Contents

[Contents 1](#_Toc515501149)

[1 SHELL Scripting 2](#_Toc515501150)

[1.1 Environment Variable 2](#_Toc515501151)

[1.1.1 Example 2](#_Toc515501152)

[1.2 " if " conditional commands 3](#_Toc515501153)

[1.2.1 Syntax 3](#_Toc515501154)

[1.2.2 Example 3](#_Toc515501155)

[2 IPCs(Inter Process Communication) 4](#_Toc515501156)

[2.1 Message Queues 4](#_Toc515501157)

[2.2 System calls used for message queues: 4](#_Toc515501158)

[2.3 MESSAGE QUEUE PROGRAM FOR WRITER PROCESS 5](#_Toc515501159)

[2.4 MESSAGE QUEUE PROGRAM FOR READER PROCESS 6](#_Toc515501160)

[3 Multithreading(POSIX) 7](#_Toc515501161)

[3.1 What Is a Thread? 7](#_Toc515501162)

[3.2 pthread\_create() 7](#_Toc515501163)

[3.2.1 SYNOPSIS 7](#_Toc515501164)

[3.2.2 PARAMETERS 7](#_Toc515501165)

[3.2.3 Program 8](#_Toc515501166)

[3.2.4 Run The Program 8](#_Toc515501167)

[3.2.5 Output 8](#_Toc515501168)

[4 Makefile 9](#_Toc515501169)

[4.1 Make Tool: Syntax Overview 9](#_Toc515501170)

[4.1.1 make command syntax is: 9](#_Toc515501171)

[4.2 Basic Syntax of Makefiles 9](#_Toc515501172)

[4.2.1 More Special Characters 10](#_Toc515501173)

[5 Networking 11](#_Toc515501174)

[5.1 Client-Server Communication 11](#_Toc515501175)

[5.2 TCP 3-Way HandShake 11](#_Toc515501176)

[6 Memory Layout of C Programs 12](#_Toc515501177)

[7 The Build Process - C/C++ 13](#_Toc515501178)

[8 Important Questions & Answers in UNIX Operating System 14](#_Toc515501179)

[8.1 How to create Static library & Dynamic Library and Use it in Linux? 14](#_Toc515501180)

[8.1.1 Static Library Creation 14](#_Toc515501181)

[8.1.2 Dynamic Library Creation 16](#_Toc515501182)

# SHELL Scripting

## Environment Variable

|  |  |
| --- | --- |
| **Variable name** | **Purpose** |
| $0 | the name of the shell program |
| $1 thru $9 | the first thru to ninth parameters |
| $# | the number of parameters |
| $\* | all the parameters passed represented as a single word with individual parameters separated |
| $@ | all the parameters passed with each parameter as a separate word |
| $? | hold the exit status of the previous command |
| $$ | the process id of the current process |

### Example

shift.sh

#!/bin/sh

echo $@

echo $# $\*

shift

echo $# $\*

shift

echo $# $\*

shift

echo Script Name is $0

echo $?

echo $$

**Output**

./shift.sh hello jyoti ranjan

hello jyoti ranjan

3 hello jyoti ranjan

2 jyoti ranjan

1 ranjan

Script Name is ./shift.sh

0

52248

## Compare Numbers, Strings and Files in Bash Shell Script

### Numeric Comparisons

* num1 -eq num2                  check if 1st  number is equal to 2nd number
* num1 -ge num2                  checks if 1st  number  is greater than or equal to 2nd number
* num1 -gt num2                  checks if 1st  number is greater than 2nd number
* num1 -le num2                   checks if 1st number is less than or equal to 2nd number
* num1 -lt num2                   checks if 1st  number  is less than 2nd number
* num1 -ne num2                  checks if 1st  number  is not equal to 2nd number

