SILIGURI INSTITUTE OF TECHNOLOGY

LABPRATORY MANUAL

PROGRAMMING WITH PYTHON

SILIGURI INSTITUTE OF TECHNOLOGY

VISON

Siliguri Institute of Technology is To be a recognized institution offering high quality education, opportunities to students to become globally employable Engineers/Professionals in best ranked industries and research organization.

MISSION

To impart quality technical education for holistic development of students who will full fil the needs of the industry/society and be actively engaged in making a successful career in industry/research/higher education in India & abroad

PROGRAM EDUCATIONAL OBJECTIVES (PEO) :

The graduates will be:

- Competent professionals with knowledge of Computer Science & Engineering to pursue variety of careers/higher education.
- Proficient in successfully designing innovative solutions to real life problems that are technically sound, economically viable and socially acceptable.
- Efficient team leaders, effective communicators and capable of working in multi-disciplinary environment following ethical values.
- Capable of adapting to new technologies and constantly upgrade their skills with an attitude towards lifelong learning.

PROGRAM OUTCOMES (PO)

Engineering Graduates will be able to:

- Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

- Problem analysis: Identify, formulate, review research literature, and analyze complexengineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- Design/development of solutions: Design solutions for complex engineering problems anddesign system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- Conduct investigations of complex problems: Use research-based knowledge and researchmethods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
- Modern tool usage: Create, select, and apply appropriate techniques, resources, and modernengineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- The engineer and society: Apply reasoning informed by the contextual knowledge to assesssocietal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
- Environment and sustainability: Understand the impact of the professional engineering solutionsin societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms ofthe engineering practice.
- Individual and team work: Function effectively as an individual, and as a member or leader indiverse teams, and in multidisciplinary settings.
- Communication: Communicate effectively on complex engineering activities with the engineeringcommunity and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- Project management and finance: Demonstrate knowledge and understanding of theengineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadestcontext of technological change.

Programming with Python :

Course Objective

CO1:Student will able to install, set pat variable of Python 2.7 versions and write, test, and debug simple Python programs.
CO2:To implement Python programs with conditionals and loops.
CO3:Use functions for structuring Python programs.
CO4:Represent compound data using Python lists, tuples, dictionaries.
CO5: Importing module ,Read and write data from/to files in Python.

LABORATORY

Maulana Abul Kalam Azad University of Technology,
(Formerly West Bengal University of Technology) West Bengal

Syllabus for B. Tech in Information Technology
(Applicable from the academic session 2018-2019)
Subject Code : PCC-CS 393
Category: Professional Core course
Subject Name : IT Workshop (Sci Lab/MATLAB/Python/R)
Semester : Third L-T-P : 1-0-3 Credit:3

Pre-Requisites: No-prerequisite

Programming with Python

Introduction :

History, Features ,Setting up path, Working with Python, Basic Syntax, Variable and Data Type, Operator

Conditional Operator :

If , if-else, Nested if-else, looping : For, While, Nested loops

Control Statements:

Break, Continue, pass

String Manipulation:

Accessing String Basic Operations, String slices, Function and Methods.

List :

Introduction, Accessing list, Operations, Working with lists, Function and Methods.

Tuple :

Introduction, Accessing tuples, Operations, Working, Functions and Methods.

Dictionary:

Introduction, Accessing values in dictionaries, Working with dictionaries, Properties

Function :

Defining a function, Calling a function, Types of functions, Function Arguments, Anonymous functions, Global and local variables.

Module :

Importing module, Math module, Random module, Packages, Composition, Input-Output

Printing on screen, Reading data from keyboard, Opening and closing file, Reading and writing files, Functions.

Exception Handling:

Exception, Exception Handling, Except clause, Try ? finally clause, User Defined Exceptions.

Experim ent NO.	Topic	Title
1	BASIC	A).Install Python and Set Path variable
		B). Running instructions in Interactive interpreter and a Python Script
		C). Write a program to purposefully raise Indentation Error and Correct it. [Display your name and Department in two separate line]
2	Operator	A). Write a program to compute distance between two points taking input from the user (Pythagorean Theorem)
		B). Write a program add.py that takes 2 numbers as command line arguments and prints its sum.
3	Conditional Statement	A) Write a program to check a given number is even or odd.
		B) Write a program to check a given year is leap year or not.
		C) Write a program to calculate real roots of a quadratic equation.
4	Loop Statement	A) Write a program using a while loop that asks the user for a number, and prints a countdown from that number to zero. [using range() method]
		B) Write a program to calculate the Sum of even Fibonacci numbers below 4 Thousand.
		C) Write a program to calculate GCD of two number.
		D) Write a program to print the following pattern : i) ii) iii) i) | * | | | | * | * | | | * | * | * | ii) | | | * | | |

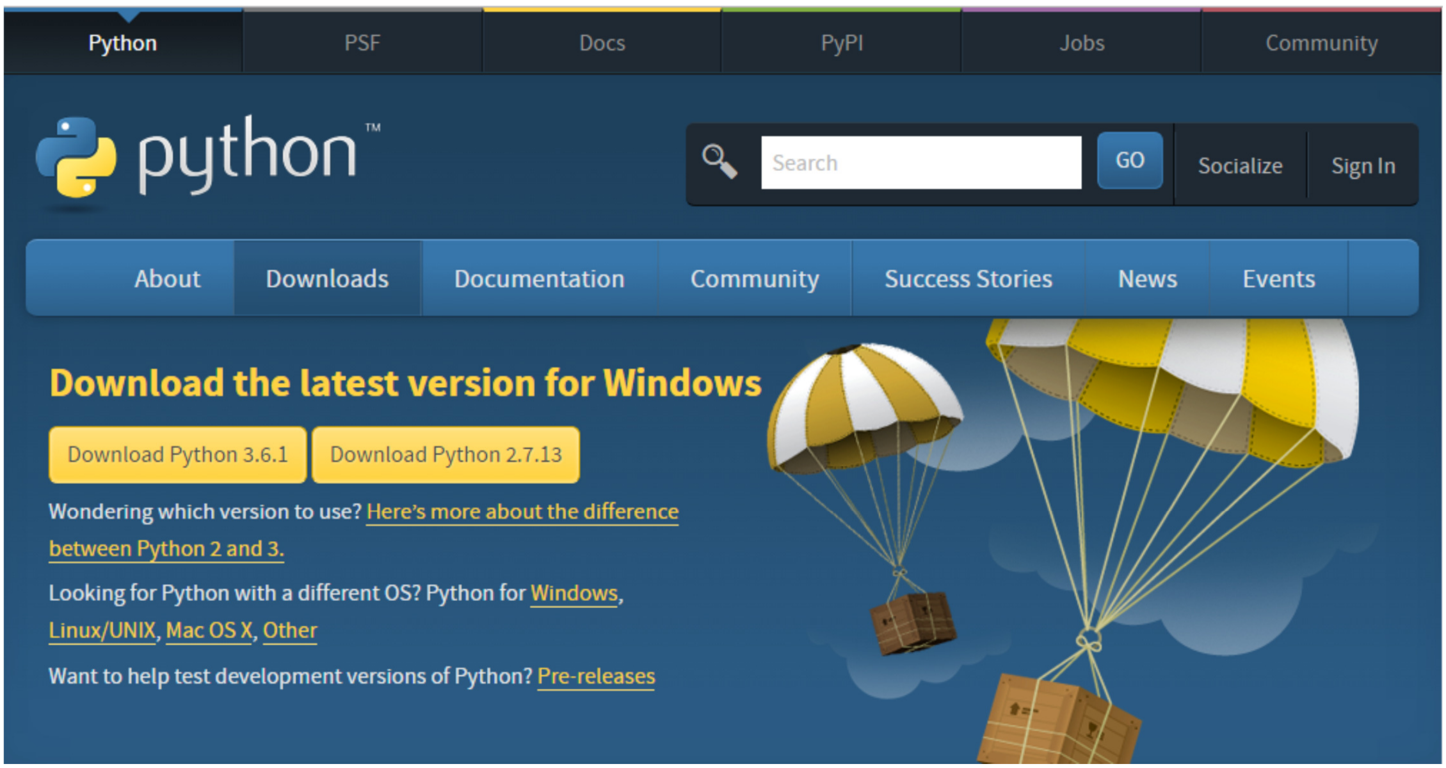
		* * * * * * * * * * * * * *
5	String Operation	A) Write a program to count no of vowel in a string(using in operator)
		B) Write a program to perform following operation on strong: i) The total number of characters in the string ii) The last three characters of the string iii) Print The string backwards direction iv) Print The string in all caps
6	Tuple and Set	A)Write a program to initialize and Display tuple data structure. B) Write a program to initialize and Display two Set data structure and do the following operation : i) union ii)intersection ii)difference
7	List	A) Print the total number of items in the list
		B) Print the list in reverse order.
		C) Remove the first and last items from the list, sort the remaining items, and print the result.
		D) Write a program that generates a list of 20 random numbers between 1 and 100. (a) Print the list. (b) Print the average of the elements in the list. (c) Print the largest and smallest values in the list. (d) Print the second largest and second smallest entries in the list (e) Print how many even numbers are in the list.
		E) Write a program that takes any two lists L and M of the same size and adds their elements together to form a new list N whose elements are sums of the corresponding elements in L and M. For instance, if L=[3,1,4] and M=[1,5,9], then N should equal [4,6,13].
		F)Write a program to perform multiplication of two square matrices
8	Function	A) Write a function called rectangle that takes two integers m and n as arguments and prints out an m n box consisting of asterisks. Shown below is the output of rectangle(2,4). **** ****
		B) Write a function called sum_digits() that is given an integer num and returns the sum of the digits of num.
		c)The digital root of a number n is obtained as follows: Add up the digits n to get a new number. Add up the digits of that to get another new number. Keep doing this until you get a number that has only one digit. That number is the digital root. For example, if n = 45893, we add up the digits to get 4 + 5 + 8 + 9 + 3 = 29. We then add up the digits of 29 to get 2 + 9 = 11. We then add up the digits of 11 to get 1 + 1 = 2. Since 2 has only one digit, 2 is our digital root. Write a function that returns the digital root of an integer n. [Note: there is a shortcut, where the digital root is equal to n mod 9, but do not use that here.]
		C)Write a program to multiply two list using lambda function
		D) Write a program to filter out only odd number from a list.
9	Dictionary	A)Write a Python script to store(ascending and descending order) in to a dictionary by value.
		B)Write a Python script to insert a new key in to a dictionary.
		C)Write a program to take a list of student's (name, age, marks) input from key board. Print average marks and details of highest scorer using dictionary data structure.
10	File	A)Write a program to copy the content of one file in to another file.
		B)Write a program to count the frequency of each word from a file.
11	Module	A)Write a program to display i)Current date and time ii)Current year iii)Month of year iv)Week number of year v)Week day of the week vi)Day of year vii)Day of week
		B)Plot the the roll number and average marks of a list of student in a class(import mathplotlib module)

12	Exception Handling	A)Write a program to tame two number as a input and divide theme and show i) value error ii)zero division error

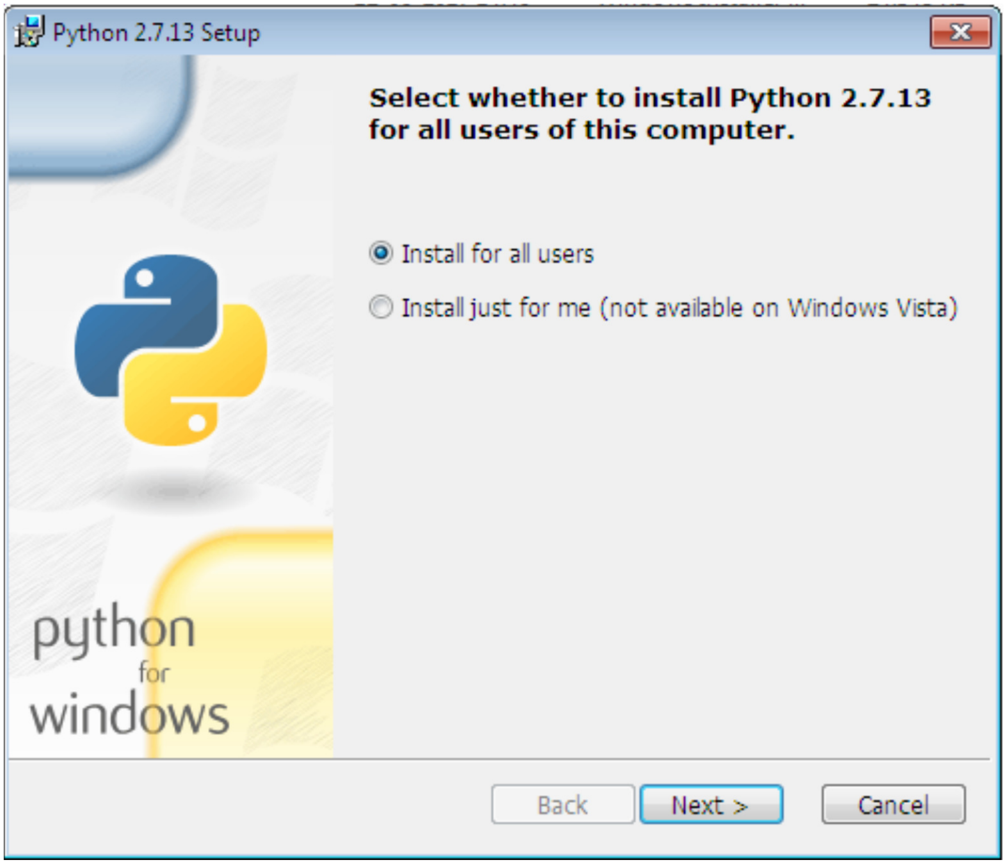
Experiment 1:

Procedure to Install and Run programs in Python:

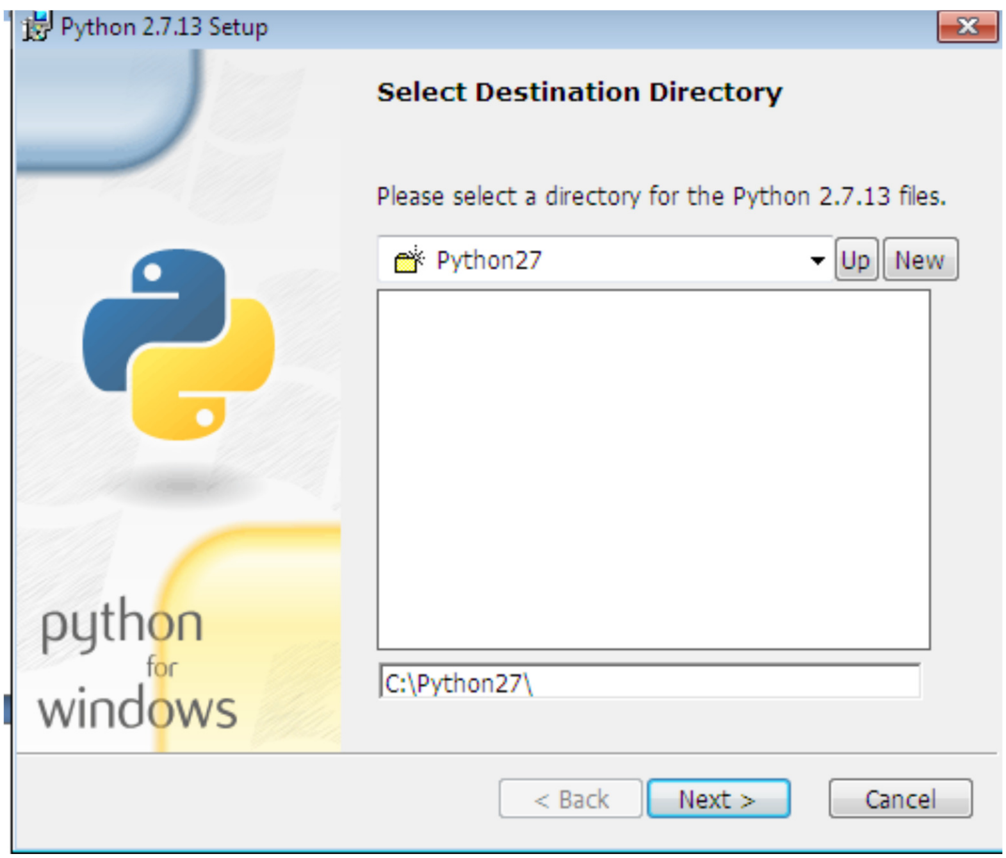
In order to install python, Visit https://www.python.org. When we visit the Python for Windows download page, we will immediately see the division. Right at the top, square and center, the repository asks if you want the latest release of Python 2 or Python 3 (2.7.13 and 3.6.1, respectively) as shown in below Figure.



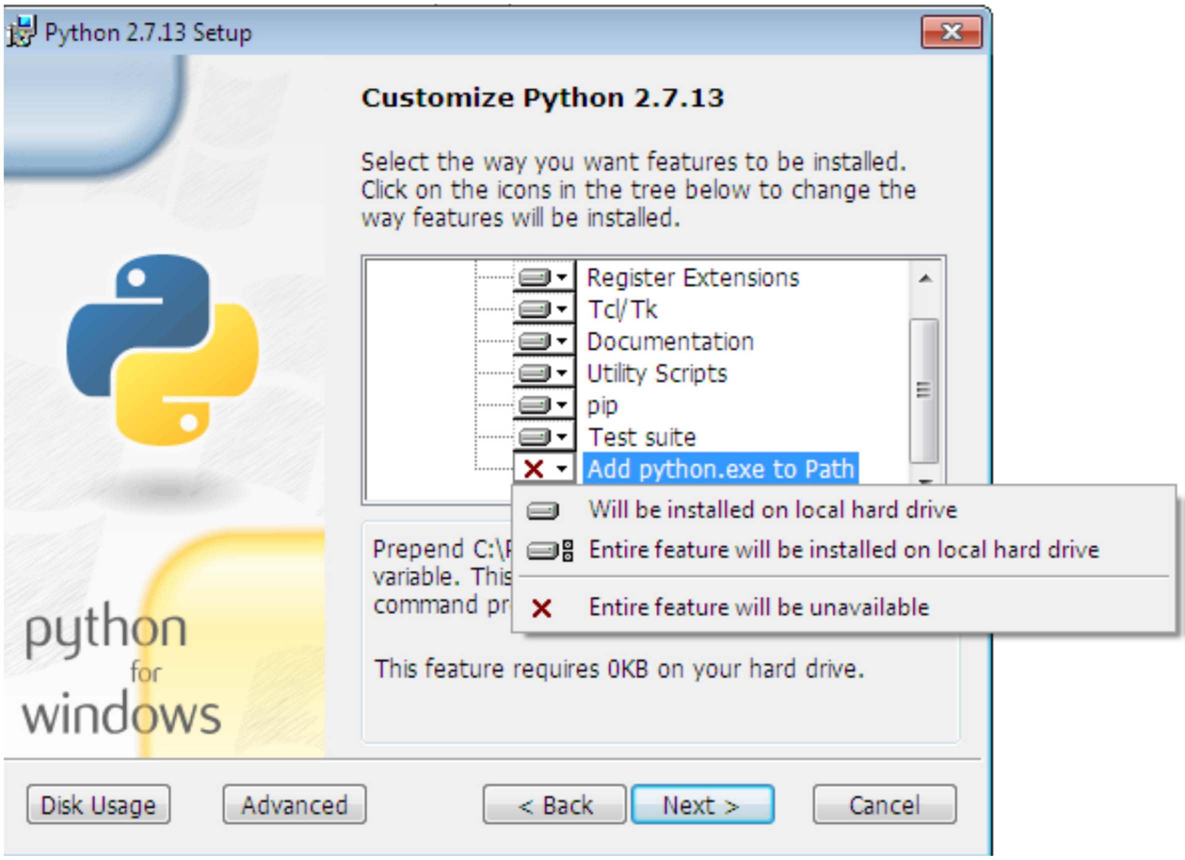
The version we want depends on our end goal. Here we will install Python 2.7.13. Click on Download Python 2.7.13 then python-2.7.13.msi file will be downloaded. Run the installer, then a window will be opened as shown below. Select “Install for all users,” and then click “Next”.



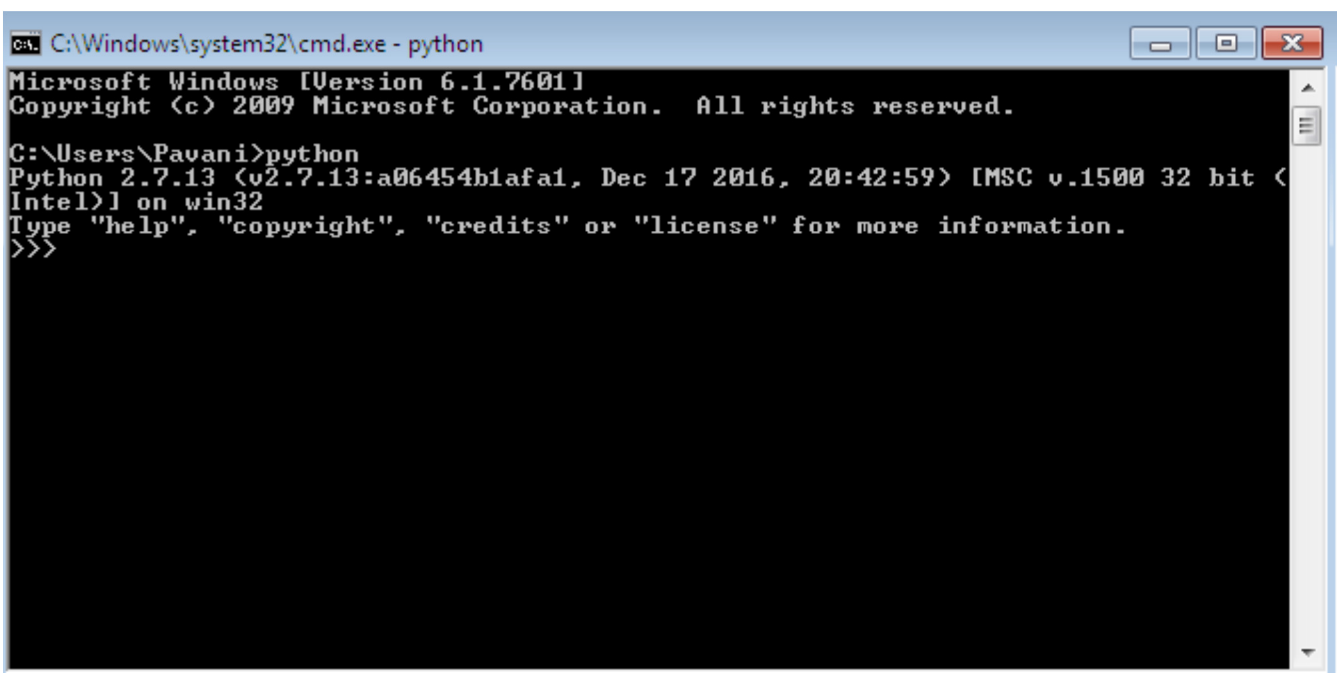
After Clicking on “Next”, a window will be opened as shown below. On the directory selection screen, leave the directory as “Python27” and click “Next”.



