LẬP TRÌNH HỆ THỐNG

ThS. Đỗ Thị Thu Hiền
(hiendtt@uit.edu.vn)

TRƯỜNG ĐH CÔNG NGHỆ THÔNG TIN - ĐHQG-HCM
KHOA MẠNG MÁY TÍNH & TRUYỀN THÔNG
FACULTY OF COMPUTER NETWORK AND COMMUNICATIONS
Tầng 8 - Tòa nhà E, trường ĐH Công nghệ Thông tin, ĐHQG-HCM
Điện thoại: (08)3 725 1993 (122)

Linking

Ví dụ:
Chương trình C có code trong nhiều file

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

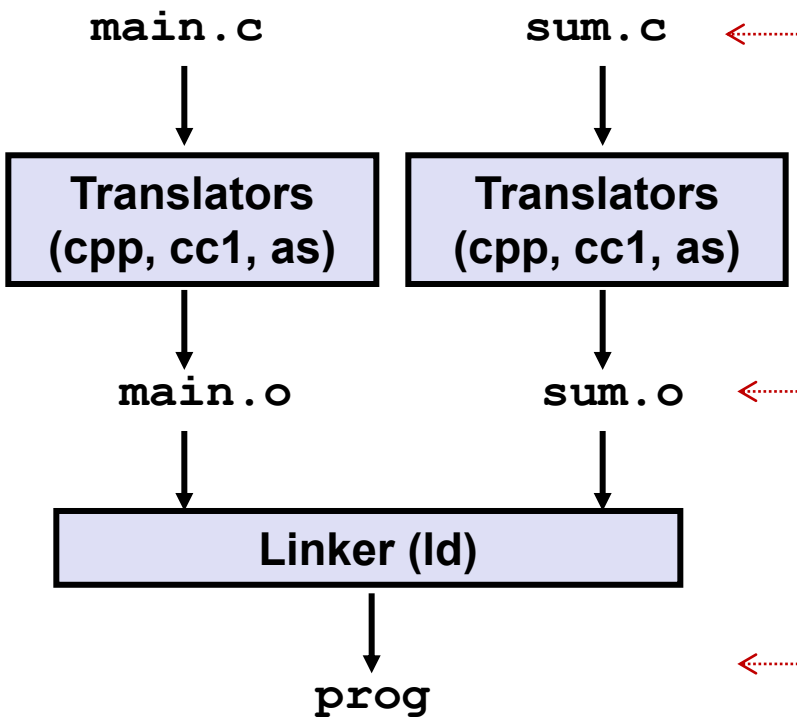
```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

File **main.c** có sử dụng hàm **sum()** được định nghĩa trong file **sum.c**

Static linking – Liên kết tĩnh

- **Các chương trình được biên dịch và liên kết bằng *compiler***
  - `linux> gcc -o prog main.c sum.c`
  - `linux> ./prog`



Các file mã nguồn

Các file object (relocatable object files) được biên dịch riêng biệt

File object thực thi được đã liên kết đầy đủ (chứa code và dữ liệu cho tất cả functions được định nghĩa trong ***`main.c`*** *và* ***`sum.c`****)*

3 kiểu Object Files (Modules)

- **Relocatable object file (`.o` file)**
  - Chứa code và data ở dạng có thể kết hợp với các relocatable object file khác để tạo thành file thực thi.
    - Mỗi file `.o` file được tạo từ chính xác 1 file source (`.c`)

- **Executable object file (`a.out` file)**
  - Chứa code và data ở dạng có thể sao chép trực tiếp lên bộ nhớ và thực thi.

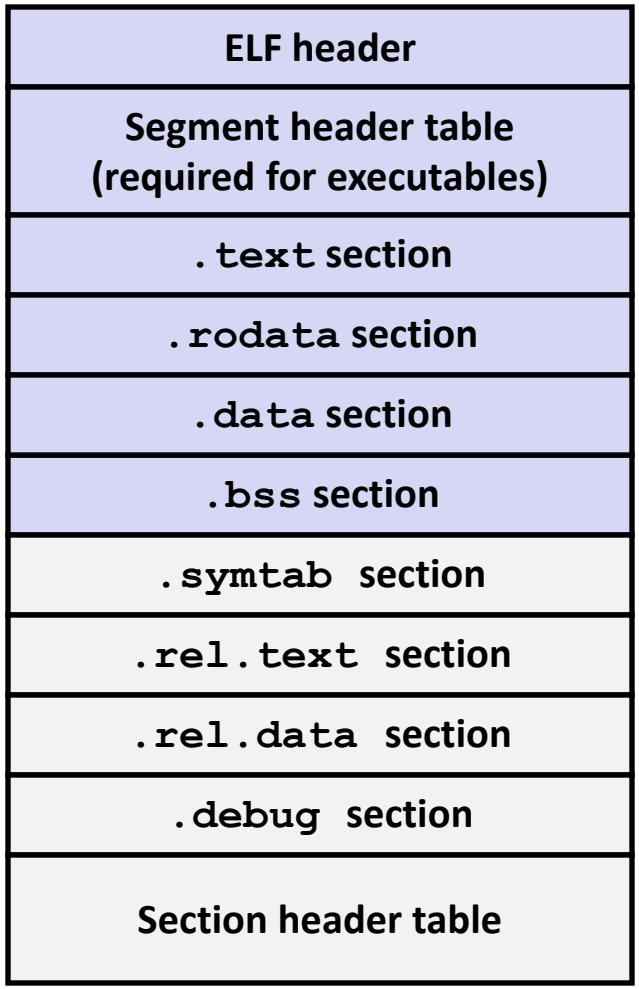
- **Shared object file (`.so` file)**
  - Dạng đặc biệt của relocatable object file, có thể được tải lên bộ nhớ và liên kết động, tại thời điểm load-time hoặc run-time.
  - Được gọi là *Dynamic Link Libraries* (DLLs) trong Windows

Định dạng ELF: Executable and Linkable Format

- **Định dạng nhị phân chuẩn cho các object files**
- **Một định dạng chung cho:**
  - Relocatable object files (`.o`),
  - Executable object files (`a.out`)
  - Shared object files (`.so`)

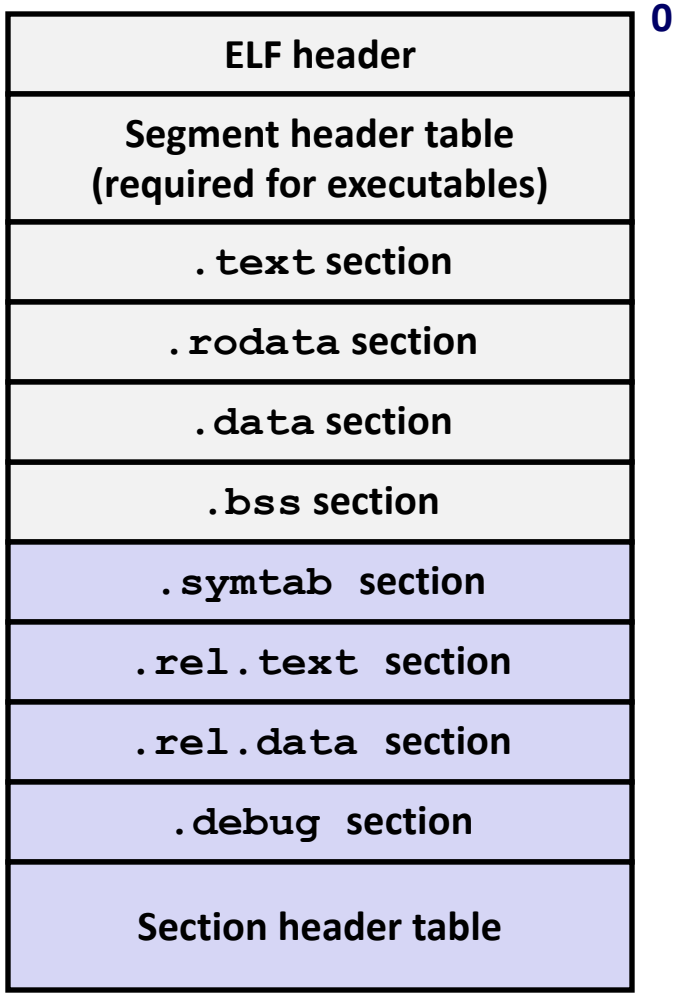
ELF Object File Format

- **Elf header**
  - Word size, byte ordering, kiểu file (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- **`.text` section:** Code
