Advanced UNIX Lab

Authors
P. Venkateswara Rao
T.S.R.K. Prasad

August 31, 2011

Version 1.1.1

This work is released under
Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

Table of Contents

Introduction

Lab Objectives

The **Advanced UNIX Lab** has the following objectives.

- Make the students familiar and comfortable with useful UNIX utilities.
- Provide sufficient hands-on experience to enable the students to write meaningful shell scripts.
- Develop the confidence and skills of the students to undertake UNIX systems programming.
- Reinforce the concepts taught in the **Advanced UNIX Programming** course through relevant lab work.
- Encourage the students to work on UNIX platforms.

Lab Outcomes

After completing the course, the student will have the following skill set.

- The student can use the UNIX platforms to complete regular computing tasks with ease.
- The student can complete complex programming tasks by pipelining the standard UNIX utilities.
- The student can write meaningful shell scripts to undertake regular administrative tasks on the computer.
- The student has sufficient knowledge of the systems programming to undertake medium complexity software projects on UNIX platforms.
- The student can write custom-utilities to replace the standard software utilities available on the UNIX platforms.

Lab Programs and Schedule

The list of lab programs, the program statements and the tentative schedule are given below for your reference.

LAB CYCLE - I

1. vi Editor Practice (06/06/2011 to 10/06/2011)
   Practice of text, command and colon modes of vi editor using the sample files provided.

2. Commands - I (13/06/2011 to 18/06/2011)
   Practice the following commands.
   Directory-related commands: *pwd, mkdir, cd, rmdir*
   File handling commands: *ls, chmod, chgrp, chown, cp, rm, more, mv, touch*
   Text processing (filter) commands: *w, who, head, tail*

3. Commands - II (20/06/2011 to 25/06/2011)
   Practice the following text processing (filter) commands:
   sort, uniq, cut, paste, join, cmp, diff, tr, grep-family, sed, awk

4. Commands - III (27/06/2011 to 02/07/2011)
   [To be done in the virtual machine]
   Practice the following commands.
   Disk utilities: *du, df, mount, umount, find*
   Backup utilities: *tar, cpio, dump, gzip, gunzip*
   Networking utilities: *mail, finger, ssh, sftp, scp, write, wall*

5. Shell Program - I (04/07/2011 to 08/07/2011)
   Write the shell scripts for the following:

   (a) Display all the words which are entered as command-line arguments.

   (b) Changes permissions of files in ***pwd***[1] as *rwx* for users.

   (c) To print the list of all subdirectories in the current directory.

   (d) Program which receives any year from the keyboard and determine whether the year is a leap year or not. If no argument is supplied, the current year should be assumed.

   (e) Program which takes two filenames as command-line arguments and if their contents are the same, then deletes the second file.

6. Shell Program - II (11/07/2011 to 16/07/2011)
   Write the shell scripts for the following:

   (a) To print the given number in the reverse order.

[1] pwd - print wo rking directory or present working directory. In the present case, pwd represents the present working directory.

(b) To print the first 25 Fibonacci numbers.

(c) To print the prime numbers between any specified range.

(d) To print the first 50 prime numbers.

(e) To generate prime numbers using Euler's prime generating equation.

7. Shell program - III (18/07/2011 to 23/07/2011)
   Write the shell scripts for the following:

   (a) To delete all the lines containing the word 'unix' in the files supplied as command-line arguments.

   (b) Menu driven program which has the following options.

      i. Display the contents of **/etc/passwd**.
      ii. List of users who are currently login.
      iii. Print the ***pwd***.
      iv. exit

   (c) A file contains sorted list of cities. Write a menu driven program to perform the following operations on the file.

      i. Insert a new city into the list.
      ii. Delete an existing city from the list.
      iii. Search the list for a city.
      iv. exit

LAB CYCLE - II

8. File Management - I (25/07/2011 to 30/07/2011)
   Program to transfer the data from one file to another file by using unbuffered I/O.

9. File Management - II (08/08/2011 to 12/08/2011)
   Write a program to demonstrate the *stat()*, *fcntl()* and the directory-related system calls.

10. Process Management - I (16/08/2011 to 20/08/2011)

    (a) Program to create two processes that run a loop in which one process adds all the even numbers and the other process adds all the odd numbers (Hint: use *fork()*).

    (b) Program to create a process 'i' and send data to another process 'j', which prints the same after receiving it. (Hint: use *vfork()*).

11. Process Management - II (22/08/2011 to 27/08/2011)

    (a) Program to demonstrates an orphan process.

    (b) Program to demonstrate a zombie process. Demonstrate a solution to the zombie problem using *wait4()* system call.

12. Deadlocks (29/08/2011 to 03/09/2011)
    Program which demonstrates the deadlock between two processes.

13. Pipes and Shared Memory (05/09/2011 to 09/09/2011)
    Programs on interprocess communication (IPC) using pipes and shared memory.

    (a) Write a program to demonstrate the size of the unnamed pipe.

    (b) Using named pipes, write the **reader** – **writer** programs to exchange variable length messages and print them.

    (c) Use shared memory to exchange the data between two processes through a data structure.

14. Semaphores (12/09/2011 to 17/09/2011)
    Interprocess communication (IPC) through the shared memory using semaphores for synchronization.

    (a) Create **reader** – **writer** programs to exchange the information through the shared memory. The access to the shared memory is to be synchronized with the help of a semaphore.

    (b) Implement multiway synchronization between multiple **writers** and single **reader** by using a semaphore set.

15. Sockets (19/09/2011 to 24/09/2011)
    Write the client/server programs to implement the *echo service* using UNIX, TCP and UDP sockets.

LAB EXAM (26/09/2011 to 30/09/2011)

Lab Guidelines

Here are a few guidelines to have productive experience in the lab:

Text Book The **Advanced UNIX Lab** requires significant use of UNIX commands and function calls. A student with a text book at hand for reference purpose, has greater chance of completing the lab successfully. Hence, all the students are required to bring a relevant text book to the lab. The list of accepted text books is specified for each lab.

Observation Book To bring uniformity in the quality of the observation books, we are providing a few guidelines. You are going to write a lot of program code and program output in the observation book. Hence it pays to start with a bigger book so that there is no need to get a second observation book in the middle of the semester. Please use an observation book with at least 200 pages in it. On average, the **Advanced UNIX Lab** takes more than 1.5X the paper when compared with the other labs. Use your past experience and get a bigger book if necessary.

Another suggestion is not to skimp here and go for a new non-recycled and unruled paper book. A very specific difficulty the instructors face while correcting the observation books is the lack of visibility on the paper. Visibility is clearly improved by having a milky white paper background. Hence, get a long, unruled white paper book.

The lab observation book is to be exclusively used for **Advanced UNIX Lab** and not for any other purpose.

Observation Front Matter To help put the lab schedule into perspective, all the students are required to copy the **lab schedule** onto the first pages of the observation book. Optionally, you can pin a printout of the **lab schedule** to the second paper of the observation book.

Prelab Work All the labs have viva voce questions. The students are expected to read the theory behind the questions in the text books, distill the answers in their own words and complete the viva voce questions. **It is mandatory that all the students complete the viva voce questions in their record before coding the programs on the computer**.

Each lab has the detailed list of program objectives. The students are advised to go through these objectives well in advance and prepare a skeleton code sketch to complete the lab. The students are advised to write the tentative code in the observation book before they come to the lab. This practice of thinking ahead helps the students complete the lab well in-time.

Lab Work The major emphasis in the lab is to complete the learning objectives of the lab program. During the practice of commands and utilities, the students are advised to practice all major options of a utility to get a glimpse of the versatility of each utility.

The shell programs and system programs require the students to write program code and execute the code. **As future programmers you are expected to write clear,**

indented and efficient code with sufficient comments. The student should provide the proposed input to the program and the expected output from the program. A program without proper indentation and commenting shall not be corrected.

Postlab Work A well prepared student completes the lab within the three hours allocated for the lab. A student may not be able to complete the lab within three hours if she/he hits a debugging issue. Those who could not complete the lab in-time are expected to complete the lab in their free time. The students are expected to get the lab work signed by the lab instructor at least two days before the next lab date.

Observation Outline

The students are advised to write the observation book for the labs #1 to #4 in the following order.

1. Lab number, program title and date at the top (right side)
2. Program statement and program explanation (right side)
3. Lab instructions, if any (right side)
4. Commands and utilities with proper syntax followed by the output (left side)
   Make sure that the lab instructions and the commands, output are at similar locations on the facing pages.
5. A blank paper (2pages) between labs

The students are advised to write the observation book for the labs #5 to #15 in the following order.

1. Lab number, program title and date at the top (right side)
2. Program statement and program explanation (right side)
3. Skeleton code, if any (right side)
4. Proposed input to the program(left side)
5. Expected output from the program.(left side)
   Make sure that the skeleton code and the proposed input, output are written at similar locations on the facing pages.
6. Complete working code (right side)
7. Compilation steps, observations and output (left side)
   Make sure that the working code and the compilation steps, observations, output are written at similar locations on the facing pages.
8. A blank paper (2pages) between labs

Online Submissions All the students are required to submit the transcript of their lab sessions and the program files through the MOODLE course management system. In order to complete the online submission successfully, you are advised to follow the routine outlined below while you do the lab work.

1. Create a directory named `regno_lab#`
2. Do all the lab work in that directory. For some labs, the sample files are provided on the lab website. Download the files into the directory created in step-(1).
3. Before you start the work, type the command
   `$script -a transcript.txt`
   to start the transcript recording session
4. Do the lab work
5. At the end of the session, type the command
   `$exit`
   to stop the transcript recording session.
6. If you need to complete the lab at home, copy the entire directory into pen drive / server and continue the work by repeating the steps (3)-(4)
7. Once the work is complete, compress the directory
   `$tar cvf regno_lab#.tar regno_lab#`
8. Upload the `regno_lab#.tar` file into the course site.

The online submission of the lab work is a mandatory requirement to get the record corrected.

Record Outline

The students are advised to write the record for the labs#1 to #4 in the following format.

1. Lab number, program title and date at the top (right side)
2. Program statement (right side)
3. Viva voce questions and answers (right side)
4. Lab instructions (right side)
5. Commands and utilities with proper syntax followed by the output (left side)
   Make sure that the lab instructions and the commands, utilities with proper syntax, output are written at similar locations on the facing pages.
6. A blank paper (2 pages) between the labs

The students are advised to write the record for the labs#5 to #15 in the following format.

1. Lab number, program title and date at the top (right side)
2. Program statement (right side)
3. Viva questions (right side)
4. Complete code (right side)
5. Compilation steps, observations and output (left side)
6. A blank paper (2 pages) between the labs

Handwriting The past experience suggests that the students are very meticulous when it comes to presentation. This reminder is only for those who have the habit of completing the observation / record writing work in a haphazard manner. Even though the instructors do not look for calligraphic writing, the handwriting must at least be legible in order to evaluate the submission.

Time line In the interest of continuous student learning, we follow a very strict time line for the lab. The required learning objectives and the verification steps are provided in the *table 1*. Please follow the time line for obtaining full lab credit.

