains the environment variables of the context that launched the Perl program. @ARGV and %ENV make the most sense in a Unix environment.

13.19 IF/WHILE SYNTAX

Perl's control syntax looks like C's control syntax. Blocks of statements are surrounded by curly braces { }. Statements are terminated with semicolons (;). The parenthesis and curly braces are **required** in if/while/for forms. There is not a distinct "boolean" type, and there are no "true" or "false" keywords in the language. Instead, the empty string, the empty array, the number 0 and undef all evaluate to false, and everything else is true. The logical operators &&, ||, ! work as in C. There are also keyword equivalents (and, or, not) which are almost the same, but have lower precedence.

13.19.1 IF

```
if (expr) {       ## basic if -- ( ) and { } required
   stmt;
   stmt;
}
```

```
if (expr) {        ## if + elsif + else
   stmt;
   stmt;
}
elsif (expr) {     ## note the strange spelling of "elsif"
   stmt;
   stmt;
}
else {
   stmt;
   stmt;
}
unless (expr) {    ## if variant which negates the boolean test
   stmt;
   stmt;
}
```

13.19.2 If Variants

As an alternative to the classic `if() { }` structure, you may use `if`, `while`, and `unless` as modifiers that come **after** the single statement they control ...

```
$x = 3 if $x > 3;  ## equivalent to: if ($x > 3) {$x = 3;}
$x = 3 unless $x <= 3;
```

For these constructs, the parentheses are not required around the boolean expression. This may be another case where Perl is using a structure from human languages. I never use this syntax because I just cannot get used to seeing the condition after the statement it modifies. If you were defusing a bomb, would you like instructions like this: "Locate the red wire coming out of the control block and cut it. Unless it's a weekday—in that case cut the black wire."

13.19.3 Loops

These work just as in C...

```
while (expr) {
   stmt;
   stmt;
}
for (init_expr; test_expr; increment_expr) {
   stmt;
   stmt;
}
```

```
## typical for loop to count 0..99
for ($i=0; $i<100; $i++) {
   stmt;
   stmt;
}
```

The "next" operator forces the loop to the next iteration. The "last" operator breaks out of the loop like break in C. This is one case where Perl (last) does not use the same keyword name as C (break).

13.19.4 Array Iteration—foreach

The "foreach" construct is a handy way to iterate a variable over all the elements in an array. Because of foreach, you rarely need to write a traditional for or while loop to index into an array. Foreach is also likely to be implemented efficiently. (It's a shame Java does not include a compact iteration syntax in the language. It would make Java a better language at the cost of some design elegance.)

```
foreach $var (@array) {
   stmt;              ## use $var in here
   stmt;
}
```

Any array expression may be used in the foreach. The array expression is evaluated once before the loop starts. The iterating variable, such as $var, is actually a pointer to each element in the array, so assigning to $var will actually change the elements in the array.

Example, assuming array lengths are passed before arrays:

```
sub aeq { # compare two array values
local(@a) = splice(@_,0,shift);
local(@b) = splice(@_,0,shift);
return 0 unless @a == @b; # same len?
while (@a) {
return 0 if pop(@a) ne pop(@b);
}
return 1;
}
```

13.20 FILE INPUT

Variables which represent files are called "file handles", and they are handled differently from other variables. They do not begin with any special character—they are just plain words. By convention, file handle variables are written in all upper case, like FILE_OUT or SOCK. The file handles are all in a global namespace, so you cannot allocate them locally like other variables. File handles can be passed from one routine to another like strings (detailed below).

The standard file handles STDIN, STDOUT, and STDERR are automatically opened before the program runs. Surrounding a file handle with <> is an expression that returns one line from the file including the "\n" character, so <STDIN> returns one line from standard input. The <> operator returns undef when there is no more input. The "chop" operator removes the last character from a string, so it can be used just after an input operation to remove the trailing "\n". The "chomp" operator is similar, but only removes the character if it is the end-of-line character.

```
$line = <STDIN>;       ## read one line from the STDIN file handle
chomp($line);          ## remove the trailing "\n" if present
$line2 = <FILE2>;      ## read one line from the FILE2 file handle
                       ## which must be have been opened previously
```

Since the input operator returns undef at the end of the file, the standard pattern to read all the lines in a file is ...

```
## read every line of a file
while ($line = <STDIN>) {
   ## do something with $line
}
```

13.20.1 Open and Close

The "open" and "close" operators operate as in C to connect a file handle to a filename in the file system.

```
open(F1, "filename");      ## open "filename" for reading as file handle F1
open(F2, ">filename");     ## open "filename" for writing as file handle F2
open(F3, ">>appendtome")   ## open "appendtome" for appending
close(F1);                 ## close a file handle
```

Open can also be used to establish a reading or writing connection to a separate process launched by the OS. This works best on Unix.

```
open(F4, "ls -l |");          ## open a pipe to read from an ls process
open(F5, "| mail $addr");     ## open a pipe to write to a mail process
```

Passing commands to the shell to launch an OS process in this way can be very convenient, but it's also a famous source of security problems in CGI programs. When writing a CGI, do not pass a string from the client side as a filename in a call to open().

Open returns undef on failure, so the following phrase is often to exit if a file can't be opened. The die operator prints an error message and terminates the program.

```
open(FILE, $fname) || die "Could not open $fname\n";
```

In this example, the logical-or operator || essentially builds an if statement, since it only evaluates the second expression if the first if false. This construct is a little strange, but it is a common code pattern for Perl error handling.

13.20.2 Input Variants

In a scalar context the input operator reads one line at a time. In an array context, the input operator reads the entire file into memory as an array of its lines ...

```
@a = <FILE>;  ## read the whole file in as an array of lines
```

This syntax can be dangerous. The following statement looks like it reads just a single line, but actually the left hand side is an array context, so it reads the whole file and then discards all but the first line ...

```
my($line) = <FILE>;
```

The behaviour of <FILE> also depends on the special global variable $/ which is the current the end-of-line marker (usually "\n"). Setting $/ to undef causes <FILE> to read the whole file into a single string.

```
$/ = undef;
$all = <FILE>;          ## read the whole file into one string
```

You can remember that $/ is the end-of-line marker because "/" is used to designate separate lines of poetry. I thought this mnemonic was silly when I first saw it, but sure enough, I now remember that $/ is the end-of-line marker.

