AMD

DYNAMIC LINKING WITH SYSCALL EMULATION MODE

BRANDON POTTER

JUNE 14TH, 2015

STATIC LINKING IS PROBLEMATIC

AMD

- Alters the simulation workload.
  - Changes binary (execution paths, layout, size, and segments).
  - Exhibits less runtime overhead than dynamic linking (dynamic is the default).

- Burdens users.
  - Requires `-static` flags.
  - Requires using archives (`.a`) instead of shared objects (`.so`) for libraries.
  - Requires relinking applications when libraries are altered.
  - Requires sleuthing through configuration files and/or makefiles.

- Can be perilous in sophisticated code bases.
  - OpenMPI FAQ - "Fully static linking is not for the weak, and it is not recommended." https://www.open-mpi.org/faq/?category=mpi-apps#static-mpi-apps
  - Valgrind - static linking affects Memcheck's ability to intercept malloc calls. http://valgrind.org/docs/manual/faq.html#faq.hiddenbug

OUTLINE

AMD

- Review dynamic linking basics.
- Loader discovery of object file type.
- Segment loading into process address space.
- Begin execution with correct PC.
- Jump table example (time permitting).

WHAT IS LD.SO?

AMD

- The ld.so (ld-linux.so) library is the linker/loader for dynamic executables.
  - Locates shared libraries at runtime.
  - Loads shared libraries into process address space at runtime.
  - Resolves unresolved symbols at runtime using a jump table.

- If ld.so is a shared object, how is it loaded?
  - Linux kernel copies ld.so into process address space during a call to execve.
  - Linux kernel also records runtime metadata in the auxiliary vector.
  - Execution begins with program counter set to a special address inside ld.so.
  - Ld.so initializes itself.
  - Ld.so subsequently jumps to the text section.

- So what's the goal here?
  - The goal is to load ld.so into memory, setup the auxiliary vector, and set PC-entry.

WHAT WERE THE CHANGES TO GEM5?

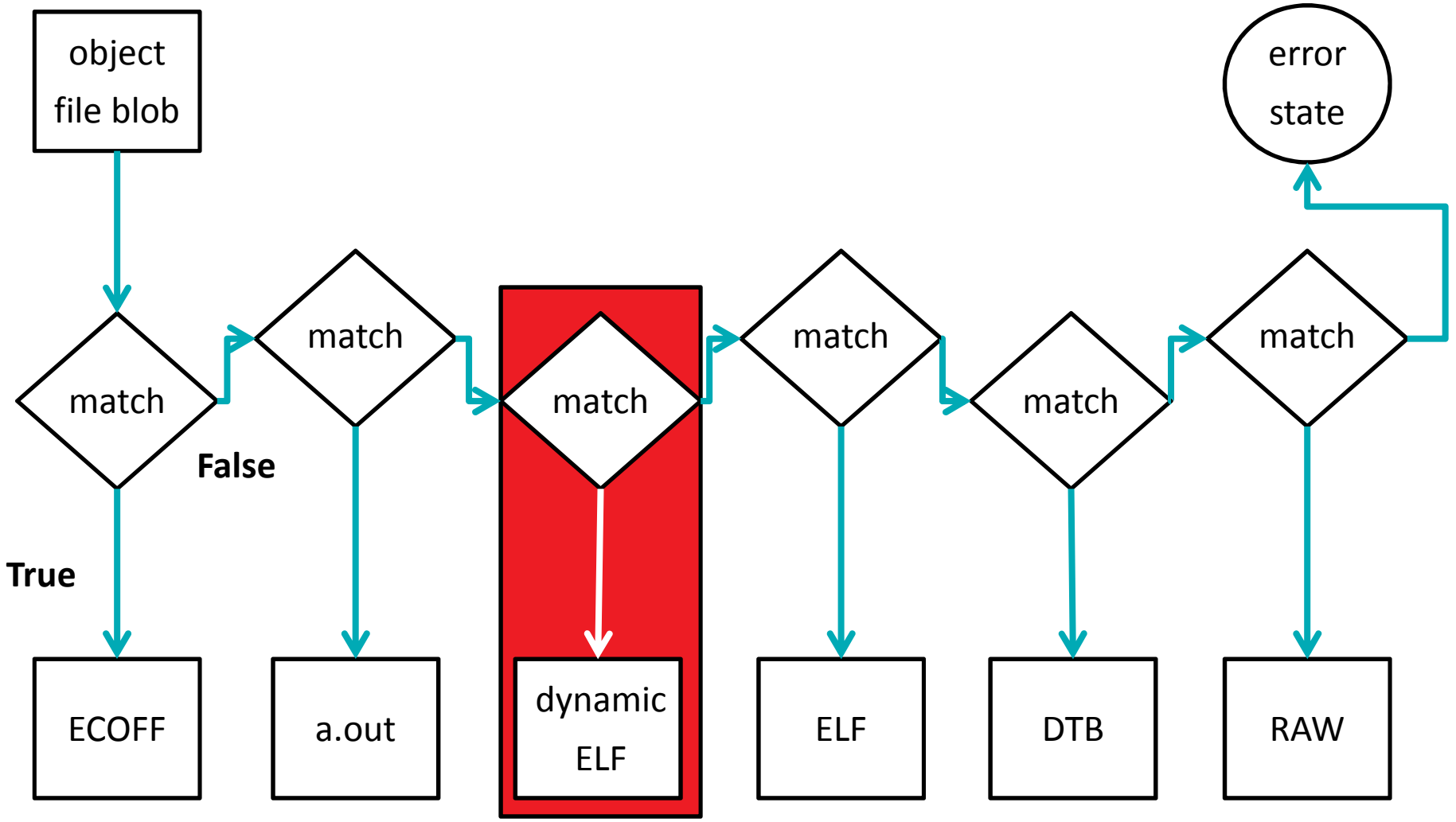
AMD

- The loader required modifications.
  - Added checks to know if binary is a dynamic executable.
  - New derived ELF object class that supports dynamic executables.
  - Special handling of .interp section path and dynamic segment loading.
  - Ability to insert ld.so to process address space.

- Required modifications to `mmap` to support file-backed memory mappings.
  - Ld.so uses `mmap` to load segments into address space.
  - Previously, only anonymous mappings were possible.
  - Needed to add support to determine which direction `mmap` grows on different ISAs.

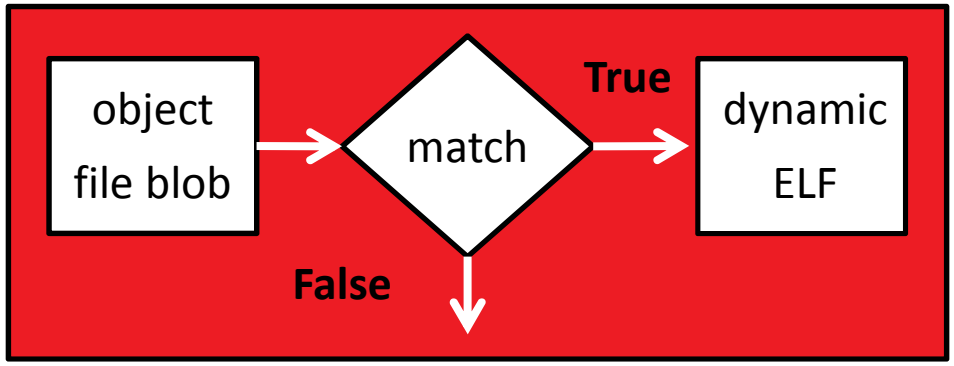
- The object file class required modifications.
  - Added auxiliary vector fields.
  - Added support for changing process entry point.

