THE ELF OBJECT FILE FORMAT

PROGRAM EXECUTION

gcc/cc output an executable in the ELF format (Linux)

- “Executable and Linkable Format”

Standard unified binary format for:

- Relocatable object files (.o),
- Shared object files (.so)
- Executable object files

Equivalent to Windows “PE” (Portable Executable) format

THE ELF OBJECT FILE FORMAT

ELF Header

- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

Program Header Table

- Page size, addresses of memory segments (sections), segment sizes.

.text section

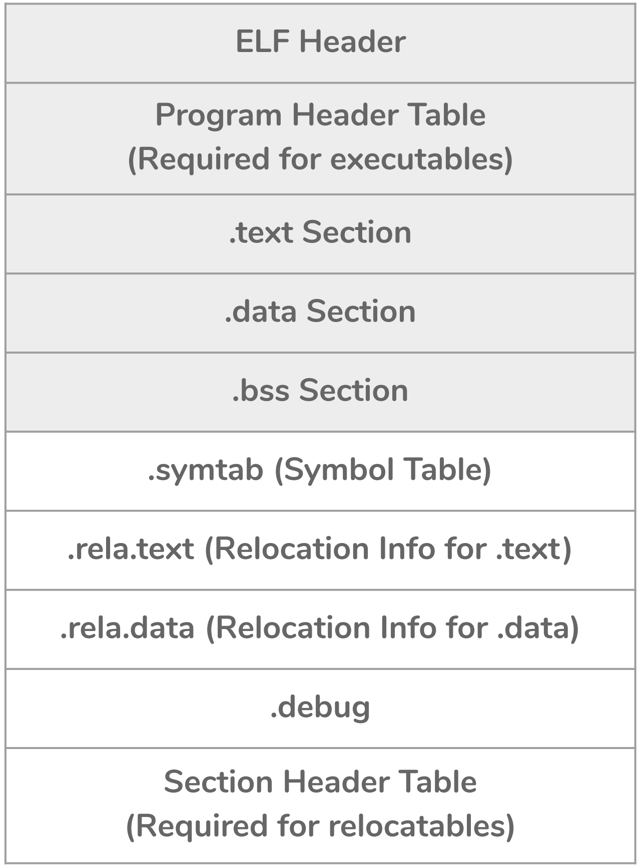
- Program code

.data section

- Initialized (static) global data

.bss section

- Uninitialized (static) global data
- “Block Started by Symbol”


ELF Header
Program Header Table (Required for executables)
.text Section
.data Section
.bss Section
.symtab (Symbol Table)
.rela.text (Relocation Info for .text)
.rela.data (Relocation Info for .data)
.debug
Section Header Table (Required for relocatables)

THE ELF OBJECT FILE FORMAT

.rela.text section

- Relocation info for .text section (For dynamic Linker)

.rela.data section

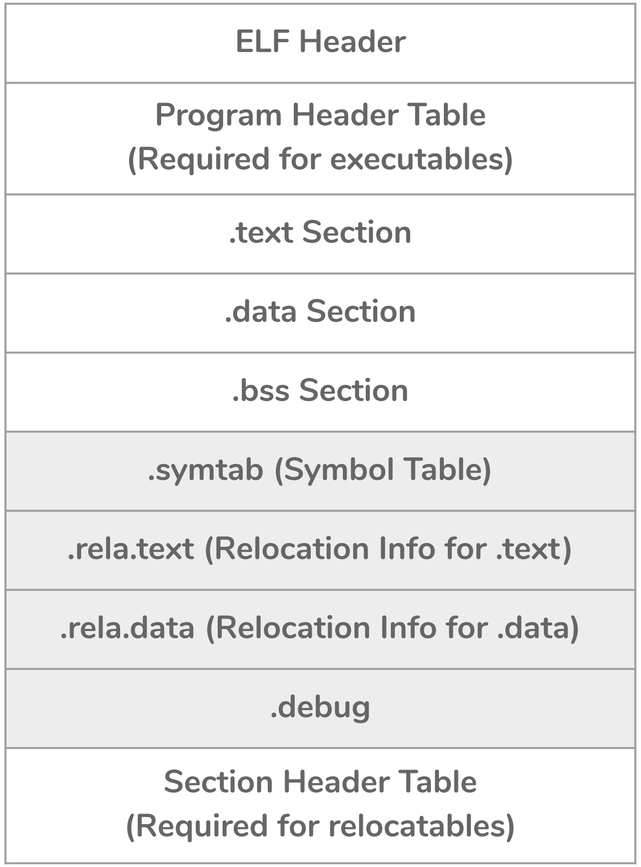
- Relocation info for .data section (For dynamic Linker)