Milestone	Deadline (w.r.t. lab day)
Writeup of the viva voce in the record	lab day
Skeleton code in the observation book (Not mandatory)	lab day
Completed observation book with results	lab day + 5 days
Completed record and online submission	next lab

Table 1: Lab Time line

A penalty of 10% of the marks is levied for each day of late submission for the record.

Record Index Page The index page of the lab record can be filled up according to the *table 2*.

Serial No.	Name of the Experiment	Page No.	Date of Exp.	Date of Submission	Remarks
Week No.	Program Title	Page No.s in the record	Date on which **you** did the lab	Date on which **you** submitted the record	Marks Obtained

Table 2: Record Index Page Conventions

Linux Server All the students enrolled in the lab get a Linux server account. The server is a Cent OS 4.8 server. The account credentials for this server shall be distributed in the first lab. A major advantage of the new Linux server is that it is available on the Internet. This should enable the students to work on the Linux environment from the confines of their home/hostel. The Linux server has a location-specific identification. The identification details of the server are as follows.

College and City Centre
IP address: 10.1.1.3
Other Locations
IP address: 218.248.4.107

You can login to the server using a SSH client like ***putty***. If you are sitting in front of a Linux computer, you can use the existing terminal to login to the server using the following settings.
College and City Centre: `$ssh -p 222 username@10.1.1.3`
Other Locations: `$ssh -p 222 username@218.248.4.107`
Provide the password when prompted by the system.

More information on the remote access to the Linux server from a Windows OS based machine is available in the Appendix.

One final reminder to the students. The server is to be used exclusively to complete the lab-related work. Any other activity on the server is considered an academic offense.

Lab Environment The lab is tentatively scheduled in the *New Computer Center - I* (NC-I). All the computers of the **NC-I** lab have the dual boot capability. We will primarily utilize the **Ubuntu Linux** environment. The Linux environment does not have the Network login facility (we are still working on it!). You can log into the local user account with the credentials,
User name: exam
Password: e

Lab 1

vi Editor Practice

Program Statement:

Practice of text, command and colon modes of **vi** editor using the sample files provided.

1.1 Viva Questions

This being the very first lab in this course, the viva questions are geared more towards UNIX operating system.

1. What are the features of UNIX?
2. Is UNIX case sensitive?
3. Who created UNIX?
4. What is a terminal in UNIX?
5. What is the relationship between UNIX and Linux?
6. What are the modern versions of the **vi** editor?
7. What is the relation between the **vi** editor and the **ex** editor?
8. What other terminal text editors are popular with the programmers?
9. Write any five **vi** editor commands that you found useful in the past.
10. Write the commands to recover the unsaved files from crashed **vi** editor sessions.

1.2 Lab Session

Download the file called **lab1.tar** from the lab website. Extract the file in your working directory using the command.

```
$ tar xvf vi-lab.tar; cd vi-lab
```

1.2.1 Create and Save the File

1. Open a new file called practice
   `$vi practice`

2. Enter the insert (text) mode by selecting any of the four keys – i, I, a, I. Type the following text: "This is a practice file" into the file.

3. Now go to command mode by pressing `ESC`

4. Save the file with any of the following key combinations in colon mode: `wq, x, wq!`

Question: What happens if you just type the following commands in colon mode instead of 'x'?
`ZZ, q, q!, X, WQ, e!`
Note down your observations for all the commands.

Question: What happens if you just type the following commands in colon mode instead of 'x'?
`:w practice2`
`:w! practice2`
Note down your observations for both the commands.

1.2.2 Cursor Movement

1. Open the file called **cursor_move.txt**.
   `$vi cursor_move.txt`

2. Place the cursor at 's' of the word 'seeing' on the third line and practice the commands as shown in the figure 1.1.

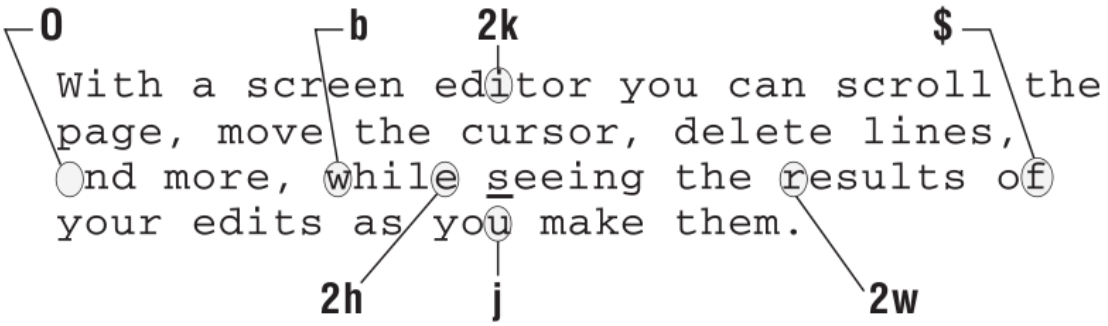

Figure 1.1: vi editor cursor movement

Question: Place the cursor at the first letter of the file and execute the following command sequence in the command mode of the **vi** editor.
`16w 4b 2j 0 6l k $`
Note down the cursor position.
Repeat the exercise for: `16W 4B 2j 0 6l k $`

where is the cursor now? Explain your answer.

1.2.3 Simple Edit

1. Open a file called simple_edit
   `$vi simple_edit`

2. Practice the simple edit commands on the file with the help of figure 1.2.

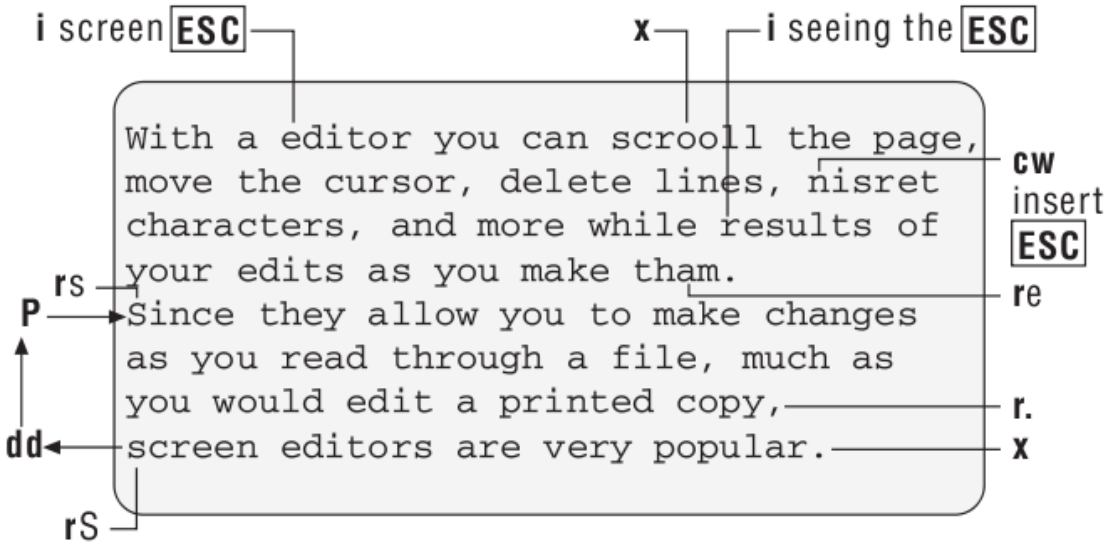

Figure 1.2: Simple edit commands in vi editor

Question: Place the cursor on the second line and execute - `2dd`; Go to next line and execute – `P` (capital letter).
Explain the result.

Question: Place the cursor on the second line and execute - `2dd`; Go to next line and execute – `PPPP` (capital letter).
Explain the result. Is the behavior like *ctl-x* and *ctl-v ctl-v ctl-v ctl-v* of the modern word processors?

Question: Place the cursor in the middle of the second line, and practice the commands – `c2, c2w, c0` and `c$` - one after the other.

Write down your observations.

Question: Type 'J' in the command mode and observe the result. What happened?

1.2.4 Complex Edits

Open the file **vi-intro-practice.txt** and perform the formatting commands to join the command summary into proper tabular form.

Ex:	SPACE	advance the cursor one position
	u	undoes the last change

Join the lines of the third paragraph to correct the inappropriate new lines. The resulting file should have the same format as **vi-intro.txt**.

Question: Go to line 5, select `y}`; Now go to line 13 and select `p`. What is the result? What would have happened if you went to line 3, typed `d{` and followed it up with `p` on line 13?

1.2.5 Screen Movement

1. Use **services** file for practice.
   `$vi services`
2. Start at the first line and input 'ctl-f' five times. Write down the contents of the current line.
3. Go to the last line and input 'ctl-b'five times. Write down the contents of the current line.
4. What will happen if you select '`z`.'?

1.2.6 Named Buffers

1. Continue to work with **services** file for this part.
2. Delete the lines 1-12 at one go using `12dd`; Delete the new 1-12 lines at one go using `12dd`. Now type `"2P` followed by `"1P`.
   What do you observe?
3. Go to the line
   tcpmux 1/tcp
4. Use `"bdd` to delete the line to buffer **b**. Use `"cyy` to copy the line to buffer **c**. Execute the following command sequence.
   `"cp "bP` (note: opening double quote is part of the command)
   Write your observations.

Question: Write the command sequence to reverse the first 5 lines of a file using unnamed buffers.

1.2.7 Exit Commands

1. Continue to work with **services** file for this part.
2. Save the first 30 lines of the file to **partial.txt** with the command.
   `:1,30w partial.txt`
3. Now execute
   `:1,30w» partial.txt`
   What are your observations?
4. Open **partial.txt** without saving **services** by executing
   `:e partial.txt`

   The above command should give you an error. The changes to the current file can be discarded by using,
   `:e!  partial.txt`

Question: Write the command sequence to save the lines 30-40, 60-120 and 170-200 to file **mixed.txt**.

1.2.8 Search and Replace

1. Continue to work with **services** file for this part.
2. Search for the word *pop3* using the following string in the command mode.
   /pop3

   The next occurrence of the word can be found by selecting the key '`n`'; the previous occurrence of the same word can be found by selecting the key '`N`'.
3. Replace all the occurrences of the word *tcp* with *sctp* by executing,
   `:%s/tcp/sctp/g`

Question: Write the command string to replace the word *kerberos* with the word *mit* if it occurs between lines 1 to 100.

1.3 Books

Suggested Books - [Rob11, Gla03]
Reference Books - [Sau08, Sob05, For03]

Lab 2

Commands - I

Program Statement:

Practice the following commands.
Directory-related commands: *pwd, mkdir, cd, rmdir*
File handling commands: *ls, chmod, chgrp, chown, cp, rm, more, mv, touch*
Text processing (filter) commands: *w, who, head, tail*

2.1 Viva Questions

1. Write the syntax of all the commands used in the lab.
2. What is a present working directory? What is a home directory? What is a root directory?
3. What is the difference between the absolute pathname and the relative pathname?
4. How do you ascend the directory hierarchy?
5. How to create a directory tree?
6. What are the conditions to be satisfied for the *rmdir* to be successful?
7. Under what conditions the *mkdir* may fail?
8. What are the metacharacters?
9. What are the different file types supported by UNIX?
10. Distinguish between the *owner*, the *group* and the *others*.
11. What is the difference the *who* and *w* commands.

