Linking II: Static and Dynamic Linking

COMP402127: Introduction to Computer Systems

Hao Li
Xi’an Jiaotong University

Today

- **Libraries and Static Linking**
- Dynamic Linking
- Case Study: Library Interpositioning

Libraries: Packaging a Set of Functions

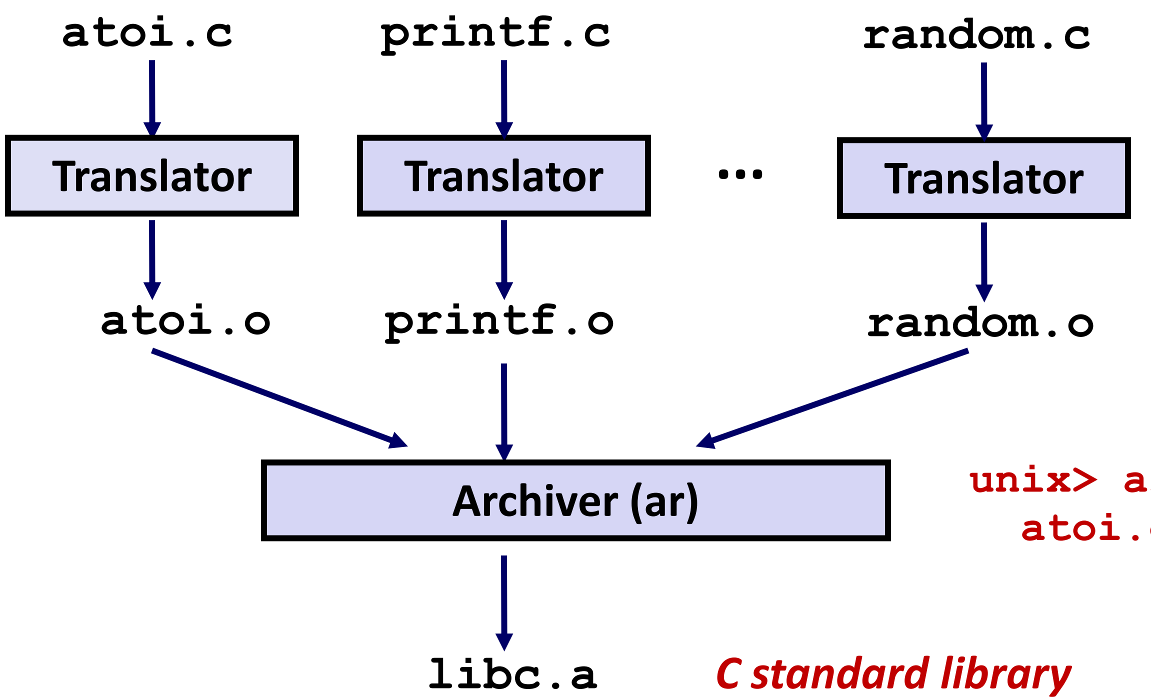
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **Awkward, given the linker framework so far:**
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Old-fashioned Solution: Static Libraries

- **Static libraries (.a archive files)**
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Creating Static Libraries




```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

C standard library

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

`libc.a` (the C standard library)

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries

```
#include <stdio.h>
#include "vector.h"

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

int main(int argc, char**
argv)
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n",
           z[0], z[1]);
    return 0;
}
```

main2.c

libvector.a

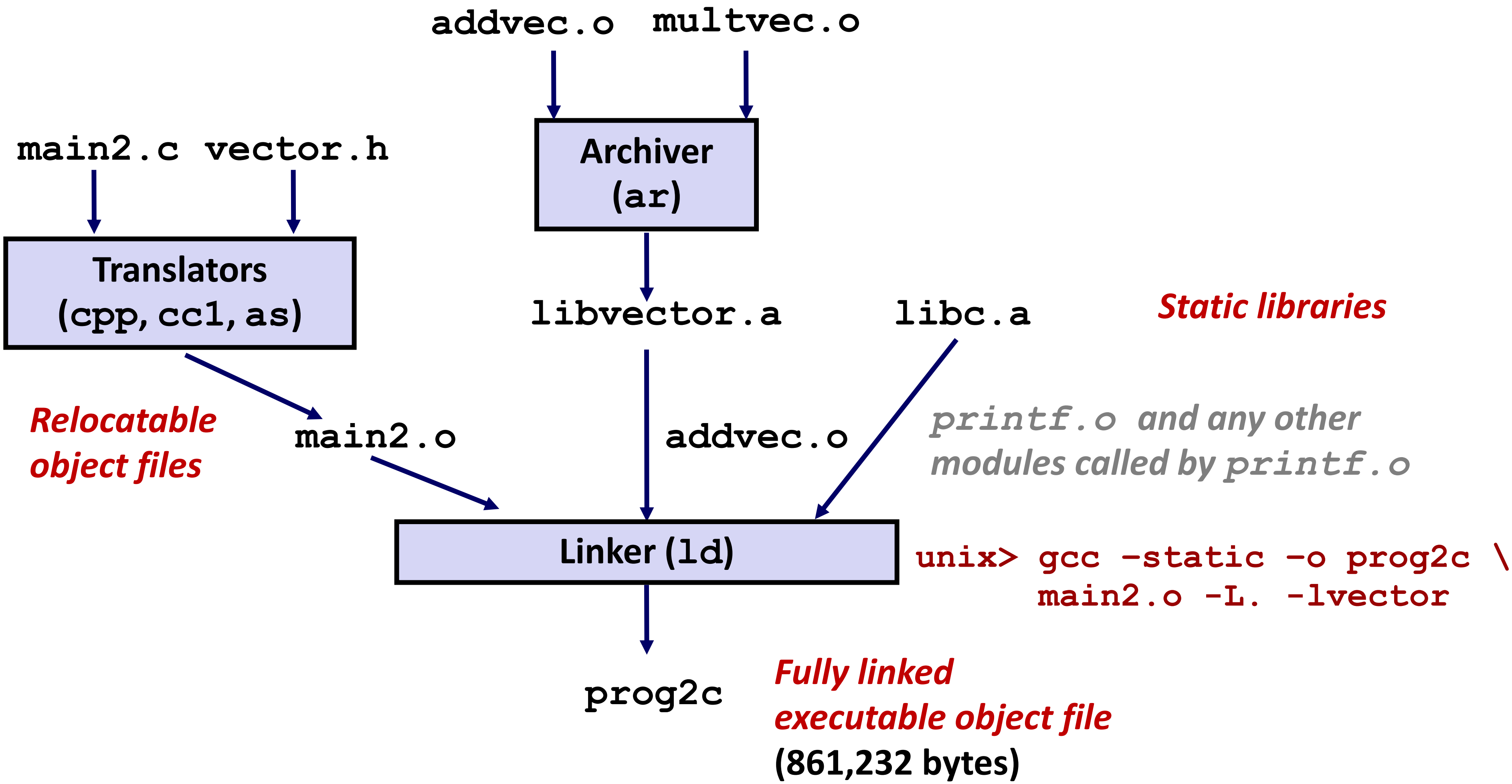
```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

```
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c

Linking with Static Libraries



“c” for “compile-time”

Using Static Libraries

- **Linker's algorithm for resolving external references:**
  - Scan `.o` files and `.a` files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.

- **Problem:**
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -static -o prog2c -L. -lvector main2.o
main2.o: In function `main':
main2.c:(.text+0x19): undefined reference to `addvec'
collect2: error: ld returned 1 exit status
```