13.20.3 Print Output

Print takes a series of things to print separated by commas. By default, print writes to the STDOUT file handle.

```
print "Woo Hoo\n";   ## print a string to STDOUT
$num = 42;
$str = " Hoo";
print "Woo", $a, " bbb $num", "\n";  ## print several things
```

An optional first argument to print can specify the destination file handle. There is no comma after the file handle.

```
print FILE "Here", " there", " everywhere!", "\n"; ## no comma after FILE
```

File Processing Example1

As an example, here's some code that opens each of the files listed in the @ARGV array, and reads in and prints out their contents to standard output ...

```
#!/usr/bin/perl -w
require 5.004;
## Open each command line file and print its contents to standard out
foreach $fname (@ARGV) {
      open(FILE, $fname) || die("Could not open $fname\n");
      while($line = <FILE>) {
        print $line;
      }
      close(FILE);
}
```

The above uses "die" to abort the program if one of the files cannot be opened. We could use a more flexible strategy where we print an error message for that file but continue to try to process the other files. Alternately we could use the function call `exit(-1)` to exit the program with an error code. Also, the following shift pattern is a common alternative way to iterate through an array ...

```
while($fname = shift(@ARGV)) {...
```

Example 2

Here is the basic perl program which does the same as the UNIX **cat** command on a certain file.

```
#!/usr/local/bin/perl
#
# Program to open the password file, read it in,
# print it, and close it again.

$file = '/etc/passwd';      # Name the file
open(INFO, $file);          # Open the file
@lines = <INFO>;            # Read it into an array
close(INFO);                # Close the file
print @lines;               # Print the array
```

13.21 STRING PROCESSING WITH REGULAR EXPRESSIONS

Perl's most famous strength is in string manipulation with regular expressions. Perl has a million string processing features—we'll just cover the main ones here. The simple syntax to search for a pattern in a string is ...

```
    ($string =~ /pattern/)  ## true if the pattern is found somewhere in the
string
    ("binky" =~ /ink/)  ==> TRUE
    ("binky" =~ /onk/)  ==> FALSE
```

In the simplest case, the exact characters in the regular expression pattern must occur in the string somewhere. All of the characters in the pattern must be matched, but the pattern does not need to be right at the start or end of the string, and the pattern does not need to use all the characters in the string.

13.21.1 Character Codes

The power of regular expressions is that they can specify patterns, not just fixed characters. First, there are special matching characters ...

- **a**, **X**, **9**—ordinary characters just match that character exactly
- **.** (a period)—matches any single character except "\n"
- **\w**—(lowercase w) matches a "word" character: a letter or digit [a-zA-Z0-9]
- **\W**—(uppercase W) any non word character
- **\s**—(lowercase s) matches a single whitespace character — space, newline, return, tab, form [\n\r\t\f]
- **\S**—(uppercase S) any non whitespace character
- **\t**, **\n**, **\r** -- tab, newline, return
- **\d**—decimal digit [0-9]
- \ —inhibit the "specialness" of a character. So, for example, use **\.** to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can always put a slash in front of it \@ to make sure it is treated just as a character.

```
    "piiig" =~ /p...g/          ==> TRUE    . = any char (except \n)
    "piiig" =~ /.../            ==> TRUE    need not use up the whole string
    "piiig" =~ /p....g/         ==> FALSE   must use up the whole pattern (the
g is not matched)
    "piiig" =~ /p\w\w\wg/       ==> TRUE    \w = any letter or digit
    "p123g" =~ /p\d\d\dg/       ==> TRUE    \d = 0..9 digit
```

The modifier "i" after the last / means the match should be case insensitive...

```
    "PiIIg" =~ /pIiig/ ==> FALSE
    "PiIIg" =~ /pIiig/i ==> TRUE
```

String interpolation works in regular expression patterns. The variable values are pasted into the expression once before it is evaluated. Characters like * and + continue to have their special meanings in the pattern after interpolation, unless the pattern is bracketed with a \Q..\E. The following examples test if the pattern in $target occurs within brackets < > in $string ...

```
    $string =~ /<$target>/          ## Look for <$target>, '.' '*' keep their
special meanings in $target
    $string =~ /<\Q$target\E>/      ## The \Q..\E puts a backslash in front of
every char,
                                    ## so '.' '*' etc. in $target will not have
their special meanings
```

Similar to the \Q..\E form, the quotemeta() function returns a string with every character \ escaped. There is an optional "m" (for "match") that comes before the first /. If the "m" is used, then any character can be used for the delimiter instead of / — so you could use " or # to delimit the pattern. This is handy if what you are trying to match has a lot of /'s in it. If the delimiter is the single quote (') then interpolation is suppressed. The following expressions are all equivalent ...

```
"piiig" =~ m/piiig/
"piiig" =~ m"piiig"
"piiig" =~ m#piiig#
```

13.21.2 Control Codes

Things get really interesting when you add in control codes to the regular expression pattern ...

- **?** —match 0 or 1 occurrences of the pattern to its left
- * —0 or more occurrences of the pattern to its left
- **+** —1 or more occurrences of the pattern to its left
- | —(vertical bar) logical or—matches the pattern either on its left or right
- parenthesis **( )**—group sequences of patterns
- **^** —matches the start of the string
- **$** —matches the end of the string

13.21.3 Leftmost and Largest

First, Perl tries to find the leftmost match for the pattern, and second it tries to use up as much of the string as possible—*i.e.*, let + and * use up as many characters as possible.

13.21.4 Regular Expression Examples

The following series gradually demonstrate each of the above control codes. Study them carefully—small details in regular expressions make a big difference. That's what makes them powerful, but it makes them tricky as well.

Old joke: What do you call a pig with three eyes? Piiig!

