CS29206: Systems Programming Lab
Spring 2025
Test 1, February 27, 2025

Time: 1 Hour **Full marks: 60**

ROLL NO.: ____________________ **NAME:** ________________________________

Q1	Q2	Q3	Q4	Q5	Q6	TOTAL

1. Suppose that you are writing a program to implement a dictionary that will insert, delete, and search strings from a dictionary. You put all functions (including the main function) in a file called *dictionary.c*, with an associated *dictionary.h* file, in your current directory. Your functions need to manipulate strings, for which it uses functions from a static library called *libstr.a*, which is present in the directory */home/spl/lib/strings* directory. The corresponding header file *str.h* is present in */home/spl/include/strings* directory. Your program also needs to use efficient data structures, for which it uses functions from a static library called *libds.a*, which is present in the directory */home/spl/lib/datastructs* directory. The corresponding header file *ds.h* is present in */home/spl/include/datastructs* directory. All header files in *dictionary.c* uses the #include <…> format. Write a single gcc command that will create an executable file called *dictionary*. The compilation should use *–Wall* option and should stop on any warning or error. No explanation is needed. **(6)**

```
gcc –Wall –Werror  –I.  –I/home/spl/include/strings  -I/home/spl/include/datastructs
              –L/home/spl/lib/strings  -L/home/spl/lib/datastructs dictionary.c  –o dictionary –lds  -lstr
```

[1 mark for basic gcc, 2 marks for –I, 1 mark –L, 1 mark for –Werror, 1 mark for –l]

2. Consider the following program fragment in a file called *myfile.c*:

```
1        if (n > 0) printf(“Error: n is %d\n”, n);
2        m = func1(n);
3        if (m==0) printf(“Error: m should not be 0\n);
```

In the *normal* mode, no error checking is done and the program should not print anything irrespective of the values of *n* and *m*. In the *debug* mode, all error checkings are on and all printfs should be printed if applicable depending on the values of *n* and *m*. The user decides during compilation time whether (s)he uses the *normal* or the *debug* mode. Modify the above code (without deleting any printf and without using any extra variables) so that the user can select the error checking (printing) mode (*normal* or *debug*) using appropriate compilation options. Show both the modified code and the gcc command for turning on error checking in the code. **(6)**

```
#ifdef DEBUG
        <line 1 from above here>
#endif

<line 2 from above here>

#ifdef DEBUG
        <line 3 from above here>
#endif
```

For debug mode, compile as “gcc –DDEBUG myfile.c”
For normal mode, compile as “gcc myfile.c”

[2 marks for first ifdef-endif, 2 marks for last ifdef-endif, 2 marks for –D flag]

If you have put all 3 lines inside a single ifdef-endif, you have got 4.5 out of 6 if otherwise correct. This is because then in debug mode *func1* is also not called which may be an important function for the program.

Some variations possible, given marks as appropriate.

3. Consider a software package written in C. The package contains one makefile and following source files inside it (all are in one directory).

angles.c   *build.c*   *coord.c*   *electrostat.c*   *ftdock.c*   *grid.c*   *rspin.c*   *structures.h*

All the .c files include only *structures.h* as one header file that contains all the structure definitions along with the prototype declaration of all the functions used in the C source files. Each of the three C source files *ftdock.c*, *build.c*, and *rspin.c* uses functions defined in itself, as well as functions defined in each of the source files *angles.c*, *coord.c*, *electrostat.c*, and *grid.c*. However, none of the files *ftdock.c*, *build.c*, and *rspin.c* uses function in each other (for example, *ftdock.c* does not use anything from *build.c* and *rspin.c*, and the same for *build.c* and *rspin.c*). All C files use functions from the standard math library.

