CS360 Midterm 2 - March 29, 2018 - James S. Plank - Page 1

Put all answers on the answer sheet. In all of these questions, please assume the following:

- Pointers and longs are 4 bytes.
- The machine is little endian, but that doesn't matter in any of these questions.
- There are no segmentation violations or bus errors in any of this code.
- Any assembly code questions use the **jassem** assembly code.
- Any assembly code that corresponds to compiled C code is unoptimized.

If you give any numeric answer, I don't care whether you give it in decimal or hexadecimal. However, if you give it in hexadecimal, then you **must** precede it with `"0x"`. In particular, if you give me an answer of 10, but you really meant `0x10`, you will not receive credit for the answer.

Question 1

Your colleague has written the following procedure. You can see what it does, but if you'd like an explanation, here's one. The procedure takes an array of integers called **ints**, and array of **chars** called **select**. They are the same size, which is **size**. For each element of **ints**, if its corresponding element of **select** is non-zero, then the procedure writes the integer to file descriptor **fd**.

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

void write_ints(int fd, int size, int *ints, char *select)
{
  int i;

  for (i = 0; i < size; i++) {
    if (select[i] != 0) write(fd, &(ints[i]), sizeof(int));
  }
}
```

Now, your colleague is in trouble -- **write_ints()** is *really* slow.

Part 1: Why is **write_ints()** slow? Choose your answer from the following selections, and circle the correct answer on the answer sheet:

- *a*. It is written in C, rather than Python.
- *b*. There is too much system call overhead.
- *c*. Integers are four bytes, while chars are one.
- *d*. **write_ints** is $O(size^2)$.
- *e*. Disks are slow, so the program's speed is limited by disk speed.
- *f*. **select** is wasteful of memory, because you can pack eight boolean values into a char, rather than one.
- *g*. There are too many parameters to **write_ints**, so you spend too much time pushing values onto the stack.

Part 2: Your job is to rewrite it and make faster, while still using **write()**. Use the answer sheet. You may only use the variables declared there, and you cannot use any system or library calls besides **write()**.

CS360 Midterm 2 - March 29, 2018 - James S. Plank - Page 2

Question 2

Answer the following questions:

- *A*. (1/2 sentences): From the shell, How do you determine whether two files are hard links to each other?
- *B*. (1/2 sentences): Inside your C program, how do you determine whether a file is a directory?
- *C*. (1/2 sentences): What information is held in a directory?
- *D*. (1/2 sentences): When emitting assembly code for a procedure, why would you spill **r2**?
- *E*. Suppose I perform the following actions, with the following results:

```
UNIX> cd xxx
UNIX> ls f1
f1
UNIX>
```

Now, I run a program that does:

```
fd = open("f1", O_WRONLY | O_CREAT | O_TRUNC, 0666);
```

Give me two significantly different reasons why **fd** may end up being -1.

Question 3

Please write the assembly code for the following three procedures:

```
int b(int j)
{
  int i;

  for (i = 0; i < 10; i++) {
    j += i;
  }
  return j;
}

int c(int *p, int k)
{
  return p[k];
}

int d()
{
  return e(5, g()) + f();
}
```

Question 4

Suppose we are running `z()`, and we are are about to run the instruction `"jsr y"`. The frame pointer equals `0x300444` and the program counter equals `0x3458`. At the point where `y()` is about to call `"ret"`, please tell me the following:

- *a*. What is the value of the frame pointer?
- *b*. What is the value of the stack pointer?
- *c*. What is the value of the register **r0**?
- *d*. What is the value of the four bytes starting at `0x300430`?
- *e*. What is the value of the four bytes starting at `0x300434`?
- *f*. What is the value of the four bytes starting at `0x300438`?
- *g*. What is the value of the four bytes starting at `0x30043c`?
- *h*. What is the value of the four bytes starting at `0x300440`?
- *i*. What is the value of the four bytes starting at `0x300444`?
- *j*. What is the instruction at memory location `0x345c`?

```
int y(int j, int k)
{
  int i;

  i = j*k;
  k--;
  return i;
}
```

```
int z()
{
  int m;

  m = 5;
  m = y(11, 7);
}
```

Question 5

Below are 240 bytes of memory, all of which are either allocated or free. Suppose that **malloc()** and **free()** have been implemented as described in class, and the head of the free list is **0x78649d48**.

Part A: On the answer sheet, please list the starting address and size of each block on the free list, in the order in which it is on the free list. As you can see, I have filled in the `0x78649` parts of the addresses, so you don't have to write them yourselves.

Part B: On the answer sheet, please list the starting address and size of each allocated block of memory. List this in ascending order of address.

Part C: If I call **malloc(16)**, what address will be returned?

Part D: Suppose **sbrk(0)** returns `0x78049e10`. Will I segfault if I access memory location `0x78049e20`? Why or why not?

Part E: Suppose instead that **sbrk(0)** returns `0x78049df8`. We have a problem now, with the state of our memory system. If we never call **malloc()** or **free()** again, no bugs will manifest. Give me a plausible sequence of actions that will result in serious problems (and of course why the problems will result).

Address	Value	Address	Value	Address	Value
0x78649d20	0x10	0x78649d70	0x20	0x78649dc0	0x78649da8
0x78649d24	0x78649de4	0x78649d74	0x78649d6c	0x78649dc4	0x8
0x78649d28	0x20	0x78649d78	0x78649dc0	0x78649dc8	0x18
0x78649d2c	0x78649e08	0x78649d7c	0x1c	0x78649dcc	0x78649d4c
0x78649d30	0x18	0x78649d80	0x78649d30	0x78649dd0	0x78649d70
0x78649d34	0x78649df8	0x78649d84	0x78649df8	0x78649dd4	0x20
0x78649d38	0x78649da0	0x78649d88	0x18	0x78649dd8	0x18
0x78649d3c	0x10	0x78649d8c	0x78649dec	0x78649ddc	0x8
0x78649d40	0x78649d5c	0x78649d90	0x10	0x78649de0	0x18
0x78649d44	0x8	0x78649d94	0x78649d38	0x78649de4	0x78649ddc
0x78649d48	0x10	0x78649d98	0x78649e04	0x78649de8	0x78649db4
0x78649d4c	0x78649da0	0x78649d9c	0x0	0x78649dec	0x24
0x78649d50	0x0	0x78649da0	0x18	0x78649df0	0x78649d40
0x78649d54	0x78649db0	0x78649da4	0x78649d30	0x78649df4	0x78649dc0
0x78649d58	0x18	0x78649da8	0x78649d48	0x78649df8	0x18
0x78649d5c	0x78649dd4	0x78649dac	0x78649dbc	0x78649dfc	0x0
0x78649d60	0x78649d84	0x78649db0	0x0	0x78649e00	0x78649d30
0x78649d64	0x20	0x78649db4	0x78649e00	0x78649e04	0x78649da0
0x78649d68	0x78649d54	0x78649db8	0x10	0x78649e08	0x18
0x78649d6c	0x20	0x78649dbc	0x78649d34	0x78649e0c	0x0