2.2 Program Explanation

Practice the example commands from the [Gla03] book.

2.3 Books

Suggested Books - [Gla03, Sau08, For03]
Reference Books - [Sob05, Ker84]

Lab 3

Commands - II

Program Statement:

Practice the following **text processing (filter)** commands:
sort, uniq, cut, paste, join, cmp, diff, tr, grep-family, sed, awk

3.1 Viva Questions

1. Write the syntax of all the commands used in the lab.
2. What is a filter?
3. Distinguish between *diff* and *cmp* commands.
4. What is a regular expression?
5. What is an interval regular expressions?
6. What is a tagged regular expression?
7. List the special characters that are part of the basic regular expressions. State their meaning.
8. Frame the regular expressions to match the following words.

(a) jefferies	jeffery	jeffreys	
(b) hitchen	hitchin	hitching	
(c) Heard	herd	Hird	
(d) dix	dick	dicks	dickson
(e) Mcgee	mcghee	magee	

9. Write the syntax of the *awk* functions.
10. Write the syntax of the *awk* control structures.
11. What is an associative array? How are the associative arrays used in *awk*?

3.2 Program Explanation

Practice the example commands from the [Gla03] book.

3.3 Books

Suggested Books - [Gla03, Sau08, For03]
Reference Books - [Sob05, Ker84]

Lab 4

Commands - III

Program Statement:

Practice the following commands.
Disk utilities: *du, df, mount, umount, find*
Backup utilities: *tar, cpio, dump, gzip, gunzip*
Networking utilities: *mail, finger, ssh, sftp, scp, write, wall*

4.1 Viva Questions

1. Write the syntax of the commands used in the lab.
2. Give the command sequence for determining the total size of a directory.
3. Compare the service provided by the **pipe** with that of the *xargs* utility.
4. Write the command sequence to find all the **a.out** files that are older than 10 days and delete them.
5. Explain the full and incremental backup algorithm of the *tar* utility.
6. What is the purpose of the *mount* and the *umount* commands?
7. How are the permissions of the removable media get mapped by the *mount* command?
8. List the media that can be used by the *tar* and the *cpio* commands for backup and restore.
9. What is the default cipher algorithm used by the SSH? What other cipher algorithms of the SSH are available to the users?
   (NOTE: Cipher is an algorithm that converts normal plain information to secret information and back.)
10. Name a scenario where the *wall* is useful.

4.2 Program Explanation

Practice the example commands from the [Gla03] book.

Use different options of the *ssh* command to connect to the server. Practice the *sftp* and *scp* commands by copying files to the Linux server from the local computer.

Practice the *mount* and *umount* on your home computer or on the virtual machine.

Practice the following commands on the server: *finger, dump, restore, write, wall, mail*

Practice the following commands with the help of the server: *ssh, sftp, scp*

4.3 Books

Suggested Books - [Gla03, Sau08, For03]
Reference Books - [Sob05, Ker84]

Lab 5

Shell Program - I

Program Statement:

Write the shell scripts for the following:

1. Display all the words which are entered as command line arguments.
2. Changes permissions of files in ***pwd*** as *rwx* for users.
3. To print the list of all subdirectories in the current directory.
4. Program which receives any year from the keyboard and determine whether the year is a leap year or not. If no argument is supplied, the current year should be assumed.
5. Program which takes two filenames as command line arguments and if their contents are the same, then deletes the second file.

5.1 Viva Questions

1. What is a shell?
2. List the names of the widely used shells.
3. What are the positional parameters of a shell?
4. How are the positional parameters of a shell set? Give an example.
5. How to delete a positional parameter in a shell?
6. What are the environment variables of a shell?
7. What is a subshell?
8. Are the variables created in the subshell visible in the parent shell? Justify your answer.
9. What are interpreter files? How are they different from executable binary files?
10. What are the prompt strings of a shell? When are they used?
11. Write at least two other ways to print just the subdirectories of a directory.

5.2 Program Explanation

1. Write one shell script per program.

2. For program-(3), print the list of subdirectories in the present working directory; do not print the list of all the files in the present directory. One possible way to print the list of sub-directories is -
   `$ls -l|grep '^d'|cut -d '□' -f 10`
   □ - denotes whitespace

3. A year is considered a leap year if it is (divisible by 400) or (divisible by 4 but not 100).

5.3 Books

Suggested Books - [Gla03, Koc03, Sob05, Qui04]
Reference Books - [For03, Sau08]

Lab 6

Shell Program - II

Program Statement:

Write the shell scripts for the following:

1. To print the given number in the reverse order.
2. To print the first 25 Fibonacci numbers.
3. To print the prime numbers between any specified range.
4. To print the first 50 prime numbers.
5. To generate prime numbers using Euler's prime generating equation.

6.1 Viva Questions

1. How are the environment variables of a shell created?
2. Give the command to create an environment variable of a shell.
3. Explain the meaning of the following environment variables: PATH, HOME, USER, MAILDIR, SHELL, PS1, UMASK
4. How to delete a variable in the bourne shell?
5. Write the syntax of the *echo* and the *read* commands.
6. Write and explain the meaning of any ten conditional operators of the bourne shell.
7. Write the syntax of the basic arithmetic operations in the bourne shell.
8. What are the methods available for accessing the variables in the bourne shell?
9. What are the predefined local variables of the bourne shell?
10. In the context of a shell, what is the difference between single quotes, double quotes and grave quotes (back quotes)?
11. List and explain all the metacharacters used by the bourne shell.

6.2 Program Explanation

1. For program-(1), the number must be given at the command line.

2. The Fibonacci sequence is defined by,
   $F_n = F_{n-1} + F_{n-2}$ with seed values $F_0 = 0$ and $F_1 = 1$

3. A number (N) is a prime if it is not divisible by any number between 2 and (N/2). The number range must be specified as command line arguments.

4. The Euler's prime generating equation is:
   $f(n) = n^2 + n + k$ where $k = 2, 3, 5, 11, 17, 41$ and $n = 0$ to $k - 2$

   Use the equation to generate the primes for $k = 11$ and $n = 0$ to 9. Repeat the program for $k = 41$ and $n = 0$ to 39.

6.3 Books

Suggested Books - [Gla03, Koc03, Sob05, Qui04]
Reference Books - [For03, Sau08]

Lab 7

Shell Program - III

Program Statement:

Write the shell scripts for the following:

1. To delete all the lines containing the word 'unix'in the files supplied as command line arguments.
2. Menu driven program which has the following options.
   (a) Display the contents of **/etc/passwd**.
   (b) List of users who are currently login.
   (c) Print the ***pwd***.
   (d) exit
3. A file contains sorted list of cities. Write a menu driven program to perform the following operations on the file.
   (a) Insert a new city into the list.
   (b) Delete an existing city from the list.
   (c) Search the list for a city.
   (d) exit

7.1 Viva Questions

1. List the common core built-ins of the Bourne shell. State their functionality.
2. How is **.profile** file used by the Bourne shell?
3. What are the advantages of command grouping? Give three examples.
4. What is a shebang line? Give an example.
5. Write the syntax of the Bourne shell control structures.

6. List the Bourne shell redirection commands and state their functionality.
7. In the context of a shell, what is the significance of the numbers *0,1* and *2*?
8. Write the shell commands for redirecting *stdin*, *stdout* and *stderr*.
9. What job control commands are available in the Bourne shell?
10. How are the default permissions determined for the newly created files and directories?
11. State the technique used to include the header files in a Bourne shell script.
12. Are the variables created in the header files visible in the shell script? Justify your answer.
13. Write the bourne shell syntax for function declaration and usage.

7.2 Program Explanation

1. The program-(1) must take the filename as command-line argument, delete all the lines containing 'unix'from the file and save the result into the same file.
2. The following command sequence displays the list of all the users currently login to the system.
   `$who|cut -d '□' -f 1|uniq`
   □ - represents whitespace
3. For program-(3), the file name can be taken to be **cities.txt**. The list of cities in the **cities.txt** must always be in sorted order. When the script is run for the first time, **cities.txt** gets created. During the later iterations, the existing file must get used.

7.3 Books

Suggested Books - [Gla03, Koc03, Sob05, Qui04]
Reference Books - [For03, Sau08]

Lab 8

File Management - I

Program Statement:

Program to transfer the data from one file to another file by using unbuffered I/O.

8.1 Viva Questions

1. Write the syntax of the following system calls.

   a) open() *b) creat()* *c) read()* *d) write()*
   e) lseek() *f) close()*

2. Differentiate between the buffered and unbuffered file I/O.

3. Write the equivalent *open()* system call for *creat()*.

4. Why does *lseek()* system call execute faster than other system calls?

5. What is a hole? How does it occur in a file?

6. What are the conditions under which the number of bytes read through *read()* system call is less than the requested number?

7. How can the *write()* system call fail?

8. Is it possible to get a value of *0,1* or *2* as file descriptor for an ordinary file? How can it be accomplished?

9. What is an atomic operation?

10. What are the conditions under which the following errors occur?

    a) EAGAIN b) EINTR c) EIO d) ENOSPC
    e) EINVAL f) EFAULT g) EBADF h) EOVERFLOW

8.2 Program Explanation

1. Take any text file that is about 1kB in size. Rename the file as **source.txt**.

2. Write a C program to copy *source.txt* to *target.txt*. The file names must be taken as command-line arguments. For this simple program, copy one byte at a time.

   ```
   $gcc -Wall -Werror -o unbufcp unbufcp.c
   $./unbufcp source.txt target.txt
   $cmp source.txt target.txt;echo $?
   ```

   The result must be '0' for a successful copy.

3. Let us apply the program to copy a very big file. Take any file that is greater than 4MB in size. Get a sample large file from the local computer and make a fresh copy in the ***pwd*** using **unbufcp**.

   ```
   $sync[1]
   $cp /boot/vmlinuz-2.6.32-31-generic kernel
   $sync
   $./unbufcp kernel slowcopy
   ```

   The second command should take a long time. Get a comparative feel of the copying efficiency by executing:

   ```
   $sync
   $time cp kernel fastcopy
   $sync
   $time ./unbufcp kernel slowcopy
   ```

   Note down the timing results given by the computer.

4. Efficient copying can be achieved by changing the number of bytes copied per iteration to 4096 (or whatever the block size). Change the block size to 4096, recompile the `unbufcp.c` and repeat step-(2).

8.3 Books

Suggested Books - [Lov07, Gla03, Ste05]
Reference Books - [Sau08]

[1] *sync* command flushes the kernel page buffer to the hard disk. The next time a file is read, it is read from the hard disk.

Lab 9

File Management - II

Program Statement:

Write a program to demonstrate the *stat()*, *fcntl()* and the directory-related system calls.

