CSE 30 Fall 2017 – Midterm 2

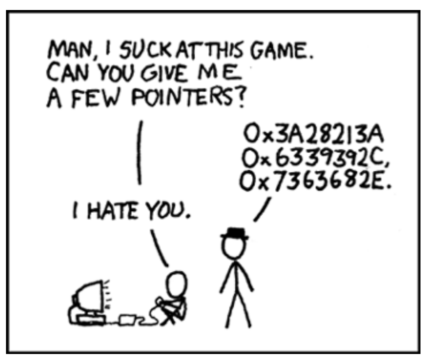
Name:

PID:

Email:

Only answers on this sheet will be graded. Per-question points are in italics. There are 70 total points.

1. *(5)*	8. *(12)*
2. *(6)*	9. *(3)*
3. *(3)*	10. *(4)*
4. *(8)*	11. *(4)*
5. *(6)*	12. *(3)*
6. *(4)*	13. *(3)*
7. *(7)*	14. *(3)*



Instructions If the answer is "None of the above," make sure to answer the appropriate letter for that response. If the answer is a sequence of letters, make sure to give an answer for each, and put them one after another in the answer cell. If an answer says "Write your answer directly in the answer sheet," follow the instructions in the question for the answer. You might write a single constant in a particular format, a short line of code, or several names of instructions or labels.

If you need to write an answer in hexadecimal, prefix it with `0x`.

For questions that are multiple-select (e.g. choose ALL that apply), you gain points both for putting correct answers and lose points for putting incorrect answers. You can't get negative points on a question.

Unless otherwise stated in the question, assume 32-bit words, ARM assembly semantics, and the version of `gcc` installed on the Raspberry Pis we use for programming assignments.

Remember to put your answers in the answer key!

We will not answer questions about the exam during the session. If you believe there is a mistake on the exam that makes a question be nonsense or un-answerable, note it on the question and on your answer sheet by writing "BAD QUESTION," and try to answer the question as best you can.

Memory Instructions

Consider the following layout of values on the heap, and register values (assume the little-endian layout that we usually see in GDB. For example, the byte at address `0x10001` is `0x53`):

r0	0x00010008
r1	0x00010000
r2	0x00000093
r3	0x12345693
r4	0x00000000

Address	Value
0x10000	0x51525354
0x10004	0x61626364

1. Which of the following sets of instructions, if run at this point, would end with the value `0x64` in `r4`? Choose ALL that apply:

   A. `ldr r4, [r1]`
   `and r4, #0x000000FF`
   `add r4, #10`

   B. `ldr r4, [r0, #-4]`
   `and r4, #0x000000FF`

   C. `ldrb r4, [r1]`
   `add r4, #10`

   D. `ldrb r4, [r1, #4]`
   `add r4, #10`

   E. `ldrb r4, [r0, #-4]`

   F. None of the above

2. Which of the following instructions, if run at this point, would change the byte holding `0x61` to hold `0x93`, while leaving the rest of memory unchanged? Consider the values in registers as shown above. Choose ALL that apply:

   A. `strb r2, [r0, #-1]`

   B. `strb r2, [r1, #7]`

   C. `strb r3, [r0, #-1]`

   D. `strb r3, [r1, #-1]`

   E. `str r2, [r0, #-1]`

   F. `str r2, [r1, #7]`

   G. `str r3, [r0, #-1]`

   H. `str r3, [r1, #7]`

3. What is the address of the byte above that holds the value `0x61`? Choose one:

   A. `0x10004`

   B. `0x10007`

   C. `0x10001`

   D. `0x10005`

   E. None of the above

C Strings

Consider the C library function `strcat` (adapted from the manual page on `strcat` provided with gcc):

```
char* strcat(char* dest, char* src)

       The  strcat() function appends the src string to the dest string, over-
       writing the terminating null byte ('\0') at the end of dest,  and  then
       adds  a  terminating  null  byte.  The strings may not overlap, and the
       dest string must have enough space for the  result.   If  dest  is  not
       large  enough, program behavior is unpredictable.
```

4. Which of the following programs WOULD have unpredictable behavior based on the description above? Note that by the length of a string above, the description above is referring to what `strlen` would return (which includes the count of characters up to but not including the 0 byte). Choose ALL that apply.