- **`.rodata` section**
  - Dữ liệu chỉ đọc như jump tables, ...
- **`.data` section**
  - Các biến toàn cục đã được khởi tạo giá trị
- **`.bss` section**
  - Biến toàn cục chưa được khởi tạo giá trị
  - "**B**lock **S**tarted by **S**ymbol"/ "**B**etter **S**ave **S**pace"
  - Có section header nhưng không chiếm không gian nào



ELF Object File Format (tt)

- **`.symtab` section**
  - Symbol table
  - Hàm và các biến toàn cục, static
  - Tên section và vị trí
- **`.rel.text` section**
  - Thông tin tái cấu trúc cho `.text` section
  - Địa chỉ của một số instructions cần phải thay đổi trong file thực thi
  - Hướng dẫn cho việc thay đổi.
- **`.rel.data` section**
  - Thông tin tái cấu trúc cho `.data` section
  - Địa chỉ của một số dữ liệu con trỏ cần thay đổi trong file thực thi tổng hợp.
- **`.debug` section**
  - Thông tin cho debug (`gcc -g`)
- **Section header table**
  - Offsets và kích thước của mỗi section



Vì sao phải sử dụng Linker?

- **Lý do 1: Tính mô-đun**
  - Chương trình có thể được viết dưới dạng gồm nhiều file source nhỏ thay vì viết nguyên một file source code lớn.
  - Có thể build thư viện của một số hàm thông dụng
    - Ví dụ: Thư viện toán (Math library), các thư viện C chuẩn

Vì sao phải sử dụng Linker?

- **Lý do 2: Tính hiệu quả**
  - Về thời gian: Biên dịch riêng biệt
    - Thay đổi 1 file source thì chỉ biên dịch lại 1 file source đó và liên kết lại.
    - Không cần phải biên dịch lại các file source khác.
  - Về không gian: Libraries – Các thư viện
    - Các hàm thông dụng có thể tích hợp vào chung 1 file
    - Tuy nhiên các file thực thi và bộ nhớ đang thực thi chỉ chứa code của những function mà nó sử dụng.

Linker sẽ làm gì?

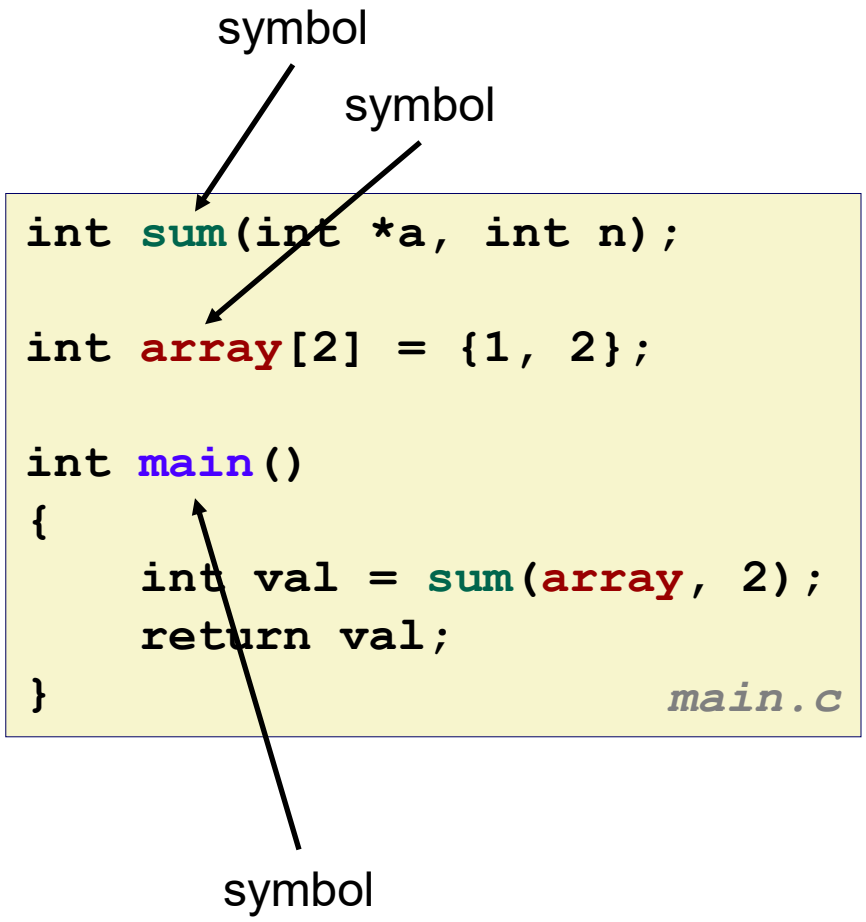
- **Hành động 1:** Ánh xạ các tham chiếu hàm/biến đến định nghĩa tương ứng của chúng trong các file code
  - Tham chiếu và định nghĩa có thể nằm cùng file hoặc khác file

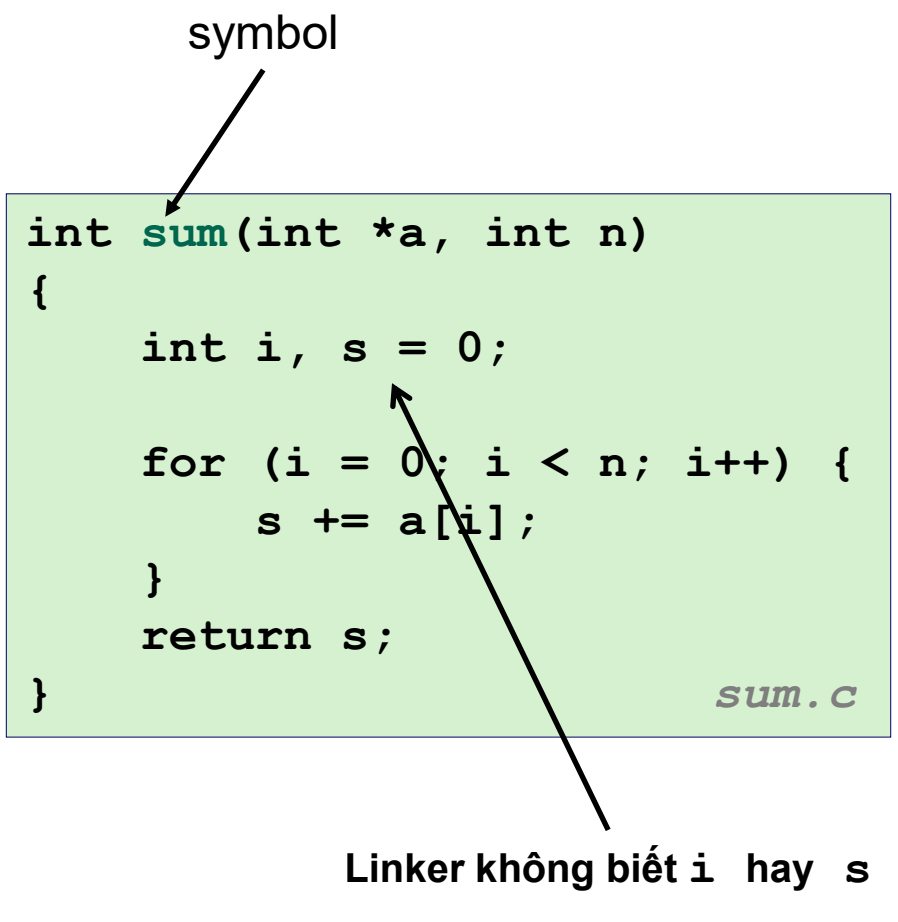
- **Hành động 2:** Sắp xếp lại code/dữ liệu của các file code một cách phù hợp trong file thực thi cuối cùng

