Homework #2 M1522.000800 System Programming

Sample Solution

Question 1

Integer Arithmetic

Consider the following code that attempts to sum the elements of an array a, where the number of elements is given by parameter *length*:

```
/* WARNING: This is buggy code */
float sum_elements (float a[], unsigned length)
{
    int i;
    float result = 0;

    for (i = 0; i <= length-1; i++)
        result += a[i];
    return result;
}
```

When run with argument *length* equal to 0, this code should return 0.0. Instead it encounters a memory error. Explain why this happens. Show how this code can be corrected.

=> The variable type of *length* is unsigned, and *length-1* will be *0xFFFF_FFFF* which is the maximum value of unsigned integer. The operation between signed integer and unsigned one make these auto-cast to unsigned ones. This invokes that **for** loop condition to be true every time by when *i* would be *0xFFFF_FFFF*. That's why this buggy code accesses over-array area, and encounters a memory error like segmentation faults.

Question 2
SYMBOLS & SYMBOL TABLES

Consider the following version of the swap.c function that counts the number of times it has been called, and main.c.

```
/* main.c */
void swap();

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

int main()
{
    swap();
    return 0;
}
```

```
/* swap.c */
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

static void incr()
{
    static int count = 0;
    count++;
}

void swap()
{
    int temp;

    incr();
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

For each symbol that is defined and referenced in swap.o, indicate if it will have a symbol table entry in the .symtab section in module swap.o. If so, indicate the module that defines the symbol (swap.o or main.o), the symbol type (local, global, or extern) and the section (.text, .data, or .bss) it occupies in that module.

Symbol	Swap.o's .symtab entry?	Symbol type	Module where define	Section
buf	Yes	extern	main.o	.data
bufp0	Yes	global	swap.o	.data
bufp1	Yes	local	swap.o	.bss
swap	Yes	global	swap.o	.text
temp	No	-	-	-
incr	Yes	local	swap.o	.text
count	Yes	local	swap.o	.data

Question 3

SYMBOL RESOLUTION

Consider the following example, where x is defined as an *int* in one module and a *double* in another:

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

int x = 0x01234567;
int y = 0x89ABCDEF;

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

```
/* bar.c */
long long x;

void f()
{
    x = 0xDEADBEEFBABEFACE;
}
```

After compiling and linking with the following commands using GCC compiler driver, we can found that variable x, y is overwritten by f().

```
linux > gcc -o foobar foo.c bar.c
( Warning … )
linux > ./foobar
BABEFACE DEADBEEF
```

Explain why this happens. Show how this code can print “01234567 89ABCDEF” instead of that.

=> Both **foo.c** & **bar.c** have global symbol which name is same as **x**. Linker initialize this symbol as *int* **x** in **foo.c** because this declaration is more strong than **bar.c**. (There is an initialization of value in **foo.c**, but no initialization in **bar.c**.) However Linker has no type checking phase, and in compiler there is no compile error in each source. The size of type *long long* is 8 bytes, and *integer* has 4 bytes. Therefore **f()** overwrites **x** area and **y** area. There is many ways to print *01234567 89ABCDEF* instead of that. For example, in **bar.c** *long long* **x** could be changed with *static long long* **x**.

Question 4

LINKING WITH STATIC LIBRARIES

Consider the two different approaches to use C standard libraries. One approach [A1] is to have the compiler recognize calls to the standard functions and to generate the appropriate code directly. Another [A2] is to have the compiler link archive files where standard functions are defined.

```
[A1] unix > gcc main.c /usr/lib/libc.o
[A2] unix > gcc main.c /usr/lib/libc.a
```

Explain which approach is better in common case and why.

=> **A1** makes huge object file which contains even not-called functions in **main.c**. It needs to be recompiled every time when only one standard function is re-written. On the other hands, **A2** makes object file which contains only called functions in **main.c**. Therefore, **A2** is better in common cases.

Question 5
STATIC LIBARARIES & REFERENCE RESOLUTION

During the symbol resolution phase in linking with static libraries, the linker scans the relocatable object files and archives left to right in the same sequential order that they appear on the compiler driver's command line. (The driver automatically translates any .c files on the command line into .o files.) During this scan, the linker maintains a set *E* of relocatable object files that will be merged to form the executable, a set *U* of unresolved symbols (i.e., symbols referred to but not yet defined), and a set *D* of symbols that have been defined in previous input files. Initially, *E*, *U*, and *D* are empty.

- For each input file **f** on the command line, the linker determines if **f** is an object file or an archive. If **f** is an object file, the linker adds **f** to *E*, updates *U* and *D* to reflect the symbol definitions and references in **f**, and proceeds to the next input file.

- If **f** is an archive, the linker attempts to match the unresolved symbols in *U* against the symbols defined by the members of the archive. If some archive member, **m**, defines a symbol that resolves a reference in *U*, then **m** is added to *E*, and the linker updates *U* and *D* to reflect the symbol definitions and references in **m**. This process iterates over the member object files in the archive until a fixed point is reached where *U* and *D* no longer change. At this point, any member object files not contained in *E* are simply discarded and the linker proceeds to the next input file.

- If *U* is nonempty when the linker finishes scanning the input files on the command line, it prints an error and terminates. Otherwise it merges and relocates the object files in *E* to build the output executable file.

Assume that './libfoo.a' has a definition of function 'foo()' which called in './main.c'. If we compile and link with the following command, we encounter the linking error.

```
unix > gcc -static ./libfoo.a main.c
/tmp/cc9XH6Rp.o: In function 'main':
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to 'foo'
```

Explain why this happens and how it could be fixed. (Consider linker symbol resolution scan rule.)

=> **main.c** calls **foo()** which is defined in **./libfoo.a**. GCC compiler driver scans input files in sequential order ; **./libfoo.a** → **main.c**. When linker analyze **./libfoo.a**, there might be no unresolved symbols. After that, linker analyze **main.c** and there is undefined reference **foo()**, but there is no remaining input files to resolve this reference error. Therefore, it is important to make right order when you use linker in command line. You should consider which one is dependent to others, and make it right order. *Gcc -static main.c ./libfoo.a* would be good in upper case.