Biên dịch và liên kết chương trình

Các bước biên dịch chương trình

Giả sử chương trình của chúng ta gồm có 2 modules sau

(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  }
```

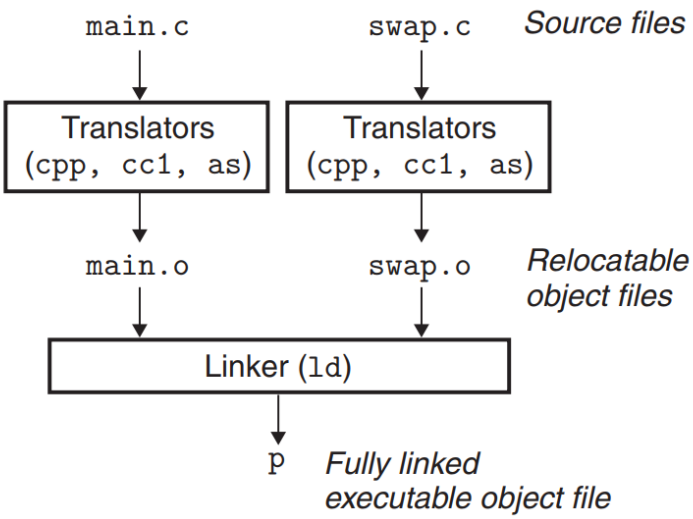
code/link/swap.c

Figure 7.1 **Example program 1:** The example program consists of two source files, `main.c` and `swap.c`. The `main` function initializes a two-element array of ints, and then calls the `swap` function to swap the pair.

Sử dụng bộ dịch gcc

gcc –o output main.c swap.c

Các bước thực hiện bởi bộ dịch gcc



Cụ thể gcc sẽ gọi các chương trình sau để thực hiện việc biên dịch. Ví dụ với main.c, làm tương tự với swap.c

1) Tiền xử lý (pre-processing) loại bỏ chú thích, chuyển đổi các macro trong chương trình C
   cpp main.c /tmp/main.i
2) Biên dịch chương trình C sang mã assembly
   cc1 /tmp/main.i -o /tmp/main.s
3) Dùng bộ dịch assembly chuyển sang mã máy dưới dạng file object cho từng module
   as /tmp/main.s –o /tmp/main.o
4) Dùng bộ linker để liên kết các file object lại thành chương trình mã máy
   ld /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/crt1.o -lc main.o swap.o -o output

Bốn bước trên có thể thực hiện qua gcc như sau:

1) Tiền xử lý: gcc –E
2) Biên dịch sang assembly: gcc –S
3) Tạo object file: gcc –c
4) Liên kết object file: gcc

Các loại tệp object

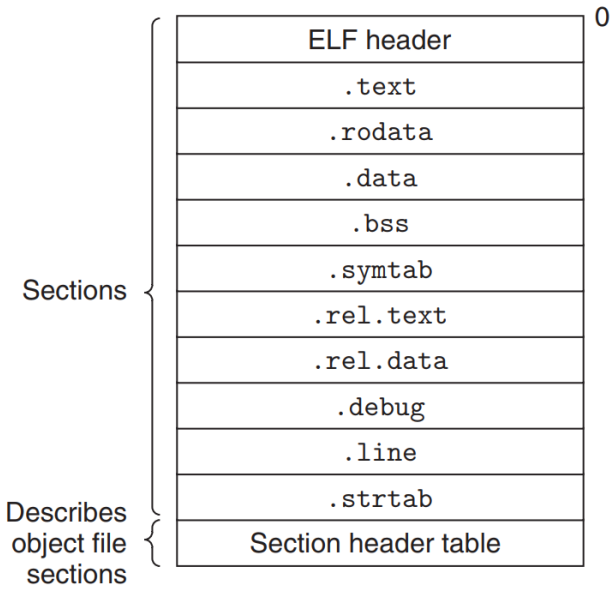
Relocatable object files: Chứa mã nhị phân chương trình, các khối dữ liệu chờ để liên kết với các tệp object khác để tạo thành tệp mã máy.

Executable object files: Chứa mã nhị phân chương trình và các khối dữ liệu có thể copy trực tiếp vào bộ nhớ và thực hiện khi chạy chương trình.

Shared object files: Tương tự như Relocatable object files nhưng việc liên kết chỉ được thực hiện vào thời điểm chạy khi tệp object được tải vào trong bộ nhớ.

Cấu trúc tệp Relocatable object

Định dạng ELF (Executalbe and Linkable Format) được sử dụng trong Linux và nhiều phiên bản Unix.



Mục ELF header chứa word size và thứ tự byte của hệ thống, loại tệp object, thông tin về các vùng cấu trúc tệp (kích thước ELF header, loại tệp oject, vị trí section header, kích thước và số trường trong section header).

Bảng Section header chứa vị trí và kích thước của các vùng trong tệp ELF.

Các vùng:

- .text: chứa đoạn mã nhị phân của chương trình
- .rodata: chứa phần dữ liệu chỉ đọc như xâu định dạng printf, bảng nhảy trong lệnh switch
- .data: chứa các biến tổng thể đã được khởi tạo giá trị.
- .bss (Block Starting Symbol / “Better Save Space”): chứa các biến tổng thể nhưng chưa có giá trị khởi tạo, do vậy không cần không gian lưu trữ như với vùng .data.
- .symtab: bảng chứa tên các biến tổng thể, các hàm được sử dụng tới trong chương trình
- .rel.text: danh sách những chỗ trong .text phải được thay đổi khi bộ linker liên kết các objects lại thành tệp đích. Những địa chỉ cần thay đổi là các lời gọi hàm bên ngoài hoặc tham chiếu tới biến tổng thể, do bộ linker sẽ sắp xếp lại các địa chỉ khi nghép với các tệp object khác.
- .debug: bảng thông tin gỡ rối chương trình, chứa thông tin về các biến tổng thể, cục bộ, gắn với mã nguồn (dùng với option –g)
- .line: bảng ánh xạ stt dòng mã nguồn sang dòng mã assembly (chỉ có nếu dịch với option –g)
- .strtab: bảng chứa xâu ký tự cho các tên biến, hàm được sử dụng trong vùng .symtab và .debug.

Cách bộ linker so khớp các ký hiệu

Nhiệm vụ của linker là gắn kết mỗi tham chiếu (tới biến, hàm) trong mã chương trình với một ký hiệu (symbol) duy nhất được định nghĩa trong bảng ký hiệu của tệp relocatable object.

Đối với các biến cục bộ, hoặc biến/hàm static (riêng cho mỗi module), việc liên kết là đơn giản bởi vì tên của ký hiệu đó là duy nhất trong phạm vi của module.

Đối với các biến tổng thể, các hàm có thể được khai báo và sử dụng tới ở nhiều object modules, do vậy việc so khớp và liên kết trở nên phức tạp hơn. Nếu không hiểu rõ cách thức so khớp này, có thể gây ra các lỗi không dễ phát hiện được.

Tại thời điểm biên dịch các ký hiệu ở mức khai báo global được xuất ra dạng assembly theo 2 loại strong hoặc weak. Bộ biên dịch assembly ngầm thể hiện 2 loại ký hiệu strong và weak này trong định dạng tệp object ELF (vùng .data và vùng .bss). Nếu 1 ký hiệu được định nghĩa trong nhiều object modules thì quy tắc để so khớp và liên kết là như sau

1. Sự tồn tại của 2 ký hiệu strong hoặc nhiều hơn là không được phép. Báo lỗi.
2. Nếu có 1 ký hiệu strong và ký hiệu đó trong các modules khác được coi là weak thì chọn bản strong
3. Nếu 1 ký hiệu xuất hiện đều dưới dạng weak trong nhiều modules thì chọn 1 bản bất kỳ

Ví dụ 1:

```c
/* foo1.c */
int main()
{
    return 0;
}
```

```c
/* bar1.c */
int main()
{
    return 0;
}
```

Khi dịch bằng lệnh *gcc foo1.c bar1.c* sẽ báo lỗi.

Ví dụ 2:

```c
/* foo3.c */
#include <stdio.h>
void f(void);

int x = 15213;

int main()
{
    f();
    printf("x = %d\n", x);
    return 0;
}
```

```c
/* bar3.c */
int x;

void f()
{
    x = 15212;
}
```

Bộ linker sẽ chọn ký hiệu *x* thuộc foo3.c do là ký hiệu mạnh. Khi chạy hàm f sẽ thay đổi giá trị *x* tổng thể, điều mà người viết foo3.c có thể không lường trước.

Ví dụ 3:

```c
/* foo5.c */
#include <stdio.h>
void f(void);

int x = 15213;
int y = 15212;

int main()
{
    f();
    printf("x = 0x%x y = 0x%x \n",
           x, y);
    return 0;
}
```

```c
/* bar5.c */
double x;

void f()
{
    x = -0.0;
}
```

Khi dịch *gcc foo5.c bar5.c* sẽ nhận được thông báo lỗi cảnh báo khá khó hiểu
/usr/bin/ld: Warning: alignment 4 of symbol `x' in /tmp/ccSUHHVG.o is smaller than 8 in /tmp/ccdivll7.o

Tại thời điểm liên kết, bộ linker sẽ chọn ký hiệu mạnh x từ foo5.c thay vì ký hiệu yếu x từ bar5.c. Tuy nhiên trong bar5.c, x có kiểu double dài 8 bytes, do vậy khi thực hiện hàm f sẽ đè vào 4 bytes của x và 4 bytes của y trong foo5.c, dẫn tới sai lệch kết quả.

Nếu thay bằng khai báo *float x;* trong bar5.c thì sẽ không có lỗi warning nhưng chương trình vẫn chứa một lỗi rất khó phát hiện.

Liên kết với thư viện tĩnh

Một số ngôn ngữ như Pascal cung cấp không nhiều các hàm chuẩn sẵn có, do vậy chương trình dịch sẽ tự tìm lấy đoạn mã của các hàm chuẩn này và copy vào mã đích của chương trình. Cách làm này không khả thi đối với C do số lượng các hàm là lớn hơn nhiều.

Việc lưu trữ mỗi hàm như một tệp object và đưa vào dòng lệnh thực sự là bất tiện, ví dụ

gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...

Do vậy giải pháp là gộp các tệp đối tượng có liên quan lại vào cùng một thư viện. Chỉ cần bổ sung tên thư viện vào trong dòng lệnh biên dịch. Bộ linker sẽ tìm những object module nào chứa hàm liên quan và copy vào mã đích của chương trình.

Ví dụ: tạo thư viện libvector.a từ hai object modules

(a) `addvec.o`

code/link/addvec.c

```
1  void addvec(int *x, int *y,
2              int *z, int n)
3  {
4      int i;
5
6      for (i = 0; i < n; i++)
7          z[i] = x[i] + y[i];
8  }
```

code/link/addvec.c

(b) `multvec.o`

code/link/multvec.c

