Outline

1. PIC mechanism
   - Operations in the code
   - Operations in the PLT
   - Operations in the GOT

2. Static and dynamic linkers
   - Difference between static and dynamic linking
   - Load time dynamic linking
   - *ld-linux.so*

3. Load addresses
   - Memory Map
   - Library load addresses

4. Special Sections in ELF
TOC: PIC mechanism

- Operations in the code
- Operations in the PLT
- Operations in the GOT
Lazy binding and constraints

THree steps in a far jump

Operations in the code
Lazy binding and constraints

- ELF dynamic linking defers the resolution of jump / call addresses until the last minute.

- Constraints:
  - should not force a change in the assembly code produced for apps but may cause changes as an assembly code is changed for PIC
  - all executable codes must not be modified at run time
  - any modified data must not be executed at run time

Three steps in a far jump

1. start in the code
2. go through the PLT
3. using a pointer from the GOT

- the GOT entries that are used for PLT execution have default addresses initially
- give control back to the corresponding PLT entry stub
- consisting of push and jmp PLT[0] sequence

Operations in the code

call function_call_n

- the *relative* jump or call
- the target is a PLT entry PLT[n+1]
  - it is \((n+1)-th\) entry not the \(n-th\) entry
  - PLT[0] is the special first entry
- call PLT[n+1]: similar to a normal call
- assume \(n\) is a number

TOC: Operations in the PLT

- PLT entry: stub code
- Indirect call through the GOT
- `push, jmp PLT[0]` sequence
- overriding the default `GOT[n+3]`
- the special entry `PLT[0]`
- Summary of steps
the PLT is a synthetic area, created by the linker
exists in both executable and libraries
an array of stubs, one per imported function call
through PLT[0], the resolver is called at last

```
PLT[n+1]: jmp *GOT[n+3]
push #n ; push n as a argument to the resolver
jmp PLT[0]
```

a call to PLT[n+1] will result in *indirect call* through GOT[n+3]

because of three special GOT entries: GOT[0,1,2]

```
jmp *GOT[n+3] ; 6-byte long
```

initially, the value at GOT[n+3] points back to PLT[n+1]+6

the next instruction after the 6 byte instruction `jmp *GOT[n+3]`

```
push #n ; push n as a argument to the resolver
jmp PLT[0]
```

(3) push, jmp PLT[0] sequence

- by the instruction at PLT[n+1]+6, \( n \) is pushed onto the stack as an **argument** for the **resolver** (push #\( n \))
- consider \( n \) as an **ID** for the called library function
- the resolver uses the argument \( n \) on the stack in resolving the symbol \( n \) (here \( n \) is treated as a symbol)

\[
\begin{align*}
\text{PLT}[n+1]: & \quad \text{jmp} \quad \ast\text{GOT}[n+3] \quad ; \ 6 \text{ bytes instruction} \\
\text{PLT}[n+1]+6: & \quad \text{push} \quad \#n \quad ; \ \text{push n as a argument to the resolver} \\
& \quad \quad \text{jmp} \quad \text{PLT}[0]
\end{align*}
\]

(4) overriding the default GOT[n+3]

- the resolver is called by the stub at PLT[0]
- the resolver modifies the default value at GOT[n+3] to point the correct target symbol \( n \)
- overrides PLT[n+1]+6 (the default value at G[n+3])

thus after the first call, the control is taken directly to the correct target symbol \( n \) (function_call_n) instead of executing the push-jump sequence (through

\[
\begin{align*}
\text{PLT}[n+1] &: \quad \text{jmp} \quad \ast \text{GOT}[n+3] \quad ; \ 6 \text{ bytes instruction} \\
\text{PLT}[n+1]+6 &: \quad \text{push} \quad \#n \quad ; \ \text{push } n \text{ as a argument to the resolver} \\
& \quad \text{jmp} \quad \text{PLT}[0]
\end{align*}
\]

the special entry PLT[0]

- the resolver needs 2 arguments
  - symbol \( n \) is already on the stack
  - pointer to the relocation table: GOT[1]
  - \&GOT[1] is added on the stack

- the resolver that is located in \texttt{ld-linux.so.2} can determine \textit{which library function} is asked for its service using these two arguments on the stack

- GOT[2]: entry point of dynamic linker

\[
\text{PLT[0]: push } \&\text{GOT[1]} \\
\text{jmp } \text{GOT[2]} ; \text{entry point of dynamic linker}
\]

