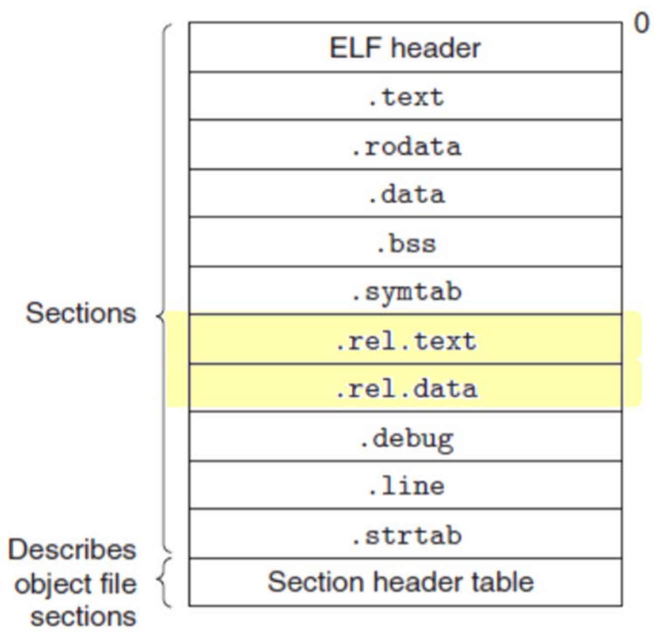
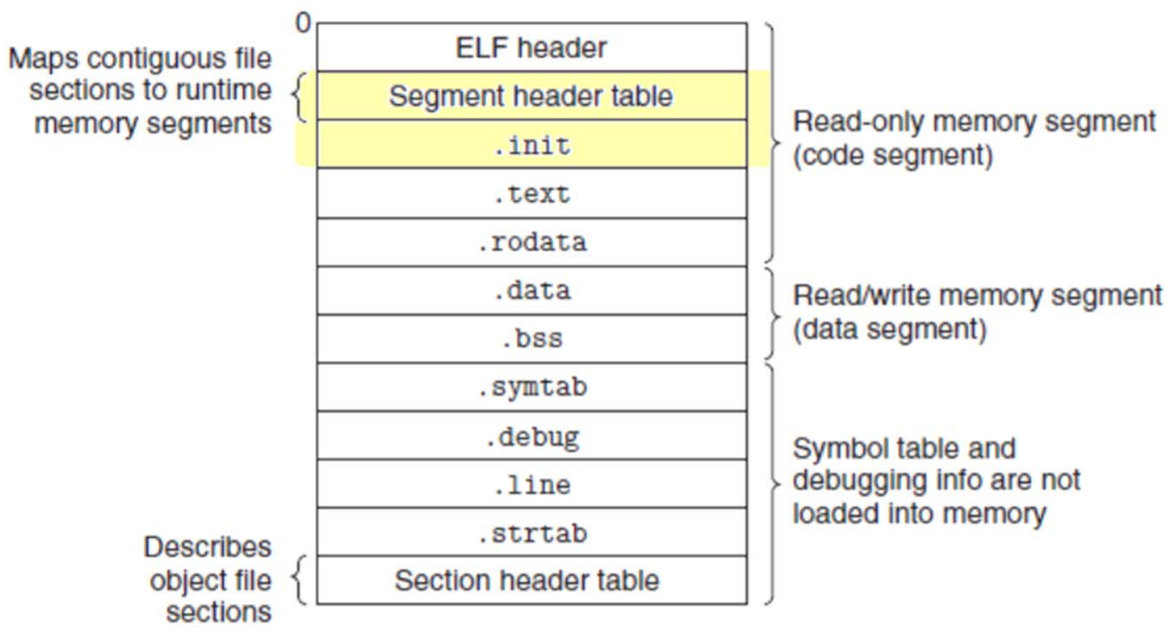
CSE320 System Fundamentals II
Linking 2