#!/bin/bash

# Script to do numeric comparisons

var1=10

var2=20

if [ $var2 -gt $var1 ]

then

echo "$var2 is greater than $var1"

fi

# Second comparison

if [ $var1 -gt 30 ]

then

echo "$var is greater than 30"

else

echo "$var1 is less than 30"

fi

### Strings Comparisons

* var1 = var2     checks if var1 is the same as string var2
* var1 != var2    checks if var1 is not the same as var2
* var1 < var2     checks if var1 is less than var2
* var1 > var2     checks if var1 is greater than var2
* -n var1             checks if var1 has a length greater than zero
* -z var1             checks if var1 has a length of zero

when using **“<” and “>”**in scripts, they should be used with escape character i.e. use it as **“\>” or “\<“.**

**Or**

For **string comparison** we can use **double square brackets** if [[ $var1 < $var2 ]]. And for **integer comparison** we can use **double brackets** if (( $var1 > $var2 )).

Here we only used ‘-n’ parameter but we can also use “-z“. The only difference is that with ‘-z’, it searches for string with zero length while “-n” parameter searches for value that is greater than zero.

#!/bin/sh

a="abc"

b="efg"

if [ $a = $b ]

then

echo "$a = $b : a is equal to b"

else

echo "$a = $b: a is not equal to b"

fi

if [ $a != $b ]

then

echo "$a != $b : a is not equal to b"

else

echo "$a != $b: a is equal to b"

fi

if [ -z $a ]

then

echo "-z $a : string length is zero"

else

echo "-z $a : string length is not zero"

fi

if [ -n $a ]

then

echo "-n $a : string length is not zero"

else

echo "-n $a : string length is zero"

fi

if [ $a ]

then

echo "$a : string is not empty"

else

echo "$a : string is empty"

fi

### File comparison

* -d file                    checks if the file exists and is it’s a directory
* -e file                    checks if the file exists on system
* -w file                    checks if the file exists on system and if it is writable
* -r file                     checks if the file exists on system and it is readable
* -s file                    checks if the file exists on system and it is not empty
* -f file                     checks if the file exists on system and it is a file
* -O file                   checks if the file exists on system and if it’s is owned by the current user
* -G file               checks if the file exists and the default group is the same as the current user
* -x file                    checks if the file exists on system and is executable
* file A -nt file B       checks if file A is newer than file B
* file A -ot file B       checks if file A is older than file B

#!/bin/bash

# Script to check file comparison

dir=/mnt/f/Study/UBUNTU/shell\_scripting

if [ -d $dir ]

then

echo "$dir is a directory"

cd $dir

ls -a

else

echo "$dir is not exist"

fi

**1:33:59**

## " if " conditional commands

In the following

command represents a single command,

command-list represents one or more commands,

The numbers 1 and 2 are used to indicate different commands.

### Syntax

if command1

then

command-list

fi

------------------------------------------------------

if command1

then

command-list1

elif command1

command-list2

fi

-------------------------------------------------------

if command1

then

command-list1

else

command-list2

fi

### Example

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

elif [ $a -gt $b ]

then

echo "a is greater than b"

elif [ $a -lt $b ]

then

echo "a is less than b"

else

echo "None of the condition met"

fi

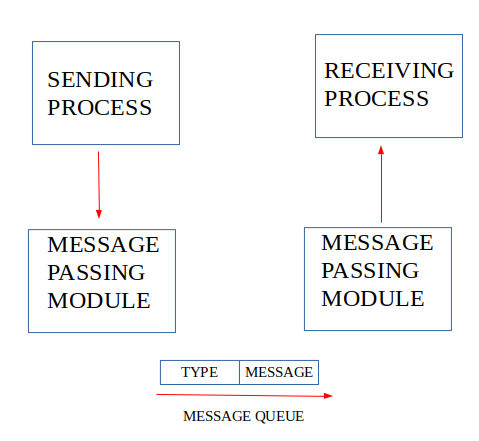
**Output**

a is less than b

# IPCs(Inter Process Communication)

## Message Queues

* **Message Queues** allow one or more processes to write messages which will be read by one or more processes.
* Message Queues allow you to take packets out of order in some cases. Each message has a message type associated with it. A Message Queue reader can specify which type of message that it will read. Or it can say that it will read all messages in order.
* It is quite possible to have any number of Msg Queue readers, or writers. In fact the same process can be both a writer and a reader.
* One real bad characteristic: Msg Queues are based on a system buffer resource. It is possible for one process to let its messages pile up. This can result in all processes on a given system to be hung up because the system is out of resources. This is bad news when it happens.
* There is a limit to the size of each packet and there is a limit to the total number of bytes that can show up in any given Message Queue.