Linker: Khái niệm Symbol

- **Symbol là gì?**
  - Symbol là các hàm (function) hoặc các biến toàn cục
  - Các chương trình có thể định nghĩa hoặc tham chiếu đến ***symbol***
    - `void swap() {…}  /* define symbol swap */`
    - `swap();  /* reference symbol swap */`
    - `int *xp = &x;  /* define symbol xp, reference x */`
  - Các symbol được lưu trong file object trong *symbol table*.
    - Symbol table là 1 mảng struct
    - Mỗi entry trong bảng sẽ chứa tên, kích thước và vị trí của symbol.

Linker Symbol?





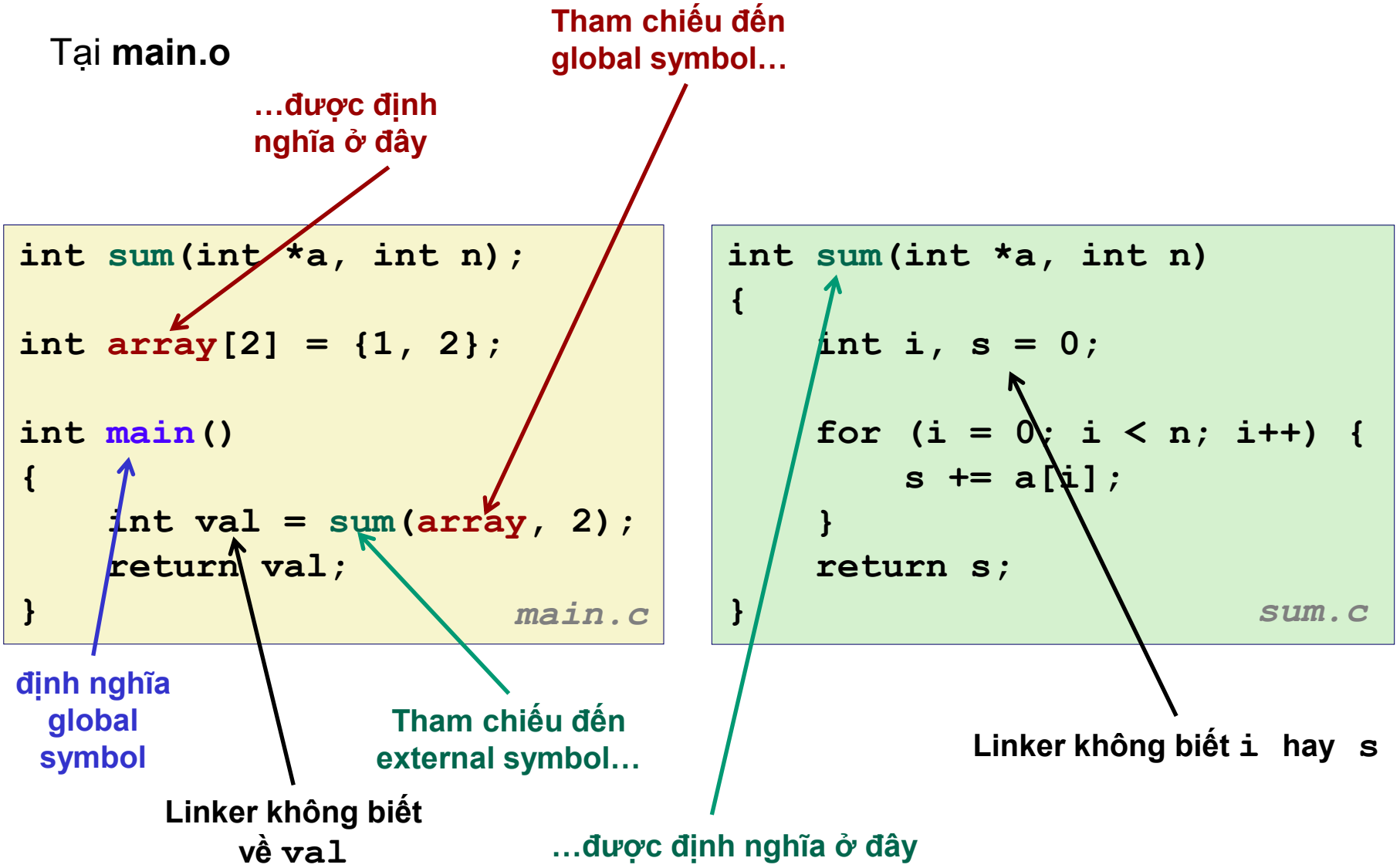
Các kiểu Linker Symbol

*Mỗi relocatable object mô-đun **m** (**m.o**) có 1 symbol table chứa thông tin các symbol được định nghĩa hoặc tham chiếu trong **m***

- **Global symbol**
  - Symbol được định nghĩa trong mô-đun *m* có thể được các mô-đun khác tham chiếu đến.
  - Ví dụ: Các hàm (function) không `static` và các biến toàn cục không `static`.
- **External symbol**
  - Global symbol được tham chiếu bởi mô-đun *m* nhưng được định nghĩa trong mô-đun khác.
- **Local symbol**
  - Symbol được định nghĩa và chỉ được phép tham chiếu trong mô-đun *m.*
  - Ví dụ: Các hàm C và biến toàn cục được định nghĩa với thuộc tính `static`.
  - **Local linker symbols *không phải* là các biến cục bộ**

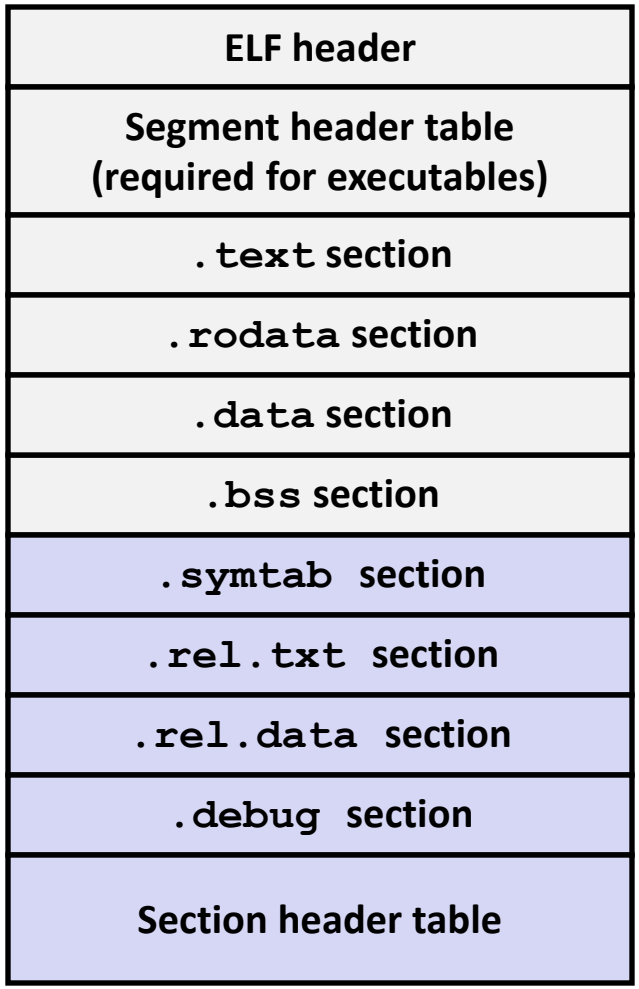
Ví dụ các kiểu Linker Symbol

Tại **main.o**



ELF Object File Format (tt)

- **`.symtab` section**
  - Symbol table
  - Hàm và các biến toàn cục, static
  - Tên section và vị trí
- **`.rel.text` section**
  - Thông tin tái cấu trúc cho `.text` section
  - Địa chỉ của một số instructions cần phải thay đổi trong file thực thi
  - Hướng dẫn cho việc thay đổi.
- **`.rel.data` section**
  - Thông tin tái cấu trúc cho `.data` section
  - Địa chỉ của một số dữ liệu con trỏ cần thay đổi trong file thực thi tổng hợp.
- **`.debug` section**
  - Thông tin cho debug (`gcc -g`)
- **Section header table**
  - Offsets và kích thước của mỗi section



Ví dụ các kiểu Linker Symbol

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

Xem symbol table của file .o