YoungMin Kwon

Executable Object Files



ELF relocatable object file

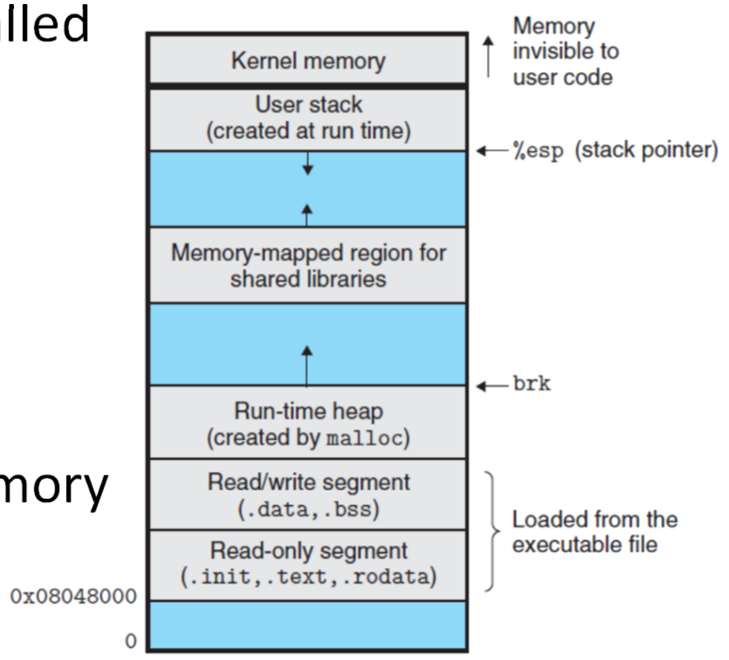


0
Maps contiguous file sections to runtime memory segments
ELF header
Segment header table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section header table
Describes object file sections
Read-only memory segment (code segment)
Read/write memory segment (data segment)
Symbol table and debugging info are not loaded into memory

ELF executable object file

ELF Executable Object Files

- .init section
  - Defines a function _init that will be called by the program's initialization code
- No .rel.text and .rel.data sections
  - The file is already relocated
- Program header table
  - Makes it easy to load the file into memory



Kernel memory
Memory invisible to user code
User stack (created at run time)
%esp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data,.bss)
Read-only segment (.init,.text,.rodata)
Loaded from the executable file
0x08048000
0

Read-only code segment
```
LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
     filesz 0x00000448 memsz 0x00000448 flags r-x
```

Read/write data segment
```
LOAD off    0x00000448 vaddr 0x08049448 paddr 0x08049448 align 2**12
     filesz 0x000000e8 memsz 0x00000104 flags rw-
```

