Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.

Name : Adwait S Purao

UID : 2021300101

Batch : B2

Experiment no. : 2

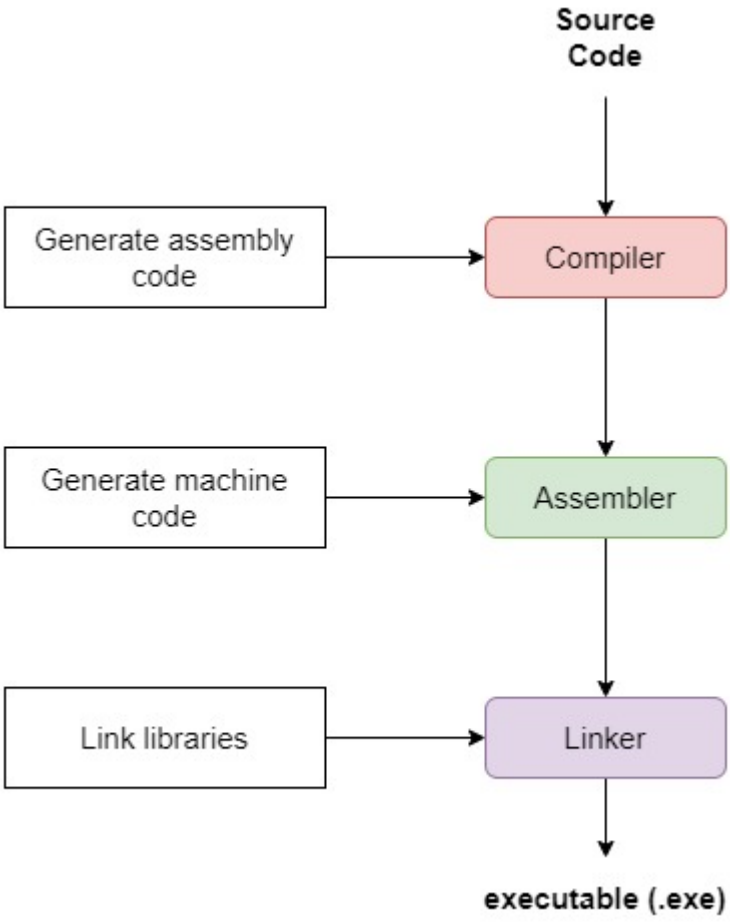
Theory:

Generating Executables

Before we start with linking methods, let's first understand the process of generating the executable of any piece of code. **Formally, a computer program is a sequence of statements in a programming language instructing the CPU what to do.**

To execute any program, we convert the source code to machine code. We do this by first compiling the code to an intermediate level and then converting it to assembly-level code. After that, we link the assembly code with other libraries or modules it uses.

So, **linking is the process of combining external programs with our program to execute them successfully.** After linking, we finally get our executable:



2.1. Linking

The assembler translates our compiled code to machine code and stores the result in an object file. It's a binary representation of our program. **The assembler gives a memory location to each object and instruction. Some of these memory locations are virtual, i.e., they're offsets relative to the base address of the first instruction.**

That's the case with references to external libraries. The linker resolves them when combining the object file with the libraries.

