DIGITAL NOTES
ON
LINUX PROGRAMMING

B.TECH III- YEAR – I-SEM
(2018-19)

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF INFORMATION TECHNOLOGY

III Year B.Tech. IT - I Sem	L	T/P/D	C
	5	1/-/-	4

(R15A0527)LINUX PROGRAMMING

Objectives:

- To develop the skills necessary for Unix systems programming including file system programming, process and signal management, and interprocess communication.
- To make effective use of Unix utilities and Shell scripting language such as bash.
- To develop the basic skills required to write network programs using Sockets.

UNIT I

Linux Utilities-File handling utilities, Security by file permissions, Process utilities, Disk utilities, Networking commands, Filters, Text processing utilities and Backup utilities.
Sed-Scripts, Operation, Addresses, Commands, Applications, awk- Execution, Fields and Records, Scripts, Operation, Patterns, Actions, Associative Arrays, String and Mathematical functions, System commands in awk, Applications.
Shell programming with Bourne again shell(bash)- Introduction, shell responsibilities, pipes and Redirection, here documents, running a shell script, the shell as a programming language, shell meta characters, file name substitution, shell variables, command substitution, shell commands, the environment, quoting, test command, control structures, arithmetic in shell, shell script examples, interrupt processing, functions, debugging shell scripts.

UNIT II

Files and Directories- File Concept, File types, File System Structure, file metadata-Inodes, kernel support for files, system calls for file I/O operations- open, create, read, write, close, lseek, dup2,file status information-stat family, file and record locking-lockf and fcntl functions, file permissions - chmod, fchmod, file ownership-chown, lchown, fchown, links-soft links and hard links – symlink, link, unlink. Directories-Creating, removing and changing Directories-mkdir, rmdir, chdir, obtaining current working directory-getcwd, Directory contents, Scanning Directories-opendir, readdir, closedir, rewinddir, seekdir, telldir functions.

UNIT III

Process – Process concept, Kernel support for process, process identification, process hierarchy, process states, process control - process creation, waiting for a process, process termination, zombie process, orphan process, system call interface for process management-fork, vfork, exit, wait, waitpid, exec family, system, I/O redirection
Signals – Introduction to signals, Signal generation and handling, Kernel support for signals, Signal function, unreliable signals, reliable signals, kill, raise , alarm, pause, abort, sleep functions.

UNIT IV

Interprocess Communication - Introduction to IPC, IPC between processes on a single computer system,IPC between processes on different systems, pipes-creation, IPC between related processes using unnamed pipes, FIFOs-creation, IPC between unrelated processes using FIFOs (Named pipes),differences between unnamed and named pipes, popen and pclose library functions.Message Queues- Kernel support for messages, APIs for message queues, client/server example.Semaphores-Kernel support for semaphores, APIs for semaphores, file locking with semaphores.

UNIT V

Shared Memory- Kernel support for shared memory, APIs for shared memory, shared memory example.

Sockets- Introduction to Berkeley Sockets, IPC over a network, Client-Server model, Socket address structures (Unix domain and Internet domain),Socket system calls for connection oriented protocol and connectionless protocol, example-client/server programs-Single Server-Client connection, Multiple simultaneous clients, Comparison of IPC mechanisms.

TEXT BOOKS:

1. Unix System Programming using C++, T.Chan, PHI.
2. Unix Concepts and Applications, 4th Edition, Sumitabha Das, TMH,2006.
3. Unix Network Programming, W.R.Stevens, PHI

REFERENCE BOOKS:

1. Linux System Programming, Robert Love, O'Reilly, SPD, rp-2007.
2. Unix for programmers and users, 3rd Edition, Graham Glass, King Ables, Pearson2003,
3. Advanced Programming in the Unix environment, 2nd Edition, W.R.Stevens, Pearson .
4. System Programming with C and Unix, A.Hoover, Pearson.

Outcomes:

- Students will be able to use Linux environment efficiently
- Solve problems using bash for shell scripting
- Work confidently in Unix/Linux environment

INDEX

S. No	Unit	Topic	Page no
1	I	Linux Utilities-File handling utilities, Security by file permissions, Process utilities, Disk utilities,	7-12
2		Networking commands Filters, Text processing utilities and Backup utilities.	13-18
3		Sed-Scripts, Operation, Addresses, Commands, Applications	19-21
4		awk- Execution, Fields and Records, Scripts, Operation, Patterns, Actions, Associative Arrays	22-23
5		String and Mathematical functions, System commands in awk, Applications.	24-43
6		Shell programming with Bourne again shell(bash)- Introduction, shell responsibilities, pipes and Redirection, here documents, running a shell script, the shell as a programming language, shell meta characters, file name substitution	44-49
		shell variables, command substitution, shell commands, the environment, quoting, test command, control structures	50-56
7	II	Files and Directories- File Concept, File types, File System Structure, file metadata-Inodes,	57-59
8		kernel support for files, system calls for file I/O operations- open, create, read, write, close, lseek, dup2	59-62
9		file and record locking-lockf and fcntl functions, file permissions - chmod, fchmod, file ownership-chown, lchown, fchown, links-soft links and hard links – symlink, link, unlink	63-65
		Directories-Creating, removing and changing Directories-mkdir, rmdir, chdir, obtaining current working directory-getcwd, Directory contents	66-71
10	III	Process – Process concept, Kernel support for process, process identification, process hierarchy, process states, process control - process creation, waiting for a process, process termination, zombie process	72-74
11		orphan process, system call interface for process management- fork, vfork, exit, wait, waitpid, exec family, system, I/O redirection	75-80
		Signals – Introduction to signals, Signal generation and handling, Kernel support for signals, Signal function, unreliable signals, reliable signals, kill, raise , alarm, pause, abort, sleep functions	81-86
12	IV	Interprocess Communication - Introduction to IPC, IPC between processes on a single computer system,IPC between processes on different systems, pipes-creation, IPC between related processes using unnamed pipes	87-90

UNIT-1

"Unit-I - Linux Utilities"

Introduction to Linux

Linux is a Unix-like computer operating system assembled under the model of free and open source software development and distribution. The defining component of Linux is the Linux kernel, an operating system kernel first released 5 October 1991 by Linus Torvalds.

Linux was originally developed as a free operating system for Intel x86-based personal computers. It has since been ported to more computer hardware platforms than any other operating system. It is a leading operating system on servers and other big iron systems such as mainframe computers and supercomputers more than 90% of today's 500 fastest supercomputers run some variant of Linux, including the 10 fastest. Linux also runs on embedded systems (devices where the operating system is typically built into the firmware and highly tailored to the system) such as mobile phones, tablet computers, network routers, televisions and video game consoles; the Android system in wide use on mobile devices is built on the Linux kernel.

A distribution oriented toward desktop use will typically include the X Window System and an accompanying desktop environment such as GNOME or KDE Plasma. Some such distributions may include a less resource intensive desktop such as LXDE or Xfce for use on older or less powerful computers. A distribution intended to run as a server may omit all graphical environments from the standard install and instead include other software such as the Apache HTTP Server and an SSH server such as OpenSSH. Because Linux is freely redistributable, anyone may create a distribution for any intended use. Applications commonly used with desktop Linux systems include the Mozilla Firefox web browser, the LibreOffice office application suite, and the GIMP image editor.

Since the main supporting user space system tools and libraries originated in the GNU Project, initiated in 1983 by Richard Stallman, the Free Software Foundation prefers the name *GNU/Linux*.

The Unix operating system was conceived and implemented in 1969 at AT&T's Bell Laboratories in the United States by Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna. It was first released in 1971 and was initially entirely written in assembly language, a common practice at the time. Later, in a key pioneering approach in 1973, Unix was re-written in the programming language C by Dennis Ritchie (with exceptions to the kernel and I/O). The availability of an operating system written in a high-level language allowed easier portability to different computer platforms.

Today, Linux systems are used in every domain, from embedded systems to supercomputers, and have secured a place in server installations often using the popular LAMP application stack. Use of Linux distributions in home and enterprise desktops has been growing. They have also gained popularity with various local and national governments. The federal government of Brazil is well known for its support for Linux. News of the Russian military creating its own Linux distribution has also surfaced, and has come to fruition as the G.H.ost Project. The Indian state of Kerala has gone to the extent of mandating that all state high schools run Linux on their computers.

Design

A Linux-based system is a modular Unix-like operating system. It derives much of its basic design from principles established in Unix during the 1970s and 1980s. Such a system uses a monolithic kernel, the Linux kernel, which handles process control, networking, and peripheral and file system access. Device drivers are either integrated directly with the kernel or added as modules loaded while the system is running.

Separate projects that interface with the kernel provide much of the system's higher-level functionality. The GNU userland is an important part of most Linux-based systems, providing the most common implementation of the C library, a popular shell, and many of the common Unix tools which carry out many basic operating system tasks. The graphical user interface (or GUI) used by most Linux systems is built on top of an implementation of the X Window System.

Linux Advantages

1. **Low cost:** You don't need to spend time and money to obtain licenses since Linux and much of its software come with the GNU General Public License. You can start to work immediately without worrying that your software may stop working anytime because the free trial version expires. Additionally, there are large repositories from which you can freely download high quality software for almost any task you can think of.
2. **Stability:** Linux doesn't need to be rebooted periodically to maintain performance levels. It doesn't freeze up or slow down over time due to memory leaks and such. Continuous up-times of hundreds of days (up to a year or more) are not uncommon.
3. **Performance:** Linux provides persistent high performance on workstations and on networks. It can handle unusually large numbers of users simultaneously, and can make old computers sufficiently responsive to be useful again.
4. **Network friendliness:** Linux was developed by a group of programmers over the Internet and has therefore strong support for network functionality; client and server systems can be easily set up on any computer running Linux. It can perform tasks such as network backups faster and more reliably than alternative systems.
5. **Flexibility:** Linux can be used for high performance server applications, desktop applications, and embedded systems. You can save disk space by only installing the components needed for a particular use. You can restrict the use of specific computers by installing for example only selected office applications instead of the whole suite.

6. **Compatibility:** It runs all common Unix software packages and can process all common file formats.
7. **Choice:** The large number of Linux distributions gives you a choice. Each distribution is developed and supported by a different organization. You can pick the one you like best; the core functionalities are the same; most software runs on most distributions.
8. **Fast and easy installation:** Most Linux distributions come with user-friendly installation and setup programs. Popular Linux distributions come with tools that make installation of additional software very user friendly as well.
9. **Full use of hard disk:** Linux continues work well even when the hard disk is almost full.
10. **Multitasking:** Linux is designed to do many things at the same time; e.g., a large printing job in the background won't slow down your other work.
11. **Security:** Linux is one of the most secure operating systems. "Walls" and flexible file access permission systems prevent access by unwanted visitors or viruses. Linux users have to option to select and safely download software, free of charge, from online repositories containing thousands of high quality packages. No purchase transactions requiring credit card numbers or other sensitive personal information are necessary.
12. **Open Source:** If you develop software that requires knowledge or modification of the operating system code, Linux's source code is at your fingertips. Most Linux applications are Open Source as well.

The difference between Linux and UNIX operating systems?

UNIX is copyrighted name only big companies are allowed to use the UNIX copyright and name, so IBM AIX and Sun Solaris and HP-UX all are UNIX operating systems. The Open Group holds the UNIX trademark in trust for the industry, and manages the UNIX trademark licensing program.

Most UNIX systems are commercial in nature.

Linux is a UNIX Clone

But if you consider Portable Operating System Interface (POSIX) standards then Linux can be considered as UNIX. To quote from Official Linux kernel README file:

Linux is a Unix clone written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX compliance.

However, "Open Group" do not approve of the construction "Unix-like", and consider it misuse of their UNIX trademark.

Linux is just a kernel. All Linux distributions includes GUI system + GNU utilities (such as cp, mv, ls,date, bash etc) + installation & management tools + GNU c/c++ Compilers + Editors (vi) + and various applications (such as OpenOffice, Firefox). However, most UNIX operating systems are considered as a complete operating system as everything come from a single source or vendor.

As I said earlier Linux is just a kernel and Linux distribution makes it complete usable operating systems by adding various applications. Most UNIX operating systems comes with A-Z programs such as editor, compilers etc. For example HP-UX or Solaris comes with A-Z programs.

License and cost

Linux is Free (as in beer [freedom]). You can download it from the Internet or redistribute it under GNU licenses. You will see the best community support for Linux. Most UNIX like operating systems are not free (but this is changing fast, for example OpenSolaris UNIX). However, some Linux distributions such as Redhat / Novell provides additional Linux support, consultancy, bug fixing, and training for additional fees.

User-Friendly

Linux is considered as most user friendly UNIX like operating systems. It makes it easy to install sound card, flash players, and other desktop goodies. However, Apple OS X is most popular UNIX operating system for desktop usage.

Security Firewall Software

Linux comes with open source netfilter/iptables based firewall tool to protect your server and desktop from the crackers and hackers. UNIX operating systems comes with its own firewall product (for example Solaris UNIX comes with ipfilter based firewall) or you need to purchase a 3rd party software such as Checkpoint UNIX firewall.

Backup and Recovery Software

UNIX and Linux comes with different set of tools for backing up data to tape and other backup media. However, both of them share some common tools such as tar, dump/restore, and cpio etc.

File Systems

- Linux by default supports and use ext3 or ext4 file systems.

File Handling utilities:

cat COMMAND:

cat linux command concatenates files and print it on the standard output.

SYNTAX:

The Syntax is

```
cat [OPTIONS] [FILE]...
```

OPTIONS:

-A	Show all.
-b	Omits line numbers for blank space in the output.
-e	A $ character will be printed at the end of each line prior to a new line.
-E	Displays a $ (dollar sign) at the end of each line.
-n	Line numbers for all the output lines.
-s	If the output has multiple empty lines it replaces it with one empty line.
-T	Displays the tab characters in the output.
-v	Non-printing characters (with the exception of tabs, new-lines and form-feeds) are printed visibly.

EXAMPLE:

1. To Create a new file:

   ```
   cat > file1.txt
   ```

   This command creates a new file file1.txt. After typing into the file press control+d (^d) simultaneously to end the file.

2. To Append data into the file:

   ```
   cat >> file1.txt
   ```

   To append data into the same file use append operator >> to write into the file, else the file will be overwritten (i.e., all of its contents will be erased).

3. To display a file:

   ```
   cat file1.txt
   ```

   This command displays the data in the file.

4. To concatenate several files and display:

```
cat file1.txt file2.txt
```

The above cat command will concatenate the two files (file1.txt and file2.txt) and it will display the output in the screen. Some times the output may not fit the monitor screen. In such situation you can print those files in a new file or display the file using less command.

```
cat file1.txt file2.txt | less
```

5. To concatenate several files and to transfer the output to another file.

```
cat file1.txt file2.txt > file3.txt
```

In the above example the output is redirected to new file file3.txt. The cat command will create new file file3.txt and store the concatenated output into file3.txt.

rm COMMAND:

rm linux command is used to remove/delete the file from the directory.

SYNTAX:

The Syntax is

```
rm [options..] [file | directory]
```

OPTIONS:

-f	Remove all files in a directory without prompting the user.
-i	Interactive. With this option, rm prompts for confirmation before removing any files.
-r (or) -R	Recursively remove directories and subdirectories in the argument list. The directory will be emptied of files and removed. The user is normally prompted for removal of any write-protected files which the directory contains.

EXAMPLE:

1. To Remove / Delete a file:

   ```
   rm file1.txt
   ```

   Here rm command will remove/delete the file file1.txt.

2. To delete a directory tree:

   ```
   rm -ir tmp
   ```

   This rm command recursively removes the contents of all subdirectories of the tmp directory, prompting you regarding the removal of each file, and then removes the tmp directory itself.

3. To remove more files at once

   ```
   rm file1.txt file2.txt
   ```

   rm command removes file1.txt and file2.txt files at the same time.

cd COMMAND:

cd command is used to change the directory.

SYNTAX:

The Syntax is

```
cd [directory | ~ | ./ | ../ | - ]
```

OPTIONS:

-L	Use the physical directory structure.
-P	Forces symbolic links.

EXAMPLE:

1. To Remove / Delete a file:

   ```
   rm file1.txt
   ```

   Here rm command will remove/delete the file file1.txt.

2. To delete a directory tree:

   ```
   rm -ir tmp
   ```

   This rm command recursively removes the contents of all subdirectories of the tmp directory, prompting you regarding the removal of each file, and then removes the tmp directory itself.

3. To remove more files at once

   ```
   rm file1.txt file2.txt
   ```

   rm command removes file1.txt and file2.txt files at the same time.

cd COMMAND:

cd command is used to change the directory.

SYNTAX:

The Syntax is

```
cd [directory | ~ | ./ | ../ | - ]
```

OPTIONS:

-L	Use the physical directory structure.
-P	Forces symbolic links.

EXAMPLE:

1. cd linux-command

   This command will take you to the sub-directory(linux-command) from its parent directory.

2. cd ..

   This will change to the parent-directory from the current working directory/sub-directory.

3. cd ~

   This command will move to the user's home directory which is "/home/username".

cp COMMAND:

cp command copy files from one location to another. If the destination is an existing file, then the file is overwritten; if the destination is an existing directory, the file is copied into the directory (the directory is not overwritten).

SYNTAX:

The Syntax is

```
cp [OPTIONS]... SOURCE DEST
cp [OPTIONS]... SOURCE... DIRECTORY
cp [OPTIONS]... --target-directory=DIRECTORY SOURCE...
```

OPTIONS:

-a	same as -dpR.
--backup[=CONTROL]	make a backup of each existing destination file
-b	like --backup but does not accept an argument.
-f	if an existing destination file cannot be opened, remove it and try

Process utilities:

ps COMMAND:

ps command is used to report the process status. ps is the short name for Process Status.