After Clicking on “Next”, a window will be opened as shown below. On the customization screen, scroll down, click “Add python.exe to Path,” and then select “Will be installed on local hard drive.” then click “Next.”



We don‟t have to make any more decisions after this point. Just click through the wizard to complete the installation. When the installation is finished, set the variable path. After setting up the path, we can confirm the installation by opening up Command Prompt and type the following command as shown below.



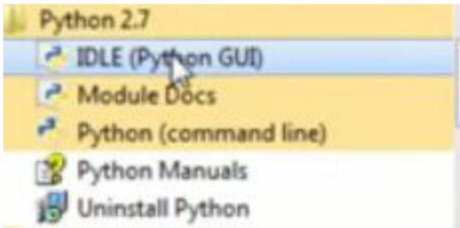
Now, we can say that Python 2.7.13 is installed on our machine.

Different Ways of Invoking Python:

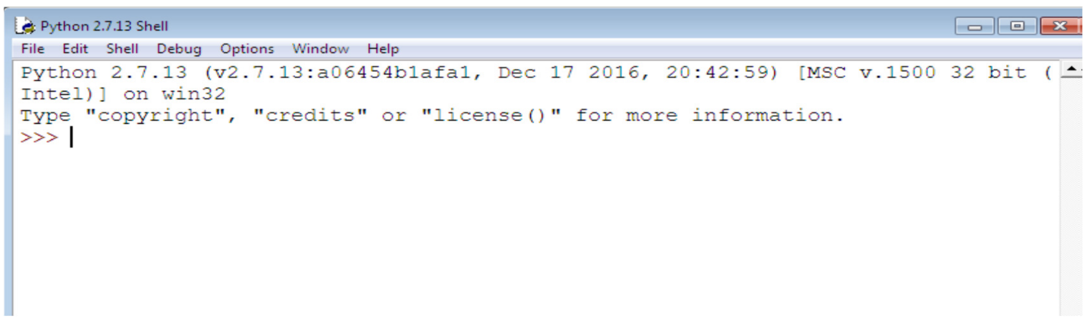
- Python GUI
- Python command line
- Command prompt from windows

Python GUI:

Click on start -> all programs -> python 2.7 -> IDLE(Python GUI).

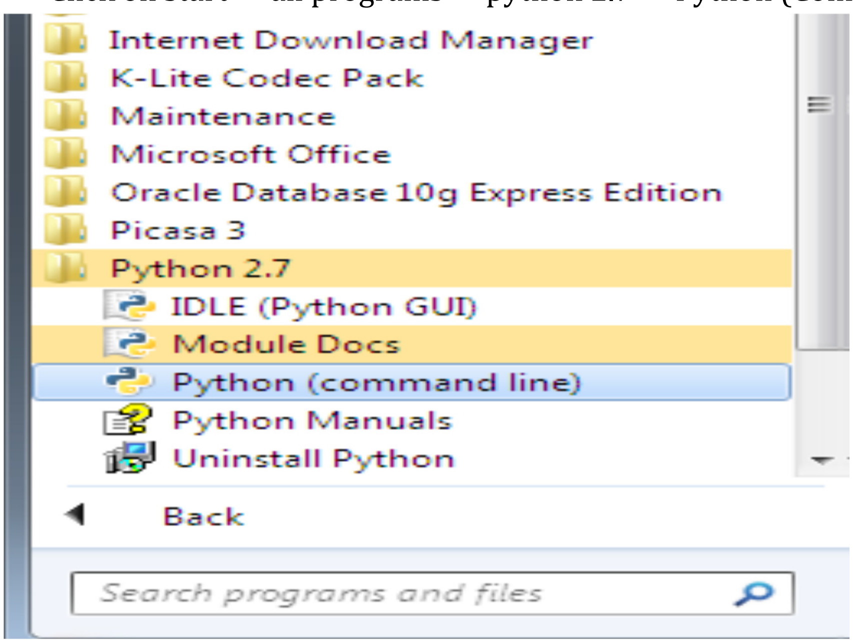


After Clicking on IDLE(Python GUI), a window will be opened as shown below. Python command line: Click on

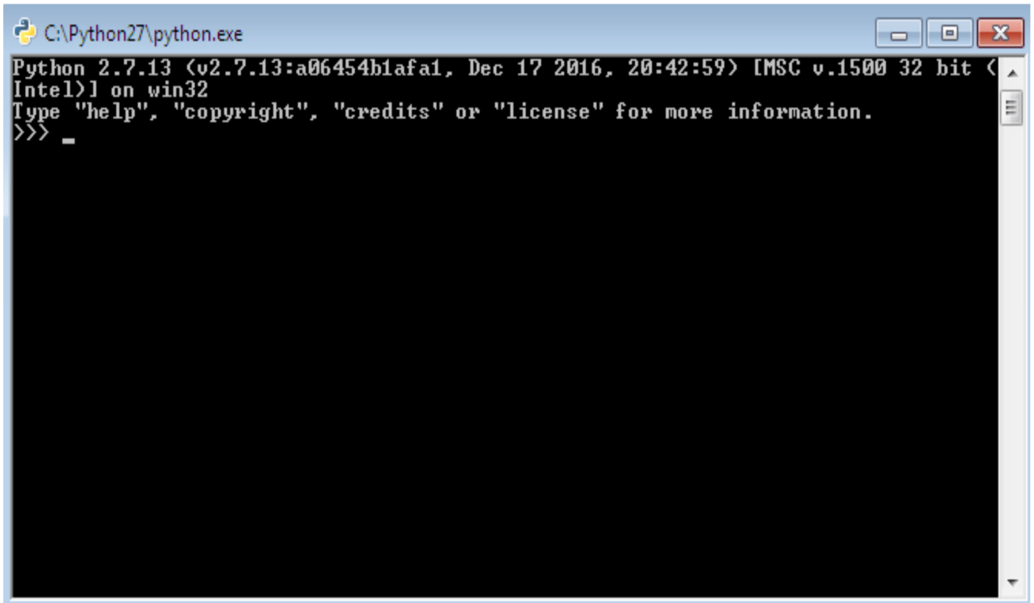


Python command line:

Click on start -> all programs -> python 2.7 -> Python (Command line).



After Clicking on Python (command line), a window will be opened as shown below:

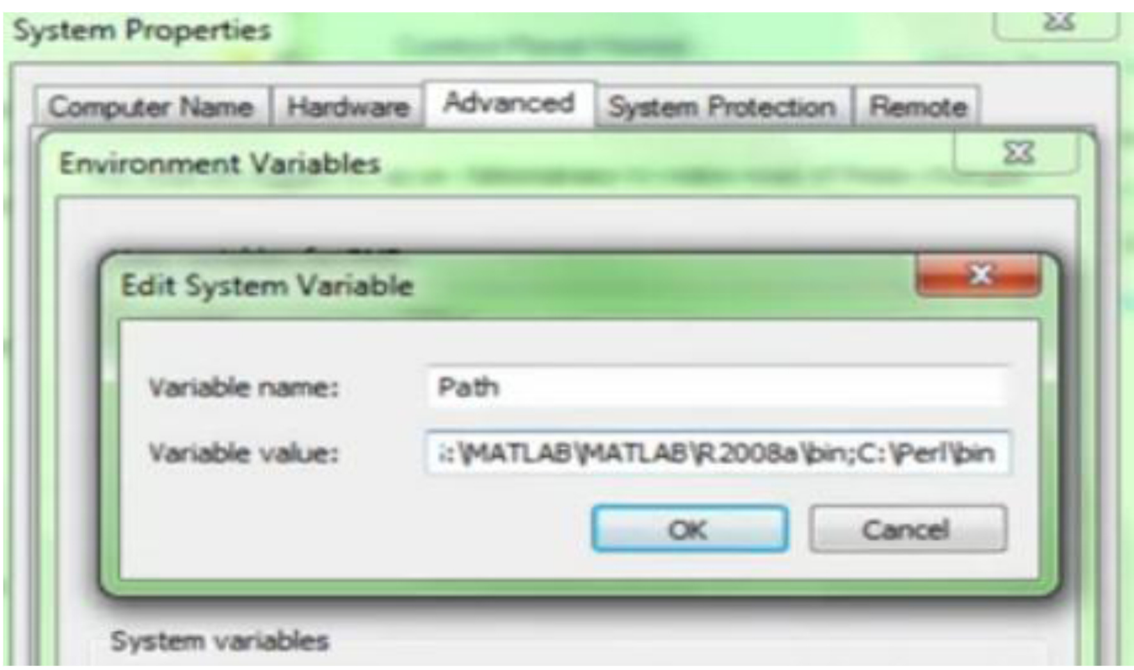


Command prompt from windows:

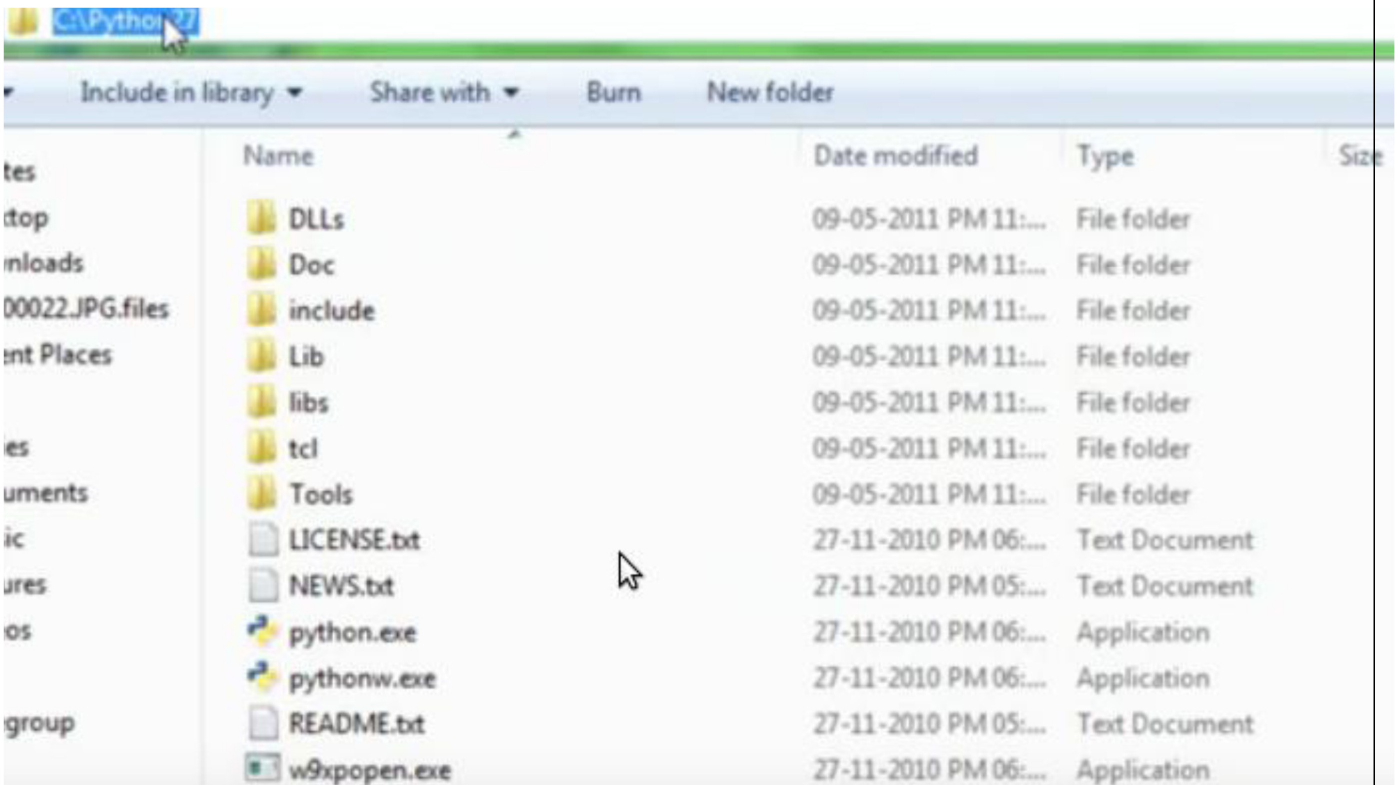
To open Python from Windows command prompt, We need to **set path**. The procedure to set the path is as follows :

Go to My Computer -> right click and open properties, then a window will be opened as shown below:

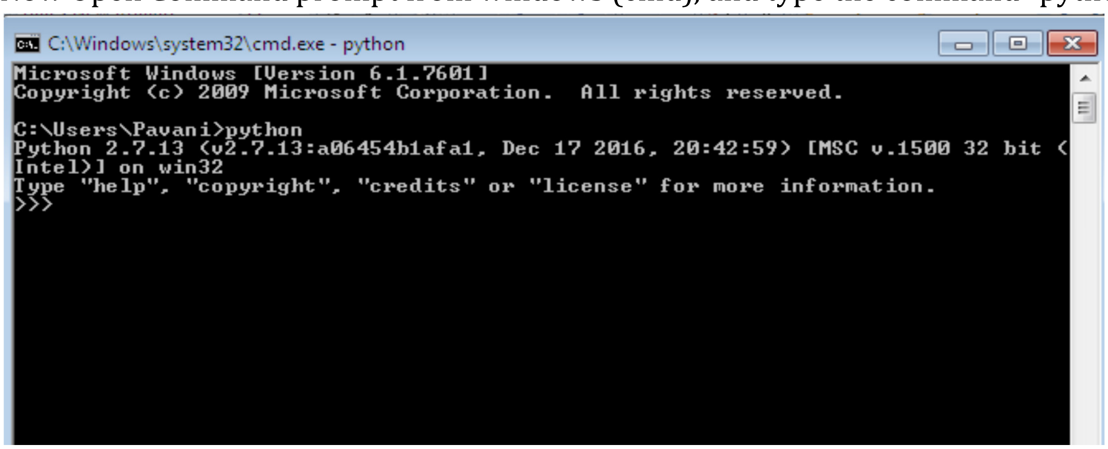
Now, Click on Advanced system settings -> Environmental Variables -> system variables and under system variable, click on Path variable and click on Edit. Then, a window will be opened as follows:



Add python path in variable value and click on **OK** as follows:



Now Open Command prompt from windows (cmd), and type the command "python" as follows:



Experiment : 1(C)

Write a program to purposefully raise Indentation Error and Correct it.

Description:

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.

A code block (body of a function, loop etc.) starts with indentation and ends with the first unintended line. The amount of indentation is depends on our choice, but it must be consistent throughout that block. Generally, Four whitespaces are used for indentation and is preferred over tabs. The enforcement of indentation in Python makes the code look neat and clean. This results into Python programs that look similar and consistent. Incorrect indentation will result into Indentation Error.

Program that shows Indentation Error:

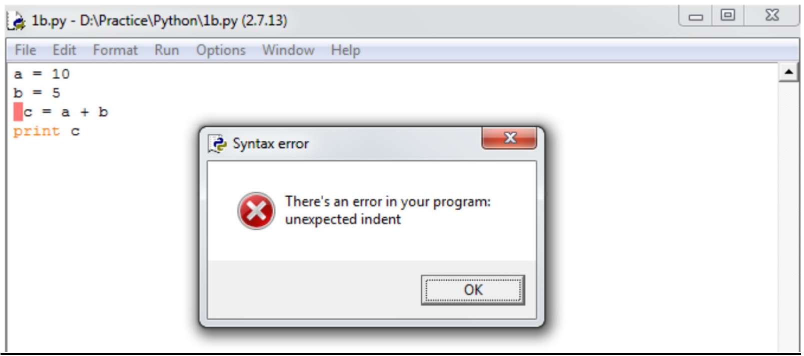
a = 10

b = 5

  c = a + b

 print c

Output:



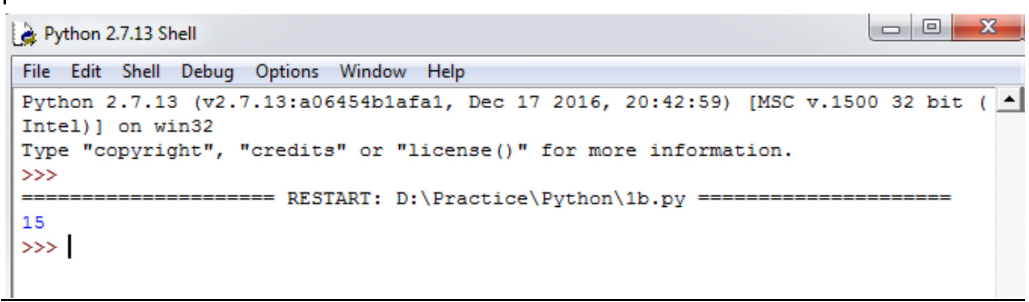
Program without Indentation Error:

a = 10

b = 5

c = a + b

print c



Experiment 2(A) :

Write a program to compute distance between two points taking input from the user (Pythagorean Theorem).

Description: The Pythagorean theorem is the basis for computing distance between two points. Let (x_1,y_1) and (x_2,y_2) be the co-ordinates of points on xy-plane. From Pythagorean theorem, the distance between two points is calculated using the formulae:

Distance D= $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Distance between two point A(x1,y1) and B(x2,y2)

```
import math as m

print(" Enter the Co ordinate of first point")
x1=int(input())
y1=int(input())
```

```
print("Enter the co ordinate of second point")
x2,y2=[int (x) for x in input("Enter x and y seperated by space").split()]
print(x2,y2)
d=m.sqrt((x2-x1)*(x2-x1) + (y2-y1)**2)
print(m.ceil(d))
print(m.floor(d))
```

 # 2_B) Program for Arithmatic operation like addition ,multiplication ,division ....

```
'''
x=int(input("Enter first number"))
y=int(input("enter second number"))
print(" Addition result=",x-y)
print(" Multiplication result=",x*y)
print(" Division result=",x/y)
print(" modular result=",x%y)
print(" integer division result=",x//y)

'''
```

#3_A) Check given number is even or odd...

```
'''

x=int(input("Enter The number"))
if (x%2==0):
 print("The Number",x, "is Even")
else:
 print("The Number",x,"is Odd")

'''
```

#3_B) Check given yuear is leap year or not...

```
'''
x=int(input("Enter The Year"))
if(x%400==0 or x%100!=0 and x%4==0 ):
 print("The year",x,"is leap year")
else:
 print("The Year",x,"Is not leap year")

'''
```

#3_C Check a given charactor is alphabet ,digit or specal char or not

```
ch=input("enter any symbol from keybord")
```

```
print("Ascii value of the Symbol is",ord(ch))
if((ch>='A' and ch<='Z') or (ch>='a' and ch<='z')):
  print("The Symbol",ch,"is alphabet")
elif(ch>='0' and ch<='9'):
  print("The symbol",ch,"is Digit")
else:
  print("The symbol",ch,"Is Special charactor")

# 4_A  print serasse of number and count down of this number .
'''
x=int(input("Enter a number "))
y=x
while(x>0):
   print(x)
   x=x-1
# countdown using range method

for i in range(y,0,-1):
   print(i)


# 4_B  Sum of even fibo nacci number below 4000

a=-1
b=1
s=0
c=a+b
while(c<=4000):
   a=b
   b=c
   c=a+b
   if(c%2==0):
       print(c)
        s=s+c
print("Sum of all even fibo nacci bello 4000 is =",s)


# 4_c GCD of two number

x=int(input("Enter First number"))
y=int(input("enter second number"))
m=x
n=y
while(x!=y):
```

```
  if(x>y):
    x=x-y
  else:
    y=y-x
print(" Gcd of ",m, "and",n,"is=",x)
```

 # 4d_patt_i Print pattaen :

```
x=int(input("Enter no of row"))
for i in range(0,x+1):
  for j in range(0,i+1,1):
    if(i>j):
       print("*",end='')
    else:
       print()
```

 # 4d_patt_ii Print pattaen :

```
x=int(input("Enter no of row"))
for i in range(0,x):
  for k in range(0,x-i+1):
    print(" ",end="")
  for j in range(0,i+1):
    print(" * ",end='')
  print()
```

 # 4d_patt_iii Print pattaen :

```
x=int(input("Enter no of row"))
for i in range(0,x):
  for k in range(0,i+1):
    print(" ",end="")
  for j in range(0,x-i):
    print("*",end='')
  print()
```

'''

 # 5 A write a program to count no of vowel in a srting ...

'''

```
s=input("Enter a string")
s=s.lower()
c=0
for item in s:
  if item in ('a','e','i','o','u'):
    c=c+1
print(" Total vowel = ",c)
```

```
'''

#  5_B write a program to perform the following operation in a srting ...
s=input("Enter a string")
print(" Total no of charactor is=",len(s))
print(" last 3 charactor is=",s[-3:])
print(" Reverse String is = ",s[-1::-1])
print(" All capital of string= ",s.upper())


'''
# 6_A_Initialize and display tuple and set

t=(1,2,"Kritt",'a',1,'d',23.7)
# in set all element are unique do not contain duplicate element(automatic delete)
s={ "kritt",3,5,9,23.0,3}

print(t)
print(s)

'''
# 6_B_Initialize and display two  set and do the following operation

# initialize empty set
s1=set()
s2=set()
while (1):

  item=input("Enter set item for set 1: ")
  s1.add(item)
  print(" Press 1  for continue and For quit press 0 ")
  ch=int(input())
  if ch==0:
    break
  else:
    continue

print(" item of first  set = :",s1)
while (1):
  item=input("Enter set item for set 2: ")
  s2.add(item)
  print("Press 1  for continue and For quit press 0")
  ch=int(input())
  if ch==0:
```

```
    break
  else:
    continue

print("Total item of First SET = ",s1,"Total item of Second  set = :",s2)

print("Union of SET 1 and SET 2 is=: ",s1.union(s2))
#print(s1|s2)
print("Intersection of SET 1 and SET 2 is =: ",s1.intersection(s2))
#print(s1&s2)
print("Difference of SET 1 and SET 2 is =: ",s1.difference(s2))
#print(s1-s2)

# list Operation ....
# initialize a list
'''
L=[1,2,8,4,3]

#insert a item in to list

item=int(input(" Enter item for insert .."))
L.append(item)
print(L)

#count no of item in the list
c=len(L)
print("Total no of element is =",c)
print(" Print in reverse order",L[-1::-1])
L.remove(L[0])
print(L)
L.pop()
print("After remove last element",L)
#print decending order
L.sort(reverse=True)
print(" Sotred order",L)
```

 # Program 7_D generate 20 random number between 1 to 100

```
import random
L=[]
for i in range(5):
  x=random.randint(1,100)
  L.append(x)
print(L)
```

```
m=max(L)
n=min(L)
s=sum(L)
a=s/len(L)
print(" max=",m,"min=",n,"sum=",s,"Average=",a)
L.sort()
print(L)
print("Second Highest =",L[-2])
print("Second lowest=",L[1])
c=[i for i in L if i%2==0]
print("No of Even =",len(c))
```

#Program 7_E Take two list and add them store in to third list

```
import random

L=[]
M=[]

for i in range(5):
  x=random.randint(1,100)
  L.append(x)
print(L)
for i in range(5):
  x=random.randint(1,100)
  M.append(x)
print(M)

N=[ 0 for i in range(5)]
for j in range(5) :
  N[j]=L[j]+M[j]
print("Addition od Two list =",N)
'''
# multiply two matrox...
M=[[1,2,3],[1,1,2],[2,2,1]]
N=[[1,2,1],[1,1,2],[1,2,1]]
R=[]
for i in range(3):
  l=[]
  for j in range(3):
    l.append(0)
    R.append(l)
print(M,N,R)
```

```
for i in range(3):
  for j in range(3):
    for k in range(3):
        R[i][j]=R[i][j]+M[i][k]*N[k][j]
print(" Matrix Result ")
for i in R:
  print(i)

'''
```

 # 8_A program print rectangle with * using function..

```
def rect(r,c):
  for i in range(r):
    for j in range(c):
      print("*",end='')
                                                                          print()


print("Enter no of row")
r=int(input())
print("Enter no of collumn")
c=int(input())
rect(r,c)
```

 # 8_B program print some of digit using function..

```
def sum_dig(n):
  s=0
  while(n>0):
    r=n%10
    s=s+r
    n=n//10
  return(s)

print(" Enter the number ")
n=int(input())

print("SUM of digit if a number",n," is =",sum_dig(n))
```

8_C program finding digital root using function.

```
def sum_dig(n):
  s=0
  while(n>0):
```

```
    r=n%10
    s=s+r
    n=n//10
  return(s)

print(" Enter the number ")
n=int(input())
sod=sum_dig(n)
while(sod>9):
  sod=sum_dig(sod)
  print(sod)
print(" Digital root is =",sod)
'''
```

 # 8_D program add two list using lambda function (both list must be in same in size.

```
L1=[1,3,2,5]
L2=[3,5,1,2]
res=map(lambda x,y:x*y,L1,L2)

print(list(res))

# lambda with filter function
L=[2,1,4,5,8,9,23.10,3]
res1=list(filter(lambda x :(x%2==1),L))
print("Odd number in list  ")
print(res1)
```

SILIGURI INSTITUTE OF TECHNOLOGY

COMPUTER SCIENCE
AND
ENGINEERING DEPARTMENT

OPERATING SYSTEM

LABORATORY MANUAL

LM Rev No: 01

Contents

Chapter 1 - Shell Programming

1. **Shell Programming**

The shell has similarities to the DOS command processor Command.com (actually Dos was design as a poor copy of UNIX shell), it's actually much more powerful, really a programming language in its own right.

