ADDIS ABABA INSTITUTE OF

TECHNOLOGY

ITSC Department

SYSTEM PROGRAMING

ASSIGNMENT 1

## Name: Hanna Kibret

## Id: ATR/9509/10

## Section: 2

## ELF(Executable and Linkable Format)

ELF is the abbreviation for **Executable and Linkable Format** and defines the structure for binaries, libraries, and core files. The formal specification allows the operating system to interpreter its underlying machine instructions correctly.

ELF files produced by the assembler are relocatable files that hold code and/or data. They are input files for the linker. The linker combines these relocatable files with other ELF object files to create an executable file or a shared object file in the next stage of program building, after translation from source files into object files.

The ELF file type is very flexible and provides support for multiple CPU types, machine architectures, and operating systems. It is also very extensible: each file is differently constructed, depending on the required parts.

The three main kinds of ELF files are relocatable, executable and shared object files.

The assembler can also produce subordinate output incidental to the translation process. For example, if the assembler is invoked with the -V option, it can write information to standard output and to standard error.

ELF files are for execution or for linking. Depending on the primary goal, it contains the required segments or sections. Segments are viewed by the kernel and mapped into memory (using mmap). Sections are viewed by the linker to create executable code or shared objects.

## Object Files in Executable and Linking Format (ELF)

Relocatable ELF files produced by the assembler consist of:

* An ELF header
* A section header table
* Sections

The ELF header is always the first part of an ELF file. It is a structure of fixed size and format. The fields, or members, of the structure describe the nature, organization and contents of the rest of the file. The ELF header has a field that specifies the location within the file where the section header table begins.

The section header table is an array of section headers that are structures of fixed size and format. The section headers are the elements of the array, or the entries in the table. The section header table has one entry for each section in the ELF file. However, the table can also have entries (section headers) that do not correspond to any section in the file. Such entries and their array indices are reserved. The members of each section header constitute information useful to the linker about the contents of the corresponding section, if any.

All of a relocatable file's information that does not lie within its ELF header or its section header table lies within its sections. Sections contain most of the information needed to combine relocatable files with other ELF files to produce shared object files or executable files. Sections also contain the material to be combined. For example, sections can hold:

* Relocation tables
* Symbol tables
* String tables

Each section in an ELF file fills a contiguous (possibly empty) sequence of that file's bytes. Sections never overlap. However, the (set theoretic) union of a relocatable ELF header, the section header table, and all the sections can omit some of the bytes. Bytes of a relocatable file that are not in the ELF header, or in the section header table, or in any of the sections constitute the inactive space. The contents of a file's inactive space, if any, are unspecified.

## Header

The ELF **header** is always located at the beginning of the ELF file. It describes the ELF file organization and contains the actual sizes of the object file control structures.

Headers form an important part of the file, describing exactly the contents of an ELF file. By using the right tools, you can gain a basic understanding of the purpose of the file.

## Section Header

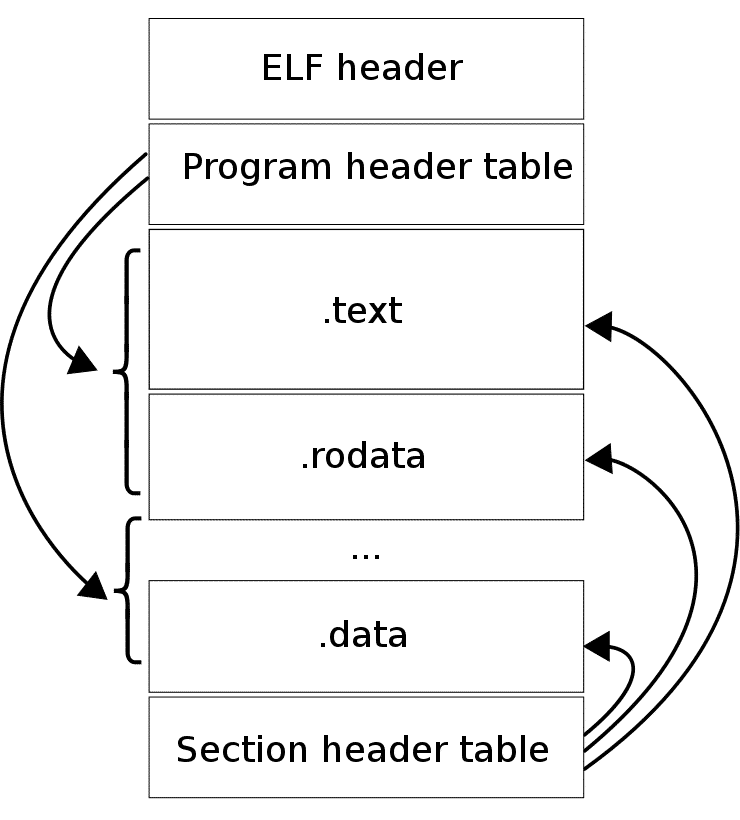
The section header table has all of the information necessary to locate and isolate each of the file's sections. A section header entry in a section header table contains information characterizing the contents of the corresponding section, if the file has such a section.

## Sections

A section is the smallest unit of an object file that can be relocated.

Sections do not need to occur in any particular order within the object file. The sections of a relocatable ELF file contain all of the file information that is not contained in the ELF header or in the section header table.

The figure below describes this schematically.



## Dynamic Loading and Linking

Static describes something done before program execution. For example, assemblers statically allocate memory for variables declared with .word, .float, .double directives.

Dynamic describes something done during program execution.