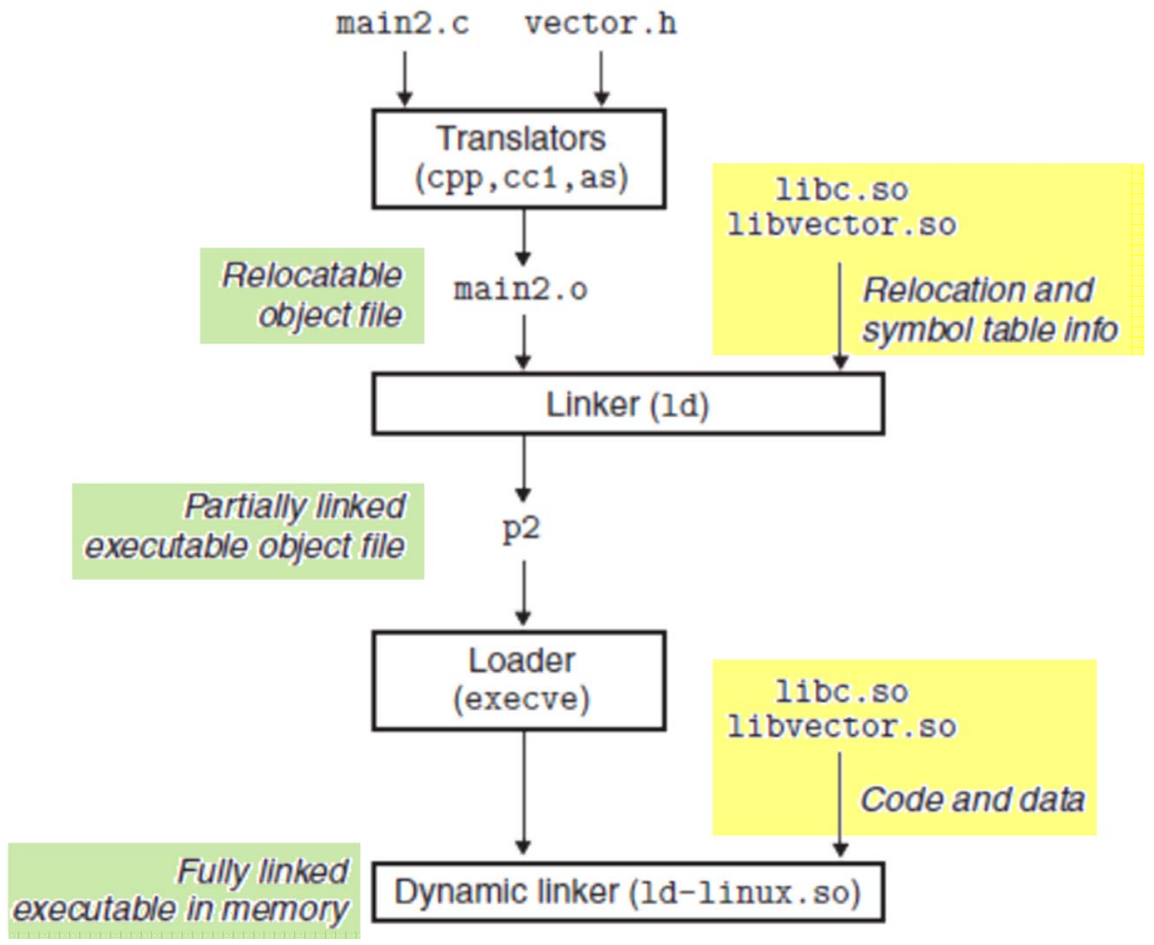
Dynamic Linking with Shared Libraries

- Shared libraries
  - An object module that can be loaded at an arbitrary memory address and linked with a program at run-time or load-time
  - This process is called dynamic linking

- To build a shared library
  - `gcc -shared -fpic –o libvector.so addvec.c mulvec.c`

- To link shared objects at **load-time**
  - `gcc main2.c ./libvector.so`

Dynamic Linking with Shared Libraries



Loading and Linking from Applications

- Dynamic linking at **run-time**
  - Applications can request the dynamic linker to load and link shared libraries at run-time

- Real world usages
  - Distributing software updates
  - High-performance web servers
    - Instead of creating a child process to run CGI programs, load, link, and run the appropriate functions directly

Loading and Linking from Applications

- Related functions:

```
#include <dlfcn.h>

// Loads and links the shared library filename
// flag: RTLD_GLOBAL, RTLD_NOW, RTLD_LAZY, ...
// Returns a pointer to handle or NULL on error
void *dlopen(const char *filename, int flag);

// Returns the address of the symbol or NULL
void *dlsym(void *handle, char *symbol);

// Unloads the shared library
int dlclose(void *handle);

// Returns an error message if previous call to
// dlopen, dlsym, or dlclose failed
const char *dlerror(void);
```

Loading and Linking from Applications

```c
//main_dynamic.c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#define ON_FALSE_EXIT(exp, msg) {\
    if(!(exp)) {\
        char *str = (char*)msg;\
        if(str && str[0] != '\0')\
            fprintf(stderr, "%s in file: %s, function: %s, line: %d\n",\
                str, __FILE__, __FUNCTION__, __LINE__);\
        exit(1);\
    }\
}
#define CHECKNULL_EXIT(p, msg)\
    ON_FALSE_EXIT((p) != NULL, msg)

void (*_addvec)(int *x, int *y, int *z, int n);
void (*_mulvec)(int *x, int *y, int *z, int n);
int *_addcnt;
int *_mulcnt;

int x[2] = {1, 2}, y[2] = {3, 4}, z[2];
```