.symtab section

- Procedure and static variable names
- Section names and locations

.debug section

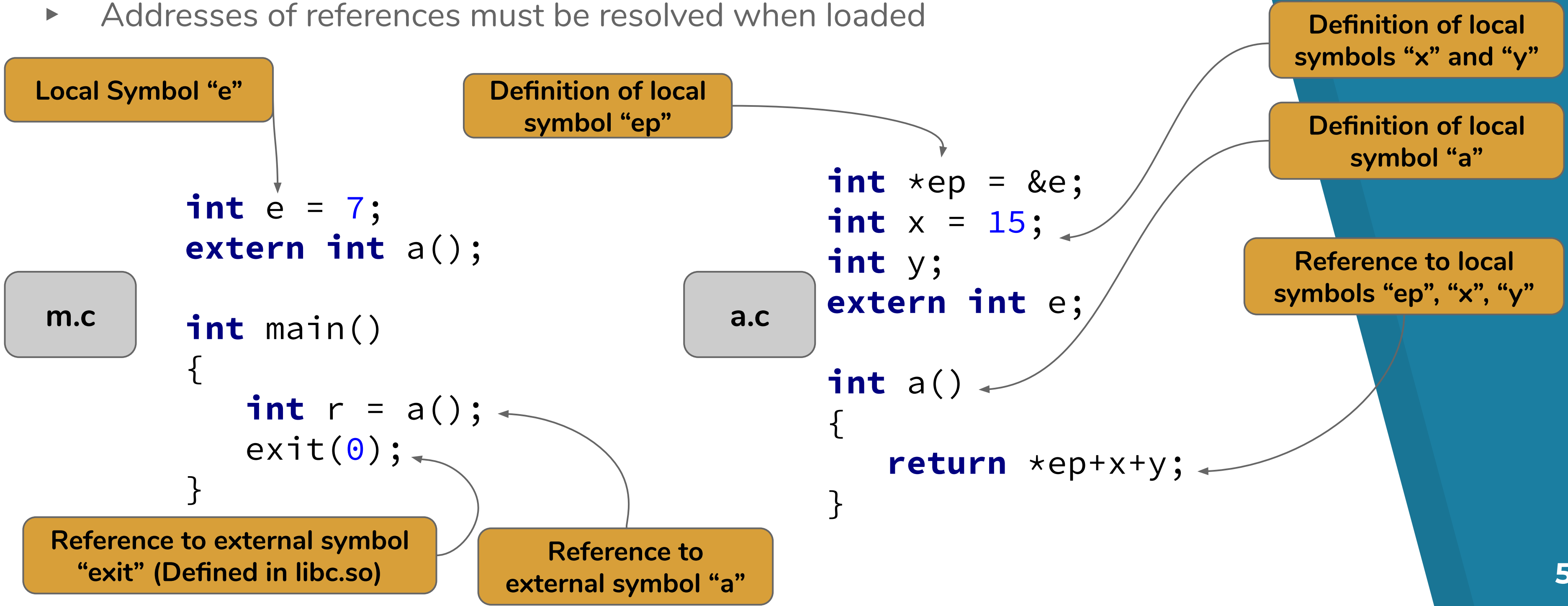
- Information for symbolic debugging (gcc -g)


ELF Header
Program Header Table (Required for executables)
.text Section
.data Section
.bss Section
.symtab (Symbol Table)
.rela.text (Relocation Info for .text)
.rela.data (Relocation Info for .data)
.debug
Section Header Table (Required for relocatables)

ELF EXAMPLE

Program with symbols for code and data

- Contains **definitions** and **references** that are either **local** or **external**
- Addresses of references must be resolved when loaded



m.c

```
int e = 7;
extern int a();

int main()
{
    int r = a();
    exit(0);
}
```

a.c

```
int *ep = &e;
int x = 15;
int y;
extern int e;

int a()
{
    return *ep+x+y;
}
```

Local Symbol “e”
Definition of local symbol “ep”
Definition of local symbols “x” and “y”
Definition of local symbol “a”
Reference to local symbols “ep”, “x”, “y”
Reference to external symbol “exit” (Defined in libc.so)
Reference to external symbol “a”