Summary of steps

1. call PLT[n+1]

2. jmp *GOT[n+3]

   at the 1st call, jmp PLT[n+1]+6

   - push #n
   - jmp PLT[0]

       - push &GOT[1] (pointer to the reloc table)
       - jmp GOT[2] (entry point of dynamic linker)

   after the 1st call, jmp n

TOC: Operations in the GOT

- Three types of GOT entries
- the three special GOT entries
- the PLT-fixup
- the PLT-fixup vs data-fixup
the GOT contains *helper pointers* for both PLT fixups and GOT fixups

- the first $3$ entries are special and reserved
- the next $M$ entries belong to the PLT fixups
- the next $D$ entries belong to various data fixups

the GOT is a synthetic area, created by the linker exists in both *executables* and *libraries*

- each *library* and *executable* gets its own PLT and GOT array

the three special GOT entries

- the special 3 entries in the GOT
- GOT[ 0] : linked list pointer used by the dynamic linker
  address of .dynamic section
- GOT[ 1] : pointer to the reloc table for this module
  module identification info for the linker
- GOT[ 2] : pointer to the fixup / resolver code, located in ld-linux.so.2
  entry point in dynamic linker

(3) the PLT-fixup

- when the GOT is first set up, all the GOT entries related to PLT fixups are pointing to code back at PLT[0]

- GOT[n+3] are pointing back to PLT[n+1]+6 which eventually jump to PLT[0] to call the resolver

```plaintext
PLT[n+1]: jmp *GOT[n+3]
push #n ; push n as a argument to the resolver
jmp PLT[0]
```

the PLT-fixup vs data-fixup

- **M** entries belong to the **PLT fixups**
  - \( \text{GOT}[3] \) indirect function call helpers
  - \( \text{GOT}[4] \) indirect function call helpers
  - \( \ldots \)
  - \( \text{GOT}[3+M-1] \) indirect function call helpers, one per imported function

- **D** entries belong to various **data fixups**
  - \( \text{GOT}[3+M] \) indirect pointers to global data references
  - \( \text{GOT}[3+M+1] \) indirect pointers to global data references
  - \( \ldots \)
  - \( \text{GOT}[\text{end}] \) indirect pointers to global data references

Difference between static and dynamic linking
Load time dynamic linking
ld-linux.so
ld vs. ld.so (1)

- **ld** is a **static linker**
  - a static library has the suffix name `.a` denoting archive
  - created by the **ar** utility

- **ld.so** is a **dynamic linker**
  - so represents **shared object**
  - a suffix name of **shared libraries**
  - libraries that may be **dynamically** linked into programs
  - one library is **shared** among several programs

[https://unix.stackexchange.com/questions/356709/difference-between-ld-and-ld-so](https://unix.stackexchange.com/questions/356709/difference-between-ld-and-ld-so)
A static linker links a program or library at compile time usually as the last step in the compilation process, creating a binary executable or a library.

A dynamic linker loads the libraries

- into the process’ address space at run time.
- that were dynamically linked at compile time

https://unix.stackexchange.com/questions/356709/difference-between-ld-and-ld-so
- a **statically linked binary**
  with all libraries loaded into the executable itself

- a **dynamically linked binary**
  with only some libraries **statically linked**

https://unix.stackexchange.com/questions/356709/difference-between-ld-and-ld-so
When you **statically** link a file into an executable, the **contents** of that file are included at **link** time.

When you **dynamically** link a file into an executable, a **pointer** to the file is included in the executable but the **contents** of the file are **not** included at **link** time.

these dynamically linked files are **not** bought in until you **run** the executable

- they are only bought into the in-memory copy of the executable, **not** the one on disk.
  - no files are modified on the disk
  - a shared library is shared across several processes

https://stackoverflow.com/questions/311882/what-do-statically-linked-and-dynamically-linked
statically linked vs dynamically linked (3)

- **statically linked** files are included in the executable at link time
  - statically linked executable and library files never change
    (the last step in the compilation process)

- **dynamically linked** files are referenced by an executable and loaded into memory by the dynamic linker at run time
  - dynamically loaded libraries can change at the next run time just by replacing the corresponding files on the disk.