9.1 Viva Questions

1. Write the syntax of the following system calls.

a) fcntl()	*b) mkdir()*	*c) rmdir()*
d) opendir()	*e) readdir()*	*f) closedir()*

2. Write the syntax of the *dirent* and *stat* data structures.
3. What is the purpose of the *umask()* system call?
4. What are the boot block, super block and inodes in the context of UNIX file systems?
5. What is the difference between O_SYNC, O_DSYNC and O_RSYNC flags?
6. What is the difference between the synchronous, asynchronous and synchronized operations?
7. State the characteristics of the following devices.

a) /dev/stdin	b) /dev/stdout	c) /dev/stderr
d) /dev/null	e) /dev/zero	f) /dev/full

8. What is the meaning of execute and write flags for files and directories?
9. What is a sticky bit? How is it useful?
10. What are the conditions under which the following errors occur?

a) EACCESS	b) EEXIST	c) ENOTDIR
d) EROFS	e) EPERM	f) ENOTEMPTY

9.2 Program Explanation

1. Write a menu driven C program to read a filename from command-line and perform the following tasks on the file.

   (a) Print the file type

   (b) Print the preferred block size for I/O operations

   (c) Print the file size

   (d) Print the file access times

   Use the *stat* data structure of the file to obtain the necessary information. The function call *ctime()* may be used to perform the necessary time conversion.

   The desirable sequence of compilation and execution steps are:
   `$gcc -Wall -Werror -o fileattr fileattr.c`
   `$./fileattr source.c` ...and first choice in menu
   The file is a text file.
   `$./fileattr directory` ...and first choice in menu
   The file is a directory.
   `$./fileattr pipe` ...and first choice in menu
   The file is a pipe.
   `$./fileattr source.c` ...and second choice in menu
   The preferred block size for source.c is 4096.
   `$./fileattr source.c` ...and third choice in menu
   The size of source.c is 117.
   `$./fileattr source.c` ...and fourth choice in menu
   The last access time for the file source.c is Wed Jun 30 21:49:08 1993.

2. Write a C program to print the list of files in a directory (Implement the *ls* command without any options). The directory pathname must be provided as command-line argument.

3. Write a menu driven program to take a filename from command-line and perform the following tasks.

   (a) Redirect the program output to the file.

   (b) Print the mode (O_READ, O_WRITE or O_RDWR) of the standard input for the program.

   (c) Open the file and set the O_SYNC status flag for the corresponding file descriptor.

9.3 Books

Suggested Books - [Lov07, Gla03, Ste05]
Reference Books - [Sau08]

Lab 10

Process Management - I

Program Statement:

1. Program to create two processes that run a loop in which one process adds all the even numbers and the other process adds all the odd numbers (Hint: use *fork()*).
2. Program to create to a process 'i' and send data to another process 'j', which prints the same after receiving it. (Hint: use *vfork()*).

10.1 Viva Questions

1. What is a process? What is a thread?
2. What is a process environment?
3. Write the syntax of the following system calls.

   a) fork() *b)vfork()* *c)exit()*
   d) _exit() *e) exec()*

4. What is unique about the *fork()* system call?
5. Distinguish between *fork()* and *vfork()* system calls.
6. What is **copy-on-write(CoW)**? How is it useful to the *fork()* system call?
7. Differentiate between a parent process and a child process.
8. What is the purpose of *exec*-family of system calls?
9. Distinguish between *fork()* and *exec()* system calls.
10. What are the attributes of a process? Where are they stored?
11. How is a deadlock possible in *vfork()*?

12. What are the conditions under which the following errors occur?
    a) ENOEXEC b) ETXTBSY

10.2 Program Explanation

1. Write a C program and take the integer(n) as input from the user. Use *fork()* system call to create two processes; add all the even numbers <n in the parent and add all the odd numbers < n in the child.

   Print the even and odd sums on the standard output.

2. Use *vfork()* system call to create two processes. Read a variable 'n' from the user in the child process. Print the value of 'n' in the child process and exit from the child process. In the parent process, print the value of 'n' and exit.

   What difference do you see in the output w.r.t. the program in step-(1)?

3. Convert the child part of the program in step-(2) into an infinite loop. Explain the resulting output.

4. Convert the child part of the program in step-(2) to an *execlp("ls","ls","-al",NULL)* system call. Explain the resulting output.

10.3 Books

Suggested Books - [Lov07, Gla03, Ste05]
Reference Books - [Sau08]

Lab 11

Process Management - II

Program Statement:

1. Program to demonstrate an orphan process.
2. Program to demonstrate a zombie process. Demonstrate a solution to the zombie problem using the *wait4()* system call.

11.1 Viva Questions

1. Write the syntax of the following system calls.
   a) wait() *b)waitpid()* *c) wait4()* *d) sleep()* *e) atexit()*
2. What is the relationship between real-user-ID, effective-user-ID, saved-user-ID and set-user-ID?
3. Write the syntax of the *tms* and the *rusage* data structures.
4. Distinguish between the following process times.

   a) User CPU time b) System CPU time
   c) User CPU time, terminated children d) System CPU time, terminated children
5. Differentiate between a zombie and an orphan process.
6. Define the macros for the *status* pointer returned by the *wait*-family of system calls.
7. How does the parent know about the termination of a child?
8. Who is the parent of an orphan process?
9. What are the process attributes that are not shared across *fork()*?
10. What are the conditions under which a *fork()* may fail?
11. What are the conditions under which the following errors occur?
    a) ECHILD b) EINTR c) EINVAL

11.2 Program Explanation

1. Write a C program to create two processes. Make the child process sleep for 10 sec. In the parent process, print the *processid*, the *parent processid* and return from the parent process. Once the child process wakes up, print the *processid* and the *parent processid*.

2. Modify the program in step-(1) to demonstrate zombie process.

3. Improve the program of step-(2) to avoid zombie process by using *wait()* or *waitpid()* system call.

4. Modify the program of step-(3) to print the resource usage parameters of the child by using either *wait4()* system call or *getrusage()* system call.

11.3 Books

Suggested Books - [Lov07, Gla03, Ste05]
Reference Books - [Sau08]

Lab 12

Deadlocks

Program Statement:

Program which demonstrates a deadlock between two processes.

12.1 Viva Questions

1. Write the syntax of the following system calls.
   a) *fcntl* for record locking b) *lockf*
2. Write the syntax of the *flock* data structure. Also state the need for the data structure.
3. State the rules that govern the inheritance and release of the record locks.
4. State the following problems.
   (a) Dining philosophers problem.
   (b) Readers and Writers problem.
   (c) Sleeping barber problem.
5. What is the difference between the advisory locking and the mandatory locking?
6. How is the mandatory locking implemented in UNIX?
7. Describe the file locking scenarios that lead to starvation. Propose solutions to avoid starvation.
8. State the advantages and the disadvantages of using the record locks.
9. Is a record lock better than a file lock? Justify.
10. How does the UNIX operating system detect deadlocks? How does UNIX solve the deadlock problem?

12.2 Program Explanation

1. File locking using *fcntl()* to be used for demonstration of the deadlock concept.
2. Create two new files by executing the following command sequence.
   `$w>file;ls>file1`
3. Use the **unbufcp.c** program from **lab8** to complete the present lab.
4. Copy the **unbufcp.c** file to **deadlock1.c**; Modify the **deadlock1.c** file to obtain the read lock on the source file first and then obtain the write lock on the target file. The program must obtain the locks before it commences the copy operation.
5. Copy the **unbufcp.c** file to **deadlock2.c**; Modify the **deadlock2.c** file to obtain the write lock on the target file first and then obtain the read lock on the source file. The program must obtain both the locks before it commences the copy operation.
6. Execute the following commands.
   ```
   $gcc -Wall -Werror -o deadlock1 deadlock1.c
   $gcc -Wall -Werror -o deadlock2 deadlock2.c
   $./deadlock1 source.c target.c & ./deadlock2 source.c target.c &
   ```
   Considering the execution speed of the modern computers, the chance of step-(5) resulting in a deadlock is very small.
7. In **deadlock1.c** program, add the *sleep(30)* system call after successfully acquiring the read lock on the source file.

   Similarly in **deadlock2.c** program, add the *sleep(30)* system call after successfully acquiring the write lock on the target file.

   Repeat step-(6) and verify the deadlock.

12.3 Books

Suggested Books - [Lov07, Ste05, Ste99]
Reference Books - [Roc04, Sau08, Tan01]

Lab 13

Pipes and Shared Memory

Program Statement:

Programs on interprocess communication (IPC) using pipes and shared memory.

1. Write a program to demonstrate the size of the unnamed pipe.
2. Using named pipes, write the `reader` – `writer` programs to exchange variable length messages and print them.
3. Use shared memory to exchange the data between two processes through a data structure.

13.1 Viva Questions

1. Write the syntax of the following system calls.

a) pipe()	*b) mkfifo()*	*c) popen()*	*d) pclose()*
e) shmget()	*f) shmat()*	*g) shmdt()*	*h) shmctl()*
i) mmap()	*j) munmap()*	*k) msync()*	

2. What are the advantages and the disadvantages of the following IPC mechanisms?
   a) unnamed pipes b) named pipes c) shared memory
3. List the size restrictions and other system limits on -
   a) unnamed pipes b) named pipes c) shared memory
4. State the conditions for atomic operations on pipes
5. How does O_NONBLOCK flag impact the read and the writ operations of a pipe?
6. State any two differences between a named pipe and a file.
7. Illustrate the implementation of the coprocess concept using pipes.

8. State the conditions under which the following error messages occur.

   a) SIGPIPE b) EAGAIN c) EPIPE d) EMFILE
   e) ENFILE f) SIGSEGV g) SIGBUS

9. Draw the memory layout of a C program on an Intel-based Linux system. Use the diagram to illustrate the implementation of the shared memory.

10. Is the shell **pipe** a named or an unnamed pipe?

11. Draw the pipe equivalent illustration for the following command set.
    `$grep 'hello' file1.c|sort|tee sorted.c`
    `$mkfifo fifo1; prog3 < fifo1 & prog1 < infile|tee fifo1|prog2`

12. What is the persistence level of the following.
    a) unnamed pipes b) named pipes c) shared memory

13. Write the prototypical system calls for the implementation of the shared memory using anonymous memory mapping and */dev/zero*.

13.2 Program Explanation

1. Write a C program to implement the unnamed pipe between two processes. In the program, parent writes into the pipe and child reads from the pipe.
   To determine the pipe size, we need to find the maximum data block size for atomic operations.
   In the parent process, write a data block of size 2048 bytes to pipe. Do not read the pipe data in the child. Is the operation blocking?
   Now repeat the procedure for 4096,4097 and 5000 bytes. When does the write operation becomes blocking?

   You can confirm the result by obtaining the PIPE_BUF size for the system using the command,
   `$mkdir pipe;getconf PIPE_BUF pipe`

2. Write the **reader** and the **writer** programs to exchange variable length messages using named pipes. The messages are to be exchanged with the help of the data structure given below.

```
struct {
        int msg_id;
        int msg_size;
        char msg[1024];
        } message;
```

msg_id is incremented by the **writer** for each new message; *msg_size* indicates the size of the variable length message; *msg* hold a message of maximum length 1023 bytes. The terminating '\0' of the message is not counted in the message size, but it is stored in the array. The **writer** must append '\0' at the end of the message.

The **reader** gets the message and prints the message on the standard output. The **reader** must not print the same message multiple times.