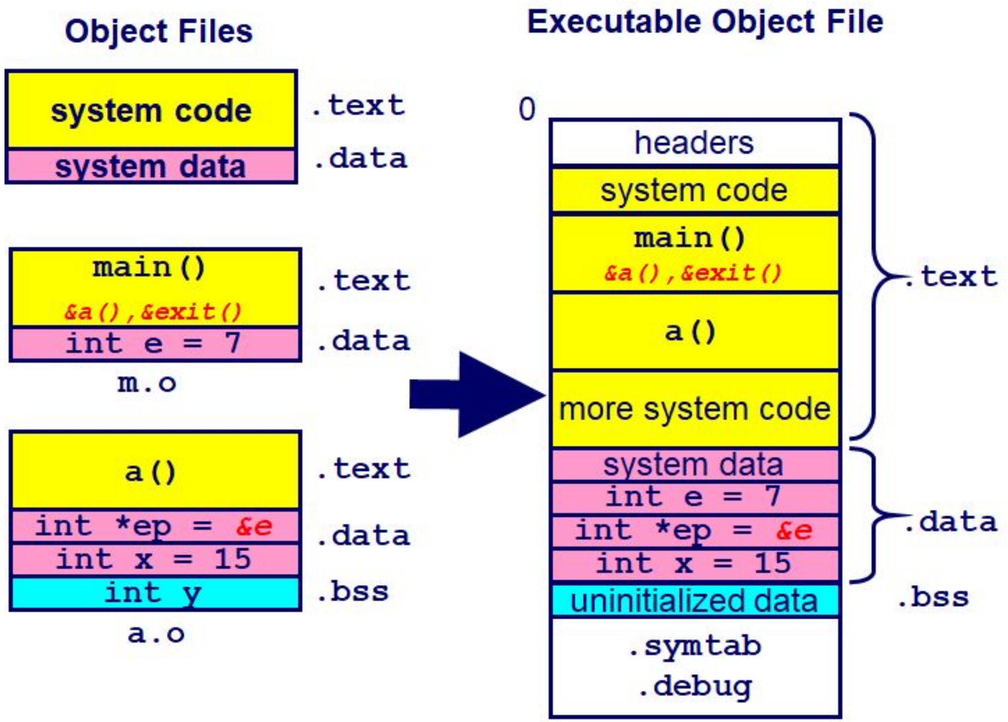
MERGING OBJECT FILES INTO AN EXECUTABLE OBJECT FILE

```
int e = 7;
extern int a();

int main()
{
   int r = a();
   exit(0);
}
```

m.c

```
int *ep = &e;
int x = 15;
int y;
extern int e;

int a()
{
   return *ep+x+y;
}
```

a.c



RELOCATION

Compiler does not know where code will be loaded into memory upon execution

- Instructions and data that depend on location must be “fixed” to actual addresses
- i.e. variables, pointers, jump instructions

.rela.text section

- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying
- e.g. **&a()** in main()

.rela.data section

- Addresses of pointer data that will need to be modified in the merged executable
- e.g. ep reference to **&e** in a()

RELOCATION

m.c

```c
int e = 7;
extern int a();

int main()
{
    int r = a();
    exit(0);
}
```

a.c

```c
int *ep = &e;
int x = 15;
int y;
extern int e;

int a()
{
    return *ep+x+y;
}
```

What is in .text, .data, .rela.text, and .rela.data?

```
readelf -r a.o     ; .rela.text contains ep, x, and y from a()
                   ; .rela.data contains e to initialize ep

objdump -d a.o     ; Shows relocations in .text

readelf -r m.o     ; .rela.text contains a and exit from main()

objdump -d m.o     ; Show relocations in.text

objdump -d m       ; After linking, symbols resolved in <main>
                   ; for <a> and <exit>. References in <a> placed at fixed relative offsets to RIP
```

THE ROLE OF THE OPERATING SYSTEM

Program runs on top of operating system that implements abstract view of resources

- Files as an abstraction of storage and network devices
- System calls an abstraction for OS services
- Virtual memory a uniform memory space abstraction for each process
  - Gives the illusion that each process has entire memory space
- A process (in conjunction with the OS) provides an abstraction for a virtual computer
  - Slices of CPU time to run in
  - CPU state
  - Open files
  - Thread of execution
  - Code and data in memory

Protection

- Protects the hardware/itself from user programs
- Protects user programs from each other
- Protects files from unauthorized access