SYNTAX:

The Syntax is

```
ps [options]
```

OPTIONS:

-a	List information about all processes most frequently requested: all those except process group leaders and processes not associated with a terminal..
-A or e	List information for all processes.
-d	List information about all processes except session leaders.
-e	List information about every process now running.
-f	Generates a full listing.
-j	Print session ID and process group ID.
-l	Generate a long listing.

EXAMPLE:

1. ps

   Output:

   ```
   PID TTY        TIME CMD
   2540 pts/1    00:00:00 bash
   2621 pts/1    00:00:00 ps
   ```

   In the above example, typing ps alone would list the current running processes.

2. ps -f

-v	Print the operating system version.
-X	Print expanded system information, one information element per line, as expected by SCO Unix. The displayed information includes: • system name, node, release, version, machine, and number of CPUs. • BusType, Serial, and Users (set to "unknown" in Solaris) • OEM# and Origin# (set to 0 and 1, respectively)
-S systemname	The nodename may be changed by specifying a system name argument. The system name argument is restricted to SYS_NMLN characters. SYS_NMLN is an implementation specific value defined in <sys/utsname.h>. Only the super-user is allowed this capability.

Examples

uname -arv

List the basic system information, OS release, and OS version as shown below.

SunOS hope 5.7 Generic_106541-08 sun4m sparc SUNW,SPARCstation-10

uname -p

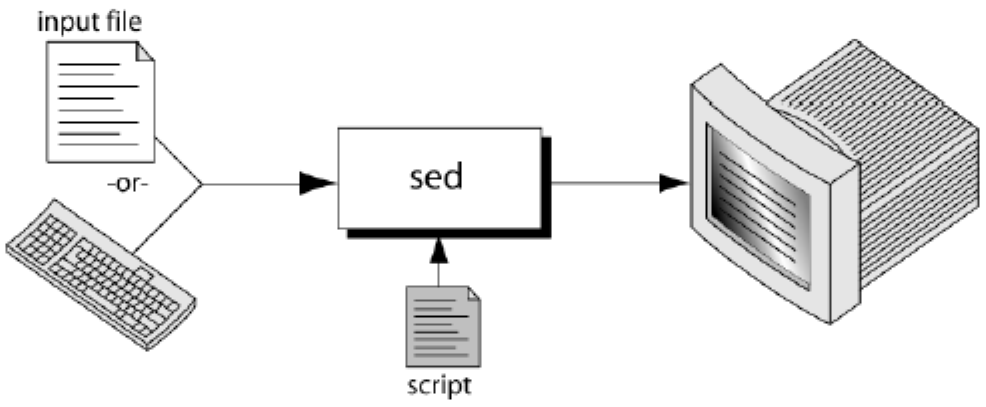
Display the Linux platform.

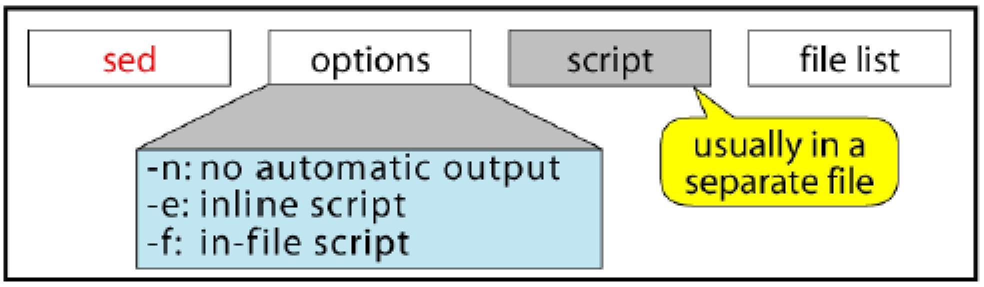
SED:

What is sed?

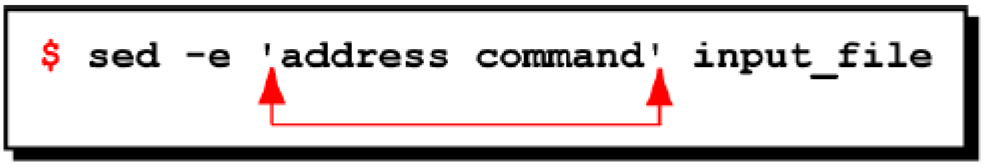
- A non-interactive stream editor

- Interprets sed instructions and performs actions
- Use sed to:
    - Automatically perform edits on file(s)
    - Simplify doing the same edits on multiple files
    - Write conversion programs





sed command syntax



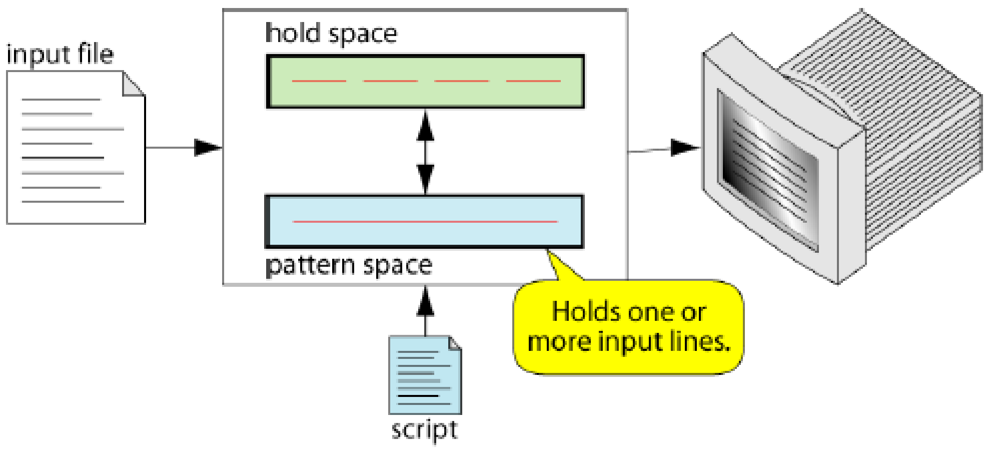
```
$ sed -e 'address command' input_file
```

(a) Inline Script

```
$ sed -f script.sed input_file
```

(b) Script File

sed Operation



How Does sed Work?

- sed reads line of input
    - line of input is copied into a temporary buffer called pattern space
    - editing commands are applied
        - subsequent commands are applied to line in the pattern space, not the original input line
        - once finished, line is sent to output

(unless –n option was used)

- line is removed from pattern space

- sed reads next line of input, until end of file

<u>Note:</u> input file is unchanged

sed instruction format

- address determines which lines in the input file are to be processed by the command(s)
    - if no address is specified, then the command is applied to each input line
- address types:
    - Single-Line address
    - Set-of-Lines address
    - Range address
    - Nested address

Single-Line Address

- Specifies only one line in the input file
    - special: dollar sign ($) denotes last line of input file

<u>Examples:</u>

- show only line 3

sed -n -e '3 p' input-file

- show only last line

sed -n -e '$ p' input-file

- substitute "endif" with "fi" on line 10

sed -e '10 s/endif/fi/' input-file

Set-of-Lines Address

- use regular expression to match lines
  - written between two slashes
  - process only lines that match
  - may match several lines
  - lines may or may not be consecutives

<u>Examples:</u>

sed -e '/key/ s/more/other/' input-file

sed -n -e '/r..t/ p' input-file

Range Address

- Defines a set of consecutive lines

<u>Format:</u>

start-addr,end-addr (inclusive)

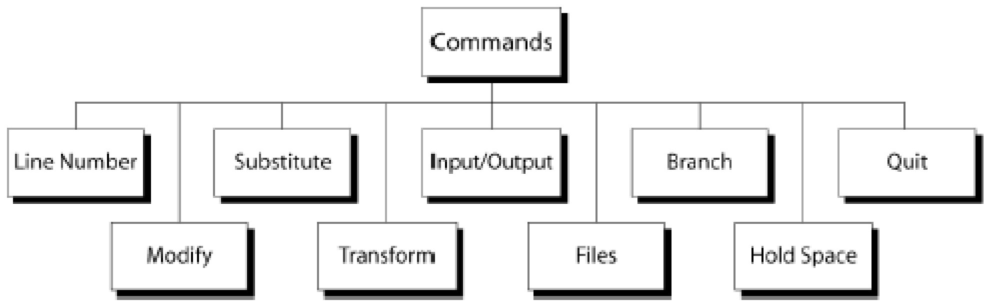
<u>Examples:</u>

Example:

print lines that do not contain "obsolete"

sed -e '/obsolete/!p' input-file

sed commands



Line Number

- line number command (=) writes the current line number before each matched/output line

Examples:

AWK

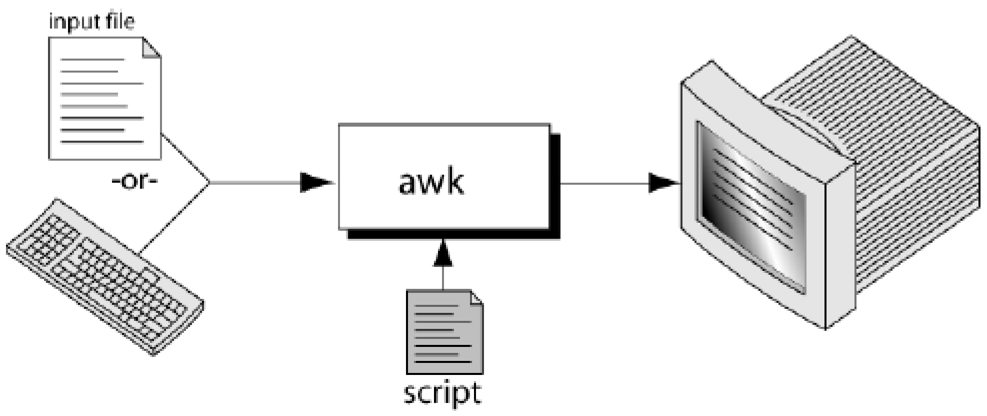
What is awk?

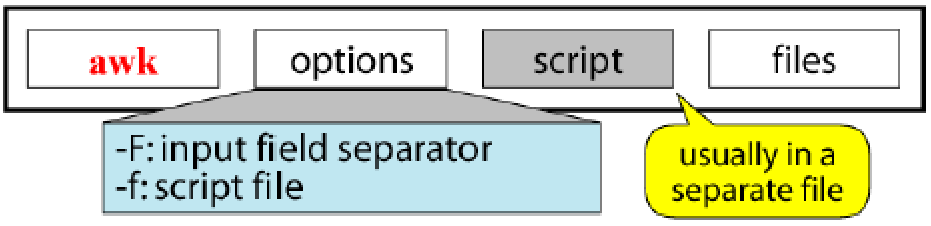
- created by: Aho, Weinberger, and Kernighan
- scripting language used for manipulating data and generating reports
- versions of awk
    - awk, nawk, mawk, pgawk, …
    - GNU awk: gawk

What can you do with awk?

- awk operation:
    - scans a file line by line
    - splits each input line into fields
    - compares input line/fields to pattern
    - performs action(s) on matched lines
- Useful for:
    - transform data files
    - produce formatted reports
- Programming constructs:
    - format output lines
    - arithmetic and string operations
    - conditionals and loops

The Command: awk





Basic awk Syntax

- **awk [options] 'script' file(s)**
- **awk [options] –f scriptfile file(s)**

<u>Options:</u>

-F	to change input field separator
-f	to name script file

Basic awk Program

- consists of patterns & actions:

  pattern {action}

  - if pattern is missing, action is applied to all lines

- if action is missing, the matched line is printed
- must have either pattern or action

Example:

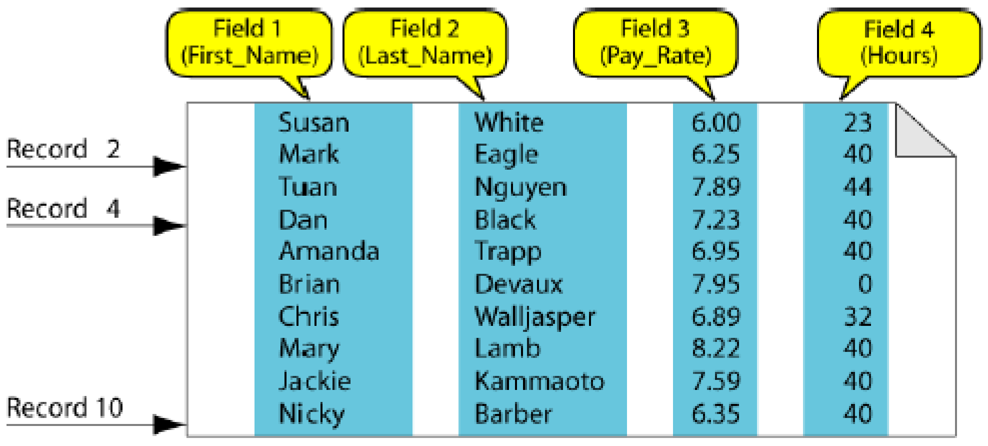
awk '/for/' testfile

- prints all lines containing string "for" in testfile

Basic Terminology: input file

- A field is a unit of data in a line
- Each field is separated from the other fields by the field separator
  - default field separator is whitespace
- A record is the collection of fields in a line
- A data file is made up of records

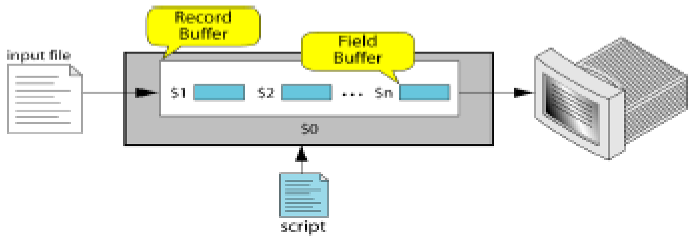
Example Input File



Susan	White	6.00	23
Mark	Eagle	6.25	40
Tuan	Nguyen	7.89	44
Dan	Black	7.23	40
Amanda	Trapp	6.95	40
Brian	Devaux	7.95	0
Chris	Walljasper	6.89	32
Mary	Lamb	8.22	40
Jackie	Kammaoto	7.59	40
Nicky	Barber	6.35	40

A file with 10 records, each with four fields

Buffers



- awk supports two types of buffers:

  <u>record</u> and <u>field</u>

- field buffer:
  - one for each fields in the current record.
  - names: $1, $2, …
- record buffer :
  - $0 holds the entire record

Some System Variables

FS	Field separator (default=whitespace)
RS	Record separator (default=\n)
NF	Number of fields in current record
NR	Number of the current record
OFS	Output field separator (default=space)
ORS	Output record separator (default=\n)
FILENAME	Current filename

Example: Records and Fields

% cat emps

```
Tom Jones      4424    5/12/66        543354

Mary Adams      5346    11/4/63      28765

Sally Chang     1654    7/22/54        650000

Billy Black     1683    9/23/44          336500

% awk '{print NR, $0}' emps

1 Tom Jones     4424    5/12/66        543354

2 Mary Adams    5346    11/4/63      28765

3 Sally Chang   1654    7/22/54       650000

4 Billy Black   1683    9/23/44          336500
```

Example: Space as Field Separator

```
% cat emps

Tom Jones      4424    5/12/66 543354

Mary Adams      5346    11/4/63 28765

Sally Chang     1654    7/22/54 650000

Billy Black     1683    9/23/44 336500

% awk '{print NR, $1, $2, $5}' emps

1 Tom Jones 543354

2 Mary Adams 28765

3 Sally Chang 650000
```

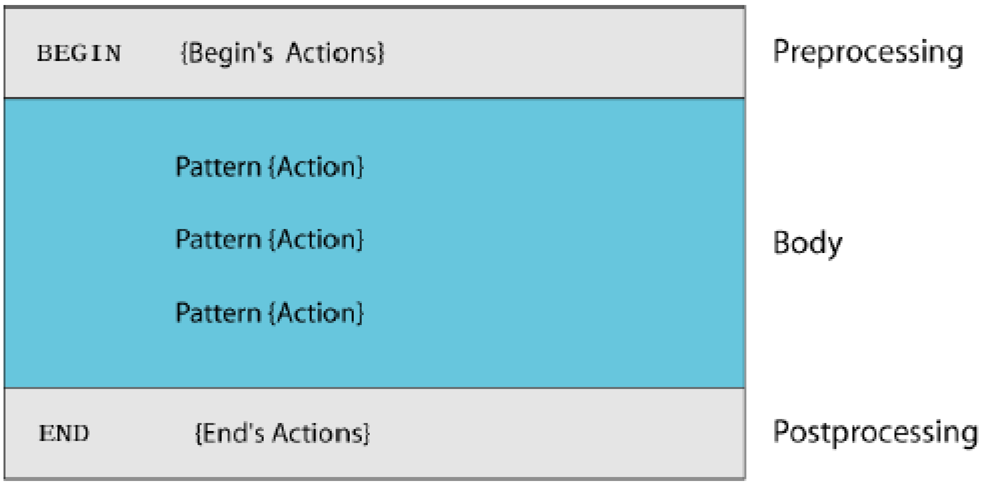
4 Billy Black 336500

Example: Colon as Field Separator

```
% cat em2

Tom Jones:4424:5/12/66:543354

Mary Adams:5346:11/4/63:28765

Sally Chang:1654:7/22/54:650000

Billy Black:1683:9/23/44:336500

% awk -F: '/Jones/{print $1, $2}' em2

Tom Jones 4424
```

awk Scripts

- awk scripts are divided into three major parts:



- comment lines start with #

awk Scripts

- BEGIN: pre-processing
  - performs processing that must be completed before the file processing starts (i.e., before awk starts reading records from the input file)
  - useful for initialization tasks such as to initialize variables and to create report headings
- BODY: Processing
  - contains main processing logic to be applied to input records
  - like a loop that processes input data one record at a time:
    - if a file contains 100 records, the body will be executed 100 times, one for each record
- END: post-processing
  - contains logic to be executed after all input data have been processed
  - logic such as printing report grand total should be performed in this part of the script

Pattern / Action Syntax

```
pattern {statement}
```

(a) One Statement Action

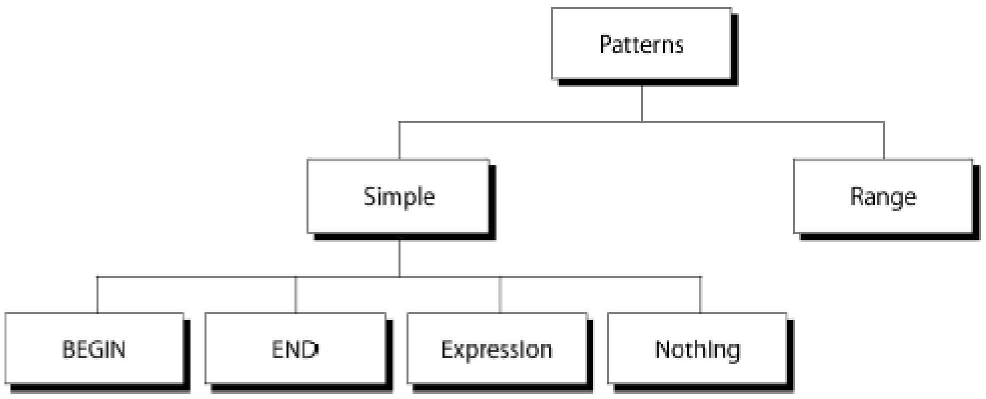
```
pattern {statement1; statement2; statement3}
```

(b) Multiple Statements Separated by Semicolons

```
pattern
{
    statement1
    statement2
    statement3
}
```

(c) Multiple Statements Separated by Newlines

Categories of Patterns



Expression Pattern types

- match
  - entire input record

  regular expression enclosed by '/'s

  - explicit pattern-matching expressions

  ~ (match), !~ (not match)

- expression operators
  - arithmetic
  - relational
  - logical

Example: match input record

% cat employees2

Tom Jones:4424:5/12/66:543354

Mary Adams:5346:11/4/63:28765

```
Sally Chang:1654:7/22/54:650000

Billy Black:1683:9/23/44:336500

% awk –F: '/00$/' employees2

Sally Chang:1654:7/22/54:650000

Billy Black:1683:9/23/44:336500
```

Example: explicit match

```
% cat datafile

northwest NW    Charles Main       3.0    .98    3   34

western   WE    Sharon Gray        5.3    .97    5   23

southwest SW    Lewis Dalsass      2.7    .8     2   18

southern  SO    Suan Chin          5.1    .95    4   15

southeast SE    Patricia Hemenway  4.0    .7     4   17

eastern   EA    TB Savage          4.4    .84    5   20

northeast NE    AM Main            5.1    .94    3   13

north     NO    Margot Weber       4.5    .89    5    9

central   CT    Ann Stephens       5.7    .94    5   13

% awk '$5 ~ /\.[7-9]+/' datafile

southwest SW    Lewis Dalsass      2.7    .8     2   18

central   CT    Ann Stephens       5.7    .94    5   13
```

Examples: matching with REs

```
% awk '$2 !~ /E/{print $1, $2}' datafile

northwest NW

southwest SW

southern SO

north NO

central CT

% awk '/^[ns]/{print $1}' datafile

northwest

southwest

southern

southeast

northeast

north
```

Arithmetic Operators

Operator	Meaning	Example
+	Add	x + y
-	Subtract	x – y
*	Multiply	x * y

/	Divide	x / y
%	Modulus	x % y
^	Exponential	x ^ y

Example:

% awk '$3 * $4 > 500 {print $0}' file

Relational Operators

Operator	Meaning	Example
<	Less than	x < y
< =	Less than or equal	x < = y
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
> =	Greater than or equal to	x > = y
~	Matched by reg exp	x ~ /y/
!~	Not matched by req exp	x !~ /y/

Logical Operators

Operator	Meaning	Example
&&	Logical AND	a && b
||	Logical OR	a || b

!	NOT	! a

Examples:

% awk '($2 > 5) && ($2 <= 15)   {print $0}' file

% awk '$3 == 100 || $4 > 50' file

Range Patterns

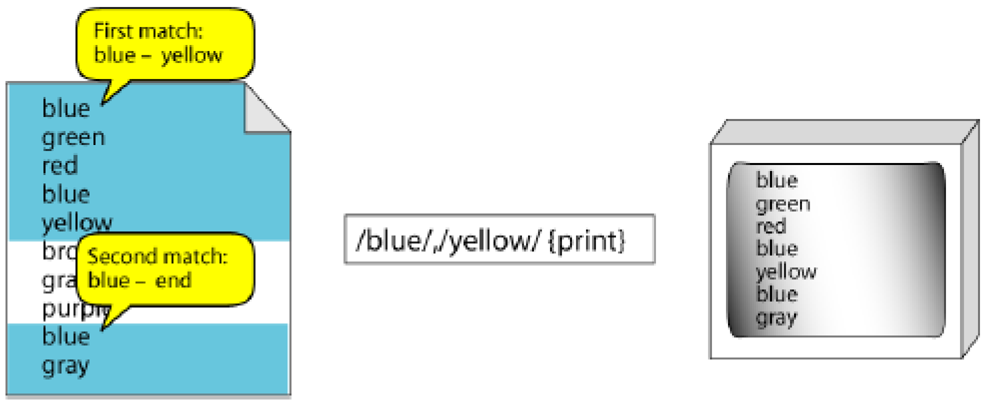
- Matches ranges of consecutive input lines

Syntax:

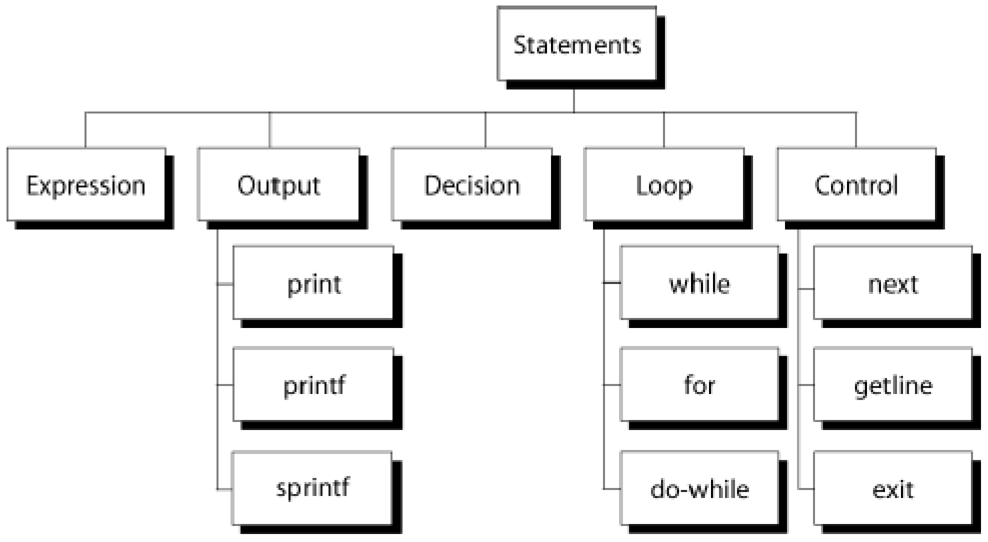
pattern1 , pattern2 {action}

- pattern can be any simple pattern
- **pattern1** turns action on
- **pattern2** turns action off

Range Pattern Example



awk Actions



awk expressions

- Expression is evaluated and returns value
  - consists of any combination of numeric and string constants, variables, operators, functions, and regular expressions
- Can involve variables
  - As part of expression evaluation
  - As target of assignment

awk variables

- A user can define any number of variables within an awk script
- The variables can be numbers, strings, or arrays
- Variable names start with a letter, followed by letters, digits, and underscore
- Variables come into existence the first time they are referenced; therefore, they do not need to be declared before use
- All variables are initially created as strings and initialized to a null string “”

- File: grades

```
john 85 92 78 94 88

andrea 89 90 75 90 86

jasper 84 88 80 92 84
```

- awk script: average

```
# average five grades

{ total = $2 + $3 + $4 + $5 + $6

  avg = total / 5

  print $1, avg }
```

- Run as:

```
awk –f average grades
```

Output Statements

print

print easy and simple output

printf

print formatted (similar to C printf)

sprintf

format string (similar to C sprintf)

Function: print

- Writes to standard output
- Output is terminated by ORS
    - default ORS is newline
- If called with no parameter, it will print $0
- Printed parameters are separated by OFS,
    - default OFS is blank
- Print control characters are allowed:
    - \n \f \a \t \\ …

print example

```
% awk '{print}' grades

john 85 92 78 94 88

andrea 89 90 75 90 86

% awk '{print $0}' grades

john 85 92 78 94 88

andrea 89 90 75 90 86

% awk '{print($0)}' grades

john 85 92 78 94 88

andrea 89 90 75 90 86
```

Redirecting print output

- Print output goes to standard output

unless redirected via:

> “file”

>> “file”

| “command”

- will open file or command only once
- subsequent redirections append to already open stream

print Example

```
% awk '{print $1 , $2 > "file"}' grades

% cat file

john 85

andrea 89

jasper 84

% awk '{print $1,$2 | "sort"}' grades

andrea 89

jasper 84

john 85

% awk '{print $1,$2 | "sort –k 2"}' grades

jasper 84

john 85

andrea 89

% date
```

```
Wed Nov 19 14:40:07 CST 2008

% date |

        awk '{print "Month: " $2 "\nYear: ", $6}'

Month: Nov

Year:  2008
```

printf: Formatting output

<u>Syntax:</u>

printf(format-string, var1, var2, …)

- works like C printf
- each format specifier in "format-string" requires argument of matching type

Format specifiers

%d, %i	decimal integer
%c	single character
%s	string of characters
%f	floating point number
%o	octal number
%x	hexadecimal number
%e	scientific floating point notation
%%	the letter "%"

Format specifier examples

Given: x = 'A', y = 15, z = 2.3, and $1 = Bob Smith

Printf Format Specifier	What it Does
%c	*printf("The character is %c \n", x)* **output: The character is A**
%d	*printf("The boy is %d years old \n", y)* **output: The boy is 15 years old**
%s	*printf("My name is %s \n", $1)* **output: My name is Bob Smith**
%f	*printf("z is %5.3f \n", z)* **output: z is 2.300**

Format specifier modifiers

- between "%" and letter

  %10s

  %7d

  %10.4f

  %-20s

- meaning:
    - width of field, field is printed right justified
    - precision: number of digits after decimal point
    - "-" will left justify

sprintf: Formatting text

<u>Syntax:</u>

sprintf(format-string, var1, var2, …)

- Works like printf, but does not produce output
- Instead it returns formatted string

<u>Example:</u>

```
{
        text = sprintf("1: %d – 2: %d", $1, $2)
        print text
}
```

awk builtin functions

tolower(string)

- returns a copy of string, with each upper-case character converted to lower-case. Nonalphabetic characters are left unchanged.

  <u>Example:</u> tolower("MiXeD cAsE 123")

  returns "mixed case 123"

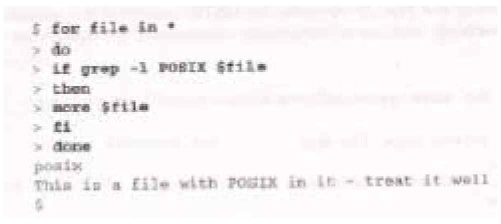
toupper(string)

The Shell as a Programming Language

You can type in a sequence of commands and allow the shell to execute them interactively, or youu can sotre these commands in a file which you can invoke as a program.

Interactive Programs

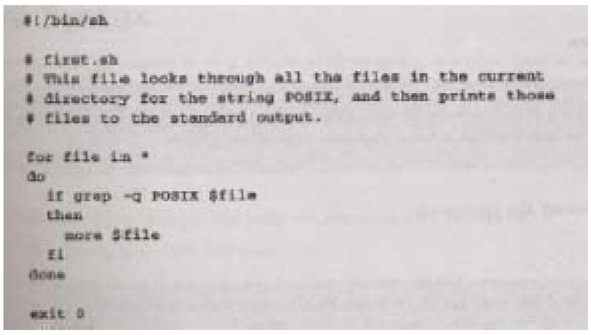
A quick way of trying out small code fragments is to just type in the shell script on the command line. Here is a shell program to compile only files that contain the string POSIX.



```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done
posix
This is a file with POSIX in it - treat it well
$
```

Creating a Script

To create a **shell script** first use a text editor to create a file containing the commands. For example, type the following commands and save them as first.sh



```
#!/bin/sh

# first.sh
# This file looks through all the files in the current
# directory for the string POSIX, and then prints those
# files to the standard output.

for file in *
do
  if grep -q POSIX $file
  then
    more $file
  fi
done

exit 0
```

Note: commands start with a #.

The line

#!/bin/sh

is special and tells the system to use the /bin/sh program to execute this program.

The command

```
exit 0
```

Causes the script program to exit and return a value of 0, which means there were not errors.

Making a Script Executable

There are two ways to execute the script. 1) invoke the shell with the name of the script file as a parameter, thus:

```
/bin/sh first.sh
```

Or 2) change the mode of the script to executable and then after execute it by just typing its name.

```
chmod +x first.sh
first.sh
```

Actually, you may need to type:

```
./first.sh
```

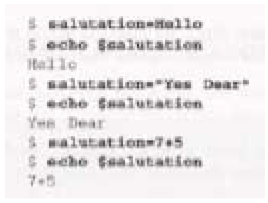
to make the file execute unles the path variable has your directory in it.

Shell Syntax

The modern UNIX shell can be used to write quite large, structured programs.

Variables

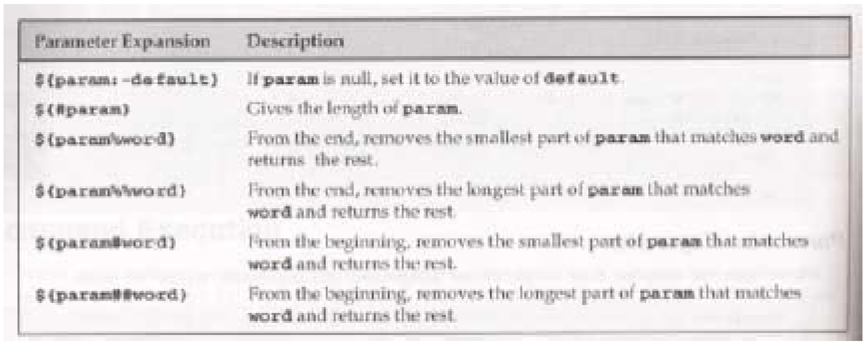
Variables are generally created when you first use them. By default, all variables are considered and stored as strings. Variable names are case sensitive.



```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

Quoting

Normally, parameters are separated by white space, such as a space. Single quot marks can be used to enclose values containing space(s). Type the following into a file called quot.sh



Parameter Expansion	Description
`${param:-default}`	If **param** is null, set it to the value of **default**.
`${#param}`	Gives the length of **param**.
`${param%word}`	From the end, removes the smallest part of **param** that matches **word** and returns the rest.
`${param%%word}`	From the end, removes the longest part of **param** that matches **word** and returns the rest.
`${param#word}`	From the beginning, removes the smallest part of **param** that matches **word** and returns the rest.
`${param##word}`	From the beginning, removes the longest part of **param** that matches **word** and returns the rest.

How It Works

The try it out exercise uses parameter expansion to demonstrate how parameter expansion works.

Here Documents

A here document is a special way of passing input to a command from a shell script. The document starts and ends with the same leader after <<. For example:

```
#!/bin/sh

cat < this is a here
document
!FUNKY!
```

How It Works

It executes the here document as if it were input commands.

Debugging Scripts

When an error occurs in a script, the shell prints out the line number with an error. You can use the set command to set various shell option. Here are some of them.

```
        exit 0
```
Causes the script program to exit and return a value of 0, which means there were not errors.

Making a Script Executable

There are two ways to execute the script. 1) invoke the shell with the name of the script file as a parameter, thus:

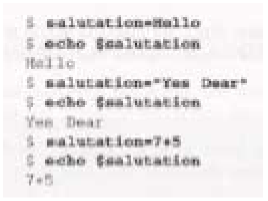
```
        /bin/sh first.sh
```
Or 2) change the mode of the script to executable and then after execute it by just typing its name.
```
        chmod +x first.sh
        first.sh
```
Actually, you may need to type:
```
        ./first.sh
```
to make the file execute unles the path variable has your directory in it.

Shell Syntax

The modern UNIX shell can be used to write quite large, structured programs.

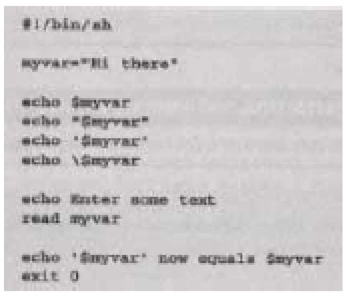
Variables

Variables are generally created when you first use them. By default, all variables are considered and stored as strings. Variable names are case sensitive.