Today

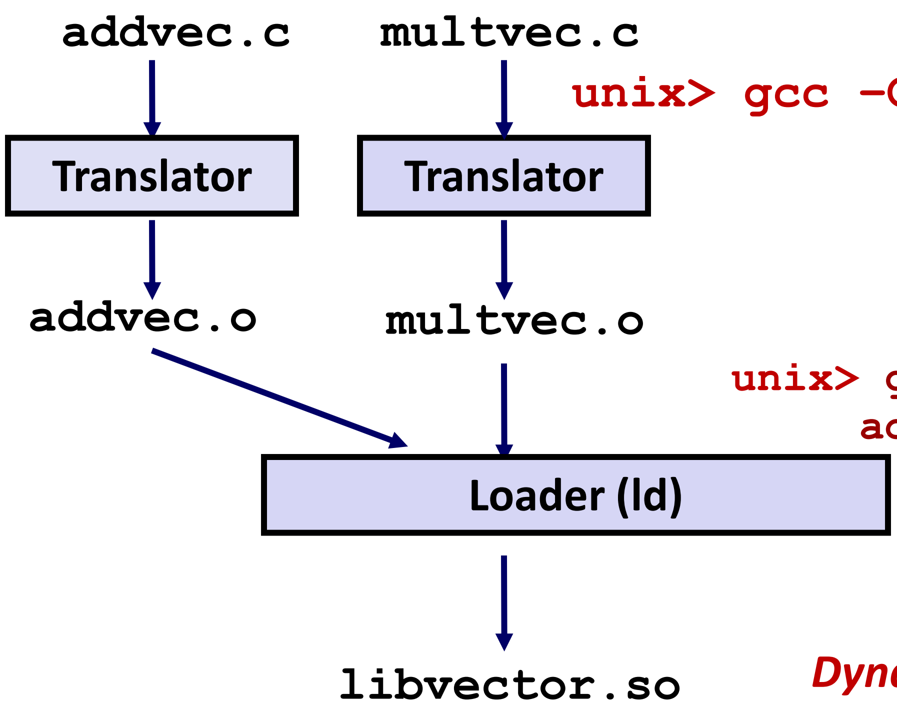
- Libraries and Static Linking
- **Dynamic Linking**
- Case Study: Library Interpositioning

Modern Solution: Shared Libraries

- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
    - Rebuild everything with glibc?
    - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

- **Modern solution: Shared Libraries**
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, `.so` files

Shared Library Example

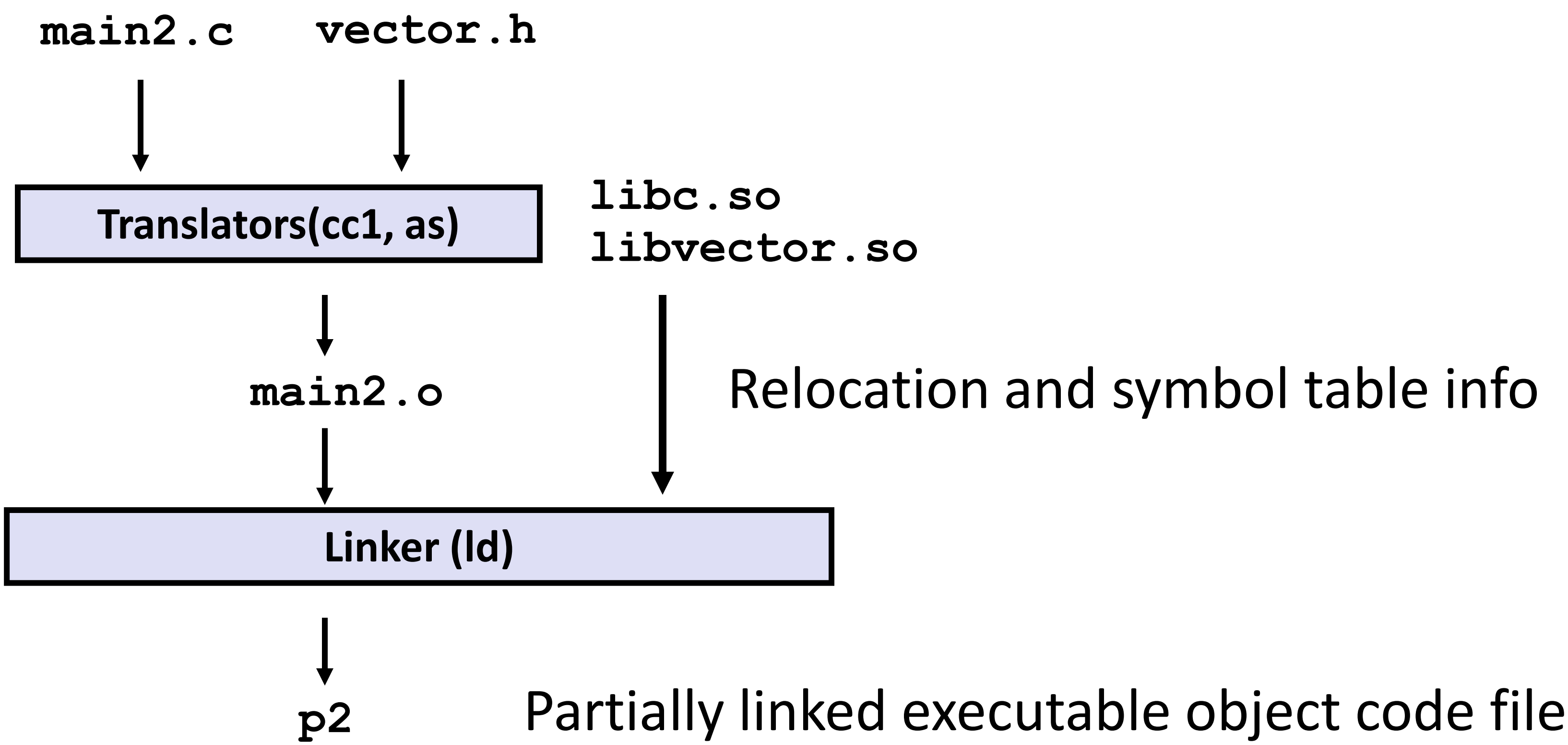



```
unix> gcc -Og -c addvec.c multvec.c -fpic
```

```
unix> gcc -shared -o libvector.so \
        addvec.o multvec.o
```