OBJECT FILE TYPE SEARCH

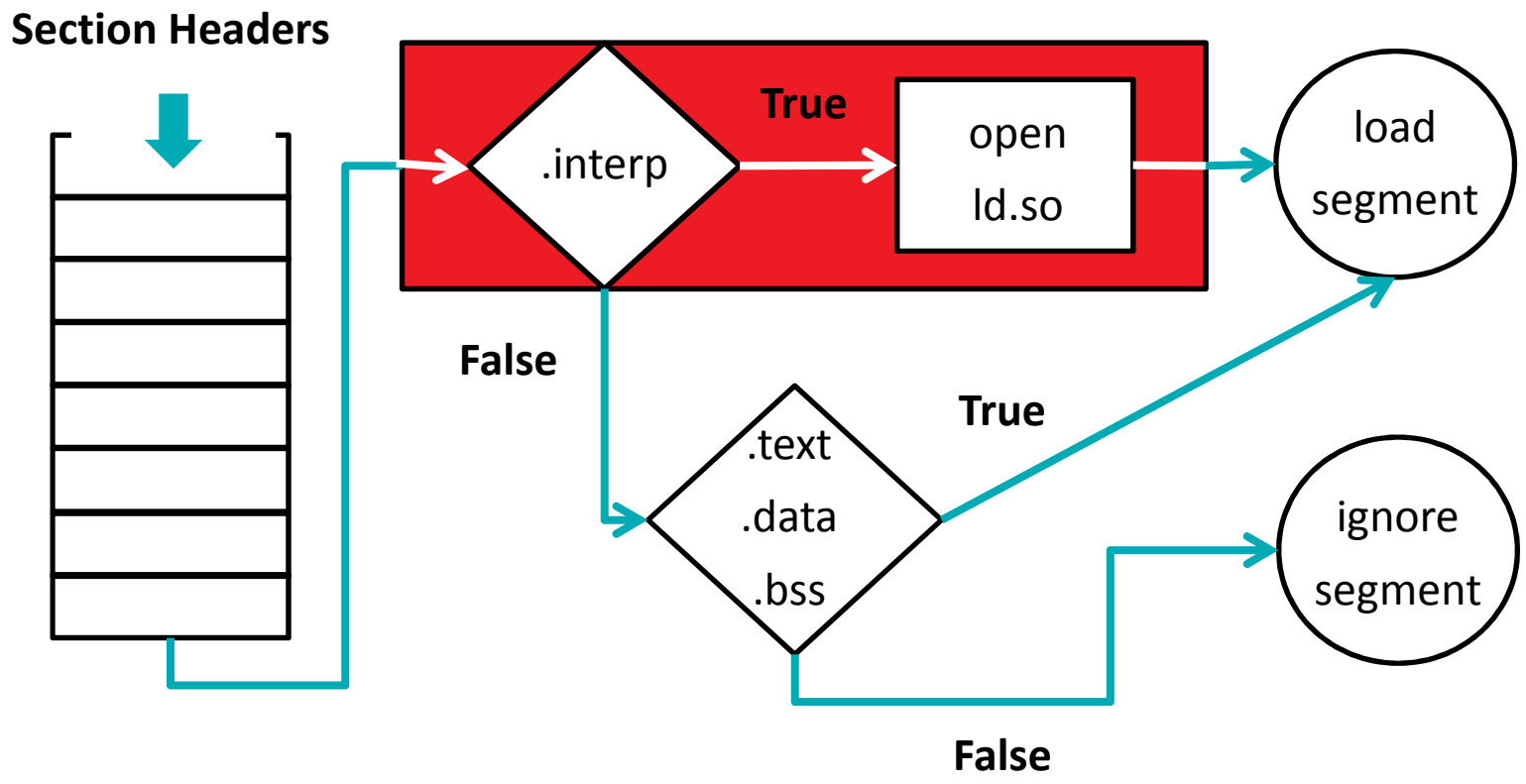


MATCHING PROCESS

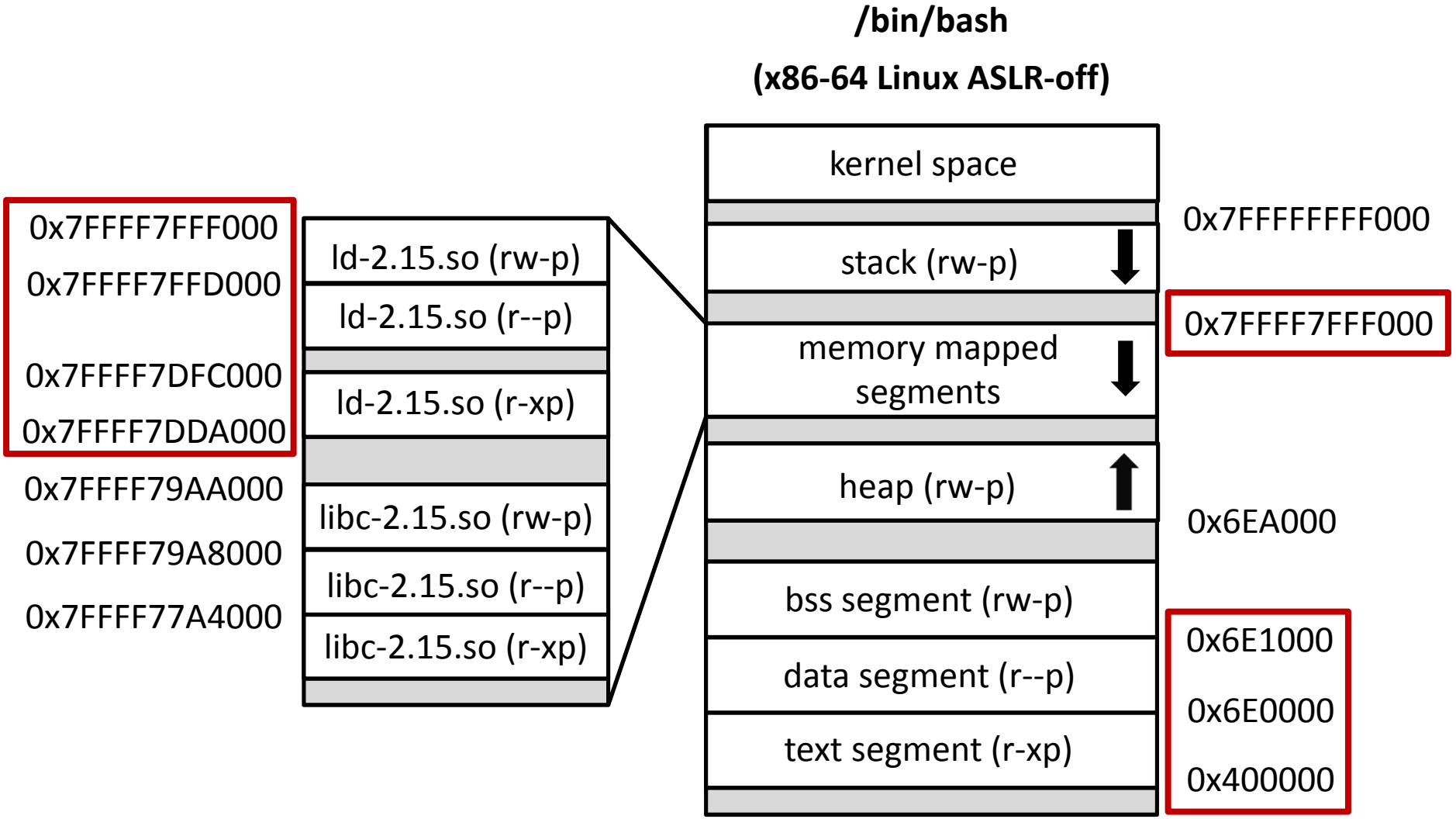
```
if (gelf_getehdr(elf, &ehdr) == 0) {
    DPRINTF(Loader, "Not ELF\n");
    return NULL;
} else {
    Elf_Scn *section = elf_getscn(elf, sectionIndex);
    bool found = false;
    while (search(SHT_DYNAMIC, section, &found)); // semicolon
    if (!found) return NULL;
}
```



