ICS Homework 12

May 11, 2022

1 Organization

1.1 Symbol Resolution

foobar.c:

```
1: #include <stdio.h>
2:
3: int x;
4: int y;
5: int z = 10;
6: static int a = 10;
7: static int b = 20;
8:
9: void bar();
10:
11: void foo() {
12:     static int b = 10;
13:     int c = 5;
14:
15:     printf("%d\n", x);
16:     printf("%d\n", b);
17:     printf("%d\n", c);
18:     bar();
19: }
20:
21: __attribute__((weak)) void bar() {
22:     static int d = 10;
23:     int c = 5;
24:
25:     printf("%d\n", x);
26:     printf("%d\n", b);
27:     printf("%d\n", c);
28:     printf("%d\n", d);
29: }
```

barbaz.c:

```
1: #include <stdio.h>
2:
3: int x = 5;
4: int y = 10;
5: extern int z;
6: static int a = 40;
7: static int b = 30;
8:
9: void bar();
10:
11: void baz() {
12:     static int b = 10;
13:     int c = 5;
```

```
14:
15:     printf("%d\n", x);
16:     printf("%d\n", b);
17:     printf("%d\n", c);
18:     printf("%d\n", y);
19:     bar();
20: }
21:
22: void bar() {
23:     static int d = 10;
24:     int c = 5;
25:
26:     printf("%d\n", x);
27:     printf("%d\n", b);
28:     printf("%d\n", c);
29:     printf("%d\n", z);
30: }
```

Please show the actually used variable in each reference:

Reference	Variable
foobar.c:15:x	
foobar.c:16:b	
foobar.c:17:c	
foobar.c:18:bar	
foobar.c:25:x	
foobar.c:26:b	
foobar.c:27:c	
foobar.c:28:d	
barbaz.c:15:x	
barbaz.c:16:b	
barbaz.c:17:c	
barbaz.c:18:y	
barbaz.c:19:bar	
barbaz.c:26:x	
barbaz.c:27:b	
barbaz.c:28:c	
barbaz.c:29:z	

1.2 Static Libraries

Let a and b denote object modules or static libraries in the current directory, and let a → b denote that a depends on b, in the sense that b defines a symbol that is referenced by a. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that will allow the static linker to resolve all symbol references.

A. p.o → libx.a
B. p.o → libx.a → liby.a
C. p.o → libx.a → liby.a and liby.a → libx.a → p.o
D: p.o → libx.a → liby.a → libz.a and liby.a → libx.a → libz.a

1.3 Dynamic Linking

ICSTA wrote two C programs as shown below: **subvec.c** and **dynamic_line.c**. We compile **subvec.c** as a shared library (gcc -shared -fpic -o libvector.so subvec.c):

```c
/* subvec.c */
int delcnt;
void subvec(int *x, int *y, int *z, int n) {
    for(int i = 0; i < n; i++) {
        z[i] = x[i] - y[i];
    }
}

/* dynamic_link.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) {
    /* we can call subvec() like */
    /* any other function */
    subvec(x, y, z, 2);
    printf("z=[%d,%d]", z[0], z[1]);

    return 0;
}
```

1.3.1

Please give two ways that we can link the shared libraries **libvector.so** (NOTE: You can modify the **dynamic_link.c**)

1.3.2

Why GOT is needed to relocate functions in **libvector.so**?

1.3.3

After the shared libraries **libvector.so** was loaded in memory, please fill in the address of **GOT entry of subvec** before and after first invocation. Suppose that the address of PLT[0] is 0x404360, the address of PLT entry of subvec is 0x404560, and the address of subvec() is 0x400128, the value of GOT[0] is 0x406670.

Before:

After:

2 System Software

2.1 Concurrent

Please fill in the blanks with initial values for the three semaphores and add **P()** and **V()** semaphore operations such that the process is guaranteed to terminate. (NOTE: You can fill in zero or multiple **P(x)** or **V(x)** operations)

```
1  /* Initialize x */
2  int x = 1;
3
4  /* Initialize semaphores */
5  sem_t a, b, c;
6  sem_init(&a, 0, _____);
7  sem_init(&b, 0, _____);
8  sem_init(&c, 0, _____);
9
10 void thread1()
11 {
12     while (x != 12) {
13         _________;
14         x = x * 2;
15         _________;
16     }
17     exit(0);
18 }
19
20 void thread2()
21 {
22     while (x != 12) {
23         _________;
24         x = x * 3;
25         _________;
26     }
27     exit(0);
28 }
```

2.2 Shared Varibles in multi-threading

```
#include "csapp .h"
#define N 4
void *print_thread(void *vargp)
{
    int myid = *((int)vargp);
    printf("in thread %d\n", myid);
    return NULL;
}

int main() {
    pthread t tid[N];
    int *ptr;
    for (int i = 0; i < N; i++) {
        ptr = malloc(sizeof(int));
        *ptr = i;
        // Creat a thread to run the "print_thread func with arg ptr
        // Your core here: _____________
        free(ptr);
    }

    for (int i = 0; i < N; i++)
        pthread join(tid[i], NULL);
}
```

1. Complete the previous code according to the comment.
2. Is there any race condition in the previous code? Why or why not?