Dynamic vector library

Partially Linking with Shared Libraries

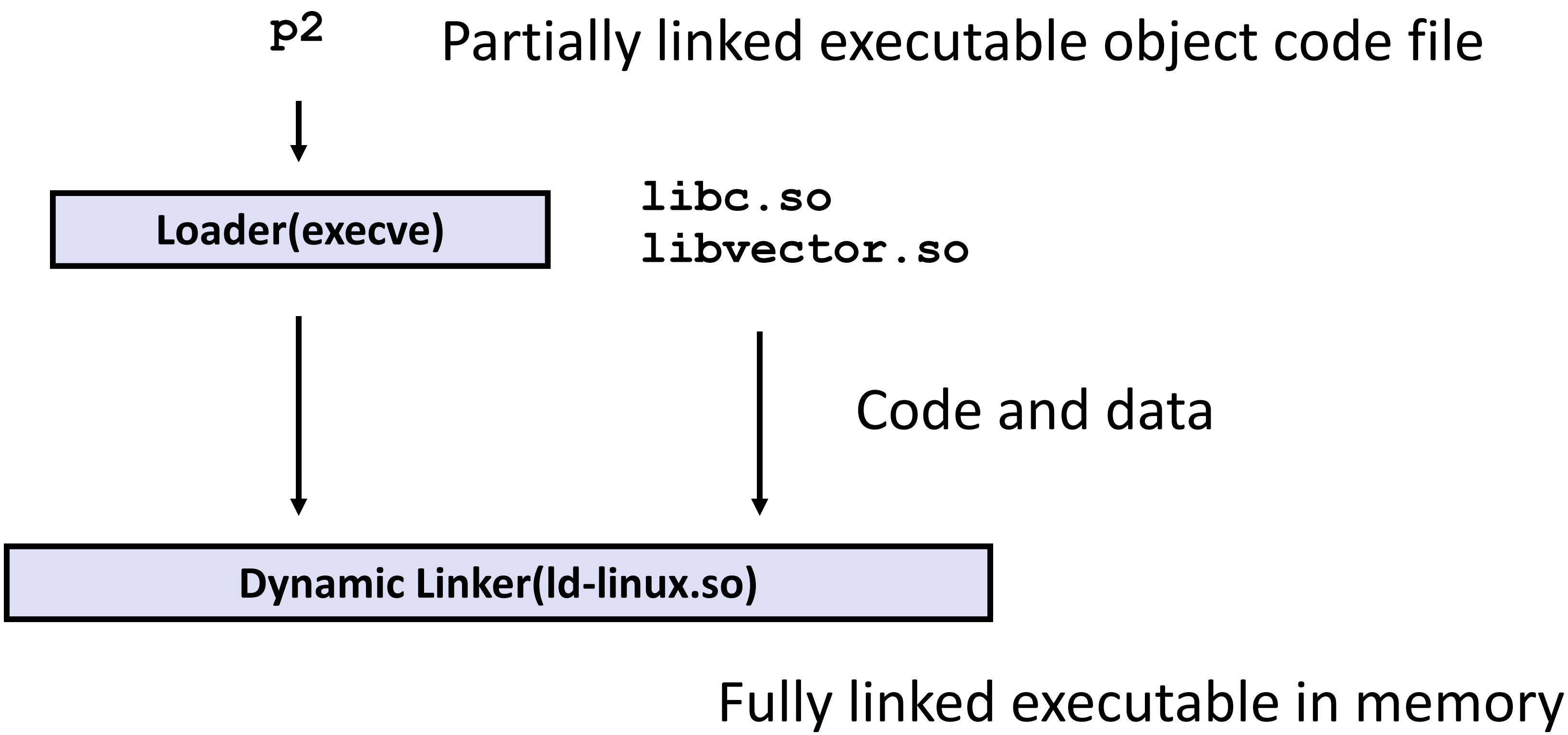



```
Unix>gcc -o p2 main2.c  ./libvector.so
```

Partially Linking with Shared Libraries

- Which parts in libvector.so are copied into p2
  - The code and data sections **No**
  - Relocation and symbol table information **Some**

Dynamic Linking at Load-time



What have done by dynamic linker?

- **Done by execve() & ld-linux.so**
  - Copy code and data of libc.so and libvector.so into some memory segments
  - Relocate any references in p2 to symbols defined by libc.so and libvector.so
- **After linking, the locations of the shared libraries are fixed and do not change during the execution time**

What dynamic libraries are required?