## System calls used for message queues:

|  |  |
| --- | --- |
| **System calls** | **Description** |
| **ftok()** | Use to generate a unique key. |
| **msgget()** | Either returns the message queue identifier for a newly created message  queue or returns the identifiers for a queue which exists with the same key value. |
| **msgsnd()** | Data is placed on to a message queue by calling msgsnd(). |
| **msgrcv()** | Messages are retrieved from a queue. |
| **msgctl()** | It performs various operations on a queue. Generally it is use to destroy message queue. |

## MESSAGE QUEUE PROGRAM FOR WRITER PROCESS

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAXSIZE 128

//We need to create a structure for sending the message

//and to decide on type of text

struct msgbuf

{

long mtype; // For message type: i.e. From which process to which process

char mtext[MAXSIZE]; // The text data that we are going to send

} msg; // An instance of the structure

int main(){

int msgid; //Every message is represented by an id which we access the nessages

key\_t key; //Every queue needs a key, which the sender and reciver will agree

//for the communication

/////////////////////STAGE 1 of PROGRAM//////////////////

**if ((key = ftok("msgq\_send.c", 'b')) == -1){**

perror("key");//if not created perror function will tell why it has not

//been created

exit(1);

}

/////////////////////STAGE 2 of PROGRAM//////////////////

**if ((msgid = msgget(key, 0644 | IPC\_CREAT)) == -1){**

perror("msgid");

exit(1);

}

printf("\n The msgid is %d", msgid);

printf("\n Enter the text here\n");

msg.mtype = 1;// An agreement between the sender and reciver processes.

while (gets(msg.mtext), !feof(stdin)){

/////////////////////STAGE 3 of PROGRAM//////////////////

**if (msgsnd(msgid, &msg, sizeof(msg), 0) == -1){**

perror("msgsnd");

exit(1);

}

}

/////////////////////STAGE 4 of PROGRAM//////////////////

**if (msgctl(msgid, IPC\_RMID, NULL) == -1){**

perror("msgctl");

exit(1);

}

return 0;

}

## MESSAGE QUEUE PROGRAM FOR READER PROCESS

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAXSIZE 128

//We need to create a structure for sending the message

//and to decide on type of text

struct msgbuf

{

long mtype; // For message type: i.e. From which process to which process

char mtext[MAXSIZE]; // The text data that we are going to send

} msg;

int main(){

//msgbuf msg; // An instance of the structure

int msgid; //Every message is represented by an id which we access the nessages

key\_t key; //Every queue needs a key, which the sender and reciver will agree

//for the communication

/////////////////////STAGE 1 of PROGRAM//////////////////

**if ((key = ftok("msgq\_send.c", 'b')) == -1){**

perror("key");//if not created perror function will tell why it has not

//been created

exit(1);

}

/////////////////////STAGE 2 of PROGRAM//////////////////

if ((msgid = msgget(key, 0644 | IPC\_CREAT)) == -1){

perror("msgid");

exit(1);

}

/////////////////////STAGE 3 of PROGRAM//////////////////

for (;;){

**if (msgrcv(msgid, &msg, sizeof(msg), 1, 0) == -1){**

perror("msgsnd");

exit(1);

}

printf("%s\n", msg.mtext);

}

return 0;

}

# Multithreading(POSIX)

## What Is a Thread?

* A thread is a semi-process, that has its own stack, and executes a given piece of code.
* Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them).
* A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc.
* All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).
* Threads in the same process share:
  + Process instructions
  + Most data
  + open files (descriptors)
  + signals and signal handlers
  + current working directory
  + User and group id
* Each thread has a unique:
  + Thread ID
  + set of registers, stack pointer
  + stack for local variables, return addresses
  + signal mask
  + priority
  + Return value: errno
* pthread functions return "0" if OK.
  + Because threads within the same process share resources:
  + Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  + Two pointers having the same value point to the same data.
  + Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

## pthread\_create()

### SYNOPSIS

#include <pthread.h>

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start)(void \*), void \*arg);

### PARAMETERS

* thread

Is the location where the ID of the newly created thread should be stored, or NULL if the thread ID is not required.