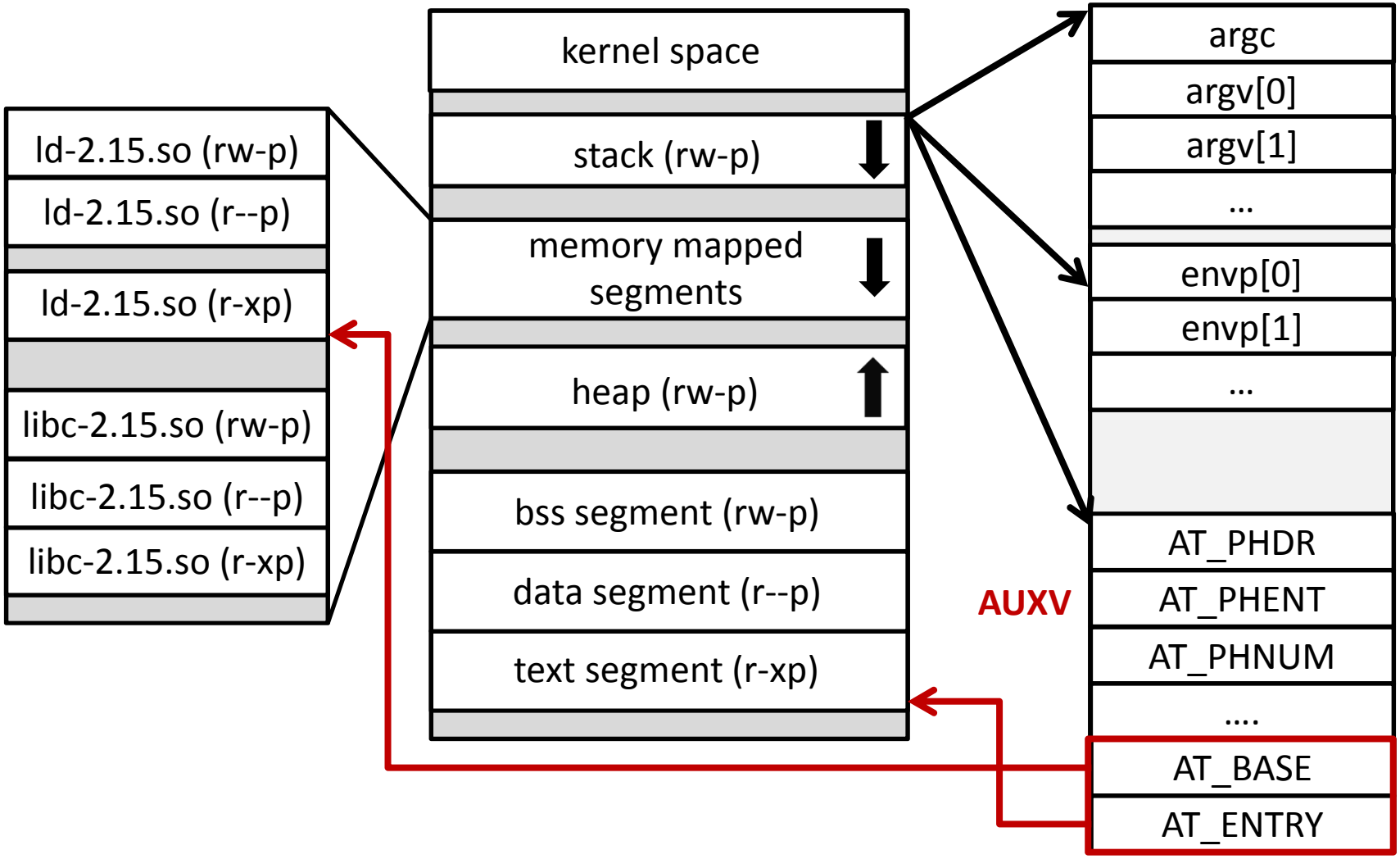
LOADING SEGMENTS



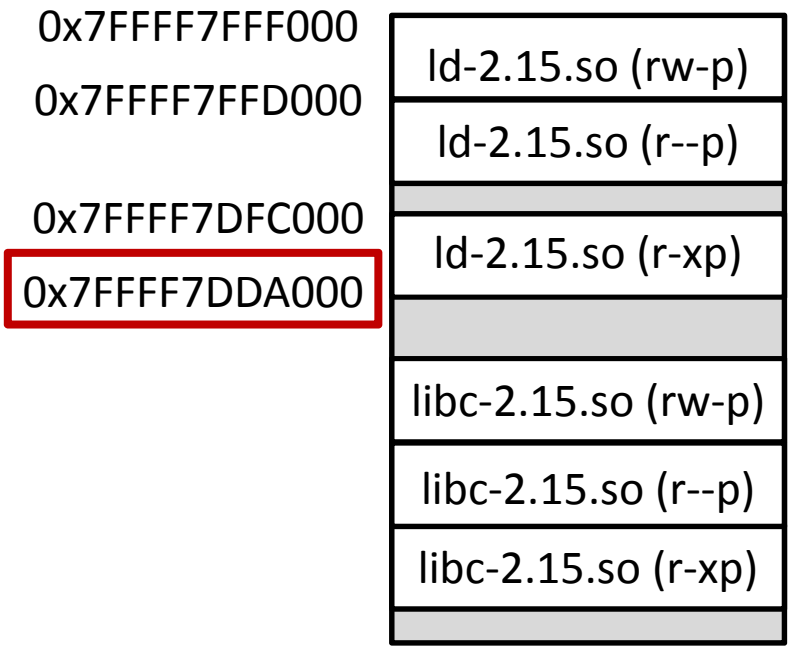
PROCESS ADDRESS SPACE



ENTRY POINT


kernel space
stack (rw-p)
memory mapped segments
heap (rw-p)
bss segment (rw-p)
data segment (r--p)
text segment (r-xp)
ld-2.15.so (rw-p)
ld-2.15.so (r--p)
ld-2.15.so (r-xp)
libc-2.15.so (rw-p)
libc-2.15.so (r--p)
libc-2.15.so (r-xp)
argc
argv[0]
argv[1]
...
envp[0]
envp[1]
...
AUXV
AT_PHDR
AT_PHENT
AT_PHNUM
....
AT_BASE
AT_ENTRY