```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

Quoting

Normally, parameters are separated by white space, such as a space. Single quot marks can be used to enclose values containing space(s). Type the following into a file called quot.sh

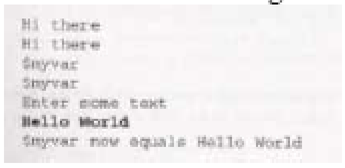


```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

make sure to make it executable by typing the command:

< chmod a+x quot.sh

The results of executing the file is:



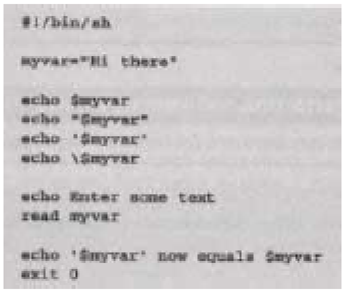
```
Hi there
Hi there
$myvar
$myvar
Enter some text
Hello World
$myvar now equals Hello World
```

How It Works

The variable myvar is created and assigned the string Hi there. The content of the variable is displyed using the echo $. Double quotes don't effect echoing the value. Single quotes and backslash do.

Environment Variables

When a shell starts, some variables are initialized from values in the environment. Here is a sample of some of them.

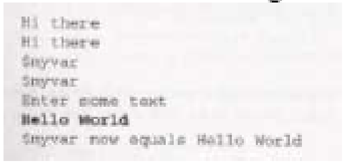


```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

make sure to make it executable by typing the command:

< chmod a+x quot.sh

The results of executing the file is:



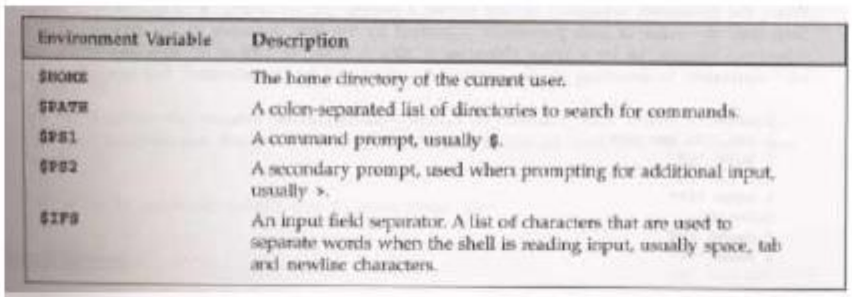
```
Hi there
Hi there
$myvar
$myvar
Enter some text
Hello World
$myvar now equals Hello World
```

How It Works

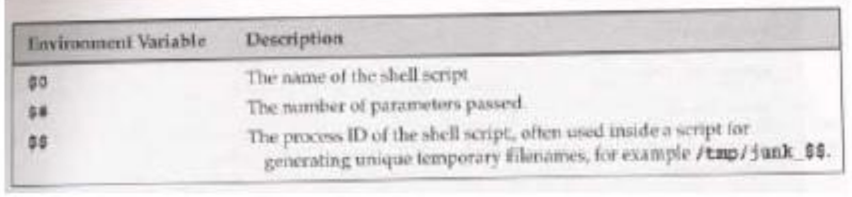
The variable myvar is created and assigned the string Hi there. The content of the variable is displyed using the echo $. Double quotes don't effect echoing the value. Single quotes and backslash do.

Environment Variables

When a shell starts, some variables are initialized from values in the environment. Here is a sample of some of them.



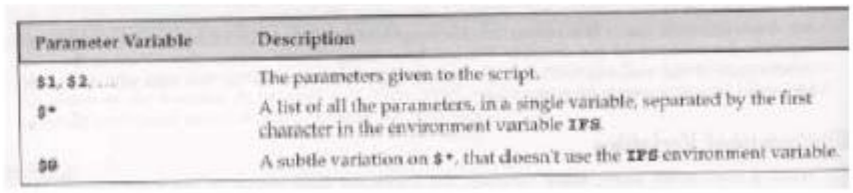
Environment Variable	Description
`$HOME`	The home directory of the current user.
`$PATH`	A colon-separated list of directories to search for commands.
`$PS1`	A command prompt, usually `$`.
`$PS2`	A secondary prompt, used when prompting for additional input, usually `>`.
`$IFS`	An input field separator. A list of characters that are used to separate words when the shell is reading input, usually space, tab and newline characters.



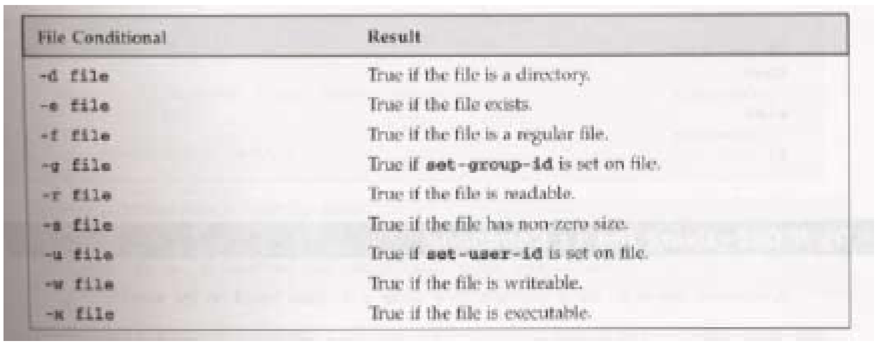
Environment Variable	Description
`$0`	The name of the shell script
`$#`	The number of parameters passed
`$$`	The process ID of the shell script, often used inside a script for generating unique temporary filenames, for example `/tmp/junk_$$`.

Parameter Variables

If your script is invoked with parameters, some additional variables are created.



Parameter Variable	Description
`$1, $2, ...`	The parameters given to the script.
`$*`	A list of all the parameters, in a single variable, separated by the first character in the environment variable `IFS`.
`$@`	A subtle variation on `$*`, that doesn't use the `IFS` environment variable.



File Conditional	Result
-d file	True if the file is a directory.
-e file	True if the file exists.
-f file	True if the file is a regular file.
-g file	True if set-group-id is set on file.
-r file	True if the file is readable.
-s file	True if the file has non-zero size.
-u file	True if set-user-id is set on file.
-w file	True if the file is writeable.
-x file	True if the file is executable.

Control Structures

The shell has a set of control structures.

if

The if statement is vary similar other programming languages except it ends with a fi.

```
if condition
then
        statements
else
        statements
fi
```

elif

the elif is better known as "else if". It replaces the else part of an if statement with another if statement. You can try it out by using the following script.

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]
then
        echo "Good morning"
elif [ $timeofday = "no" ]; then
        echo "Good afternoon"
else
```

```
        echo "Sorry, $timeofday not recognized. Enter yes or no"
        exit 1
fi

exit 0
```

How It Works

The above does a second test on the variable timeofday if it isn't equal to yes.

A Problem with Variables

If a variable is set to null, the statement

```
if [ $timeofday = "yes" ]
```

looks like

```
if [ = "yes" ]
```

which is illegal. This problem can be fixed by using double quotes around the variable name.

```
if [ "$timeofday" = "yes" ]
```

.

for

The for construct is used for looping through a range of values, which can be any set of strings. The syntax is:

```
for variable in values
do
        statements
done
```

Try out the following script:

```
#!/bin/sh

for foo in bar fud 43
do
        echo $foo
done
exit 0
```

When executed, the output should be:

```
bar
fud
43
```

How It Works

The above example creates the variable foo and assigns it a different value each time around the for loop.

How It Works

Here is another script which uses the $(command) syntax to expand a list to chap3.txt, chap4.txt, and chap5.txt and print the files.

```
#!/bin/sh

for file in $(ls chap[345].txt); do
        lpr $file
done
```

while

While loops will loop as long as some condition exist. OF course something in the body statements of the loop should eventually change the condition and cause the loop to exit. Here is the while loop syntax.

```
while condition do
        statements
done
```

Here is a whil loop that loops 20 times.

```
#!/bin/sh

foo=1

while [ "$foo" -le 20 ]
do
        echo "Here we go again"
        foo=$(($foo+1))
done

exit 0
```

How It Works

The above script uses the [ ] command to test foo for <= the value 20. The line

```
foo=$(($foo+1))
```

increments the value of foo each time the loop executes..

until

The until statement loops until a condition becomes true! Its syntax is:

```
until condition
do
        statements
done
```

Here is a script using until.

```
#!/bin/sh

until who | grep "$1" > /dev/null
do
        sleep 60
done

# now ring the bell and announce the expected user.

echo -e \\a
echo "**** $1 has just loogged in ****"

exit 0
```

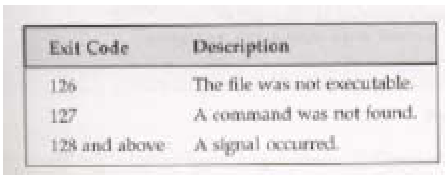
case

The case statement allows the testing of a variable for more then one value. The case statement ends with the word esac. Its syntax is:

```
case variable in
        pattern [ | pattern] ...) statements;;
        pattern [ | pattern] ...) statements;;
        ...
esac
```

Here is a sample script using a case statement:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
        "yes") echo "Good Morning";;
        "no" ) echo "Good Afternoon";;
```



Exit Code	Description
126	The file was not executable.
127	A command was not found.
128 and above	A signal occurred.

export

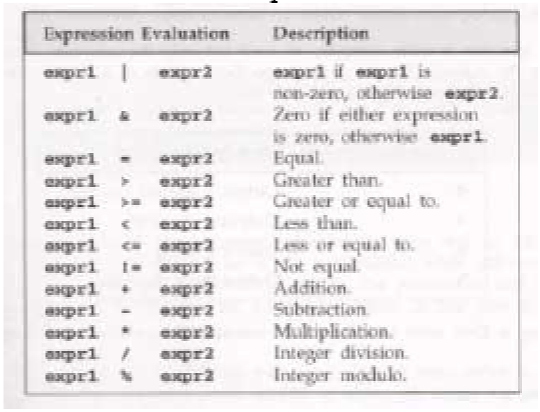
The export command makes the variable named as its parameter available in subshells.

expr

The expr command evaluates its arguments as an expression.

```
x = `expr $x + 1`
```

Here are some of its expression evaluations



Expression Evaluation	Description
`expr1 | expr2`	**expr1** if **expr1** is non-zero, otherwise **expr2**.
`expr1 & expr2`	Zero if either expression is zero, otherwise **expr1**.
`expr1 = expr2`	Equal.
`expr1 > expr2`	Greater than.
`expr1 >= expr2`	Greater or equal to.
`expr1 < expr2`	Less than.
`expr1 <= expr2`	Less or equal to.
`expr1 != expr2`	Not equal.
`expr1 + expr2`	Addition.
`expr1 - expr2`	Subtraction.
`expr1 * expr2`	Multiplication.
`expr1 / expr2`	Integer division.
`expr1 % expr2`	Integer modulo.

printf

The printf command is only available in more recent shells. It works similar to the echo command. Its general form is:

```
printf "format string" parameter1 parameter2 ...
```

Here are some characters and format specifiers.

return

The return command causes functions to return. It can have a value parameter which it returns.

set

The set command sets the parameter variables for the shell.

shift

The shift command moves all the parameters variables down by one, so $2 becomes $1, $3 becomes $2, and so on.

trap

The trap command is used for secifying the actions to take on receipt of signals. It syntax is:

```
trap command signal
```

Here are some of the signals.

Unit II – Files and Directories

Working with Files

In this chapter we learn how to create, open, read, write, and close files.

UNIX File Structure

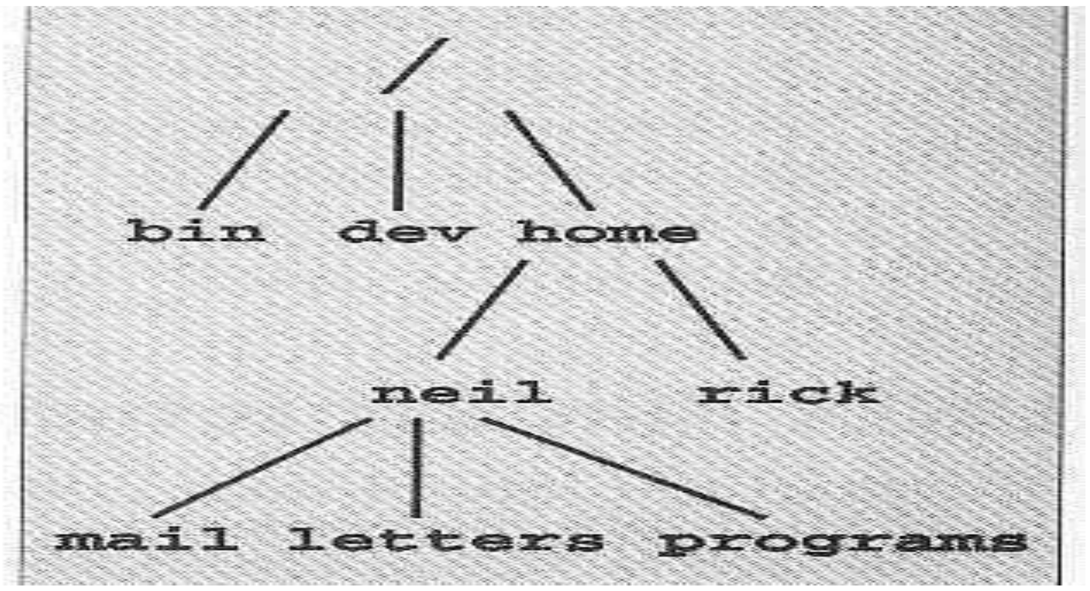
In UNIX, everything is a file.
Programs can use disk files, serial ports, printers and other devices in the exactly the same way as they would use a file.
Directories, too, are special sorts of files.

Directories

As well as its contents, a file has a name and 'administrative information', i.e. the file's creation/modification date and its permissions.

The permissions are stored in the inode, which also contains the length of the file and where on the disc it's stored.

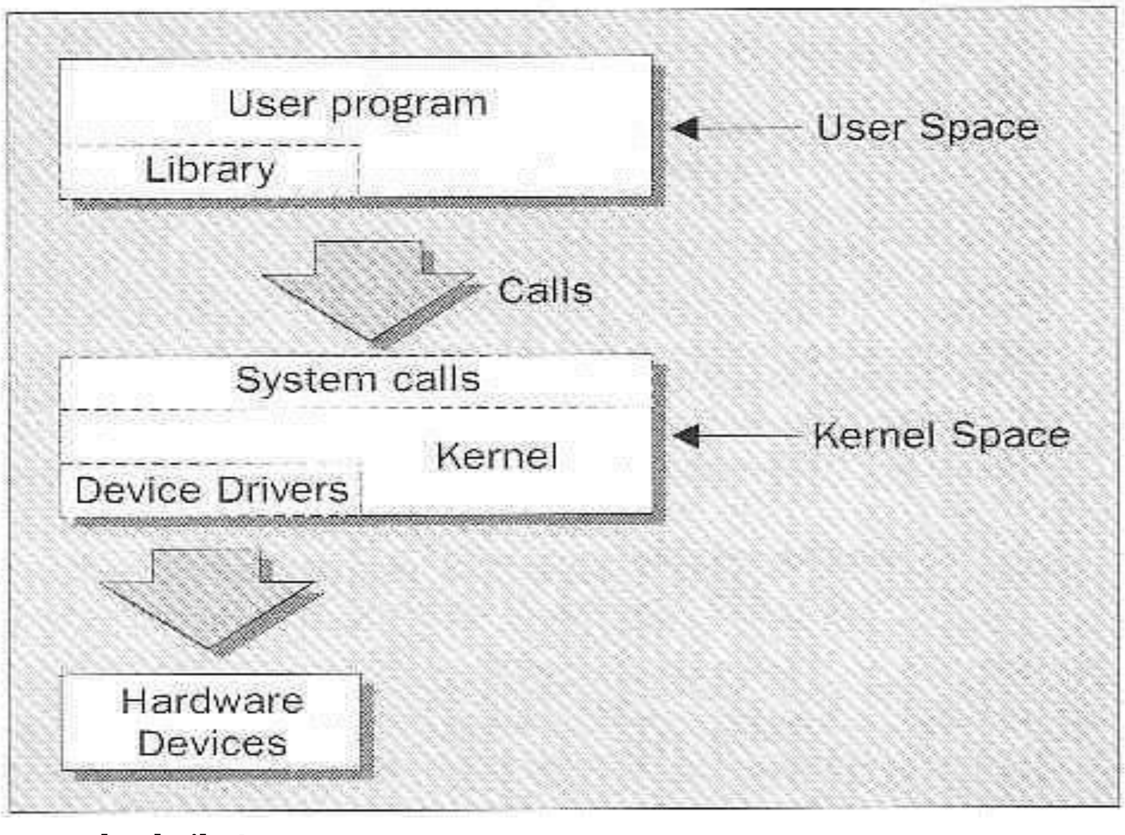
A directory is a file that holds the inodes and names of other files. Files are arranged in directories, which also contain subdirectories.
A user, neil, usually has his files stores in a 'home' directory, perhaps /home/neil.



Files and Devices

Even hardware devices are represented (mapped) by files in UNIX. For example, as
root, you mount a CD-ROM drive as a file,

```
$ mount -t iso9660 /dev/hdc /mnt/cd_rom
$ cd /mnt/cd_rom
```



Low-level File Access

Each running program, called a **process**, has associated with it a number of file descriptors.

When a program starts, it usually has three of these descriptors already opened. These are: The **write** system call arranges for the first **n bytes** bytes from **buf** to be written to the file associated with the file descriptor **files**.