Search for the pattern 'iiig' in the string 'piiig'

"piiig" =~ **m/iiig/** ==> TRUE

The pattern may be anywhere inside the string

"piiig" =~ **m/iii/** ==> TRUE
All of the pattern must match
"piiig" =~ **m/iiii/** ==> FALSE
. = any char but \n
"piiig" =~ **m/...ig/** ==> TRUE
"piiig" =~ **m/p.i../** ==> TRUE
The last . in the pattern is not matched
"piiig" =~ **m/p.i.../** ==> FALSE
\d = digit [0-9]
"p123g" =~ **m/p\d\d\dg/** ==> TRUE
"p123g" =~ **m/p\d\d\d\d/** ==> FALSE
\w = letter or digit
"p123g" =~ **m/\w\w\w\w\w/** ==> TRUE
i+ = one or more i's
"piiig" =~ **m/pi+g/** ==> TRUE
matches iii
"piiig" =~ **m/i+/** ==> TRUE
"piiig" =~ **m/p+i+g+/** ==> TRUE
"piiig" =~ **m/p+g+/** ==> FALSE
i* = zero or more i's
"piiig" =~ **m/pi*g/** ==> TRUE
"piiig" =~ **m/p*i*g*/** ==> TRUE
X* can match zero X's
"piiig" =~ **m/pi*X*g/** ==> TRUE
^ = start, $ = end
"piiig" =~ **m/^pi+g$/** ==> TRUE
i is not at the start
"piiig" =~ **m/^i+g$/** ==> FALSE
i is not at the end
"piiig" =~ **m/^pi+$/** ==> FALSE
"piiig" =~ **m/^p.+g$/** ==> TRUE
"piiig" =~ **m/^p.+$/** ==> TRUE
"piiig" =~ **m/^.+$/** ==> TRUE
g is not at the start
"piiig" =~ **m/^g.+$/** ==> FALSE
Needs at least one char after the g

"piiig" =~ **m/g.+/** ==> FALSE
Needs at least zero chars after the g
"piiig" =~ **m/g.*/** ==> TRUE
| = left or right expression
"cat" =~ **m/^(cat|hat)$/** ==> TRUE
"hat" =~ **m/^(cat|hat)$/** ==> TRUE
"cathatcatcat" =~ **m/^(cat|hat)+$/** ==> TRUE
"cathatcatcat" =~ **m/^(c|a|t|h)+$/** ==> TRUE
"cathatcatcat" =~ **m/^(c|a|t)+$/** ==> FALSE

Matches and stops at first 'cat'; does not get to 'catcat' on the right
"cathatcatcat" =~ **m/(c|a|t)+/** ==> TRUE
? = optional
"12121x2121x2" =~ **m/^(1x?2)+$/** ==> TRUE
"aaaxbbbabaxbb" =~ **m/^(a+x?b+)+$/** ==> TRUE
"aaaxxbbb" =~ **m/^(a+x?b+)+$/** ==> FALSE
Three words separated by spaces
"Easy does it" =~ **m/^\w+\s+\w+\s+\w+$/** ==> TRUE
Just matches "gates@microsoft" — \w does not match the "."
"bill.gates@microsoft.com" =~ **m/\w+@\w+/** ==> TRUE
Add the .'s to get the whole thing
"bill.gates@microsoft.com" =~ **m/^(\w|\.)+@(\w|\.)+$/** ==> TRUE
words separated by commas and possibly spaces
"Klaatu, barada,nikto" =~ **m/^\w+(,\s*\w+)*$/** ==> TRUE

13.22 SUBROUTINES

Perl subroutines encapsulate blocks of code in the usual way. You do not need to define subroutines before they are used, so Perl programs generally have their "main" code first, and their subroutines laid out toward the bottom of the file. A subroutine can return a scalar or an array.

```
$x = Three();          ## call to Three() returns 3
exit(0);               ## exit the program normally
sub Three {
   return (1 + 2);
}
```

13.22.1 Local Variables and Parameters

Historically, many Perl programs leave all of their variables global. It's especially convenient since the variables do not need to be declared. This "quick 'n dirty" style does not scale well when trying to write larger programs. With Perl 5, the "my" construct allows one or more variables to be declared. (Older versions of perl had a "local" construct which should be avoided.)

```
my $a;                          ## declare $a
my $b = "hello"                 ## declare $b, and assign it "hello"
my @array = (1, 2, 3);          ## declare @array and assign it
                                (1, 2, 3)
my ($x, $y);                    ## declare $x and $y
my ($a, $b) = (1, "hello");     ## declare $a and $b, and assign
                                $a=1, $b="hello"
```

The "my" construct is most often used to declare local varaiables in a subroutine...

```
sub Three {
   my ($x, $y);       # declare vars $x and $y
   $x = 1;
   $y = 2;
   return ($x + $y);
}

# Variant of Three() which inits $x and $y with the array trick
sub Three2 {
   my ($x, $y) = (1, 2);
   return ($x + $y);
}
```

13.22.2 @_ Parameters

Perl subroutines do not have formal named parameters like other languages. Instead, all the parameters are passed in a single array called "@_". The elements in @_ actually point to the original caller-side parameters, so the called function is responsible for making any copies. Usually the subroutine will pull the values out of @_ and copy them to local variables. A Sum() function which takes two numbers and adds them looks like...

```
sub Sum1 {
   my ($x, $y) = @_;        # the first lines of many functions look like
this
                            # to retrieve and name their params
   return($x + $y);
}
```

```
# Variant where you pull the values out of @_ directly
# This avoids copying the parameters
sub Sum2 {
   return($_[0] + $_[1]);
}

# How Sum() would really be written in Perl -- it takes an array
# of numbers of arbitrary length, and adds all of them...
sub Sum3 {
   my ($sum, $elem);          # declare local vars
   $sum = 0;
   foreach $elem (@_) {
     $sum += $elem;
   }
   return($sum);
}

## Variant of above using shift instead of foreach
sub sum4 {
   my ($sum, $elem);
   $sum = 0;
   while(defined($elem = shift(@_))) {
     $sum += $elem;
   }
   return($sum);
}
```

13.23 INTRODUCTION TO SED

How to use sed, a special editor for modifying files automatically? If you want to write a program to make changes in a file, sed is the tool to use. *Sed* is the ultimate **s**tream **ed**itor.

Anyhow, *sed* is a marvelous utility. Unfortunately, most people never learn its real power. The language is very simple, but the documentation is terrible. The Solaris on-line manual pages for *sed* are five pages long, and two of those pages describe the 34 different errors you can get. A program that spends as much space documenting the errors than it does documenting the language has a serious learning curve.

13.23.1 The Essential Command: s for Substitution

Sed has several commands, but most people only learn the substitute command: *s*. The substitute command changes all occurrences of the regular expression into a new value. A simple example is changing "day" in the "old" file to "night" in the "new" file:

```
sed s/day/night/ <old >new
```

Or another way

```
sed s/day/night/ old >new
```

and for those who want to test this:

```
echo day | sed s/day/night/
```

This will output "night".