3. Modify the programs written in step-(2) to exchange the message using shared memory.

13.3 Books

Suggested Books - [Ste99, Gla03, Ste05]
Reference Books - [Sau08, Bac90, Roc04]

Lab 14

Semaphores

Program Statement:

Interprocess communication (IPC) through the shared memory using semaphores for synchronization.

1. Create **reader** – **writer** programs to exchange the information through the shared memory. The access to the shared memory is to be synchronized with the help of a semaphore.

2. Implement multiway synchronization between multiple **writers** and single **reader** by using a semaphore set.

14.1 Viva Questions

1. Write the syntax of the following system calls.
   a) ftok() b) semget() c) semctl() d) semop()

2. Write the syntax of the following data structures.
   a) ipc_perm b) semid_ds c) sembuf d) semun
   e) anonymous semaphore structure used by the kernel

3. State the advantage and the disadvantage of using the SEM_UNDO flag.

4. What is the difference between a binary semaphore, a mutex and a counting semaphore.

5. What is a spinlock?

6. What is the need for a System V semaphore set? What are the disadvantages of using the semaphore set?

7. What is the motivation for IPC?

8. Draw the logic diagram for creating or opening an IPC object.

9. Why do we need IPC_PRIVATE key?

10. What are the ways in which an IPC key can be exchanged between the processes?
11. Separate the IPC methods into two categories: Message passing and Synchronization.
12. State the advantages and the disadvantages of using the semaphores.
13. What is the meaning of an atomic operation on a semaphore set? What happens if atomicity is not guaranteed?

14.2 Program Explanation

1. The program is an extension to the shared memory program done in **lab13**. To provide synchronized access to the shared memory, a semaphore is to be used. Modify the **writer** program of the shared memory to use binary semaphore for the shared memory access.
2. Generate the IPC *key* using *ftok()* function. Use a path name of */tmp* and an id of '0'.
3. Extend the program in step-(1) to provide multiway synchronization. In the new scenario, there are two **writers** and one **reader**. We will use semaphore set idea to implement the synchronization.
4. create a semaphore set with one semaphore for each **writer**. Create two shared memory segments.
5. **writer1** uses **semaphore-1** to write the message to **shared memory-1**. **writer2** uses **semaphore-2** to write the message to **shared memory-2**. The writers take the messages from the user.

   You need multiple terminals for demonstrating the program. More terminals can be opened as tabs by using 'ctl+shift+t'. Use 'ctl+page down' to navigate between the terminals.

   In terminal-I, type
   `$./writer1`
   `Enter the message: hello` ← user enters
   `Enter the message: lab work` ← user enters

   In terminal-II, type
   `$./writer2`
   `Enter the message: there` ← user enters
   `Enter the message: is complete` ← user enters

   In terminal-III, type
   `$./reader`
   The **reader** gives output based on the activity in the terminals - I and II
   `The first message is: hello there`
   `The second message is: lab work is complete`

6. **reader** acquires both semaphores (semaphore set) before attempting a read operation on both the shared memory regions. The **reader** reads both the messages and prints the concatenated message to the standard output.

7. The **reader** must wait for both the **writers** to put in a new message before printing out the concatenated result on the screen. The **reader** is not allowed to print a message more than once.

14.3 Books

Suggested Books - [Ste99, Ste05]
Reference Books - [Sau08, Roc04]

Lab 15

Sockets

Program Statement:

Write the client/server programs to implement the *echo service* using UNIX, TCP and UDP sockets.

15.1 Viva Questions

1. Write the syntax of the following system calls.
   a) socket( ) *b) connect( )* *c) bind( )*
   e) accept( ) *f) close( )* *g) send( )*
   i) sendto( ) *j) recvfrom( )* *k) shutdown( ) l)*
   m) htons( ) *n) ntol( )* *o) ntohs( )*
   q) inet_pton( ) *r) inet_ntop( )*

2. Write the syntax of the following data structures.
   a) sockaddr_un b) sockaddr_in c) in_addr

3. Draw the client-server interaction diagram for TCP.

4. Draw the client-server interaction diagram for UDP.

5. What is the difference between the concurrent and the iterative servers?

6. What is the need for the byte ordering functions?

7. Compare the two widely used forms of IPC: shared memory and sockets.

8. Name at least four popular applications that make use of sockets?

9. Name the addresses used to identify the socket end points. Give an example.

10. Which of the file I/O system calls work on socket file descriptors? What do they do?

15.2 Program Explanation

1. Write **unixclient** and **unixserver** programs to implement the *echo service* based on UNIX sockets. The socket addresses (IP and port addresses) of the server and client are to be taken from the command line arguments.

   ```
   $./unixserver <server_port> &
   $./unixclient <server_port> <client_port>
   ```

2. Write **udpclient** and **udpserver** programs to implement the *echo service* based on UDP sockets. The socket addresses (IP and port addresses) of the server and client are to be taken from the command-line arguments.

   ```
   $./udpserver <server_port> &
   $./udpclient <server_ip> <server_port> <client_port>
   ```

3. Write **tcpclient** and **tcpserver** programs to implement *echo service* based on TCP sockets. The socket addresses (IP and port addresses) of the server and client are to be taken from the command line arguments.

   ```
   $./tcpserver <server_port> &
   $./tcpclient <server_ip> <server_port> <client_port>
   ```

NOTE:
A proper demonstration of the Internet sockets (TCP and UDP) programs require that the client and the server programs be run on different computers. The *udpserver* and the *tcpserver* can be run on the Linux server; the clients can be run on the local computer.

In the case of the UNIX sockets, both the server and the client programs must be run from the same computer.

15.3 Books

Suggested Books - [Ste04, Gla03, Ste05]
Reference Books - [Sau08]

Bibliography

[Bac90] Bach, Maurice J. *The Design of the UNIX Operating System.* Pearson Education, New Delhi, 1990.

[For03] Forouzan, Behrouz A. and Gilberg, Richard F. *UNIX and Shell Programming.* Thompson Asia Pte. Ltd., Bangalore, 2003.

[Gla03] Glass, Graham and King Ables. *UNIX for Programmers and Users.* Pearson Education, New Delhi, Third edition, 2003.

[Ker84] Kernighan, Brian W. and Pike, Rob. *The UNIX Programming Environment.* PHI, New Delhi, 1984.

[Koc03] Kochan, Stephen G. and Wood, Patrick. *UNIX Shell Programming.* Pearson Education, New Delhi, third edition, 2003.

[Lov07] Love, Robert. *Linux System Programming.* SPD / O'Reilly Media, Inc., Mumbai, 2007.

[Qui04] Quigley, Ellie. *UNIX Shells by Example.* Pearson Education, New Delhi, fourth edition, 2004.

[Rob11] Robbins, Arnold. *vi and Vim Editors.* O'Reilly Media Inc., Sebastopol, CA, USA, second edition, 2011.

[Roc04] Rochkind, Marc J. *Advanced UNIX Programming.* Pearson Education, New Delhi, second edition, 2004.

[Sau08] Saurabh Kumar. *UNIX Programming - The First Drive.* Wiley India Pvt. Ltd., New Delhi, 2008.

[Sob05] Sobell, Mark G. *A Practical Guide to Linux Commands, Editors and Shell Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[Ste99] Stevens, Richard W. *UNIX Network Programming:Interprocess Communication, Volume 2.* PHI, New Delhi, Second edition, 1999.

[Ste04] Stevens Richard W. and Fenner, Bill and Rudoff, Andrew M. *UNIX Network Programming: The Socket Networking API, Volume 1.* PHI, New Delhi, third edition, 2004.

[Ste05] Stevens, Richard W. and Rago, Stephen A. *Advanced Programming in the UNIX Environment*. Pearson Education, New Delhi, second edition, 2005.

[Tan01] Tanenbaum, Andrew S. *Modern Operating Systems*. PHI, New Delhi, second edition, 2001.

System Calls Summary

Files

File Editing

open, creat - open and possibly create a file or device
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);

NOTE: creat() is equivalent to open() with flags equal to O_CREAT|O_WRONLY|O_TRUNC

Returns: fd number on success, -1 on error

---

read - read from a file descriptor
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);

Returns: number of bytes read on success, zero on EOF, -1 on error

---

write - write to a file descriptor
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);

Returns: number of bytes written on success, zero on zero count value, -1 on error

---

lseek - reposition read/write file offset
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);

possible values for WHENCE are:

SEEK_SET	from beginning
SEEK_CUR	from current cursor position
SEEK_END	from the end

Returns: new offset on success, (off_t - 1) on failure

---

close - close a file descriptor
#include <unistd.h>
int close(int fd);

Returns: 0 on success, -1 on error

---

sync - commit buffer cache to disk
#include <unistd.h>
void sync(void);

The system call is always successful; sync() first commits inodes to buffers, and then buffers to disk.

---

fsync, fdatasync - synchronize a file's in-core state with storage device
#include <unistd.h>
int fsync(int fd);
int fdatasync(int fd);

Returns: 0 on success, -1 on error

---

stat, fstat, lstat - get file status
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);

Return: 0 on OK, -1 on error.

stat structure used in the stat-family of system calls is:

```
struct stat {
        dev_t     st_dev;     /* ID of device containing file */
        ino_t     st_ino;     /* inode number */
        mode_t    st_mode;    /* protection */
        nlink_t   st_nlink;   /* number of hard links */
        uid_t     st_uid;     /* user ID of owner */
        gid_t     st_gid;     /* group ID of owner */
        dev_t     st_rdev;    /* device ID (if special file) */
        off_t     st_size;    /* total size, in bytes */
        blksize_t st_blksize; /* blocksize for file system I/O */
        blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
        time_t    st_atime;   /* time of last access */
        time_t    st_mtime;   /* time of last modification */
        time_t    st_ctime;   /* time of last status change */
        };
```

The following POSIX macros are defined to check the file type using the st_mode field:

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	FIFO (named pipe)?
S_ISLNK(m)	symbolic link?
S_ISSOCK(m)	socket?

The following flags are defined for the st_mode field:

S_IFMT	0170000	bit mask for the file type bit fields
S_IFSOCK	0140000	socket
S_IFLNK	0120000	symbolic link
S_IFREG	0100000	regular file
S_IFBLK	0060000	block device
S_IFDIR	0040000	directory
S_IFCHR	0020000	character device
S_IFIFO	0010000	FIFO
S_ISUID	0004000	set UID bit
S_ISGID	0002000	set-group-ID bit (see below)
S_ISVTX	0001000	sticky bit (see below)

---

fcntl - manipulate file descriptor
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */);

The fcntl function is used for five different purposes.

1. Duplicate an existing descriptor (cmd = F_DUPFD)
2. Get/set file descriptor flags (cmd = F_GETFD or F_SETFD)
3. Get/set file status flags (cmd = F_GETFL or F_SETFL)

Returns: Depends on the command
F_DUPFD new file descriptor is returned.
F_GETFD returns the file descriptor flags (FD_CLOEXEC).
F_SETFD 0.
F_GETFL returns the file status flags.
For all commands, returns -1 on error.

