CS29206 Systems Programming Laboratory, Spring 2022–2023

Class Test 1

15–March–2023 07:00pm–08:00pm Maximum marks: 60

Roll no: _____ **Name:** _____

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

1. Suppose that you compile the program **`badstart.c`** using gcc. What warning message you would get? How do you repair the code to avoid the warning message? Write your answer in the box given below. **(3 + 3)**

```
$ cat badstart.c
int main ()
{
   printf("Hello world!\n");
   return 0;
}
$ gcc -Wall badstart.c
```

The compiler complains about the implicit definition of the function **`printf`**.

The problem can be repaired by inserting

```
#include <stdio.h>
```

at the beginning of the code.

2. Consider the following C program **`argwork.c`**. The **`stdlib`** function **`atoi()`** converts a numeric string to an **`int`**.

```
#include <stdio.h>
#include <stdlib.h>
#define NUM(a,b) ((a) > (b) ? a : b)
int main(int argc, char* argv[])
{
   int x = atoi(argv[1]), y = atoi(argv[argc-1]);
   #ifdef PRINT1
      printf("%d\n", NUM(x,y));
   #endif
   #ifdef PRINT2
      printf("%d\n", x + y - NUM(x,y));
   #endif
}
```

(a) What would happen if you run this program without any command-line argument? **(2)**

The program will encounter a segmentation fault (**`argv[]`** is a NULL-terminated array, and **`argv[1]`** is accessed).

(b) How should you compile the code so that:

(i) the program prints the larger of the first and the last arguments, **(2)**

```
gcc -Wall -DPRINT1 argwork.c
```

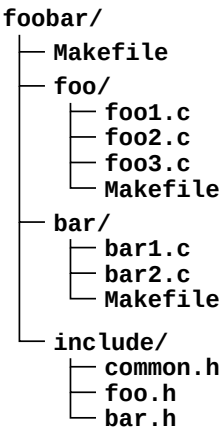
(ii) the program prints both the first and the last arguments, **(2)**

```
gcc -Wall -DPRINT1 -DPRINT2 argwork.c
```

(iii) the program prints nothing. **(2)**

```
gcc -Wall argwork.c
```

3. You have a project with root directory **foobar** (see the adjacent figure) for preparing a shared (that is, dynamic) library **libfoobar.so**. The source codes are in two subdirectories: **foo** contains the foomatic functions, and **bar** contains the bargodic functions. All the required header files reside in the subdirectory **include**. Both the foomatic and the bargodic functions require the header file **common.h**. The foomatic functions additionally require the header file **foo.h**, and the bargodic functions the header file **bar.h**. A makefile in the **foo** directory is meant for generating the object files from the foomatic source files, and a makefile in the **bar** directory is meant for generating the object files from the bargodic source files. No libraries are to be prepared in the **foo** and the **bar** directories. A top-level makefile in the **foobar** directory is meant for recursively invoking the makefiles of the **foo** and the **bar** subdirectories and finally combining all the object files into the dynamic library file (in the **foobar** directory). The source files use the **#include <...>** format. Write the three makefiles in the boxes below. There is no need to write **install** and **clean** targets.



(4 × 3)

Makefile in **foobar/**

```
FOOOBJS = foo/foo1.o foo/foo2.o foo/foo3.o
BAROBJS = bar/bar1.o bar/bar2.o
all:
        cd foo; make
        cd bar; make
        gcc -shared -o libfoobar.o $(FOOOBJS) $(BAROBJS)
```

Makefile in **foobar/foo/**

```
CFLAGS = -Wall -fPIC -I../include
OBJS = foo1.o foo2.o foo3.o
all: $(OBJS)
$(OBJS): ../include/common.h ../include/foo.h
```

Makefile in **foobar/bar/**

```
CFLAGS = -Wall -fPIC -I../include
OBJS = bar1.o bar2.o
all: $(OBJS)
$(OBJS): ../include/common.h ../include/bar.h
```

4. Suppose that a C source file contains only the following functions.

```
void f ( int n ) { printf("%d\n", n); }
void g ( int n ) { while (n > 0) { --n; f(n); } }
int main () { for (int n=1; n<=8; ++n) { f(n); g(n); } }
```

Running the code with gprof gives the following output (contiguous) in the call graph. Fill in the blanks to complete the gprof output for the given fragment. Show your calculations in the box given below. **(5 + 5)**

```
-------------------------------
                0.00    0.00       ____8____ / ____8____          main [8]
[2]     0.0     0.00    0.00       ____8____                   g [2]
                0.00    0.00       ___36____ / ___44____          f [1]
-------------------------------
```

The transcript corresponds to the call records of the function `g()`.