library built with \texttt{-fPIC}

- the \textit{vast majority} of pages are exactly the same for every process
- different processes load the library at different logical addresses, but they will point to the same physical pages thus, the memory will be shared.
- the data in RAM exactly matches what is on disk, so it can be loaded only when needed by the page fault handler.

library built without -fPIC

- *most* pages of the library will need *link edits*, and will be *different*
- each process has *separate* physical pages because they contain *different* data (as a result of execution)
- that means they’re *not* shared.
- the *pages* don’t *match* what is on *disk*
- in the worst case, the entire library could be *loaded* and then subsequently be *swapped* out to disk (in the swapfile)

https://stackoverflow.com/questions/311882/what-do-statically-linked-and-dynamically-linked
the concept of re-entrant code, i.e., programs that cannot modify themselves while running. It is necessary to write libraries.

re-entrant code is useful for shared libraries

Some functions in a library may be reentrant, whereas others in the same library are non-reentrant.

A library is reentrant if and only if all of the functions in it are reentrant.

https://bytes.com/topic/c/answers/528112-basic-doubt-shared-libraries
a shared library does not need to be reentrant
the **code** area of the library is shared by multiple processes
the **data** area of the library is copied separately for each process

reentrant codes are required when running in **multi-thread**

https://bytes.com/topic/c/answers/528112-basic-doubt-shared-libraries
Load time dynamic linking

- Load time dynamic linking vs. run time dynamic linking
- At the link time
load time dynamic linking vs. run time dynamic linking

- **load-time** dynamic linking is when symbols in the library that are referenced by the executable (or another library) are handled when the executable / library is loaded into memory, by the os.

- **run-time** dynamic linking is when you use an API provided by the OS or through a library to load a .so when you need it, and perform the symbol resolution then.

At the link time (1)

- ld is not called at either compile or run time.
- only at the link step is /usr/bin/ld is invoked.
- a link step is performed as a final step in producing an executable, or a shared library.
  - this is called static linking, to differentiate this step from dynamic loading that will happen at run time.
- on Linux, ld is part of the binutils package.

https://stackoverflow.com/questions/52118756/is-ld-called-at-both-compile-time-and
At the link time (2)

- The linker records
  - which shared libraries are required at the run time,
  - possibly which versions of libraries or symbols are required.
  - which run time loader should be used

- The kernel loads executable into memory, and checks whether the run time loader was requested at static link time.
- If it was, the dynamic loader is also loaded into memory, and execution control is passed to it (instead of the main executable).

https://stackoverflow.com/questions/52118756/is-ld-called-at-both-compile-time-and
then the **dynamic loader** is to **examine** the executable for instructions on which **other libraries** are required, check whether correct **versions** can be found, load them into memory, and **prepare** symbol resolution between the **main executable** and the **shared libraries**.

This is the **run time loading** step, often also called **dynamic linking**

The dynamic loader can be part of the OS, but on Linux it’s part of libc (GLIBC, uClibc and musl each have their own loader).

https://stackoverflow.com/questions/52118756/is-ld-called-at-both-compile-time-and
TOC: ld-linux.so

- ld-linux.so vs. ld.so
- glibc
- ld-linux.so
The programs ld.so and ld-linux.so find and load the shared libraries required by a program, prepare the program to run, and then run it.

Linux binaries require dynamic linking (linking at run time) unless the -static option was given to ld(1) during compilation.

ld.so handles a.out binaries (a format used long ago)

ld-linux.so handles ELF
  /lib/ld-linux.so.1 for libc5,
  /lib/ld-linux.so.2 for glibc2

https://linux.die.net/man/8/ld-linux
glibc

1. **C library** described in ANSI, c99, c11 standards.
   - It includes **macros**, **symbols**, **function** implementations etc.
   - `(printf(), malloc() etc)

2. **POSIX standard library.**
   - The "userland" glue of **system calls**. (open(), read() etc.
   - No actual implementations of system calls. (kernel does it)
   - But glibc provides the user land interface to the services provided by kernel so that user application can use a system call just like a ordinary function.

3. **Also some nonstandard but useful stuff.**

https://linux.die.net/man/8/ld-linux
**ld-linux.so (1)**

- *ld.so, ld-linux.so* are Linux’s **dynamic loader / linker**
- Most modern programs are **dynamically linked**
- When a **dynamically linked application** is **loaded** by the operating system, the **dynamic loader** must **locate** and **load** the **dynamic libraries** it needs for execution.
- On Linux, that job is handled by *ld-linux.so.2*
- You can see the **libraries** used by a given **application** with the **ldd** command:

  https://www.cs.virginia.edu/~dww4s/articles/ld_linux.html
The dynamic linker can be executed either indirectly by running some dynamically linked program or shared object.