"Using the strong (single quote) character, that would be:

```
sed 's/day/night/' <old >new
```

I must emphasize the the sed editor changes exactly what you tell it to. So if you executed

```
echo Sunday | sed 's/day/night/' <old >new
```

This would output the word "Sunnight" bacause sed found the string "day" in the input.

There are four parts to this substitute command:

s	Substitute command
/../../	Delimiter
day	Regular Expression Pattern Search Pattern
night	Replacement string

The search pattern is on the left hand side and the replacement string is on the right hand side.

13.23.2 The Slash as a Delimiter

The character after the *s* is the delimiter. It is conventionally a slash, because this is what *ed*, *more*, and *vi* use. It can be anything you want, however. If you want to change a pathname that contains a slash—say /usr/local/bin to /common/bin—you could use the backslash to quote the slash:

```
sed 's/\/usr\/local\/bin/\/common\/bin/' <old >new
```

13.23.3 Using & as the Matched String

Sometimes you want to search for a pattern and add some characters, like parenthesis, around or near the pattern you found. It is easy to do this if you are looking for a particular string:

```
sed 's/abc/(abc)/' <old >new
```

This won't work if you don't know exactly what you will find. How can you put the string you found in the replacement string if you don't know what it is?

The solution requires the special character "&." It corresponds to the pattern found.

```
sed 's/[a-z]*/(&)/' <old >new
```

You can have any number of "&" in the replacement string. You could also double a pattern, *e.g.*, the first number of a line:

```
% echo "123 abc" | sed 's/[0-9]*/& &/'
123 123 abc
```

Let me slightly amend this example. Sed will match the first string, and make it as greedy as possible. The first match for '[0–9]*' is the first character on the line, as this matches zero of more numbers. So if the input was "abc 123" the output would be unchanged. A better way to duplicate the number is to make sure it matches a number:

```
% echo "123 abc" | sed 's/[0-9][0-9]*/& &/'
123 123 abc
```

The string "abc" is unchanged, because it was not matched by the regular expression.

```
echo abcd123 | sed 's/\([a-z]*\).*/\1/'
```

This will output "abcd" and delete the numbers.

If you want to switch two words around, you can remember two patterns and change the order around:

```
sed 's/\([a-z]*\) \([a-z]*\)/\2 \1/'
```

Note the space between the two remembered patterns. This is used to make sure two words are found.

The "\1" doesn't have to be in the replacement string (in the right hand side). It can be in the pattern you are searching for (in the left hand side). If you want to eliminate duplicated words, you can try:

```
sed 's/\([a-z]*\) \1/\1/'
```

You can have up to nine values: "\1" thru "\9."

13.23.4 /g-Global Replacement

Most Unix utilties work on files, reading a line at a time. *Sed*, by default, is the same way. If you tell it to change a word, it will only change the first occurrence of the word on a line. You may want to make the change on every word on the line instead of the first. For an example, let's place parentheses around words on a line. Instead of using a pattern like "[A-Za-z]*" which won't match words like "won't," we will use a pattern, "[^]*," that matches everything except a space. Well, this will also match anything because "*" means **zero** or **more**. The following will put parenthesis around the first word:

```
sed 's/[^ ]*/(&)/' <old >new
```

If you want it to make changes for every word, add a "g" after the last delimiter and use the work-around:

```
sed 's/[^ ][^ ]*/(&)/g' <old >new
```

13.23.5 /p-Print

By default, *sed* prints every line. If it makes a substitution, the new text is printed instead of the old one. If you use an optional argument to sed, "sed -n," it will not, by default, print any

new lines. I'll cover this and other options later. When the "-n" option is used, the "p" flag will cause the modified line to be printed. Here is one way to duplicate the function of *grep* with *sed*:

```
sed -n 's/pattern/&/p' <file
```

13.23.6 Write to a File with /w Filename

There is one more flag that can follow the third delimiter. With it, you can specify a file that will receive the modified data. An example is the following, which will write all lines that start with an even number to the file *even*:

```
sed -n 's/^[0-9]*[02468] /&/w even' <file
```

In this example, the output file isn't needed, as the input was not modified. You must have exactly one space between the *w* and the filename. You can also have ten files open with one instance of *sed*. This allows you to split up a stream of data into separate files. Using the previous example combined with multiple substitution commands described later, you could split a file into ten pieces depending on the last digit of the first number. You could also use this method to log error or debugging information to a special file.

13.23.7 Multiple Commands with -e Command

One method of combining multiple commands is to use a *-e* before each command:

```
sed -e 's/a/A/' -e 's/b/B/' <old >new
```

A "-e" isn't needed in the earlier examples because *sed* knows that there must always be one command. If you give *sed* one argument, it must be a command, and *sed* will edit the data read from standard input.

13.23.8 Filenames on the Command Line

You can specify files on the command line if you wish. If there is more than one argument to *sed* that does not start with an option, it must be a filename. This next example will count the number of lines in three files that don't begin with a "#:"

```
sed 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l
```

The *sed* substitute command changes every line that starts with a "#" into a blank line. *Grep* was used to filter out empty lines. *Wc* counts the number of lines left. *Sed* has more commands that make *grep* unnecessary. But I will cover that later.

Of course you could write the last example using the "-e" option:

```
sed -e 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l
```

13.23.9 sed -n: no Printing

The "-n" option will not print anything unless an explicit request to print is found. I mentioned the "/p" flag to the substitute command as one way to turn printing back on. Let me clarify this. The command

```
sed 's/PATTERN/&/p' file
```

acts like the *cat* program if PATTERN is not in the file: *e.g.*, nothing is changed. If PATTERN is in the file, then each line that has this is printed twice. Add the "-n" option and the example acts like grep:

```
sed -n 's/PATTERN/&/p' file
```

Nothing is printed, except those lines with PATTERN included.

13.23.10 sed -f Scriptname

If you have a large number of *sed* commands, you can put them into a file and use

```
sed -f sedscript <old >new
```

where *sedscript* could look like this:

```
# sed comment—This script changes lower case vowels to upper case
s/a/A/g
s/e/E/g
s/i/I/g
s/o/O/g
s/u/U/g
```

When there are several commands in one file, each command must be on a separate line.