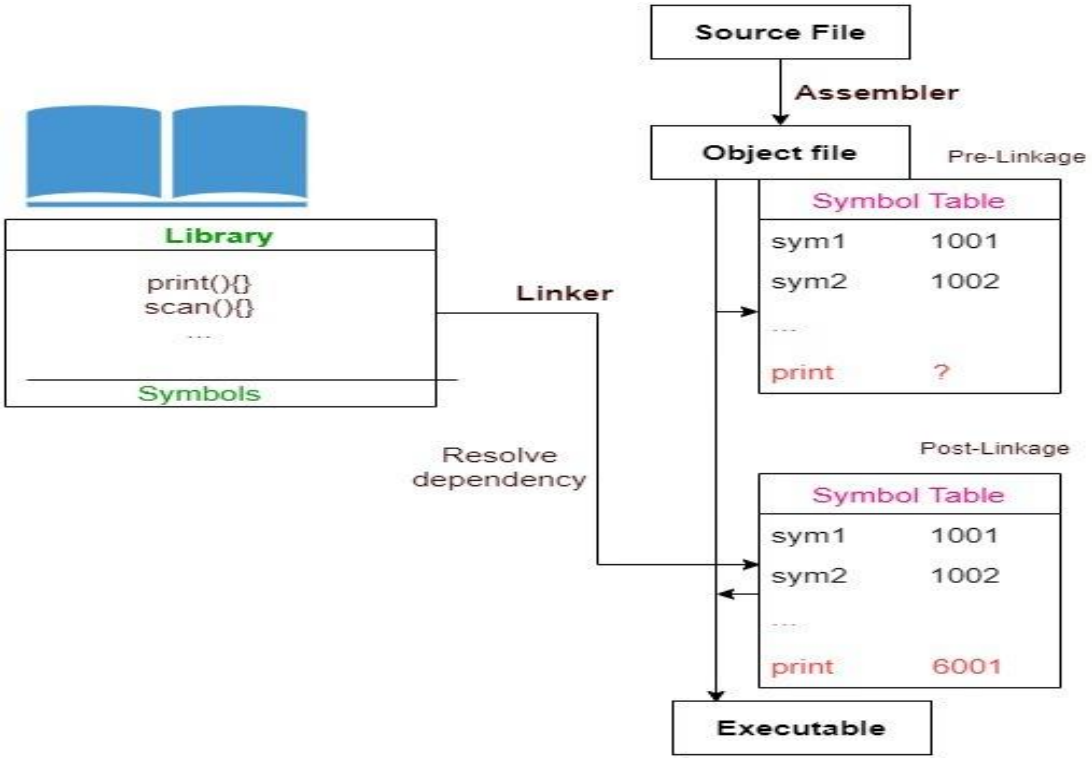
Overall, we have two mechanisms for linking:

1. Static
2. Dynamic

3. Static Linking

In static linking, the system linker copies the dependencies into the final executable. At the time of linking an external library, the linker finds all dependencies that are defined in that library. And it replaces them with the corresponding functions from the library to resolve dependencies in our code. Afterward, the linker generates the final executable file that we can execute on the underlying machine.

For example, let's say our application calls the function *print()* from an external library named *Library.* The assembler generates the object file with all native symbols resolved to their memory addresses. The external reference *print()* cannot be resolved. The linker loads this library and finds the definition of *print()* in it. Then, it maps to *print()* to a memory location and thus resolves the dependency:



So, a statistically linked file contains our program's code as well as the code of all the libraries it invokes. Since we copy complete libraries, we

need space on both the disk and in the main memory because the resulting file may be very large.

3.1. Benefits of Static Linking

Firstly, static linking ensures the application can run as a standalone binary because we integrate external libraries with it at the compile time.

Secondly, it ensures exclusivity. What does that mean?

Each statically linked process gets its copy of the code and data. So, in cases where security is very important (e.g., financial transactions), we use static linking. This is so because it completely isolates one process from another by providing each an independent environment.

Thirdly, for statically linked applications, we bundle everything into our application. Hence, we don't have to ensure that the client has the right version of the libraries available on their system

Further on, static linking offers faster execution because we copy the entire library content at compile time. Hence, we don't have to run the query for unresolved symbols at runtime. Thus, we can execute a statically linked program faster than a dynamically linked one.

3.2. When to Use Static Linking?

We use static linking where we require secure and mutually exclusive processes that don't share any code. We also use static linking for embedded projects where we want a controlled and speedy execution environment with no run-time linkage issues. Most small form factor devices such as video controllers in a mobile handset use static linking in their boot process.

4. Dynamic Linking

In dynamic linking, we copy the names of the external libraries into our final executable as unresolved symbols. We do the actual linking of these unresolved symbols only at runtime. How?

When encountering an unresolved symbol, we query RAM for it. If the corresponding library isn't loaded, the operating system loads it in the memory. So, **the operating system performs dynamic linking for us by resolving each external symbol on the first muss.** As a result, we load only a single copy of a library in memory and all processes use it.

4.1. Benefits of Dynamic Linking

In dynamic linking, we maintain only one copy of a shared library in the memory. Therefore, our program's executable file is smaller as compared to that of a statically linked one. Also, it's more memory efficient because all processes can share the library in RAM instead of using a separate copy. Similarly, all the processes sharing the library benefit from the cache too.

Dynamic linking results in a lower average load time. On average, many programs use a small number of external libraries. So, we load each library only once instead of multiple times. Thus, all the calls after the first one will take less load time. That's because only the missing unresolved symbols will be loaded in memory, but after the first loading, they'll all be there for subsequent calls.

From the deployment and maintenance perspective, dynamic linking offers us easier updating and deployment. We can update and recompile the external libraries to offer the latest changes to our programs. After recompilation, we reload new versions.