ACTUAL ENTRY POINT




```
$ readelf -a /lib64/ld-linux-x86-64.so.2
ELF Header:
 Magic: 7f 45 4c 46 02 01 01 00 ...
 Class: 2's complement, little endian
 Version: 1 (current)
 ...
 Entry point address: 0x16B0
 Start of program headers: 64
 Start of section headers: 147744
 ...
```

```
ld_addr := mmap_end; // text loaded here (0x7FFFF7DDA000)
actual_entry := ld_addr + entry_point_address;
binary_entry_point := auxv(AT_ENTRY);
```

CONCLUSION

AMD

- Did not require extensive changes.
  - Only need to load ld.so correctly and kickoff execution.
  - Implementation requires extension to mmap, loader, and object file class.
  - GLIBC developers do much of the heavy lifting with elf/rtld.

- Shake out bugs over time.
  - Current source is first pass implementation.
  - Subsequent iteration will more fully support dynamic linking.
  - Corner case where ld.so is invoked as the executable; does not work properly.

- Searching through ELF sections for segment existence needs proper fix.
  - Relies on the linkage information being present in executable; possible to strip out.
  - Correct thing to do is only search through segments and never consider sections.

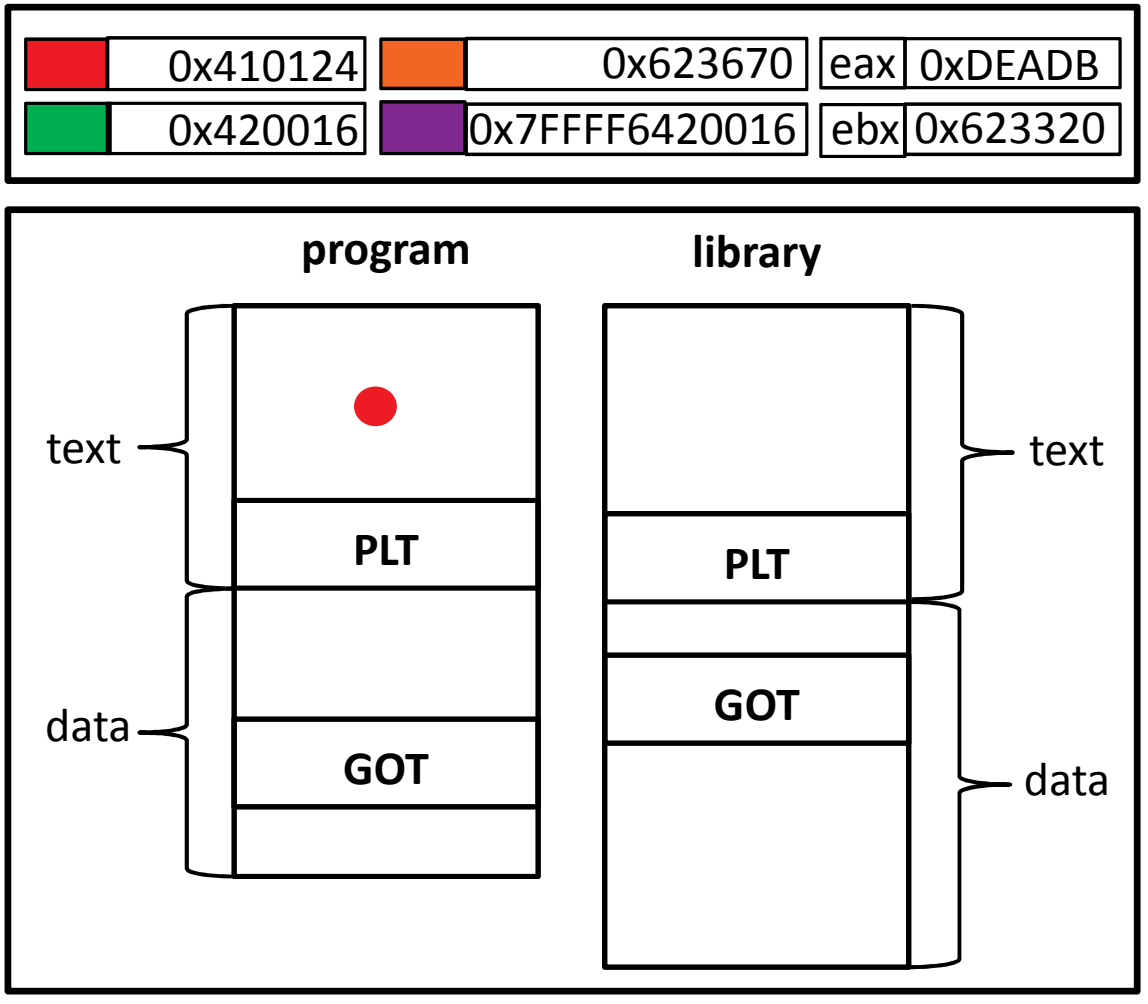
HOW DOES THE JUMP TABLE WORK?

AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```



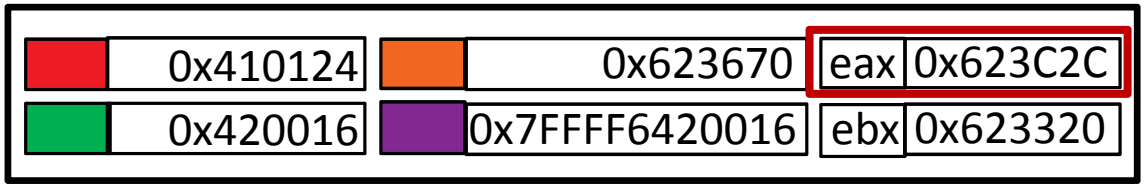
HOW DOES THE JUMP TABLE WORK?