```
1  void multvec(int *x, int *y,
2               int *z, int n)
3  {
4      int i;
5
6      for (i = 0; i < n; i++)
7          z[i] = x[i] * y[i];
8  }
```

code/link/multvec.c

Figure 7.5 **Member object files in `libvector.a`.**

gcc -c addvec.c multvec.c
ar rcs libvector.a addvec.o multvec.o

Tham số *rcs* trong lệnh archive *ar* là Replace (thay bằng nội dung mới), Create (tạo tệp archive mới), Sort (tạo bảng chỉ mục indexed để tìm kiếm nhanh trong thư viện.

Khi biên dịch chương trình có sử dụng các hàm trong thư viện libvector.a

gcc –static main.c ./libvector.a

Lưu ý, thư viện chuẩn *libc.a* luôn được gcc ngầm đưa vào tham số dòng lệnh khi gọi tới bộ linker, do vậy chúng ta thường bỏ qua.

Vấn đề về thứ tự tham số dòng lệnh khi liên kết

Khi thực hiện liên kết với nhiều thư viện, ví dụ

gcc –static main.c lib1.a lib2.a ...

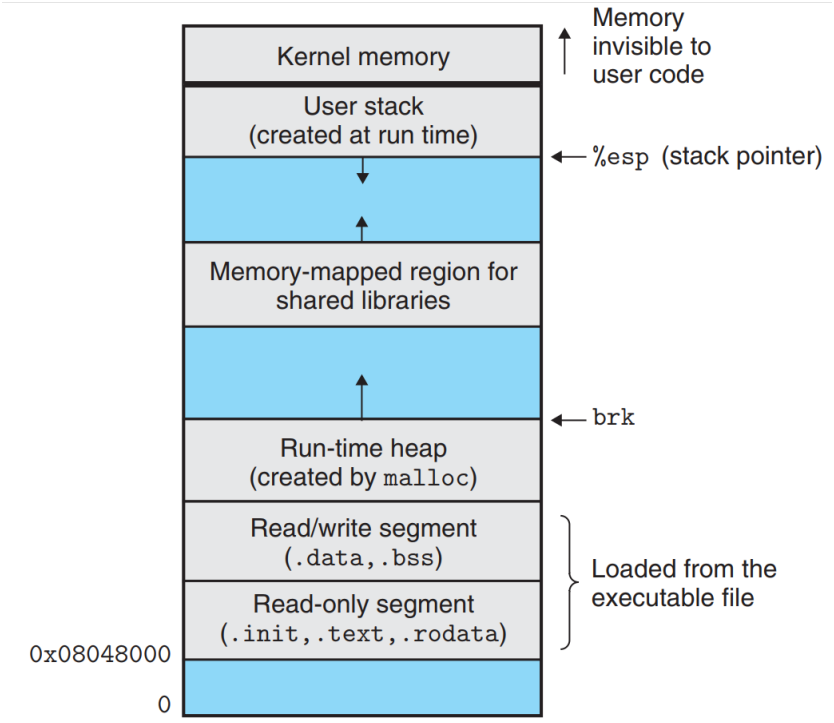
bộ linker sẽ đọc lần lượt các tham số dòng lệnh để so khớp và liên kết lại. Trong quá trình duyệt qua các tệp, gọi U tập các ký hiệu chưa so khớp được. Nếu đọc tới tệp tiếp theo và so khớp được thì ký hiệu đó sẽ được xóa khỏi U, object module tương ứng sẽ được copy vào mã đích chương trình. Thứ tự của các tệp tham số là quan trọng bởi chẳng hạn nếu một đoạn mã trong *lib2* tham chiếu tới hàm *f* trong *lib1* thì *x* sẽ được bổ sung vào U. Tuy nhiên *x* sẽ không bao giờ được so khớp bởi vì *lib1* đã bị quét qua trước đó nên đoạn mã hàm *f1* sẽ không được copy vào mã đích chương trình.

Trong một số trường hợp có sự phụ thuộc lẫn nhau giữa các thư viện thì việc lặp lại tham số là cần thiết. Chẳng hạn, nếu *lib1* phụ thuộc *lib2*, *lib2* phụ thuộc *lib3* và *lib3* phụ thuộc *lib1* thì cần đưa tham số lặp như sau đối với lib1

gcc –static main.c lib1.a lib2.a lib3.a lib1.a

Tải và chạy đoạn mã thực thi của chương trình

Khi chạy một chương trình từ dòng lệnh, hệ điều hành sẽ kích hoạt một chương trình loader để tải, copy các đoạn mã, dữ liệu của tệp object thực thi (executable object file) vào bộ nhớ và nhảy tới lệnh đầu tiên của chương trình để thực hiện. Trong unix/linux, có thể gọi loader từ một ứng dụng khác thông qua hàm *execve*. Dưới đây là bản đồ bộ nhớ của một chương trình đang thực thi.



Bắt đầu là phân đoạn bộ nhớ dành cho mã chương trình, dữ liệu tĩnh và tiếp theo là dữ liệu cấp phát động trong quá trình chạy (heap). Phía trên cùng là ngăn xếp, chứa giá trị của các biến cục bộ trước khi thực hiện các lời gọi hàm. Phía giữa là vùng nhớ được dự trữ cho thư viện động dùng chung (thảo luận sau) nếu dùng tới. Khi vùng nhớ ngăn xếp phát triển xuống và đụng độ với vùng nhớ phía dưới sẽ gây lỗi stack overflow.

Liên kết với thư viện động dùng chung

Đối với thư viện tĩnh, mã của các object module được copy thẳng vào mã chương trình đích. Nếu có hàng trăm chương trình chạy đồng thời sẽ có một lượng lớn bộ nhớ trùng lặp gây lãng phí. Ngoài ra, khi có bản cập nhật thư viện mới, cần thiết phải biên dịch lại tất cả các chương trình.

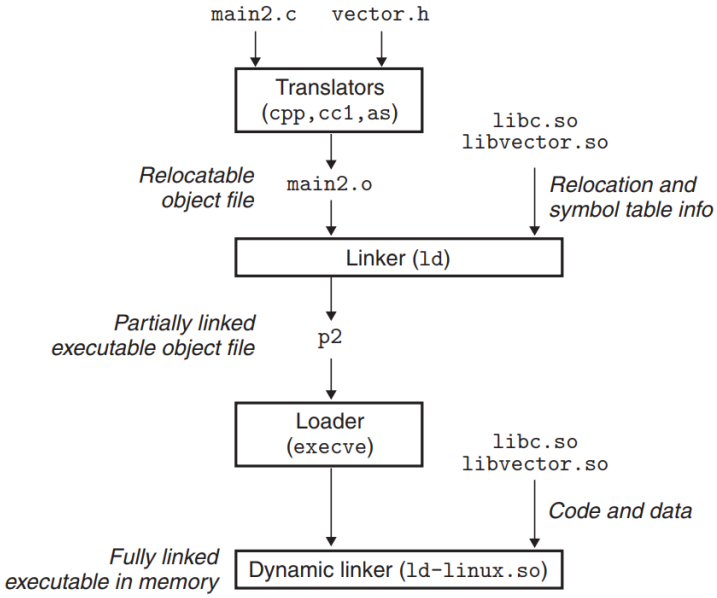
Thư viện động được tải vào bộ nhớ và cho phép việc liên kết với chương trình được thực hiện vào lúc chạy. Do vậy giải quyết được vấn đề trên.

Trong Windows, các tệp thư viện động có đuôi DLL, còn trong unix/linux tệp thư viện động có đuôi so. Có thể tạo tệp thư viện động so qua dòng lệnh, ví dụ

