Operating System Lab

Experiment 3

Aim : Programming with gcc under Linux

Programming in C under Linux involves the use of GCC (GNU Compiler Collection). GCC is one of the best compilers available under Linux. The compiler is invoked through the command line or through any gui tools. However to properly understand the compilation, the command mode operation is the preferred one.

First we have to write a C file which is to be compiled.

file1.c
==========
```
#include<stdio.h>
int main(){
 printf("Hello World");
 return 0;
}
```

Invocation of the compiler:-

Syntax
gcc filename.c <enter>
gcc -o newname filename.c <enter>

For our program file1.c we will type “gcc file1.c <enter>”. This will produce a file named “a.out” on successful compilation. There will not be any output text displayed on screen. If there is any error, in that condition only error message will be displayed on screen.
If we want to save the “a.out” file in a different name then “-o” switch can be used. Suppose we want to save the file in the name “outfile” we will type “gcc -o outfile file1.c <enter>”

Execution of the binary:-

Syntax
./<output file's name> <enter>

So for our program it will be “./a.out” for the first case and “./outfile” in the second case.
Here “.” denotes the current directory and “/” is the separator.

Compilation steps with GCC
==========
GCC is a very clever metaprogram. It performs a lot of tasks depending on the type of input.

1. preprocessor of c will run (cpp)
   produce preprocessed code
2. code generator of c will produce assembly language code
   extension of assembly code is .s (to get the .s file type “gcc -S file.c”)
3. assembler (as) will prepare the object files
   extension of object files are .o (to get th .o file type “gcc -c file.c”)
4. linker will link together all objects and prepare application binary
   ELF binaries (no extension is required) (if -o is used a new name will be for the binary rather than the default “a.out”)

Operating System Lab

Multifile programming in C:-
Generally multifile programming requires the preparation of header file/s. Header files generally store functions.

head.h
==========
```
void one(int x){
 printf("\nthe passed value was: %d\n",x);
}

void two(int x){
 printf("\nthe passed value was: %d\n",x);
}
```

main.c
==========
```
#include<stdio.h>
#include "head.h"
int main(){
 one(1);
 two(2);
 return 0;
}
```

task 1
Prepare a header file with a factorial function and call it in the main
function of the program.

Modular Programming in C:-

When programs become very large, it is better to break them up into manageable modules or subroutines.
Driver program : A program which calls subroutines for different functions and tasks.
Stub program : programs/subroutines which are called.

Steps
compile first module with gcc -c
compile second module with gcc -c
compile any other modules in the similar fashion.
object files will be generated.

compile the driver program and pass the objects as arguments

Example:-
==========
File in use: Stub no 1 “one.c”
==========
```
void one(int x){
 printf("\nin module1\nthe passed value was: %d\n",x);
}
```

File in use: Stub no 2 “two.c”
==========
```
void two(int x){
 printf("\nin module2\nthe passed value was: %d\n",x);
}
```
File in use: Driver “new.c”

==========

```
#include<stdio.h>
int main(){
 one(1);
 two(2);
 return 0;
}
```

Compilation : “gcc -c one.c” → “gcc -c two.c” → the “.o” files will be generated.
Linking with driver program : “gcc new.c one.o two.o”
Run by executing the binary : “./a.out <enter>”

Task 2
Prepare three modules and a driver program for addition, multiplication and substraction. Each module will perform a single function only.
Separately compile the modules and link together with the main program. The main program will only accept values and show output.

Preparing and using makefiles:-
When the programs become large and modularised it is difficult to keep track of which parts need compilation, which does not and linking all parts together etc. So Makefile can be prepared which holds all that information and “make” command will do all compilation and linking for you.

for the discussed example:-
one.c
two.c
new.c

Syntax of makefile
<target> : <sources . . . >
<tab><command . . .>

NOTE : makefile must have the name “makefile” exactly.
NOTE : Makefile must be prepared with vim otherwise it may not work.

makefile
==========

```
outfile : main.c one.o two.o
	gcc -o outfile abcd.c one.o two.o
one.o : one.c
	gcc -c one.c
two.o : two.c
	gcc -c two.c
clean :
	rm one.o two.o outfile
```

Compilation
==========
Run “make” at the commandline.
Make will prepare only the required objects and link to prepare the required binary. If the binary is up to date it will be shown on screen. To clean the binary objects the clean clause can be invoked by typing “make clean”.

Task 3
Prepare a Makefile for the problem in Task 2