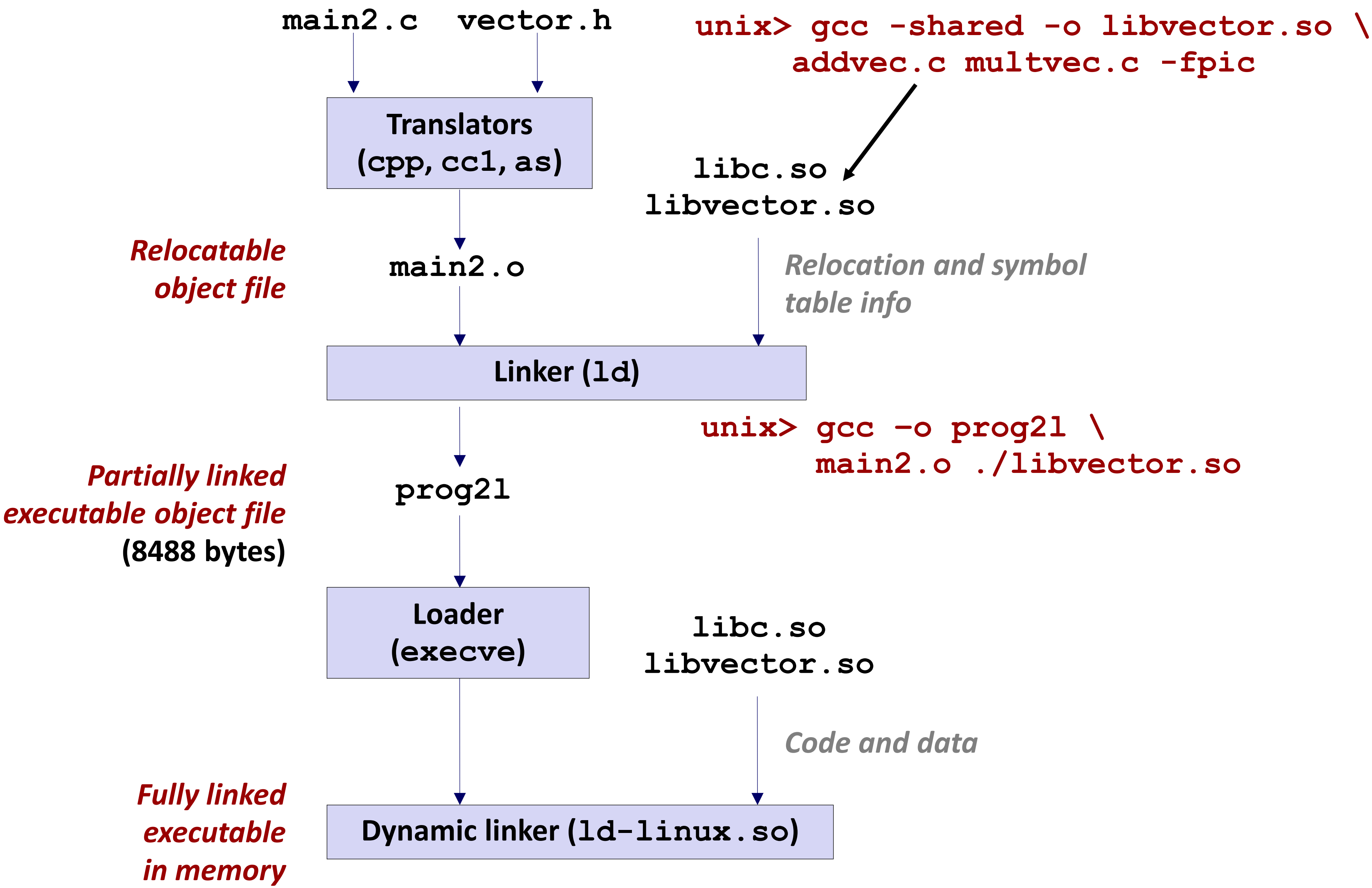
- **.interp section**
  - Specifies the dynamic linker to use (i.e., **`ld-linux.so`**)
- **.dynamic section**
  - Specifies the names, etc of the dynamic libraries to use
  - Follow an example of **`prog`**

```
(NEEDED)                    Shared library: [libm.so.6]
```