- the dynamic linker is specified in the `.interp` section of an ELF file - the program to be executed
- no command-line options to the dynamic linker

executed directly by the command-line

/`lib/ld-linux.so.*` [OPTIONS] [PROGRAM [ARGUMENTS]]

`man ld-linux.so`
run time linker for dynamic objects

a dynamic applications
- consist of one or more dynamic objects
- typically a dynamic executable and
  one or more shared object dependencies

In Solaris, this interpreter is referred to as the run time linker
- dynamic linker
- dynamic loader

As part of the *initialization* and *execution* of a *dynamic* application, an *interpreter* is called.

This *interpreter* completes the *binding* of the application to its *shared object dependencies*.

TOC: Load addresses

- Memory Map
- Library load addresses
TOC: Memory Map

- Load address
- i386 Load addresses 1999 (increasing from the top)
- i386 Load addresses 1999 (increasing from the bottom)
- Linux run-time memory image
- mmpa
- sys_brk
in a typical Linux system, the addresses 0 - 3fff_ffff (4 GB) are available for the user program space.

Executable binary files include header information that indicates a load address.

Libraries, because they are position-independent, do not need a load address, but contain a 0 in this field.

### i386 load addresses 1999 (increasing from the top)

<table>
<thead>
<tr>
<th>Start</th>
<th>Len</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_0000</td>
<td>4k</td>
<td>zero page</td>
</tr>
<tr>
<td>0000_1000</td>
<td>128M</td>
<td>not used</td>
</tr>
<tr>
<td>0800_0000</td>
<td>896M</td>
<td>app code/data space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>followed by small-malloc() space</td>
</tr>
<tr>
<td>4000_0000</td>
<td>1G</td>
<td>mmap space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>library load space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large-malloc() space</td>
</tr>
<tr>
<td>8000_0000</td>
<td>1G</td>
<td>stack space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>working back from BFFF.FFE0</td>
</tr>
</tbody>
</table>

### i386 load addresses 1999 (increasing from the bottom)

<table>
<thead>
<tr>
<th>Start</th>
<th>Len</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000_0000</td>
<td>1G</td>
<td>stack space working back from BFFF.FFE0</td>
</tr>
<tr>
<td>4000_0000</td>
<td>1G</td>
<td>memory mapped region for shared libraries</td>
</tr>
<tr>
<td>0800_0000</td>
<td>896M</td>
<td>large-malloc() space</td>
</tr>
<tr>
<td>0000_1000</td>
<td>128M</td>
<td>small-malloc() space</td>
</tr>
<tr>
<td>0000_0000</td>
<td>4k</td>
<td>app data / code space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not used</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zero page</td>
</tr>
</tbody>
</table>

**Linux Run-time Memory Image (increasing from the bottom)**

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xc000_0000</td>
<td>Kernel virtual memory</td>
<td>memory invisible to the user code</td>
</tr>
<tr>
<td>User stack</td>
<td>created at run time</td>
<td>← %esp stack ptr</td>
</tr>
<tr>
<td>0x4000_0000</td>
<td>memory mapped region</td>
<td>for shared libraries</td>
</tr>
<tr>
<td>Run time heap</td>
<td>created by malloc</td>
<td>← brk</td>
</tr>
<tr>
<td>0x0804_8000</td>
<td>R/W segment (data, bss)</td>
<td></td>
</tr>
<tr>
<td>0x0000_0000</td>
<td>RO segment (init, text, rodata)</td>
<td></td>
</tr>
</tbody>
</table>
mmap (2) is a POSIX-compliant Unix system call that maps files or devices into memory.

- a method of memory-mapped file I/O
- implements demand paging,
  - file contents are not read from disk directly
  - initially do not use physical RAM at all.