File Management

dup, dup2 - duplicate a file descriptor
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);

Returns: new file descriptor on success, -1 on error.

---

umask - set file mode creation mask
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);

Returns: Previous umask value.

---

chmod, fchmod - change permissions of a file
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);

Symbolic permissions:

S_ISUID	04000	set user ID on execution
S_ISGID	02000	set group ID on execution
S_ISVTX	01000	sticky bit
S_IRUSR	00400	read by owner
S_IWUSR	00200	write by owner
S_IXUSR	00100	execute/search by owner
S_IRGRP	00040	read by group
S_IWGRP	00020	write by group
S_IXGRP	00010	execute/search by group
S_IROTH	00004	read by others
S_IWOTH	00002	write by others
S_IXOTH	00001	execute/search by others

Returns: This system call always succeeds and the previous value of the mask is returned.

---

link - make a new name for a file (create hard link)
#include <unistd.h>
int link(const char *oldpath, const char *newpath);

If newpath exists it will not be overwritten.

Returns: 0 on success, -1 on error.

---

unlink - delete a name and possibly the file it refers to
#include <unistd.h>
int unlink(const char *pathname);

If the name referred to a symbolic link the link is removed.
Returns: 0 on success, -1 on error.

---

rename - change the name or location of a file
#include <stdio.h>
int rename(const char *oldpath, const char *newpath);

Returns: 0 on success, -1 on error.

---

chown, fchown, lchown - change ownership of a file
#include <sys/types.h>
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);

Conditions: Only a privileged process may change the owner of a file.
If the owner or group is specified as -1, then that ID is not changed.

Returns: 0 on success, -1 on error.

---

truncate, ftruncate - truncate a file to a specified length
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);

If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as null bytes ('\0').

Conditions: With ftruncate(), the file must be open for writing.
with truncate(), the file must be writable.

Returns: 0 on success, -1 on error.

Directories

mkdir - create a directory
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);

Returns: 0 on success, -1 on error.

---

opendir - open a directory
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);

Returns: pointer to the directory stream on success, NULL on error.

---

readdir - read a directory
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);

The Linux dirent structure is defined as -

```
struct dirent {
        ino_t           d_ino;          /* inode number */
        off_t           d_off;          /* offset to the next dirent */
        unsigned short  d_reclen;       /* length of this record */
        unsigned char   d_type;         /* type of file */
        char            d_name[256];    /* filename */
        };
```

Returns: pointer to structure dirent on success, NULL on EOF or error.

---

closedir - close a directory
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);

Returns: 0 on success, -1 on error.

---

getdents - get directory entries
#include <dirent.h>
int getdents(unsigned int fd, struct linux_dirent *dirp, unsigned int count);

d_type is a byte at the end of the structure that indicates the file type. It contains one of the following values (defined in <dirent.h>):

DT_BLK	This is a block device.
DT_CHR	This is a character device.
DT_DIR	This is a directory.
DT_FIFO	This is a named pipe (FIFO).
DT_LNK	This is a symbolic link.
DT_REG	This is a regular file.
DT_SOCK	This is a Unix domain socket.
DT_UNKNOWN	The file type is unknown.

Returns: number of bytes read on success, 0 on end-of-directory, NULL on error.

---

chdir, fchdir - change working directory
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fd);

Returns: 0 on success, -1 on error.

---

rmdir - delete a directory
#include <unistd.h>
int rmdir(const char *pathname);

Conditions: The directory must be empty.
Returns: 0 on success, -1 on error.

---

getcwd - Get current working directory
#include <unistd.h>
char *getcwd(char *buf, size_t size);

Returns: buf on success, NULL on error.

Process Management

Process Control

getenv - get an environment variable
#include <stdlib.h>
char *getenv(const char *name);

Returns: pointer to value associated with name, NULL if not found.

---

putenv - change or add an environment variable
#include <stdlib.h>
int putenv(char *str);

Returns: 0 on success, nonzero on error.

---

setenv - change or add an environment variable
#include <stdlib.h>
int unsetenv(const char *name);

Returns: 0 on success, -1 on error.

---

unsetenv - delete an environment variable
#include <stdlib.h>
int unsetenv(const char *name);

Returns: 0 on success, -1 on error.

---

exit - cause normal process termination
#include <stdlib.h>
void exit(int status);

Returns: The function does not return.

_exit, _Exit - terminate the calling process
#include <unistd.h>
void _exit(int status);
void _Exit(int status); /* equivalent to _exit() system call */
Returns: The functions do not return.

---

atexit - register a function to be called at normal process termination
#include <stdlib.h>
int atexit(void (*func)(void));

Returns: 0 on success, nonzero on error.

---

fork - create a new process
#include <unistd.h>
pid_t fork(void);

Returns: 0 in child; process ID of child in parent, -1 on error.

---

vfork - spawn new process in a virtual memory efficient way
(create a child process and block the parent)
#include <unistd.h>
pid_t vfork(void);

Returns: 0 in child; process ID of child in parent; -1 on error.

---

wait, waitpid - wait for process to change state
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

The value of pid can be:

<-1	meaning wait for any child process whose process group ID is equal to the absolute value of pid.
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.
>0	meaning wait for the child whose process ID is equal to the value of pid.

The value of options is an OR of zero or more of the following constants:

WNOHANG	return immediately if no child has exited.
WUNTRACED	also return if a child has stopped.
WCONTINUED	also return if a stopped child has been resumed by delivery of SIGCONT.

If status is not NULL, wait() and waitpid() store status information in the int to which it points. This integer can be inspected with the following macros.

WIFEXITED(status)	returns true if the child terminated normally, by calling exit() or _exit(), or by returning from main().
WEXITSTATUS(status)	returns the exit status of the child.
WIFSIGNALED(status)	returns true if the child process was terminated by a signal.
WTERMSIG(status)	returns the number of the signal that caused the child process to terminate.
WCOREDUMP(status)	returns true if the child produced a core dump.
WIFSTOPPED(status)	returns true if the child process was stopped by delivery of a signal;
WSTOPSIG(status)	returns the number of the signal which caused the child to stop.
WIFCONTINUED(status)	returns true if the child process was resumed by delivery of SIGCONT.

Returns: wait(): process ID of the terminated child on success, -1 on error.
waitpid(): process ID of the child whose state has changed on success, -1 on error.

---

waitid() - wait for a process to change state
#include <sys/types.h>
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

The waitid() system call provides more precise control over which child state changes to wait for.

The idtype and id arguments select the child(ren) to wait for, as follows:

idtype == P_PID	Wait for the child whose process ID matches id.
idtype == P_PGID	Wait for any child whose process group ID matches id.
idtype == P_ALL	Wait for any child; id is ignored.

The child state changes to wait for are specified by ORing one or more of the following flags in options:

WEXITED	Wait for children that have terminated.
WSTOPPED	Wait for children that have been stopped by delivery of a signal.
WCONTINUED	Wait for (previously stopped) children that have been resumed by delivery of SIGCONT.

The following flags may additionally be ORed in options:

WNOHANG	As per waitpid().
WNOWAIT	Leave the child in a waitable state; a later wait call can be used to again retrieve the child status information.

Upon successful return, waitid() fills in the following fields of the siginfo_t structure pointed to by infop:

si_pid	The process ID of the child.
si_uid	The real user ID of the child.
si_signo	Always set to SIGCHLD.
si_status	Either the exit status of the child, or the signal that caused the child to terminate, stop, or continue. The si_code field can be used to determine how to interpret this field.

si_code is set to one of:

CLD_EXITED	(child called _exit());
CLD_KILLED	(child killed by signal);
CLD_DUMPED	(child killed by signal, and dumped core);
CLD_STOPPED	(child stopped by signal); or
CLD_CONTINUED	(child continued by SIGCONT).

Returns: waitid(): 0 on success, -1 on error.

---

wait3, wait4 - wait for process to change state, BSD style

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

wait3(status, options, rusage) ↔ waitpid(-1, status, options)
wait4(pid, status, options, rusage) ↔ waitpid(pid, status, options)

Returns: process ID of the child whose state has changed on success, -1 on error.

---

getrusage - get resource usage

```
#include <sys/time.h>
#include <sys/resource.h>
int getrusage(int who, struct rusage *usage);
```

who	meaning
RUSAGE_SELF	own resource usage statistics
RUSAGE_CHILDREN	resource usage statistics of self and all the children of the calling process that have terminated and been waited for.

The struct rusage is defined as follows:

```
struct rusage {
        struct timeval   ru_utime;    /* user CPU time used */
        struct timeval   ru_stime;    /* system CPU time used */
        long             ru_maxrss;   /* maximum resident set size */
        long             ru_ixrss;    /* integral shared memory size */
        long             ru_idrss;    /* integral unshared data size */
        long             ru_isrss;    /* integral unshared stack size */
        long             ru_minflt;   /* page reclaims (soft page faults) */
        long             ru_majflt;   /* page faults (hard page faults) */
        long             ru_nswap;    /* swaps */
        long             ru_inblock;  /* block input operations */
        long             ru_oublock;  /* block output operations */
        long             ru_msgsnd;   /* IPC messages sent */
        long             ru_msgrcv;   /* IPC messages received */
        long             ru_nsignals; /* signals received */
        long             ru_nvcsw;    /* voluntary context switches */
        long             ru_nivcsw;   /* involuntary context switches */
        };

struct timeval {
        time_t         tv_sec;    /* seconds */
        suseconds_t    tv_usec;   /* microseconds */
        };
```

The time_t is a typedef of long; The suseconds_t is normally a typedef to an integer type.

Returns: zero on success, -1 on error.

---

system - execute a shell command

```
#include <stdlib.h>
int system(const char *command);
```

Returns: return status of the command on success;
-1 on error and 127 if /bin/sh could not be executed.

---

execve - execute program
#include <unistd.h>
extern char **environ;

int execve(const char *filename, char *const argv[ ], char *const envp[ ]);
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[ ]);
int execv(const char *path, char *const argv[ ]);
int execvp(const char *file, char *const argv[ ]);

The differences among the six exec functions are:

Function	pathname	filename	Arg list	argv[ ]	environ	envp[ ]
execl	y		y		y	
execlp		y	y		y	
execle	y		y			y
execv	y			y	y	
execvp		y		y	y	
execve	y			y		y
letter in name		p	l	v		e

Returns: does not return on success, -1 on error.

Process Permissions

geteuid, getuid - get user identification
#include <unistd.h>
#include <sys/types.h>
uid_t getuid(void);
uid_t geteuid(void);

Returns: getuid(): real user ID of calling process. (Always successful)
geteuid(): effective user ID of calling process. (Always successful)

---

getegid, getgid - get group process identification
#include <unistd.h>
gid_t getgid(void);
gid_t getegid(void);

Returns: getgid(): real group ID of calling process. (Always successful)
getegid(): effective group ID of calling process. (Always successful)

---

getresuid, getresgid - get real, effective and saved user/group IDs
#define _GNU_SOURCE
#include <unistd.h>
int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);

Returns: 0 on success, -1 on error.

---

setuid, setgid - set user/group identity
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);

Returns: 0 on success, -1 on error.