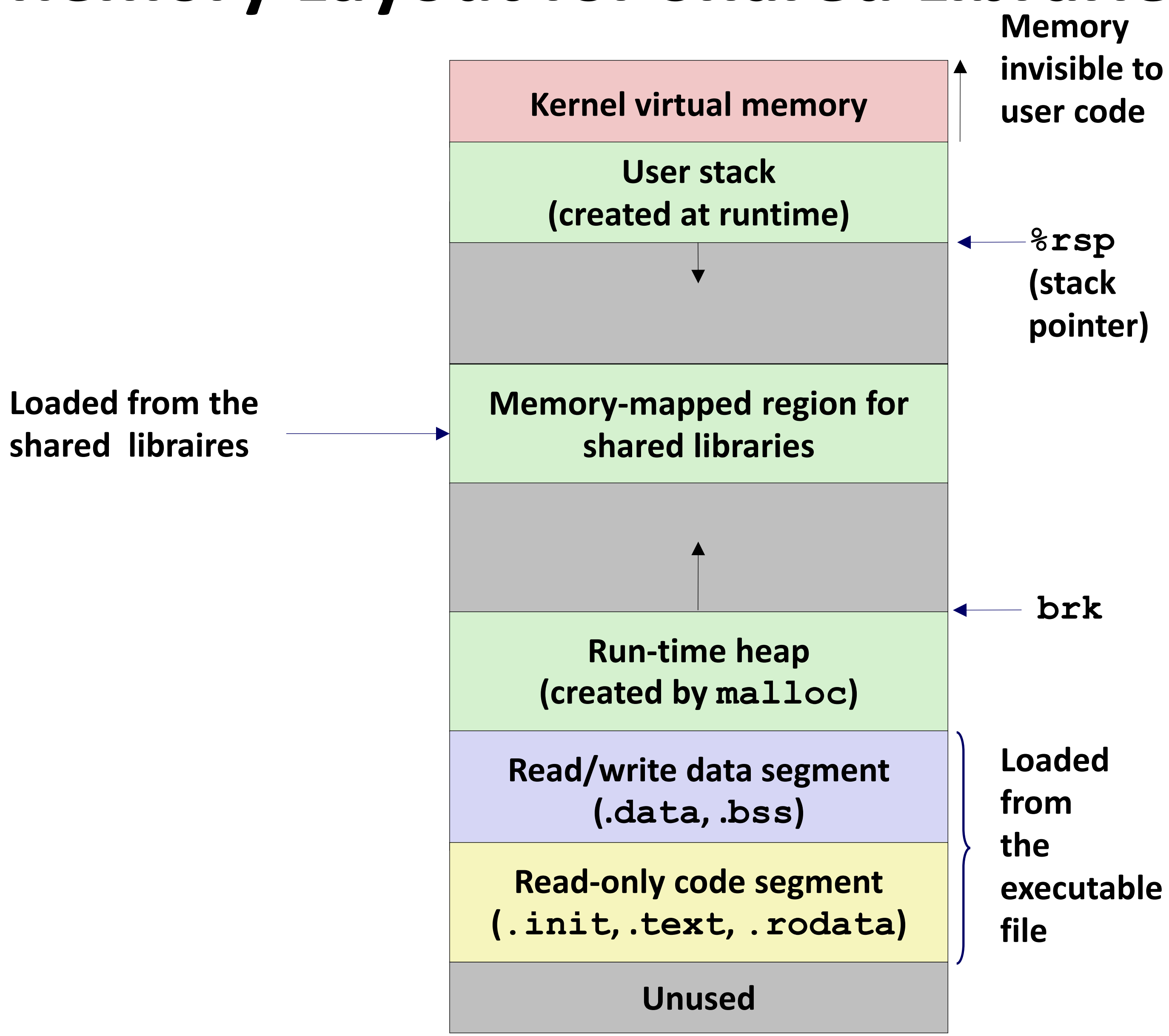
- **Where are the libraries found?**
  - Use "**`ldd`**" to find out:

```
unix> ldd prog
  linux-vdso.so.1 =>  (0x00007ffcf2998000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```

Dynamic Linking at Load-time (Complete)



Memory Layout for Shared Libraries



Dynamic Linking at Runtime

- **Dynamic linking can occur when executable is first loaded and run (load-time linking).**
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.

- **Dynamic linking can also occur after program has begun (run-time linking).**
  - In Linux, this is done by calls to the `dlopen()` interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.

Why Linking at Run-time?

- **Distributing software**
  - Developers of Microsoft Windows applications frequently use shared libraries to distribute software updates.
  - They generate a new copy of a shared library, which users can then download and use as a replacement for the current version.
  - The next time they run their application, it will automatically link and load the new shared library.

Why Linking at Run-time?

- **Building high-performance Web servers**
  - Modern high-performance Web servers can generate dynamic content using a more efficient and sophisticated approach based on dynamic linking.
  - package each function that generates dynamic content in a shared library.
  - When a request arrives from a Web browser, the server dynamically loads and links the appropriate function and then calls it directly.

Why Linking at Run-time?

- **Building high-performance Web servers**
  - The function remains cached in the server's address space, so subsequent requests can be handled at the cost of a simple function call.
  - This can have a significant impact on the throughput of a busy site.
  - Further, existing functions can be updated, and new functions can be added at run time, without stopping the server.

Dynamic Linking at Run-time

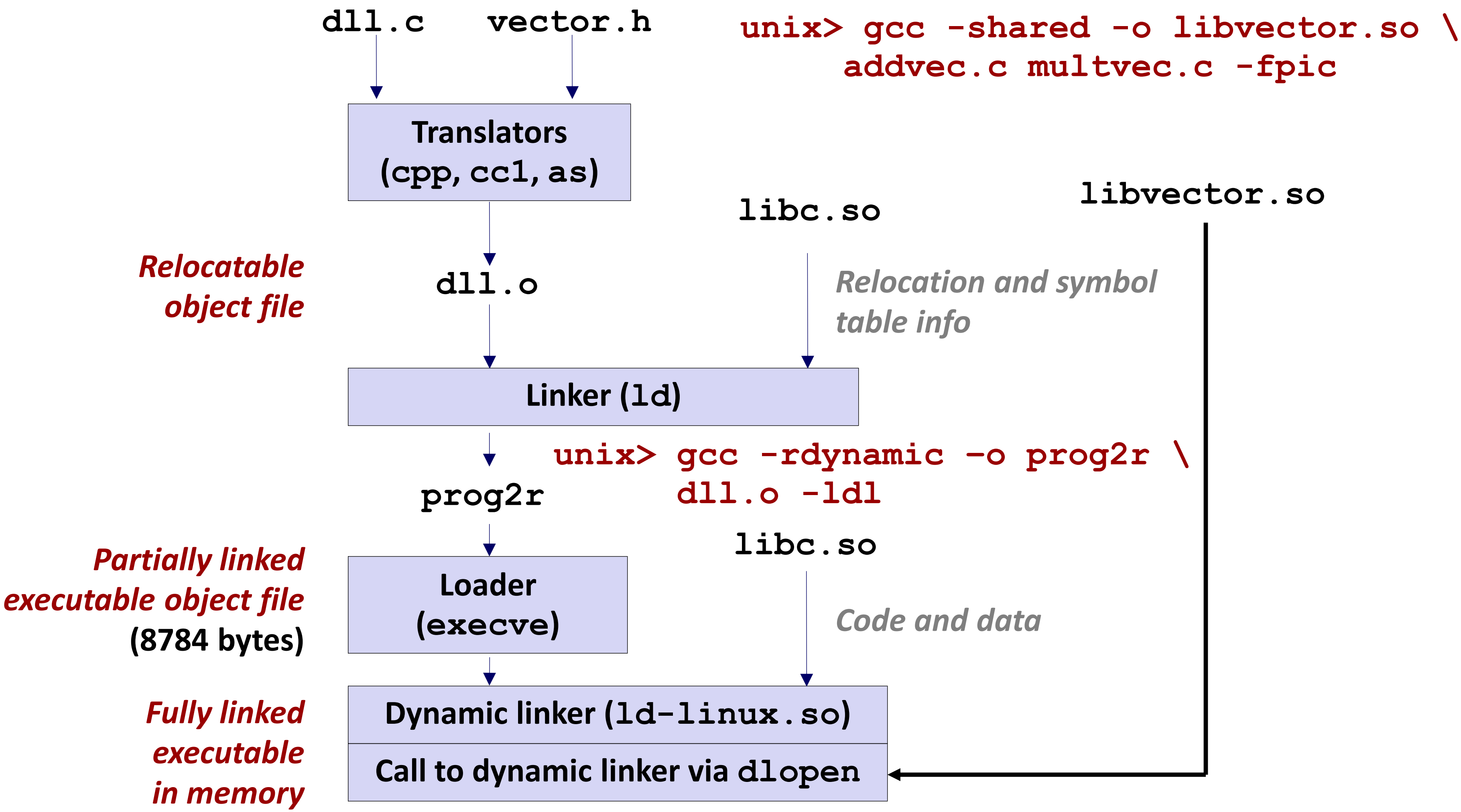
```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