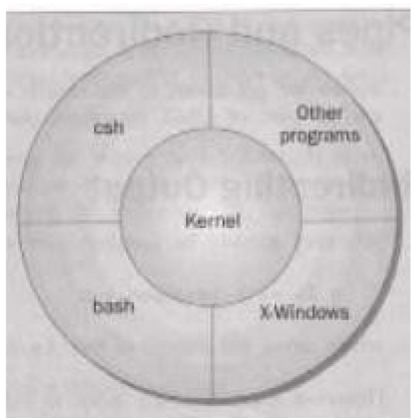
A shell is always available on even the most basic UNIX installation. You have to go through the shell to get other programs to run. You can write programs using the shell. You use the shell to administrate your UNIX system. For example:

ls -al | more

is a short shell program to get a long listing of the present directory and route the output through the more command.

What is a Shell?

A **shell** is a program that acts as the interface between you and the UNIX system, allowing you to enter commands for the operating system to execute.



Here are some common shells.

Shell Name	A Bit of History
sh (Bourne)	The original shell.
csh, **tcsh** and **zsh**	The C shell, created by Bill Joy of Berkeley UNIX fame. Probably the second most popular shell after **bash**.
ksh, **pdksh**	The Korn shell and its public domain cousin. Written by David Korn.
bash	The Linux staple, from the GNU project. **bash**, or Bourne Again Shell, has the advantage that the source code is available and even if it's not currently running on your UNIX system, it has probably been ported to it.
rc	More C than **csh**. Also from the GNU project.

Pipes and Redirection

Pipes connect processes together. The input and output of UNIX programs can be redirected.

Redirecting Output

The > operator is used to redirect output of a program. For example:

```
ls -l > lsoutput.txt
```

redirects the output of the list command from the screen to the file lsoutput.txt.

To append to a file, use the >> operator.

```
ps >> lsoutput.txt
```

Redirecting Input

You redirect input by using the < operator. For example:

```
more < killout.txt
```

Pipes

We can connect processes together using the pipe operator (|). For example, the following program means run the ps program, sort its output, and save it in the file pssort.out

```
ps | sort > pssort.out
```

The sort command will sort the list of words in a textfile into alphbetical order according to the ASCII code set character order.

The Shell as a Programming Language

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

Interactive Programs

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

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

Creating a Script

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

```
#!/bin/sh

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

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

exit 0
```

Note: commands start with a #.

The line

    #!/bin/sh

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

The command

    exit 0

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

Making a Script Executable

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

    /bin/sh first.sh

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

    chmod +x first.sh
    first.sh

Actually, you may need to type:

    ./first.sh

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

Shell Syntax

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

Variables

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

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

Quoting

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

```
#!/bin/sh

myvar="Hi there"

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

echo Enter some text
read myvar

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

make sure to make it executable by typing the command:

< chmod a+x quot.sh

The results of executing the file is:

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

How It Works

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

Environment Variables

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

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

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

Parameter Variables

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

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

The following shows the difference between using the variable $* and $@

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

notice that the first line of the above has a space between the firsr ' and the second '.

Now try your hand at typing a shell script

Carefully type the following into a file called: try_variables

```
#!/bin/sh

salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"

echo "Please enter a new greeting"
read salutation

echo $salutation
echo "The script is now complete"
exit 0
```

make sure to make it executable by typing the command:

    < chmod a+x try_variables
Execute the file with parameters by typing:
    try_variables foo bar baz

The results of executing the file is:

```
$ try_variables foo bar baz
Hello
The program ./try_variables is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The users home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
$
```

How It Works

It creates the variable salutation, displays its value, and some parameter variables.

Conditions

All programming languages have the ability to test conditions and perform different actions based on those conditions. A shell script can test the exit code of any command.

The test, or [] Command

Here is how to check for the existance of the file fred.c using the test and using the [] command.

```
if test -f fred.c
then
...
fi
```

We can also write it like this:

```
if [ -f fred.c ]
then
...
fi
```

You can even place the then on the same line as the if, if youu add a semicolon before the word then.

```
if [ -f fred.c ]; then
...
fi
```

Here are the conditon types that can be used with the test command. There are string comparison.

String Comparison	Result
`string`	True if the string is not an empty string.
`string1 = string2`	True if the strings are the same.
`string1 != string2`	True if the strings are not equal.
`-n string`	True if the string is not **null**.
`-z string`	True if the string is **null** (an empty string).

There are arithmetic comparison.

Arithmetic Comparison	Result
`expression1 -eq expression2`	True if the expressions are equal.
`expression1 -ne expression2`	True if the expressions are not equal.
`expression1 -gt expression2`	True if **expression1** is greater than **expression2**.
`expression1 -ge expression2`	True if **expression1** is greater than or equal to **expression2**.
`expression1 -lt expression2`	True if **expression1** is less than **expression2**.
`expression1 -le expression2`	True if **expression1** is less than or equal to **expression2**.
`! expression`	The ! negates the expression and returns **true** if the expression is **false**, and vice versa.

There are file conditions.

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

Control Structures

The shell has a set of control structures.

if

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

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

elif

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

```
#!/bin/sh

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

if [ $timeofday = "yes" ]
then
        echo "Good morning"
elif [ $timeofday = "no" ]; then
        echo "Good afternoon"
else
        echo "Sorry, $timeofday not recognized. Enter yes or no"
        exit 1
fi
exit 0
```

How It Works

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

A Problem with Variables

If a variable is set to null, the statement

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

looks like

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

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

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

for

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

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

Try out the following script:

```
#!/bin/sh

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

When executed, the output should be:

```
bar
fud
43
```

How It Works

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

How It Works

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

```
#!/bin/sh

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

```
done
```

while

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

```
while condition do
        statements
done
```

Here is a whil loop that loops 20 times.

```
#!/bin/sh

foo=1
while [ "$foo" -le 20 ]
do
        echo "Here we go again"
        foo=$(($foo+1))
done
exit 0
```

How It Works

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

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

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

until

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

```
until condition
do
        statements
done
```

Here is a script using until.

```
#!/bin/sh

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

# now ring the bell and announce the expected user.

echo -e \\a
```

```
echo "**** $1 has just loogged in ****"

exit 0
```

case

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

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

Here is a sample script using a case statement:

```
#!/bin/sh

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

case "$timeofday" in
    "yes") echo "Good Morning";;
    "no" ) echo "Good Afternoon";;
    "y" ) echo "Good Morning";;
    "n" ) echo "Good Afternoon";;
    * ) echo "Soory, answer not recognized";;
esac
exit 0
```

How It Works

The value in the varaible timeofday is compared to various strings. When a match is made, the associated echo command is executed.

Here is a case where multiple strings are tested at a time, to do the some action.

```
case "$timeofday" in
    "yes" | "y" | "yes" | "YES" ) echo "good Morning";;
    "n"* | "N"* ) < echo "Good Afternoon";;
    * ) < echo "Sorry, answer not recognized";;
esac
```

How It Works

The above has sever strings tested for each possible statement.

Here is a case statement that executes multiple statements for each case.

```
case "$timeofday" in
	"yes" | "y" | "Yes" | "YES" )
		echo "Good Morning"
		echo "Up bright and early this morning"
		;;
	[nN]*)
		echo "Good Afternoon"
		;;
	*)
		echo "Sorry, answer not recognized"
		echo "Please answer yes or noo"
		exit 1
		;;
esac
```

How It Works

When a match is found to the variable value of timeofday, all the statements up to the ;; are executed.

Lists

To test for multiple conditions, we can use nested if or if/elif.

The AND List

Alolows us to execute a series of command. Each command is only execute if the previous commands have succeeded. An AND list joins conditions by using &&.

```
statement1 && statement2 && statement3 && ...
```

Her is a sample AND list:

```
#!/bin/sh

touch fine_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo " there"
then
	echo "in if"
else
	echo "in else"
fi
exit 0
```

How It Works

The touch command creates an empty file. the rm come remove a file. So, before we start, file_one exists and file_two doesn't. The AND list finds the file_one, and echos the word hello, but it doesn't find the file file_two. Therefore the overall if fails and the else clause is executed.

The OR List

The OR list construct allows us to execute a series of commands until one succeeds!

```
statement1 || statement2 || statement3 || ...
```

Here is a sample Or list

```
rm -f file_one

if [ -f file_one ] || echo "hello" || echo " there"
then
	echo "in if"
else
	echo "in else"
fi
exit 0
```

How It Works

The above script removes the file file_one, then test for and fails to find the file_one, but does successfully echo hello. It then executes the then statement echoing in if.

Statement Blocks

Multiple statements can be placed inside of { } to make a statement block.

Functions

You can define functions inthe shell. The syntax is:

```
function_name () {
	statements
}
```

Here is a sample function and its execution.

```
#!/bin/sh

foo() {
	echo "Function foo is executing"
}

echo "script starting"
foo
echo "script ended"

exit 0
```

How It Works

When the above script runs, it defines the funcion foo, then script echos script starting, then it runs the functions foo which echos Function foo is executing, then it echo script ended.

Here is another sample script with a function in it. Save it as my_name

```
#!/bin/sh

yes_or_no() {
	echo "Parameters are $*"
	while true
	do
		echo -n "Enter yes or no"
		read x
		case "$x" in
			y | yes ) return 0;;
			n | no ) return 1;;
			* ) echo "Answer yes or no"
		esac
	done
}

echo "Original parameters are $*"

if yes_or_no "IS your naem $1"
then
	echo "Hi $1"
else
	echo "Never mind"
fi
exit 0
```

How It Works

When my_name is execute with the statement:

```
my_name Rick and Neil
```

. gives the output of:

```
Original parameters are Rick and Neil
Parameters are Is your name Rick
Enter yes or no
no
Never mind
```

Commands

You can execute normal command and built-in commands from a shell script. Built-in commands are defined and only run inside of the script.

break

It is used to escape from an enclosing for, while or until loop before the controlling condition has been met.

The : Command

The colon command is a null command. It can be used for an alias for true..

continue

The continue command makes the enclosing for, while, or until loop continue at the next iteration.

The . Command

The dot command executes the command in the current shell:

```
. shell_script
```

.

echo

The echo command simply outputs a string to the standard output device followed by a newline character.

eval

The eval command evaluates arguments and give s the results.

exec

The exec command can replace the current shell with a different program. It can also modify the current file descriptors.

exit n

The exit command causes the script to exit with exit code n. An exit code of 0 means success. Here are some other codes.

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

export

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

expr

The expr command evaluates its arguments as an expression.

x = `expr $x + 1`

Here are some of its expression evaluations

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

printf

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

printf "format string" parameter1 parameter2 ...

Here are some characters and format specifiers.

Escape Sequence	Description
\\	Backslash character
\a	Alert (ring the bell or beep)
\b	Backspace character
\f	Form feed character
\n	Newline character
\r	Carriage return
\t	Tab character
\v	Vertical tab character
\ooo	The single character with octal value ooo

Conversion Specifier	Description
d	Output a decimal number
c	Output a character
s	Output a string
%	Output the % character

return

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

set

The set command sets the parameter variables for the shell.

shift

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

trap

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

    trap command signal

Here are some of the signals.

Signal	Description
HUP (1)	Hang up; usually sent when a terminal goes off line, or a user logs out.
INT (2)	Interrupt; usually sent by pressing *Ctrl-C*.
QUIT (3)	Quit; usually sent by pressing *Ctrl-*.
ABRT (6)	Abort; usually sent on some serious execution error.
ALRM (14)	Alarm; usually used for handling time-outs.
TERM (15)	Terminate; usually sent by the system when it's shutting down.

How It Works

The try it out section has you type in a shell script to test the trap command. It creates a file and keeps saying that it exists until youu cause a control-C interrupt. It does it all again.

unset

The unset command removes variables or functions from the environment.

Command Execution

The result of $(command) is simply the output string from the command, which is then available to the script.

Arithmetic Expansion

The $((...)) is a better alternative to the expr command, which allows simple arithmetic commands to be processed.

    x=$(($x+1))

Parameter Expansion

Using { } around a variable to protect it against expansion.

```
#!/bin/sh

for i in 1 2
do
        my_secret_process ${i}_tmp
done
```

Here are some of the parameter expansion

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

How It Works

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

Here Documents

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

```
#!/bin/sh

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

How It Works

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

Debugging Scripts

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

Command Line Option	set Option	Description
sh -n <script>	set -o noexec set -n	Checks for syntax errors only; doesn't execute commands.
sh -v <script>	set -o verbose set -v	Echoes commands before running them.
sh -x <script>	set -o xtrace set -x	Echoes commands after processing on the command line.
	set -o nounset set -u	Gives an error message when an undefined variable is used.

Putting It All Together

The rest of this chapter is about designing a CD database application.

Requirements

The system should store basic information about each CD, search for CDs, and update or add new CDs.

Design

The three requirements--updating, searching and displaying the CD data--suggest that a simple menu willbe adequate. Here is the example titles file.

Catalog	Title	Type	Composer
CD123	Cool sax	Jazz	Bix
CD234	Classic violin	Classical	Bach
CD345	Hits91	Pop	Various

Here is the associated track file.

Catalog	Track no	Title
CD123	1	Some jazz
CD123	2	More jazz
CD345	1	Dizzy
CD234	1	Sonata in D minor

Notes

The code for the CD database is included in the try it out section. The trap command allows the user to use Ctrl-C.

Assignment 1

1 Write a **SHELL SCRIPT** to find the greatest number among the three numbers, which will inputted through command line and also check the argument must be 3. If it is not 3 then give error message.

2 Write a **SHELL SCRIPT** for calculator containing 5 arithmetic operations.

3 Write a **SHELL SCRIPT** to enter a number and find out whether it is prime or not.

4 Write a **SHELL SCRIPT** to display first n line of a file and the value of n will given through keyboard.

5 Write a **SHELL SCRIPT** to print the number in reverse order.

6 Write a **SHELL SCRIPT** that displays the contents of the currently running script.

7 Write a **SHELL SCRIPT** to find out the highest temperature among the n numbers. The value of n and temperatures will be given through keyboard.

8 Write a **SHELL SCRIPT** to display the process in the system every 30 seconds

9 Write a **SHELL SCRIPT** that displays the last 3 lines of the current directory duly preceeded by the file name

10 Write a **SHELL SCRIPT** that accepts one or more filename as arguments & converts the filenames to uppercase.

11 Write a **SHELL SCRIPT** that accept a pattern and filename as arguments and then count the occurrences of the pattern in the file.

12 Write a **SHELL SCRIPT** to search a pattern from a database using egrep and fgrep.
The pattern and the filename should be entered through keyboard

13 Write a **SHELL SCRIPT**
a. To find the number of USERS currently LOGGED IN
b. Sort the users :
    i) According to their names
    ii) According to their terminal numbers
    iii) According to their time login

14 Write a **SHELL SCRIPT** to create the following menu, enter an option and do according to the given option :
    a. Display the users and the terminal numbers
    b. Display the current date and time
    c. Display the current working directory & give the long listing of that directory.
    d. display the detailed process information of all the users.

15 Given a file name and a user name in the command line argument . Write a **SHELL SCRIPT**
    a. To find the type of the file.

b. Display the file.
c. copy the file to the home directory of the file to the given user.
d. change the ownership of the file to the given user and so it.

16 Date is given at the command line argument in the form day,month,year
Write a **SHELL SCRIPT** to find the calendar and hence to find the day of given date
a. if 3 arguments that indicate day,month and year.
b. if 2 arguments that indicate day and month . Year will be current year.
c. if 1 arguments that indicate day . current Year and month will be considered.
d. if no argument then current date will be considered.

17 Write a **SHELL SCRIPT** that will accept a string from the terminal and print a suitable message “the string does not have at least 10 charecters.

18 Write a **SHELL SCRIPT** that accepts two directory name d1 & d2. Delete those files in d2 whose contents are identical to their namesakes in d1.

Chapter 2 - Processes and Signals

Processes and Signals

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

Here are some of the things you need to understand.

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

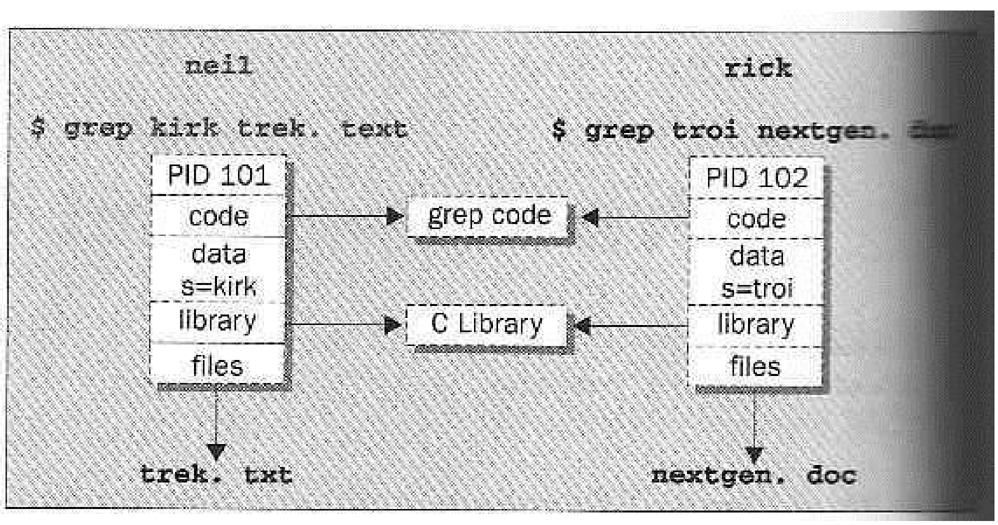
What is a Process?

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

A process is, essentially, a running program.

Process Structure

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



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

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

The system libraries can also be shared.

A process has its own stack space.

The Process Table

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

Viewing Processes

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

Here is some sample output:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

System Processes

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

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

Here we can see one very important process indeed:

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

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

When UNIX starts, it runs a single program, the prime ancestror and process number one: **init**.

One such example is the login procedure **init** starts the **getty** program once for each terminal that we can use to long in.

These are shown in the **ps** output like this:

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

Process Scheduling

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

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

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

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

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

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

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

```
$ nice oclock &
```

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

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

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

So that now the clock program will be scheduled to run less often. We can see the modified nice value with the **ps** again:

```
 F   UID   PID  PPID PRI NI SIZE  RSS WCHAN     STAT TTY   TIME COMMAND
 0   501   146     1  20 10   85  756 130b85    S N  v01   0:00 oclock
```

Notice that the status column now also contains **N**, to indicate that the nice value has changed from the default.

Starting New Processes

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

```
#include <stdlib.h>

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

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

The command is executed as if the command,

```
$ sh -c string
```

has been given to a shell.

Try It Out - system

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

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

int main()
{
    printf("Running ps with system\n");
```

```
    system("ps -ax");
    printf("Done.\n");
    exit(0);
}
```

2. When we compile and run this program, **system.c**, we get the following:

```
$ ./system
Running ps with system
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
...
  146 v01 S N   0:00 oclock
  256 pp0 S     0:00 ./system
  257 pp0 R     0:00 ps -ax
Done.
```

3. The **system** function uses a shell to start the desired program.

We could put the task in the background, by changing the function call to the following:

```
system("ps -ax &");
```

Now, when we compile and run this version of the program, we get:

```
$ ./system2
Running ps with system
Done.
$   PID TTY STAT  TIME COMMAND
      1  ?  S     0:00 init
      7  ?  S     0:00 update (bdflush)
...
    146 v01 S N   0:00 oclock
    266 pp0 R     0:00 ps -ax
```

How It Works

In the first example, the program calls **system** with the string **"ps -ax"**, which executes the **ps** program. Our program returns from the call to **system** when the **ps** command is finished.

In the second example, the call to **system** returns as soon as the shell command finishes. The shell returns as soon as the **ps** program is started, just as would happen if we had typed,

```
$ ps -ax &
```

at a shell prompt.

Replacing a Process Image

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

```
#include <unistd.h>

char **environ;

int execl(const char *path, const char *arg0, ...,  (char *)0);
int execlp(const char *path, const char *arg0, ...,  (char *)0);
int execle(const char *path, const char *arg0, ...,  (char *)0, const char
*envp[]);
int execv(const char *path, const char *argv[]);
int execvp(const char *path, const char *argv[]);
int execve(const char *path, const char *argv[], const char *envp[]);
```

The **exec** family of functions replace the current process with another created according to the arguments given.