* attr

Is the thread attribute object specifying the attributes for the thread that is being created. If attr is NULL, the thread is created with default attributes.

* start

Is the main function for the thread; the thread begins executing user code at this address.

* arg

Is the argument passed to start.

### Program

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FILE: Pthread\_Creation.c

\* DESCRIPTION:

\* A "hello world" Pthreads program. Demonstrates thread creation and termination.

\* AUTHOR: Jyotiranjan

\* LAST REVISED: 14/Dec/2017

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM\_THREADS 5

void \*PrintHello(void \*threadid)

{

long tid;

tid = (long)threadid;

printf("Hello World! It's me, thread #%ld!\n", tid);

pthread\_exit(NULL);

}

int main(int argc, char \*argv[])

{

pthread\_t threads[NUM\_THREADS];

int rc;

long t;

for(t=0;t<NUM\_THREADS;t++){

printf("In main: creating thread %ld\n", t);

rc = pthread\_create(&threads[t], NULL, PrintHello, (void \*)t);

printf(" thread\_ID %ld\n", threads[t]);

if (rc){

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

/\* Last thing that main() should do \*/

pthread\_exit(NULL);

}

### Run The Program

[root@localhost threads]# gcc Pthread\_Creation.c -o Pthread\_Creation -lpthread

[root@localhost threads]# ./Pthread\_Creation

### Output

In main: creating thread 0

thread\_ID 139851122005760

In main: creating thread 1

thread\_ID 139851113613056

In main: creating thread 2

thread\_ID 139851105220352

In main: creating thread 3

thread\_ID 139851096827648

In main: creating thread 4

thread\_ID 139851088434944

Hello World! It's me, thread #4!

Hello World! It's me, thread #3!

Hello World! It's me, thread #2!

Hello World! It's me, thread #1!

Hello World! It's me, thread #0!

# Makefile

**Makefiles** are special format files that help build and manage the projects automatically.

Compiling the source code files can be tiring, especially when you have to include several source files and type the compiling command every time you need to compile. Makefiles are the solution to simplify this task.

## Make Tool: Syntax Overview

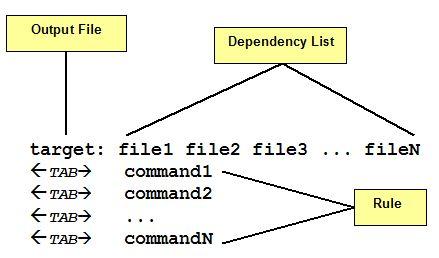
### make command syntax is:

*make [options] [target]*

The main point is **makefile** structure and how it works. **target**is a tag (or name defined) present in **makefile.** It will be described later in this article.

**make** requires a **makefile** that tells it how your application should be built. The **makefile** often resides in the same directory as other source files and it can have any name you want.

## Basic Syntax of Makefiles



A make file consists of a set of **targets**, **dependencies**and **rules**. A **target** most of time is a file to be created/updated. **target** depends upon a set of source files or even others **targets** described in **Dependency List**. **Rules** are the necessary commands to create the **target** file by using **Dependency List**.

As you see in **figure** each command in the **Rules** part must be on lines that start with a **TAB** character. Space issue errors. Also, a space at end of the **rule** line may cause **make**issues an error message.

The **makefile** is read by **make**command which determines **target** files to be built by comparing the dates and times (timestamp) of source files in **Dependency** **List**.If any dependency has a changed timestamp since the last build **make** command will execute the rule associated with the **target**.

For example, let’s assume we have the following source files.