```
gcc  -c -fPIC addvec.c multvec.c
gcc -shared -o libvector.so addvec.o multvec.o
```

gcc main.c ./libvector.so

Tùy chọn *–fPIC* yêu cầu trình biên dịch tạo mã Position-Independent Code cho tệp object, tùy chọn –shared là để tạo tệp thư viện chia sẻ đuôi so, lựa chọn. Biên dịch với thư viện động như đối với một thư viện tĩnh, tuy nhiên việc liên kết thực sự chỉ diễn ra vào thời điểm chạy như trong hình dưới đây.



Cách thực hiện liên kết động trong ứng dụng khi chạy

Các hàm cần dùng *#include <dlfcn.h>*

Nếu tệp mã thực thi được dịch với tùy chọn –rdynamic thì các ký hiệu tổng thể cũng có thể dùng để liên kết với các ký hiệu trong thư viện.

*void *dlopen(const char *filename, int flag);*	*Mở tệp thư viện động* *filename: đường dẫn tới tệp thư viện* *flag* *RTLD_GLOBAL: tệp mở với cờ này cho phép liên kết biến tổng thể. Có thể OR với các cờ dưới đây* *RTLD_NOW / RTLD_LAZY: liên kết ngay khi tải, hay khi chạy*
*void *dlsym(void *handle, char *symbol);*	*Liên kết tới một ký hiệu trong thư viện* *Trả lại con trỏ tới ký hiệu tìm được, NULL nếu lỗi*
*int dlclose (void *handle);*	*Đóng tệp thư viện động* *Trả lại 0 nếu ok, -1 nếu lỗi*
*const char *dlerror(void);*	*Trả lại thông báo lỗi, NULL nếu ok*

Ví dụ: *gcc -rdynamic -o dll dll.c -ldl*

Lựa chọn cờ *-ldl* liên kết với thư viện động /usr/lib/libdl.so chứa các hàm mô tả trên đây.

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

    /* 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;
}
```

code/link/dll.c

Figure 7.16 **An application program that dynamically loads and links the shared library** `libvector.so`.