A student wrote the following makefile for building three executable files, *ftdock*, *build*, and *rspin*, from the corresponding three C files. The makefile also gives read and execute permission to the executable files after building them. Fill in the blanks below that will create the executables and assign the permissions. While filling in the blanks, use the appropriate variables from the declarations (as much as possible). **(15)**

```
SHELL              = /bin/sh
CC                 = gcc
CC_FLAGS           = -Wall
CC_LINKERS         = -lm
SECURITY           =  chmod 555

LIBRARY_OBJECTS =  angles.o coord.o electrostat.o grid.o

PROGRAMS = ftdock build rspin

all:        $(PROGRAMS)

ftdock:    ftdock.o $(LIBRARY_OBJECTS) structures.h
           $(CC) $(CC_FLAGS) -o $@ ftdock.o $(LIBRARY_OBJECTS) $(CC_LINKERS)
            $(SECURITY) $@

build:     build.o $(LIBRARY_OBJECTS) structures.h
           $(CC) $(CC_FLAGS) -o $@ build.o $(LIBRARY_OBJECTS) $(CC_LINKERS)
           $(SECURITY) $@

rspin:     rspin.o $(LIBRARY_OBJECTS) structures.h
           $(CC) $(CC_FLAGS) -o $@ rspin.o $(LIBRARY_OBJECTS) $(CC_LINKERS)
           $(SECURITY) $@

clean:
           rm -f *.o core $(PROGRAMS)

# dependencies (mark any other file dependencies below this, if any). Add more blank lines below
the two shown if needed.
ftdock.o:       structures.h
build.o:        structures.h
rspin.o:        structures.h

angles.o:       structures.h
coord.o:        structures.h
electrostat.o:  structures.h
grid.o:         structures.h
```

[Admissible variations are accepted. 3 marks for dependencies, 1 mark for each other lines.]

4. A student wrote the following C program (in a file named *main.c*) as part of an assignment. The program creates an image of size 50×50 (in a 2-d array, each element is one pixel of the image) and fills it (pixel values) by random numbers that varies from 0 to 255. Next, the program calculates the histogram of the image, which is the frequency of the occurrence of each pixel value in the image.

```
#include <stdio.h>
#include <stdlib.h>
#define ISIZE   50
#define MAXVAL  255

int incr(int i) {  return i+1; }

int test(int i, int n) {  return (i<n)? 1:0;  }

int getVal() {
        int val = rand();
        return test(MAXVAL,val)? val%MAXVAL : val;
}

void init(int *hgram) {
        for(int i=0; test(i,MAXVAL+1); i=incr(i))  hgram[i] = 0;
}

void print(int *hgram) {
        for(int i=0; i<MAXVAL+1; i=incr(i)) printf("%d ", hgram[i]);
        printf("\n");
}

int main() {
        int i, j, m = ISIZE, n = ISIZE, img[ISIZE][ISIZE], hgram[MAXVAL+1];
        init(hgram);
        for(i=0; test(i,n); i=incr(i)) {
                for(j=0; test(j,m); j=incr(j)) {
                        img[i][j] = getVal();
                        hgram[img[i][j]] = incr(hgram[img[i][j]]);
                }
        }
        print(hgram);
        return 0;
}
```

The student does not believe the fact that function calls are costly and hence makes a large number of function calls as shown. To test the number of calls, a TA compiles the program and generates the call graph using the following commands:

```
$gcc -Wall -pg main.c
$./a.out
$gprof -b -z ./a.out gmon.out
```

The call graph as displayed on the terminal (relevant parts) is given below on the left. Fill in the blanks marked (A) to (Q) in the output of gprof in the box at the right to demonstrate to the student how many function calls are there. **(15)**

```
index % time   self  children    called      name
               0.00    0.00   _(A)_      init [4]
               0.00    0.00   _(B)_      print [5]
               0.00    0.00   _(C)_      main [11]
[1]     0.0    0.00    0.00   _(D)_        incr [1]
-----------------------------------------------
               0.00    0.00   _(E)_      init [4]
               0.00    0.00   _(F)_      getVal [3]
               0.00    0.00   _(G)_      main [11]
[2]     0.0    0.00    0.00   _(H)_        test [2]
-----------------------------------------------
               0.00    0.00     _(I)_    main [11]
[3]     0.0    0.00    0.00   _(J)_        getVal [3]
               0.00    0.00    _(K)_     test [2]
-----------------------------------------------
               0.00    0.00     1/1          main [11]
[4]     0.0    0.00    0.00      1         init [4]
               0.00    0.00     _(L)_     test [2]
               0.00    0.00    _(M)_      incr [1]
-----------------------------------------------
               0.00    0.00     1/1          main [11]
[5]     0.0    0.00    0.00      1         print [5]
               0.00    0.00     _(N)_     incr [1]
-----------------------------------------------
                         . . . . .
-----------------------------------------------
                                        <spontaneous>
[11]     0.0    0.00    0.00              main [11]
               0.00    0.00     _(O)_     incr [1]
               0.00    0.00     _(P)_     test [2]
               0.00    0.00     _(Q)_     getVal [3]
               0.00    0.00     1/1          init [4]
               0.00    0.00     1/1          print [5]
-----------------------------------------------
```