* main.cpp
* hello.cpp
* factorial.cpp
* functions.h

**main.cpp**

#include <iostream>

using namespace std;

#include "functions.h"

int main(){

print\_hello();

cout << endl;

cout << "The factorial of 5 is " << factorial(5) << endl;

return 0;

}

**hello.cpp**

#include <iostream>

using namespace std;

#include "functions.h"

void print\_hello(){

cout << "Hello World!";

}

**factorial.cpp**

#include "functions.h"

int factorial(int n){

if (n != 1){

return(n \* factorial(n - 1));

}

else return 1;

}

**functions.h**

void print\_hello();

int factorial(int n);

**Makefile**

all: hello

hello: main.o hello.o factorial.o

g++ main.o hello.o factorial.o -o hello

main.o:main.cpp

g++ -c main.cpp

hello.o: hello.cpp

g++ -c hello.cpp

factorial.o: factorial.cpp

g++ -c factorial.cpp

clean:

rm -rf \*o hello

Command to Run the **Makefile**.

*make Or make –f Makefile or make all [target name]*

### More Special Characters

What is that $< defined in suffix rule? That means name of current dependency. In the case of .c.o suffix rule, $< is replaced by xxxxx.cpp file when rule is executed. There are others:

|  |  |
| --- | --- |
| **$?** | **list of dependencies changed more recently than current target.** |
| **$@** | **name of current target.** |
| **$<** | **name of current dependency.** |
| **$\*** | **name of current dependency without extension.** |

**makefile2**

.SUFFIX: .txt .log

file.log: file.txt

@echo "Converting " $< " to " $@

mv $< $@

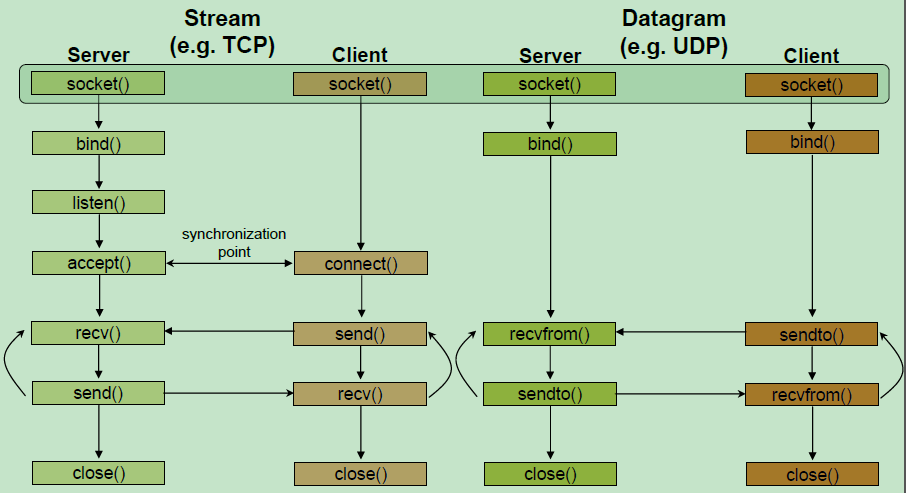
*jyoti@mohantyalone:~/study/MAKE\_FILE$* *make -f mkfile2*