Loading and Linking from Applications

```c
int main() {
    void *handle;
    handle = dlopen("./libvector.so", RTLD_LAZY|RTLD_GLOBAL);
    CHECKNULL_EXIT(_addvec = dlsym(handle, "addvec"), "dlsym");
    CHECKNULL_EXIT(_addcnt = dlsym(handle, "addcnt"), "dlsym");

    CHECKNULL_EXIT(_mulvec = dlsym(handle, "mulvec"), "dlsym");
    CHECKNULL_EXIT(_mulcnt = dlsym(handle, "mulcnt"), "dlsym");

    _addvec(x, y, z, 2);
    printf("z = [%d, %d]\n", z[0], z[1]);
    printf("addcnt = %d\n", *_addcnt);

    _mulvec(x, y, z, 2);
    printf("z = [%d, %d]\n", z[0], z[1]);
    printf("mulcnt = %d\n", *_mulcnt);

    dlclose(handle);
    return 0;
}
```

To compile: `gcc` ***`-rdynamic`*** `main_dynamic.c` ***`-ldl`***

Position-Independent Code (PIC)

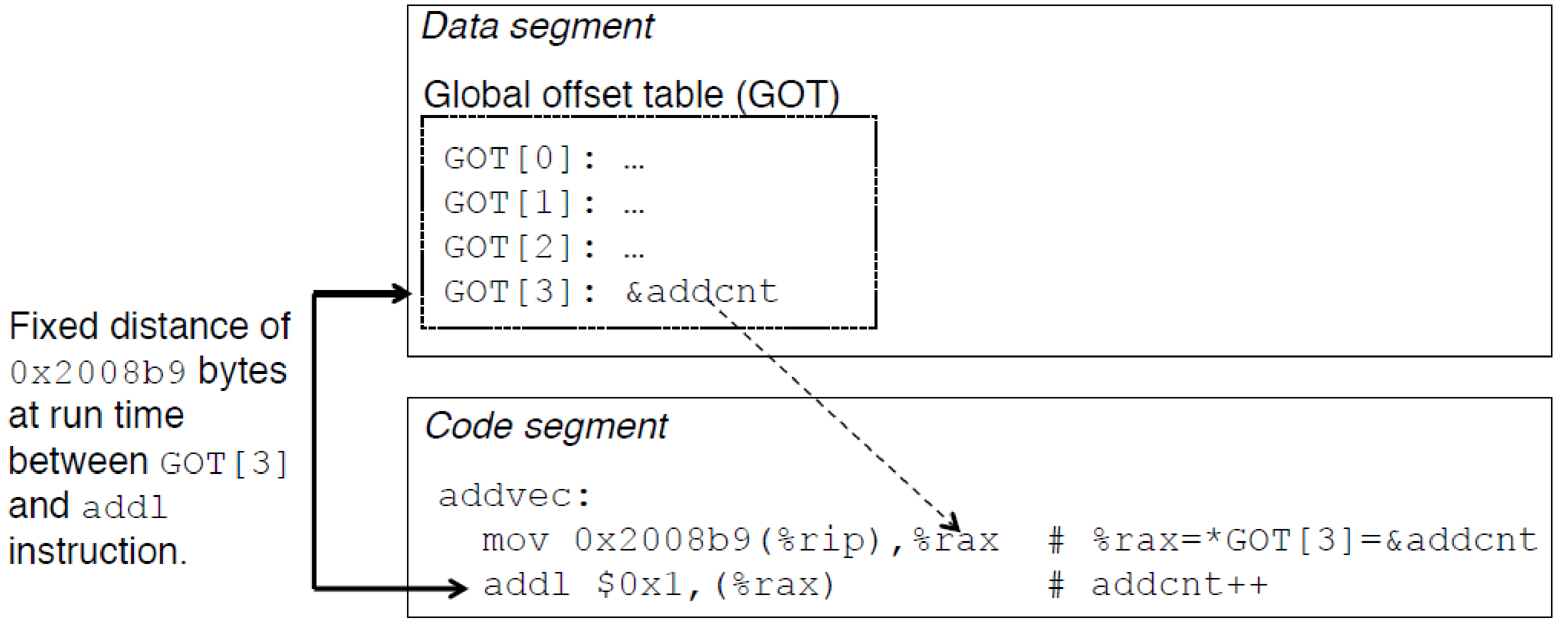
- How the same shared libraries are shared by multiple processes
- Position-Independent Code
  - Referencing symbols in the same executable object module $\rightarrow$ PC-relative addressing.
  - Referencing external symbols $\rightarrow$ GOT (Global Offset Table)
  - The code segments of shared modules can be loaded anywhere in memory without being modified by the linker
  - Each process will get its own copy of the data segment

