CSSE 132 – Introduction to Computer Systems
Rose-Hulman Institute of Technology

SOLUTIONS To Exam 2 Practice - Paper Part

KEY

This part of the exam is **closed book**. You are allowed to use one single-sided 8 1/2 by 11 inch sheet of hand-written (by you) notes. You may not use a computer, cell phone, or additional resources other than those provided by your instructor.

Write all answers on these pages — use the back if necessary. Be sure to **show all work**.

All numbers are expressed in decimal unless specifically indicated otherwise.

Write your name on this page, then write your initials on all remaining pages of this exam. You are encouraged to read the entire exam before you start.

When you are finished with this part, bring this exam to your instructor and exchange it for the "code" part of the exam.

	Points available	Your marks
1	16	
2	15	
3	9	
4	10	
code	50	
Total	100	

Problem 1 (16 Points) For each of the following questions, circle the ONE BEST answer.

1.1 Which of the following statements about floating point numbers is **false**?

(a) The `float` and `double` are both floating point type but with different precision.

(b) →A floating point number has fixed precision for numbers in the range -1.0 to 1.0.

(c) A float is stored and its value is computed much like scientific notation

(d) The number 1/10 (one-tenth) cannot be represented precisely by a (binary) `float` datatype.

1.2 Which of the following statements about structs is true?

(a) Struct instances are always stored on the stack

(b) Struct members (things in the struct) are always stored in the heap

(c) →On a 32-bit machine, all pointers to structs have the same size: 4 bytes

(d) None of the above

1.3 Which of the following snippets of code causes space to be allocated on the stack?

(a) →`int x = 4;`

(b) `p = malloc(128);`

(c) `free(p);`

(d) `printf("Hi!\n");`

(e) All of the above

1.4 In ARM assembly, which of the following steps is **not** involved with procedure calls (i.e. it is not done by the caller or callee)?

(a) The arguments are passed in `r0, r1, r2, r3`

(b) →The stack pointer is set to 0

(c) The return address is stored for later use

(d) Local variables or registers that are needed later are stored on the stack

1.5 Which of the following is **true** about strings in C?

(a) They begin with a null character `'\0'`

(b) They are always stored on the stack

(c) The length of a string is determined with `sizeof()`

(d) →An individual character can be referenced using array subscript notation: `[]`

1.6 Given an array `A` of six integers, which statement will increment (add one to) the third element?

(a) `*(A+2) = (A+2) + 1;`

(b) →`A[2] = 1 + *(A+2)`

(c) `A[3]++;`

(d) `*(A+(2*4)) = A[2] + 1;`

1.7 Which of the following does *not* require the use of files in Linux I/O?

(a) Reading lines from a regular file

(b) Reading motion and clicks from the user's mouse

(c) →Reading input arguments from the command line

(d) Writing to standard output

1.8 Which of the following is **true** about using `fgets`? Assume we use the syntax the syntax `fgets(buf, size, fs)`.

(a) `fgets` may read as many as `size` bytes from the file into the buffer.

(b) →`fgets` null-terminates the bytes read into `buf`.

(c) `fgets` can only be used with standard input.

(d) `fgets` dynamically allocates memory to store the data read in from the file.

Problem 2 (15 Points) Examine this code and answer the questions about it on the following page. (Assuming the code runs on a 32-bit machine)

```
1  int func(int x)
2  {
3    int r = 0;
4    int i = 0;
5    while (i < x) {
6      r = r + i;
7    }
8    return r;
9  }
10
11 int otherfunc(int x, int y)
12 {
13   char *p;
14   int b = x + y;
15
16   p = malloc(x);
17   p = malloc(y);
18   p = malloc(b);
19
20   free(p);
21   return b;
22 }
```

```
1  func:
2    <not available>
3
4  otherfunc:
5     sub   sp, sp, #24
6     str   lr, [sp, #20]
7     str   r0, [sp, #4] ;x
8     str   r1, [sp]     ;y
9     ldr   r2, [sp, #4]
10    ldr   r3, [sp]
11    add   r3, r2, r3
12    str   r3, [sp, #16] ;b
13    ldr   r0, [sp, #4]
14    bl    0 <malloc>
15    str   r0, [sp, #12] ;p
16    ldr   r0, [sp]      ;y
17    bl    0 <malloc>
18    str   r0, [sp, #12] ;p
19    ldr   r0, [sp, #16] ;b
20    bl    0 <malloc>
21    str   r0, [sp, #12] ;p
22    bl    0 <free>
23    ldr   r0, [sp, #16] ;b
24    ldr   lr, [sp, #20]
25    add   sp, sp, #24
26    bx    lr
```