*Converting file.txt to file.log*

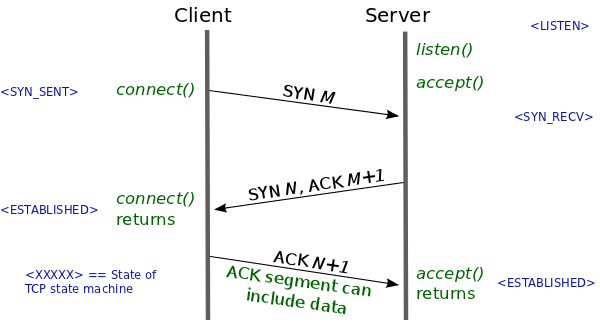
*mv file.txt file.log*

# Networking

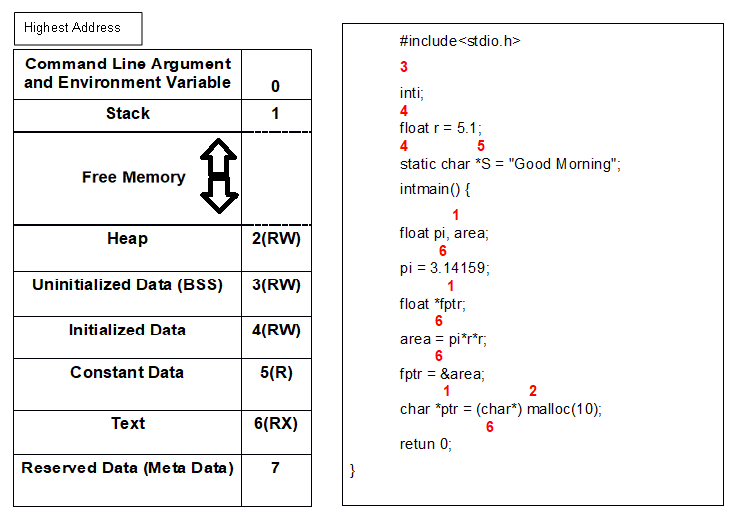
## Client-Server Communication



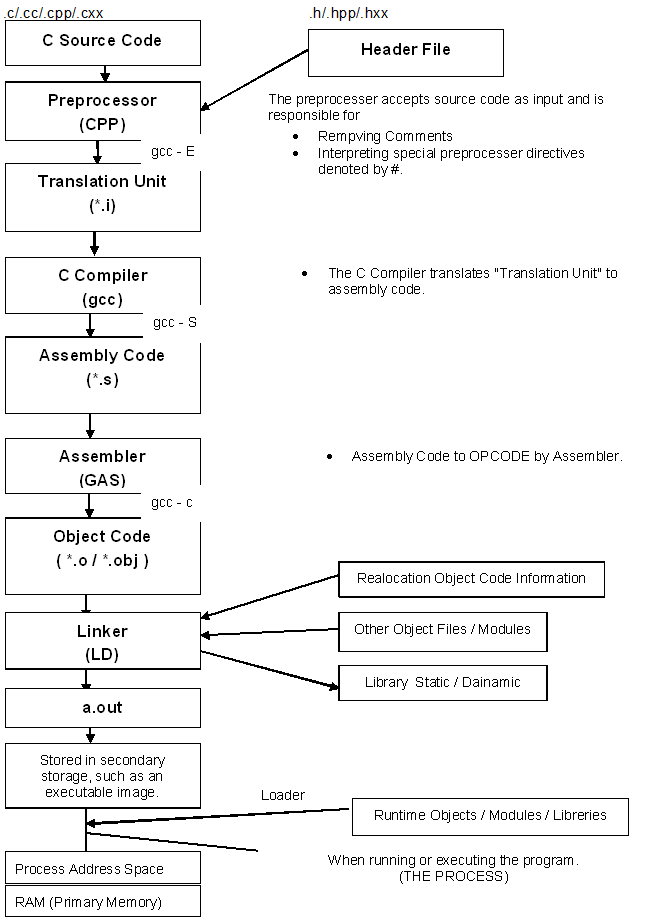
## TCP 3-Way HandShake



# Memory Layout of C Programs



# The Build Process - C/C++



# Important Questions & Answers in UNIX Operating System

## How to create Static library & Dynamic Library and Use it in Linux?

### Static Library Creation

When a C program is compiled, the compiler generates object code. After generating the object code, the compiler also invokes linker. One of the main tasks for linker is to make code of library functions (eg printf(), scanf(), sqrt(), ..etc) available to your program. A linker can accomplish this task in two ways, by copying the code of library function to your object code, or by making some arrangements so that the complete code of library functions is not copied, but made available at run-time.

**Static Linking and Static Libraries** is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are, ***.a*** files in Linux.

**Steps to create a static library** Let us create and use a Static Library in UNIX or UNIX like OS.

* Create a C files that contains functions in your library.

//File name is add.c

int add(int a, int b){

return a + b;

}

//File name is sub.c

int sub(int a, int b){

return a - b;

}

* Create a header file for the library

//File name is prac.h

int add(int, int);

int sub(int, int);

* Compile files. add.o and sub.o has been created.