Dynamic linking and loading refers to linking and loading done during program execution. Modern operating systems typically use dynamic linking and loading for programming language library functions.

When dynamic loading a program is not loaded until it is called. This is contrast to loading all dependencies at load-time together with the main program. All routines are kept on a disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether has been loaded. If not, the relocatable linking loader is called to load the desired routine into memory and update programs address tables to reflect this change. Then control is passed to newly loaded routine. The loading process completes when the library has been successfully loaded into main memory. When the program attempts to call an imported function for the first time, that library that contains that function may or may not have been loaded yet. Initially, the compiler places a temporary small function, called a stub that gets called instead of the imported function. The stub calls into the OS. If the library is currently not loaded, it gets loaded (this step is called dynamic loading). Then, the stub is modified so that it calls the imported function directly the next time it gets called. This process is called dynamic linking. The component of the OS that performs both steps is called the dynamic linker or the dynamic linking loader.

The advantage of dynamically loading is: An unused routine is never loaded. This is most useful when the program code is large where infrequently occurring cases are needed to handle such as error routines. Dynamic loading also doesn't need special support from O.S.It is the responsibility of user to design their program to take advantage of method. However, O.S can provide libraries to help the programmer.

Dynamic linker is a run time program that loads and binds all of the dynamic dependencies of a program before starting to execute that program. Dynamic linker will find what dynamic libraries a program requires, what libraries those libraries require (and so on), then it will load all those libraries and make sure that all references to functions then correctly point to the right place. Even the most basic program will usually require the C library to display the output and so the dynamic linker will load the C library before loading the program and will make sure that any calls to printf() go to the right code.

Dynamic linking and loading has three important benefits:

* Software always uses latest versions of shared libraries.
* Executable files are smaller. They do not include the shared libraries.
* The total memory footprint for multiple processes is reduced. With virtual memory, different programs using the same library function only need a single copy in physical memory. If designed carefully the shared library subprograms can have different logical addresses in different programs.

Dynamic linking also permits a program to load and unload routines at runtime, a facility that can otherwise be very difficult to provide.

**Static Libraries:**A Static library or statically-linked library is a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable. This executable and the process of compiling it are both known as a static build of the program. They are usually faster than the shared libraries because a set of commonly used object files is put into a single library executable file. One can build multiple executable without the need to recompile the file. Because it is a single file to be built, use of link commands are simpler than shared library link commands, because you specify the name of the static library.

**Shared libraries:** are libraries that use dynamic linking vs static linking in the compilation steps for compiling a file. Static and dynamic linking are two processes of collecting and combining multiple object files in order to create a single executable file. The main difference between the two is the type of linking they do when creating an executable file.

Dynamic linking defers much of the linking process until a program starts running. Compared to static libraries where a file in compiled and all the machine code is put into an executable file at compile time, dynamic linking performs as programs are executed in the system. In dynamic linking, dynamic libraries are loaded into memory by programs when they start. When you compile a shared library, the machine code is then stored on your machine, so when you recompile the program that has newly added code to it, the compilation process just adds the new code and stores it to memory vs. recompiling an entire file into an executable file like a static library.

## Differences between Shared and Static Libraries?

## Static Libraries

## Advantages:

* Speed
* All the code to execute the file is in one executable file, with little to virtually zero compatibility issues

Disadvantages:

* Constant load time every time a file is compiled and recompiled
* Larger in size because the file has to be recompiled every time when adding new code to the executable

## Shared Libraries:

## Advantages:

* Only one copy of the shared library is kept in memory, making it much faster to compile programs and significantly reducing the size of the executable program
* Dynamic linking load time will be reduced if the shared library code is already present in memory

Disadvantages:

* Slower execution time compared to static libraries
* Potential compatibility issues if a library is changed without recompiling the library into memory

The idea behind shared libraries is to have only one copy of commonly used routines and to maintain this common copy in a unique shared-library segment. These common routines can significantly reduce the size of executable programs, thereby saving disk space.

You can reduce the size of your programs by using dynamic linking, but there is usually a trade-off in performance. The shared library code is not present in the executable image on disk, but is kept in a separate library file. Shared code is loaded into memory once in the shared library segment and shared by all processes that reference it. Dynamically linked libraries therefore reduce the amount of virtual storage used by your program, provided that several concurrently running applications (or copies of the same application) use the procedures provided in the shared library. They also reduce the amount of disk space required for your program provided that several different applications stored on a given system share a library. Other advantages of shared libraries are as follows:

* Run-time performance can be enhanced because the operating system is less likely to page out shared library code that is being used by several applications, or copies of an application, rather than code that is only being used by a single application. As a result, fewer page faults occur.
* The routines are not statically bound to the application but are dynamically bound when the application is loaded. This permits applications to automatically inherit changes to the shared libraries, without recompiling or rebinding.

When a program references a limited number of procedures in a library, each page of the library that contains a referenced procedure must be individually paged into real memory. If the procedures are small enough that using static linking might have linked several procedures that are in different library pages into a single page, then dynamic linking may increase paging thus decreasing performance.

Dynamically linked programs are dependent on having a compatible library. If a library is changed (for example, a new compiler release may change a library), applications might have to be reworked to be made compatible with the new version of the library. If a library is removed from the system, programs using that library will no longer work.

In statically-linked programs, all code is contained in a single executable module. Library references are more efficient because the library procedures are statically linked into the program. Static linking increases the file size of your program, and it may increase the code size in memory if other applications, or other copies of your application, are running on the system.