With this knowledge, let's write our first program, **simple_write.c**:

```
struct stat statbuf;
mode_t modes;

stat("filename",&statbuf);
modes = statbuf.st_mode;

if(!S_ISDIR(modes) && (modes & S_IRWXU) == S_IXUSR)
    ...
```

dup and dup2

```
#include <unistd.h>

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

The **dup** system calls provide a way of duplicating a file descriptor, giving two or more, different descriptors that access the same file.

The Standard I/O Library

The standard I/O library and its header file **stdio.h**, provide a versatile interface to low-level I/O system calls.

Three file streams are automatically opened when a program is started. They are **stdin**, **stdout**, and **stderr**.

Now, let's look at:

- `fopen`, `fclose`
- `fread`, `fwrite`
- `fflush`
- `fseek`
- `fgetc`, `getc`, `getchar`
- `fputc`, `putc`, `putchar`
- `fgets`, `gets`
- `printf`, `fprintf` and `sprintf`
- `scanf`, `fscanf` and `sscanf`

fopen

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

The **fopen** library function is the analog of the low level **open** system call.

fopen opens the file named by the **filename** parameter and associates a stream with it.
The **mode**
parameter specifies how the file is to be opened. It's one of the following strings:

"r" or "rb"	Open for reading only
"w" or "wb"	Open for writing, truncate to zero length
"a" or "ab"	Open for writing, append to end of file
"r+" or "rb+" or "r+b"	Open for update (reading and writing)
"w+" or "wb+" or "w+b"	Open for update, truncate to zero length
"a+" or "ab+" or "a+b"	Open for update, append to end of file

If successful, **fopen** returns a non-null **FILE *** pointer.

fread

The **fread** library function is used to read data from a file stream. Data is read
into a data buffer given by **ptr** from the stream, **stream**.

Fwrite

The **fwrite** library call has a similar interface to **fread**. It takes data records from the specified
data buffer and writes them to the output stream.

fclose

```
#include <stdio.h>

int fclose(FILE *stream);
```

The **fclose** library function closes the specified **stream**, causing any unwritten data to be written.

Fflush

```
#include <stdio.h>

int fflush(FILE *stream);
```

The **fflush** library function causes all outpstanding data on a file stream to be written immediately.

fseek

```
#include <stdio.h>

int fseek(FILE *stream, long int offset, int whence);
```

The **fseek** function is the file stream equivalent of the **lseek** system call.
It sets the position in the stream for the next read or write on that stream.

fgetc, getc, getchar

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar();
```

The **fgetc** function returns the next byte, as a character, from a file stream. When it reaches the end of file, it returns **EOF**.
The **getc** function is equivalent to **fgetc**, except that you can implement it as a macro. The **getchar** function is equivalent to **getc(stdin)** and reads the next character from the standard input.

fputc, putc, putchar

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

The **fputc** function writes a character to an output file stream. It returns the value it has written, or **EOF** on failure.
The function **putc** is quivalent to **fputc**, but you may implement it as a macro.
The **putchar** function is equivalent to **putc(c,stdout)**, writing a single character to the standard output.

fgets, gets

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

The **fgets** function reads a string from an input file **stream**. It writes characters to the string pointed to by **s** until a newline is encountered, **n-1** characters have been transferred or the end of file is reached. **Formatted Input and Output**
There are library functions for producing output in a controlled fashion.

printf, fprintf and sprintf

```
#include <stdio.h>

int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

The **printf** family of functions format and output a variable number of arguments of different types. Ordinary characters are passed unchanged into the output. Conversion specifiers cause **printf** to fetch and format additional argumetns passed as parameters. They are start with a **%**.
For example

```
printf("Some numbers: %d, %d, and %d\n", 1, 2, 3);
```

which produces, on the standard output:

Some numbers: 1, 2, and 3

%d, %i	Print an integer in decimal.
%o, %x	Print an integer in octal, hexadecimal.
%c	Print a character.
%s	Print a string.
%f	Print a floating point (single precision) number.
%e	Print a double precision number, in fixed format.
%g	Print a double in a general format.

Here's another example:

```
char initial = 'A';
char *surname = "Matthew";
double age = 6.5;

printf("Hello Miss %c %s, aged %g\n", initial, surname, age);
```

This produces:

Hello Miss A Mathew, aged 6.5

Field specifiers are given as numbers immediatley after the % character in a conversion specifier. They are used to make things clearer.

Format	Argument	| Output |
%10s	"Hello"	| Hello|
%-10s	"Hello"	|Hello |
%10d	1234	| 1234|
%-10d	1234	|1234 |
%010d	1234	|0000001234|
%10.4f	12.34	| 12.3400|
%*s	10, "Hello"	| Hello|

The **printf** function returns an integer, the number of characters written.

scanf, fscanf and sscanf

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

The **scanf** family of functions work in a similar way to the **printf** group, except that thye read items from a stream and place vlaues into variables.

The format string for **scanf** and friends contains both ordinary characters and conversion specifiers.

Here is a simple example:

```
int num;
scanf("Hello %d", &num);
```

The call to **scanf** will succeed and place **1234** into the variable **num** given either if the following inputs

```
Hello     1234
Hello1234
```

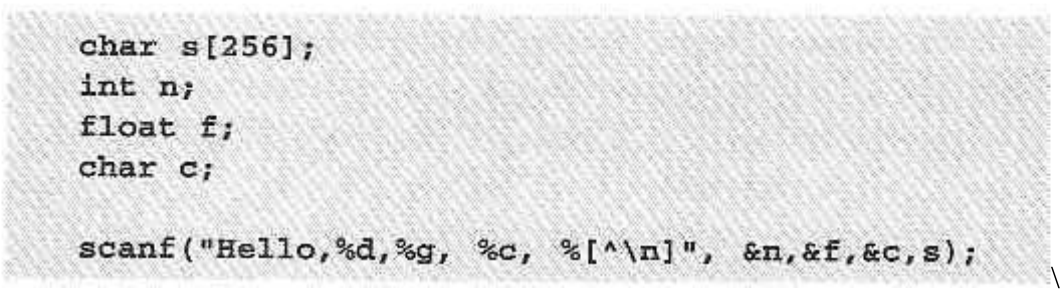
Other conversion specifiers are:

%d	Scan a decimal integer.
%o, %x	Scan an octal, hexadecimal integer.
%f, %e, %g	Scan a floating point number.
%c	Scan a character (whitespace not skipped).
%s	Scan a string.
%[]	Scan a set of characters (see below).
%%	Scan a % character.

Given the input line,

```
Hello, 1234, 5.678, X, string to the end of the line
```

this call to **scanf** will correctly scan four items:



```
char s[256];
int n;
float f;
char c;

scanf("Hello,%d,%g, %c, %[^\n]", &n,&f,&c,s);
```

\

In general, **scanf** and friends are not highly regarded, for three reasons:

Other library functions use either stream paramters or the standard streams **stdin, stdout, stderr**

In UNIX, everything is a file.

Programs can use disk files, serial ports, printers and other devices in the exactly the same way as they would use a file.

Directories, too, are special sorts of files.

Directories

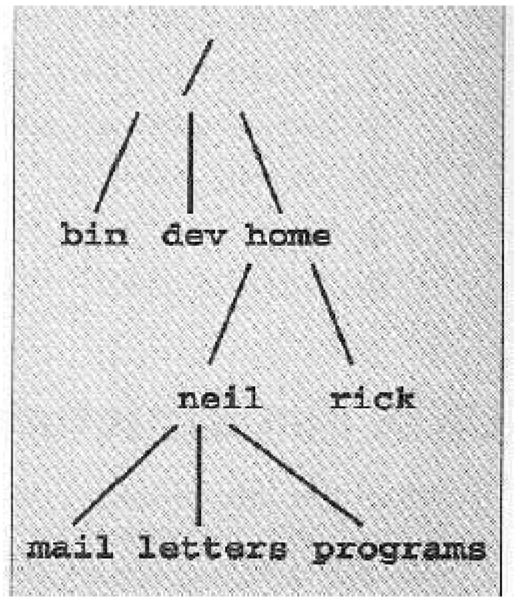
As well as its contents, a file has a name and 'administrative information', i.e. the file's creation/modification date and its permissions.

The permissions are stored in the **inode**, which also contains the length of the file and where on the disc it's stored.

A directory is a file that holds the inodes and names of other files.

Files are arranged in directories, which also contain subdirectories.

A user, **neil**, usually has his files stores in a 'home' directory, perhaps **/home/neil**.



Files and Devices

Even hardware devices are represented (mapped) by files in UNIX. For example, as **root**, you mount a CD-ROM drive as a file,

```
                    $ mount -t iso9660 /dev/hdc /mnt/cd_rom
                    $ cd /mnt/cd_rom
```

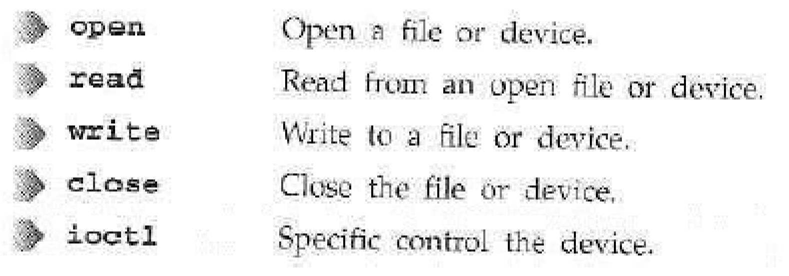
/dev/console - this device represents the system console.
/dev/tty - This special file is an alias (logical device) for controlling terminal (keyboard and screen, or window) of a process.
/dev/null - This is the null device. All output written to this device is discarded.

System Calls and Device Drivers

System calls are provided by UNIX to access and control files and devices.

A number of **device drivers** are part of the kernel.

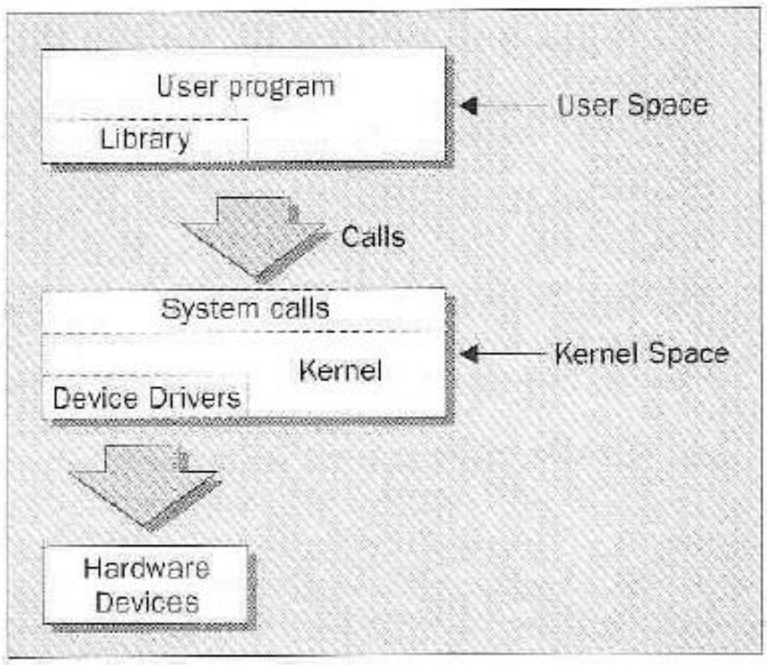
The system calls to access the device drivers include:



`open`	Open a file or device.
`read`	Read from an open file or device.
`write`	Write to a file or device.
`close`	Close the file or device.
`ioctl`	Specific control the device.

Library Functions

To provide a higher level interface to device and disk files, UNIIX provides a number of standard libraries.



Low-level File Access

Each running program, called a **process**, has associated with it a number of file descriptors.

When a program starts, it usually has three of these descriptors already opened. These are:

Digit	Value	Meaning
1	0	No user permissions are to be disallowed.
	4	User read permission is disallowed.
	2	User write permission is disallowed.
	1	User execute permission is disallowed.
2	0	No group permissions are to be disallowed.
	4	Group read permission is disallowed.
	2	Group write permission is disallowed.
	1	Group execute permission is disallowed.
3	0	No other permissions are to be disallowed.
	4	Other read permission is disallowed.
	2	Other write permission is disallowed.
	1	Other execute permission is disallowed.

For example, to block 'group' write and execute, and 'other' write, the **umask** would be:

Digit	Value
1	0
2	2
	1
3	2

Values for each digit are ANDed together; so digit 2 will have 2 & 1, giving 3. The resulting **umask** is **032**.

close

```
#include <unistd.h>

#include <sys/types.h>

off_t lseek(int fildes, off_t offset, int whence);
```

The **lseek** system call sets the read/write pointer of a file descriptor, **fildes**. You use it to set where in the file the next read or write will occur.

The **offset** parameter is used to specify the position and the **whence** parameter specifies how the offset is used.

whence can be one of the following:

SEEK_SET	offset is an absolute position
SEEK_CUR	offset is relative to the current position
SEEK_END	offset is relative to the end of the file

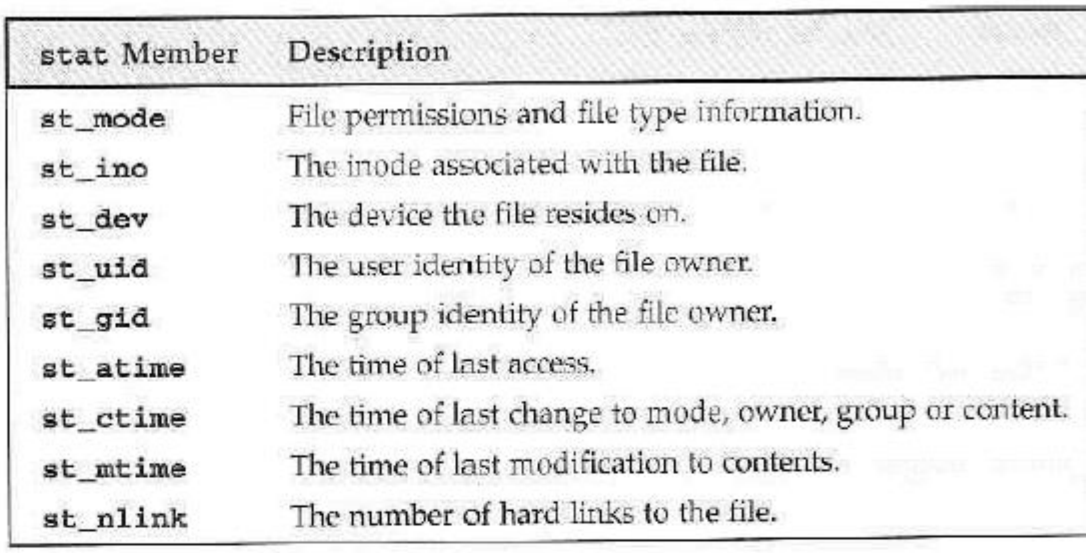
fstat, stat and lstat

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Note that the inclusion of **sys/types.h** *is deemed 'optional, but sensible'.*

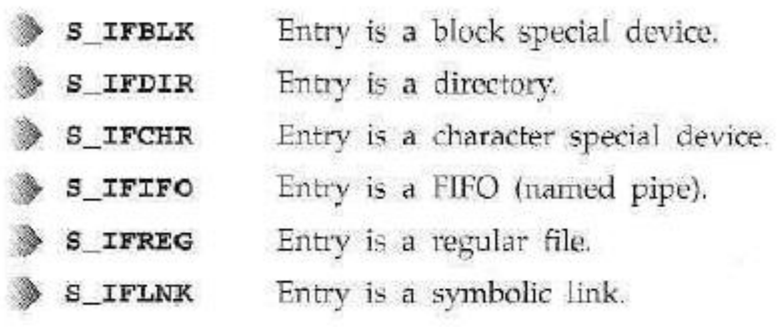
The **fstat** system call returns status information about the file associated with an open file descriptor.

The members of the structure, **stat**, may vary between UNIX systems, but will include:



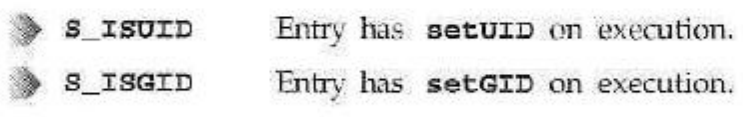
stat Member	Description
`st_mode`	File permissions and file type information.
`st_ino`	The inode associated with the file.
`st_dev`	The device the file resides on.
`st_uid`	The user identity of the file owner.
`st_gid`	The group identity of the file owner.
`st_atime`	The time of last access.
`st_ctime`	The time of last change to mode, owner, group or content.
`st_mtime`	The time of last modification to contents.
`st_nlink`	The number of hard links to the file.

The permissions flags are the same as for the **open** system call above. File-type flags include:



`S_IFBLK`	Entry is a block special device.
`S_IFDIR`	Entry is a directory.
`S_IFCHR`	Entry is a character special device.
`S_IFIFO`	Entry is a FIFO (named pipe).
`S_IFREG`	Entry is a regular file.
`S_IFLNK`	Entry is a symbolic link.

Other mode flags include:



`S_ISUID`	Entry has `setUID` on execution.
`S_ISGID`	Entry has `setGID` on execution.

Masks to interpret the **st_mode** flags include:

S_IFMT	File type.
S_IRWXU	User read/write/execute permissions.
S_IRWXG	Group read/write/execute permissions.
S_IRWXO	Others read/write/execute permissions.

There are some macros defined to help with determining file types. These include:

S_ISBLK	Test for block special file.
S_ISCHR	Test for character special fi
S_ISDIR	Test for directory.
S_ISFIFO	Test for FIFO.
S_ISREG	Test for regular file.
S_ISLNK	Test for symbolic link.

The Standard I/O Library

The standard I/O library and its header file **stdio.h**, provide a versatile interface to low-level I/O system calls.

Three file streams are automatically opened when a program is started. They are **stdin**, **stdout**, and **stderr**.

Now, let's look at:

- fopen, fclose
- fread, fwrite
- fflush
- fseek
- fgetc, getc, getchar
- fputc, putc, putchar

- fgets, gets
- printf, fprintf and sprintf
- scanf, fscanf and sscanf

`fgetpos`	Get the current position in a file stream.
`fsetpos`	Set the current position in a file stream.
`ftell`	Return the current file offset in a stream.
`rewind`	Reset the file position in a stream.
`freopen`	Reuse a file stream.
`setvbuf`	Set the buffering scheme for a stream.
`remove`	Equivalent to **unlink**, unless the **path** parameter is a directory ir case it's equivalent to **rmdir**.

You can use the file stream functions to re-implement the file copy program, by using library functions.

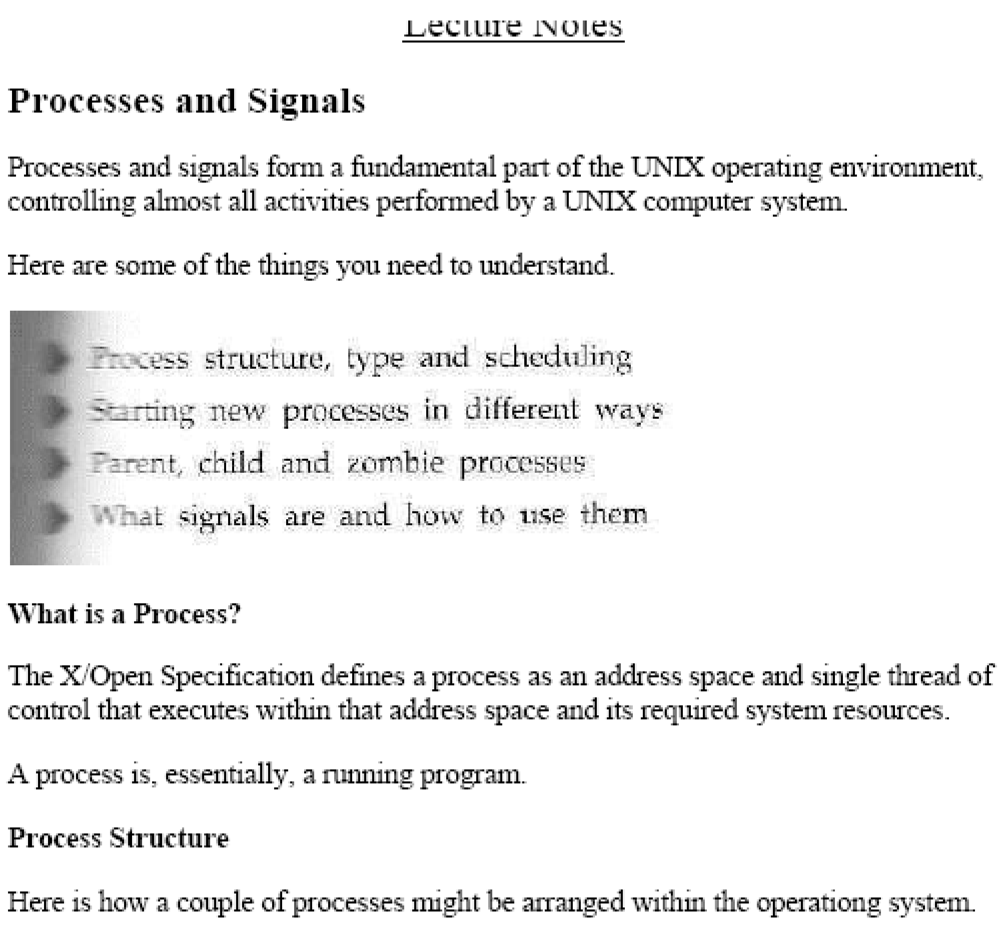
Try It Out - Another File Copy Program

This program does the character-by-character copy is accomplished using calls to the functions referenced in **stdio.h**.