(A)	**256/5562**
(B)	**256/5562**
(C)	**5050/5562**
(D)	**5562**
(E)	**257/5358**
(F)	**2500/5358**
(G)	**2601/5358**
(H)	**5358**
(I)	**2500/2500**
(J)	**2500**
(K)	**2500/5358**
(L)	**257/5358**
(M)	**256/5562**
(N)	**256/5562**
(O)	**5050/5562**
(P)	**2601/5358**
(Q)	**2500/2500**

Lines for indices [6] to [10] and lines displayed after index [11] are not relevant for answering the question and hence are not shown.

[1 mark for each, best 15. Marks given as long as values are close enough within a reasonable margin]

5. Consider the following program fragment that you are debugging, with line numbers shown against each statement.

```
10      for (int i=0; i<5; i++)  m += func1(i);
11      if (m >= 1000) m = scaleDown(m);
12      ….
```

func1(i) returns a positive integer if *i* is even, 0 otherwise. You want to see the value of *m* after return from the call to *func1()* in the for loop only if the value changes. However, you do not want to see the value of *m* after the call to *scaleDown* (if called, and even if it changes). Show the gdb commands you will use (in sequence) to do this when you run the executable through gdb. You should try to minimize the number of commands. No explanation is needed. **(6)**

```
b 10
b 11
r
<when breakpoint hits at line 10>
watch m
<when breakpoint hits at line 11>
d 3
```

continuation commands when breakpoints are hit (for ex. when m changes) are not shown and not graded.

[2 marks for watch, 1 for deleting watch, 3 for everything else]

You have got 4 out of 6 if you didn't use watch but computed explicitly when will m change and will it be printed or not etc.

Some other variations possible and has been given due credit.

SPACE FOR ROUGH WORK

6. Consider the following program:

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

char **p, **q;
int readStr()
{
        int n, i;  char *S;
        scanf("%d", &n);
        p = (char **)malloc(n*sizeof(char *));
        for (i=0; i<n; i++) {
                S = (char *)malloc(100*sizeof(char));
                scanf("%s", S);
                p[i] = (char *)malloc((strlen(S)+1)*sizeof(char));
                strcpy(p[i], S);
        }
        q = p + (n-1);
        return(n);
}
void check(int n)
{
        char *temp;
        while (p != q){
                if (strcmp(*p, *q) == 0){
                        temp = *p; *p = *q; *q = temp;
                        q--;
                }
                else p++;
        }
}
int main()
{
        int k, i;
        k = readStr();
        check(k);
       for (i=0; i<k; i++) free (p[i]);
        free(p);
}
```

If the program is run with valgrind, describe the memory leaks that will be reported for each of the following inputs given from the keyboard. For each case, list the amount (in no. of bytes) and the type of memory leaks for each type of leak, with a short explanation of why is it happening.

(12)

(a) 5 aaa bbbbb ccccc ee ee

Definitely lost: 500 bytes (The S array of size 100 bytes is malloc'ed 5 times and never freed)

Possibly lost: 56 bytes

1. **p moves to second last "ee". So memory for first three strings (16 bytes including the '\0'space) are possibly lost as valgrind cannot figure out if all pointers to it are lost or not.**
2. **Since p changes in this case in the check() function, the free(p) in main fails (free() requires the pointer value to be the same as that returned by the malloc()). So 5 address pointer space (5x8 = 40 bytes) are also possibly lost for the same reason.**

[2 marks for definitely lost, 6 marks for possibly lost (3 each for the two cases).]

(b) 4 aaa aaa aaa aaa

Definitely lost: 400 bytes (The S array of size 100 bytes is malloc'ed 4 times and never freed)
No other leak is there as p never changes.

[2 marks for definitely lost. No marks for second line as only leaks are asked to be reported]

(c) Is there any invalid access problem that can be reported by the program? Justify your answer in a few sentences only

Yes. Since p can change in the check() function (as in case (a) above), but k remains unchanged in main(), the for loop in main() may access the p[] array out of bounds.

[2 marks]