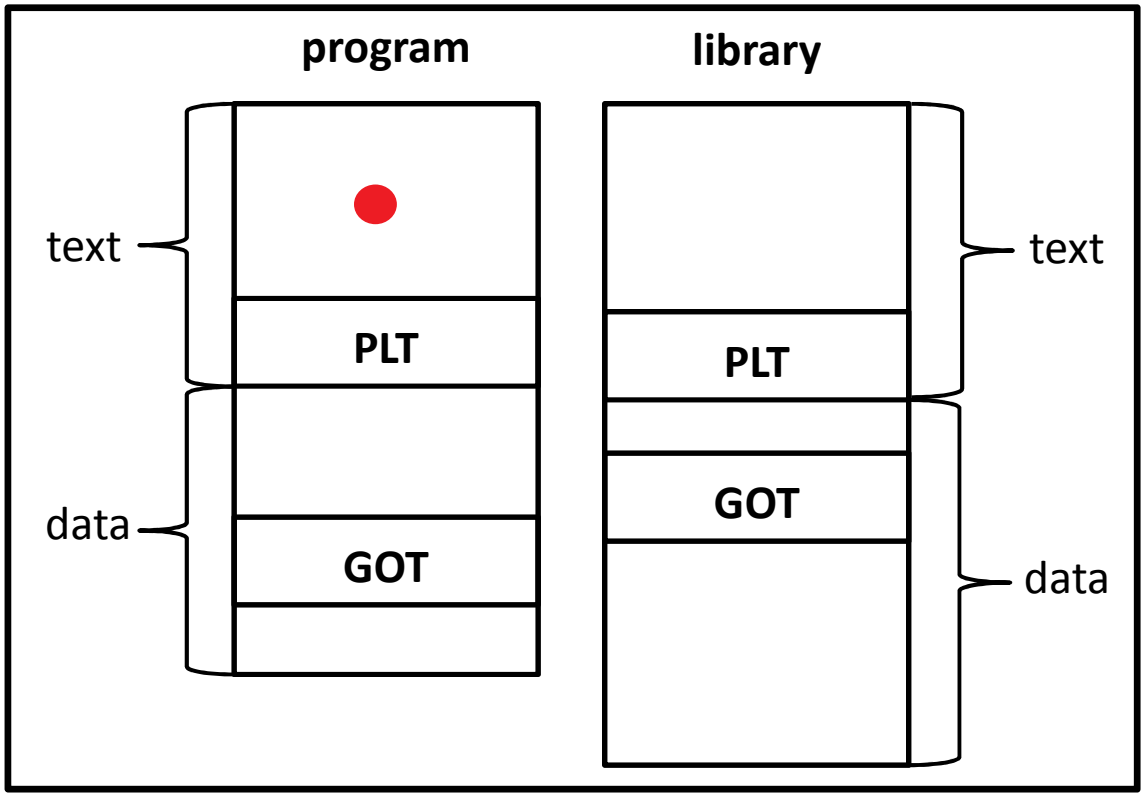
AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```





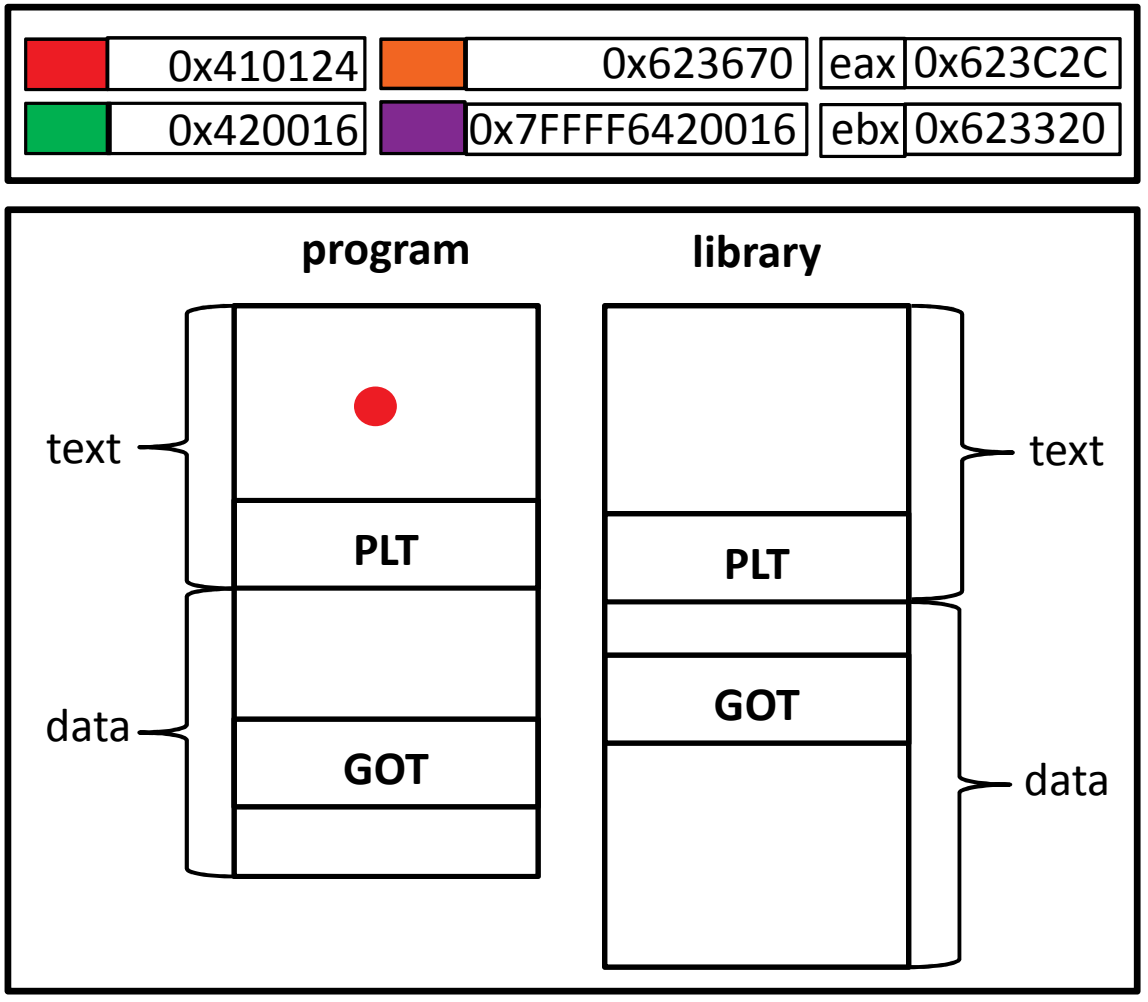
HOW DOES THE JUMP TABLE WORK?

AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```



HOW DOES THE JUMP TABLE WORK?

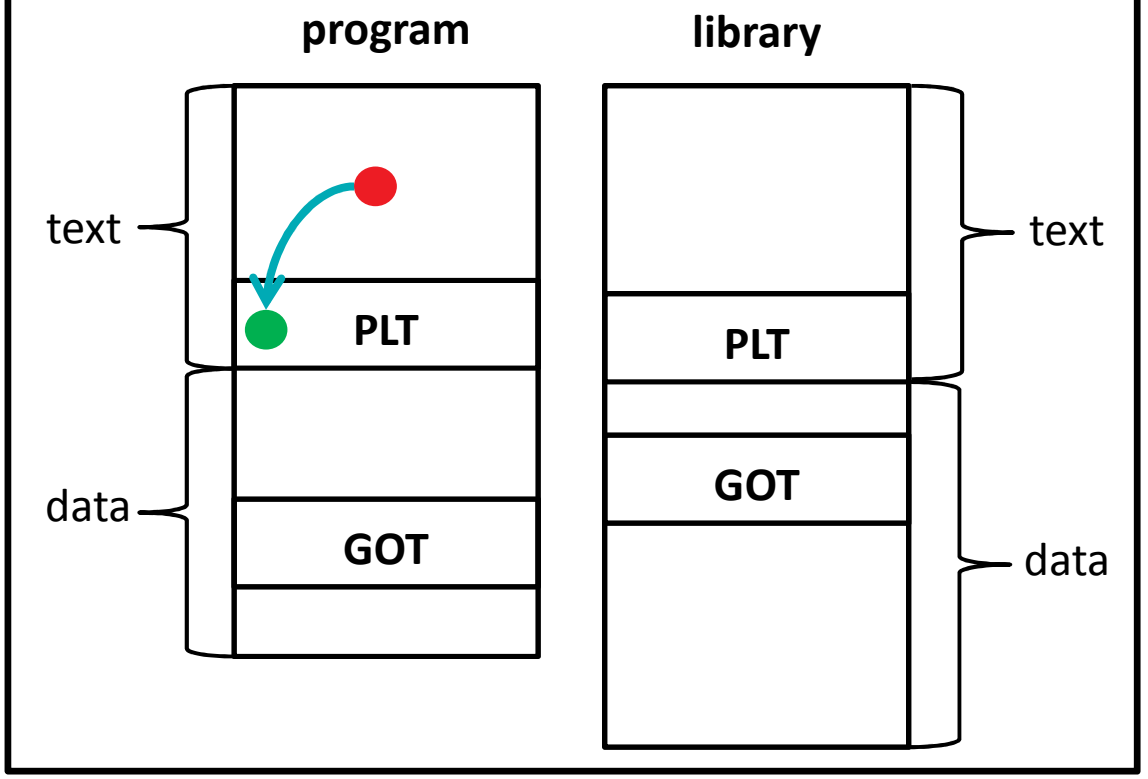
AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```