Dynamic linking promotes modularity. We use it to develop large programs that require multiple language versions having multiple modules in them. For example, the *Zoho* accounting software has many modules to implement various accounting features such as income tax, corporate tax, sales tax, etc. Each of these modules is dynamically loaded at runtime as per the user's request.

4.2. When to Use Dynamic Linking?

We use dynamic linking when we have many applications using a common set of libraries.

For example, let's suppose we have a set of microservices with each using the queue service (e.g., RabbitMQ). Then, we can optimize resources using dynamic linking.

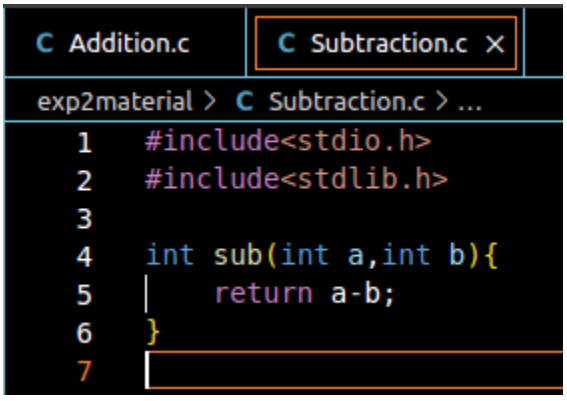
Differences Between Static and Dynamic Linking

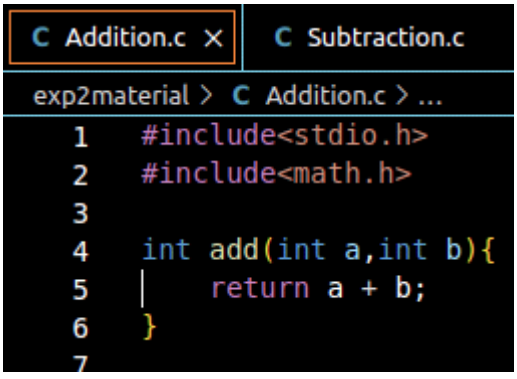
Static Linking	Dynamic Linking
Done at compile time	Done at runtime
Referenced libraries are in the binary file.	We load libraries at runtime
Larger executable files	Smaller executable files
Slower loading	Faster loading
Difficult to maintain	Easier to update

Static Linking:

Procedure (Along with sample code & output):

1. Static Library: a. Create a C file for performing any complex operation (for example: finding the area of a cylinder, factorial of a number, prime factors of a number, etc)





Addition.c | Subtraction.c | Exponent.c

exp2material > Exponent.c > expo(int)

```c
#include<stdio.h>
#include<math.h>

double expo(int x){
    x = (double)x;
    return exp(x);
}
```

b. Create a header file (Arithmetic.h and Exponent.h) for the library.

Addition.c | Subtraction.c | Arithmetic.h

exp2material > Arithmetic.h > sub(int, int)

```c
int add(int , int);
int sub(int ,int );
```

Exponent.h

exp2material > Exponent.h > expo(int)

```c
double expo(int );
```

c. Create a driver program that uses the created library

```c
#include "Arithmetic.h"
#include "Exponent.h"
#include <stdio.h>
int main()
{

int ch;
int flag=0;
do{
printf("\nEnter your choice:\n");
printf("1) Addition\n2) Subtraction\n3) Exponentiation\n4) Exit\n");
scanf("%d",&ch);
switch(ch){
```

```
case 1:
{
int a,b;
printf("Enter two numbers:\n");
scanf("%d %d",&a,&b);
printf("Addition \n%d + %d = %d\n", a, b, add(a, b));
break;

}
case 2:
{
int c,d;
printf("Enter two numbers:\n");
scanf("%d %d",&c,&d);
printf("Subtraction \n%d - %d = %d\n", c, d, sub(c, d));
break;
}
case 3:
{
int power;
printf("Enter a number for exponent:\n");
scanf("%d",&power);
printf("Exponentiation\ne ^ %d = %lf\n",power,expo(power));
break;
}
case 4:
{
printf("End of program\n");
flag=1;
break;
}
default:
{
printf("Invalid Number\n");
break;
}
}
}while(flag!=1);
return 0;
}
```

d. Creating the static library. Compiling the driver program and including the static library in it.

```
spit@spit-ThinkCentre-M70s:~/Desktop/2021300101_Adwait/exp2material$ gcc -c Addition.c -o Addition.o
spit@spit-ThinkCentre-M70s:~/Desktop/2021300101_Adwait/exp2material$ gcc -c Subtraction.c -oSubtraction.o
spit@spit-ThinkCentre-M70s:~/Desktop/2021300101_Adwait/exp2material$ gcc -c Exponent.c -o Exponent.o
```

Description of the commands above: gcc -c lib_filename.c -o lib_filename.o

This command invokes the GNU Compiler Collection (GCC) to compile the C source file "filename.c" and generates an object file "filename.o".