13.23.11 sed in Shell Script

If you have many commands and they won't fit neatly on one line, you can break up the line using a backslash:

```
sed -e 's/a/A/g' \
-e 's/e/E/g' \
-e 's/i/I/g' \
-e 's/o/O/g' \
-e 's/u/U/g' <old >new
```

13.23.12 Using sed in a Shell here-is Document

You can use *sed* to prompt the user for some parameters and then create a file with those parameters filled in. You could create a file with dummy values placed inside it, and use *sed* to change those dummy values. A simpler way is to use the "here is" document, which uses part of the shell script as if it were standard input:

```
#!/bin/sh
echo -n 'what is the value?'
read value
sed  's/XXX/'$value'/' <<EOF
The value is XXX
EOF
```

When executed, the script says:

what is the value?

If you type in "123," the next line will be:

The value is 123

I admit this is a contrived example. "Here is" documents can have values evaluated without the use of sed. This example does the same thing:

```
#!/bin/sh
echo -n 'what is the value? '
read value
cat <<EOF
The value is $value
EOF
```

13.23.13 Delete with d

Using ranges can be confusing, so you should expect to do some experimentation when you are trying out a new script. A useful command deletes every line that matches the restriction: "d." If you want to look at the first 10 lines of a file, you can use:

```
sed '11,$ d' <file
```

which is similar in function to the *head* command. If you want to chop off the header of a mail message, which is everything up to the first blank line, use:

```
sed '1,/^$/ d' <file
```

You can duplicate the function of the *tail* command, assuming you know the length of a file. *Wc* can count the lines, and *expr* can subtract 10 from the number of lines. A Bourne shell script to look at the last 10 lines of a file might look like this:

```
#!/bin/sh
#print last 10 lines of file
# First argument is the filename
lines=`wc -l $1 | awk '{print $1}' `
start=`expr $lines - 10`
sed "1,$start d" $1
```

13.23.14 Print Line Number with =

The "=" command prints the current line number to standard output. One way to find out the line numbers that contain a pattern is to use:

```
# add line numbers first,
# then use grep,
# then just print the number
cat -n file | grep 'PATTERN' | awk '{print $1}'
```

The *sed* solution is:

```
sed -n '/PATTERN/ =' file
```

Earlier I used the following to find the number of lines in a file

```
#!/bin/sh
lines=`wc -l file | awk '{print $1}' `
```

Using the "=" command can simplify this:

```
#!/bin/sh
lines=`sed -n '$=' file `
```

The "=" command only accepts one address, so if you want to print the number for a range of lines, you must use the curly braces:

```
#!/bin/sh
# Just print the line numbers
sed -n '/begin/,/end/ {
=
d
}' file
```

Since the "=" command only prints to standard output, you cannot print the line number on the same line as the pattern. You need to edit multi-line patterns to do this.

13.24 BRIEF HISTORY OF LINUX

Linux is an operating system, a software program that controls your computer. Most vendors load an operating system onto the hard drive of a PC before delivering the PC, so, unless the hard drive of your PC has failed, you may not understand the function of an operating system.

In 1991 Linus Torvalds began developing an operating system kernel, which he named "Linux" [Torvalds 1999]. This kernel could be combined with the FSF material and other components (in particular some of the BSD components and MIT's X-windows software) to produce a freely-modifiable and very useful operating system. This book will term the kernel itself the "Linux kernel" and an entire combination as "Linux". Note that many use the term "GNU/Linux" instead for this combination.

Like other server operating systems, Linux provides advanced disk management (RAID), which makes it possible to automatically duplicate stored data on several hard drives. This greatly improves the reliability of data storage; if one hard drive fails, the data can be read from another. Competing desktop operating systems such as Microsoft Windows 95/98 do not support this capability (though several third parties sell drivers that let you add this capability to your desktop operating system).

Moreover, many Linux users run Linux not as a desktop computer but as a server, which is powered up and on-online 24 hours per day, connected to the Internet, and ready to provide services to requesting clients. For example, many Linux users run web servers, hosting web sites browsed by users worldwide. But, the number of desktop Linux users — those who power on their computer to use it and power it off when they're done—is rising.

13.24.1 The Origins of Linux

Linux traces its ancestry back to a mainframe operating system known as Multics (Multiplexed Information and Computing Service). Begun in 1965, Multics was one of the first multi-user computer systems and remains in use today. Bell Telephone Labs participated in the development of Multics, along with the Massachusetts Institute of Technology and General Electric.

Two Bell Labs software engineers, Ken Thompson and Dennis Richie, worked on Multics until Bell Labs withdrew from the project in 1969. One of their favorite pastimes during the project had been playing a multi-user game called Space Travel. Now, without access to a Multics computer, they found themselves unable to indulge their fantasies of flying around the galaxy. Resolved to remedy this, they decided to port the Space Travel game to run on an otherwise unused PDP-7 computer. Eventually, they implemented a rudimentary operating system they named *Unics*, as a pun on *Multics*. Somehow, the spelling of the name became *Unix*.

13.24.2 The X Window System

Another important component of Linux is its graphical user interface, the X Window System. Unix was originally a mouseless, text-based system that used noisy teletype machines rather than modern CRT monitors. The Unix command interface is very sophisticated and, even today, some power users prefer it to a point-and-click graphical environment, using their CRT monitor as though it were a noiseless teletype. Consequently, some remain unaware that Unix long ago outgrew its text-based childhood, and now provides users a choice of graphical or command interfaces.

The X Window System (or simply X) was developed as part of the Massachusetts Institute of Technology's (MIT) Project Athena, which it began in 1984. By 1988, MIT released X to the public. MIT has since turned development of X over to the X Consortium, which released version 6 in September 1995.

X is a unique graphical user interface in two major respects. First, X integrates with a computer network, letting users access local and remote applications. For example, X lets you open a window that represents an application running on a remote server: the remote server does the heavy-duty computing; all your computer need do is pass the server your input and display the server's output.

Second, X lets you configure its look and feel to an amazing degree. To do so, you run a special application—called a *window manager*—on top of X. A variety of window managers is available, including some that closely mimic the look and feel of Microsoft Windows.