If we wish to use an **exec** function to start the **ps** program as in our previous examples, we have the following choices:

```
#include <unistd.h>

/* Example of an argument list */
/* Note that we need a program name for argv[0] */
const char *ps_argv[] =
     {"ps", "-ax", 0};

/* Example environment, not terribly useful */
const char *ps_envp[] =
     {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* Possible calls to exec functions */
execl("/bin/ps", "ps", "-ax", 0);            /* assumes ps is in /bin */

execlp("ps", "ps", "-ax", 0);                /* assumes /bin is in PATH */
execle("/bin/ps", "ps", "-ax", 0, ps_envp); /* passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

Try It Out - exclp

Let's modify our example to use an **exexlp** call.

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

int main()
{
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "-ax", 0);
    printf("Done.\n");
    exit(0);
}
```

Now, when we run this program, **pexec.c**, we get the usual **ps** output, but no **Done**. message at all.

Note also that there is no reference to a process called **pexec** in the output:

```
$ ./pexec
Running ps with execlp
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
...
  146 v01 S N   0:00 oclock
  294 pp0 R     0:00 ps -ax
```

How It Works

The program prints its first message and then calls **execlp**, which searches the directories given by the **PATH** environment variable for a program called **ps**.

It then executes this program in place of our **pexec** program, starting it as if we had given the shell command:

```
$ ps -ax
```

Duplicating a Process Image

To use processes to perform more than one function at a time, we need to create an entirely separate process from within a program.

We can create a new process by calling **fork**. This system call duplicates the current process.

Combined with **exec, fork** is all we need to create new processes to do our bidding.

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

pid_t fork(void);
```

The **fork** system call creates a new child process, identical to the calling process except that the new process has a unique process ID and has the calling process as its parent PID.

A typical code fragment using **fork** is:

```
pid_t new_pid;

new_pid = fork();

switch(new_pid) {
case -1 : /* Error */
    break;
case 0  : /* We are child */
    break;
default : /* We are parent */
    break;
}
```

Try It Out - fork

Let's look at a simple example, **fork.c**:

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid;
    char *message;
    int n;

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

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
    exit(0);
}
```

This program runs as two process. A child prints a message five times. The parent prints a message only three times.

```
$ ./fork
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

How It Works

When the call to **fork** is made, this program divides into two separate processes.

Waiting for a Process

We can arrange for the parent process to wait until the child finishes before continuing by calling **wait**.

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

pid_t wait(int *stat_loc);
```

The **wait** system call causes a parent process to pause until one of its child processes dies or is stopped.

We can interrogate the status information using macros defined in **sys/wait.h**. These include:

Macro	Definition
`WIFEXITED(stat_val)`	Non-zero if the child is terminated normally.
`WEXITSTATUS(stat_val)`	If **WIFEXITED** is non-zero, this returns child exit code.
`WIFSIGNALED(stat_val)`	Non-zero if the child is terminated on an uncaught signal.
`WTERMSIG(stat_val)`	If **WIFSIGNALED** is non-zero, this returns a signal number.
`WIFSTOPPED(stat_val)`	Non-zero if the child has stopped on a signal.
`WSTOPSIG(stat_val)`	If **WIFSTOPPED** is non-zero, this returns a signal number.

Try It Out - wait

1. Let's modify our program slightly so we can wait for and examine the child process exit status. Call the new program **wait.c**.

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
```

```c
int main()
{
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        exit_code = 37;
        break;
    default:
        message = "This is the parent";
        n = 3;
        exit_code = 0;
        break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
```

2. This section of the program waits for the child process to finish:

```c
    if(pid) {
        int stat_val;
        pid_t child_pid;

        child_pid = wait(&stat_val);

        printf("Child has finished: PID = %d\n", child_pid);
        if(WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else
            printf("Child terminated abnormally\n");
    }
    exit (exit_code);
}
```

When we run this program, we see the parent wait for the child. The output isn't confused and the exit code is reported as expected.

```
$ ./wait
fork program starting
This is the parent
This is the child
This is the parent
```

```
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 410
Child exited with code 37
$
```

How It Works

The parent process uses the **wait** system call to suspend its own execution until status information becomes available for a child process.

Zombie Processes

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

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

Try It Out - Zombies

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

Here are the relevant lines of code:

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

How It Works

If we run the above program with **fork2 &** and then call the **ps** program after the child has finished but before the parent has finished, we'll see a line like this:

```
PID TTY STAT   TIME COMMAND
420 pp0 Z      0:00 (fork2) <zombie>
```

There's another system call that you can use to wail for child processes. It's called **waitpid** and youu can use it to wait for a specific process to terminate.

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

If we want to have a parent process regularly check whether a specific child process had terminated, we could use the call,

```
waitpid(child_pid, (int *) 0, WNOHANG);
```

which will return zero if the child has not terminated or stopped or **child_pid** if it has.

Input and Output Redirection

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

Try It Out - Redirection

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

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

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

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

```
$ ./upper
hello THERE
HELLO THERE
^D
$
```

We can, of course, use it to convert a file to uppercase by using the shell redirection:

```
$ cat file.txt
this is the file, file.txt, it is all lower case.
$ upper < file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

2. What if we want to use this filter fromwithin another program? This code, **useupper.c**, accepts a file name as an argument and will respond with an error if called incorrectly:

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

int main(int argc, char *argv[])
{
     char *filename;

     if(argc != 2) {
          fprintf(stderr, "usage: useupper file\n");
          exit(1);
     }

     filename = argv[1];
```

3. The done, we reopen the standard input, again checking for any errors as we do so, and then use **execl** to call **upper**:

```
    if(!freopen(filename, "r", stdin)) {
        fprintf(stderr, "could not redirect stdin to file %s\n", filename);
        exit(2);
    }

    execl("./upper", "upper", 0);
```

4. don't forget that **execl** replaces the current process; provided there is no error, the remaining lines are not executed:

```
        fprintf(stderr, "could not exec upper!\n");
        exit(3);
    }
```

How It Works

when we run this program, we can give it a file to convert to uppercase. The job is done by the program **upper**. The program is executed by:

```
$ ./useupper file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

Because open file descriptors are preserved across the call to **execl**, the **upper** program runs exactly as it would have under the shell command:

```
$ upper < file.txt
```

Threads

UNIX processes can cooperate; they can send each other messages and they can interrupt one another.

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

Signals

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

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

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

Additional signals include:

Signal Name	Description
SIGCHLD	Child process has stopped or exited
SIGCONT	Continue executing, if stopped
SIGSTOP	Stop executing (can't be caught or ignored)
SIGTSTP	Terminal stop signal
SIGTTIN	Background process trying to read
SIGTTOU	Background process trying to write

If the shell and terminal driver are configured normally, typing the interrupt character (Ctrl-C) at the keyboard will result in the **SIGINT** signal being sent to the foreground process. This will cause the program to terminate.

We can handle signals using the **signal** library function.

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

The **signal** function itself returns a function of the same type, which is the previous value of the function set up to handle this signal, or one of these tow special values:

SIG_IGN	Ignore the signal.
SIG_DFL	Restore default behavior.

Try It Out - Signal Handling

1. We'll start by writing the function which reacts to the signal which is passed in the parameter **sig.** Let's call it **ouch**:

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

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

2. The **main** function has to intercept the **SIGINT** signal generated when we type Ctrl-C.

For the rest of the time, it just sits in an infinite loop, printing a message once a second:

```
int main()
{
    (void) signal(SIGINT, ouch);

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

3. While the program is running, typing Ctrl-C causes it to react and then continue.

When we **type** Ctrl-C again, the program ends:

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

How It Works

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

> **FYI**
>
> Note that some UNIX versions, such as those derived from Berkeley UNIX, may not restore the signal action to the default, so an explicit call to `signal` would be required. In this case, you would need a line such as,
>
> `signal(SIGINT, SIG_DFL);`
>
> within the function `ouch`.

Sending Signals

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

```
#include <signal.h>

int raise(int sig);
```

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

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

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

Signals provide us with a useful alarm clock facility.

The **alarm** function call can be used by a process to schedule a **SIGALRM** signal at some time in the future.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Try It Out - An Alarm Clock

1. In **alarm.c**, the first function, **ding**, simulates an alarm clock:

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

void ding(int sig)
{
    printf("alarm has gone off\n");
}
```

2. In **main**, we tell the child process to wait for five seconds before sending a **SIGALRM** signal to its parent:

```
int main()
{
    int pid;

    printf("alarm application starting\n");

    if((pid = fork()) == 0) {
        sleep(5);
        kill(getppid(), SIGALRM);
        exit(0);
    }
```

3. The parent process arranges to catch **SIGALRM** with a call to **signal** and then waits for the inevitable.

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

        pause();

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

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

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

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

It's declared as,

```
#include <unistd.h>

int pause(void);
```

How It Works

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

You must program your signals carefully, as there are a number of 'race conditions' that can occur in programs that use them. For example, if you intend to call **pause** *to wait for a signal and that signal occurs before the call to* **pause**, *your program may wait indefinitely for an event that won't occur. These race conditions, critical timing problems, catch many a novice programmer. Always check signal code very carefully.*

A Robust Signals Interface

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

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

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

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

Try It Out - sigaction

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

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

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

int main()
{
    struct sigaction act;

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

    sigaction(SIGINT, &act, 0);

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

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

Type Ctrl-\ to terminate the program.

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

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

How It Works

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

Signal Sets

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

```
#include <signal.h>

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

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

```
#include <signal.h>

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

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

```
#include <signal.h>

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

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

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

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

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

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

```
#include <sigpending>

int sigpending(sigset_t *set);
```

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

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

```
#include <signal.h>

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

sigaction Flags

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

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

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

access	fstat	read	sysconf
alarm	getegid	rename	tcdrain
cfgetispeed	geteuid	rmdir	tcflow
cfgetospeed	getgid	setgid	tcflush
cfsetispeed	getgroups	setpgid	tcgetattr
cfsetospeed	getpgrp	setsid	tcgetpgrp
chdir	getpid	setuid	tcsendbreak
chmod	getppid	sigaction	tcsetattr
chown	getuid	sigaddset	tcsetpgrp
close	kill	sigdelset	time
creat	link	sigemptyset	times
dup2	lseek	sigfillset	umask
dup	mkdir	sigismember	uname
execle	mkfifo	signal	unlink
execve	open	sigpending	utime
_exit	pathconf	sigprocmask	wait
fcntl	pause	sigsuspend	waitpid
fork	pipe	sleep	write
stat			

Common Signal Reference

Here we list the signals that UNIX programs typically need to get involved with, including the default behaviors:

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

The default action signals is abnormal termination of the process.

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

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

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

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

Signal Name	Description
SIGCONT	Continue executing, if stopped.

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

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

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

Assignment 2

1. Write a **c program** to create a new process , replacing a process image, duplicating a process image, waiting for a process using System call.

2. Write a **c program** to create the orphan process .

3. write a **c program** to create the zombie process .

Chapter 3- Inter-process Communication

Inter-process Communication: Pipes

Now, we look at pipes which allow more useful data to be exchanged between processes.

Here are some of the things you need to understand.

- The definition of a pipe
- Process pipes
- Pipe calls
- Parent and child processes
- Named pipes: FIFOs
- Client/server considerations

What is a Pipe?

We use the word **pipe** when we connect a data flow from one process to another.

Shell commands can be linked together so that the output of one process is fed straight to the input of another.

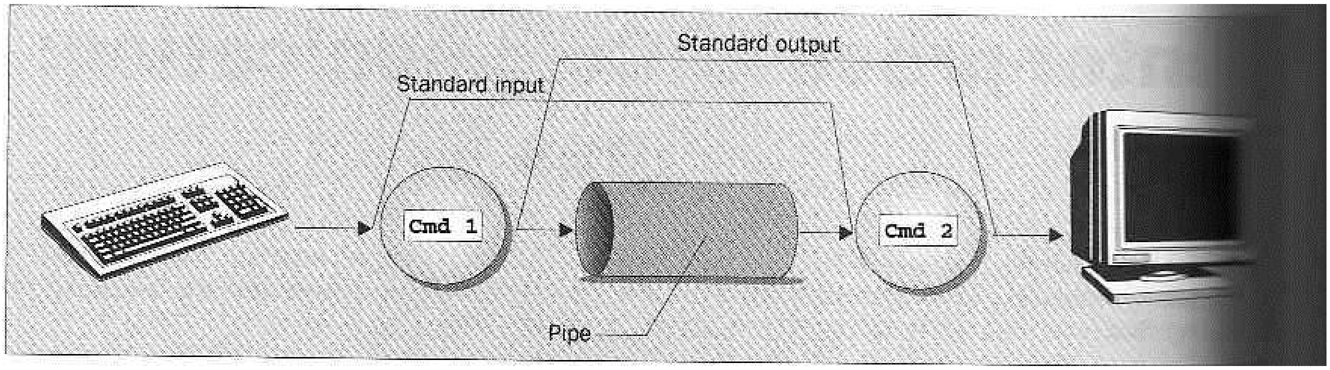
For shell commands, this is entered as:

```
cmd1 | cmd2
```

The shell arranges the standard input and output of the two commands, so that:

- The standard input to **cmd1** comes from the terminal keyboard.
- The standard output from **cmd1** is fed to **cmd2** as its standard input.
- The standard output from **cmd2** is connected to the terminal screen.

The shell has reconnected the standard input and output streams so that data flows from the keyboard input through the two commands and is then output to the screen.



Process Pipes

Perhaps the simplest way of passing data between two programs is with the **popen** and **pclose** functions. These have the prototypes:

```
#include <stdio.h>

FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

popen

The **popen** function allows a program to invoke another program as a new process and either pass data to or receive data from it.

pclose

When the process started with **popen** has finished, we can close the file stream associated with it using **pclose**.

Try It Out - Using popen and pclose

Having initialized the program, we open the pipe to **uname**, making it readable and setting **read_fp** to point to the output.

At the end, the pipe pointed to by **read_fp** is closed

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
     FILE *read_fp;
     char buffer[BUFSIZ + 1];
     int chars_read;
     memset(buffer, '\0', sizeof(buffer));
     read_fp = popen("uname -a", "r");
     if (read_fp != NULL) {
          chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
          if (chars_read > 0) {
               printf("Output was:-\n%s\n", buffer);
          }
          pclose(read_fp);
          exit(EXIT_SUCCESS);
     }
     exit(EXIT_FAILURE);
}
```

When we run this program on one of the author's machine, we get:

```
$ popen1
Output was:-
Linux stones 1.2.8 #1 Mon Sep 18 18:20:08 BST 1995 i586
```

How It Works

The program uses the **popen** call to invoke the **uname** command. It read some information and prints it to the screen.

Sending Output to popen

Here's a program, **popen2.c**, that pipes dta to another. Here, we use the **od** (octal dump).

Try It Out -Sending Output to an External Program

Have a look at the following code, even type it in if you like...

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    FILE *write_fp;
    char buffer[BUFSIZ + 1];

    sprintf(buffer, "Once upon a time, there was...\n");
    write_fp = popen("od -c", "w");
    if (write_fp != NULL) {
        fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
        pclose(write_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

When we run this program, we get the output:

```
$ popen2
0000000   O   n   c   e       u   p   o   n       a       t   i   m   e
0000020   ,       t   h   e   r   e       w   a   s   .   .   .  \n
0000037
```

How It Works

The program uses **popen** with the parameter **w** to start the **od -c** command, so that it can send data to it. The results are printed.

From the command line, we can get the same output with the command:

```
$ echo "Once upon a time, there was..." | od -c
```

Passing More Data

Multiple **fread** and **fwrite** can be used to process more data.

Try It Out - Reading Larger Amounts of Data from a Pipe

Here's a program, **popen3.c**, that reads all of the data from a pipe by using multiple **fread**.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("ps -ax", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

The output we get, edited for brevity, is:

```
$ popen3
Reading:-
   PID TTY STAT  TIME COMMAND
     1  ?   S     0:00 init
     6  ?   S     0:00 bdflush (daemon)
     7  ?   S     0:00 update (bdflush)
    24  ?   S     0:00 /usr/sbin/crond -l10
    39  ?   S     0:00 /usr/sbin/syslogd
...
   240 v02 S      0:02 emacs draft1.txt
Reading:-
   368 v04 S      0:00 popen3
   369 v04 R      0:00 ps -ax
...
```

How It Works

The progran uses **popen** with an **r** parameter, so it continues reading from the file stream until there is no more data available.

How popen is Implemented

The **popen** call runs the program you requested by first invoking the shell, **sh**, passing it the **command** string as an argument.

This has two effects, one good, the other not so good.

1. invoking the shell allows complex shell commands to be started with **popen**.

2. Each call to **popen** invokes the requested program and the shell program. So, each call to **popen** then results in two extra processes being started.

We can count all the lines in example program by **cat**ing

the files and then piping its output to **wc -1** , which counts the number of lines.

On the command line, we would use:

```
$ cat popen*.c | wc -1
```

FYI

Actually, `wc -1 popen*.c` is easier to type and more efficient, but the example serves to illustrate the principle...

Try It Out - popen Starts a Shell

This program uses exactly the command given above, but through **popen** so that it can read the results:.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
```

```
    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("cat popen*.c | wc -l", "r");
    if (read_fp != NULL) {
         chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
         while (chars_read > 0) {
              printf("Reading:-\n %s\n", buffer);
              chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
         }
         pclose(read_fp);
         exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

when we run this program, the output is:

```
$ popen4
Reading:-
     101
```

How It Works

The program shows that the shell is bing invoked to expand **popen*.c** to the list of all files starting with **popen** and ending in **.c** and also feed the output from **cat** into **wc**.

The Pipe Call

The **pipe** function has the prototype:

```
#include <unistd.h>

int pipe(int file_descriptor[2]);
```

pipe is passed an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero.

Some errors defined in the Linux **man** pages for the operation are:

- **EMFILE** Too many file descriptors are in use by the process.
- **ENFILE** The system file table is full.
- **EFAULT** The file descriptor is not valid.

Any data written to **fijle_descriptor[1]** can be read back from **file_descriptor[0]**.

It's important to realize that these are file descriptors, not file streams, so we must use the lower-level **`read`** *and* **`write`** *calls to access the data, rather than* **`fread`** *and* **`fwrite`**.

Try It Out - The pipe Function

Here's a program, **pipe1.c**, that uses **pipe** to create a pipe..

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
     int data_processed;
     int file_pipes[2];
     const char some_data[] = "123";
     char buffer[BUFSIZ + 1];

     memset(buffer, '\0', sizeof(buffer));

     if (pipe(file_pipes) == 0) {
          data_processed = write(file_pipes[1], some_data, strlen(some_data));
          printf("Wrote %d bytes\n", data_processed);
          data_processed = read(file_pipes[0], buffer, BUFSIZ);
          printf("Read %d bytes: %s\n", data_processed, buffer);
          exit(EXIT_SUCCESS);
     }
     exit(EXIT_FAILURE);
}
```

When we run this program, the output is:

```
$ pipe1
Wrote 3 bytes
Read 3 bytes: 123
```

How It Works

The program creates a **pipe** using the two file descriptors **file_pipes[]**. It then writes data into the pipe using the file descriptor **file_pipes[1]** and reads it back from **file_pipes[0]**.

Try It Out - Pipes across a fork

1. This is **pipe2.c**. It start rather like the first examples, up until we make the call to **fork**.

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    int fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
```

2. We've made sure the **fork** worked, so if **fork_result** equals zero, we're in the child process:

```c
if (fork_result == 0) {
    data_processed = read(file_pipes[0], buffer, BUFSIZ);
    printf("Read %d bytes: %s\n", data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

3. Otherwise, we must be the parent process:

```c
else {
    data_processed = write(file_pipes[1], some_data,
                           strlen(some_data));
    printf("Wrote %d bytes\n", data_processed);
```

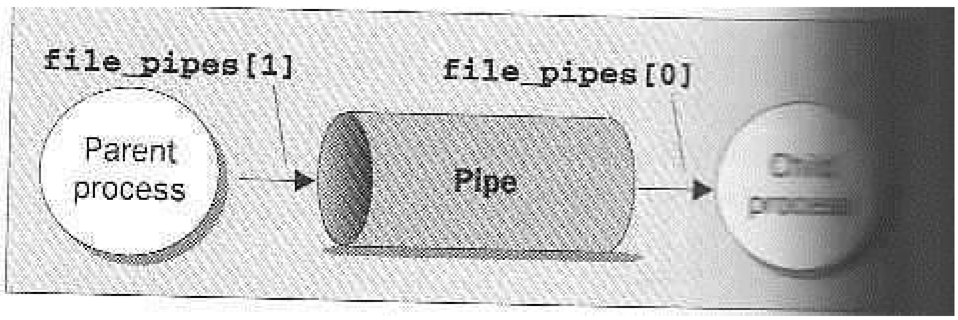
```c
        }
    }
    exit(EXIT_SUCCESS);
}
```

When we run this progra, the output is, as before:

```
$ pipe2
Wrote 3 bytes
Read 3 bytes: 123
```

How It Works

The program creates a pipe with the **pipe** call. It then uses the **fork** call to create a new process. The parent writes to the pipe and the child read from the pipe.



Parent and Child Processes

The child process can be a different program than the parent.

Try It Out - Pipes and exec

Here we have a data producer program and a data consumer program.

1. For the first program, we adapt **pipe2.c** to **pipe3.c**. The lines that we've changed are shown shaded:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
```

```c
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    int fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            sprintf(buffer, "%d", file_pipes[0]);
            (void)execl("pipe4", "pipe4", buffer, (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

2. The 'consumer' program, **pipe4.c**, that reads the data is much similer:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

Remembering that **pipe3** invokes the **pipe4** program for us, when we run **pipe3**, we get the following output:

```
$ pipe3
980 - wrote 3 bytes
981 - read 3 bytes: 123
```

How It Works

The **pipe3** program uses the **pipe** call to create a pippe and then using the **fork** call to create a new process.

pipe4 receives the descriptor number of the pipe as an argument.

A call to **execl** is used to invoke the **pipe4** program. The arguments to **execl** are:

- The program to invoke.
- `argv[0]`, which takes the program name.
- `argv[1]`, which contains the file descriptor number we want the program to read
- `(char *)0`, which terminates the parameters.

Reading Closed Pipes

A **read** on a pipe that isn't open for writing will return 0, allowing the reading process to avoid the 'blocked forever' condition.

Pipes used as Standard Input and Output

We can arange for one of the pipe file descriptors to have a known value, usually the standard input, 0, or the standard output, 1.

The advantage is that we can invoke standard programs, ones that don't expect a file descriptor as a parameter.

There are two closedly related versions of **dup**, that have the prototypes:

```
#include <unistd.h>

int dup(int file_descriptor);
int dup2(int file_descriptor_one, int file_descriptor_two);
```

File Descriptor Manipulation by close and dup

The **dup** always returns a new file descriptor using the lowest available number.

By first closing file descriptor 0 and then calling **dup**, the new file descriptor will have the number zero.

File descriptor number	Initially	After close	After dup
0	Standard input		Pipe file descriptor
1	Standard output	Standard output	Standard output
2	Standard error	Standard error	Standard error
3	Pipe file descriptor	Pipe file descriptor	Pipe file descriptor

Try It Out - Pipes and dup

1. Modify **pipe3.c** to **pipe5.c**, using the following code:

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

#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    int fork_result;

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            close(0);
            dup(file_pipes[0]);
            close(file_pipes[0]);
            close(file_pipes[1]);

            execlp("od", "od", "-c", (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            close(file_pipes[0]);
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            close(file_pipes[1]);
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

The output from this program is:

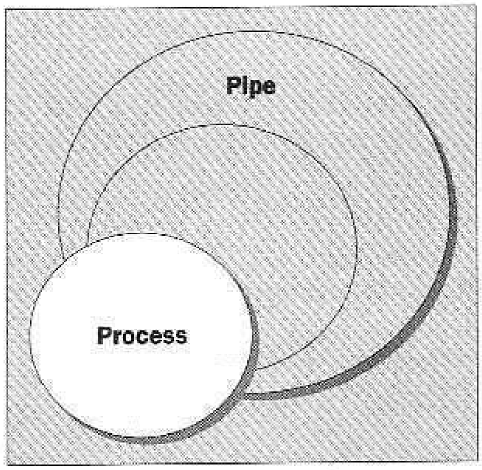
```
$ pipe5
1239 - wrote 3 bytes
0000000   1   2   3
0000003
```

How It Works

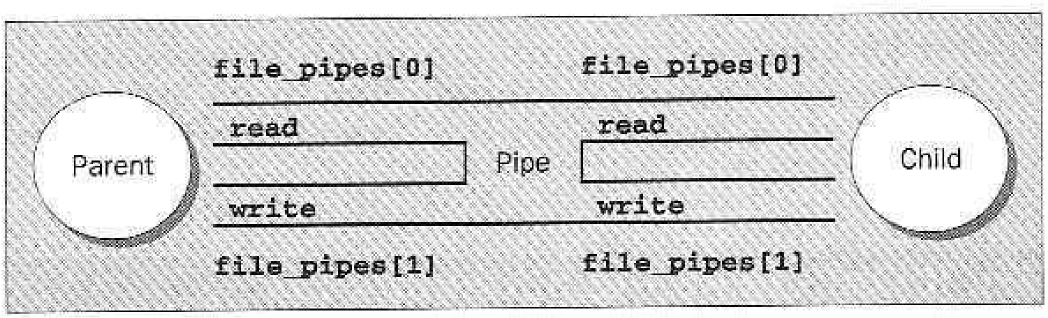
The program creates a pipe and then forks, creating a child process.

The parent and child have access to the pipe.

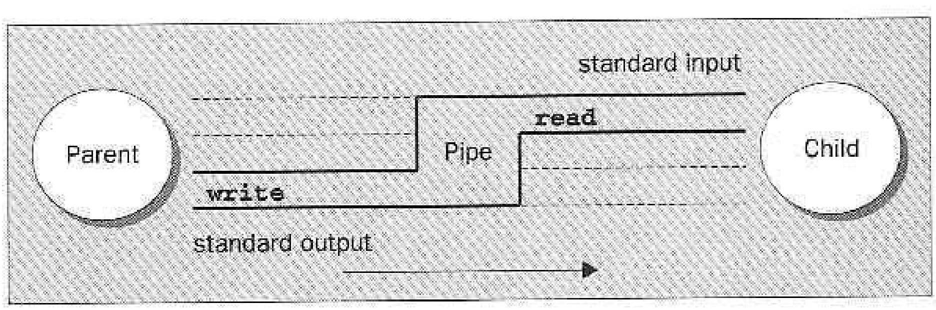
We can show the sequence pictorially. After the call to **pipe**:



After the call to **fork**:



When the program is rady to transfer data:



Named Pipes: FIFOs

We can exchange data with **FIFOs**, often referred to as **named pipes**.

A named pipe is a special type of file that exists as a name in the file system, but behaves like the unnamed pipes that we've met already.

We can create a named pipe using the old UNIX **mknod** command:

```
$ mknod filename p
```

However, it is not in X/Open/ command list, so we use the **mkfifo** command:

```
$ mkfifo filename
```

From inside a program, we can use two different calls. These are:

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

int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);
```

Try It Out - Creating a Named Pipe

For **fifo1.c**, just type in the following code:

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

int main()
{
    int res = mkfifo("/tmp/my_fifo", 0777);
    if (res == 0) printf("FIFO created\n");
    exit(EXIT_SUCCESS);
}
```

We can look for the pipe with:

```
$ ls -lF /tmp/my_fifo
prwxr-xr-x   1 rick     users          0 Dec 10 14:55 /tmp/my_fifo|
```

How It Works

The program uses the **mkfifo** function to create a special file.

Accessing a FIFO

One very useful feature of named pipes is that, because they appear in the file system, we can use them in commands where we would normally use a file name.

Try It Out - Accessing a FIFO File

1. First, let's try reading the (empty) FIFO:

```
$ cat < /tmp/my_fifo
```

2. Now try writing to the FIFO:

```
$ echo "sdsdfasdf" > /tmp/my_fifo
```

3. If we do both at once, we can pass information through the pipe:

```
$ cat < /tmp/my_fifo &
[1] 1316
$ echo "sdsdfasdf" > /tmp/my_fifo
sdsdfasdf

[1]+  Done                    cat </tmp/my_fifo
$
```

NOTICE: the first two stages simply hang until we interrupt them with Ctrl-C.

How It Works

Since there was no data in the FIFO, the **cat** and **echo** programs blocks, waiting for some data to arrive and some other process to read the data, respectively.

The thrid stage works as expected.

> *Unlike a pipe created with the* **pipe** *call, a FIFO exists as a named file, not as an open file descriptor, and must be opened before it can be read from or written to. You open and close a FIFO using the same* **open** *and* **close** *functions that we saw used earlier for files, with some additional functionality. The* **open** *call is passed the path name of the FIFO, rather than that of a regular file.*

Opening a FIFO with open

The main restriction on opening FIFOs is that a program may not open a FIFO for reading and writing with the mode **O_RDWR**.

A process will read its own output back from a pipe if it were opened read/write.

There are four legal combinations of **O_RDONLY**, **O_WRONLY** and the **O_NONBLOCK** flag. We'll consider each in turn.

```
open(const char *path, O_RDONLY);
```

In this case, the **open** call will block, i.e. not return until a process opens the same FIFO for writing.

```
open(const char *path, O_RDONLY | O_NONBLOCK);
```

The **open** will now succeed and return immediately, even if the FIFO has not been opened for writing by any process.

```
open(const char *path, O_WRONLY);
```

In this case, the **open** call will block until a process opens the same FIFO for reading.

```
open(const char *path, O_WRONLY | O_NONBLOCK);
```

This wil always return immediately, but if no process has the FIFO open for reading, **open** will return an error, -1, and the FIFO won't be opened.

Try It Out - Opening FIFO Files

1. Start with the header files, a **#define** and the check that the correct number of command-line arguments have been supplied:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"

int main(int argc, char *argv[])
{
    int res;
    int open_mode = 0;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <some combination of\
                O_RDONLY O_WRONLY O_NONBLOCK\n", *argv);
        exit(EXIT_FAILURE);
    }
```

```
    argv++;
    if (strncmp(*argv, "O_RDONLY", 8) == 0) open_mode |= O_RDONLY;
    if (strncmp(*argv, "O_WRONLY", 8) == 0) open_mode |= O_WRONLY;
    if (strncmp(*argv, "O_NONBLOCK", 10) == 0) open_mode |= O_NONBLOCK;
    argv++;
```

2. Assuming that the program passed the test, we now set the value of **open_mode** from those arguments:

```
    if (*argv) {
        if (strncmp(*argv, "O_RDONLY", 8) == 0) open_mode |= O_RDONLY;
        if (strncmp(*argv, "O_WRONLY", 8) == 0) open_mode |= O_WRONLY;
        if (strncmp(*argv, "O_NONBLOCK", 10) == 0) open_mode |= O_NONBLOC
    }
```

3. We now check whether the FIFO exists, create it if nmecessayr, open it and give it output, wait, and close it.

```
    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO\n", getpid());
    res = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), res);
    sleep(5);
    if (res != -1) (void)close(res);
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

How It Works

This program allows us to specify on the command line the combination of **O_RDONLY**, **O_WRONLY** and **O_NONBLOCK** that we wish to use.

O_RDONLY and O_WRONLY with no O_NONBLOCK

Let's try out a couple of combinations.

```
$ ./fifo2 O_RDONLY &
[1] 152
Process 152 opening FIFO
$ ./fifo2 O_WRONLY
Process 153 opening FIFO
Process 152 result 3
Process 153 result 3
Process 152 finished
Process 153 finished
```

It allows the reader process to start, wait in the **open** command and then both programs to continue when the second program opens the FIFO.

> **FYI** When a UNIX process is blocked, it doesn't consume CPU resources, so this method of process synchronization is very CPU-efficient.

Here is another combination:

```
$ ./fifo2 O_RDONLY O_NONBLOCK &
[1] 160
Process 160 opening FIFO
Process 160 result 3
$ ./fifo2 O_WRONLY
Process 161 opening FIFO
Process 161 result 3
Process 160 finished
Process 161 finished
[1]+  Done                    fifo2 O_RDONLY O_NONBLOCK
```

This time, the reader process executes the **open** call and continues immediately, even though no writer process is present.

Reading and Writing FIFOs

Using the **O_NONBLOCK** mode affects how **read** and **write** calls behave on FIFOs.

A **read** on an empty blocking FIFO will wait until some data can be read.

A **write** on a full blocking FIFO will wait until the data can be written.

A **write** on a FIFO that can't accept all of the bytes being written will either:

- Fail if the request is for **PIPE_BUF** bytes or less and the data can't be written.
- Write part of the data if the request is for more than **PIPE_BUF** bytes, returning the number of bytes actually written, which could be zero.

Try It Out - Inter-process Communication with FIFOs

To show how unrelated processes can communicate using named pipes, we need two separate program, **fifo3.c** and **fifo4.c**.

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_WRONLY;
    int bytes_sent = 0;
    char buffer[BUFFER_SIZE + 1];

    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO O_WRONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
```

```c
        while(bytes_sent < TEN_MEG) {
            res = write(pipe_fd, buffer, BUFFER_SIZE);
            if (res == -1) {
                fprintf(stderr, "Write error on pipe\n");
                exit(EXIT_FAILURE);
            }
            bytes_sent += res;
        }
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

2. Our second program, the consumer, is much simpler. It reads and discards data from the FIFO.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_RDONLY;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;

    memset(buffer, '\0', sizeof(buffer));

    printf("Process %d opening FIFO O_RDONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        do {
            res = read(pipe_fd, buffer, BUFFER_SIZE);
            bytes_read += res;
        } while (res > 0);
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
    exit(EXIT_SUCCESS);
}
```

When we run these porgrams at the same time, using the **time** command to time the reader, the output we get is:

```
$ ./fifo3 &
[1] 375
Process 375 opening FIFO O_WRONLY
$ time ./fifo4
Process 377 opening FIFO O_RDONLY
Process 375 result 3
Process 377 result 3
Process 375 finished
Process 377 finished, 10485760 bytes read
0.00user 0.42system 0:00.75elapsed 55%CPU (0avgtext+0avgdata 0maxreside
0inputs+0outputs (14major+10minor)pagefaults 0swaps
[1]+  Done                    fifo3
```

How It Works

Both programs use the FIFO in blocking mode. **fifo3** is started first and wait for the FIFO to open. When **fifo4** is started, the pipe is unblocked and data transfer occurs.

Assignment 3

1 Write a program for inter process communication between two processes using signal system calls.

2 Write a program to pass the message from one process to another process using message buffer

3 Write a program which will accept a string as input from the command console and send it as a message to the receiver program. The receiver program upon receiving the message from the sender will display the received message as well as send an acknowledgment to the sender program. The sender program will then display “Acknowledgment received from receiver” and then will wait for the next user input from the console.

Chapter 4 - Semaphores

Semaphores

We will now look at a set of Interprocess Communiction Facilities that were introducted in the AT&T System V.2 release of UNIX.

Semaphores

A semaphore is a special varible that takes only whole positive numbers and upon which only two operations are allowed: wait and signal. They are used to ensure that a single executing process has exclusive access to a resource.

Here are the signal notations:

- `P(semaphore variable)` for wait,
- `V(semaphore variable)` for signal.

Semaphore Definition

A **binary semaphore** is a variable that can take only the values 0 and 1.

The definition of **p** and **v** are surprisingly simple. Suppose we have a semaphore variable, **sv**. The two operations are then defined as:

- `P(sv)` If `sv` is greater than zero, decrement `sv`. If `sv` is zero, suspend execution of the process.
- `V(sv)` If some other process has been suspended waiting for `sv`, make it resume execution. If no process is suspended waiting for `sv`, increment `sv`.

A Theoretical Example

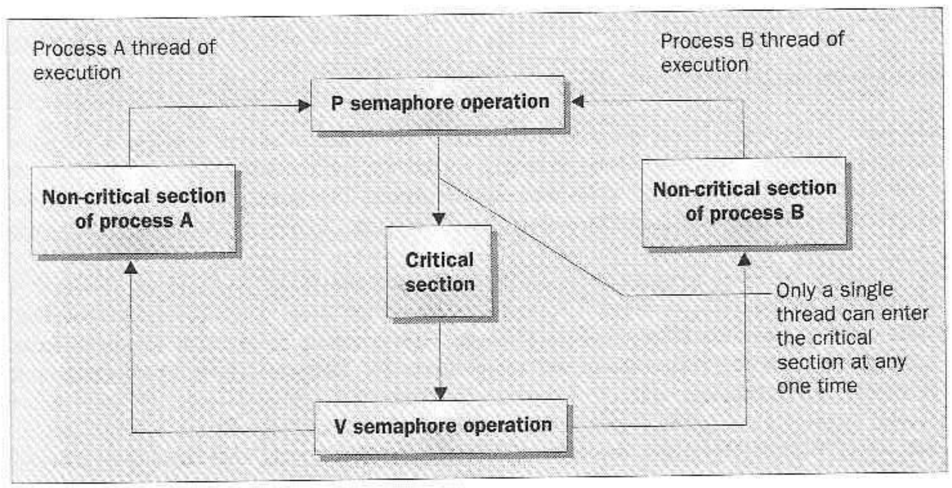
Supposed we have two processes **proc1** and **proc2**, both of which need exclusive access to a database at some point in their execution.

We define a single binary semaphore, **sv**, that starts with the value 1.

The required pseudo-code is:

```
semaphore sv = 1;

loop forever {
    P(sv);
    critical code section;
    V(sv);
    non-critical code section;
}
```

Here's a diagram showing how the **p** and **v** operatons act as a gate into critical sections of code:



UNIX Semaphore Facilities

All the UNIX semaphore functions operate on arrays of general semaphores, rahter than a single binary semaphore.

The semaphore function definitions are:

```
#include <sys/sem.h>

int semctl(int sem_id, int sem_num, int command, ...);
int semget(key_t key, int num_sems, int sem_flags);
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

In practice, the **#include** *files* **sys/types.h** *and* **sys/ipc.h** *are also usually needed to get some of the defines you need for particular operations. There are, however, a few cases when they won't be necessary.*

semget

The **semget** function creates a new semaphore or obtains the semaphore key of an existing semaphore.

```
int semget(key_t key, int num_sems, int sem_flags);
```

semop

The function **semop** is used for changing the value of the semaphore:

```
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

The first parameter, **sem_id**, is the semaphore identifier, as returned from **semget**.

The second parameter, **sem_ops**, is a pointer to an array of structures, each of which will have at least the following members:

```
struct sembuf {
    short sem_num;
    short sem_op;
    short sem_flg;
}
```

semctl

The **semctl** function allows direct control of semaphore information:

```
int semctl(int sem_id, int sem_num, int command, ...);
```

The **command** parameter is the action to take and a fourth parameter, if present, is a **union semun**, which must have at least the following members:

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
}
```

Here are two common values of **command** are:

- **SETVAL**: used for initializing a semaphore to a known value. The value required is passed as the **val** member of the union **semun**. This is required to set the semaphore up before it's used for the first time.
- **IPC_RMID**: used for deleting a semaphore identifier when it's no longer required.

Using Semaphores

To experiment with semaphores, we'll use a single program, **sem1.c**, which we can invoke several times.

We'll use an optional parameter to specify whether the program is responsible for creating and destroying the semaphore.

Try It Out - Semaphores

1. After the **#include**s, the function prototypes and the global variable, we come to the **main** function. It creates the semaphores.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);
static int sem_id;

int main(int argc, char *argv[])
{
    int i;
    int pause_time;
    char op_char = 'O';
```

```
    srand((unsigned int)getpid());

    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);

    if (argc > 1) {
        if (!set_semvalue()) {
            fprintf(stderr, "Failed to initialize semaphore\n");
            exit(EXIT_FAILURE);
        }
        op_char = 'X';
        sleep(2);
    }
```

2. The we have a loop which entersa and leaves the critical section ten times.

There, we first make a call to **semaphore_p** which sets the semaphore to wait.

```
    for(i = 0; i < 10; i++) {
        if (!semaphore_p()) exit(EXIT_FAILURE);
        printf("%c", op_char);fflush(stdout);
        pause_time = rand() % 3;
        sleep(pause_time);
        printf("%c", op_char);fflush(stdout);
```

3. After the critical section, we call **semaphore_v**, setting the semaphore available.

```
        if (!semaphore_v()) exit(EXIT_FAILURE);
        pause_time = rand() % 2;
        sleep(pause_time);
    }
    printf("\n%d - finished\n", getpid());
    if (argc > 1) {
        sleep(10);
        del_semvalue();
    }
    exit(EXIT_SUCCESS);
}
```

4. The function **set_semvalue** initializes the semaphore using the **SETVAL** command in **semctl** call.

```
static int set_semvalue(void)
{
    union semun sem_union;

    sem_union.val = 1;
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
    return(1);
}
```

5. The **del_semvalue** function has almost the same form, except the call to **semctl** uses the command **IPC_RMID** to remove the semaphore's ID:

```
static void del_semvalue(void)
{
    union semun sem_union;

    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}
```

6. **semaphore_p** changes the semaphore by -1 (waiting):

```
static int semaphore_p(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = -1; /* P */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1)
        fprintf(stderr, "semaphore_p failed\n");
        return(0);
    }
    return(1);
}
```

7. **semaphore_v** is identical except for setting the **sem_op** part of the **sembuf** structure to 1, so that the semaphore becomes available:

```
static int semaphore_v(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = 1; /* V */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1)
        fprintf(stderr, "semaphore_v failed\n");
        return(0);
    }
    return(1);
}
```

Here's some sample output, with two invocations of the program:

```
$ sem1 1 &
[1] 1082
$ sem1
OOXXOOXXOOXXOOXXOOXXOOOOXXOOXXOOXXOOXXXX
1083 - finished
1082 - finished
$
```

How It Works

The program sets up a semaphore. It then loops ten times, with pseudo-random waits in its critical and non-critical sections.

The critical section is guarded by calls to our **senaphore_p** and **senaphore_v** functions.

Assignment 4

1 Write a program to process synchronization using semaphore. Implement semaphore as different data structure

References:

1. UNIX, concepts and applications, Sumitabha Das
2. Linux Programming 3rd Edition, Neil Mathew, Richard Stones

SILIGURI INSTITUTE OF TECHNOLOGY

COMPUTER SCIENCE
AND
ENGINEERING DEPARTMENT

INTERNET TECHNOLOGY

LABORATORY MANUAL

LM Rev No: 01

Contents

HTML

1. BASICS OF HTML

- The Web pages or materials in the form of hypermedia documents accessed through the Internet, cab is located anywhere in the world.
- No matter where they are originated, most of the web documents are created using Hypertext Markup Language (HTML). HTML is powerful authoring language and found in different versions like HTML 4.2, HTML 4.0, HTML 3.2, HTML 3.0 and HTML 2.
- HTML element can be used to define document structure & format, HTML element is the inclusive region defined by either a single tag or a pair of tags. A tags is a string in the language surrounded by a less than (<) and a greater than (>) sign. An opening tag does not begin with a slash (/). An ending or closing tag is a string that begins with a slash (/).
- HTML documents format textual information with embedded markup tags that provide style and structure information. Whole document in HTML is surrounded by <HTML> and </HTML>.

1.1 HOW TO CREATE HTML DOCUMENT

HTML document cab is created using any HTML editor and text editor like notepad etc.

1.3 STEPS FOR CREATING A SIMPLE HTML PROGRAM

1. Go to Start -> Programs->Accessories->Notepad.

2. Begin with a document type tag and an <HTML> opening tag. Enter the following line in your doc.

<HTML>

3. Indicate that you are beginning the head element of document by issuing the <HEAD> opening tag. If a <HEAD> element is included, it must appear within an <HTML> element. The following line should appear next in your document:

<HEAD>

4. The <TITLE> element is used to indicate the title of an HTML document.
<TITLE> tags are placed with in the head component of a document and the title is placed between the

Opening and closing <TITLE> tags. Add this <TITLE> element to your document.

<TITLE>My First Page</TITLE>

5. To end the head area issues a <HEAD> closing tag.
</HEAD>

Thus the <HEAD> element is nested within the <HTML> element.

6. At this point the body of the document is developed. A <BODY> opening tag indicates that this point has been reached. Enter the following line.

<BODY>

7. In this case, the body of document contains a simple text statement for now; add the following statement in your file:

Hello World!

8. A </BODY> closing tag marks the end of the <BODY> element. Similar to the Head element, the <BODY> element is also completely nested within the <HTML> element. To end the <BODY> element, issue the closing tag in your document.

</BODY>

9. Finally, terminate the <HTML> tag with </HTML> as shown below:

10. Save your document as mypage.html

11. To view the document, open the .html document in the browser.

Here you will see a sample HTML page with the basic structure.

```
<html>
<head>
<title> Title that is displayed at the top of your web browser</title>
</head>
<body>
<center>
This is my new web page.
</center>
</p>
</body>
</html>
```

- The <html> tag just tells the browser where the HTML starts.
- The <title> tells your browser the title of the page and you will see this text at the very top of your web browser.
- The body of your site should be included inside the <body> tags.

Text & Font commands

SL No.	***Tags***	***Description***
1	**Comment tag:**	Comments in HTML take the form **<!--** comment here-->
2	**Heading tag:**	There are 6 types of heading tag, h1,h2,…h6. ** ** level1 heading ** **
3	**New paragraph**	** ** starts a new paragraph and creates a blank line between your new paragraph and the one above it. The closing tag is ** ** but is not mandatory.
4	**Line Break:**	** ** This will break your text to the next line . Two ** ** tags is equivalent to one **<P>** tag. There's no closing tag needed for this one.
5	**Insert a horizontal line**	**<hr>** This tag is used to insert a horizontal line across the width of the page. This tag does not have an end tag.
6	**Bold**	**** Closing tag is ****

		Or **** Closing tag is ****
7	**Underline**	**<u>** Closing tag is **</u>**
8	**Italics**	**<i>** Closing tag is **</i>**
9	**Centering text**	**<center>** Closing tag is **</center>**
10	**Left aligning text**	** ** Just use ** ** for the closing tag
11	**Right aligning text**	** ** Just use ** ** for the closing tag
12	**Change text color**	 The ending for any font tag is just ****
13	**Changing font face**	**<font face="Arial">**
14	**Change font size**	**<font size="3">** (only goes up to 7)
15	**Blinking Text**	**<blink>** (only works in Netscape)
16	**Scrolling Text**	**<marquee>** (only works in Internet Explorer)
17	**Preformatted Text Tag**	the **<pre>** tag displays preformatted text. The pre element displays all white space and line break exactly as they appear inside the tag. Closing tag is **</pre>**
18	**Order list**	Order list display a ordered or numbered list of items ** ** ** ** 1st item name(No closing tag for list item) ** ** 2nd item name . . . Closing tag is ** **
19	**Unordered list tag**	Unorder list display unnumbered or bulleted list of items ** ** ** ** 1st item name(No closing tag for list item) ** ** 2nd item name Closing tag is ** **
20	**Definition list tag**	dl tag encloses definition list, dt tag encloses definition term, dd tag encloses definition description. dt & dd tag do not require cosing tag.