The "-c" flag tells GCC to compile the source code without linking it to create an object file. This object file can later be linked with other object files to create a final executable or library.

The resulting object file "filename.o" contains machine code that can be linked with other object files to create a final executable or library.

```
spspspspspspit@spit-ThinkCentre-M70s:~/Desktop/2021300101_Adwait/exp2material$ ar rcs Arithmetic.a Addition.o Subtraction.o
spit@spit-ThinkCentre-M70s:~/Desktop/2021300101_Adwait/exp2material$ ar rcs Exponent.a Exponent.o
```

Description of the command above: ar rcs filename.a filename.o

This command creates a static library named "filename.a" from one or more object files, in this case, the "filename.o" file.

The "ar" command is used to create, modify, and extract files from archives, while the "rcs" options are used to create an archive if it doesn't already exist ("-r"), update the archive with the object file ("-c"), and create an index or table of contents for the archive ("-s").

When you create a static library, the object code of each object file is combined into a single library file, allowing you to use the library in other programs without having to recompile the object files. This can result in faster program startup times and reduced memory usage.

```
spit@spit-ThinkCentre-M70s:~/Desktop/2021300101_Adwait/exp2material$ gcc -c Driver.c -o Driver.o
spit@spit-ThinkCentre-M70s:~/Desktop/2021300101_Adwait/exp2material$ gcc -o Driver Driver.o -L. -l:Arithmetic.a -l:Exponent.a -lm
spit@spit-ThinkCentre-M70s:~/Desktop/2021300101_Adwait/exp2material$ ./Driver

Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
```

Description of the command above: gcc -c Driver.c -o Driver.o

This command compiles the C source file "Driver.c" into an object file "Driver.o" using the GNU Compiler Collection (GCC).

The "-c" flag tells GCC to compile the source code without linking it to create an object file.

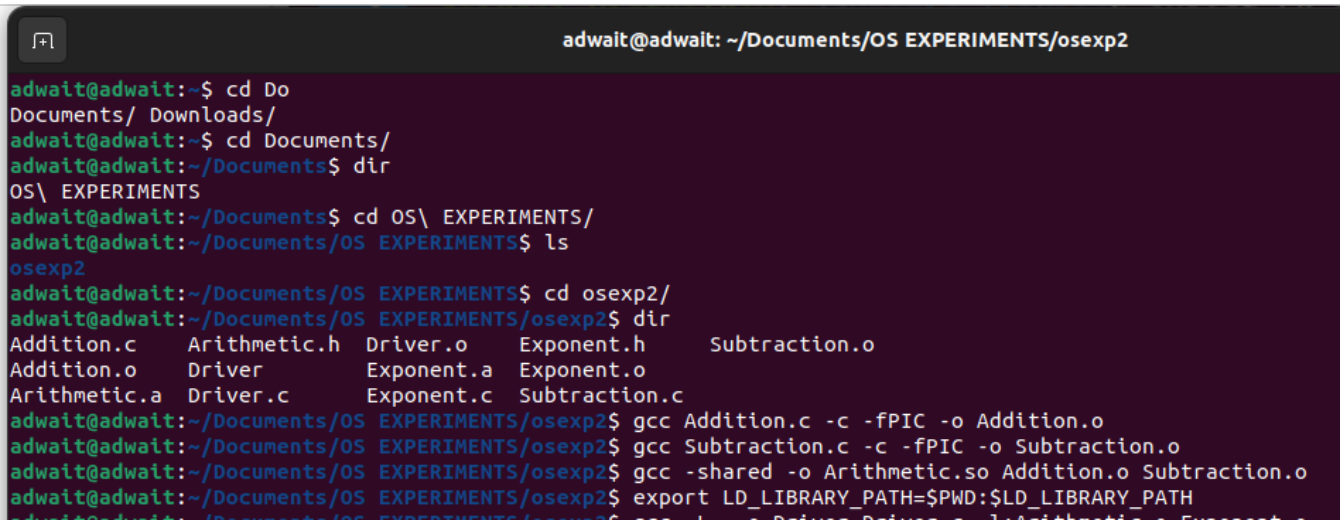
The resulting object file "Driver.o" contains machine code that can be linked with other object files, such as library files, to create a final executable program.

```
Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
1
Enter two numbers:
5 6
Addition
5 + 6 = 11

Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
2
Enter two numbers:
7 3
Subtraction
7 - 3 = 4

Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
3
Enter a number for exponent:
2
Exponentiation
e ^ 2 = 7.389056

Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
```

```
Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
4
End of program
```