13.24.3 Linux Distributions (Various Flavours of Linux.)

Because Linux can be freely redistributed, you can obtain it in a variety of ways. In the Linux community, different organizations have combined the available components differently. Each combination is called a "distribution", and the organizations that develop distributions are called "distributors". Various individuals and organizations package Linux, often combining it with free or proprietary applications. The flavor's of Linux are:

(*i*) Caldera OpenLinux

(*ii*) Red Hat Linux

(*iii*) Slackware Linux

(*iv*) Debian Linux

(*v*) SuSE. Linux

(*vi*) GNU/Linux

(*vii*) Mandriva(former MandrakeSoft)

(*viii*) Ubuntu

(*ix*) **Knoppix:** an operating system that runs from your CD-ROM, you don't need to install anything.

Caldera, Red Hat, Slackware, and SuSE are packaged by commercial companies, which seek to profit by selling Linux-related products and services. Debian GNU/Linux is the product of volunteer effort conducted under the auspices of Software In The Public Interest, Inc., a non-profit corporation. The Linux kernel is not part of the GNU project but uses the same license as GNU software.

Most Linux distributions offer a set of programs for generic PCs with special packages containing optimized kernels for the x86 Intel based CPUs. These distributions are well-tested and maintained on a regular basis, focusing on reliant server implementation and easy installation and update procedures. Examples are Debian, Ubuntu, Fedora, SuSE and Mandriva, which are by far the most popular Linux systems and generally considered easy to handle for the beginning user, while not blocking professionals from getting the most out of their Linux machines. Linux also runs decently on laptops and middle-range servers. Drivers for new hardware are included only after extensive testing, which adds to the stability of a system.

13.24.4 Advantage of Linux

1. **Low Cost:** You do not need to spend time and money to obtain License. Since Linux and much of it is software come with GNU general public license. These are large repository from which you can freely download high quality software.
2. **Stability:** Linux does not need to be rebooted periodically to maintain performance levels. It does not freeze up or slow down over time due to memory leaks.
3. **Performance:** Linux provides persistent high performance on workstation and on networks. It can handle unusual large number of user simultaneously.
4. **Network Friendliness:** Linux was developed by a group of performers over the internet and has therefore strong support for network functionality. It can perform task's such as network backup faster and more reality than alternative system.
5. **Flexibility:** Linux can be used for high performance server application, desktop application and embedded system.
6. **Compatibility:** Linux run on all common Unix software packages and can process all common file format.
7. **Choice:** The large number of Linux distribution gives you a choice you can pick the one you like the best; the core functionality are the same more software run on most of the distribution.

8. **Fast and Easy Installation:** Most Linux distribution come with user friendly installation and setup programs popular Linux distribution come with tools that make installation of additional software very user friendly as well.
9. **Full use of Hard Disk**: Linux continues to work well even when the hard disk is almost full.
10. **Multitasking:** Linux is designed to do many thing at the same time. For example a large printing job in the background would not slow down your other work.
11. **Security:** Linux is one of the most secure Operating System. Linux user have option to select and safely download software, free of charge from online repository containing thousand of high quality packages. No purchase transactions requiring credit card number and other sensitive personal information.
12. **Open Source:** If You develop software that require knowledge or modification of the operating system code, Linux source code at your fingertips. Linux application open source as well.

13.24.5 Difference between Linux and Unix

S.No.	Linux	Unix
1	Linux is free	Unix is not free
2	Linux run on many hardware platform	Unix is proprietary hardware bounded
3	Linux is inexpensive	Linux is expensive
4	Linux feels very much like DOS/WINDOWS in the late 80's/90's	Unix feel like a mainframe from the 60's/ 70's
5	Linux is more of average who wants to run his own server or engineering workstation.	Unix may considered more mature in certain areas like security, engineering application, better support for cutting edge hardware.
6	Linux lacks in this regard because of difficulty of large application to be certified.	Commercial version of Unix have historically large amount of support for enterprise level application such as oracle or SAP.
7	Linux can change kindly not only between different distribution but sometimes between release of same distribution which makes it difficult to understand the exact environment in which there tool will be use.	Unix do not change very much from realest to release.
8	The Linux code base is constantly changing.	Unix code base do not change there fore it work for longer.

13.24.6 Structure of Linux Operating System

The Linux Operating System is divided into three major components—the Kernel, the Shell and Utilities and Application Programs.

- **The-Kernel**

  This is the heart of the Linux Operating System. It performs the tasks that create and maintain the Linux environment. It keeps track of the disks, tapes, printers, terminals, communication lines and any other devices attached to the computer. It also interfaces between the computer's hardware and the users.

- **The-Shell**

  This is a program that interfaces between the user and the Linux Operating System. It listens to the user's terminal and translates the actions requested by the user. There are a number of different Shells that may be used. See section on The Shell.

- **Utilities and Application Programs**

  Utilities are the Linux Commands. Application Programs, such as Word Processors, Spreadsheets and Database Management Systems, may be installed alongside the Linux Commands. A user may run a utility or application through the shell.

The-Shell

The-Shell is a program that provides an interpreter and interface between the user and the Linux Operating System. It executes commands that are read either from a terminal or from a file. Files containing commands may be created, allowing users to build their own commands. In this manner, users may tailor Linux to their individual requirements and style.

There are a number of different Shells. Each provides a slightly different interface between the user and the Linux Operating System. The most important shells that originated from the Unix operating system are:

There are other shells that are less widely used and not available on many machines. For example, there is the Restricted Shell-rsh. This restricts the area of memory the user may access to his or her own directory, thus limiting access to all other users' files.

All of these shell interfaces are available to Linux. However, there are other shells that have been developed since, most generally for Linux: ash, tcsh and zsh are available on most versions of Linux. However, the most widely used, originally Linux-based shell is the Bourne-Again shell (bash). Based on the original Bourne shell, it has similar extensions as the Korn shell, plus its own further extensions.

Linux also offers a windows-based shell interface, commonly known as X-Windows or simply as X. More akin to the Mackintosh windows than Microsoft windows, it is another method of interfacing with the Linux kernel. However, X-Windows interfaces are not considered on this course.

A command issued by a user may be run in the present shell, or the shell itself may start up another copy of itself in which to run that command. In this way, a user may run several commands at the same time. A secondary shell is called a sub-shell.