		<dl> **<dt>** Triangle **<dd>** Three sided figure **.** **.** **.** **</dl>**
21	**Image insert**	The img tag is used to embed an image in an HTML page. The img tag does not require an end tag. **<img src=”**aaa.jpg” **width =**200 **height=**250 **alt=**”This is a picture”**>** **alt** tag is used to pop up a text when you run the mouse over the graphic. To adjust the width and height of the image width and height tag is needed.
22	**Insert sound**	**** sound ****
23	**Insert video**	**<img dynsrc=”**video.mpg” **width=**300 **height=**250>
24	**Background**	<body bgcolor=**green**> **(for color)** **<body background=”**bbb.jpg”**>** (for picture) **< bgsound src=”**aaa.mp3**” loop=**3> (for sound)
25	**Hyperlink**	****yahoo**** (for another site) ****aa**** (for another page) ****section1**** (for a section of a page) For linking a section you have to declare that section on that page by ** sss**** For changing the link color <body link=”green” vlink=”yellow” alink=”purple”> In this example, hyperlink will be green, links that have already been visited will be yellow and active links will be purple.
26	**Basic Table Tags:**	Three most important tags are **<table>** for opening table **<tr>** represents table row **<td>** represents a cell inside the row **<th>** represents table header **<table>**

		<caption align=top>Example of table caption<**/caption>** **<tr> <th>**A**</th> <th>**B**</th> <th>**C**</th>** **</tr>** **<tr> <td>**X**</td> <td>**Y**</td> <td>**Z**</td>** **</tr>** **<tr> <td>**U**</td> <td>**V**</td> <td>**W**</td>** **</tr>** **</table>** **For Entering Table Border:** **<table border=**2> For Changing a Table's Border Color **<table border=**"2" **bordercolor=**"red"**>**
27	**Adjusting Table Cell spacing and Cell Padding**	The **cellspacing** attribute adjusts the space between the cells and **cellpadding** adjusts the space within (around) the cell. **<table border=**2 **cellspacing=**10 **cellpadding=**3**>**
28	**Specifying a Table Width and Height**	You can specify the width and height of a table by using either a percentage or a pixel width. **<table height=**"200"**width =**"300" **border=**2>
29	Setting Column Widths	**<td width=**"70%"> (You can specify width by percentage or pixel width)
30	Horizontally & Vertically Aligning the Content Inside Tables	**<td width=**"210" **align="center" valign="top">**

1.2 FORM

Form tag creates an HTML form. It contain interface elements such as text fields, buttons, checkboxes, radio buttons and selection list.
<form name="sss" >
</form>

Text Field:
<input type="text" name = fname **size=**20 **maxlength=**100**>**

Password field

<input type="password" name = pwd **size=**5 **maxlength=**4>

Radio Button
<input type="radio" name= "aa" **value="**a1" **checked>** a1
<input type="radio" name= "aa" **value="**a2" **>** a2
<input type="radio" name= "aa" **value="**a3"**>** a3

Checkbox
<input type="checkbox" name= "aa" **value="**a1" **>** a1
<input type=" checkbox" name= "aa" **value="**a2" **>** a2
<input type=" checkbox" name= "aa" **value="**a3"**>** a3

Text area
<textarea name="aa**" clos=**40 **rows=**5**>** xxxxxxxxx **</textarea>**

Select

Allows the user to select items from a pull down menu.
<select name="aaa" **size=**3 **multiple >**
<option>a1
<option>a2
<option>a3
<option>a4
</select>

File
<input type="file" name="aa"**>**

Action Button

<input type="reset" name="aaa**" value="**Clear Form"> (For reset Button)
<input type="submit" name="bbb**" value="**Done"> (For submit Button)

1.3 Frames

Frame tag creates a frame . In this document the normal BODY tag is replaced by the FRAMESET tag.
<html>
<head>
<title> Simple frameset example**</title>**
</head>
<frameset cols=" 20% , 80%">
<frame src="ex1.htm**" name="**frame1**">**
<frame src="ex2.htm**" name="**frame2**">**

```
</frameset>
</html>
```

The above example is for split your window into two parts column wise .You can give the value in percentage or in the form of value of pixel. Here you have to create pages ex1.htm and ex2.htm.

Now if you want to split your window row wise in the 3 parts then the syntax will be in the form

```
<html>
<head>
<title> Simple frameset example</title>
</head>
<frameset rows=" 200 ,  100,*">
<frame src="ex1.htm" name="frame1">

<frame src="ex2.htm" name="frame2">
<frame src="ex3.htm" name="frame3">
</frameset>
</html>
```

* denote the rest of the value of pixel.

Now you want to split you window into two rows and then you split the 2nd part into two columns, then the syntax will be

```
<html>
<head>
<title> Simple frameset example</title>
</head>
<frameset rows=" 200 ,  * ">
<frame src="ex1.htm" name="frame1">
<frameset cols=" 20% ,  80%">
<frame src="ex2.htm" name="frame2">
<frame src="ex3.htm" name="frame3">
</frameset>
</frameset>
</html>
```

Assignments on HTML

1. Start your web page with an <html> tag
    - i) Add a heading.
    - ii) Add a title.
    - iii) Start the <body> section.
    - iv) Add the following text using <H1> and </H1> tags:

This Web page was designed by (your name)

v) Add the following text using <H2> and </H2> tags: My HTML assignment

vi) Add a horizontal line

vii) Insert an image to your web page.

Note: You should then refer to your image with just the filename, and NOT the entire pathname to the file.

viii) Add another horizontal line.

ix) Enter a paragraph of text.

Write about things you have learned in html.

Make sure the text in this paragraph is a color other than black, but something one can see.

Add a link that takes you to your favorite webpage.

x) Start a new paragraph. Add a three item ordered list. Make it creative (don't just say item 1, item 2, etc… and keep it clean)!

xi) Close out your body and html tags.

2. Start your web page with an <html> tag

i) Add a heading.

ii) Add a title.

iii) Start the <body> section.

iv) Start a new paragraph.

Use alignment attribute,

Use bold, italic, underline tags,

Use font tag and associated attributes,

Use heading tags,

Use preserve tag,

Use non breaking spaces (escape character).

3. Start your web page with an <html> tag

i) Add a heading.

ii) Add a title.

iii) Start the <body> section.

iv) Start a new paragraph.

Create Hyperlinks:

(a) Within the HTML document.

(b) To another URL.

(c) To a file that can be rendered in the browser.

4. Start your web page with an <html> tag

i) Add a heading.

ii) Add a title.

iii) Start the <body> section.

Create an unordered list,

Create an ordered list,
Use various bullet styles,
Created nested lists,
Use the font tag in conjunction with lists,
Create definition lists,
Use graphics as bullets.

5. Start your web page with an <html> tag
    i) Add a heading.
    ii) Add a title.
    iii) Start the <body> section.

a) Create a simple table
Create borders and adjust border size.
    Adjust table cell spacing.
    Change border color.
    Change table background color.

b) Align a new table on HTML page.
Perform cell text alignment,
Create multi-column tables,
Display information about your academic qualification into this table.

6. Start your web page with an <html> tag
    i) Add a heading.
    ii) Add a title.
    iii) Start the <body> section.

Create a frameset:
Use frame tags,
Create vertical (column) frames,
Create horizontal (row) frames,
Create complex framesets,
Use the hyperlink tag to target displaying an HTML page to another frame.

7. Start your web page with an <html> tag
    i) Add a heading.
    ii) Add a title.
    iii) Start the <body> section.

Create a simple HTML form.
Use the input tag to create a: text box; text area box; check box; list box; radio button; password field; popup menu; hidden field. Use submit and reset buttons. Create an admission form using the above information.

8. Create a web page that will include an image. Then create image map to watch different parts of that image closely.

9. Using frames as an interface, create a series of web pages where the theme is to provide resources (internet, intranet, static HTML pages) pertaining to the subject of HTML. Ideally, your goal is to create a resource that you can use long after this module when needing information on HTML. As a minimum requirement to this assignment your webpage should:
• Consist of at least 3 frames.
• Contain at least 5 URLs to internet and/or intranet sites that you can reference as part of your job.
• Contain at least 5 references to documents that you have created that you use on a regular basis.
• Contain at least 5 references to documents others have created that you use on a regular basis.
• Be organized in a fashion that is logical and intuitive to you.
• Is done with enough quality that you would not be opposed to it being a link at another site.

10. Create a web page as you wish and the html elements of the page will be styled by CSS.

Cascading Style sheets

2.0 CSS syntax

The basic css syntax is made up of 3 parts: **selector {property: value}**

```
<html>
<head>
<style type="text/css">
p
{
color: red
}
</style>
</head>
<body>
<p>
The text in this CSS example will be red.
</p>
</body>
</html>
```

2.1 Placing Style Sheets

Style sheets can be added to HTML pages on 3 different levels: in-line, internal and external.

2.1.1 Internal Style Sheets

The internal style sheet code (**<style type="text/css">**) is placed in between the **head** tags.

The following CSS code example shows how this is done.

The internal style sheet code, **<style type="text/css">**, doesn't do anything visually itself. It simply tells the web browser that an internal style sheet will be used.

```
<html>
<head>
<style type="text/css">
</style>
</head>

<body>
</body>
</html>
```

The internal style sheet in the following CSS code example tells the web browser to make the text color in **the paragraph the color red.**

```
<html>
<head>
<style type="text/css">
p
{
color: red
}
</style>
</head>

<body>
<p>
This CSS example is using an internal style sheet. The color of the text in this paragraph will be red.
</p>
</body>
</html>
```

2.1.2 Inline Style sheet

The following CSS example shows how to insert an in-line style sheet.

```
<html>
<head>
</head>

<body>
<p style="font-weight: bold">
The text in this CSS example will be made bold.
</p>
</body>
</html>
```

If you want to specify multiple properties in an in-line style sheet, each CSS statement must be separated by a semi-colon (;).

The following CSS code shows how this is done.

```
<html>
<head>
</head>

<body>
<p style="font-weight: bold; color: red">
The text in this CSS example will be made bold and the text color will be red.
</p>
</body>
</html>
```

1.1.3 External Style Sheets

The CSS code wich is used to link to the external style sheet is
<link rel="stylesheet" type="text/css" href="test.css" />.

2.1.4 Creating a CSS File

To create a sample CSS file, simply open up notepad, or any other plain text editor and type the following CSS code. Do not add any HTML tags to the external style sheet.

CSS code

```
body
{
background-color: blue
}
p
{
color: red
}
```

Next, save the file with a ".css" extension. Save the CSS file and name it "test.css"

Now create an HTML file with the following code.

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="test.css" />
</head>
<body>
<p>
The text in this paragraph will be a red font and the background will be blue.
</p>
```

</body>
</html>

Now save the HTML file as "example.htm" or "example.html". Next, open the "example.htm" file in your web browser. You have now made a web page that uses external style sheets.

Assignments on Style sheets

Problem 1 Create a html page containing some paragraph, some listing of items as follows.

- Tea
    - Black tea
    - Green tea
- Coffee

Create a CSS rule that makes all text in the paragraph 1.5 times larger than the base font of the system and colors of the text red, and shifts all the list items right by 3ems,and the nested items by 5ems.Use inline style sheets.

Problem 2 Write a css rule that places a background image at the bottom left corner of the page and tiling it horizontally .The image should remain in place when the user scrolls up or down. Use external style sheet.

Problem 3. Write a CSS rule that changes the color of all elements containing attribute class =”greenMove” to green and all heading elements a font size =36pt.Use internal style sheet

Problem 4.Write a web document containing three different style sheets.

JavaScript

3.0 How to Put a JavaScript Into an HTML Page

```
<html>
<body>
<script type="text/javascript">
document.write("Hello World!");
</script>
</body>
</html>
```

The code above will produce this output on an HTML page:

Hello World!

3.1.1 Explanation:

To insert a JavaScript into an HTML page, we use the <script> tag. Inside the <script> tag we use the type attribute to define the scripting language.

So, the <script type="text/javascript"> and </script> tells where the JavaScript starts and ends:

```
<html>
<body>
<script type="text/javascript">
...
</script>
</body>

</html>
```

The word **document.write** is a standard JavaScript command for writing output to a page.

By entering the document.write command between the <script> and </script> tags, the browser will recognize it as a JavaScript command and execute the code line. In this case the browser will write Hello World! to the page:

```
<html>
<body>
<script type="text/javascript">
document.write("Hello World!");
</script>
</body>
</html>
```

The two forward slashes at the end of comment line (//) is the JavaScript comment symbol. This prevents JavaScript from executing the --> tag.

3.1Declaring (Creating) JavaScript Variables

You can declare JavaScript variables with the **var statement**:

```
var x;

var carname;
```

However, you can also assign values to the variables when you declare them:

```
var x=5;

var carname="Volvo";
```

After the execution of the statements above, the variable **x** will hold the value **5**, and **carname** will hold the value **Volvo**.

3.2 Conditional Statements

In JavaScript we have the following conditional statements:

- **if statement** - use this statement if you want to execute some code only if a specified condition is true
- **if...else statement** - use this statement if you want to execute some code if the condition is true and another code if the condition is false
- **if...else if....else statement** - use this statement if you want to select one of many blocks of code to be executed
- **switch statement** - use this statement if you want to select one of many blocks of code to be executed

3.2.1 If Statement

You should use the if statement if you want to execute some code only if a specified condition is true.

Syntax

```
if (condition)
{
code to be executed if condition is true
}
```

Example 1

```
<script type="text/javascript">
//Write a "Good morning" greeting if
//the time is less than 10
var d=new Date();
var time=d.getHours();

if (time<10)
{
document.write("<b>Good morning</b>");
}
</script>
```

3.2.2 If...else Statement

If you want to execute some code if a condition is true and another code if the condition is not true, use the if....else statement.

Syntax

```
if (condition)
{
```

code to be executed if condition is true
}
else
{
code to be executed if condition is not true
}

Example

```
<script type="text/javascript">
//If the time is less than 10,
//you will get a "Good morning" greeting.
//Otherwise you will get a "Good day" greeting.
var d = new Date();
var time = d.getHours();

if (time < 10)
{
document.write("Good morning!");
}
else
{
document.write("Good day!");
}
</script>
```

3.2.3 The JavaScript Switch Statement

You should use the switch statement if you want to select one of many blocks of code to be executed.

Syntax

```
switch(n)
{
case 1:
 execute code block 1
 break;
case 2:
 execute code block 2
 break;
default:
 code to be executed if n is
 different from case 1 and 2
}
```

Example

```
<script type="text/javascript">
//You will receive a different greeting based
//on what day it is. Note that Sunday=0,
//Monday=1, Tuesday=2, etc.
var d=new Date();
theDay=d.getDay();
switch (theDay)
{
```

```
case 5:
 document.write("Finally Friday");
 break;
case 6:
 document.write("Super Saturday");
 break;
case 0:
 document.write("Sleepy Sunday");
 break;
default:
 document.write("I'm looking forward to this weekend!");
}
</script>
```

3.3.4 Alert Box

An alert box is often used if you want to make sure information comes through to the user.When an alert box pops up, the user will have to click "OK" to proceed.

Syntax:

```
alert("sometext");
```

3.3.5 Confirm Box

A confirm box is often used if you want the user to verify or accept something.When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed. If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.

Syntax:

```
confirm("sometext");
```

3.3.6 Prompt Box

A prompt box is often used if you want the user to input a value before entering a page.When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value. If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

Syntax:

```
prompt("sometext","defaultvalue");
```

3.3.6 How to Define a Function

The syntax for creating a function is:

function *functionname*(*var1,var2,...,varX*)
{
some code
}

var1, var2, etc are variables or values passed into the function. The { and the } defines the start and end of the function.

3.3.7 The for Loop

The for loop is used when you know in advance how many times the script should run.

Syntax

```
for (var=startvalue;var<=endvalue;var=var+increment)
{
  code to be executed
}
```

Example

```
<html>
<body>
<script type="text/javascript">
var i=0;
for (i=0;i<=10;i++)
{
document.write("The number is " + i);
document.write("<br />");
}
</script>
</body>
</html>
```

Result:-

The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7

The number is 8
The number is 9
The number is 10

3.3.8 The while loop

The while loop is used when you want the loop to execute and continue executing while the specified condition is true.

```
while (var<=endvalue)
{
   code to be executed
}
```

Example

```
<html>
<body>
<script type="text/javascript">
var i=0;
while (i<=10)

{
document.write("The number is " + i);
document.write("<br />");
i=i+1;
}
</script>
</body>
</html>
```

Result :-

The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
The number is 10

3.3.9 The do...while Loop

Syntax

```
do
```

```
{
  code to be executed
}
while (var<=endvalue);
```

Example

```
<html>
<body>
<script type="text/javascript">
var i=0;
do
{
document.write("The number is " + i);
document.write("<br />");
i=i+1;
}
while (i<0);
</script>
</body>
</html>
```

Result :-

The number is 0

3.3.10 JavaScript For...In Statement

The for...in statement is used to loop (iterate) through the elements of an array or through the properties of an object.

The code in the body of the for ... in loop is executed once for each element/property

Syntax

```
for (variable in object)
{
  code to be executed
}
```

The variable argument can be a named variable, an array element, or a property of an object.

Example

Using for...in to loop through an array:

```
<html>
<body>
```

```
<script type="text/javascript">
var x;
var mycars = new Array();
mycars[0] = "Saab";
mycars[1] = "Volvo";
mycars[2] = "BMW";

for (x in mycars)
{
document.write(mycars[x] + "<br />");
}
</script>
</body>
</html>
```

3.4 Events

By using JavaScript, we have the ability to create dynamic web pages. Events are actions that can be detected by JavaScript.

Every element on a web page has certain events which can trigger JavaScript functions. For example, we can use the onClick event of a button element to indicate that a function will run when a user clicks on the button. We define the events in the HTML tags.

3.4.1 Examples of events:

- **A mouse click**
- **A web page or an image loading**
- **Mousing over a hot spot on the web page**
- **Selecting an input box in an HTML form**
- **Submitting an HTML form**
- **A keystroke**

Note: Events are normally used in combination with functions, and the function will not be executed before the event occurs!

3.4.1.1 onLoad and onUnload

The onload and onUnload events are triggered when the user enters or leaves the page.

The onload event is often used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.

Both the onload and onUnload events are also often used to deal with cookies that should be set when a user enters or leaves a page. For example, you could have a popup asking for the user's name upon his first arrival to your page. The name is then stored in a cookie. Next time the visitor arrives at your page, you could have another popup saying something like: "Welcome John Doe!".

3.4.1.2 onFocus, onBlur and onChange

The onFocus, onBlur and onChange events are often used in combination with validation of form fields.

Below is an example of how to use the onChange event. The checkEmail() function will be called whenever the user changes the content of the field:

```
<input type="text" size="30"
id="email" onchange="checkEmail()">
```

3.4.1.3 onSubmit

The onSubmit event is used to validate ALL form fields before submitting it.

Below is an example of how to use the onSubmit event. The checkForm() function will be called when the user clicks the submit button in the form. If the field values are not accepted, the submit should be cancelled. The function checkForm() returns either true or false. If it returns true the form will be submitted, otherwise the submit will be cancelled:

```
<form method="post" action="xxx.htm"
onsubmit="return checkForm()">
```

3.4.1.4 onMouseOver and onMouseOut

onMouseOver and onMouseOut are often used to create "animated" buttons.Below is an example of an onMouseOver event. An alert box appears when an onMouseOver event is detected:

```
<a href="http://www.abc.com"
onmouseover="alert('An onMouseOver event');return false">
<img src="abc.gif" width="100" height="30">
</a>
```

3.5 JavaScript Form Validations

JavaScript can be used to validate input data in HTML forms before sending off the content to a server.

Form data that typically are checked by a JavaScript could be:

- Has the user left required fields empty?
- Has the user entered a valid e-mail address?
- Has the user entered a valid date?
- Has the user entered text in a numeric field?

3.5.1 Required Fields

The function below checks if a required field has been left empty. If the required field is blank, an alert box alerts a message and the function returns false. If a value is entered, the function returns true (means that data is OK):

```
function validate_required(field,alerttxt)
{
with (field)
{
 if (value==null||value=="")
 {
 alert(alerttxt);return false;
 }
 else
 {
 return true;
 }
}
}
```

The entire script, with the HTML form could look something like this

```
<html>
<head>
<script type="text/javascript">
function validate_required(field,alerttxt)
{
with (field)
{
if (value==null||value=="")
 {alert(alerttxt);return false;}
else {return true}
}
}
function validate_form(thisform)
{
with (thisform)
{
if (validate_required(email,"Email must be filled out!")==false)
 {email.focus();return false;}
}
}
</script>
</head>
<body>
<form action="submitpage.htm"
```

```
onsubmit="return validate_form(this)"
method="post">
Email: <input type="text" name="email" size="30">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Assignments on JavaScript

1.Design a html page that has three fields email, name and age and a Submit button. If you enter wrong email(e.g. “@” sign is missing), name (>10 characters), age (>100) corresponding alert message /s will be fired.

2.Write a program using javascript where the program chooses a number between 1 and 20. You are then prompted to enter a guess. If the player guess wrong then the prompt appears again until the guess is correct. When the player has made a successful guess the computer will give a "Well guessed!" message, and the program will exit.

3.Display a clock using javascript

4.Write a javascript code to create a button.If you click on it,a prompt box will appear asking your name. If you enter your name and click the ‘ok’ button a greeting message will appear.

5.Design an html page to compare two numbers supplied by user. The bigger number will be displayed in a separate field. Use array object.

6.Design an html page that has three buttons “red” ,“green”, “blue”. If you click any of them the background color also changes as the name of the button showing the corresponding alert message.

7.Write a program in javascript that will take two numbers as user input and calculate their sum, product, division and modulus by clicking the appropriate button/s.

8.Validate e-mail, phone number, name using regx.

PERL

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, CGI and more.

4.1 What is PERL?

- Perl is a stable, cross platform programming language.
- Perl stands for **Practical Extraction and Report Language**.
- Perl was created by Larry Wall.
- Perl is a programming language which can be used for a large variety of tasks. A typical simple use of Perl would be for extracting information from a text file and printing out a report or for converting a text file into another form.

4.2 PERL Features

- Perl takes the best features from other languages, such as C, awk, sed, sh, and BASIC, among others.
- Perls database integration interface (DBI) supports third-party databases including Oracle, Sybase, MySQL and others.
- Perl works with HTML, XML, and other mark-up languages.
- Perl supports both procedural and object-oriented programming.
- The Perl interpreter can be embedded into other systems.