0x410124	0x623670	eax	0x623C2C
0x420016	0x7FFFF6420016	ebx	0x623320



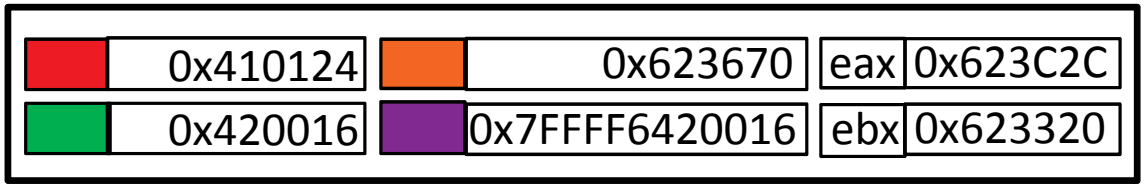
HOW DOES THE JUMP TABLE WORK?

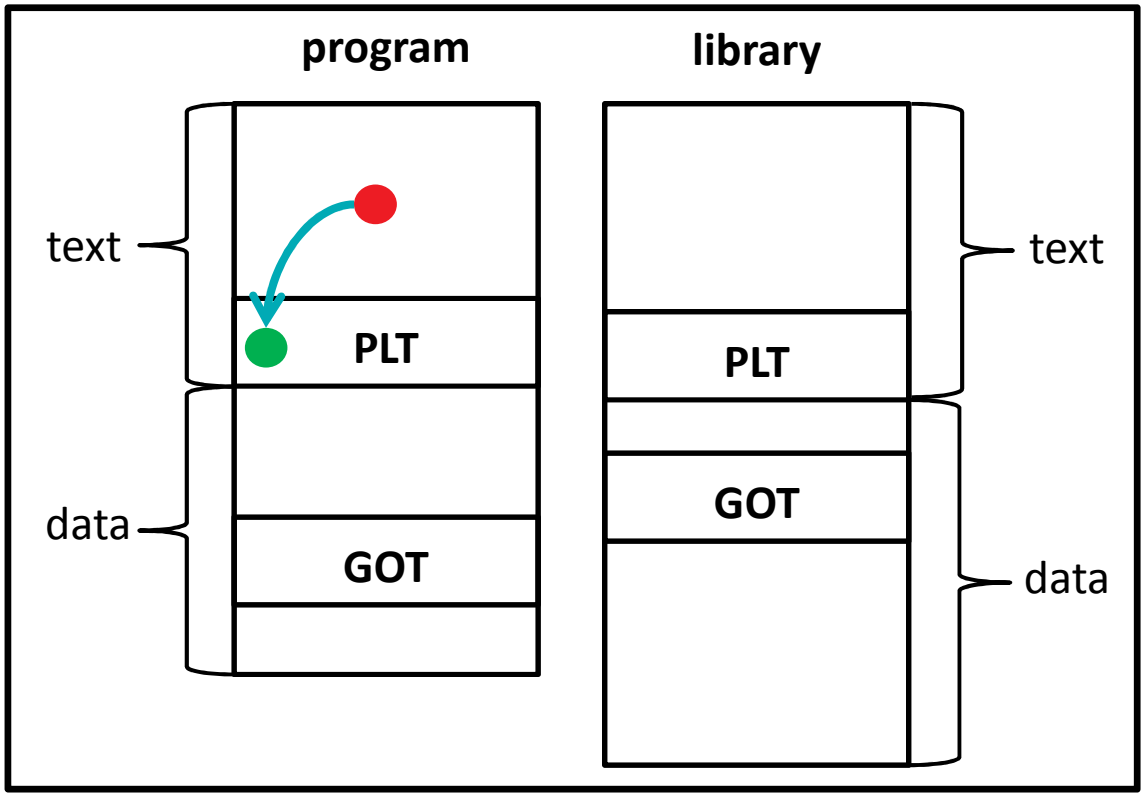
AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```





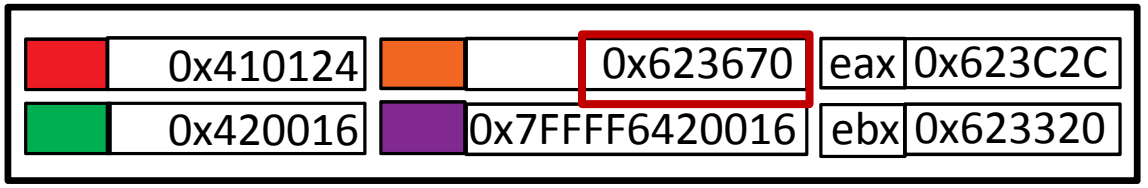
HOW DOES THE JUMP TABLE WORK?