```
$ readelf -s main.o
```

```
Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000000    24 FUNC    GLOBAL DEFAULT    1 main
     9: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 array
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sum
```

Ví dụ các kiểu Linker Symbol

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

Xem symbol table của file .o

```
$ objdump -t main.o
```

```
main.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     O .data  0000000000000008 array
0000000000000000 g     F .text  000000000000001f main
0000000000000000         *UND*  0000000000000000 sum
```

Local symbol

- **Các biến cục bộ C không static vs. các biến cục bộ C có static**
  - Các biến cục bộ không static: lưu trong stack
  - Các biến cục bộ có static: lưu trong `.bss` hoặc `.data`

```
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

Compiler cấp phát không gian trong **`.data`** cho mỗi định nghĩa của **`x`**

Tạo các local symbol trong symbol table với các tên duy nhất, ví dụ, **`x.1`** và **`x.2`**.

Local symbol

- **Các biến cục bộ C không static vs. các biến cục bộ C có static**
  - Các biến cục bộ không static: lưu trong stack
  - Các biến cục bộ có static: lưu trong `.bss` hoặc `.data`

```c
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

```
Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS static.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 x.1832
     6: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    3 x.1835
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
    10: 0000000000000000    12 FUNC    GLOBAL DEFAULT    1 f
    11: 000000000000000c    12 FUNC    GLOBAL DEFAULT    1 g
```

Linking symbol
Bài tập 1

(a) main.c

code/link/main.c

```
1   /* main.c */
2   void swap();
3
4   int buf[2] = {1, 2};
5
6   int main()
7   {
8       swap();
9       return 0;
10  }
```

code/link/main.c

(b) swap.c

code/link/swap.c

```
1   /* swap.c */
2   extern int buf[];
3
4   int *bufp0 = &buf[0];
5   int *bufp1;
6
7   void swap()
8   {
9       int temp;
10
11      bufp1 = &buf[1];
12      temp = *bufp0;
13      *bufp0 = *bufp1;
14      *bufp1 = temp;
15  }
```

code/link/swap.c

Giả sử có 2 mô-đun tương ứng là **main.o** và **swap.o**, cho biết về sự có mặt và các thông tin của các biến/hàm sau trong symbol table của **swap.o**?

Symbol	swap.o .symtab entry?	Symbol type	Module where defined	Section
buf	**Yes**	**external**	**main.o**	**.data**
bufp0	**Yes**	**global**	**swap.o**	**.data**
bufp1	**Yes**	**global**	**swap.o**	**.bss**
swap	**Yes**	**global**	**swap.o**	**.text**
temp	**No**	———	———	———

Linker sẽ làm gì?

- **Tác vụ 1:** Phân giải symbol (Symbol Resolve)
- **Tác vụ 2:** Tái định vị (Relocation)

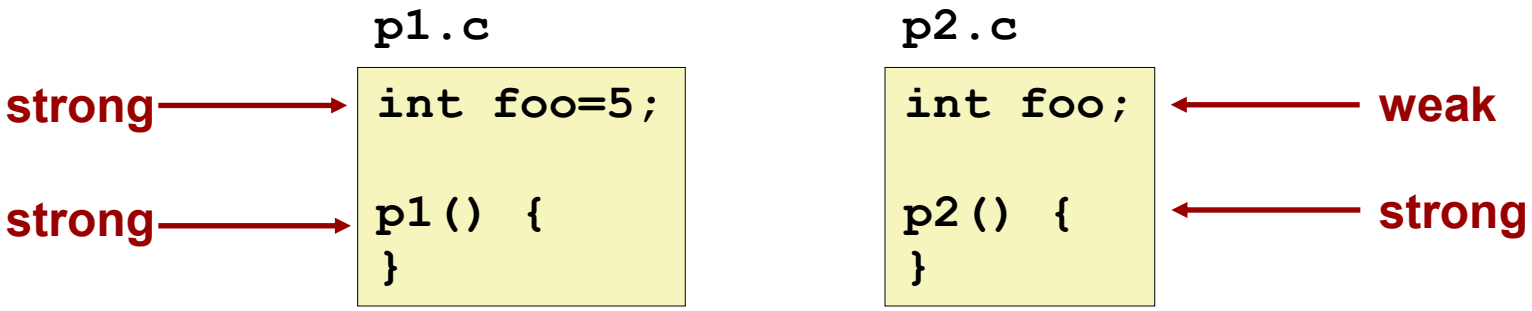
Tác vụ 1: Phân giải symbol
Phân giải symbol là làm gì?

- Phân giải symbol: Ánh xạ mỗi tham chiếu symbol đến chính xác 1 định nghĩa của symbol
  - Các symbol nằm trong symbol table của các file .o
  - **Nếu một symbol không được định nghĩa** trong mô-đun hiện tại thì vẫn có một entry trong symbol table, nhưng dành cho linker
    - Linker tìm trong tất cả các mô-đun khác → nếu không thấy thì lỗi!!
    - Ví dụ:

```
Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 array
     9: 0000000000000000    31 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sum
```

Tác vụ 1: Phân giải symbol
Nếu có nhiều định nghĩa symbol trùng tên?

- **Giải quyết: Các symbol được đánh giá là *strong* hoặc *weak***
  - ***Strong***: các hàm, các biến toàn cục đã có khởi tạo giá trị
  - ***Weak***: các biến toàn cục chưa khởi tạo giá trị



Tác vụ 1: Phân giải symbol
Các luật khi phân giải symbol

- **Luật 1: Không cho phép có nhiều strong symbol trùng tên**
  - Mỗi strong symbol chỉ được định nghĩa 1 lần
  - Nếu không: Linker error
- **Luật 2: Nếu có 1 strong symbol và nhiều weak symbol, chọn tham chiếu đến strong symbol**
  - Tham chiếu đến weak symbol sẽ được hiểu thành tham chiếu đến strong symbol
- **Luật 3: Nếu có nhiều weak symbols, chọn tham chiếu tuỳ ý đến 1 symbol**
  - Có thể thay đổi với `gcc -fno-common`

Tác vụ 1: Phân giải symbol

Ví dụ: Tham chiếu đến symbol nào?

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: 2 strong symbols (**p1**)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

Tham chiếu đến **x** sẽ cùng tham chiếu đến giá trị int chưa khởi tạo

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Gán giá trị cho **x** trong **p2** có thể ghi đè **y**! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Gán giá trị cho **x** trong **p2** sẽ ghi đè **y**! Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

Tham chiếu đến **x** sẽ luôn tham chiếu đến giá trị **x** đã được khởi tạo.

Phân giải symbol
Bài tập 2a

- **Giả sử REF(x.i) → DEF(x.k) là tham chiếu đến x** trong mô-đun **i** đến định nghĩa của **x** trong mô-đun **k.** Điền kết quả của các tham chiếu bên dưới, nếu có lỗi thì xác định:

  ERROR: link time error (Rule 1)

  UNKNOWN: lựa chọn tuỳ ý (Rule 3)

```
/* Module 1 */
int main()
{
}
```

```
/* Module 2 */
int main;
int p2()
{
}
```

(a) REF(main.1) --> DEF(**main.1**)

(b) REF(main.2) --> DEF(**main.1**)

Phân giải symbol
Bài tập 2b

- **Giả sử REF(x.i) → DEF(x.k) là tham chiếu đến x** trong mô-đun **i** đến định nghĩa của **x** trong mô-đun **k.** Điền kết quả của các tham chiếu bên dưới, nếu có lỗi thì xác định:

  ERROR: link time error (Rule 1)

  UNKNOWN: lựa chọn tuỳ ý (Rule 3)

```
/* Module 1 */
void main()
{
}
```

```
/* Module 2 */
int main=1;
int p2()
{
}
```

(a) REF(main.1) --> DEF(ERROR)

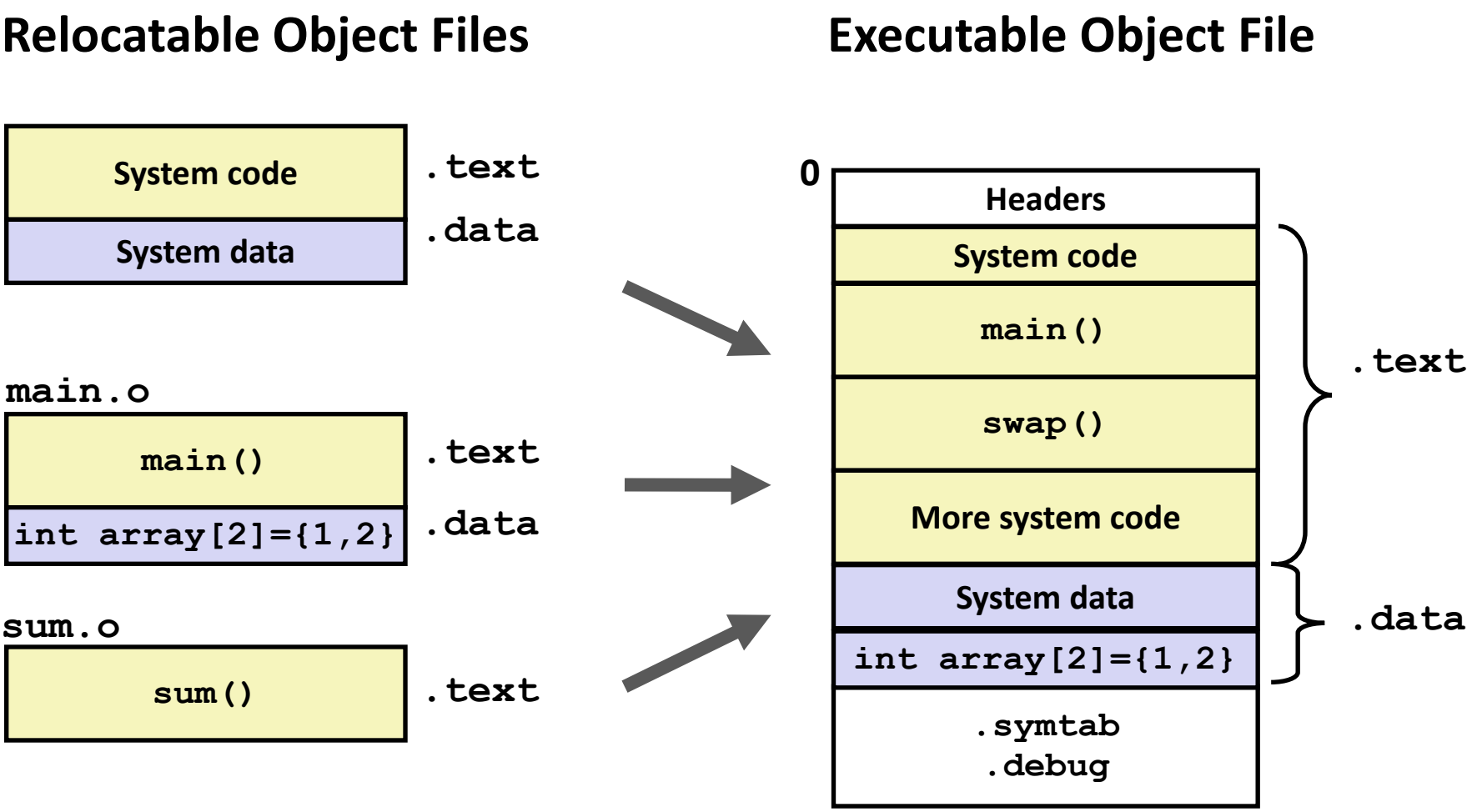
(b) REF(main.2) --> DEF(ERROR)

Biến toàn cục

- **Tránh sử dụng nếu có thể**

- **Nếu không, nên:**
  - Sử dụng **`static`** nếu có thể
  - Khởi tạo giá trị khi định nghĩa biến toàn cục
  - Sử dụng **`extern`** nếu tham chiếu đến một biến toàn cục ở file khác.

Tác vụ 2: Tái định vị (Relocation)



Tác vụ 2: Tái định vị (Relocation)

Các vị trí cần tái định vị

```
int array[2] = {1, 2};
int main()
{
    int val = sum(array, 2);
    return val;
}
```
main.c

Cần tái định vị

Thông tin về các vị trí trong section `.rela.text`