---

seteuid, setegid - set effective user or group ID
#include <sys/types.h>
#include <unistd.h>
int seteuid(uid_t euid);
int setegid(gid_t egid);

Returns: 0 on success, -1 on error.

---

setreuid, setregid - set real and/or effective user or group ID
#include <sys/types.h>
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);

Returns: 0 on success, -1 on error.

---

setresuid, setresgid - set real, effective and saved user or group ID
#define _GNU_SOURCE
#include <unistd.h>
int setresuid(uid_t ruid, uid_t euid, uid_t suid);
int setresgid(gid_t rgid, gid_t egid, gid_t sgid);

Returns: 0 on success, -1 on error.

Process Groups

getpid, getppid - get parent or calling process identification
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);

Returns: getpid(): returns the process ID of the calling process. (Always successful)
getppid(): returns the process ID of the parent process. (Always successful)

---

setpgid, getpgid, setpgrp, getpgrp - set/get process group
#define _XOPEN_SOURCE_500
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);

pid_t getpgrp(void); /* POSIX.1 version */

int setpgrp(void); /* System V version */
int setpgrp(pid_t pid, pid_t pgid); /* BSD version */

Returns: setpgid(), setpgrp() returns zero on success, -1 on error.
getpgid(), setpgrp() returns process group ID on success, -1 on error.

---

setsid - creates a session and sets the process group ID
#include <unistd.h>
pid_t setsid(void);

Returns: (new) session ID on success, -1 on error.

---

tcgetsid - get session ID
#include <termios.h>
pid_t tcgetsid(int fd);

Returns: session ID on success, -1 on error.

---

tcgetpgrp, tcsetpgrp - get and set terminal foreground process group
#include <unistd.h>
pid_t tcgetpgrp(int fd);
int tcsetpgrp(int fd, pid_t pgrp);

Returns: tcgetpgrp: foreground process group ID of the terminal on success, -1 on error.
tcsetpgrp: 0 on success, -1 on error.

Interprocess Communication (IPC)

Pipes

pipe, pipe2 - create pipe
```
#define _GNU_SOURCE                 /* only for pipe2() */
#include <unistd.h>
int pipe(int pipefd[2]);
int pipe2(int pipefd[2], int flags);
```

The following values can be bitwise ORed in flags to obtain different behavior:

O_NONBLOCK	Set the O_NONBLOCK file status flag on the two new open file descriptions.
O_CLOEXEC	Set the close-on-exec (FD_CLOEXEC) flag on the two new file descriptors.

Returns: 0 on success, -1 on error.

---

popen, pclose - pipe stream to or from a process
```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

Returns: popen(): file pointer on success, NULL on error.
pclose(): 0 on success, -1 on error.

---

mkfifo - make a fifo file
```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

Returns: 0 on success, -1 on error.

---

mknod - create a special or ordinary file
```
#define _XOPEN_SOURCE 500
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

mode = (permissions & ~umask) | file type

The file type must be one of -

file type	description	appropriate dev
S_IFREG	regular file	0
S_IFIFO	FIFO (named pipe)	0
S_IFSOCK	Unix domain socket	0
S_IFCHR	character special file	major and minor no's
S_IFBLK	block special file	major and minor no's

Returns: 0 on success, -1 on error.

---

close - close a pipe file descriptor
Refer to close() system call in the **File Editing** section.

System V IPC

ftok - convert a pathname and a project identifier to a System V IPC key

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int id);
```

key_t is a typedef of int.

Returns: key_t on success, -1 on error.

IPC Permissions

```
#include <sys/types.h>
#include <sys/ipc.h>
struct ipc_perm {
        key_t           key     /* IPC key (generated through ftok() or IPC_PRIVATE */
        uid_t           cuid;   /* creator user ID */
        gid_t           cgid;   /* creator group ID */
        uid_t           uid;    /* owner user ID */
        gid_t           gid;    /* owner group ID */
        unsigned short  mode;   /* r/w permissions */
        unsigned short  __seq;  /* sequence number */
        };
```

The only valid permissions and their interpretation is as follows:

0400	Read by user.
0200	Write by user.
0040	Read by group.
0020	Write by group.
0004	Read by others.
0002	Write by others.

The following symbolic constants are also defined in the header file.

Name	Type	Description
IPC_CREAT	key_t	Create entry if key doesn't exist.
IPC_EXCL	Literal	Fail if key exists.
IPC_NOWAIT	Literal	Error if request must wait.
IPC_PRIVATE	Literal	Private key.
IPC_RMID	Literal	Remove resource.
IPC_SET	Literal	Set resource options.
IPC_STAT	Literal	Get resource options.

Semaphores

Kernel Data Structures for Semaphores

The kernel uses **semid_ds** data structure to represent the semaphore set.

```
struct semid_ds {
        struct ipc_perm   sem_perm;
        struct sem        *sem_base;   /* ptr to array of semaphores in set */
        unsigned long     sem_nsems;   /* count of sems in set */
        time_t            sem_otime;   /* last operation time */
        time_t            sem_ctime;   /* last change time */
        };
```

sem_perm	ipc_perm structure that specifies the access permissions on the semaphore set.
sem_otime	Time of last semop() system call.
sem_ctime	Time of last semctl() system call that changed a member of the above structure or of one semaphore belonging to the set.
sem_nsems	Number of semaphores in the set. Each semaphore of the set is referenced as 0 to sem_nsems-1.

The kernel uses **sem** data structure to represent each semaphore.

```
struct sem {
        int             semval;     /* semaphore value */
        int             sempid;     /* PID for last operation */
        unsigned short  semncnt;    /* # processes awaiting semval>curval */
        unsigned short  semzcnt;    /* # processes awaiting semval==0 */
        };
```

---

semget - get a semaphore set identifier

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

semflg can be a bitwise OR of the following: IPC_CREAT | IPC_EXCL | mode

The only valid mode values are:

SEM_R	Read permission for owner / user
SEM_A	Write permission for owner / user
SEM_R » 3	Read permission for group
SEM_A » 3	Write permission for group
SEM_R » 6	Read permission for others
SEM_A » 6	Write permission for others

Returns: semaphore set identifier on success, -1 on error.

---

semop - semaphore operations

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

The second arguement **struct sembuf** has the structure as given below.

```
struct sembuf {
        unsigned short  sem_num;    /* semaphore number */
        short           sem_op;     /* semaphore operation */
        short           sem_flg;    /* operation flags */
        };
```

The possible values for sem_op are:

+ve	added to semval (resource released)
0	wait-for-zero operation
-ve	wait-for semval > curval (resource acquired)

Returns: 0 on success, -1 on error.

---

semctl - semaphore control operations
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, union semun arg);

The calling program must define this union as follows:

```
union semun {
    int               val;    /* Value for SETVAL */
    struct semid_ds   *buf;   /* Buffer for IPC_STAT, IPC_SET */
    unsigned short    *array; /* Array for GETALL, SETALL */
    struct seminfo    *__buf; /* Buffer for IPC_INFO (Linux-specific) */
    };
```

The valid values for **cmd** are:

IPC_STAT	Copy semaphore information into the **semid_ds** structure pointed to by arg.buf. second arguement is ignored; fourth arguement is not needed.
IPC_SET	Change kernel's **semid_ds** of the semaphore set; second arguement is ignored; fourth arguement is not needed.
IPC_RMID	delete the semaphore set; second arguement is ignored; fourth arguement is not needed.
GETPID	Returns the value of sempid for the n-th semaphore of the set.
GETZCNT	Returns the value of semzcnt for the n-th semaphore of the set.
GETNCNT	Returns the value of semncnt for the n-th semaphore of the set.
SETALL	Set semval for all semaphores of the set using arg.array; second arguement is ignored;
GETALL	Return semval (i.e., the current value) for all semaphores of the set into arg.array. second arguement is ignored; fourth arguement is not needed.
SETVAL	Set the value of semval to arg.val for the n-th semaphore of the semaphore set.
GETVAL	Returns the value of semval for the n-th semaphore of the set.

Returns: return value in case of success depends on the **cmd**. They are:

GETNCNT	the value of semncnt.
GETPID	the value of sempid.
GETVAL	the value of semval.
GETZCNT	the value of semzcnt.
SEM_STAT	the identifier of the semaphore set specified through semid.
All others	0

returns -1 on error for all **cmd**.

Shared Memory

Kernel Data Structure for Shared Memory

The kernel uses the **shmid_ds** data structure to represent the shared memory segments.

```
struct shmid_ds {
        struct ipc_perm   shm_perm;
        size_t            shm_segsz;   /* size of segment */
        pid_t             shm_cpid;    /* PID of creator */
        pid_t             shm_lpid;    /* PID, last operation */
        shmatt_t          shm_nattch;  /* no. of current attaches */
        time_t            shm_atime;   /* time of last attach */
        time_t            shm_dtime;   /* time of last detach */
        time_t            shm_ctime;   /* time of last change */
        };
```

---

shmget - allocates a shared memory segment

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

semflg can be a bitwise OR of the following: IPC_CREAT | IPC_EXCL | mode
All the mode flags except the execute flags of the open() system call can be used.

Returns: shared memory identifier on success, -1 on error.

---

shmctl - shared memory control

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Valid values for **cmd** are:

IPC_STAT	Copy the **shmid_ds** of the shared memory segment into buf.
IPC_SET	Set some members (fields) of the **shmid_ds** structure through buf.
IPC_RMID	Mark the segment to be destroyed.

Returns: 0 on success, -1 on error.

---

shmat, shmdt - shared memory operations

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

shmflg can be either zero or SHM_RDONLY; With SHM_RDONLY the shared memory is attached in read only mode.

Returns: shmat(): address of the attached segment on success, -1 on error.
shmdt(): 0 on success, -1 on error.

Signals

The Linux kernel supports the following signals.

Signal	**Description**	**Default Action**
SIGABRT	Sent by abort( )	Terminate with core dump
SIGALRM	Sent by alarm( )	Terminate
SIGBUS	Hardware or alignment error	Terminate with core dump
SIGCHLD	Child has terminated	Ignored
SIGCONT	Process has continued after being stopped	Ignored
SIGFPE	Arithmetic exception	Terminate with core dump
SIGHUP	Process's controlling terminal was closed (most frequently, the user logged out)	Terminate
SIGILL	Process tried to execute an illegal instruction	Terminate with core dump
SIGINT	User generated the interrupt character (Ctrl-C)	Terminate
SIGIO	Asynchronous I/O event	Terminate
SIGKILL	Uncatchable process termination	Terminate
SIGPIPE	Process wrote to a pipe but there are no readers	Terminate
SIGPROF	Profiling timer expired	Terminate
SIGPWR	Power failure	Terminate
SIGQUIT	User generated the quit character (Ctrl-\)	Terminate with core dump
SIGSEGV	Memory access violation	Terminate with core dump
SIGSTOP	Suspends execution of the process	Stop

Signal	Description	Default Action
SIGSYS	Process tried to execute an invalid system call	Terminate with core dump
SIGTERM	Catchable process termination	Terminate
SIGTRAP	Break point encountered	Terminate with core dump
SIGTSTP	User generated the suspend character (Ctrl-Z)	Stop
SIGTTIN	Background process read from controlling terminal	Stop
SIGTTOU	Background process wrote to controlling terminal	Stop
SIGURG	Urgent I/O pending	Ignored
SIGUSR1	Process-defined signal	Terminate
SIGUSR2	Process-defined signal	Terminate
SIGVTALRM	Generated by setitimer( ) when called with the ITIMER_VIRTUAL flag	Terminate
SIGWINCH	Size of controlling terminal window changed	Ignored
SIGXCPU	Processor resource limits were exceeded	Terminate with core dump
SIGXFSZ	File resource limits were exceeded	Terminate with core dump

alarm - set an alarm clock for delivery of a signal
#include <unistd.h>
unsigned int alarm(unsigned int seconds);

Returns: Remaining time of the pending alarm, if any;
0 if there are no pending alarms.