2.Dynamic Linking:

a. After creating the same C, header, and driver files as above, compile the C files. Creating a dynamic library and compiling the driver program using the dynamic library.



Command : gcc filename.c -c -fPIC -o filename.o

This command compiles the C source file "filename.c" into a position-independent object file "filename.o" using the GNU Compiler Collection (GCC).

The "-c" flag tells GCC to compile the source code without linking it to create an object file, while the "-fPIC" option generates position-independent code that can be used in shared libraries.

Position-independent code allows the code to be loaded at any memory address, which is useful for shared libraries that can be loaded at different addresses in different programs.

The resulting object file "filename.o" contains machine code that can be used in creating shared libraries. It can be linked with other position-independent object files to create a shared library that can be loaded at runtime by an operating system's dynamic linker.

Command : gcc -shared -o Library.so filename.o

This command creates a shared library named "Library.so" from the position-independent object file "filename.o" using the GNU Compiler Collection (GCC).

The "-shared" option tells GCC to generate a shared library instead of an executable file.

The resulting shared library "Library.so" can be dynamically linked with other programs at runtime, providing a way to share code across different applications without needing to recompile the code into each program.

When the shared library is loaded by the operating system's dynamic linker, the linker will resolve any unresolved symbols (function names or variables) in the library with their corresponding definitions in other shared libraries or the program itself.

Command :
export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH

This command adds the current working directory ("$PWD") to the beginning of the "LD_LIBRARY_PATH" environment variable, which

is used by the operating system to locate shared libraries when dynamically linking executable programs.

The "export" command makes the "LD_LIBRARY_PATH" environment variable available to child processes spawned by the current shell.

By adding the current directory to the "LD_LIBRARY_PATH" variable, the dynamic linker will search the current directory for shared libraries before searching other directories specified in the variable. This can be useful when testing and developing shared libraries, as it allows you to quickly load new versions of the library without needing to install them system-wide. However, care should be taken to ensure that the correct version of the library is being used and that the library is not being used by other programs unintentionally.

Command : gcc -L. -o Driver Driver.c -l:Library.a

This command links the C source file "Driver.c" with a static library named "Library.a" to create an executable program named "Driver" using the GNU Compiler Collection (GCC).

The "-L" flag tells the linker to search for libraries in the current directory, and the "-l" flag specifies the name of the library to link. The ":Library.a" syntax is used to specify the name of the library file explicitly.

The resulting executable program "Driver" can be run directly without the need for any shared libraries, as all the necessary object code is included in the static library.

Static libraries can be useful when you want to distribute a self-contained executable that doesn't depend on any external libraries or

when you want to ensure that the program will always use a specific version of a library. However, they can result in larger file sizes and increased memory usage compared to dynamic libraries.

```
adwait@adwait:~/Documents/OS EXPERIMENTS/osexp2$ gcc -L. -o Driver Driver.c -l:Arithmetic.a -l:Exponent.a -lm
adwait@adwait:~/Documents/OS EXPERIMENTS/osexp2$ ./Driver

Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
```

b. Executing the driver program.

```
Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
1
Enter two numbers:
9 4
Addition
9 + 4 = 13

Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
2
Enter two numbers:
8 1
Subtraction
8 - 1 = 7

Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
3
Enter a number for exponent:
4
Exponentiation
e ^ 4 = 54.598150

Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
45
Invalid Number

Enter your choice:
1) Addition
2) Subtraction
```

```
Enter your choice:
1) Addition
2) Subtraction
3) Exponentiation
4) Exit
4
End of program
adwait@adwait:~/Documents/OS EXPERIMENTS/osexp2$
```

Conclusion:

In the above experiment we learnt about Static and Dynamic linking along with their pros and cons.
We learnt why are the used. We understood the steps and commands needed to do static and dynamic linking . Also we understood the meaning of each command.