PROGRAM EXECUTION

The operating system creates a process

- Including among other things, a virtual memory space

System loader reads program from file system and loads its code into memory

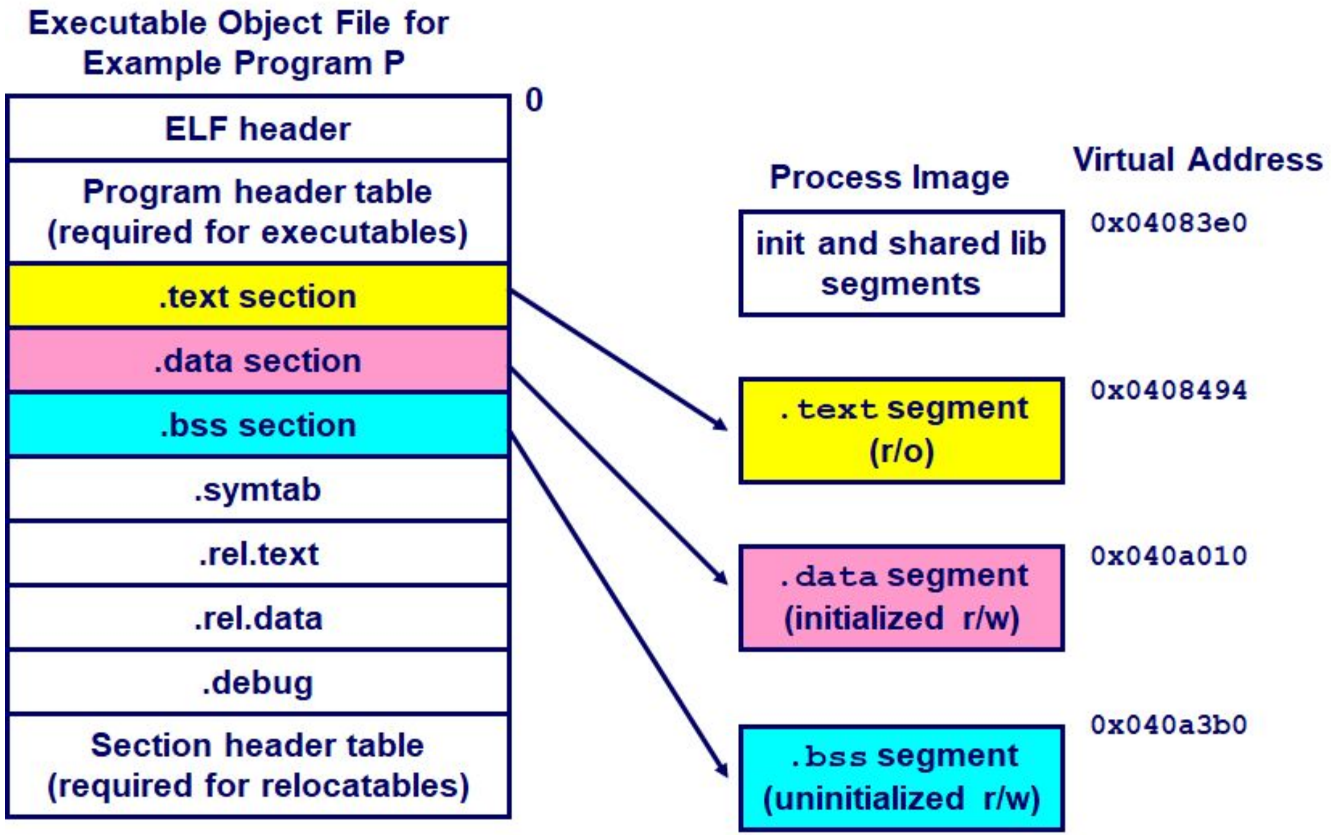
- Program includes any statically linked libraries
- Done via DMA (direct memory access)

System loader loads dynamic shared objects/libraries into memory

Links everything together and then starts a thread of execution running

- Note: the program binary in file system remains and can be executed again
- “Program is a cookie recipe, processes are the cookies”

LOADING EXECUTABLE BINARIES


Executable Object File for Example Program P
0
ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)
Process Image
Virtual Address
init and shared lib segments
0x04083e0
.text segment (r/o)
0x0408494
.data segment (initialized r/w)
0x040a010
.bss segment (uninitialized r/w)
0x040a3b0

WHERE ARE PROGRAMS LOADED IN MEMORY?

An evolution….

Primitive operating systems

- Single tasking
- Physical memory addresses go from zero to N.