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

int main(int argc, char** argv)
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
  . . .                                                              dll.c
```

Dynamic Linking at Run-time (cont)

```
    ...

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

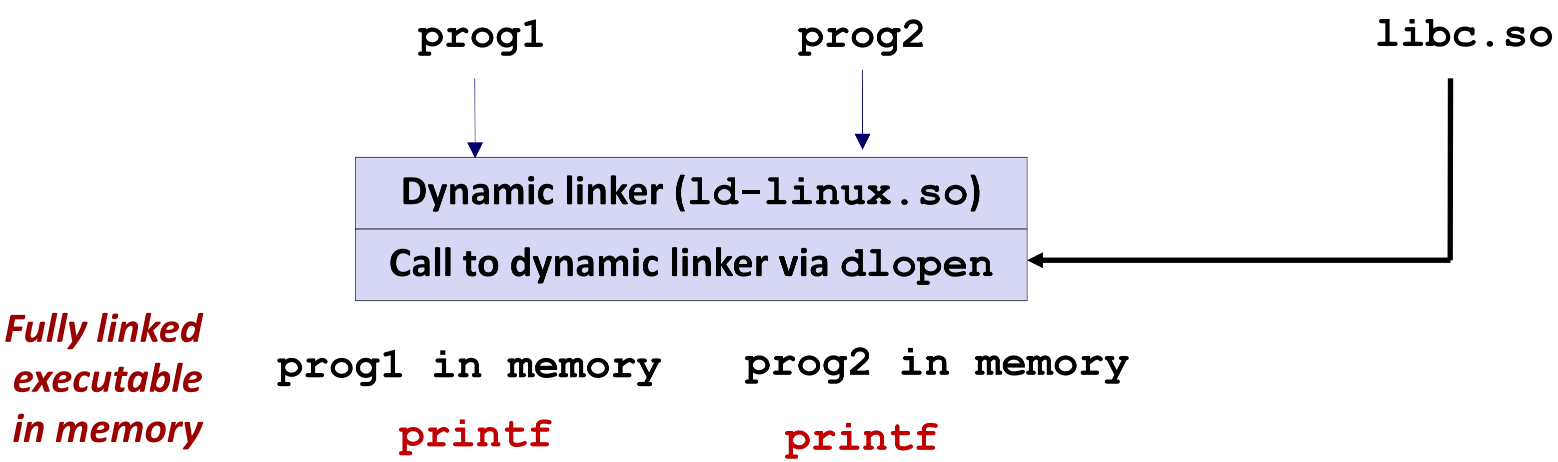
dll.c

Dynamic Linking at Run-time



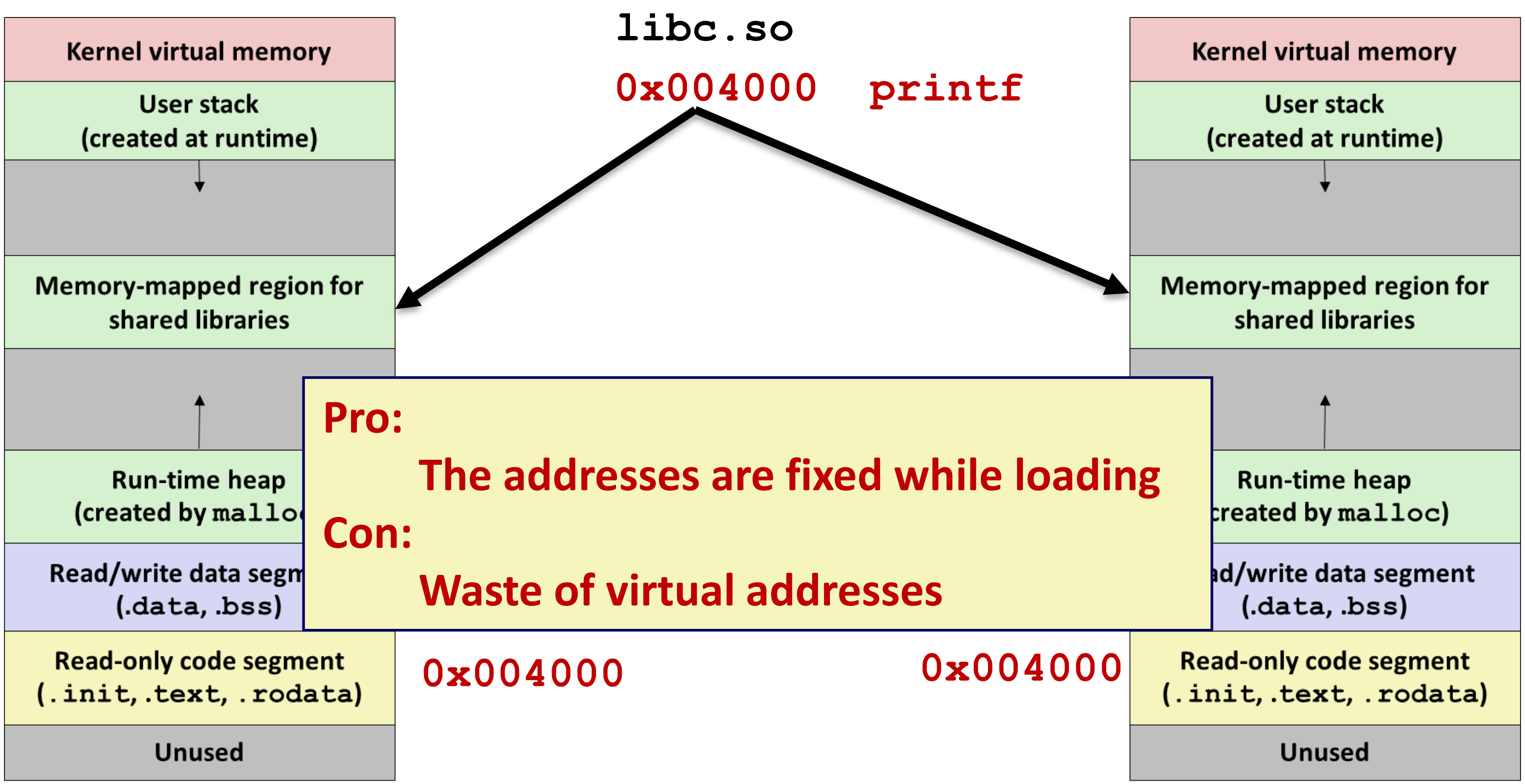
Share Libraires across Executables

- Space: Libraries. How do libraries save space?
  - Option 2: Dynamic linking