When a user logs onto the System, a shell is automatically started. This will monitor the user's terminal, waiting for the issue of any commands. The type of shell used is stored in a file called passwd in the subdirectory etc. (see Section 3.2). Any other shell may be run as a sub-shell by issuing it as a command. For example, /usr/bin/ksh will run a Korn shell. The original shell will still be running—in background mode—until the Korn shell is terminated.

Users

Linux is a multi-user operating system. Each user will require to create and access his or her own files. These files must be secure from other users on the system. Because of this, each user has a unique identification on a Linux system, with the option of a password to enhance security.

There are two types of user on a Linux system:

- **Ordinary Users**

  An ordinary user has a Home Directory under which files and sub-directories are normally stored. After logging onto the system, a user is normally taken directly to that directory.

  An ordinary user is a member of a Group of users. For security reasons, files (and directories) owned by a user may be accessed and used by the user, other members of the user's group and all other users at different levels of permission. For example, a file may be read and altered by the user that owns it, may only be read by other members of the same group and may not be accessed at all by any other user.

- **Super-User**

  A super-user is a privileged user who has full access to all files, regardless to whoever owns them or what their access permissions are:—

  The super-user has a position of responsibility: to administer and maintain the system.

  The super-user is normally known as root. root's Home directory is the primary directory of the system, under which all other directories and all files are stored.

EXERCISES

1. Explain the system structure of UNIX operating system.
2. Describe the file and directory structure of UNIX.
3. Define operating system services.
4. How many level used by a process when process is running on UNIX system? Describe in detail.
5. Draw the block diagram of the system kernel. And explain its working.
6. What is the process and context of a process? Define process states and transition.
7. What should happen? If the kernel attempts to awaken all process sleeping on an event, but no process are asleep on the event at the time of the wakeup.
8. What is the buffer header and during system initialization why kernel allocates space for a number of buffers. Suppose the kernel does a delayed write of a block. What happens when another process takes that block from its hash queue? From the free list?
9. In the algorithm getblk (Algorithm for buffer allocation), if the kernel removes the buffer from the free list, it must raise the processor priority level to block out interrupts before checking the free list. Why?
10. Suppose the kernel does a delayed write of a block. What happens when another process takes that block from its hash queue?
11. Find the physical location of byte offset 265100 in UNIX file system when the block size is 2048 and rest of the architecture remains unchanged.
12. Find the largest possible file size in UNIX OS when it contains 10 direct address, 1 single indirect address, 1 double indirect address and no triple indirect. Each indirect address (single and double indirect) points to 1 block of 2K(2048 bytes) each. Block numbers are addressable by 32 bits.
13. Describe an algorithm that ask for and receives any free buffer from the buffer pool.
14. What do you understand by forward and backward search?
15. Define system response time as the average time it takes to complete a system cell. Describe how the buffer cache can help response time. Does it necessary help system throughput?

16. Write an algorithm getblk for buffer allocation.
17. Describe how the buffer cache can help response time. Does it necessarily help system throughput.
18. The C language convention count array from 0. Why do Inode numbers start from 1 and not 0?
19. If a process sleep in algorithm iget when it finds the Inode locked in the cache, why must it start the loop again from the beginning after waking up?
20. Explain the three modes of vi editor.
21. Describe an algorithm that takes an in-core Inode as input and updates the corresponding disk Inode.
22. Discuss a system implementation that keep tracks of a free disk block with a bit map instead of a linked list of block. What are the advantage and disadvantage of this scheme.
23. Write down the algorithm for disk block allocation.
24. How the Inode assign to a new created file?
25. Define the following terms:
    (*i*) File-subsystem
    (*ii*) Zombic state of a process
    (*iii*) User file descriptor
26. Write the short notes on following operator:
    (*i*) d-delete
    (*ii*) c-change
    (*iii*) y-yank
    (*iv*) !-filter
27. Write the short notes on:
    (i) Boot Block
    (ii) Super Block
    (iii) Inode Block
    (iv) Data Block
28. Design a directory structure that improve the efficiency of searching for path name by avoiding the linear search.
29. Describe the structure of a regular file.
30. Discuss the salient features of Memory Management in UNIX.
31. What do you mean by System call? Write any 10 system calls.
32. What do you understand by parent and children process? Also explain the status of a process.
33. Describe the Fork() system call. Why doesn't Fork return the Process ID of the parent to child and return zero to parent?

34. Discuss Unlink system call. How do you unlink an opened file?
35. Suppose a directory has read permission for a user but not execute permission. What happens when the directory is used as a parameter to is with the "-i" option? What about the –l option?
36. What strange things could happen if the kernel would allow two process to mount the same file system simultaneously at two mount points?
37. When executing the command ls –ld on a directory, note that the number of link to the directory is never 1. Why?
38. Explain in detail premature termination of a process.
39. How is /etc/passwd updated by any user, while changing his password, even though the file does not the write permission?
40. Design an algorithm that translate virtual address to physical address, given the virtual address and the address of the pregion entry.
41. Design an algorithm for allocating and freeing memory pages and page tables. What data structure would allow best performance or simplest implementation?
42. Its possible to implement the system such that the kernel stack grows on top of the user stack. Discuss the advantage and disadvantage of such an implementation.
43. Suppose a process goes to sleep and the system contains no process ready to run. What happens when the sleeping process does its context switch?
44. What happens if the kernel issue a makeup call for all process asleep on address A, but no process are asleep on that address at time?
45. Explain the following UNIX command for communication:
    (*i*) news
    (*ii*) mail
    (*iii*) wall
    (*iv*) write
    (*v*) mesg
    (*vi*) crontab
46. How the Fork () system call create a new process. Write an algorithm for Fork system call.
47. Draw the process state diagram and algorithm for checking and handling signals.
48. Write shell script that prints the current date, user name and the name of your login shell.
49. When executing the command ls –ld on a directory, the number of links of the directory is never 1. Why?
50. When the shell creates a new process to execute a command, how does it know that the file is executable? If it is executable, how does it distinguish between a shell script and a file produced by a compilation? What is the correct sequence for checking the above cases?