```
Relocation section '.rela.text' at offset 0x1e8 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000a  00090000000a R_X86_64_32       0000000000000000 array + 0
00000000000f  000a00000002 R_X86_64_PC32     0000000000000000 sum - 4
```

Source: `objdump -r -d main.o`

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi        # %edi = &array
                        a: R_X86_64_32 array           # Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4       # Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```
main.o

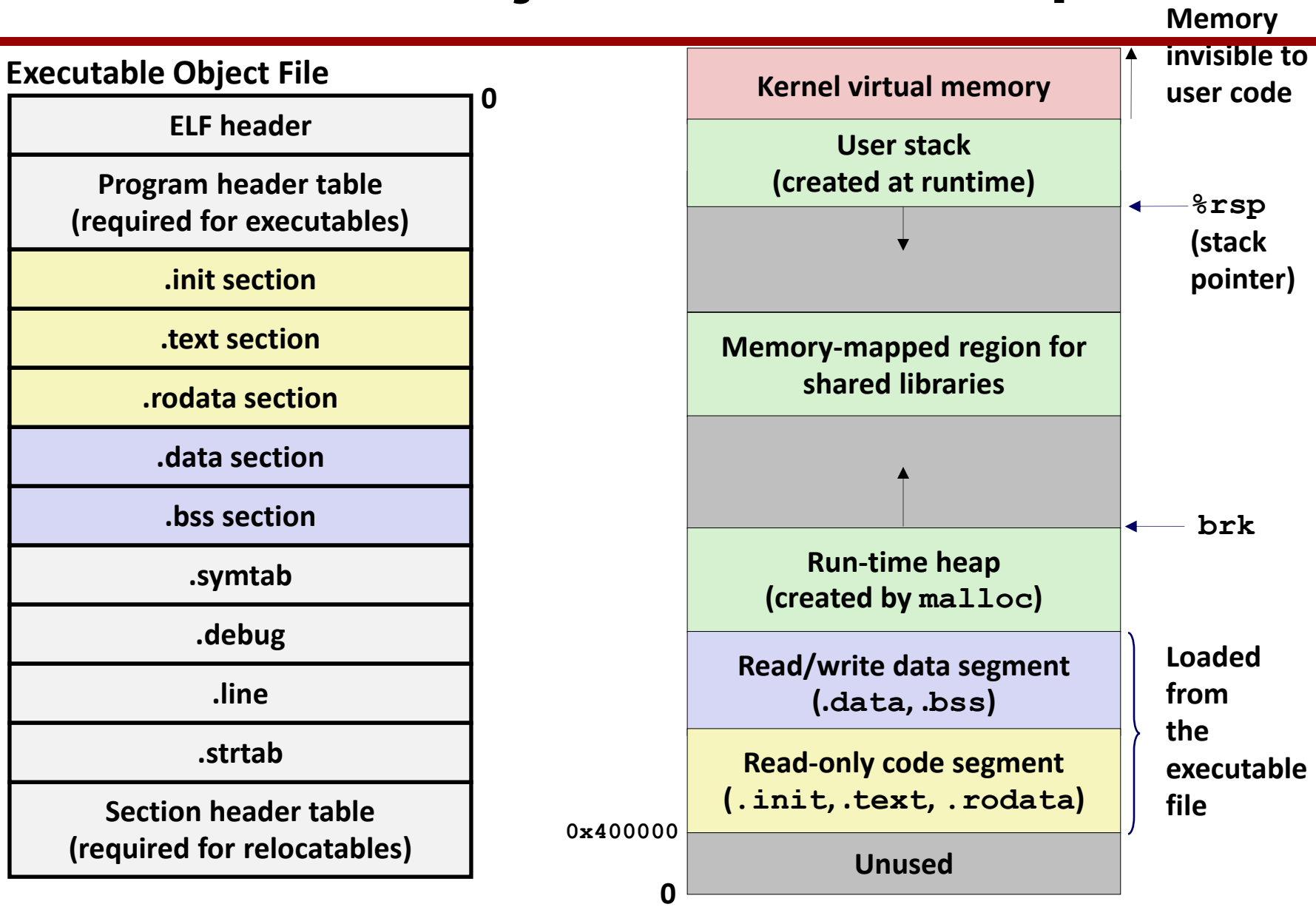
.text section sau khi tái định vị