- The actual reads from disk are performed in a lazy manner, after a specific location is accessed.

https://en.wikipedia.org/wiki/Mmap
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);

- creates a **new mapping** in the **virtual address space** of the **calling process**
- the starting address for the new mapping is specified in **addr**
- the **length** argument specifies the length of the mapping
- the contents of a file mapping are initialized using **length** bytes starting at **offset** offset in the file (or other object) referred to by the **file descriptor** **fd**

http://man7.org/linux/man-pages/man2/mmap.2.html
the `sys_brk` system call is provided by the kernel, to allocate memory without the need of moving it later.

- Allocates memory right behind the application image in the memory.
- Allows you to set the highest available address in the data section.
  - Takes one parameter (the highest memory address)

https://www.tutorialspoint.com/assembly_programming/assembly_memory_management.htm
sys_brk (2)

- include <unistd.h>

```c
int brk(void *addr);
void *sbrk(intptr_t increment);
```

- `brk()` and `sbrk()` change the location of the program break, which defines the end of the process’s data segment
- the program break is the first location after the end of the uninitialized data segment
- increasing / decreasing the program break has the effect of allocating / deallocating memory to the process;
- `sbrk()` increments the program’s data space by increment bytes.

http://man7.org/linux/man-pages/man2/brk.2.html
Library load addresses
Shared library address
Dyn loader names
load address example
The kernel has a preferred location for **mmap data objects** at 0x4000_0000. Since the shared **libraries** are loaded by **mmap**, they end up here.

**Large mallocs** are realized by creating a **mmap**, so these end up in the pool at 0x4000_0000.

Library load addresses (2)

- the library GLIBC that is mostly used for malloc handles **small mallocs** by calling `sys_brk()`, which extends the **data** area after the app, at 0x0800_0000+sizeof(app).

- As the **mmap pool** grows upward, the **stack** grows downward. Between them, they share 2G bytes.

The shared library design usually loads app first, then the loader notices that it needs support and loads the dynamic loader library (using .interp section) (usually /lib/ld-linux.so.2) at 0x4000_0000.

- other libraries are loaded after ld.so.1.
- see which and where libraries will be loaded by ldd.
- ldd foo_app.

Dynamic loader names

- dynamic loader
- dynamic linker
- runtime linker
- interpreter

- ld-linux.so.2
- ld-linux.so
- ld.so

• consider a diagnostic case where the app (foo_app) is invoked by /lib/ld-linux.so.2 foo_app foo_arg ....
  • the ld-linux.so.2 is loaded as an app
  • since it was built as a library, it tries to load at 0
  • [In ArmLinux, this is forbidden, so the kernel pushes it up to 0x1000

• Once ld-linux.so.2 is loaded, it reads it argv[1] and loads the foo_app at its preferred location (0x0800.0000)

• other libraries are loaded up a the mmap area.

So, in this case, the user memory map appears as

<table>
<thead>
<tr>
<th>start</th>
<th>Len</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_0000</td>
<td>128M</td>
<td>ld-linux.so.2 followed by small-malloc() space</td>
</tr>
<tr>
<td>0800_0000</td>
<td>896M</td>
<td>app code/data space</td>
</tr>
<tr>
<td>4000_0000</td>
<td>1G</td>
<td>mmap space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lib space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large-malloc() space</td>
</tr>
<tr>
<td>8000_0000</td>
<td>1G</td>
<td>stack space, working backward from BFFF_FFE0</td>
</tr>
</tbody>
</table>

Notice that the small malloc space is much smaller in this case (128M), but this is supposed to be for load testing and diagnostics.

Sections in relocatable object files

- **.text**, the machine code of the compiled program.
- **.rodata**, read-only data, such as the format strings in printf statements.
- **.data**, initialized global variables.
- **.bss**, uninitialized global variables. BSS stands for block storage start, and this section actually occupies no space in the object file; it is merely a placeholder.
- **.symtab**, a symbol table with information about functions and global variables defined and referenced in the program. This table does not contain any entries for local variables; those are maintained on the stack.

https://www.linuxjournal.com/article/6463
Sections in relocatable object files

- `.rel.text`, a list of locations in the .text section that need to be modified when the linker combines this object file with other object files.
- `*=.rel.data*=`, relocation information for global variables referenced but not defined in the current module.
- `.debug`, a debugging symbol table with entries for local and global variables. This section is present only if the compiler is invoked with a `-g` option.
- `.line`, a mapping between line numbers in the original C source program and machine code instructions in the .text section. This information is required by debugger programs.
- `.strtab`, a string table for the symbol tables in the .symtab and .debug sections.