PIC Data Reference

- Global Offset Table (GOT)
  - For each referenced global data objects (function, variable), a pointer entry is prepared that will be replaced by the absolute address of the object at the load-time

- PIC reference
  - The data segment is always at the same distance from the code segment
  - Place GOT at the beginning of the data segment
  - Each object module has its own GOT

PIC Data Reference



Data segment
Global offset table (GOT)
Code segment
Fixed distance of 0x2008b9 bytes at run time between GOT[3] and addl instruction.

```
GOT[0]: …
GOT[1]: …
GOT[2]: …
GOT[3]: &addcnt
```

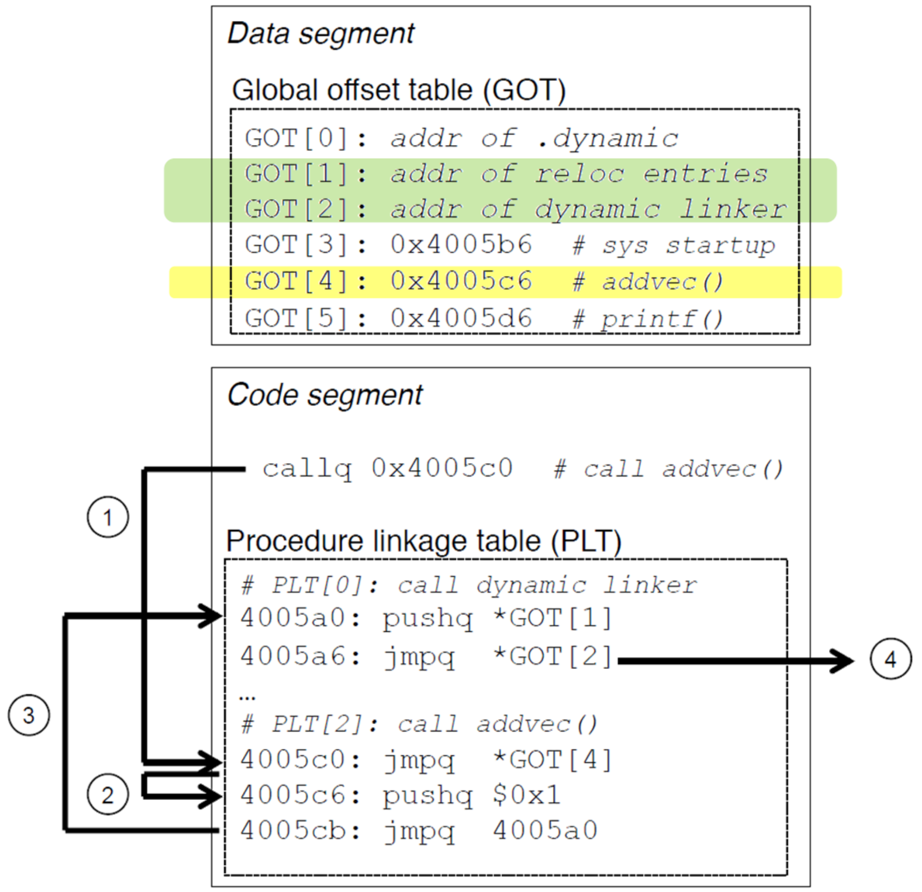
```
addvec:
  mov  0x2008b9(%rip),%rax   # %rax=*GOT[3]=&addcnt
  addl  $0x1,(%rax)          # addcnt++
```