51. Write a menu driven program which has the following options:
    (*i*) Contents of /etc/passwd
    (*ii*) Present working Directory
    (*iii*) Lists of users who have currently logged in
    (*iv*) Exit
52. Illustrate the development of open general public licence in case of Linux OS. Give the history of development of Linux Operating System.
53. The algos iget and iput do not require the processor execution level to be raised to block out interrupt what this imply.
54. Describe the implementation of the kill system call.
55. A process check for signal when it enters or leave the sleep state and when it returns to user mode from the kernel after completion of a system call or after handling a interrupt. Why does the process not have to check for signals when entering the system for execution of a system call?
56. Explain the security problem that exist if a setuid program is not write-protected.
57. If Anoop uses su command to become a super user, he can't execute any of the shell script in his directory. Explain with reason.
58. Explain the use of following shell variables: $#, $*, $@ and $.
59. Write a shell script that reports in descending order of their size, name and size of all files whose size exceeds 40 bytes in a specific directory (Supplied as an argument). Total number of search files is also displayed.
60. Mention different grep family of commands and explain each one of them very briefly. Is it possible to use multiple search pattern with all the grep family of commands.
61. Write a shell script that would pickup all 'C' program files from the current directory and add the extension '.CPP' at the end of each such file.
62. Write sed command to count the number of students born in the year 1977 from the database.
63. Discuss how one can input insert text before the contents of input file using sed.
64. When the shell create a new process to execute a command. How does it know that the file is executable? If its executable how does it distinguish between a shell script and a file produce by a compilation? What is the correct sequence for checking the above case?
65. What is the function of following UNIX commands? Explain with example by writing proper syntax of these command:
    (*i*) tr.
    (*ii*) awk
    (*iii*) grep, egrep, fgrep
    (*iv*) finger
    (*v*) batch

(*vi*) bc

(*vii*) sort

(*viii*) cut

(*ix*) copy

(*x*) umask

66. What do you understand by mounting and unmounting a file system in UNIX? How is this achieved?

67. What are the basic function of shell? Discuss different types of shell used in UNIX OS.

68. What is shell programming? Write a shell script for tacking the backup in UNIX.

69. What is the meaning of password file and group file? Write the different entries existing in these files.

70. Explain the case statement in UNIX.

71. Write the algorithm for process scheduling. Write the scheduling parameter.

72. Write the algorithm for client process and receiving message.

73. How the shared memory attach once or twice to a process?

74. Describe the sockets model in detail.

75. What is the problem of a multiprocessor system? How it can be solved with master and slave processor and with semaphore?

76. Describe the Linux structure and also define the feature of Linux.

77. What are the specials built in pattern in awk? Describe these patterns.

78. What are the advantage of delayed write mechanism?

79. Write an algorithm for allocation of a buffer for a block. Trace your algorithm for all the possible variation in input data.

80. What are the contents of a incore inode and what additional information is to be stored in in-core inode and why?

81. Give the layout of the UNIX system memory. Describe each section.

82. Write down the security features of UNIX.

83. What are awk patterns? Describe BEGIN and END patterns.

84. Describe in brief any one technique of process synchronization used in UNIX.

85. What are pipes? Differentiate between named and unnamed pipes.

86. Describe how write system calls work. What are its input parameters and returns information. Describe with the help of algorithm.

87. Discuss the structure of a regular file. How byte offset can be converted into a block number give algorithm?

88. Write short notes on:

(*i*) Features of linux

(*ii*) Device Deliver

(*iii*) Mounting of a file system

(*iv*) Limitation/problems of multiprocessor system.

(*v*) Flavors of Linux

89. How will you replace the string "Linux" by "Red Hat Linux"?

90. Explain Lists, Arrays and Lashe's in perl. How can you obtain the following:

(*i*) Only keys in associative arrays

(*ii*) Only values in associative arrays

(*iii*) Delete an element from associative arrays

(*iv*) Insert an element from associative arrays

91. Write a **sed** sequence to find out the number of occurrences of a pattern in a file.

92. Discuss the structure of awk script. Explain the operational mechanism of awk. Mention different awk print function and list the differences in their behaviour, if any between them.

93. What are associative arrays? A file books list holds the number of books sold in different engineering discipline per month. Write awk script that finds total number of books sold in each of the disciplines as well as total number of books sold.

94. How will you remove all trailing spaces from a file?

95. Use **sed** to insert two spaces at the beginning of each line.

96. Print the string "Anoop" 15 times without using the loop.

97. What is perl programming? Write a **perl** script to convert binary number (supplied as argument) to decimal.

98. Find out the occurrences of two consecutives and identical word character (like **aa** or **bb**) using (*i*) grep (*ii*) sed (*iii*) perl.

99. Write a perl script to print lines in reverse order.

100. Explain Linux security features.

INDEX

T

U

ABOUT THE BOOK

This book, designed to meet the course on **UNIX and SHELL Programming** for graduate and undergraduate engineering students, presents an illuminative and objective exploration of algorithms and programming skills on UNIX and SHELL. The authors discuss all the concepts such as operating system services, file system, buffer cache, reading and writing disk blocks, inode assignment to a new file, system calls, structure and control of a process, inter-process communication, and sockets with the help of illustrations, tables, block diagrams, and industry based practical examples. This volume also examines the detail of UNIX command, SHELL, AWK, and PERL programming with a large number of practical examples supported by their algorithm.

Written in a student-friendly manner the book is enriched with the following features:

- Algorithms for a number of problems have been given to inculcate real programming skills
- Describe and compare the algorithm used in UNIX system to that used in other operating systems
- Syntax for a large number of UNIX command
- SHELL script for a number of real life problems
- Brief of AWK and PERL programming
- Straight forward exercises to illustrate the concepts at the end of the book

The book is useful for general students, engineering students of graduate and postgraduate level, subject teachers, professionals, application programmers, and industry specialist associated with the UNIX and SHELL programming.

ABOUT THE AUTHORS

Prof. Anoop Chaturvedi, an M.Tech. in Computer Science from RGPV Bhopal (MP), is presently working as an Associate Professor in the Department of Computer Science & Engineering at LNCT, Bhopal. An expert on UNIX and SHELL programming, he has more than 10 years of teaching experience and 15 national and 3 international papers to his credit.

Prof. B. L. Rai, who did his M.Tech. from RGPV Bhopal (MP) is presently working as an Associate Professor in the Department of Computer Science & Engineering at JNCT, Bhopal. With a more than 9 years teaching experience, he has published more than 15 papers in journals of national and international repute.

ISBN 978-93-81159-05-7

9 789381 159057

UNIVERSITY SCIENCE PRESS