https://www.linuxjournal.com/article/6463
Reoc sections

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.rel.bss</td>
<td>contains all the <strong>R_386_COPY</strong> relocs</td>
</tr>
<tr>
<td>.rel.plt</td>
<td>contains all the <strong>R_386_JUMP_SLOT</strong> relocs</td>
</tr>
<tr>
<td></td>
<td>these modify the first half of the GOT elements</td>
</tr>
<tr>
<td>.rel.got</td>
<td>contains all the <strong>R_386_GLOB_DATA</strong> relocs</td>
</tr>
<tr>
<td></td>
<td>these modify the second half of the GOT elements</td>
</tr>
<tr>
<td>.rel.data</td>
<td>contains all the <strong>R_386_32</strong> and <strong>R_386_RELATEIVE</strong> relocs</td>
</tr>
</tbody>
</table>

=,rela.dyn= runtime/dynamic relocation table

- For dynamic binaries, .rela.dyn relocation table holds information of variables which must be relocated upon loading.
- Each entry in this table is a struct Elf64_Rela (see /usr/include/elf.h) which has only three members:
  - offset: the variable's [usually position-independent] virtual memory address which holds the "patched" value during the relocation process
  - info: index into .dynsym section and relocation type
  - addend

.rela.plt runtime/dynamic relocation table

- .rela.plt relocation table is similar to the one in .rela.dyn section;
  - .rela.plt is for functions
  - .rela.dyn is for variables
- The relocation type of entries in this table is R_386_JMP_SLOT or R_X86_64_JUMP_SLOT and the offset refers to memory addresses which are inside .got.plt section.
- .rela.plt table holds information to relocate entries in .got.plt section.

For programs compiled with -c option, this section provides information to the link editor ld where and how to patch executable code in .text section.

The difference between .rel.text and .rela.text entries in .rel.text does not have addend member instead, the addend is taken from the memory location described by offset member. (Compare struct Elf64_Rel with struct Elf64_Rela in /usr/include/elf.h)

Whether to use .rel or .rela is platform-dependent. For x86_32, it is .rel and for x86_64, .rela

Compile-time/Static relocation table for other sections.

For example, `.rela.init_array` is the relocation table for `.init_array` section.

For dynamic binaries, this Global Offset Table holds the addresses of variables which are relocated upon loading. See paragraphs below.

For dynamic binaries, this Global Offset Table holds the addresses of functions in dynamic libraries.

They are used by trampoline code in .plt section.

If .got.plt section is present, it contains at least three entries, which have special meanings.

For dynamic binaries, this Procedure Linkage Table holds the trampoline/linkage code.

Uninitialized global data ("Block Started by Symbol").

Depending on the compilers, uninitialized global variables could be stored in a nameness section called COMMON (named after Fortran 77’s "common blocks"). To wit, consider the following code:

```c
int globalVar;
static int globalStaticVar;
void dummy() {
    static int localStaticVar;
}
```

Compile with gcc -c, then on x86_64, the resulting object file has the following structure:

```bash
$ objdump -t foo.o
```

**SYMBOL TABLE:**

```
....
0000000000000000 1 0 .bss 0000000000000004 globalStaticVar
0000000000000004 1 0 .bss 0000000000000004 localStaticVar.1619
....
0000000000000004 0 *COM* 0000000000000004 globalVar
```

so only the file-scope and local-scope global variables are in the .bss section

If one wants globalVar to reside in the .bss section, use the -fno-common compiler command-line option. Using -fno-common is encouraged, as the following example shows:

Compile with gcc -c, then on x86_64, the resulting object file has the following structure:

```bash
$ cat foo.c
int globalVar;
$ cat bar.c
double globalVar;
int main(){
$ gcc foo.c bar.c
```

Not only there is no error message about redefinition of the same symbol in both source files (notice we did not use the extern keyword here), there is no complaint about their different data types and sizes either. However, if one uses -fno-common, the compiler will complain:

/tmp/ccM71JR7.o:(.bss+0x0): multiple definition of 'globalVar'
/tmp/ccIbS5M0.o:(.bss+0x0): first defined here
ld: Warning: size of symbol 'globalVar' changed from 8 in /tmp/ccIbS5M0.o to 4 in /

• Initialized data.

Similar to .data section, but this section should be made Read-Only after relocation is done.

• Read-only data.