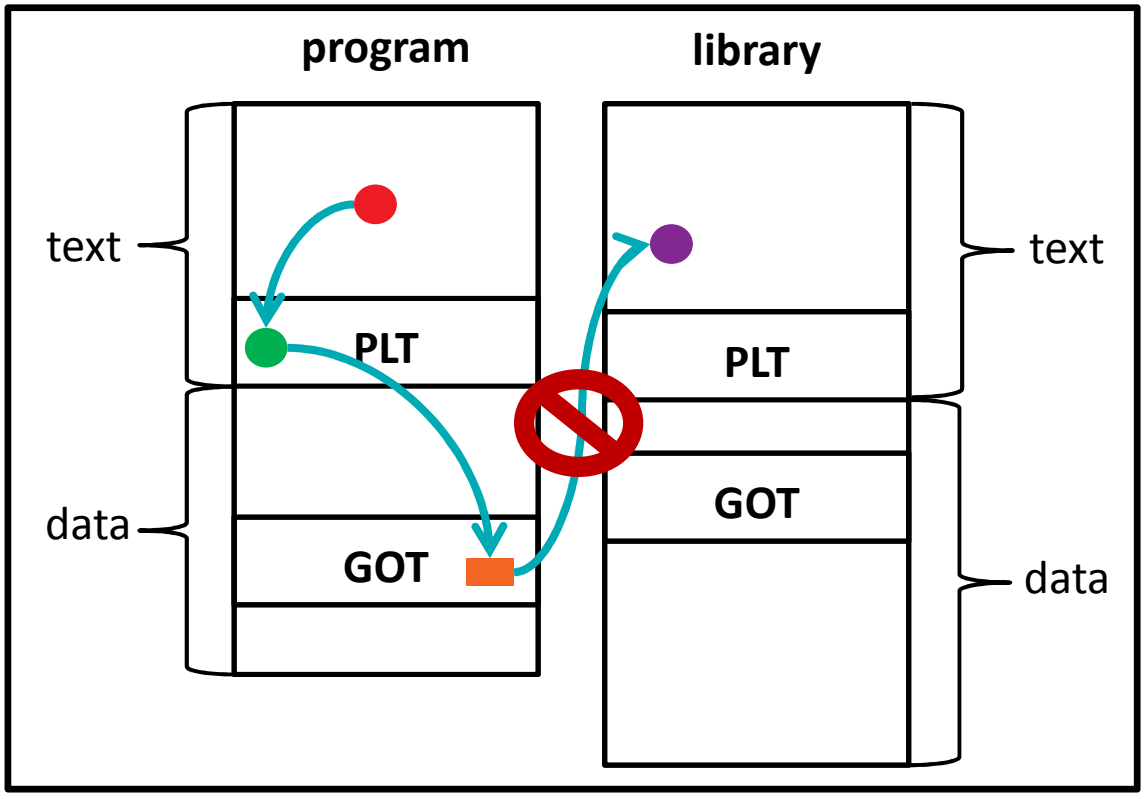
AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```





HOW DOES THE JUMP TABLE WORK?

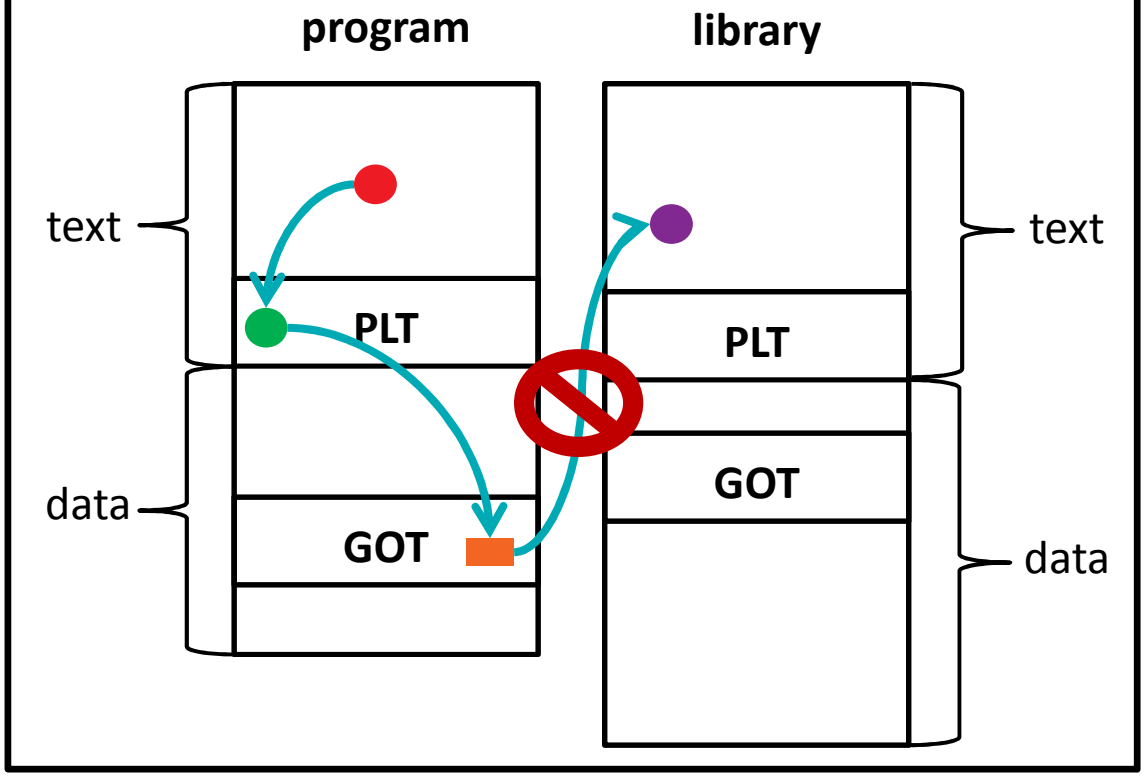
AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```

0x410124	0x623670	eax	0x623C2C
0x420016	0x7FFFF6420016	ebx	0x623320



HOW DOES THE JUMP TABLE WORK?

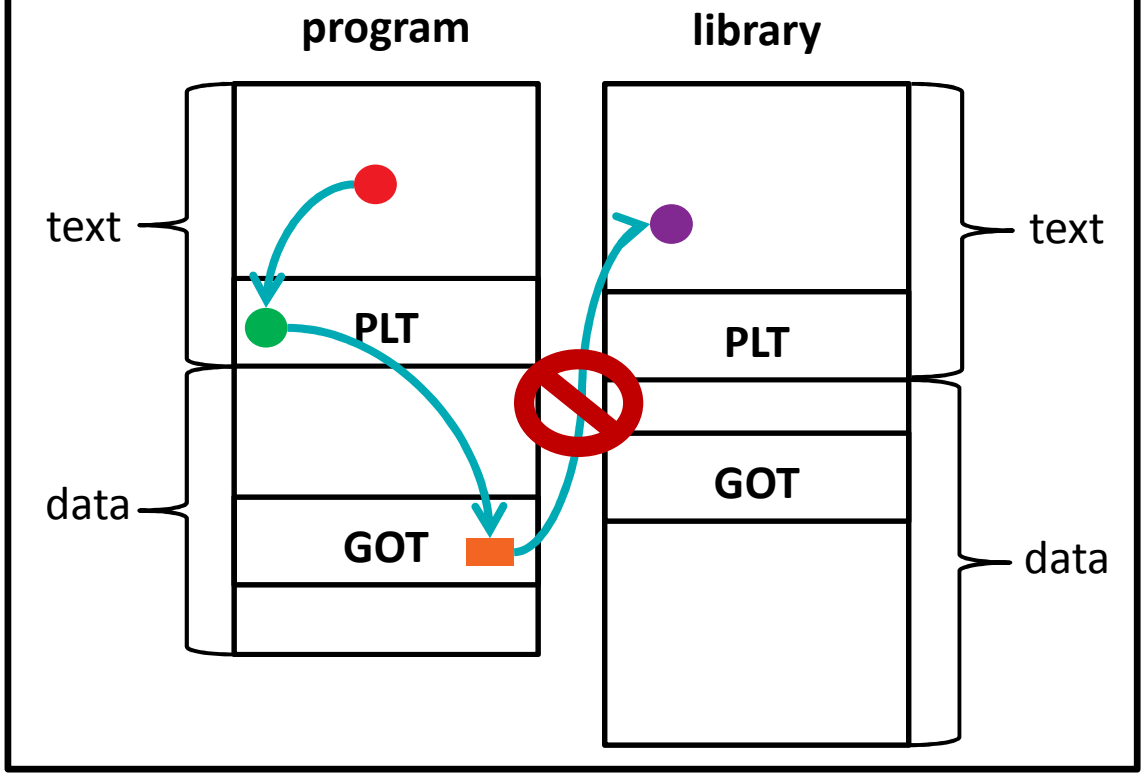
AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```

0x410124	0x623670	eax	0x623C2C
0x420016	0x7FFFF6420016	ebx	0x623320



HOW DOES THE JUMP TABLE WORK?