4.3 Is Perl Compiled or Interpreted?

Perl is implemented as an interpreted (not compiled) language. Traditional compilers convert programs into machine language. When you run a Perl program, it's first compiled into a byte code, which is then converted (as the program runs) into machine instructions. So it is not quite the same as shells, which are "strictly" interpreted without an intermediate representation. Nor it is like most versions of C or C++, which are compiled directly into a machine dependent format.

4.3.1 PERL Syntax Overview

1. **Perl statements end in a semi-colon:** print "Hello, world";
2. **Comment Statement:** # This is a comment
3. **White Space is irrelevant:** print   "Hello, world";
4. Double quotes or single quotes may be used around literal strings:

```
print "Hello, world";
print 'Hello, world';
```

BUT THE IMPORTANT PONT IS THAT - A String in-between Single quotes **(' ')** has value exactly the sequence of characters. In case of **(" ")** Substitution is occurred.

Example:

```
$i=10;
$s1='winter for last $i months';
$s2="winter for last $i months";

print $i;
print $s1;
print $s2;
```

Output:

```
10
winter for last $i months
winter for last 10 months
```

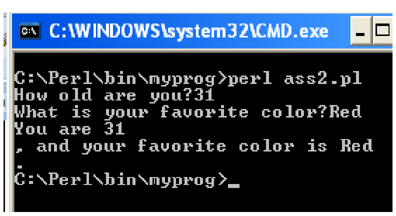
4.3.2 Chomp () function in PERL

The chomp() function will remove (usually) any newline character from the end of a string. When reading user input from the standard input stream (STDIN) for instance, you get a newline character with each line of data. chomp() is really useful in this case because you do not need to write a regular expression and you do not need to worry about it removing needed characters.

Normal Syntax:

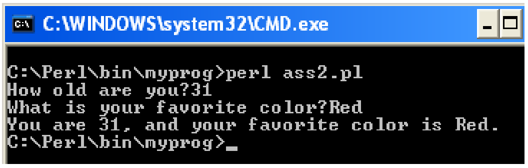
```
print "How old are you?";
$age = <>;
print "What is your favorite color?";
$color = <>;
print "You are $age, and your favorite color is $color.";
```

Output:



Using Chomp():

```
print "How old are you?";
chomp($age = <>);
print "What is your favorite color?";
chomp($color = <>);
print "You are $age, and your favorite color is $color.";
```



4.4PERL Variable Types

Perl has three built in variable types:

- **Scalar ($)**
- **Array (@)**
- **Hash (%)**

```
    scalar => {
description => "single item",
sigil => '$',
},
    array  => {
description => "ordered list of items",
sigil => '@',
},
    hash   => {
description => "key/value pairs",
sigil => '%',
},
```

4.5 Scalar Variables:

A scaler variable is represented by doller sign ($).

A scalar represents a single value as follows:

```
my $animal = "camel"; my $answer = 42;
```

Here **my** is a keyword.

A scalar values can be strings, integers or floating point numbers, and Perl will automatically convert between them as required. There is no need to pre-declare your variable types. Scalar values can be used in various ways:

```
$age = 25;  integer
$name = “Anupam”  String
$Salary = 1445.50  Floating
```

4.6 Array Variables:

An array is a variable that stors an ordered list of Scalar variables. It is represented through “@” Symbol.

@ages = (25,30,40);

@name = (“Ram”, “Hari”, “Madhu”);

print “ages[0] = $ages[0]”;   print “ages[1] = $ages[1]”;   print “ages[2] = $ages[2]”;

Output: ages[0]=25   ages[1]=30   ages[2]=40

4.6 Hash Variables:

Hash variables are represented through “%” symbol.

A hash is a set of key/value pairs. To refer a single element of a hash, you will use the hash variable name followed by the “key” associated with the value in brackets.

%data = (‘John’,45,’Lisa’,30,’Kumar’,40);

Print “\data{‘John’}=$data{‘John’}\n”;

Print “\data{‘Lisa’}=$data{‘Lisa’}\n”;

Print “\data{‘Kumar’}=$data{‘Kumar’}\n”;

4.7 SOME BASIC PARL PROGRAM:

Assignment 1: Write a perl script to take input from the user
#such as name,Roll,Department,Stream and diplay it with proper syntax.

NOTE: (#) it is **Comment statement**.

- Perl statements end in a **semi-colon**(;)

  print"hello\n world";

- **variable declearation**
  $name=anupam;
- Variable print
  print"\n$name";

  <STDIN> stands for standard input.
- It can be abbreviated by using simple <>.

  print "\nHow old are you?";
  $age = <>;
  print "WOW! You are $age years old!";

#Assignment : WAP in perl to take input from user terminal and display it by using chomp function.

```
print "How old are you?";
chomp ($age = <>);
print "What is your favorite color?";
chomp ($color = <>);
print "You are $age, and your favorite color is $color.";
```

Assignments

1.Write a perl script to take input from the use such as name,Roll,Department,Stream and diplay it with proper syntax.It is Comment statement.
Perl statements end in a semi-colon(;)this would print with a linebreak in the middle

2.WAP in perl to take input from user terminal and display it by using chomp function.

3. Write a simple Perl script to take input name, college, stream as a input from the terminal and display it.
4. Write a Perl script to search a word from a sentence.
   a) Using String matching method.
   b) Using Substitution method.
5. Write a Perl script to implement Celsius to Fahrenheit Converter
6. Write a Perl script to convert all lower case sentence to upper case.
7. Write a Perl script to convert all first letter of a sentence to upper case.

8. Write a Perl script to implement the regular expression as follows:
9.If a string starts with MCA and end with bw, print 1 else 0.
10.Implement the following with regular expression in Perl:

   A) a* at least 2 b's

   B) a* exactly 3 b's
   C) a*bc

12.Write a perl script to implement associative array.

Socket Programming using Java

5. **What is socket?** Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a twoway communication protocol, so data can be sent across both streams at the same time. There are following usefull classes providing complete set of methods to implement sockets.

5.1 ServerSocket Class Methods

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests

The ServerSocket class has four constructors:

SN	Methods with Description
1	**public ServerSocket(int port) throws IOException** Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	**public ServerSocket(int port, int backlog) throws IOException** Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.

public ServerSocket(int port, int backlog, InetAddress address) throws IOException

3 Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on

public ServerSocket() throws IOException

4 Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Here are some of the common methods of the ServerSocket class:

SN

5.2 Methods with Description

public int getLocalPort()

1 Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.

public Socket accept() throws IOException

2 Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely

public void setSoTimeout(int timeout)

3 Sets the time-out value for how long the server socket waits for a client during the accept().

public void bind(SocketAddress host, int backlog)

4 Binds the socket to the specified server and port in the SocketAddress object. Use this method if you instantiated the ServerSocket using the no-argument constructor.

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and server, and communication can begin.

5.3 Socket Class Methods

The **java.net.Socket** class represents the socket that both the client and server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.

The Socket class has five constructors that a client uses to connect to a server:

SN	5.4 Methods with Description
1	**public Socket(String host, int port) throws UnknownHostException, IOException.** This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
2	**public Socket(InetAddress host, int port) throws IOException** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.
3	**public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.** Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4	**public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String
5	**public Socket()** Creates an unconnected socket. Use the connect() method to connect this socket to a server.

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here. Notice that both the client and server have a Socket object, so these methods can be invoked by both the client and server.

SN	5.5 Methods with Description
1	**public void connect(SocketAddress host, int timeout) throws IOException** This method connects the socket to the specified host. This method is needed only when you

instantiated the Socket using the no-argument constructor.

2 **public InetAddress getInetAddress()**

This method returns the address of the other computer that this socket is connected to.

3 **public int getPort()**

Returns the port the socket is bound to on the remote machine.

4 **public int getLocalPort()**

Returns the port the socket is bound to on the local machine.

5 **public SocketAddress getRemoteSocketAddress()**

Returns the address of the remote socket.

6 **public InputStream getInputStream() throws IOException**

Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.

7 **public OutputStream getOutputStream() throws IOException**

Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket

8 **public void close() throws IOException**

Closes the socket, which makes this Socket object no longer capable of connecting again to any server

InetAddress Class Methods:

This class represents an Internet Protocol (IP) address. Here are following usefull methods which you would need while doing socket programming:

SN 5.6 Methods with Description

1 **static InetAddress getByAddress(byte[] addr)**

Returns an InetAddress object given the raw IP address .

2 **static InetAddress getByAddress(String host, byte[] addr)**

Create an InetAddress based on the provided host name and IP address.

3 **static InetAddress getByName(String host)**

Determines the IP address of a host, given the host's name.

4 **String getHostAddress()**

Returns the IP address string in textual presentation.

String getHostName()
5
Gets the host name for this IP address.
static InetAddress InetAddress getLocalHost()
6
Returns the local host.
String toString()
7
Converts this IP address to a String.

5.7 Socket Client Example

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

```
// File Name GreetingClient.java

import java.net.*;
import java.io.*;

public class GreetingClient
{
   public static void main(String [] args)
   {
      String serverName = args[0];
      int port = Integer.parseInt(args[1]);
      try
      {
         System.out.println("Connecting to " + serverName +
                 " on port " + port);
         Socket client = new Socket(serverName, port);
         System.out.println("Just connected to "
                 + client.getRemoteSocketAddress());
         OutputStream outToServer = client.getOutputStream();
         DataOutputStream out = new DataOutputStream(outToServer);
         out.writeUTF("Hello from "
                      + client.getLocalSocketAddress());
         InputStream inFromServer = client.getInputStream();
         DataInputStream in =
                        new DataInputStream(inFromServer);
         System.out.println("Server says " + in.readUTF());
         client.close();
      }catch(IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

Socket Server Example:

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument:

```java
// File Name GreetingServer.java

import java.net.*;
import java.io.*;

public class GreetingServer extends Thread
{
   private ServerSocket serverSocket;

   public GreetingServer(int port) throws IOException
   {
      serverSocket = new ServerSocket(port);
      serverSocket.setSoTimeout(10000);
   }

   public void run()
   {
      while(true)
      {
         try
         {
            System.out.println("Waiting for client on port " +
            serverSocket.getLocalPort() + "...");
            Socket server = serverSocket.accept();
            System.out.println("Just connected to "
                  + server.getRemoteSocketAddress());
            DataInputStream in =
                  new DataInputStream(server.getInputStream());
            System.out.println(in.readUTF());
            DataOutputStream out =
                 new DataOutputStream(server.getOutputStream());
            out.writeUTF("Thank you for connecting to "
              + server.getLocalSocketAddress() + "\nGoodbye!");
            server.close();
         }catch(SocketTimeoutException s)
         {
            System.out.println("Socket timed out!");
            break;
         }catch(IOException e)
         {
            e.printStackTrace();
            break;
         }
      }
   }
   public static void main(String [] args)
   {
      int port = Integer.parseInt(args[0]);
      try
      {
         Thread t = new GreetingServer(port);
         t.start();
      }catch(IOException e)
      {
         e.printStackTrace();
      }
   }
```

```
}
```

Compile client and server and then start server as follows:

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

Check client program as follows:

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

Assignments on Socket progrmas

1.Write a socket program in java to create Echo client and Echo server.
2. Write a socket program in java to display the system date and time .
3. Write a socket program in java to convert lowercase letter to uppercase.
4. Write a socket program in java to create chat client and chat server.

RMI

6 Objectives to Learn RMI

• Capitalizes on “Java Object Model”
• Distributed application protocols in term of interfaces, classes, and method invocations
• Insulated from low level details of network communications (sockets, byte layout, etc.)
• Minimizes Complexity

• Preserves safety of the   Preserves safety of the Java runtime environment

6.1 Overview of RMI

There are three processes that participate in supporting remote method invocation.

1. The *Client* is the process that is invoking a method on a remote object.
2. The *Server* is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.
3. The *Object Registry* is a name server that relates objects with names. Objects are *registered* with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

There are two kinds of classes that can be used in Java RMI.

1. A **Remote class** is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:

1. Within the address space where the object was constructed, the object is an ordinary object which can be used like any other object.
2. Within other address spaces, the object can be referenced using an **object handle**. While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object.

For simplicity, an instance of a Remote class will be called a **remote object**.

2. A **Serializable** class is one whose instances can be copied from one address space to another. An instance of a **Serializable** class will be called a **serializable object**. In other words, a serializable object is one that can be marshaled. Note that this concept has no connection to the concept of **serializability** in database management systems.

   If a serializable object is passed as a parameter (or return value) of a remote method invocation, then the value of the object will be copied from one address space to the other. By contrast if a remote object is passed as a parameter (or return value), then the object handle will be copied from one address space to the other.

One might naturally wonder what would happen if a class were both Remote and Serializable. While this might be possible in theory, it is a poor design to mix these two notions as it makes the design difficult to understand.

6.2 Serializable Classes

We now consider how to design Remote and Serializable classes. The easier of the two is a Serializable class. A class is Serializable if it implements the `java.io.Serializable` interface. Subclasses of a Serializable class are also Serializable. Many of the standard classes are Serializable, so a subclass of one of these is automatically also Serializable. Normally, any data within a Serializable class should also be Serializable. Although there are ways to include non-serializable objects within a serializable objects, it is awkward to do so. See the documentation of `java.io.Serializable` for more information about this.

Using a serializable object in a remote method invocation is straightforward. One simply passes the object using a parameter or as the return value. The type of the parameter or return value is the Serializable class. Note that both the Client and Server programs must have access to the definition of any Serializable class that is being used. If the Client and Server programs are on different machines, then class definitions of Serializable classes may have to be downloaded from one machine to the other. Such a download could violate system security.

The only Serializable class that will be used in the "Hello, world!" example is the String class, so no problems with security arise.

6.3 Remote Classes and Interfaces

Next consider how to define a **Remote class**. This is more difficult than defining a Serializable class. A Remote class has two parts: the interface and the class itself. The Remote interface must have the following properties:

1. The interface must be public.
2. The interface must extend the interface `java.rmi.Remote`.
3. Every method in the interface must declare that it throws `java.rmi.RemoteException`. Other exceptions may also be thrown.

The Remote class itself has the following properties:

1. It must implement a Remote interface.
2. It should extend the `java.rmi.server.UnicastRemoteObject` class. Objects of such a class exist in the address space of the server and can be invoked remotely. While there are other ways to define a Remote class, this is the simplest way to ensure that objects of a class can be used as remote objects. See the documentation of the `java.rmi.server` package for more information.
3. It can have methods that are not in its Remote interface. These can only be invoked locally.

Unlike the case of a Serializable class, it is not necessary for both the Client and the Server to have access to the definition of the Remote class. The Server requires the definition of both the Remote class and the Remote interface, but the Client only uses the Remote interface. Roughly speaking, the Remote interface represents the type of an object handle, while the Remote class represents the type of an object. If a remote object is being used remotely, its type must be declared to be the type of the Remote interface, not the type of the Remote class.

In the example program, we need a Remote class and its corresponding Remote interface. We call these `Hello` and `HelloInterface`, respectively. Here is the file `HelloInterface.java`:

```
import java.rmi.*;
/**
 * Remote Interface for the "Hello, world!" example.
 */
public interface HelloInterface extends Remote {
  /**
   * Remotely invocable method.
   * @return the message of the remote object, such as "Hello, world!".
   * @exception RemoteException if the remote invocation fails.
   */
  public String say() throws RemoteException;
}
```

Here is the file `Hello.java`:

```
import java.rmi.*;
import java.rmi.server.*;
/**
 * Remote Class for the "Hello, world!" example.
 */
public class Hello extends UnicastRemoteObject implements HelloInterface {
  private String message;
  /**
   * Construct a remote object
```

```
 * @param msg the message of the remote object, such as "Hello, world!".
 * @exception RemoteException if the object handle cannot be constructed.
 */
public Hello (String msg) throws RemoteException {
  message = msg;
}
/**
 * Implementation of the remotely invocable method.
 * @return the message of the remote object, such as "Hello, world!".
 * @exception RemoteException if the remote invocation fails.
 */
public String say() throws RemoteException {
  return message;
}
}
```

All of the Remote interfaces and classes should be compiled using `javac`. Once this has been completed, the stubs and skeletons for the Remote interfaces should be compiled by using the `rmic` stub compiler. The stub and skeleton of the example Remote interface are compiled with the command:

```
rmic Hello
```

The only problem one might encounter with this command is that `rmic` might not be able to find the files `Hello.class` and `HelloInterface.class` even though they are in the same directory where `rmic` is being executed. If this happens to you, then try setting the `CLASSPATH` environment variable to the current directory, as in the following command:

```
setenv CLASSPATH .
```

If your `CLASSPATH` variable already has some directories in it, then you might want to add the current directory to the others.

6.3.1 Example code of RMI Interface

```
import java.rmi.*;
import java.rmi.server.*;
public interface Rmi_Interface extends Remote
{
  public double add(double a,double b)throws RemoteException;
  public double sub(double a,double b)throws RemoteException;
  public double mul(double a,double b)throws RemoteException;
  public double div(double a,double b)throws RemoteException;
}
```

6.4 Steps to run RMI application

Steps

Java Files Are:-

Rmi_Interface.java

Rmi_Client.java

Rmi_Server.java

1. Compile all files (javac Rmi_Interface.java, javac Rmi_Client.java, javac Rmi_Server.java)
2. rmic Rmi_Server (RMI Compilation of Rmi_Server.class file)
(it will create Rmi_Server_Skel.class and Rmi_Server_Stub.class)
3. Copy Rmi_Client.class, Rmi_interface.class and Rmi_Server_Stub.class file to another machine
(that machine will play as a Client machine. Present machine will the Server machine)
4. In the server Machine (present machine) run c:\jdk1.3\bin>rmiregistry
(default port will be 1099 or c:\jdk1.3\bin>rmiregistry 2210 then port will be 2210)
5. In the Server Machine run c:\jdk1.3\bin>java Rmi_Server
(Server is ready)

6. In the Client machine run c:\jdk1.3\bin>java Rmi_Client 20 5
(20 and 5 is the command line argument as per Prog.)

Summary Table:-

Server Machine	**Client Machine**
Rmi_Interface.class	**Rmi_Interface.class**
Rmi_Server.class	**Rmi_Client.class**
Rmi_Server_Skel.class	**Rmi_Server_Stub.class**
Rmi_Server_Stub.class	
c:\jdk1.3\bin>rmiregistry	**c:\jdk1.3\bin>java Rmi_Client 20 5**
c:\jdk1.3\bin>java Rmi_Server	

6.5 Programming a Client

Having described how to define Remote and Serializable classes, we now discuss how to program the Client and Server. The Client itself is just a Java program. It need not be part of a Remote or Serializable class, although it will use Remote and Serializable classes.

A remote method invocation can return a remote object as its return value, but one must have a remote object in order to perform a remote method invocation. So to obtain a remote object one must already have one. Accordingly, there must be a separate mechanism for obtaining the first remote object. The Object Registry fulfills this requirement. It allows one to obtain a remote object using only the name of the remote object.

The name of a remote object includes the following information:

1. The Internet name (or address) of the machine that is running the Object Registry with which the remote object is being registered. If the Object Registry is running on the same machine as the one that is making the request, then the name of the machine can be omitted.
2. The port to which the Object Registry is listening. If the Object Registry is listening to the default port, 1099, then this does not have to be included in the name.
3. The local name of the remote object within the Object Registry.

Here is the example Client program:

```
/**
 * Client program for the "Hello, world!" example.
 * @param argv The command line arguments which are ignored.
 */
public static void main (String[] argv) {
  try {
    HelloInterface hello =
      (HelloInterface) Naming.lookup ("//ortles.ccs.neu.edu/Hello");
    System.out.println (hello.say());
  } catch (Exception e) {
    System.out.println ("HelloClient exception: " + e);
  }
}
```

The `Naming.lookup` method obtains an object handle from the Object Registry running on `ortles.ccs.neu.edu` and listening to the default port. Note that the result of `Naming.lookup` must be cast to the type of the Remote interface.

The remote method invocation in the example Client is `hello.say()`. It returns a String which is then printed. A remote method invocation can return a String object because String is a Serializable class.

The code for the Client can be placed in any convenient class. In the example Client, it was placed in a class `HelloClient` that contains only the program above.

Assignments on RMI

1.Write a program to design a simple calculator using RMI.

XML

7. XML stands for E**x**tensible **M**arkup **L**anguage. It is a general purpose specification to create custom markup language for sharing structure data via Internet, encode documents and to serialize data. HTML was design to format and display data, whereas XML is to store and transport data. The content and structure of XML documents are accessed by a software module, called XML processor.

Similar to HTML documents, data are marked by tags in XML. These tags are not predefined rather user defined. XML design is self descriptive and follows W3C recommendation. Since more such tags may be defined in XML, it is said to be extensible.

An XML has two correctness levels.

1. Well-formed. A well-formed document conforms to the XML syntax rules; i.e. a start-tag (< >) must corresponds with an end-tag (</>).
2. Valid. A valid document additionally conforms to semantic rules, either user-defined or an XML schema, especially DTD.

XML documents must contain a root element. This element is 'Parent' of all other elements. The elements in an XML document form a document tree. The tree starts at the root and branches to the lowest level of the tree.

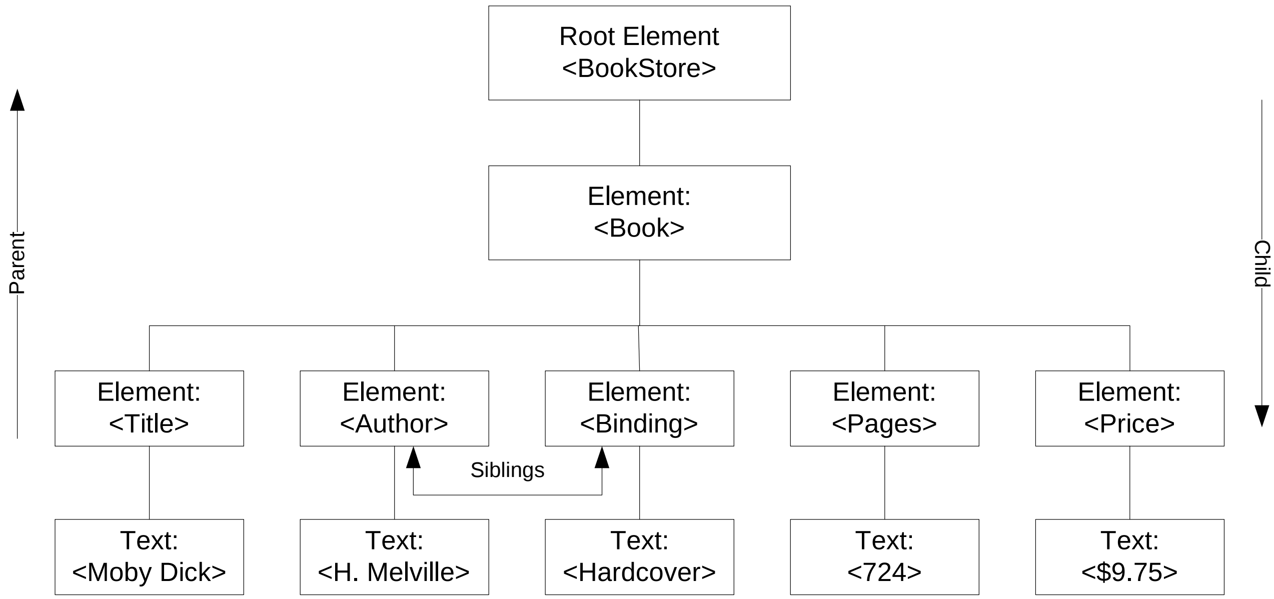


Fig. 1 XML Tree

Figure 1 depicts a XML Tree. BookStore is the root element. All <Book> elements in the document are contained within <BookStore>. The <Book> element has 5 children Title, Author, Binding, Pages and Price; they are siblings to each other.

Elements <BookStore> and <Book> have element contents and Elements <Title>, <Author> etc. has text element.

Elements may have an attribute (not shown in the figure) for e.g. <Book> can have an attribute (category="Children").

Let us create the XML file described above with five books.

Experiment 1: Creating BookStore.xml file.

Step 1 Open Notepad or any or any other suitable application and type the following code. (I have used Crimson Editor and Macromedia Dreamweaver.)