*gcc -c add.c*

*gcc -c sub.c*

* Create static library. This step is to bundle multiple object files in one static library (see ar for details). The output of this step is static library.

*ar rs libprac.a add.o sub.o*

Library file name should start with lib\* and extension should be (.a).

Example :: lib\*.a

* Let us create a driver program (C file with main function) that uses above created static library.

//File name is prac.c

#include<stdio.h>

#include "prac.h" // include created header file

int main(){

printf("Addition:%d\n", add(12, 23)); //call the add() and sub() which

printf("Substraction:%d\n", sub(123, 23));//has been defined in “libprac.a”

return 0;} // static library

* Compile the driver program(C file with main function). prac.o will be created.

*gcc -c prac.c*

* Link the compiled driver program to the static library. Note that -L. is used to tell that the static library is in current folder.

*gcc -o prac prac.o libprac.a*

*OR*

*gcc -o prac -L . prac.o –lprac*

* Run the driver program

*./prac*

fun() called from a static library

OUTPUT:

*Addition:35*

*Substraction:100*

Following are some important points about static libraries.

* For a static library, the actual code is extracted from the library by the linker and used to build the final executable at the point you compile/build your application.
* Each process gets its own copy of the code and data. Where as in case of dynamic libraries it is only code shared, data is specific to each process. For static libraries memory footprints are larger. For example, if all the window system tools were statically linked, several tens of megabytes of RAM would be wasted for a typical user, and the user would be slowed down by a lot of paging.
* Since library code is connected at compile time, the final executable has no dependencies on the library at run time i.e. no additional run-time loading costs, it means that you don’t need to carry along a copy of the library that is being used and you have everything under your control and there is no dependency.
* In static libraries, once everything is bundled into your application, you don’t have to worry that the client will have the right library (and version) available on their system.
* One drawback of static libraries is, for any change (up-gradation) in the static libraries, you have to recompile the main program every time.
* One major advantage of static libraries being preferred even now “is speed”. There will be no dynamic querying of symbols in static libraries. Many production line software use static libraries even today.

### Dynamic Library Creation

We were compiling our code using the command “gcc -o sample shared.c” When we compile our code, the compiler won’t resolve implementation of the function printf(). It only verifies the syntactical checking. The tool chain leaves a stub in our application which will be filled by dynamic linker. Since printf() is standard function the compiler implicitly invoking its shared library.

**Steps to create a dynamic library (.so)** Let us create and use a Static Library in UNIX or UNIX like OS.

* Create a C files that contains functions in your library.

//File name is add.c

int add(int a, int b){

return a + b;

}

//File name is sub.c

int sub(int a, int b){

return a - b;

}

* Create a header file for the library

//File name is prac.h

int add(int, int);

int sub(int, int);

* Compile files. add.o and sub.o has been created.

#PIC – Position Independent Code

*gcc -fPIC -c add.c*

*gcc -fPIC -c sub.c*

* Create static library

*gcc -shared -o libprac.so add.o sub.o*

Dynamic Library file name should start with lib\* and extension should be (.a).

Example :: lib\*.so

* Let us create a driver program (C file with main function) that uses above created static library.

//File name is prac.c

#include<stdio.h>

#include "prac.h" // include created header file

int main(){

printf("Addition:%d\n", add(12, 23)); //call the add() and sub() which

printf("Substraction:%d\n",sub(123, 23));//has been defined in “libprac.so”

return 0;} // Dynamic library

* Compile the driver program(C file with main function). prac.o will be created.

*gcc -c prac.c*

* Link the compiled driver program to the dynamic library.

*gcc -o prac prac.o libprac.so*

* Move or Install Dynamic library to “/usr/local/lib” or “/usr/lib”

*sudo mv libprac.so /usr/local/lib*

* Configure dynamic linker run time bindings

*sudo ldconfig*

* Run the driver program

*./prac*

fun() called from a static library

OUTPUT:

*Addition:35*

*Substraction:100*