```
#include <stdio.h>

int main()
{
    int c;
    FILE *in, *out;

    in = fopen("file.in","r");
    out = fopen("file.out","w");

    while((c = fgetc(in)) != EOF)
        fputc(c,out);

    exit(0);
}
```

Running this program as before, we get:

UNIT-III PROCESSES AND SIGNALS



Lecture Notes

Processes and Signals

Processes and signals form a fundamental part of the UNIX operating environment, controlling almost all activities performed by a UNIX computer system.

Here are some of the things you need to understand.

- Process structure, type and scheduling
- Starting new processes in different ways
- Parent, child and zombie processes
- What signals are and how to use them

What is a Process?

The X/Open Specification defines a process as an address space and single thread of control that executes within that address space and its required system resources.

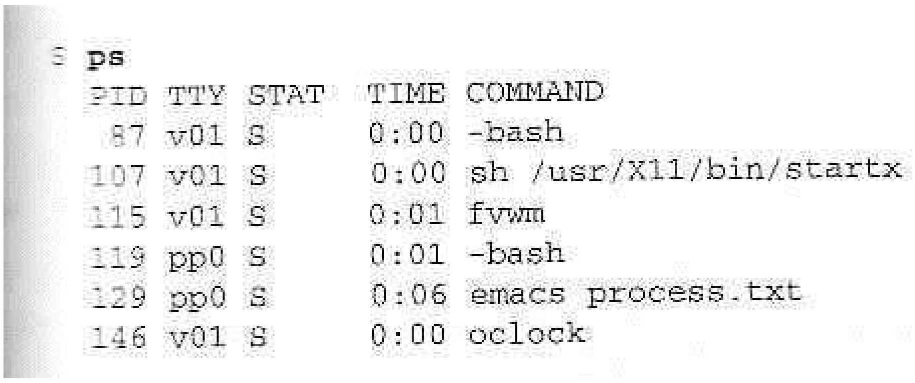
A process is, essentially, a running program.

Process Structure

Here is how a couple of processes might be arranged within the operationg system.

Viewing Processes

We can see what processes are running by using the **ps** command. Here is some sample output:

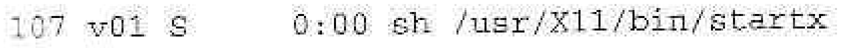


```
$ ps
  PID TTY STAT  TIME COMMAND
   87 v01 S     0:00 -bash
  107 v01 S     0:00 sh /usr/X11/bin/startx
  115 v01 S     0:01 fvwm
  119 pp0 S     0:01 -bash
  129 pp0 S     0:06 emacs process.txt
  146 v01 S     0:00 oclock
```

The **PID** column gives the PIDs, the **TTY** column shows which terminal started the process, the **STAT** column shows the current status, **TIME** gives the CPU time used so far and the **COMMAND** column shows the command used to start the process.
Let's take a closer look at some of these:

```
87 v01 S      0:00 -bash
```

The initial login was performed on virtual console number one (**v01**). The shell is running **bash**. Its status is **s**, which means sleeping. Thiis is because it's waiting for the X Windows sytem to finish.



```
107 v01 S      0:00 sh /usr/X11/bin/startx
```

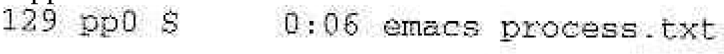
X Windows was started by the command **startx**. It won't finished until we exit from X. It too is sleeping.

```
115 v01 S      0:01 fvwm
```

The **fvwm** is a window manager for X, allowing other programs to be started and windows to be arranged on t screen

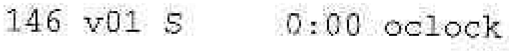
```
119 pp0 S      0:01 -bash
```

This process represents a window in the X Windows system. The shell, bash, is running in the new window. T window is running on a new pseudo terminal (/dev/ptyp0) abbreviated pp0.



```
129 pp0 S      0:06 emacs process.txt
```

This is the EMACS editor session started from the shell mentioned above. It uses the pseudo terminal



```
146 v01 S      0:00 oclock
```

This is a clock program started by the window manager. It's in the middle of a one- minute wait between updates of the clock hands.

System Processes

Let's look at some other processes running on this Linux system.
The output has been abbreviated for clarity:

```
$ ps -ax
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
   40  ?  S     0:01 /usr/sbin/syslogd
   46  ?  S     0:00 /usr/sbin/lpd
   51  ?  S     0:00 sendmail: accepting connections
   88 v02 S     0:00 /sbin/agetty 38400 tty2
  109  ?  R     0:41 X :0
  192 pp0 R     0:00 ps -ax
```

Here we can see one very important process indeed:

```
    1  ?  S     0:00 init
```

In general, each process is started by another, known as its **parent process**. A process so started is known as a
child process.

When UNIX starts, it runs a single program, the prime ancestror and process number one: **init**. One such example is the login procedure **init** starts the **getty** program once for each terminal that we can use to long in.
These are shown in the **ps** output like this:

```
   88 v02 S     0:00 /sbin/agetty 38400 tty2
```

Process Scheduling

One further **ps** output example is the entry for the **ps** command itself:

```
  192 pp0 R     0:00 ps -ax
```

This indicates that process 192 is in a run state (**R**) and is executing the command **ps- ax**.
We can set the process priority using **nice** and adjust it using **renice**, which reduce the priority of a process by 10. High priority jobs have negative values.
Using the **ps -l** (forlong output), we can view the priority of processes. The value we are interested in is shown in the **NI** (nice) column:

```
$ ps -l
  F   UID   PID  PPID PRI NI SIZE  RSS WCHAN      STAT TTY   TIME COMM
  0   501   146     1   1  0   85  756 130b85     S    v01   0:00 ocloc
```

Here we can see that the **oclock** program is running with a default nice value. If it had been stated with the command,

```
$ nice oclock &
```

it would have been allocated a nice value of +10.

We can change the priority

How It Works

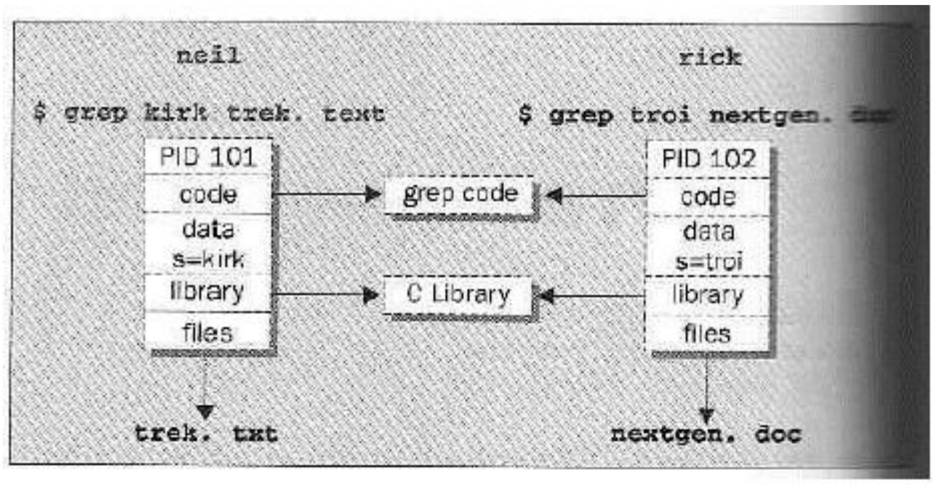
In the first example, the program calls **system** with the string **"ps -ax"**, which executes the **ps** program. Our program returns from the call to **system** when the **ps** command is finished.
In the second example, the call to **system** returns as soon as the shell command finishes. The shell returns as soon as the **ps** program is started, just as would happen if we had typed,

```
$ ps -ax &
```

at a shell prompt.

Replacing a Process Image

There is a whole family of related functions grouped under the **exec** heading. They differ in the way that they start processes and present program arguments.



Each process is allocated a unique number, a **process identifier**, or PID.

The program code that will be executed by the **grep** command is stored in a disk file.

The system libraries can also be shared.

A process has its own stack space.

The Process Table

The UNIX **process table** may be though of as a data structure describing all of the processes that are currently loaded.

Viewing Processes

We can see what processes are running by using the **ps** command.

Here is some sample output:

```
$ ps
  PID TTY STAT  TIME COMMAND
   87 v01 S     0:00 -bash
  107 v01 S     0:00 sh /usr/X11/bin/startx
  115 v01 S     0:01 fvwm
  119 pp0 S     0:01 -bash
  129 pp0 S     0:06 emacs process.txt
  146 v01 S     0:00 oclock
```

The **PID** column gives the PIDs, the **TTY** column shows which terminal started the process, the **STAT** column shows the current status, **TIME** gives the CPU time used so far and the **COMMAND** column shows the command used to start the process.

Let's take a closer look at some of these:

```
87 v01 S     0:00 -bash
```

The initial login was performed on virtual console number one (**v01**). The shell is running **bash**. Its status is s, which means sleeping. Thiis is because it's waiting for the X Windows sytem to finish.

```
107 v01 S     0:00 sh /usr/X11/bin/startx
```

X Windows was started by the command **startx**. It won't finished until we exit from X. It too is sleeping.

```
115 v01 S     0:01 fvwm
```

This is the EMACS editor session started from the shell mentioned above. It uses the pseudo terminal.

```
146 v01 S     0:00 oclock
```

This is a clock program started by the window manager. It's in the middle of a one-minute wait between updates of the clock hands.

System Processes

Let's look at some other processes running on this Linux system. The output has been abbreviated for clarity:

```
$ ps -ax
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
   40  ?  S     0:01 /usr/sbin/syslogd
   46  ?  S     0:00 /usr/sbin/lpd
   51  ?  S     0:00 sendmail: accepting connections
   88 v02 S     0:00 /sbin/agetty 38400 tty2
  109  ?  R     0:41 X :0
  192 pp0 R     0:00 ps -ax
```

Process Scheduling

One further **ps** output example is the entry for the **ps** command itself:

```
192 pp0 R      0:00 ps -ax
```

This indicates that process 192 is in a run state (**R**) and is executing the command **ps-ax**.

We can set the process priority using **nice** and adjust it using **renice**, which reduce the priority of a process by 10. High priority jobs have negative values.

Using the **ps -l** (forlong output), we can view the priority of processes. The value we are interested in is shown in the **NI** (nice) column:

```
$ ps -l
  F    UID   PID   PPID PRI NI SIZE  RSS WCHAN         STAT TTY   TIME C
  0    501   146      1   1  0   85  756 130b85        S    v01   0:00 o
```

Here we can see that the **oclock** program is running with a default nice value. If it had been stated with the command,

```
$ nice oclock &
```

it would have been allocated a nice value of +10.

We can change the priority of a ruinning process by using the **renice** command,

```
$ renice 10 146
146: old priority 0, new priority 10
```

Starting New Processes

We can cause a program to run from inside another program and thereby create a new process by using the **system**. library function.

```
#include <stdlib.h>

int system (const char *string);
```

The **system** function runs the command passed to it as **string** and waits for it to complete.

The command is executed as if the command,

```
$ sh -c string
```

has been given to a shell.

Try It Out - system

1. We can use **system** to write a program to run **ps** for us.

When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls **wait**.

This terminated child process is known as a **zombie process**.

Try It Out - Zombies

fork2.c is jsut the same as **fork.c**, except that the number of messages printed by tl child and paent porcesses is reversed.

Here are the relevant lines of code:

```
switch(pid)
{
case -1:
     exit(1);
case 0:
     message = "This is the child";
     n = 3;
     break;
default:
     message = "This is the parent";
     n = 5;
     break;
}
```

Input and Output Redirection

We can use our knowledge of processes to alter the behavior of programs by exploiting the fact that open file descriptors are preserved across calls to **fork** and **exec**.

Try It Out - Redirection

1. Here's a very simple filter program, **upper.c**, to convert all characters to uppercase:

```
#include <stdio.h>
#include <ctype.h>

int main()
{
     int ch;
     while((ch = getchar()) != EOF) {
          putchar(toupper(ch));
     }
     exit(0);
}
```

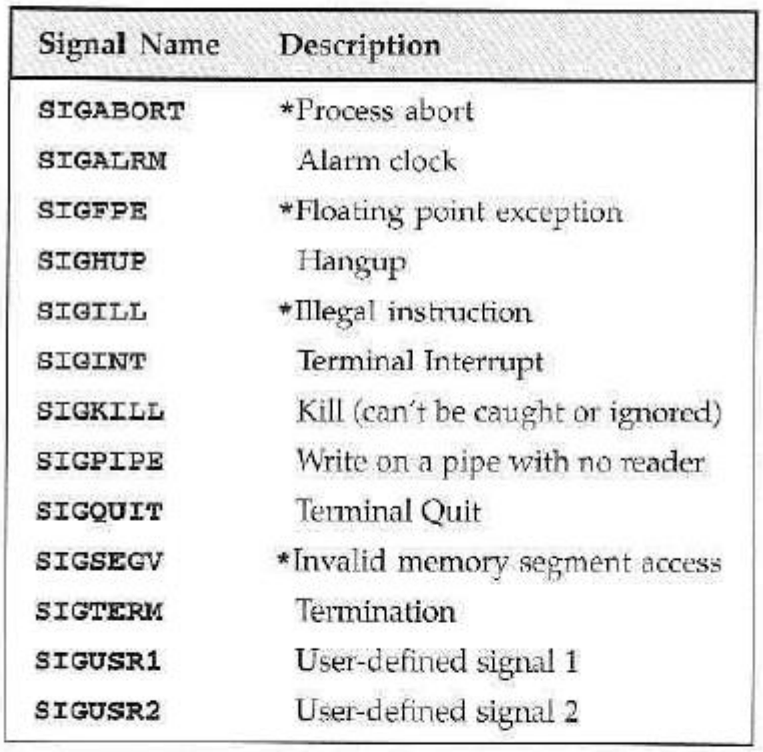
When we run this program, it reads our input and converts it:

There is a class of process known as a **thread** which are distinct from processes in that they are separate execution streams within a single process.

Signals

A **signal** is an event generated by the UNIX system in response to some condition, upon receipt of which a process may in turn take some action.

Signal names are defined in the header file **signal.h**. They all begin with **SIG** and include:



Signal Name	Description
SIGABORT	*Process abort
SIGALRM	Alarm clock
SIGFPE	*Floating point exception
SIGHUP	Hangup
SIGILL	*Illegal instruction
SIGINT	Terminal Interrupt
SIGKILL	Kill (can't be caught or ignored)
SIGPIPE	Write on a pipe with no reader
SIGQUIT	Terminal Quit
SIGSEGV	*Invalid memory segment access
SIGTERM	Termination
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

Additional signals include:

```
$ ./ctrlc
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
$
```

How It Works

The program arranges for the function **ouch** to be called when we type Ctrl-C, which gives the **SIGINT** signal.

Sending Signals

A process may send a signal to itself by calling **raise**.