The problem of loading is simple

- Load the program starting at address zero
- Use as much memory as it takes
- Linker binds the program to absolute addresses at compile time
- Code starts at zero
- Data concatenated after that
- etc.

WHERE ARE PROGRAMS LOADED IN MEMORY?

Next imagine a multi-tasking operating system on a primitive computer.

- Physical memory space, from zero to N
- Applications share space
- Memory allocated at load time in unused space
- Linker does not know where the program will be loaded
- Binds together all the modules, but keeps them relocatable

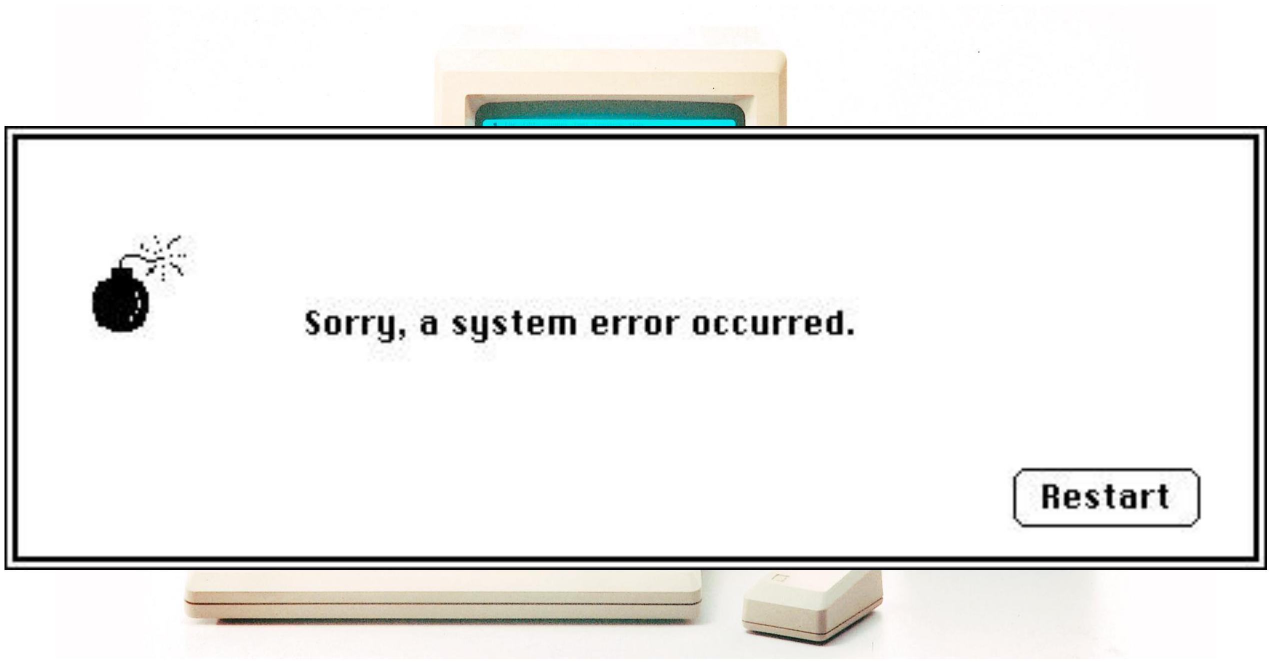
How does the operating system load this program?

- Not a pretty solution, must find contiguous unused blocks

How does the operating system provide protection?

- Not pretty either

WHERE ARE PROGRAMS LOADED IN MEMORY?


Sorry, a system error occurred.

https://www.youtube.com/watch?v=4FOOmoukpJc

WHERE ARE PROGRAMS LOADED IN MEMORY?

Next, imagine a multi-tasking operating system on a modern computer, with hardware-assisted virtual memory (Intel 80286/80386)

OS creates a virtual memory space for each program

- As if program has all of memory to itself.

Back to the simple model

- The linker statically binds the program to virtual addresses
- At load time, OS allocates memory, creates a virtual address space, and loads the code and data.
- Binaries are simply virtual memory snapshots of programs (Windows .com format)

MODERN LINKING AND LOADING

Want to reduce storage

- Dynamic linking and loading versus static
- Single, uniform VM address space still
- But, library code must vie for addresses at load-time
  - Many dynamic libraries, no fixed/reserved addresses to map them into
  - Code must be relocatable again
  - Useful also as a security feature to prevent predictability in exploits (Address Space Layout Randomization)