```xml
<!-- BookStore.xml-->
<?xml version="1.0"?>
<BookStore>
    <Book>
        <Title> Moby-Dick </Title>
        <Author> H. Melville </Author>
        <Binding> HardBidning </Binding>
        <Pages> 724 </Pages>
        <Price> $9.75 </Pages>
    </Book>

    <Book>
        <Title> Godaan </Title>
        <Author> Munshi Premchand </Author>
        <Binding> Paperback </Binding>
        <Pages> 245 </Pages>
        <Price> $7.5 </Price>
    </Book>

    <Book>
        <Title> Geetanjali </Title>
        <Author> R. N. Tagore </Author>
        <Binding> Paperback </Binding>
        <Pages> 125 </Pages>
        <Price> $5.75 </Price>
    </Book>

    <Book>
        <Title> Teesta Parer Britanto </Title>
        <Author> Debesh Roy </Author>
        <Binding> Hardbinding </Binding>
        <Pages> 475 </Pages>
        <Price> $19.5 </Price>
    </Book>

    <Book>
        <Title> The adventure of Huckleberry Finn </Title>
        <Author> Mark Twain </Author>
        <Binding> Paperback </Binding>
        <Pages> 290 </Pages>
        <Price> $6.95 </Price>
    </Book>
</BookStore>
```

Points to be noted regarding Well-formed XML:

- Am XML Document must have one and exactly one root element.
- All tags must be closed.
- All tags must be properly nested.
- XML tags are case-sensitive.
- Attributes must always be quoted.
- Reserve word cannot be used in XML documents

Step 2 Save the file as BookStore.xml

Step 3 Open this file using any web browser.

The file in browser will look like Fig. 2.

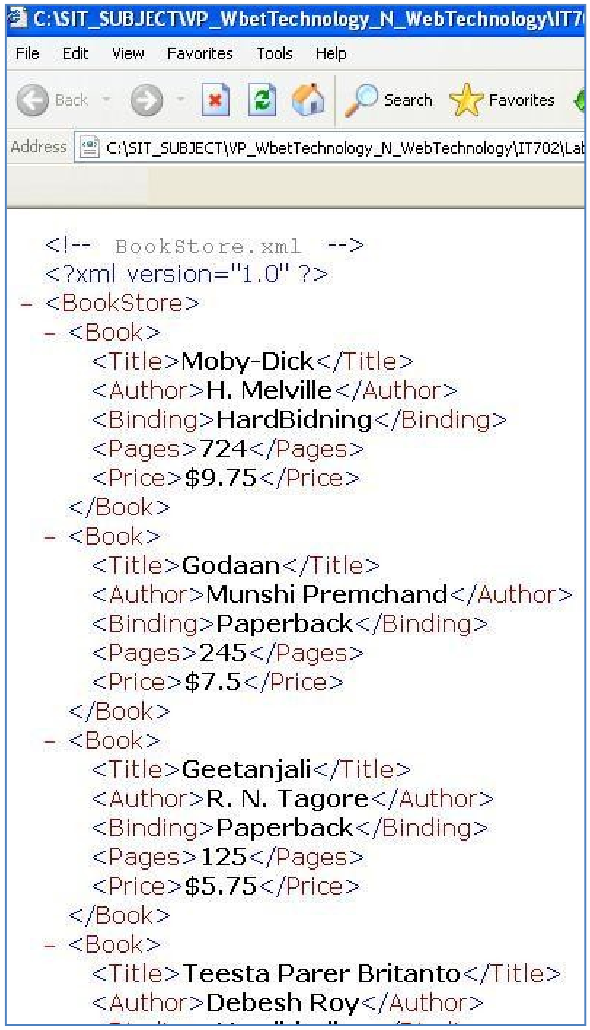


Fig. 2 XML file opened with a browser.

Experiment 2: Crating CSS file to display .XML file in formatted output.
Step 1 Type the following code and save it as BookSotre.css. (Fig. 3)

```
Book
{
    display:block;
    margin-top:12pt;
    font-size:10pt
}

Title
{
    font-style:italic ;
}

Author
{
    font-weight:bold
}
```

Fig. 3 Code for BookStore.css

Step 2 Open BookStore.xml file (you have already created in Experiment 1) and add the following code in Line 2 (Fig. 4): <?xml-stylesheet type=”text/css” href=”BookStore.css”?>

```
1 <!-- BookStore.xml-->
2 <?xml version="1.0"?>
3 <?xml-stylesheet type="text/css" href="BookStore.css"?>
4 <BookStore>
```

Fig. 4 BookStore.xml file; now referencing BookStore.css to display in a formatted way.

Step 2 Now open BookStore.xml file using any web browser. The output will look like the following figure (Fig. 5).

Fig. 5 BookStore.xml is now displayed in a formatted manner

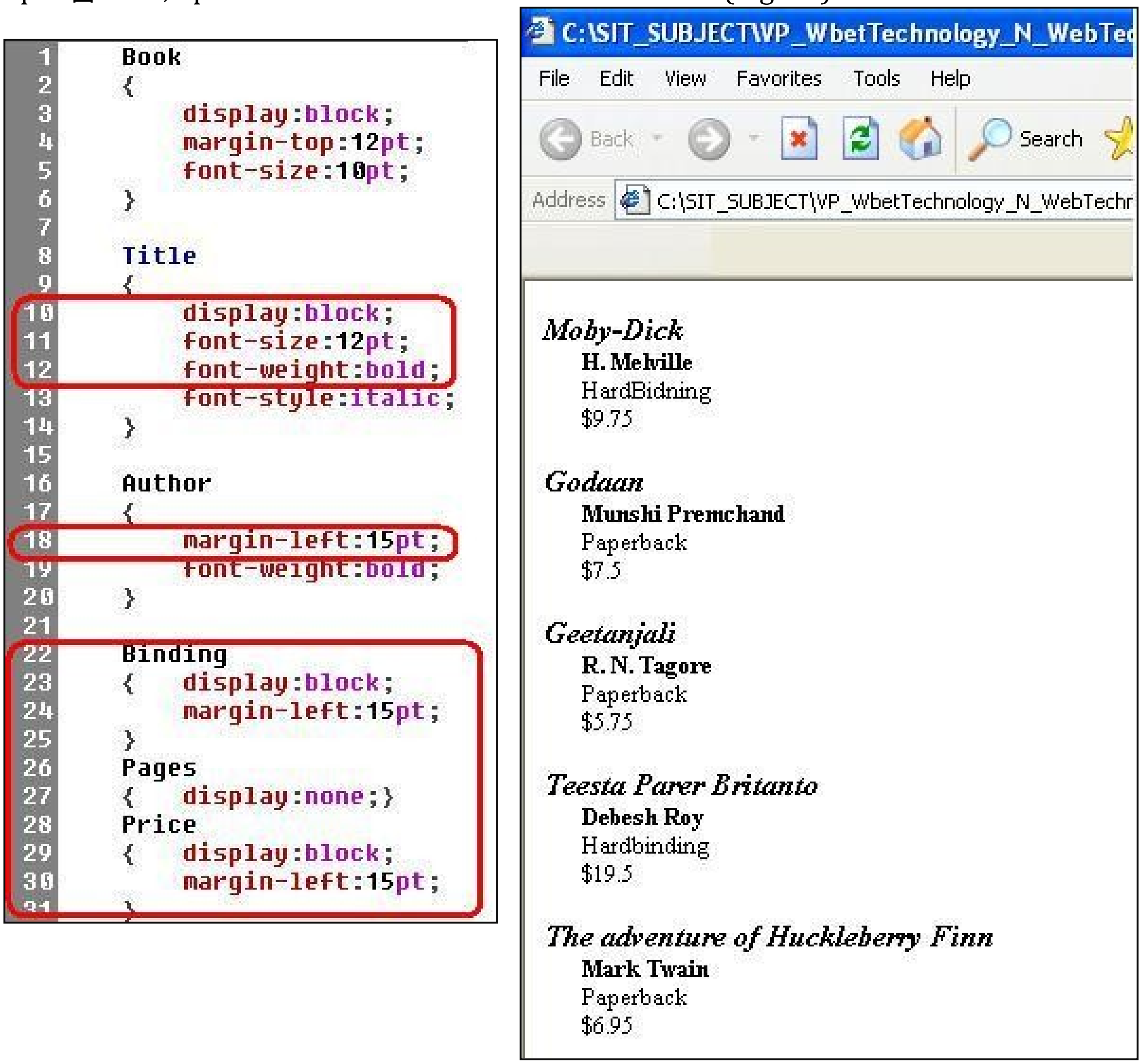
Question: Can you say why each Book is displayed in different paragraph?

Experiment 3: Using different format.

Step 1 Open BookStore.css and the following tags in line 10, 11, 12, 18, 22-31 (Fig. 6a). Here each element will be displayed in block except the Element Pages; as it is set as none.

Step 2 Save BookStore.css.

Step 3 Now, open BookStore.xml file and see the effect (Fig. 6b).



```
1   Book
2   {
3       display:block;
4       margin-top:12pt;
5       font-size:10pt;
6   }
7
8   Title
9   {
10      display:block;
11      font-size:12pt;
12      font-weight:bold;
13      font-style:italic;
14  }
15
16  Author
17  {
18      margin-left:15pt;
19      font-weight:bold;
20  }
21
22  Binding
23  {   display:block;
24      margin-left:15pt;
25  }
26  Pages
27  {   display:none;}
28  Price
29  {   display:block;
30      margin-left:15pt;
31  }
```


Fig. 6 a) CSS code b) XML formatted output

Experiment 4.1 Viewing XML file using Extended Style Sheet (XSL)

Step 1 Create the following XSL file and save it as display.xsl

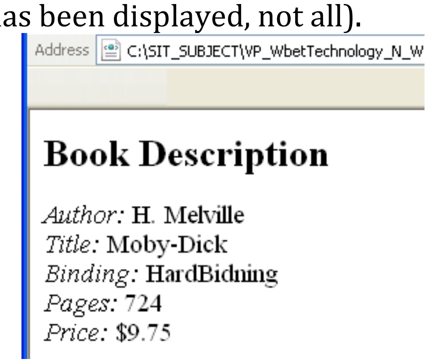
```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl"/>
<xsl:template match="/">
<h2> Book Description </h2>

<span style="font-style:italic"> Author: </span>
<xsl:value-of select="BookStore/Book/Author"/>
<br/>

<span style="font-style:italic"> Title: </span>
<xsl:value-of select="BookStore/Book/Title"/>
<br/>

<span style="font-style:italic"> Binding: </span>
<xsl:value-of select="BookStore/Book/Binding"/>
<br/>

<span style="font-style:italic"> Pages: </span>
<xsl:value-of select="BookStore/Book/Pages"/>
<br/>

<span style="font-style:italic"> Price: </span>
<xsl:value-of select="BookStore/Book/Price"/>
<br/>

</xsl:template>
</xsl:stylesheet>
```

display.xsl

Step 2 Open the BookStore.xml and modify line 3 to link display.xsl. Save the file as BookStore1.xml (just to make a separate file).

```
1 <!-- BookStore.xml-->
2 <?xml version="1.0"?>
3 <?xml-stylesheet type="text/xsl" href="display.xsl"?>
4 <BookStore>
5     <Book>
6         <Title> Moby-Dick </Title>
   .
   .
```

Step 3 Open BookStore1.xml using any web browser. Output will be as follows. (Note that only one element has been displayed, not all).



Experiment 4.2 Displaying all records from XML file using Extended Style Sheet (XSL)

Step 1 Open display.xsl and modify codes as given below and save it as DisplayAll.xsl. Note the modifications on lines 6, 29 and 9,13,17,21,25

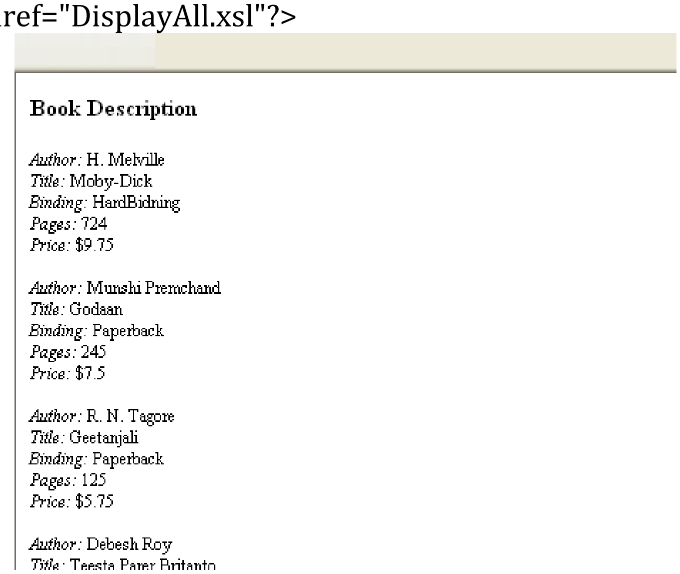
```
1 <?xml version="1.0"?>
2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
3 <xsl:template match="/">
4 <h2> Book Description </h2>
5
6 <xsl:for-each select ="BookStore/Book">
7
8 <span style="font-style:italic"> Author: </span>
9 <xsl:value-of select="Author"/>
10 <br/>
11
12 <span style="font-style:italic"> Title: </span>
13 <xsl:value-of select="Title"/>
14 <br/>
15
16 <span style="font-style:italic"> Binding: </span>
17 <xsl:value-of select="Binding"/>
18 <br/>
19
20 <span style="font-style:italic"> Pages: </span>
21 <xsl:value-of select="Pages"/>
22 <br/>
23
24 <span style="font-style:italic"> Price: </span>
25 <xsl:value-of select="Price"/>
26 <br/>
27 <br/>
28
29 </xsl:for-each>
30 </xsl:template>
31 </xsl:stylesheet>
```

DisplayAll.xsl

Step 2 Open the BookStore1.xml and modify line 3 to link DisplayAll.xsl as:

<?xml-stylesheet type="text/xsl" href="DisplayAll.xsl"?>

Step 3 Open BookStore1.xml using any web browser. Output will be as follows. Here all the elements available in BookStore1.xml have been displayed.



Experiment 4.3 Displaying all records from XML along with **Filtering and Sorting.**

Step 1 Open DisplayAll.xsl and modify line 6 as <xsl:for-each select ="BookStore/Book[Binding='Paperback']" order-by="+Author">

Step 2 Open BookStore1.xml using any web browser. All the books with Paperback binding will be displayed in ascending order of Author.

Experiment 4.4 Displaying all records **in a table** using XSL file.

Step 1 Open DisplayAll.xsl and modify code as follows. Save file as DispalyTable.xsl.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
<h2> Book Stock </h2>

<table border="2" cellpadding="5">
<thead>
    <th> Title </th>
    <th> Author </th>
    <th> Pages </th>
    <th> Binding </th>
    <th> Price </th>
</thead>
<xsl:for-each select ="BookStore/Book">
  <tr align="center">
    <td> <xsl:value-of select="Title"/> </td>
    <td> <xsl:value-of select="Author"/> </td>
    <td> <xsl:value-of select="Binding"/> </td>
    <td> <xsl:value-of select="Pages"/> </td>
    <td> <xsl:value-of select="Price"/> </td>
  </tr>
</xsl:for-each>
</table>
</xsl:template>
</xsl:stylesheet>
```

DisplayTable.xsl

Step 2 Open the BookStore1.xml and modify line 3 to link DisplayTable.xsl as:

<?xml-stylesheet type="text/xsl" href="DisplayTable.xsl"?>

Step 2 Open BookStore1 using any web browser. Output will be as follows:-

Book Stock

Title	Author	Pages	Binding	Price
Moby-Dick	H. Melville	HardBidning	724	$9.75
Godaan	Munshi Premchand	Paperback	245	$7.5
Geetanjali	R. N. Tagore	Paperback	125	$5.75
Teesta Parer Britanto	Debesh Roy	Hardbinding	475	$19.5
The adventure of Huckleberry Finn	Mark Twain	Paperback	290	$6.95

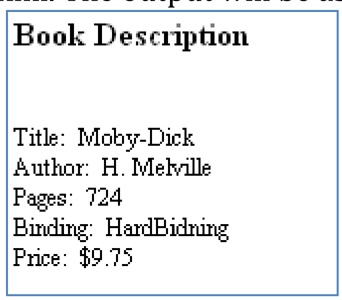
XML Data Binding

Experiment 5.1 Displaying single Record.

Step 1 Create this HTML file for embedding the data in BookStore.xml. Save the file as DataBind.xml

```
<html>
    <head>
        <title> Book Invenroty </title>
    </head>
    <body>
        <xml src="BookStore.xml" id="BS"></xml>
        <h2> Book Description </h2>
        <br />  Title: <span datasrc="#BS" datafld="Title"> </span>
        <br />  Author: <span datasrc="#BS" datafld="Author"> </span>
        <br />  Pages: <span datasrc="#BS" datafld="Pages"> </span>
        <br />  Binding: <span datasrc="#BS" datafld="Binding"> </span>
        <br />  Price: <span datasrc="#BS" datafld="Price"> </span>
    </body>
</html>
```

Step 2 Open the file DataBind.xml. The output will be as follows:-



Experiment 5.1 Navigation between records using buttons.

Step 1 Open DataBind.html and modify it as follows.

```
<html>
    <head>
        <title> Book Invenroty </title>
    </head>
    <body>
        <xml src="BookStore.xml" id="BS"></xml>
        <h2> Book Description </h2>
        <br />  Title: <span datasrc="#BS" datafld="Title"> </span>
        <br />  Author: <span datasrc="#BS" datafld="Author"> </span>
        <br />  Pages: <span datasrc="#BS" datafld="Pages"> </span>
        <br />  Binding: <span datasrc="#BS" datafld="Binding"> </span>
        <br />  Price: <span datasrc="#BS" datafld="Price"> </span>

        <p>
        <button onClick="BS.recordset.moveFirst()"> &lt; &lt; First </button>
        <button onClick="BS.recordset.movePrevious()"> &lt; Previous </button>
        <button onClick="BS.recordset.moveNext()"> &gt; Next </button>
        <button onClick="BS.recordset.moveLast()"> &gt; &gt; Last </button>
    </body>
</html>
```

Step 2 Open the file DataBind.xml. You can navigate records using buttons

Assignments on XML

1. Write a XML program that will create an XML document which contains your mailing address.

2. Write a XML program that will create an XML document which contains description of three book category.

3. Create an XML document that contains the name and price per pound of coffee beans.

i) In your XML document mention all properties of XML declaration.

ii) The root element has name <coffee_bean>

iii) Create nested elements for different types of coffee.

iv) Validate the document and if any parsing error is present, fix them.

4. Create an XML document that contains airline flight information.

i) In your XML document mention all properties of XML declaration.

ii) The root element has name <airlines>

iii) Create three nested <carrier> elements for three separate airlines. Each element should include a name attribute.

iv) Within each <carrier> nest at least two <flight> ,each of which contains departure_city, destination_ city, fl_no, dept_time.

v) Validate the document and if any parsing error is present fix them.

5. Create an XML version of your resume. Include elements such as your name and position desired. Nest each of your former employers within an <employer> element. Also, nest your educational experience within an <education> element. Create any other nested elements that you deem appropriate, such as <references> or <spcl_skills> elements.

6. Create a DTD on product catalog.

Applet Programming

8.1 How to create a basic Applet?

Solution:

Following example demonstrates how to create a basic Applet by extending Applet Class. You will need to embed another HTML code to run this program.

```
import java.applet.*;
import java.awt.*;

public class Main extends Applet{
   public void paint(Graphics g){
      g.drawString("Welcome in Java Applet.",40,20);
   }
}
```

Now compile the above code and call the generated class in your HTML code as follows:

```
<HTML>
```

```
<HEAD>
</HEAD>
<BODY>
<div >
<APPLET CODE="Main.class" WIDTH="800" HEIGHT="500">
</APPLET>
</div>
</BODY>
</HTML>
```

Result: Welcome in Java Applet.

8.2 How to create a banner using Applet?

Solution:

Following example demonstrates how to play a sound using an applet image using Thread class. It also uses drawRect(), fillRect(), drawString() methods of Graphics class.

```
import java.awt.*;
import java.applet.*;

public class SampleBanner extends Applet
implements Runnable{
   String str = "This is a simple Banner ";
   Thread t ;
   boolean b;
   public void init() {
      setBackground(Color.gray);
      setForeground(Color.yellow);
   }
   public void start() {
      t = new Thread(this);
      b = false;
      t.start();
   }
   public void run () {
      char ch;
      for( ; ; ) {
      try {
         repaint();
         Thread.sleep(250);
         ch = str.charAt(0);
         str = str.substring(1, str.length());
         str = str + ch;
      }
      catch(InterruptedException e) {}
      }
   }
   public void paint(Graphics g) {
      g.drawRect(1,1,300,150);
      g.setColor(Color.yellow);
      g.fillRect(1,1,300,150);
      g.setColor(Color.red);
```

```
      g.drawString(str, 1, 150);
   }
}
```

8.3 How to go to a link using Applet?
Solution:

Following example demonstrates how to go to a particular webpage from an applet using showDocument() method of AppletContext class.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.awt.event.*;

public class tesURL extends Applet implements ActionListener{
   public void init(){
      String link = "yahoo";
      Button b = new Button(link);
      b.addActionListener(this);
      add(b);
   }
   public void actionPerformed(ActionEvent ae){
      Button src = (Button)ae.getSource();
      String link = "http://www."+src.getLabel()+".com";
      try{
         AppletContext a = getAppletContext();
         URL u = new URL(link);
         a.showDocument(u,"_self");
      }
      catch (MalformedURLException e){
         System.out.println(e.getMessage());
      }
   }
}
```

Assignments

- Display clock using Applet
- Create different shapes using Applet
- Goto a link using Applet
- Display image using Applet
- Open a link in a new window using Applet
- Play sound using Applet

References:

1. Perl Programming, Larry Wall and Randal L. Schwartz, Oreilly
2. JavaScript: The Definitive Guide, David Flanagan, Oreilly
3. Java Programming, Herbert Schildt, PHI
4. HTML & CSS: Design and Build Web Sites, Jon Duckett