---

pause - wait for signal
#include <unistd.h>
int pause(void);

Returns: Blocks until a signal is caught; returns -1 when a signal is caught.

---

kill - send signal to a process
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);

PID	Target Process(es)
+pid	process with the ID specified by pid.
0	every process in the process group of the calling process.
-1	every process for which the calling process has permission to send signals, except for process 1 (init)
-pid	every process in the process group whose ID is -pid

Returns: 0 on success, -1 on error.

---

raise - send a signal to the caller

```
#include <signal.h>
int raise(int sig);
```

Returns: 0 on success, non-zero on error.

---

signal - ANSI C signal handling

```
#include <signal.h>
void ( *signal(int signum, void (*handler)(int)) ) (int);
                              (OR)
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

signal( ) sets the disposition of the signal signum to handler, which is either

SIG_IGN	Ignore the signal
SIG_DFL	Set to default
handler	provide the signal handler function

The signals SIGKILL and SIGSTOP cannot be caught or ignored.

Returns: previous value of the signal handler on success, SIG_ERR on error.

---

sigaction - examine and change a signal action

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

The kernel defines the **sigaction** as follows:

```
struct sigaction {
        void        (*sa_handler)(int);
        void        (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t    sa_mask;
        int         sa_flags;
        void        (*sa_restorer)(void);                    /*obsolete; do not use */
        };
```

On some architectures a union is involved: do not assign to both sa_handler and sa_sigaction.

sa_handler( ) sets the disposition of the signal signum to handler, which is either

SIG_IGN	Ignore the signal
SIG_DFL	Set to default
handler	provide the signal handler function

sa_flags specifies a set of flags which modify the behavior of the signal. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP	If signum is SIGCHLD, do not receive notification when child processes stop.
SA_NODEFER	Do not prevent the signal from being received from within its own signal handler.
SA_RESETHAND (one-time handler)	Restore the signal action to the default state once the signal handler has been called.
SA_RESTART	Restart certain system calls across signals.
SA_SIGINFO	sa_sigaction should be set instead of sa_handler.

The simplified siginfo_t argument to sa_sigaction is a struct with the following elements:

```
struct siginfo_t {
    int     si_signo;   /* Signal number */
    int     si_errno;   /* An errno value */
    int     si_code;    /* Signal code */
    pid_t   si_pid;     /* Sending process ID */
    uid_t   si_uid;     /* Real user ID of sending process */
    int     si_status;  /* Exit value or signal */
    void    *si_addr;   /* Memory location which caused fault */
    long    si_band;    /* Band event for SIGPOLL */
    };
```

Returns: 0 on success, -1 on error.

Sockets

socket - create an endpoint for communication

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

The valid domains are:

Domain	**Purpose**	**Man page**
AF_UNIX, AF_LOCAL	Local communication	unix
AF_INET	IPv4 Internet protocols	ip

The socket types are defined as follows.

SOCK_STREAM	Provides sequenced, reliable, two-way, connection-based byte streams. These are full-duplex byte streams, similar to pipes.)
SOCK_DGRAM	Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

Addtional options that can be bitwise ORed to the type arguement are:

SOCK_NONBLOCK	Set the O_NONBLOCK file status flag on the new open file description.
SOCK_CLOEXEC	Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor.

Set the protocol arguement to 0 for default protocol.

Returns: new socket file descriptor on success, -1 on error.

---

connect - initiate a connection on a socket

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

The sockaddr structure is defined as follows:

```
struct sockaddr {
    sa_family_t   sa_family;     /* address family */
    char          sa_data[14];   /*variable length address */
    ...
    };
```

Depending on the address family, either **sockaddr_un** or **sockaddr_in** are typecast to **sockaddr**. The **sockaddr_un** or **sockaddr_in** structures are defined as:

```
struct sockaddr_un {
    sa_family_t   sun_family;      /* AF_UNIX */
    char          sun_path[108];   /* path name */
    }
struct sockaddr_in {
    sa_family_t      sin_family;   /* address family */
    in_port_t        sin_port;     /* port number */
    struct in_addr   sin_addr;     /* IPv4 address */
    unsigned char    sin_zero[8];  /* filler */
    };
struct in_addr {
    in_addr_t   s_addr;   /* IPv4 address */
    };
```

sa_family_t is a typedef of unsigned integer type (32-bits, typically); in_addr_t is a typedef of uint32_t; in_port_t is a typedef of uint16_t.

Returns: 0 on success, -1 on error.

---

bind - bind a name to a socket
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

Returns: 0 on success, -1 on error.

---

listen - listen for connections on a socket
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockfd, int backlog);

Returns: 0 on success, -1 on error.

---

accept - accept a connection on a socket
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

#define _GNU_SOURCE
#include <sys/socket.h>
int accept4(int sockfd, struct sockaddr *addr, socklen_t *addrlen, int flags);

NOTE: socklen_t is usually typedef of "int".

When addr is NULL, nothing is filled in; in this case, addrlen is not used, and should also be NULL.

If flags is 0, then accept4() is the same as accept(). The following values can be bitwise ORed in flags to obtain different behavior:

SOCK_NONBLOCK	Set the O_NONBLOCK file status flag on the new open file description.
SOCK_CLOEXEC	Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor.

Returns: a descriptor for the accepted socket is returned on success, -1 on error.

---

send, **sendto** - send a message on a socket
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
const struct sockaddr *dest_addr, socklen_t addrlen);

send(sockfd, buf, len, flags) ↔ sendto(sockfd, buf, len, flags, NULL, 0)

Returns: number of characters sent on success, -1 on error.

---

recv, **recvfrom** - receive a message from a socket
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
struct sockaddr *src_addr, socklen_t *addrlen);

Returns: number of characters read on success, -1 on error.
0 if the peer has shutdown the socket.

---

shutdown - shut down part of a full-duplex connection
#include <sys/socket.h>
int shutdown(int sockfd, int how);

The possible values for how are:

SHUT_RD	further receptions will be disallowed.
SHUT_WR	further transmissions will be disallowed.
SHUT_RDWR	further receptions and transmissions will be disallowed.

Returns: 0 on success, -1 on error.

---

htonl, **htons**, **ntohl**, **ntohs** - convert values between host and network byte order
#include <arpa/inet.h>
#include <netinet/in.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);

Returns: The calls are always successful.

---

inet_aton, **inet_ntoa**, **inet_addr** - Internet address manipulation routines

```
#define _BSD_SOURCE
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
in_addr_t inet_addr(const char *cp);
```

Avoid the use of inet_addr( ) in favor of inet_aton( ), inet_pton( ), or getaddrinfo( ).

Returns: inet_aton( ): 0 on success, -1 on error.
inet_addr( ): address in network byte order, -1 on error.
inet_ntoa( ): address in dotted decimal notation, NULL on error.

---

gethostbyname, **gethostbyaddr** - get network host entry

```
#include <netdb.h>
extern int h_errno;
struct hostent *gethostbyname(const char *name);

#include <sys/socket.h>          /* for AF_INET */
struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);
```

The hostent structure is defined in <netdb.h> as follows:

```
struct hostent {
        char   *h_name;         /* official name of host */
        char   **h_aliases;     /* alias list */
        int    h_addrtype;      /* host address type */
        int    h_length;        /* length of address */
        char   **h_addr_list;   /* list of addresses */
        };
```

Returns: hostent structure on success, -1 on error (h_errno is set).

---

gethostname, **sethostname** - get/set hostname

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
int gethostname(char *name, size_t len);
int sethostname(const char *name, size_t len);
```

Returns: 0 on success, -1 on error.

---

bzero - write zero-valued bytes
#include <strings.h>
void bzero(void *s, size_t n);

Returns: does not return anything.

---

memset - fill memory with a constant byte
#include <string.h>
void *memset(void *s, int c, size_t n);

Returns: pointer to memory area s (Always successful)

Comaprison of IPC

Name	Description	Scope	Use
File	Data is written to and read from a typical UNIX file. Any number of processes can interoperate.	Local	Sharing large data sets
Pipe	Data is transferred between two processes using dedicated file descriptors. Communication occurs only between a parent and child process.	Local	Simple data sharing, such as producer and consumer.
Named pipe	Data is exchanged between processes via dedicated file descriptors. Communication can occur between any two peer processes on the same host.	Local	Producer and consumer, or command-and-control, as demonstrated with MySQL server and its command-line query utility.
Signal	An interrupt alerts the application to a specific condition.	Local	Cannot transfer data in a signal, so mostly useful for process management.
Shared memory	Information is shared by reading and writing from a common segment of memory.	Local	Cooperative work of any kind, especially if security is required.
Socket	After special setup, data is transferred using common input/output operations.	Local or remote	Network services such as FTP, ssh, and the Apache Web Server

References

Linux Manual Pages

Errors from System Calls

This chapter discusses the possible errors from making system calls. It also gives a detailed explanation about the possible causes of these errors.
All the system calls set a variable called **errno** upon encountering error conditions. The variable **errno** is defined in `<error.h>` library as follows.
`extern int errno;`

The **errno** variable can be read from or written to. But it makes no sense to write to this variable. The standard C library provides **perror** function to generate a meaningful error reporting string. The **perror** prints the error reporting string to `stderr`. The syntax of **perror** is,

#include<stdio.h>

void perror(const char *str);

A preprocessor #define maps to the **errno** value. These #defines can be used to insert obvious error checking mechanisms into the system programs. The list of preprocessor #defines are explained here.

E2BIG Argument list too long

EACCESS Permission denied

EAGAIN Try again

EBADF Bad file number

EBUSY Device or resource busy

ECHILD No child processes

EDOM Math argument outside of domain of function

EEXIT File already exists

EFAULT Bad address

EFBIG File too large

EINTR System call was interrupted

EINVAL Invalid argument

EIO I/O error

EISDIR Is a directory

EMFILE Too many open files

EMLINK Too many links

ENFILE File table overflow

ENODEV No such device

ENOENT No such file or directory

ENOEXEC Exec format error

ENOMEM Out of memory

ENOSPC No space left on device

ENOTDIR Not a directory

ENOTTY Inappropriate I/O control operation

ENXIO No such device or address

EPERM Operation not permitted

EPIPE Broken pipe

ERANGE Result too large

EROFS Read-only filesystem

ESPIPE Invalid seek

ESRCH No such process

ETXTBSY Text file busy

EXDEV Improper link

Index