```
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
```

Xác định địa chỉ hàm **sum()** (PC-relative): **0x4004e8** = **0x4004e3** + **0x5**

Source: objdump -dx prog

Executable Object Files trên bộ nhớ



Thêm: Đóng gói các hàm thông dụng

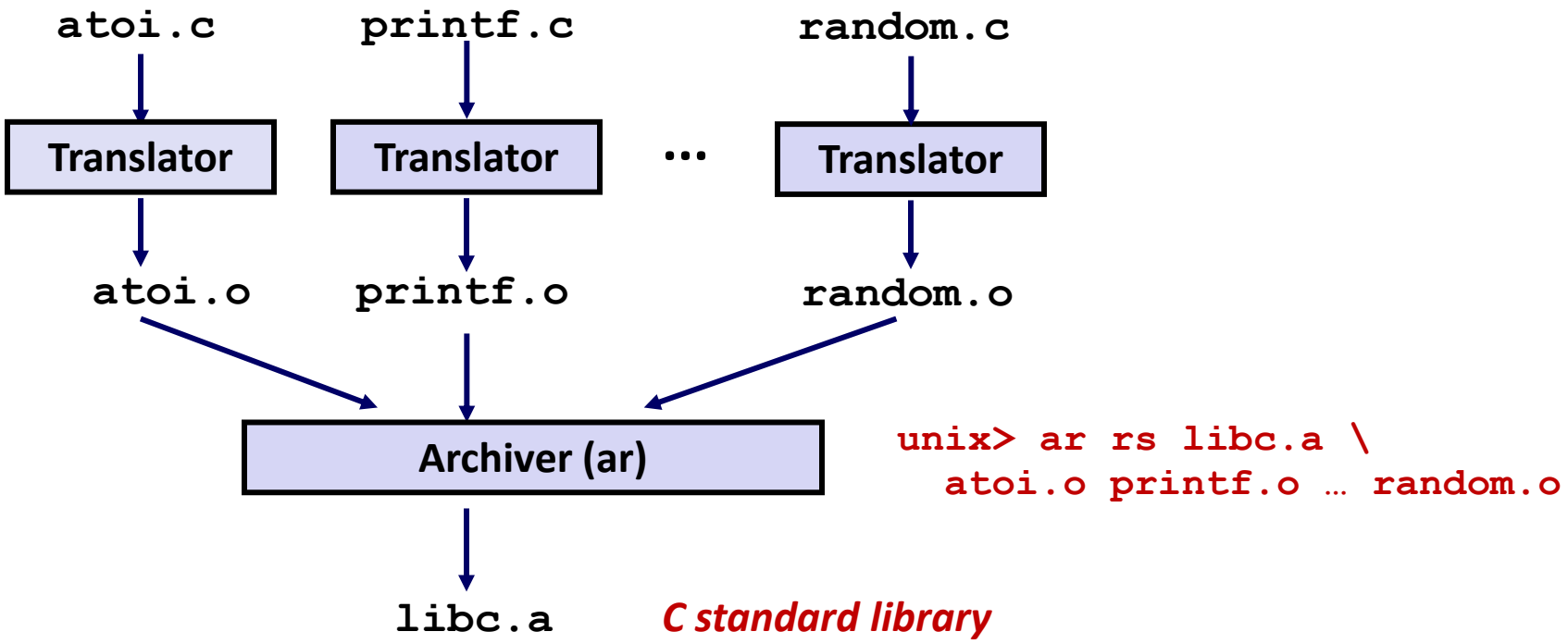
- **Làm thế nào đóng gói các hàm hay được lập trình viên sử dụng?**
  - Toán học, I/O, quản lý bộ nhớ, string, v.v..

- **Với linker:**
  - **Lựa chọn 1:** Đặt tất cả các hàm vào 1 file source
    - Lập trình viên link object file lớn vào chương trình của họ
    - Không hiệu quả về không gian và thời gian
  - **Lựa chọn 2:** Mỗi hàm được đặt trong 1 file source riêng
    - Lập trình viên chỉ link các file cần thiết vào chương trình của họ
    - Hiệu quả hơn, nhưng là gánh nặng cho lập trình viên

Giải pháp cũ: Static Libraries

- **Thư viện tĩnh - Static libraries (các file .a)**
  - Ghép các related relocatable object files (.o) thành 1 file duy nhất với chỉ số index (gọi là *archive*).
  - Là Linker nâng cao, cố gắng phân giải các tham chiếu bằng cách tìm các symbol trong 1 hoặc nhiều archive.
  - Nếu 1 archive file có thể phân giải tham chiếu, link file đó với file thực thi.

Cách tạo Static Libraries



- Archiver cho phép cập nhật
- Biên dịch lại hàm cần thay đổi và thay thế file .o tương ứng trong archive.

Linking với Static Libraries

libvector.a

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

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

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

main2.c

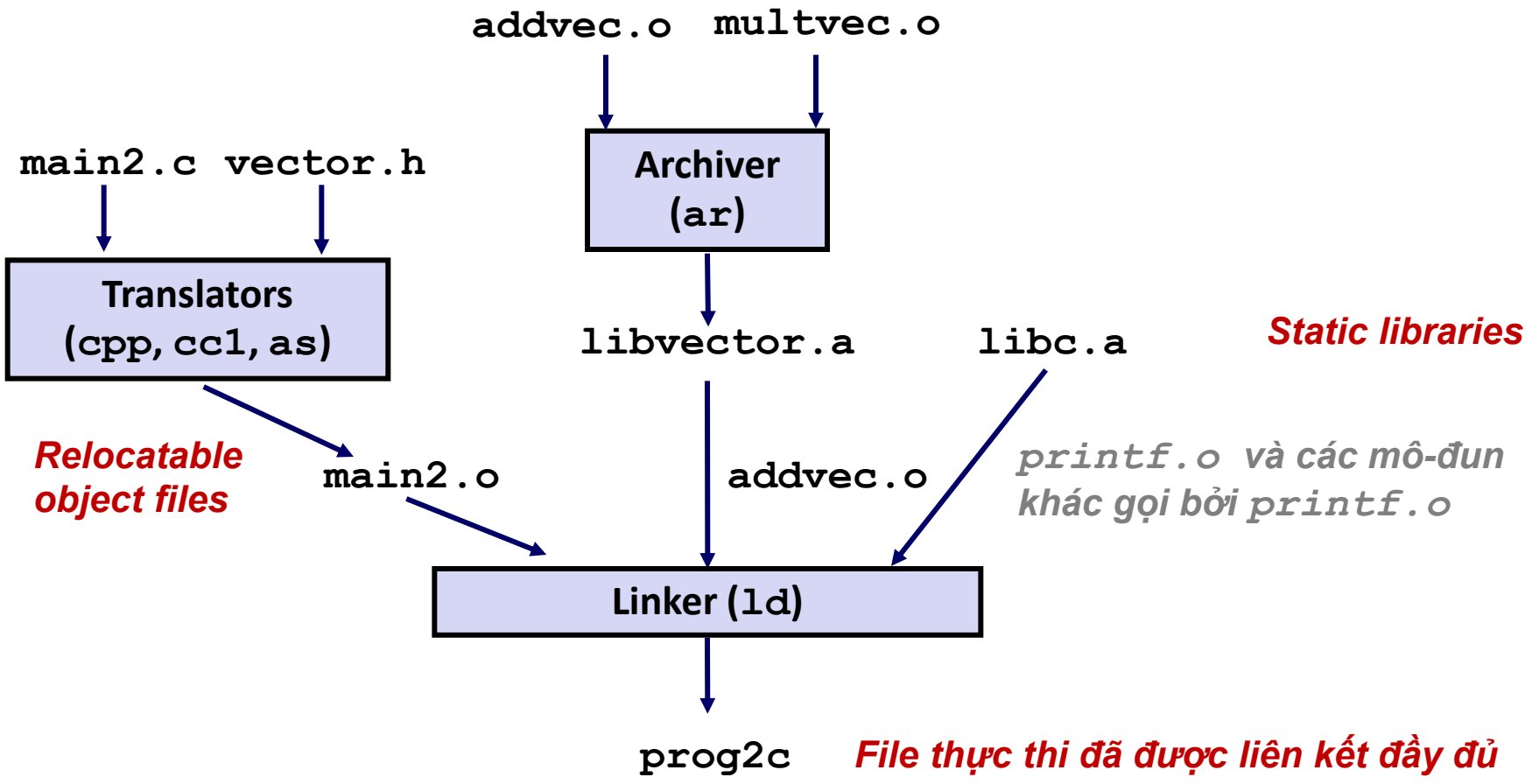
```c
int addcnt = 0;