- addvec loads the address of addcnt indirectly from GOT[3]
- The distance from %rip to GOT[3] is a constant (0x2008b9)

PIC Function Call

- PIC function call
  - Uses GOT and Procedure Linkage Table (PLT)
    - GOT is part of data segment
    - PLT is part of code segment
  - Lazy binding: defers the binding of each procedure address until the first time the procedure is called

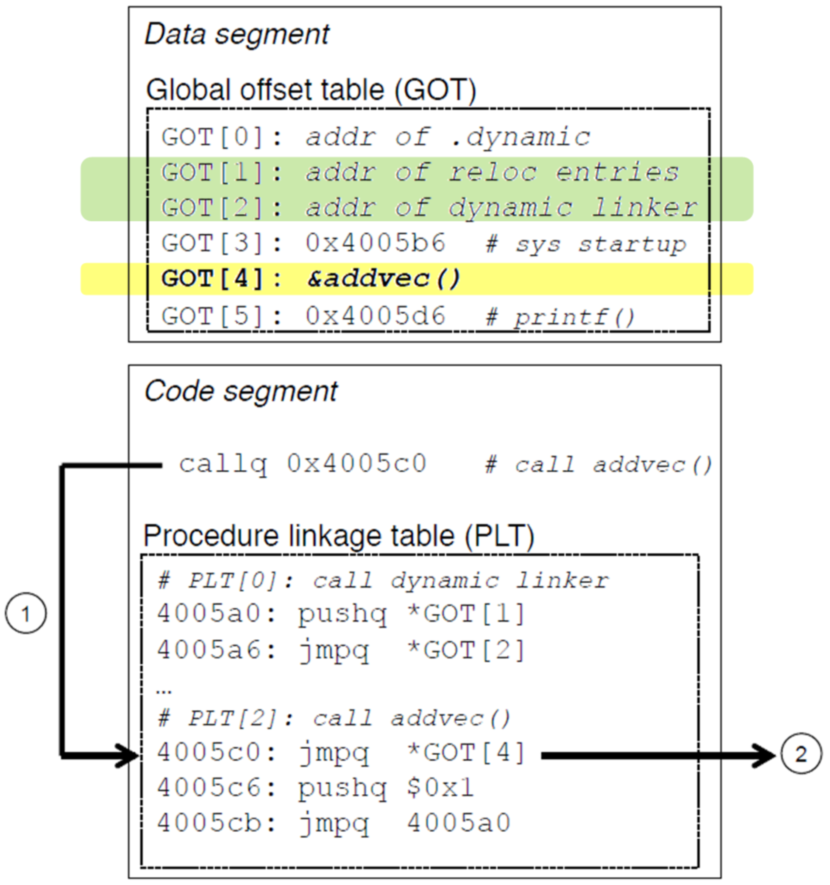
PIC Function Call



```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6   # sys startup
GOT[4]: 0x4005c6   # addvec()
GOT[5]: 0x4005d6   # printf()

  callq 0x4005c0   # call addvec()

# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
…
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```



```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6   # sys startup
GOT[4]: &addvec()
GOT[5]: 0x4005d6   # printf()

  callq 0x4005c0   # call addvec()

# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
…
# PLT[2]: call addvec()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

First invocation of addvec
- In pushq $0x1, the id of addvec is 1

Subsequent invocations of addvec

Thank you for your attention during the semester!

Any questions or comments?

- Please submit your Course Evaluation at

  https://p22.courseval.net/etw/ets/et.asp?nxappid=SU2&nxmid=start

  https://stonybrook.campuslabs.com/eval-home/