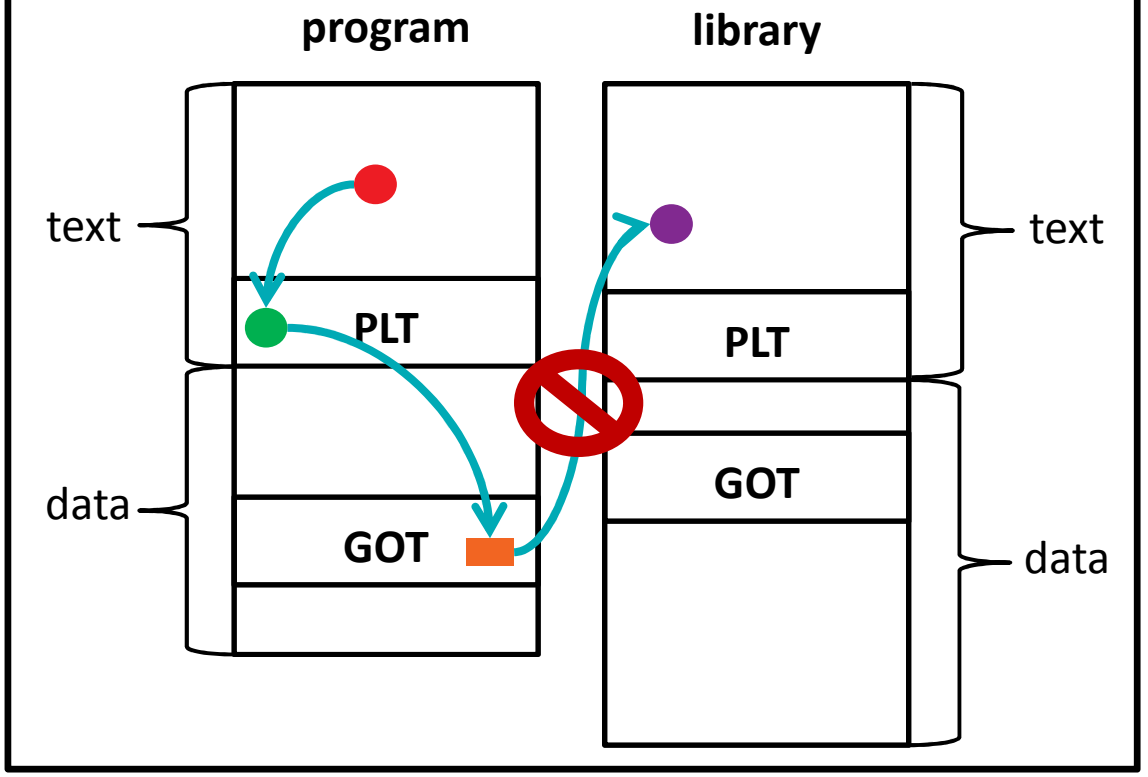
AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```

0x410124	0x623670	eax	0x623C2C
0x420016	0x7FFFF6420016	ebx	0x623320



HOW DOES THE JUMP TABLE WORK?

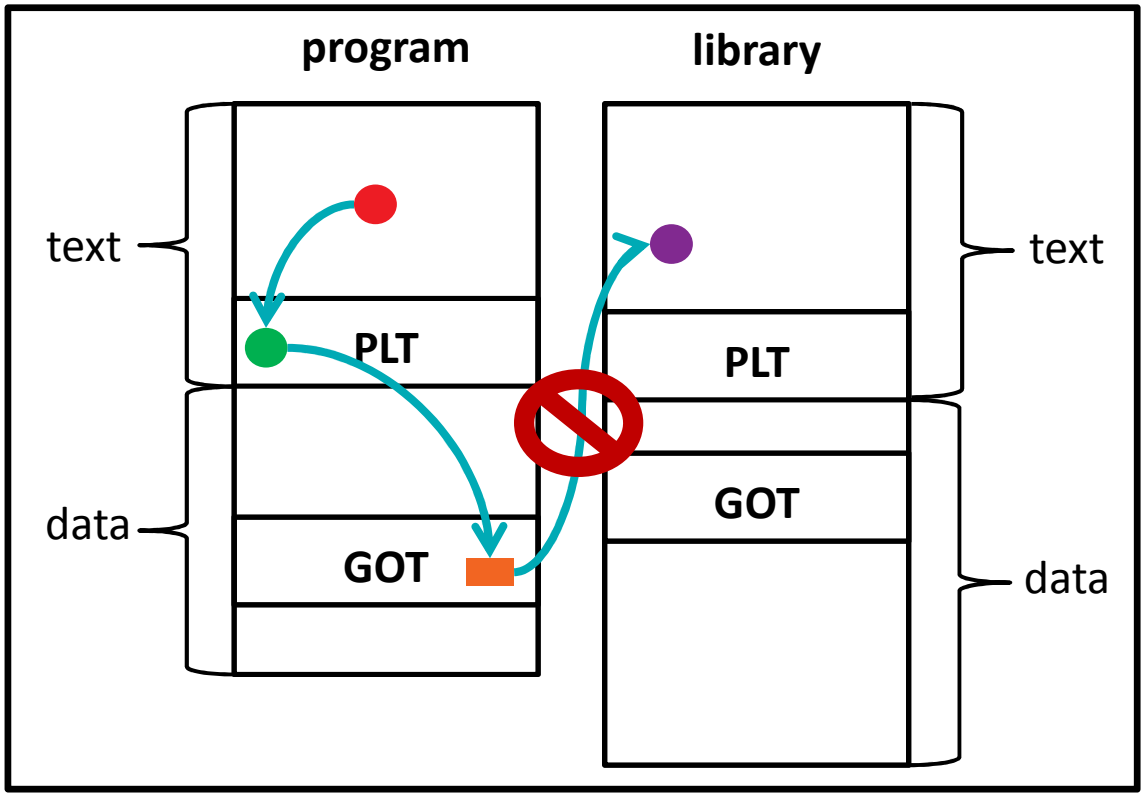
AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```

0x410124	0x7FFFF5DFCCF6	eax	0xDEADB
0x420016	0x7FFFF6420016	ebx	0x623320



HOW DOES THE JUMP TABLE WORK?

AMD

```
extern int foo;
extern int bar (int);
int call_bar (void) {
    return bar(foo);
}
```

```
movl foo@GOT(%ebx), %eax
pushl (%eax)
call bar@PLT
```

```
.PLT0: pushl 4(%ebx)
       jmp *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp *name1@GOT(%ebx)
       pushl $offset1
       jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
       pushl $offset2
       jmp .PLT0@PC
```

0x410124	0x7FFFF5DFCCF6	eax	0x623C2C
0x420016	0x7FFFF6420016	ebx	0x623320