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

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

addvec.c

```c
int multcnt = 0;

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

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

multvec.c

Linking với Static Libraries



"c" for "compile-time"

Sử dụng Static Libraries

- **Giải thuật của linker để phân giải các tham chiếu ngoài:**
  - Quét các file `.o` và `.a` theo thứ tự trong command.
  - Trong quá trình quét, lưu lại danh sách các tham chiếu chưa phân giải.
  - Với mỗi file `.o` hay `.a` mới, ví dụ *obj,* cố gắng phân giải các tham chiếu chưa phân giải trong danh sách với các symbol định nghĩa trong *obj*.
  - Nếu quét xong mà vẫn còn tham chiếu trong danh sách chưa phân giải được thì báo lỗi.

- **Vấn đề:**
  - Thứ tự khai báo trong command rất quan trọng!
  - Đặt các thư viện ở phía cuối command.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Giải pháp mới: Shared Libraries

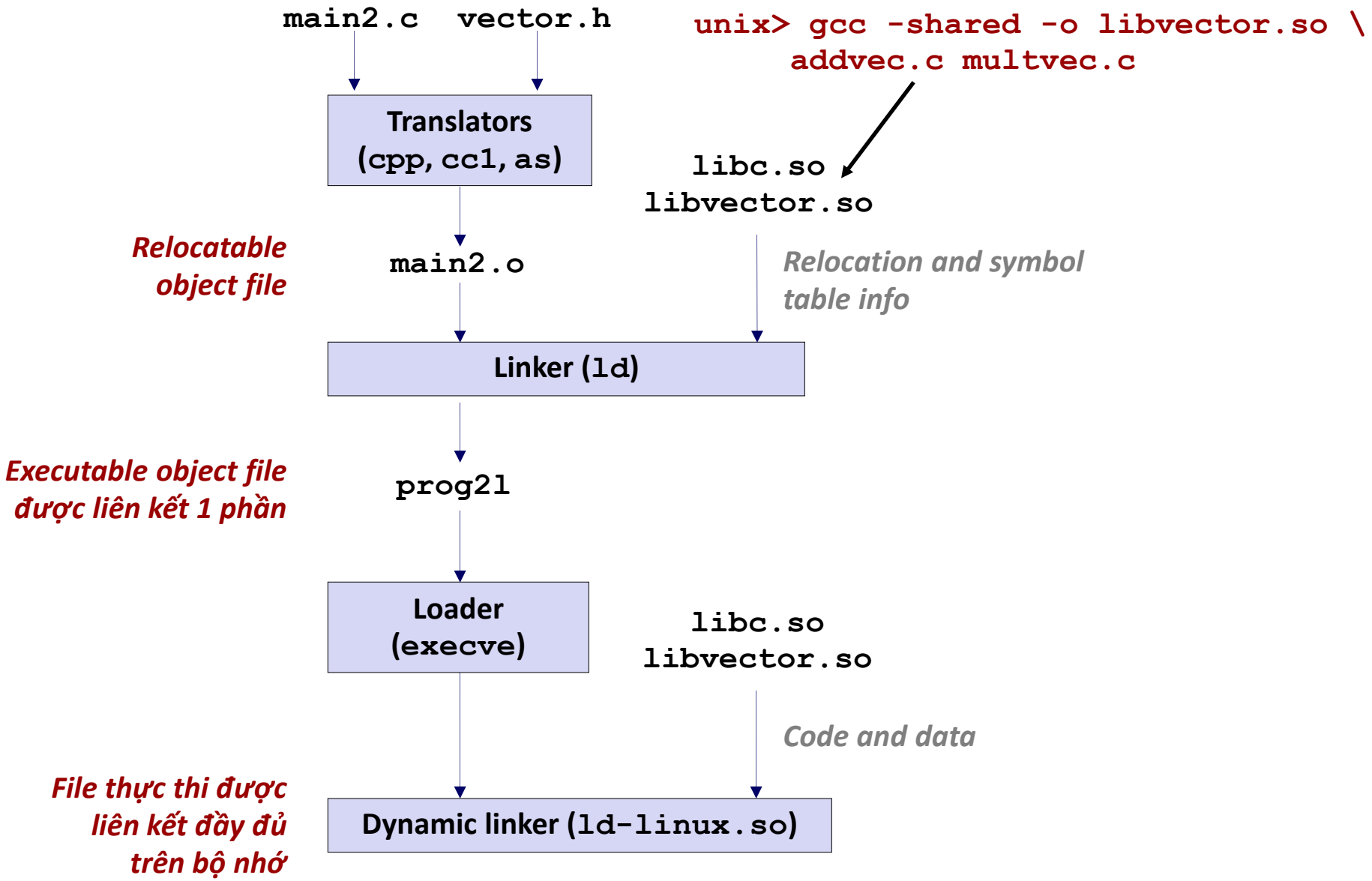
- **Static libraries có những điểm yếu:**
  - Trùng lặp trong các file thực thi (ví dụ: mọi function đều cần libc)
  - Nếu có chỉnh sửa nhỏ trong libraries thì mỗi chương trình cần thực hiện link lại.

- **Giải pháp mới: Shared Libraries**
  - Các object files chứa code and data được load và link vào một ứng dụng *một cách linh động,* tại thời điểm *load-time* hoặc *run-time*
  - Còn được gọi là dynamic link libraries, DLLs, .so files

Shared Libraries (tt)

- **Dynamic linking có thể xảy ra khi file thực thi được loaded và thực thi (load-time linking).**
  - Thường thấy ở Linux, tự động thực hiện bởi dynamic linker (**`ld-linux.so`**).
  - Standard C library (**`libc.so`**) thường được dynamically linking.
- **Dynamic linking cũng có thể xảy ra khi file đã chạy (run-time linking).**
  - Trong Linux, có thể thực hiện với hàm **`dlopen()`**.
- **Shared library có thể được chia sẻ giữa nhiều tiến trình.**

Dynamic Linking khi Load-time



Dynamic Linking khi Run-time

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

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

int main()
{
   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 khi Run-time

```
    . . .

  /* 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

```c
/* main.c */
void fib(int n);
int main (int argc, char** argv){
    int n = 0;
    ...
    fib(n);
}
```

```c
/* fib.c */
#define N 16
static unsigned int ring[3][N];
static void print_bignat(unsigned int* a){
    int i;
    ...
}
void fib (int n) {
    int i, carry;
    ...
}
```

1) Xem xét file main.o, đếm số lượng:

- Local symbol?
- Global symbol?
- External symbol?

2) Xem xét file fib.o, đếm số lượng:

- Local symbol?
- Global symbol?
- External symbol?

```c
/* main.c */
1. void fib(int n);
2. int main (int argc, char** argv){
3.     int n = 0;
4.     ...
5.     fib(n);
6. }
```

```c
/* fib.c */
1.  #define N 16
2.  static unsigned int ring[3][N];
3.  static void print_bignat(unsigned int* a){
4.      int i;
5.      ...
6.  }
7.  void fib (int n) {
8.      int i, carry;
9.      ...
10. }
```

- Ghi '-' ở cả 2 cột nếu tên không có trong symbol table của mô-đun tương ứng.
- Ghi N/A ở cột ***Strong hay weak*** nếu loại symbol là local.

Symbol table của main.o

Tên symbol	Loại symbol	Strong hay weak
main		
fib		
n		

Symbol table của fib.o

Tên symbol	Loại symbol	Strong hay weak
ring		
print_bignat		
fib		
canary		

Nội dung

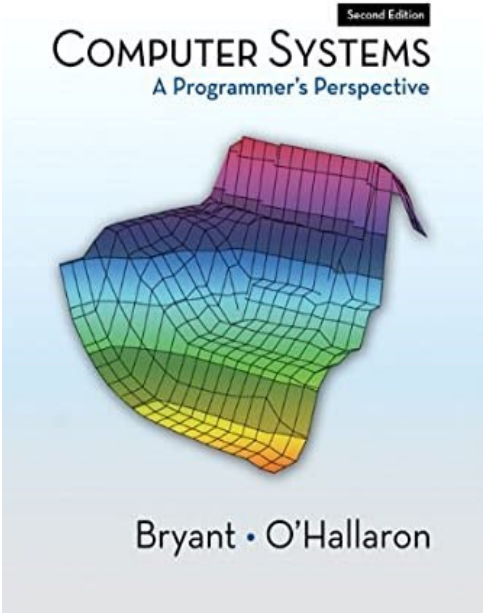
- **Các chủ đề chính:**
  1) Biểu diễn các kiểu dữ liệu và các phép tính toán bit
  2) Ngôn ngữ assembly
  3) Điều khiển luồng trong C với assembly
  4) Các thủ tục/hàm (procedure) trong C ở mức assembly
  5) Biểu diễn mảng, cấu trúc dữ liệu trong C
  6) Một số topic ATTT: reverse engineering, bufferoverflow
  7) Linking trong biên dịch file thực thi
  8) Phân cấp bộ nhớ, cache (tự tìm hiểu)
- **Lab liên quan**
  - Lab 1: Nội dung **1**
  - Lab 2: Nội dung 1, **2, 3**
  - Lab 3: Nội dung 1, **2, 3, 4, 5, 6**
  - Lab 4: Nội dung 1, **2,** 3**,** **4**, 5, **6**
  - Lab 5: Nội dung 1, **2,** 3**,** **4**, 5, **6**
  - Lab 6: Nội dung **1, 2, 3, 4, 5, 6**

Giáo trình

- **Giáo trình chính**

Computer Systems: A Programmer's Perspective

  - Second Edition (CS:APP2e), Pearson, 2010
  - Randal E. Bryant, David R. O'Hallaron
  - http://csapp.cs.cmu.edu



- **Tài liệu khác**
  - *The C Programming Language*, Second Edition, Prentice Hall, 1988
    - Brian Kernighan and Dennis Ritchie
  - *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, 1st Edition, 2008
    - Chris Eagle
  - *Reversing: Secrets of Reverse Engineering*, 1st Edition, 2011
    - Eldad Eilam


KEEP
CALM
AND
ENJOY YOUR
SEMESTER :)