ICS EXE 10

May 19, 2021

1 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):

```
1  /* subvec.c */
2  int delcnt;
3  void subvec(int *x, int *y, int *z, int n) {
4      for(int i = 0; i < n; i++) {
5          z[i] = x[i] - y[i];
6      }
7  }
8
9  /* dynamic_link.c */
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <dlfcn.h>
13 int x[2] = {1, 2};
14 int y[2] = {3, 4};
15 int z[2];
16 int main(void) {
17     /* we can call subvec() like */
18     /* any other function */
19     subvec(x, y, z, 2);
20     printf("z = [%d %d]", z[0], z[1]);
21
22     return 0;
23 }
```

1.1

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

We can link it when the application is loaded. gcc -o prog dynamic link.c ./libvector.so

We can link it when the application executes through modifying the main function

```
int main() {
    void handle;
    void (*subvec)(int*, int*, int*, int);
    handle = dlopen("./libvector.so", RTLD_LAZY);
    subvec = dlsym(handle, "subvec");
    subvec(x, y, z, 2);
    printf("z = [%d %d ]", z[0], z[1]);
    dlclose(handle);
}
```

1.2

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

Because with GOT, the code segment is position-independent(PIC). So a single copy of a shared module's code segment can be shared by an unlimited number of processes

1.3

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: 0x404566

After: 0x400128

2 Concurrent1

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 only fill in zero or multiple **P(x)** or **V(x)** operations)

```
/* Initialize x */
int x = 1;

/* Initialize semaphores */
sem_t a, b, c;
sem_init(&a, 0, 0);
sem_init(&b, 0, 1);
sem_init(&c, 0, 0); /* Not used */

void thread1()
{
```

```
12      while (x != 12) {
13          x = x * 2;
14          V(a);
15          P(b);
16      }
17      exit(0);
18  }
19
20  void thread2()
21  {
22      while (x != 12) {
23          P(a);
24          P(a);
25          x = x * 3;
26          V(b);
27      }
28      exit(0);
29  }
```

3 Concurrent2

The following code implements a simple stack.

```
1   typedef struct Node {
2       struct Node *next;
3       int value;
4   } Node;
5
6   void push(Node **top_ptr, Node *n) {
7       n->next = *top_ptr;
8       *top_ptr = n;
9   }
10
11  Node *pop(Node **top_ptr) {
12      if (*top_ptr == NULL)
13          return NULL;
14      Node *p = *top_ptr;
15      *top_ptr = (*top_ptr)->next;
16      return p;
17  }
```

3.1

is this implementation thread-safe?

No. Multiple threads may modify the *top_ptr simultaneously.

3.2

Use semaphore to protect the operations towards the stack.

Create a new struct – stack which contains a top pointer points to the top node of the stack and a sem which protects the stack operation. For all stack operation, P(stack->sem) before start and V(stack->sem) after finish.

4 Concurrent3

Initially, counting semaphore S is initialized with value 2. What is the **maximum possible value of x** after all the processes complete execution.

Process1	Process2	Process3	Process4
P(S)	P(S)	P(S)	P(S)
x = *addr	x = *addr	x = *addr	x = *addr
x = x + 1	**x = x + 1**	**x = x - 2**	**x = x - 2**
*addr = x	*addr = x	*addr = x	*addr = x
V(S)	V(S)	V(S)	V(S)

To obtain the maximum value of x, the processes must execute in such a way that only the impact of the processes 1 and 2 remains on the value of x. The impact of processes 3 and 4 gets lost on the value of x. One Possible solution is:

1. Block P1 after execute x = *addr.
2. Allow P3 and P4 finish their execution.
3. P1 can continue execute and *addr = x + 1.
4. P2 can execute and *addr now is the old value + 2.