The `main()` function calls `f()` and `g()` eight times each. No other function calls `g()`, so the total number of calls of `g()` is 8 (second line in the transcript), and all these are from `main()` (first line).

For a given n, $g(n)$ calls $f(n-1), f(n-2), \ldots, f(0)$ (a total of n calls). Since $g(n)$ is called for $n = 1, 2, 3, \ldots, 8$, the total number of times `f()` is called by `g()` is $1+2+3+\cdots+8 = 36$. So the total number of calls of `f()` is $8+36 = 44$, out of which 36 calls are by `g()` (the remaining 8 are by `main()`).

5. A C program creates a linked list $71 \to 35 \to 47 \to 40 \to 22 \to 46 \to 58 \to 11$ headed by a node pointer `L`. Assume that there is **no** dummy node at the beginning of the list, that is, only the eight nodes in the list are allocated memory in the program. There is a function `listsearch(L,x)` that makes a linear search for `x` in the linked list headed by `L`, and returns a pointer to a node storing `x` if such a node exists, or NULL otherwise. After creating the above list, the program executes the line

```
L = listsearch(L,40);
```

and then exits. Assume that there are no global variables, and `L` is the only pointer declared and used in `main()`. Find the types of memory leaks as detected by valgrind on this program in each of the following cases. No explanations are needed. Only fill out the following table.

Case 1: `L` heads a singly linked list with the `next` pointer of the last node set to `NULL`. Assume that we have `sizeof(node) = 16` in this case. **(6)**

Case 2: `L` heads a doubly linked list with the `next` pointer of the last node and the `prev` pointer of the first node set to `NULL`. Assume that we have `sizeof(node) = 24` in this case. **(6)**

Loss Type	Number of blocks	Number of bytes	Which blocks?
Case 1: L is a singly linked list			
Still in use	5	80	40, 22, 46, 58, 12
Definitely lost	1	16	71
Indirectly lost	2	32	35, 47
Case 2: L is a doubly linked list			
Still in use	8	192	71, 35, 47, 40, 22, 46, 58, 12
Definitely lost	0	0	None
Indirectly lost	0	0	None

6. You compile the adjacent program (called **myprog.c**) using gcc with the **-g** option, and run the resulting executable under gdb. The line numbers in the program are as shown. You set a breakpoint at Line 19. You then make two runs of the program. In the first run, enter 4 as **x**, and in the second run, enter 5 as **x**. If a run hits the breakpoint, you enter the gdb command **bt**. After this, you allow the program to finish. Show the transcripts of your gdb sessions for the two runs in the boxes provided below. Also explain the outputs produced by gdb. **(6 + 6)**

Assume that in both the cases, **main()** gets loaded at the memory location **0x20010**, **f()** gets loaded at **0x123A4** and **g()** gets loaded at **0x234B5**. The transcripts you write need not match the real ones available from actual experiments, but you should mention the essential points, and furnish proper explanations.

```
 1:  #include <stdio.h>
 2:  void f (int);
 3:  void g (int);
 4:  int main ()
 5:  {
 6:     int x;
 7:     printf("Enter x: ");
 8:     scanf("%d",&x);
 9:     f(x);
10:  }
11:  void f(int x)
12:  {
13:     if (x>0) g(x-1);
14:     else return;
15:  }
16:  void g(int y)
17:  {
18:     if (y>0) f(y-1);
19:     else return;
20:  }
```

```
gdb> run
Enter x: 4

[Inferior 1 (process 12345) exited normally]
(gdb)
```

Explanation:

Here, the sequence of calls goes as follows.

$$main() \to f(4) \to g(3) \to f(2) \to g(1) \to f(0)$$

The breakpoint is reached if and only if **g()** is called with the parameter **y = 0**. The situation never arises here.

```
gdb> run
Enter x: 5

Breakpoint 1, g (y=0) at myprog.c:19
19          else return;
(gdb) bt
#0  g (y=0) at myprog.c:19
#1  0x123a4 in f (x=1) at myprog.c:13
#2  0x234b5 in g (y=2) at myprog.c:18
#3  0x123a4 in f (x=3) at myprog.c:13
#4  0x234b5 in g (y=4) at myprog.c:18
#5  0x123a4 in f (x=5) at myprog.c:13
#6  0x20010 in main () at myprog.c:9
(gdb) c
Continuing.
[Inferior 1 (process 12358) exited normally]
(gdb)
```

Explanation:

Here, the sequence of calls goes as follows.

$$main() \to f(5) \to g(4) \to f(3) \to g(2) \to f(1) \to g(0)$$

In this case, $g(0)$ is called, so the breakpoint is reached, and **bt** (backtrace) reports the call stack as given above.