```
#include <signal.h>

int raise(int sig);
```

A process may send a signal to another process, including itself, by calling **kill**.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

```
        printf("waiting for alarm to go off\n");
        (void) signal(SIGALRM, ding);

        pause();

        printf("done\n");
        exit(0);
    }
```

When we run this program, it pauses for five seconds while it waits for the simulated alarm clock.

```
$ ./alarm
alarm application starting
waiting for alarm to go off
<5 second pause>
alarm has gone off
done
$
```

This program introduces a new function, **pause**, which simply causes the program to suspend execution until a signal occurs.

It's declared as,

```
#include <unistd.h>

int pause(void);
```

How It Works

The alarm clock simulation program starts a new process via **fork**. This child process sleeps for five seconds and then sends a **SIGALRM** to its parent.

A Robust Signals Interface

X/Open specification recommends a newer programming interface for signals that is more robust: **sigaction**.

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigactic
```

The **sigaction** structure, used to define the actions to be taken on receipt of the signal specified by **sig**, is defined in **signal.h** and has at least the following members:

`void (*) (int) sa_handler`	function, `SIG_DFL` or `SIG_IGN`
`sigset_t sa_mask`	signals to block in `sa_handler`
`int sa_flags`	signal action modifiers

Try It Out - sigaction

Make the changes shown below so that **SIGINT** is intercepted by **sigaction**. Call the new program **ctrlc2.c**.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
    struct sigaction act;

    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Running the program, we get a message when we type Ctrl-C because **SIGINT** is handled repeated;y by **sigaction**.

Type Ctrl-\ to terminate the program.

```
$ ./ctrlc2
Hello World!
Hello World!
```

```
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^\
Quit
$
```

How It Works

The program calls **sigaction** instead of **signal** to set the signal handler for Ctrl-C (**SIGINT**) to the function **ouch**.

Signal Sets

The header file **signal.h** defines the type **sigset_t and functions used to manipulate sets of signals.**

```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signo);
```

The function **sigismember** determines whether the given signal is amember of a signal set.

```
#include <signal.h>

int sigismember(sigset_t *set, int signo);
```

The process signal mask is set or examined by calling the function **sigprocmask**.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

sigprocmask can change the process signal mask in a number of ways according to the **how** argument.

The **how** argument can be one of:

SIG_BLOCK	The signals in `set` are added to the signal mask.
SIG_SETMASK	The signal mask is set from `set`.
SIG_UNBLOCK	The signals in `set` are removed from the signal ma

If a signal is blocked by a process, it won't be delivered, but will remain pending.

A program can determine which of its blocked signals ar pending by calling the function **sigpending**.

```
#include <sigpending>

int sigpending(sigset_t *set);
```

A process can suspend execution until the delivery of one of a set of signals by calling **sigsuspend**.

This is a more general form of the **pause** function we met earlier.

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

sigaction Flags

The **sa_flags** field of the **sigaction** structure used in **sigaction** may contain the following values to modify signal behavior

SA_NOCLDSTOP	Don't generate **SIGCHLD** when child processes stop.
SA_RESETHAND	Reset signal action to **SIG_DFL** on receipt.
SA_RESTART	Restart interruptible functions rather than error with
SA_NODEFER	Don't add the signal to the signal mask when caug

Functions that are safe to call inside a signal handler, those guaranteed by the X/Open specification either to be re-entrant or not to raise signals themselves include:

Signal Name	Description
SIGALRM	Generated by the timer set by the **alarm** function.
SIGHUP	Sent to the controlling process by a disconnecting terminal, or by controlling process on termination to each foreground process.
SIGINT	Typically raised from the terminal by typing *Ctrl-C* or the configu interrupt character.
SIGKILL	Typically used from the shell to forcibly terminate an errant proce signal can't be caught or ignored.
SIGPIPE	Generated if a pipe with no associated reader is written to.
SIGTERM	Sent as a request for a process to finish. Used by UNIX when shu to request that system services stop. This is the default signal sent command.
SIGUSR SIGUSR2	May be used by processes to communicate with each other, possib cause them to report status information.

The default action signals is abnormal termination of the process.

Signal Name	Description
SIGFPE	Generated by a floating point arithmetic exception.
SIGILL	An illegal instruction has been executed by the processor. Usually c corrupt program or invalid shared memory module.
SIGQUIT	Typically raised from the terminal by typing *Ctrl-* or the configure
SIGSEGV	A segmentation violation, usually caused by reading or writing at a location in memory either by exceeding array bounds or de-referen pointer. Overwriting a local array variable and corrupting the stack **SIGSEGV** to be raised when a function returns to an illegal address

By default, these signals also cause abnormal termination. Additionally, implementation-dependent actions, such as creation of a core file, may occur.

Signal Name	Description
SIGSTOP	Stop executing (can't be caught or ignored).
SIGTSTP	Terminal stop signal, often raised by typing *Ctrl-Z*.
SIGTTIN **SIGTTOU**	Used by the shell to indicate that background jobs have stopped be to read from the terminal or produce output.

A process is stopped by default on receipt of one of the above signals.

Signal Name	Description
SIGCONT	Continue executing, if stopped.

SIGCONT restarts a stopped process and is ignored if received by a process which is not stopped.

Signal Name	Description
SIGCHLD	Raised when a child process stops or exits.

The **SIGCHLD** signal is ignored by default.

UNIT-IV
Interprocess communication, Message Queues and Semaphores

Interprocess Communication

IPC between processes on

a single computer system

IPC between processes on

different systems

Pipes- creation

IPC between related processes using unnamed pipes

FIFOs- creation, IPC between unrelated processes using FIFOs(named pipes)

Differences Between

Unnamed

And Named

Pipes popen

& pclose

library

functions.

Message Queues-

Kernel support for

messages

Semaphores-Kernel support for semaphores

APIs for semaphores

File locking with Semaphores

Introduction to IPC

Interprocess Communication- *“Interprocess communication(IPC) is the transfer of data among different processes”.*

Interprocess communication (IPC) includes thread synchorization and data exchange between threads beyond the process boundaries. If threads belong to the same process, they execute in the same address space, i.e. they can access global (static) data or heap directly, without the help of

the operating system. However, if threads belong to different processes, they cannot access each others address spaces without the help of the operating system.

There are two fundamentally different approaches in IPC:

- processes are residing on the same computer
- processes are residing on different computers

The first case is easier to implement because processes can share memory either in the user space or in the system space. This is equally true for uniprocessors and multiprocessors.

In the second case the computers do not share physical memory, they are connected via I/O device(for example serial communication or Ethernet). Therefore the processes residing in different computers can not use memory as a means for communication.

IPC between processes on a Single System

Most of this chapter is focused on IPC on a single computer system, including four general approaches:

- Shared memory
- Messages
- Pipes
- Sockets

The synchronization objects considered in the previous chapter normally work across the process boundaries (on a single computer system). There is one addition necessary however: the synchronization objects must be named. The handles are generally private to the process, while the object names, like file names, are global and known to all processes.

IPC between processes on different systems

IPC between processes on different systems

IPC is Inter Process Communication, more of a technique to share data across different processes within one machine, in such a way that data passing binds the coupling of different processes.

- The first, is using memory mapping techniques, where a memory map is created, andothers open the memory map for reading/writing...
- The second is, using sockets, to communicate with one another...this has a high overhead, as each process would have to open up the socket, communicate across... althougheffective
- The third, is to use a pipe or a named pipe, a very good example

PIPES:

A pipe is a *serial* communication device (i.e., the data is read in the order in which it was written),which allows a *unidirectional communication*. The data written to end isreadbackfromtheotherend.

The pipe is mainly used to communicate between two threads in a single process or between parent and child process. Pipes can only connect the related process. In shell, thesymbolcanbeusedtocreateapipe.

In pipes the *capacity of data is limited*. (i.e.) If the writing process is faster than the reading process which consumes the data, the pipe cannot store the data. In this situation the writer process will block until more capacity becomes available. Also if the reading process tries to read data when there is no data to read, it will be blocked until the data becomes available. By this, pipes *automatically synchronize the two process*.

Creatingpipes:
The *pipe()* function provides a means of passing data between two programs and also allows to read and write the data.

```
#include<unistd.h>
int pipe(int file_descriptor[2]);
```

pipe()function is passed with an array of file descriptors. It will fill the array with new file descriptors and returns zero. On error, returns -1 and sets the errno to indicate the reason of

failure.

The file descriptors are connected in a way that is data written to file_ descriptor[1] can be read back from the file_descriptor[0].

(Note: As this uses file descriptors and not the file streams, we must use read and write system calls to access the data.)

Pipes are originally used in UNIX and are made even more powerful in Windows 95/NT/2000.

Pipes are implemented in file system. Pipes are basically files with only two file offsets: one for reading another for writing. Writing to a pipe and reading from a pipe is strictly in FIFO manner. (Therefore pipes are also called FIFOs).

For efficiency, pipes are in-core files, i.e. they reside in memory instead on disk, as any other global data structure. Therefore pipes must be restricted in size, i.e. number of pipe blocks must be limited. (In UNIX the limitation is that pipes use only direct blocks.)Since the pipes have a limited size and the FIFO access discipline, the reading and writing processes are synchronized in a similar manner as in case of message buffers. The access functions for pipes are the same as for files: WriteFile() and ReadFile().

Pipes used as standard input and output:

We can invoke the standard programs, ones that don't expect a file

The purpose of dup call is to open a new file descriptor, which will refer to the same file as an existing file descriptor. In case of dup, the value of the new file descriptor is the lowest number available. In dup2 it is same as, or the first available descriptor greater than the parameter file_descriptor_2.

We can pass data between process by first closing the file descriptor 0 and call is made to dup. By this the new file descriptor will have the number 0.As the new descriptor is the duplicate of an existing one, standard input is changed to have the access. So we have created two file descriptors for same file or pipe, one of them will be the standard input.

(Note: The same operation can be performed by using the fcntl() function. But compared to this dup and dup2 are more efficient)

Named pipes (FIFOs)

Similar to pipes, but allows for communication
between unrelated processes. This is done by naming
the communication channel and making it permanent.

Like pipe, FIFO is the unidirectional data stream.

FIFO creation:

int mkfifo (const char *pathname, mode_t mode);
 - makes a FIFO special file with name pathname.

(mode specifies the FIFO's permissions,
as common in UNIX-like file systems).

- A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system by calling mkfifo()

Once a FIFO special file has been created, any process can open it for reading or writing, in the same way as an ordinary file.

A First-in, first-out(FIFO) file is a pipe that has a name in the filesystem. It is also called as named pipes.

Creation of FIFO:
We can create a FIFO from the command line and within a program.

To create from **command line** we can use either *mknod* or *mkfifo* commands.
$ mknod filename p
$ mkfifo filename
(Note**: The mknod command is available only n older versions, you can make use of mkfifo in new versions.)**

To create FIFO **within the program** we can use two system calls. They are, #include<sys/types.h>
#include<sys/stat.h>

int mkfifo(const char
*filename,mode_t mode);
int
mknod(const
char
*filename,
mode_t
mode|S_IFIF
O,(dev_t) 0);
Accessing FIFO:
Let us first discuss how to access FIFO in command line using file commmands. The useful feature of named pipes is, as they appear in the file system, we can use them in commands.

We can read from the FIFO(empty)
$ cat < /tmp/my_fifo
Now, let us write to the FIFO.
$ echo "Simple!!!" > /tmp/my_fifo
(Note: These two commands should be executed in different terminals because first command will be waiting for some data to appear in the FIFO.)
Pipe processing:(popen &pclose library functions)

The process of passing data between two programs can be done
with the help of popen() and pclose() functions.

```
#include<stdio.h>
FILE
*popen(co
nst char
*command
, const char
*open-
mode);
int pclose(FILE *stream_to_close);
```

popen():
The popen function allows a program to invoke another program as a new process and either write the data to it or to read from it. The parameter command is the name of the program to run. The open_mode parameter specifies in which mode it is to be invoked, it can be only either "r" or "w". On failure popen() returns a NULL pointer. If you want to perform bi-directional communication you have to use two pipes.

pclose():
By using pclose(), we can close the filestream associated with popen() after the process started by it has been finished. The pclose() will return the exit code of the process, which is to be closed. If the process was already executed a *wait* statement before calling pclose, the exit status will be lost because the process has been finished. After closing the filestream, pclose() will wait for the child process to terminate.

Messagequeue:
This is an easy way of passing message between two process. It provides a way of sending a *block of data* from one process to another. The main advantage of using this is, each block of data is considered to have a type, and a receiving process receives the blocks of data having different type values independently.

Creation and accessing of a message queue:
You can create and access a message queue
using the *msgget()* function.

```
#include<sys/msg.h>
int msgget(key_t key,int msgflg);
```

The first parameter is the key value, which specifies the particular message queue. The special constant IPC_PRIVATE will create a private queue. But on some Linux systems the message queue may not actually be private.

The second parameter is the flag value, which takes nine permission flags.

Adding a message:

The *msgsnd()* function allows to add a message to a
message queue. #include<sys/msg.h>
int msgsnd(int msqid,const void *msg_ptr ,size_t msg_sz,int msgflg);

The first parameter is the message queue identifier returned from an msgget function.

The second parameter is the pointer to the message to be sent. The third parameter is the size of the message pointed to by msg_ptr. The fourth parameter, is the flag value controls what happens if either the current message queue is full or within the limit. On success, the function returns 0 and a copy of the message data has been taken and placed on the message queue, on failure -1 is returned.

Retrieving a message:

The *smirch()* function retrieves message from the message queue.

#include<sys/msg.h>
int msgsnd(int msqid,const void *msg_ptr
,size_t msg_sz,long int msgtype ,int msgflg); The second parameter is a pointer t

The fourth parameter allows a simple form of reception priority. If its value is 0,the first available message in the queue is retreived. If it is greater than 0,the first message type is retrived. If it is less than 0,the first message that has a type the same a or less than the absolute value of msgtype is retrieved.

On success, msgrcv returns the number on bytes placed in the receive buffer, the message is copied into the user-allocated buffer and the data is deleted from the message queue. It returns -1 on error.

#include<sys/msg.h>
int
msgctl(i
nt
msgid,in
t
comman
d, struct
msqid_d
s *buf);

The second parameter takes the values as given below:

1.) IPC_STAT - Sets the data in the msqid_ds to reflect the values

associated with the message queue.

2.) IPC_SET - If the process has the permission to do so, this sets the values associated with the message queue to those provided in the msgid_ds data structure.

3.) IPC_RMID-Deletes the message queue.

(Note: If the message queue is deleted while the process is writing in a msgsnd or msgrcv function, the send or receive function will fail.

UNIT-V

Shared Memory:

Shared memory is a highly efficient way of data sharing between the running programs. It allows two unrelated processes to access the same logical memory. It is the fastest form of IPC because all processes share the same piece of memory. It also avoidscopyingdataunnecessarily.

As kernel does not synchronize the processes, it should be handled by the user. Semaphore can also be used to synchronize the access to shared memory.

Usageofsharedmemory:
To use the shared memory, first of all one process should allocate the segment, and then each process desiring to access the segment should attach the segment. After accessing the segment, each process should detach it. It is also necessary to deallocate thesegmentwithoutfail.

Allocating the shared memory causes virtual pages to be created. It is important to note that allocating the existing segment would not create new pages, but will return theidentifierfortheexistingpages.

All the shared memory segments are allocated as the integral multiples of the system's page size, which is the number of bytes in a page of memory.

Unix kernel support for shared memory

- There is a shared memory table in the kernel address space that keeps track of all shared memory regions created in the system.
- Each entry of the tables store the followingdata:

1. Name
2. Creator user ID and group ID.
3. Assigned owner user ID and group ID.
4. Read-write access permission of the region.
5. The time when the last process attached to the region.
6. The time when the last process detached from the region.
7. The time when the last process changed control data of the region.
8. The size, in no. of bytes of the region.

UNIX APIs for

shared memory

shmget

- Open and create a shared memory.

- Function prototype:

#include<sys/types.h>
#include<sys/ipc.h> #include<sys/shm.h>

int shmget (key_t key, int size, int flag);

- Function returns a positive descriptor if it succeeds or -1 if it fails.

Shmat

- Attach a shared memory to a process virtual address space.
- Function prototype:

  void * shmat (int shmid, void *addr, int flag);

- Function returns the mapped virtual address of he shared memory if it succeeds or -1 ifit fails.

Shmdt

- Detach a shared memory from the process virtual address space.
- Function prototype:
- Function returns 0 if it succeeds or -1 if it fails.