   A.
   ```
   char* s1 = malloc(7);
   char* s2 = malloc(2);
   s1[0] = 'a'; s1[1] = 'b'; s1[2] = 'c'; s1[3] = '\0';
   s2[0] = 'd'; s2[1] = '\0';
   strcat(s1, s2);
   ```
   B.
   ```
   char* s1 = malloc(5);
   char* s2 = malloc(3);
   s1[0] = 'a'; s1[1] = 'b'; s1[2] = 'c'; s1[3] = '\0';
   s2[0] = 'd'; s2[1] = 'e'; s2[2] = '\0';
   strcat(s1, s2);
   ```
   C.
   ```
   char* s1 = malloc(5);
   char* s2 = malloc(3);
   s1[0] = 'a'; s1[1] = 'b'; s1[2] = 'c'; s1[3] = '\0';
   s2[0] = 'd'; s2[1] = '\0';
   strcat(s1, s2);
   ```
   D.
   ```
   char* s1 = malloc(5);
   s1[0] = 'a'; s1[1] = 'b'; s1[2] = 'c'; s1[3] = '\0';
   strcat(s1, s1 + 1);
   ```
   E. None of the above

5. Consider this assembly implementation of `index_of`, which takes a `char*` and a `char` in `r0` and `r1` respectively, and ends with `r2` containing the index of where the character appears (starting with index 0), and -1 if it doesn't appear.

```
@ r0 contains address of string
@ r1 contains character to search for
@ r2 used for current index
@ r3 used to store current character
index_of:
  mov r2, #0
  mov r3, #0
loop:
  ------------------
  ------------------
  beq not_found
  ------------------
  beq found
  add r2, #1
  b loop
not_found:
  mov r2, #-1
found:
  @ do nothing, r2 already contains the correct value
```

Fill in the implementation above using the following instructions:

A. `cmp r3, #0`
B. `cmp r0, #0`
C. `cmp r2, #0`
D. `cmp r1, #0`
E. `cmp r3, r2`
F. `cmp r3, r1`
G. `cmp r0, r3`
H. `cmp r1, r2`
I. `ldrb r0, [r2, r3]`
J. `ldrb r3, [r0, r2]`
K. `ldrb r3, [r1, r2]`
L. `ldrb r3, [r0, r1]`

Calling Convention and the Stack

```
.text
.global f
f:
  sub sp, #8
  str lr, [sp, #4]
  str r4, [sp]
  mov r4, r0
  bl strlen
  lsl r0, #1
  add sp, #8
  ldr r4, [sp, #-8]
  ldr lr, [sp, #-4]
```

```
extern int f(char* s);
int main() {
  char* s = strdup("abcd");
  int twolen = f(s);
}
```

6. When called from C main as above, this implementation produces a segmentation fault. What's a likely explanation and fix?

   A. It never returns control to the caller by changing `pc`. Replace the use of `lr` in the last line with `pc`

   B. The program attempts to modify the string, which is read-only. The string shouldn't be allocated on the heap with `strdup`.

   C. It doesn't save the argument to the stack, so it gets overwritten. Create space for a local variable and store `r0` there by using `push` and `pop`.

   D. It doesn't save the original value of `sp`, so that register is overwritten. Add `push {sp}` at the beginning and `pop {sp}` at the end.

   E. It doesn't save the original value of `lr`, so that register is overwritten. Add `push {lr}` at the beginning and `pop {lr}` at the end.

   F. None of the above

7. Which of the following invariants does a callee need to maintain in the ARM 32-bit convention? Choose ALL that apply.

   A. The `sp` register must to be the same when returning as at the start of the function.

   B. The values of `r0-r3` must be the same when returning as at the start of the function.

   C. The values of `r4-r10` must be the same when returning as at the start of the function.

   D. The value of `lr` must be the same when returning as at the start of the function.

   E. The value of `pc` must be the same when returning as at the start of the function.

   F. The value of `pc` must be the same when returning as at the start of the function.

   G. All memory that has been allocated on the heap since the beginning of the function must be freed.

8. Consider the following program, and the corresponding ARM assembly that GCC generated for it (slightly shortened):

```
check_mod:
  push  {lr}
  sub sp, sp, ___________
  str r0, _____________
  str r1, _____________
  str r2, _____________
  ldr r0, [sp, #12]
  ldr r1, [sp, #8]
  bl  __aeabi_divmod
  mov r2, r1
  ldr r3, [sp, #4]
  cmp r2, r3
  moveq r0, #1
  movne r0, #0
  add sp, sp, #20
  pop {pc}
main:
  push  {r3, lr}
  mov r0, #27
  mov r1, ___________
  mov r2, ___________
  bl  1041c <check_mod>
  mov r0, r3
  pop {r3, pc}
```

```
int check_mod(int n, int d, int c) {
  return n % d == c;
}

int main() {
  check_mod(27, 5, 2);
}
```

There are some holes in the generated assembly. Fill them in, in order, using the instructions and instruction fragments below. Give your answer as a sequence of 6 letters in order.