    - Executable files contain no library code
    - **During execution, single copy of library code can be shared across all executing processes**



How to know the address of `printf`?

- **Naïve Solution: Fixed address**
  - `libc.so` fixes the address of each function
  - Process reserves those addresses while loading



Pro:
The addresses are fixed while loading

Con:
Waste of virtual addresses

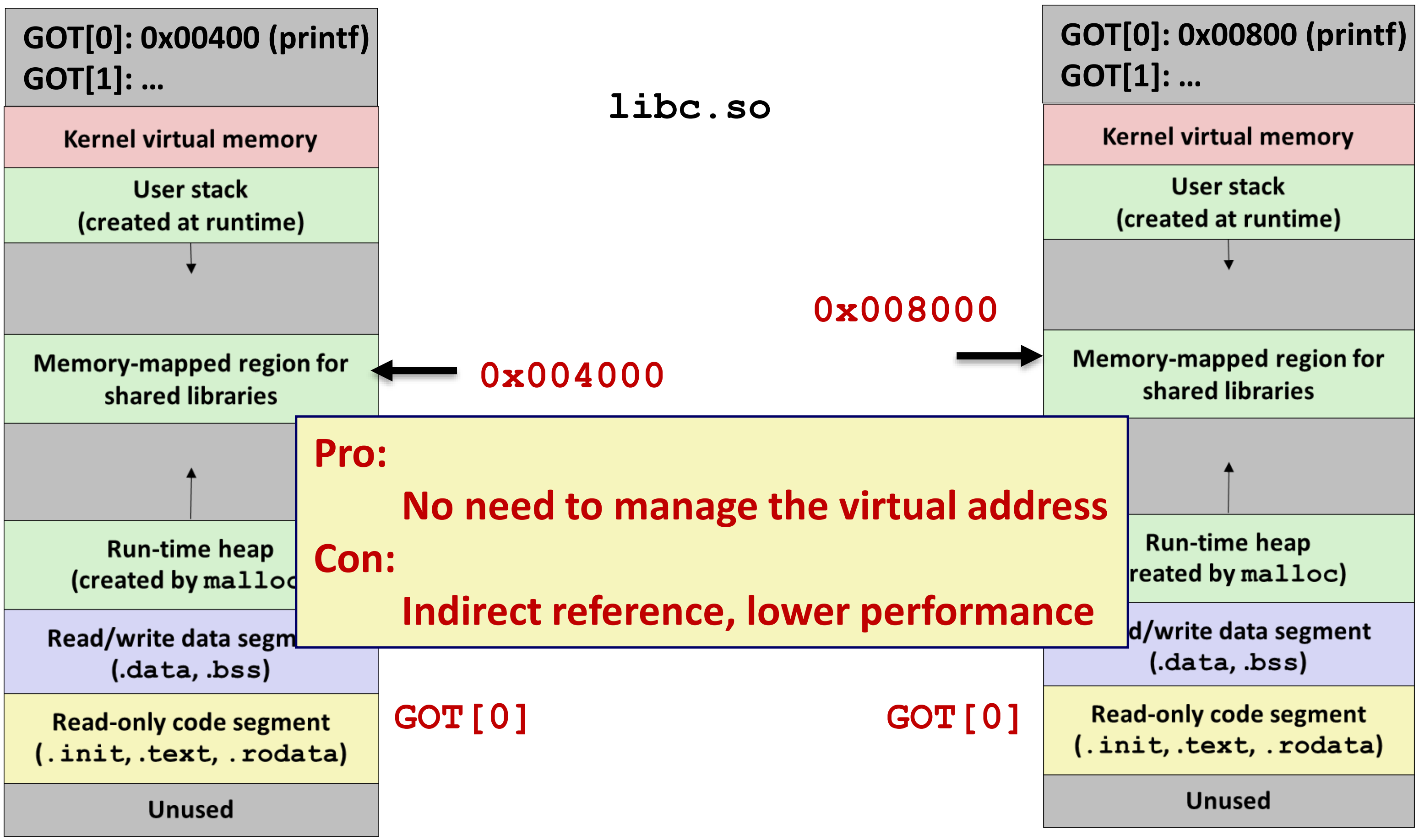
Position Independent Code (PIC)

- Code that can be execute from any address
- Internally-defined procedures
  - PC-relative reference

