CSSE 132 – Introduction to Computer Systems
Rose-Hulman Institute of Technology
Computer Science and Software Engineering Department

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	10	
3	15	
4	9	
5	10	
code	40	
Total	100	

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

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 and range from -1.0 to 1.0.
(c) A float is stored and its value is computed much like scientific notation
(d) None of the above are false

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) →All pointers to structs have the same size in your Pi: 4 bytes
(d) None of the above

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

Consider the 32-bit int `0xAB917310` stored in a little-endian system. Which byte is stored in the *lowest* memory address?

(a) `AB`
(b) `91`
(c) `73`
(d) →`10`

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

What tool do we most often use to convert high-level code into assembly code?

(a) A text editor

(b) A converter

(c) An assembler

(d) →A compiler

(e) None of the above

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: `[]`

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;`

Problem 2 (10 points) Read this ARM assembly code and answer the following questions. A simplified ARM guide is provided on the last page for your reference.

```
mystery
  mov r1, #0      ; sum=0:
  mov r3, #0      ; val=0
  b .L2           ;
.L3:
  add r1, r1, r3 ; sum += val
  add r3, r3, #1 ; val += 1
.L2:
  cmp r3, #9      ; while val <= 9
  ble .L3
  mov r0, r1     ; return value goes into r0
  b   lr
```

(a) Given the following initial register values below, fill in the values in all the registers after the above code finishes executing.

Register	initial value	final value
r0	0	45
r1	0	45
r2	0	0
r3	0	10

(b) Write a C function for `mystery` that does something equivalent to this assembly code. Assume `mystery` has one parameter stored in `r0`, and also assume local variables are not stored on the stack (registers are used instead).

```
int mystery() {
  int sum = 0;
  int val = 0;
  while (val <= 9) {
    sum = sum + val;
    val = val + 1;
  }
  return sum;
}
```

Problem 3 (15 Points) Examine this code and answer the questions about it on the following page.

```
int func(int x)
{
  int r = 0;
  int i = 0;
  while (i < x) {
    r = r + i;
  }
  return r;
}

int otherfunc(int x, int y)
{
  char *p;
  int b = x + y;

  p = malloc(x);
  p = malloc(y);
  p = malloc(b);

  free(p);
  return b;
}
```

```
func:
  <not available>

otherfunc:
   sub  sp, sp, #24
   str  lr, [sp, #20]
   str  r0, [sp, #4] ;x
   str  r1, [sp]     ;y
   ldr  r2, [sp, #4]
   ldr  r3, [sp]
   add  r3, r2, r3
   str  r3, [sp, #16] ;b
   ldr  r0, [sp, #4]
   bl   0 <malloc>
   str  r0, [sp, #12] ;p
   ldr  r0, [sp]      ;y
   bl   0 <malloc>
   str  r0, [sp, #12] ;p
   ldr  r0, [sp, #16] ;b
   bl   0 <malloc>
   str  r0, [sp, #12] ;p
   bl   0 <free>
   ldr  r0, [sp, #16] ;b
   ldr  lr, [sp, #20]
   add  sp, sp, #24
   bx   lr
```

(a) What is the minimum number of bytes that will be allocated on the stack for `func(2)`? *EXPLAIN how you arrived at this answer.*
8 bytes – or 12 if link register or x is stored (but it does not need to be).

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

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

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

(e) 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 4 (9 Points) The code is compiled to the program `a.out` and then executed with the command line:

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

```
#include <stdio.h>

int main(int argc, char** argv) {
  char** p = argv;
  int i;

  for(i=0; i<argc; i++) {
    printf("%s ", *p);
    p++;
  }

  // printf("%s ", *(p-1));

  return (p-argv);
}
```

Based on the given information, answer the following questions.

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

(b) What will be printed?
`a.out 1 2 3 4`

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

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

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

Problem 5 (10 Points) The code in `part2.c` is not complete. Finish the code (fill in the blanks) so that when it executes, the program:

(a) takes two command line arguments as input: a text and a key to be searched in the text,

(b) it will `count` the total number of occurrences of the key in the text,

(c) creates a string on the heap big enough to store the key `count` times,

(d) copies the key `count` times into the heap string, then

(e) prints the new string.

Be sure to avoid any memory leaks.

HINT: `strncpy` copies bytes from one string to another. It takes three arguments: destination string, source string, 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
```

RUN CODE: After you finish completing the code, run `./test_part2.sh` to test your implementation. The script will automatically compile the code and run several test cases.

```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) {
        strncpy(result + i * key_len, key, key_len);
    }
    result[count * key_len] = '\0';
    printf("Search Result: %s\n", result);
    free(result);
}
```

Basic ARM Assembly Guide

Register	Purpose
`lr`	holds return address
`sp`	address of stack top
`r0-r10`	general purpose registers
`r0`	also used for return value
`r0-r4`	also used for arguments

Instruction	What it does
`str x, y`	Store x into memory at y
`str x, [y, #a]`	Store x into memory at y+a
`ldr x, y`	Load memory at y into x
`ldr x, [y, #a]`	Load memory at y+a into x
`mov x, y`	Copy value of y into x
`sub x, y, z`	x = y - z
`add x, y, z`	x = y + z
`cmp x, y`	compare x to y, set conditions
`b LABEL`	go to LABEL (unconditionally)
`bl LABEL`	call procedure at LABEL
`bx lr`	return
`blt LABEL`	go to LABEL if condition says x $<$y
`ble LABEL`	go to LABEL if condition says x $\leq$ y
`bgt LABEL`	go to LABEL if condition says x $>$ y
`bge LABEL`	go to LABEL if condition says x $\geq$ y