A. `#27`
B. `#16`
C. `#8`
D. `[sp, #8]`
E. `[sp, #4]`
F. `[sp]`
G. `#20`
H. `#5`
I. `#2`
J. `[sp, #20]`
K. `[sp, #16]`
L. `[sp, #12]`

Dynamic Memory Allocation

Consider this C program for the next two questions:

```
int* make_list(int upto) {
  int i;
  int* nums = malloc(sizeof(int) * upto);
  for(i = 0; i < upto; i += 1) {
    nums[i] = i * i;
  }
  return nums;
}
int main() {
  int* some_nums = make_list(7);
  int i;
  int sum = 0;
  for(i = 0; i < 7; i += 1) {
    sum += some_nums[i];
  }
}
```

9. Which of the following statements could replace `nums[i] = i` above without changing the program's meaning? Choose ALL that apply.

   A. `nums + i = i * i;`

   B. `*(nums + i) = i * i;`

   C. `(*nums) + i = i * i;`

   D. `*nums[i] = i * i;`

   E. None of the above

10. This program leaks memory – all of the bytes allocated for the array. Where is the most appropriate place to `free` the memory?

   A. At the end of `make_list`, before the `return nums` statement.

   B. Before the `for` loop in the `main` function, but after `sum` is declared.

   C. Before the `for` loop in `make_list`

   D. Immediately after the `make_list` function returns

   E. None of the above

11. Consider this assembly program and two lines of C code that generated them (lightly edited from the raw output of `gcc`).

```
mov r0, #12
bl  malloc
mov r3, r0             int* m = malloc(sizeof(int) * 3);
add r3, r3, #8         __________ = 10;
mov r2, #10
str r2, [r3]
```

Which of the following left-hand-sides would fill in the C program to correspond to the generated assembly? Assume that `int`s take 4 bytes of memory.

A. `nums[0]`

B. `nums[8]`

C. `nums[7]`

D. `nums[2]`

E. `nums[3]`

F. None of the above

Structs

Consider the following program:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  
4  struct Point {
5    int x;
6    int y;
7  };
8  
9  struct Pair {
10   struct Point* left;
11   struct Point* right;
12 };
13 
14 struct Point* f(struct Point* p) {
15   p->x = 10;
16   p = malloc(sizeof(struct Point)); // Malloc A
17   p->x = 9;
18   p->y = 11;
19   return p;
20 }
21 int main() {
22   struct Point* pt1 = malloc(sizeof(struct Point)); // Malloc B
23   pt1->x = 1;
24   pt1->y = 2;
25 
26   struct Point* pt2 = malloc(sizeof(struct Point)); // Malloc C
27   pt2->x = 3;
28   pt2->x = 4;
29 
30   struct Pair* pair = malloc(sizeof(struct Pair)); // Malloc D
31   pair->left = pt1;
32   pair->right = pt2;
33 
34   struct Point* pt = f(pair->left);
35 
36   printf("%d, %d, %d", pair->left->x, pt1->x, pt->x);
37 
38   free(pair->left);
39   free(pair->right);
40   free(pair);
41 }
```

When run on this program, Valgrind reports (among other things):

```
==4112== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4112==    at 0x4833970: malloc (vg_replace_malloc.c:263)
==4112==    by 0x104A3: f (structs.c:16)
==4112==    by 0x10567: main (structs.c:34)
```

12. Assuming `int`s are 4 bytes, and pointers are 4 bytes, and structs take space equal to the sum of their members, how much total memory does this program allocate with `malloc`, whether it's freed or not by the end? Give your answer directly in the answer sheet as a number of bytes.

13. For each of the three `free` expressions at the bottom, indicate the line of the `malloc` that produced the address being freed. Use the letter given on the malloc line (A-D), give your answer as a sequence of three letters.

14. What does this program print? Write your answer directly into the answer sheet as three numbers separated by commas.

Scratch paper

Scratch paper