2.1 What is the minimum number of bytes that will be allocated on the stack for `func(2)`? *EXPLAIN how you arrived at this answer.*
12 bytes.

2.2 How many bytes does `otherfunc(1, 2)` allocate for itself on the stack?
24 bytes (first line of assembly)

2.3 How much of the allocated space is unused?
Only 20 bytes are needed, so 4 bytes are unused.

2.4 How many bytes does `otherfunc(1, 2)` allocate for itself on the heap?
$x + y + b = 1 + 2 + 3 = 6$ bytes

2.5 If you call the function `otherfunc(2,3)` with 1024 bytes available on the heap and 256 bytes available on the stack, how many bytes are available after `otherfunc` returns...

- On the stack?
  256 – *no change*

- On the heap?
  $1024 - x - y - b + b = 1024 - 2 - 3 = 1019$

Problem 3 (9 Points) The code is compiled to the program `a.out` and then executed with the command line:

```
>>> ./a.out 1 2 3 4
```

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv) {
4    char** p = argv;
5    int i;
6
7    for(i=0; i<argc; i++) {
8      printf("%s ", *p);
9      p++;
10   }
11
12   // printf("%s ", *(p-1));
13
14   return (p-argv);
15 }
```

Based on the given information, answer the following questions.

3.1 How many times will the `printf` in the loop be called?
argc is 5, so 5 times

3.2 What will be printed?
`./a.out 1 2 3 4`

3.3 What is `p` pointing to when the function completes?
The element just past the last argv element

3.4 What would the second `printf` print if it was uncommented?
p-1 would point to the last argv element, so 4 would be printed

3.5 What value does the function return?
5, since p points just past argv. Partial credit for 20

Problem 4 (10 Points) The code in `part2.c` is not complete. Finish the code (fill in the blanks) so that when it executes, the program:

- takes two command line arguments as input: a text and a key to be searched in the text,
- it will `count` the total number of occurrences of the key in the text,
- creates a string on the heap big enough to store the key `count` times,
- copies the key `count` times into the heap string, then
- prints the new string.

Be sure to avoid any memory leaks.

HINT: `memcpy` copies bytes from one buffer to another. It takes three arguments: destination address, source address, number of bytes.

EXAMPLE: if the program is called `part2`, it would behave like this when run 3 times:

```
pi ~$ ./part2 "hello, hello, I said hello you said goodbye" hello
Search Result: hellohellohello

pi ~$ ./part2 "hello, hello, I said hello you said goodbye" "hello goodbye"
Search Result:

pi ~$ ./part2 "hello, hello, I said hello you said goodbye" goodbye
Search Result: goodbye
```

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

void main(int argc, char** argv) {
    char* text = argv[1];
    char* key = argv[2];
    int text_len = strlen(text);
    int key_len = strlen(key);

    int i, count = 0;
    for (i = 0; i <= text_len - key_len; ++i) {
        int j, match = 1;
        for (j = 0; j < key_len; ++j) {
            if (text[i + j] != key[j]) {
                match = 0;
                break;
            }
        }
        if (match)
            count++;
    }
    char* result = malloc(count * key_len + 1);
    for (i = 0; i < count; ++i) {
        memcpy(result + i * key_len, key, key_len);
    }
    result[count * key_len] = '\0';
    printf("Search Result: %s\n", result);
    free(result);
}
```