Shmctl

- Query or change control data of a shared memory or delete thememory.
- Function prototype:

  #inc

  lude

  int shmctl (int shmid, int cmd, struct shmid_ds *buf);

Function returns 0 if it succeeds or -1 if it fails.

Shared memory Example

//shmry1.c

TEXT_

SZ

2048

struct

```
shared_

use_st
{
int written_by_you;
char some_text[TEXT_SZ];
};

int main()
{
int running = 1;
void
*shared_memor
y   =   (void   *)0;
struct
shared_use_st
*shared_stuff;
int shmid;
srand(
(unsigned
int)getpid() );
shmid =
shmget(
(key_t)1234,
sizeof(struct
shared_use_st
), 0666
|IPC_CREAT
);

if (shmid == -1)
{
fprintf(stderr, "shmget failed\n");
exit(EXIT_FAILURE);
}
shared_memory =
shmat(shmid,(void *)0, 0);
if (shared_memory ==
(void *)-1)
{
fprintf(stderr,
"shmat failed\n");
exit(EXIT_FAILU
RE);
}

printf("Memory
Attached at
%x\n",
(int)shared_mem
ory);

 shared_stuff =
(struct shared_use_st
*) shared_memory;
shared_stuff-
>written_by_you =
0; while(running)

{
if(shared_stuff->written_by_you)
{

printf("You Wrote: %s",
shared_stuff->some_text);

sleep( rand() %4 );
shared_stuff->written_by_you = 0;

if
(strncmp(shared_st
uff->some_text,
"end", 3)== 0)
{
running = 0;
}
}
}


if (shmdt(shared_memory) == -1)

{
fprintf(stderr,
"shmdt
failed\n");
exit(EXIT_F
AILURE);
}
if (shmctl(shmid, IPC_RMID, 0) == -1)
{
fprintf(stderr, "failed to delete\n");
exit(EXIT_FAILURE);
}

  exit(E
  XIT_
  SUC
  CESS
    );
```

```
        }

        .
        h
        >

        #include<sys/shm.h>

        #define
        TEXT_
        SZ
        2048
        struct
        shared_
        use_st
        {
        int written_by_you;
        char some_text[TEXT_SZ];
        };

        int main()
        {
        int running =1
void *shared_memory = (void *)0; struct shared_use_st *shared_stuff;

        int shmid;

                                        shmid
        =shmget( (key_t)1234, sizeof(struct
        shared_use_st),
        0666 | IPC_CREAT);
        if (shmid == -1)
        {
        fprintf(stderr,
        "shmget
        failed\n");
        exit(EXIT_FA
        ILURE);
        }

        shared_mem
        ory=shmat(s
        hmid, (void
        *)0, 0);
        if (shared_memory == (void *)-1)
        {
        fprintf(stderr,
        "shmat failed\n");
        exit(EXIT_FAILU
        RE);
        }
```

```
printf("Memory Attached at %x\n", (int)
shared_memory); shared_stuff = (struct
shared_use_st *)shared_memory;
while(running)
{
while(shared_stuff->written_by_you== 1)
{
sleep(1);
printf("waiting for client... \n");
}
printf("Ent
er Some
Text: ");
fgets
(buffer,
BUFSIZ,
stdin);
strncpy(shared_stuff-
>some_text, buffer,
TEXT_SZ);
shared_stuff->written_by_you = 1;
if(strncmp(buffer, "end", 3) == 0)
{
running = 0;
}
}
if (shmdt(shared_memory) == -1)
{
fprintf(stderr,
"shmdt
failed\n");
exit(EXIT_F
AILURE);
}
exit(
EXIT
_SU
CCE
SS);
}
```

The *shmry1.c* program will create the segment using *shmget()* function and returns the identifier shmid. Then that segment is attached to its address space using *shmat()* function.

The structure *share_use_st* consists of a flag *written_by_you* is set to 1 when data is available. When it is set, program reads the text, prints it and clears it to show it has read the data. The string end is used to quit from the loop. After this the segment is detached and deleted.

The *shmry2.c* program gets and attaches to the same memory segment. This is possible

with the
help of same key value *1234* used in the shmget() function. If the
written_by_you text is set, the process will wait until the previous process
reads it. When the flag is cleared, the data is written and sets the flag. This
program too will use the string "*end*" to terminate. Then the segment is
detached.

5.2 Sockets

A *socket* is a bidirectional communication device that can be used to communicate withanother process on the same machine or with a process running on other machines.Sockets are the only interprocess communication we‟ll discuss in this chapter thatpermit communication between processes on different computers. Internet programs such as Telnet, rlogin, FTP, talk, and the World Wide Web use sockets.

For example, you can obtain the WWW page from a Web server using theTelnet program because they both use sockets for network communications.To open a connection to a WWW server at www.codesourcery.com, use telnet www.codesourcery.com 80.The magic constant 80 specifies a connection to the Web server programming running www.codesourcery.com instead of some other process.Try typing GET / after the connection is established.This sends a message through the socket to the Web server, which replies by sending the home page‟s HTML source and then closing the connection—for example:

```
% telnet
www.codesourcer
y.com 80 Trying
206.168.99.1...
Connected to
merlin.codesourcery.com
(206.168.99.1). Escape character is
„^]‟.
GET /
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
...
```

3. Note that only Windows NT can create a named pipe;Windows 9*x*
programs can form only client connections.
4. Usually, you‟d use telnet to connect a Telnet server for remote logins.
But you can also use telnet to connect to a server of a different kind and
then type comments directly at it.

Introduction to Berkeley sockets

Berkeley sockets (or BSD sockets) is a computing library with an application programming interface (API) for internet sockets and Unix domain sockets, used for inter-process communication (IPC).

This list is a summary of functions or methods provided by the Berkeley sockets API library:

- socket() creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.
- bind() is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.
- listen() is used on the server side, and causes a bound TCP socket to enter listening state.

connect() is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.

- accept() is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.
- send() and recv(), or write() and read(), or sendto() and recvfrom(), are used for sending and receiving data to/from a remote socket.
- close() causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.
- gethostbyname() and gethostbyaddr() are used to resolve host names and addresses. IPv4 only.
- select() is used to pend, waiting for one or more of a provided list of sockets to be ready to read, ready to write, or that have errors.
- poll() is used to check on the state of a socket in a set of sockets. The set can be tested to see if any socket can be written to, read from or if an error occurred.
- getsockopt() is used to retrieve the current value of a particular socket option for the specified socket.
- setsockopt() is used to set a particular socket option for the specified socket.

IPC over a network Socket Concepts

When you create a socket, you must specify three parameters:

- communication style,
- namespace,
- protocol.

A communication style controls how the socket treats transmitted data and specifies the number of communication partners.When data is sent through a socket, it is ackaged into chunks called *packets*.The communication style determines how these packets are handled and how they are addressed from the sender to the receiver.

Connection styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the

network, the receiver automatically requests their retransmission
from the sender.
A connection-style socket is like a telephone call:The addresses of the sender and receiver are fixed at the beginning of the communication
when the connection is established.

Datagram styles do not guarantee delivery or arrival order.
Packets may be lost or reordered in transit due to network
errors or other conditions. Each packet must be labeled with its destination and is not guaranteed to be delivered.The system guarantees only "best effort," so packets may disappear or
arrive in a different order than shipping.
A datagram-style socket behaves more like postal mail.The sender specifies the receiver‟s address for each individual message.

A socket namespace specifies how *socket addresses* are written. A socket address identifies one end of a socket connection. For example, socket addresses in the "local namespace" are ordinary
filenames. In "Internet namespace," a socket address is composed of the Internet address (also known as an *Internet Protocol address* or *IP address*) of a host attached to the network and a port number.The port number distinguishes among multiple sockets on the same host.A protocol specifies how data is transmitted. Some protocols are TCP/IP, the primary networking protocols used by the Internet; the AppleTalk network protocol; and the UNIX local communication

Client-server datagram socket — example

To experiment with datagram sockets in the UNIX domain we will write a client/server application where:

- the client takes a number of arguments on its command line and send them to the server using separate datagrams
- for each datagram received, the server converts it to uppercase and send it back to the client
- the client prints server replies to standard output

For this to work we will need to bind all involved sockets to pathnames.

Client-server datagram socket example — protocol

```
#includ
e
<ctype
.h>
#includ
e
<sys/un
.h>
#includ
e
<sys/so
cket .h>
#includ
e
```

```
<unistd
.h>
#includ
e "
helpers
.h"
#define SRV_SOCK_PATH "
/tmp/uc_srv_socket " #define
CLI_SOCK_PATH " /tmp/ uc_cl
i_socket .%ld " #define MSG_LEN 10
#include "uc�proto .h"
int main( int argc ,
char *argv [ ] ) {
struct sockaddr_un
srv_addr , cl i_addr ;
int srv_fd , i ;
s
s
i
z
e
_
t

b
y
t
e
s

;

s
o
c
k
l
e
n
_
t

l
e
n

;
char buf [MSG_LEN] ;
i f ( ( srv_fd = socket (AF_UNIX ,
SOCK_DGRAM, 0) ) < 0) err_sys ( " socket
error " ) ;
```

```
memset(&srv_addr , 0, sizeof (
struct sockaddr_un ) ) ; srv_addr .
sun_family = AF_UNIX ;
strncpy ( srv_addr . sun_path ,
SRV_SOCK_PATH, sizeof (
srv_addr . sun_path ) �1) ;
i f ( access ( srv_addr .
sun_path , F_OK) == 0)
unlink ( srv_addr . sun_path
) ;
i f ( bind ( srv_fd , ( struct
sockaddr * ) &srv_addr , sizeof (
struct sockaddr_un ) ) < 0)
err_sys ( " bind error " ) ;
for ( ; ; ) {
len = sizeof ( struct sockaddr_un ) ;
i f ( ( bytes = recvfrom( srv_fd ,
buf , MSG_LEN, 0, ( struct
sockaddr * ) &cl i_addr , &len ) )
< 1) err_sys ( " recvfrom error " )
;
pr int f ( " server received %ld
bytes from %s\n" , ( long) bytes
, cl i_addr . sun_path ) ;
for ( i = 0; i < bytes ; i ++)
buf [ i ] = toupper ( (
unsigned char ) buf [ i ] ) ; i f
( sendto ( srv_fd , buf , bytes
, 0,
( struct sockaddr * ) &cl
i_addr , len ) != bytes )
err_sys ( " sendto error " ) ;

#include "uc�proto .h"
int main( int argc ,
char *argv [ ] ) {
struct sockaddr_un
srv_addr , cl i_addr ;
int srv_fd , i ;
s
i
z
e
_
t

l
e
n

;
```

```
s
s
i
z
e
_
t

b
y
t
e
s

;
char
resp
[MS
G_LE
N] ; i
f (
argc
< 2)
er r_qui t ( "Usage : uc�c l ient MSG. . . " ) ;
i f ( ( srv_fd = socket (AF_UNIX ,
SOCK_DGRAM, 0) ) < 0) err_sys ( " socket
error " ) ;
memset(&cl i_addr , 0, sizeof (
struct sockaddr_un ) ) ; cl i_addr .
sun_family = AF_UNIX ;
snpr int f ( cl i_addr . sun_path , sizeof ( cl
i_addr . sun_path ) , CLI_SOCK_PATH, (
long) getpid ( ) ) ;
i f ( bind ( srv_fd , ( struct
sockaddr * ) &cl i_addr , sizeof (
struct sockaddr_un ) ) == �1)
err_sys ( " bind error " ) ;
```

Notes:
the server is persistent and processes one datagram at a time, no matter the client rocess, i.e. there is no notion of connection messages larger than 10 bytes are silently truncated

Socket address structures(UNIX domain &

Internet domain) UNIX domain Sockets:

We now want to give an example of stream sockets. To do so, we can longer remain in the abstract of general sockets, but we need to pick a domain. We pick the UNIX domain. In the UNIX domain, addresses are

pathnames. The corresponding Cstructure is sockaddr_un: struct
sockaddr_un {
sa_fami ly_t sun_family ; /* = *AF_UNIX* */

char sun_path[108] ; /* *socket*
pathname, NULL�terminated */
}
The field sun_path contains a regular pathname, pointing to a special file of type socket (. pipe) which will be created at bind time.
During communication the file will have no content, it is used only as a *rendez-vous* point between processes.

Internet-Domain Sockets

UNIX-domain sockets can be used only for communication between two processes on the same computer. *Internet-domain sockets*, on the other hand, may be used to connect processes on different machines connected by a network.
Sockets connecting processes through the Internet use the Internet namespace represented by PF_INET.The most common protocols are TCP/IP.The *Internet Protocol (IP)*, a low-level protocol, moves packets through the Internet, splitting and rejoining the packets, if necessary. It guarantees only "best-effort" delivery, so packets may vanish or be reordered during transport. Every participating computer is specified using a unique IP number.

The *Transmission Control Protocol (TCP)*, layered on top of IP, provides reliable connection-ordered transport. It permits telephone-like connections to be established between computers an ensures that data is delivered reliably and inorder.

DNS Names

Because it is easier to remember names than numbers, the *Domain Name Service (DNS* associates names such as www.codesourcery.com with computers‟ unique IP numbers. DNS i implemented by a worldwide hierarchy of name servers, but you don‟t need to understand DN protocols to use Internet host names in your programs.

Internet socket addresses contain two parts: a machine and a port number.This information i stored in a struct sockaddr_in variable. Set the sin_family field to AF_INET to indicate that thi is an Internet namespace address.The sin_addr field stores the Internet address of the desire machine as a 32-bit integer IP number.A *port number* distinguishes a given machine‟s differen sockets. Because different machines store multibyte values in different byte orders, use htons t convert the port number to

network byte order. See the man page for ip for more information.To convert human-readabl hostnames, either numbers in standard dot notation (such as 10.0.0.1) or DNS names (such a www.codesourcery.com) into 32-bit IP numbers, you can use gethostbyname.This returns pointer to the struct hostent structure; the h_addr field contains the host‟s IP number.

System Calls

Sockets are more flexible than previously discussed communication techniques.These are the system calls involving sockets:

socket—Creates a socket

closes—Destroys a socket

connect—Creates a connection between two sockets

bind—Labels a server socket with an address

listen—Configures a socket to accept conditions

accept—Accepts a connection and creates a new socket for the connection

Sockets are represented by file descriptors.

Creating and Destroying Sockets

Sockets are IPC objects that allow to exchange data between processes running:

either on the same machine (*host*), or on different ones over a network.

The UNIX socket API first appeared in 1983 with BSD 4.2. It has been finally standardized fo the first time in POSIX.1g (2000), but has been ubiquitous to every UNIX implementation sinc the 80s.

The socket API is best discussed in a network programming course,which this one is *not*. W will only address enough general socketconcepts to describe how to use a specific socket famil UNIXdomain sockets.

Connection Oriented Protocol

Client-server setup

Let‟s consider a typical client-server application scenario — no matter if they are located on the same or different hosts.

Sockets are used as follows:

each application: create a socket

idea: communication between the two applications will flow through an imaginary "pipe" that *will* connect the two sockets together

server: bind its socket to a well-known address

we have done the same to set up *rendez-vous* points for other IPC objects.

e.g. FIFOs
client: locate server socket (via its well-known address) and “initiate communication”1 with the server.

Socket options:

In order to tell the socket to get the information about the packet destination, we should call setsockopt().

setsockopt()* and *getsockopt() - set and get options on a socket. Both methods return 0 on success and -1 on error.

Prototype: int setsockopt(int sockfd, **int level**, int optname,...
There are two levels of socket options:

To manipulate options at the sockets API level: SOL_SOCKET
To manipulate options at a protocol level, that protocol number should be used;
for example, for UDP it is IPPROTO_UDP or SOL_UDP (both are equal 17) ; see include/linux/in.h and include/linux/socket.h
● SOL_IP is 0.
● There are currently 19 Linux socket options and one another on option for BSD compatibility.
● There is an option called IP_PKTINFO.

We will set the IP_PKTINFO option on a socket in the following example.
// from /usr/include/bits/in.h
#define IP_PKTINFO 8 /* bool */
/* Structure used for IP_PKTINFO. */

```
#include <fcntl.h>

int fcntl(int fildes, int cmd);
int fcntl(int fildes, int cmd, long arg);
```

returns a new file descriptor with a numerical value equal to or greater than the integer **newfd**.

The call,

```
fcntl(fildes, F_GETFD)
```

returns the file descriptor flags as defined in **fcntl.h**.

The call,

```
fcntl(fildes, F_SETFD, flags)
```

is used to set the file descriptor flags, usually just **FD_CLOEXEC**.

The calls,

```
fcntl(fildes, F_GETFL)
fcntl(fildes, F_SETFL, flags)
```

respectively get and set the file status flags and access modes**.**

5.9 Comparision of IPC mechanisms.

IPC mechanisms are mianly 5 types
1.pipes:it is related data only send from one pipe output is giving to another pipe input toshare resouses pipe are used drawback:itis only related process onlycommunicated
2.message queues:message queues are un related process are also communicate with message queues.
3.sockets:sockets also ipc it is comunicate clients and server
with socket system calls connection oriented and connection less also
4.PIPE: Only two related (eg: parent & child) processess can be communicated. Data reading would be first in first out manner.
Named PIPE or FIFO : Only two processes (can be related or unrelated) can communicate. Data read from FIFO is first in first out manner.
5.Message Queues: Any number of processes can read/write from/to the queue. Data can be read selectively. (need not be in FIFO manner)
6.Shared Memory: Part of process's memory is shared to other processes. other processes ca read or write into this shared memory area based on the permissions. Accessing Shared memor is faster than any other IPC mechanism as this does not involve any kernel leve switching(Shared memory resides on user memory area).
Semaphore: Semaphores are used for process synchronisation. This can't be used for bulk data tra between processes.