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08          sub    $0x8,%rsp
  4004d4:       be 02 00 00 00       mov    $0x2,%esi
  4004d9:       bf 18 10 60 00       mov    $0x601018,%edi   # %edi = &array
  4004de:       e8 05 00 00 00       callq  4004e8 <sum>     # sum()
  4004e3:       48 83 c4 08          add    $0x8,%rsp
  4004e7:       c3                   retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00             mov    $0x0,%eax
  4004ed:       ba 00 00 00 00             mov    $0x0,%edx
  4004f2:       eb 09                      jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                   movslq %edx,%rcx
  4004f7:       03 04 8f                   add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                   add    $0x1,%edx
  4004fd:       39 f2                      cmp    %esi,%edx
  4004ff:       7c f3                      jl     4004f4 <sum+0xc>
  400501:       f3 c3                      repz retq
```

Position Independent Code (PIC)

- Externally-defined procedures and global variables
  - Global offset table (GOT)



Pro:

No need to manage the virtual address

Con:

Indirect reference, lower performance

Position-Independent Code (PIC)

- **PIC Data References**

Data segment

Global offset table (GOT)

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

Code segment

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

Fixed distance of 0x2008b9 bytes at run time between GOT[3] and addl instruction

31

Linking Summary

- **Linking is a technique that allows programs to be constructed from multiple object files.**

- **Linking can happen at different times in a program’s lifetime:**
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

- **Understanding linking can help you avoid nasty errors and make you a better programmer.**

Today

- Libraries and Static Linking
- Dynamic Linking
- **Case Study: Library Interpositioning**

Case Study: Library Interpositioning

- **Library interpositioning : powerful linking technique that allows programmers to intercept calls to arbitrary functions**
- **Interpositioning can occur at:**
  - Compile time: When the source code is compiled
  - Link time: When the relocatable object files are statically linked to form an executable object file
  - Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

Some Interpositioning Applications

- **Security**
  - Confinement (sandboxing)
  - Behind the scenes encryption
- **Debugging**
  - In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
  - Code in the SPDY networking stack was writing to the wrong location
  - Solved by intercepting calls to Posix write functions (write, writev, pwrite)

Source: Facebook engineering blog post at:
https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/

Some Interpositioning Applications (cont)

- **Monitoring and Profiling**
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - Malloc tracing
    - Detecting memory leaks
    - **Generating address traces**
- **Error Checking**
  - C Programming Lab used customized versions of malloc/free to do careful error checking
  - Other labs (malloc, shell, proxy) also use interpositioning to enhance checking capabilities

Example program

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int i;
  for (i = 1; i < argc; i++) {
    void *p =
          malloc(atoi(argv[i]));
    free(p);
  }
  return(0);
}
```

int.c

- Goal: trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.
- Three solutions: interpose on the library `malloc` and `free` functions at compile time, link time, and load/run time.

Compile-time Interpositioning

- **You have a file that calls libc's malloc and free functions**
  - int.c
- **You have your own implementation of malloc and free**
  - mymalloc.c
  - void *mymalloc(size_t size)
  - void myfree(void *ptr)
- **How do you call mymalloc instead of malloc in int.c without modifying int.c?**
  - Assume you can recompile int.c but cannot modify int.c

Compile-time Interpositioning

```
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Compile-time Interpositioning

```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```

malloc.h

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc 10 100 1000
malloc(10)=0x1ba7010
free(0x1ba7010)
malloc(100)=0x1ba7030
free(0x1ba7030)
malloc(1000)=0x1ba70a0
free(0x1ba70a0)
linux>
```

Search for <malloc.h> leads to /usr/include/malloc.h

Search for <malloc.h> leads to

Link-time Interpositioning

- **You have a file that calls libc's malloc and free functions**
  - int.c
- **You have your own implementation of malloc and free**
  - mymalloc.c
  - void *mymalloc(size_t size)
  - void myfree(void *ptr)
- **How do you call mymalloc instead of malloc in int.c without modifying and recompiling int.c?**
  - You cannot modify or recompile int.c

Link-time Interpositioning

```
#ifdef LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Link-time Interpositioning

```
linux> make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl \
    int.o mymalloc.o
linux> make runl
./intl 10 100 1000
malloc(10) = 0x91a010
free(0x91a010)
. . .
```

Search for `<malloc.h>` leads to `/usr/include/malloc.h`

- **The "`-Wl`" flag passes argument to linker, replacing each comma with a space.**
- **The "`--wrap,malloc`" `arg` instructs linker to resolve references in a special way:**
  - Refs to `malloc` should be resolved as `__wrap_malloc`
  - Refs to `__real_malloc` should be resolved as `malloc`

Load/Runtime Interpositioning

- **You have a file that calls libc's malloc and free functions**
  - int.c
- **You have your own implementation of malloc and free**
  - mymalloc.c
  - void *mymalloc(size_t size)
  - void myfree(void *ptr)
- **How do you call mymalloc instead of malloc in int.c without modifying, recompiling or relinking int.c?**
  - You cannot modify or recompile int.c
  - You cannot relink the executable

Load/Run-time Interpositioning

```
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Observe that DON'T have `#include <malloc.h>`

Load/Run-time Interpositioning

```
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Load/Run-time Interpositioning

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr 10 100 1000)
malloc(10) = 0x91a010
free(0x91a010)
. . .
linux>
```

Search for `<malloc.h>` leads to `/usr/include/malloc.h`

- The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved refs (e.g., to `malloc`) by looking in `mymalloc.so` first.
- Type into (some) shells as:

```
env LD_PRELOAD=./mymalloc.so ./intr 10 100 1000)
```

Interpositioning Recap

- **Compile Time**
  - Apparent calls to **`malloc`**/**`free`** get macro-expanded into calls to **`mymalloc`**/**`myfree`**
  - Simple approach. Must have access to source & recompile
- **Link Time**
  - Use linker trick to have special name resolutions
    - **`malloc`** → **`__wrap_malloc`**
    - **`__real_malloc`** → **`malloc`**
- **Load/Run Time**
  - Implement custom version of **`malloc`**/**`free`** that use dynamic linking to load library **`malloc`**/**`free`** under different names
  - Can use with ANY dynamically linked binary

```
env LD_PRELOAD=./mymalloc.so gcc -c int.c)
```

Linking Recap

- **Usually: Just happens, no big deal**
- **Sometimes: Strange errors**
  - Bad symbol resolution
  - Ordering dependence of linked .o, .a, and .so files
- **For power users:**
  - Interpositioning to trace programs with & without source