CS360 Midterm – Spring, 2015 – James S. Plank – March 10

In all questions, assume that pointers are 4 bytes, and that the machine architecture is like **jassem**.
Do your answers on the answer sheets given. Do not do them on the exam itself.

Question 1

The first line of this program's output is "Line 0: 0xef".

Tell me the remaining lines.

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

main()
{
  unsigned int i;
  unsigned int j;
  unsigned int h;
  unsigned char k[8];
  unsigned int *p;

  p = (unsigned int *) k;
  for (i = 0; i < 8; i++) k[i] = i;
  h = 0xf7d9c;
  i = 0x12345678;
  j = 0x90abcdef;

  memcpy(k+2, &j, sizeof(int));

  printf("Line 0: 0x%x\n", k[2]);
  printf("Line 1: 0x%x\n", (i >> 4));
  printf("Line 2: 0x%x\n", (h << 1));
  printf("Line 3: 0x%x\n", k[7]);
  printf("Line 4: 0x%x\n", k[5]);
  printf("Line 5: 0x%x\n", p[0]);
  printf("Line 6: 0x%x\n", p[1]);
}
```

Question 2

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

char *build_string(char **words, int numwords)
{
  int i, rvsize;
  char *rv;

  rvsize = (numwords == 0) ? 1 : numwords;
  for (i = 0; i < numwords; i++) {
    rvsize += strlen(words[i]);
  }
  rv = (char *) malloc(sizeof(char)*rvsize);

  strcpy(rv, "");
  for (i = 0; i < numwords; i++) {
    if (i > 0) strcat(rv, " ");
    strcat(rv, strdup(words[i]));
  }
  return rv;
}
```

A colleague has presented you with the C procedure **build_string()** above, which allocates and builds a string from an array of words. Because you have taken the first half of CS360, you instantly spot two major problems with this procedure (besides the lack of comments). Tell me what they are, and then rewrite **build_string()** so that it doesn't have these problems.

I have reproduced the code above on the answer sheet, so that you can simply cross out and add to it, rather than rewriting it.

Question 3

Write me a single line of C code that has 60 characters or less, and that will require the assembly code to spill a register. Don't worry about variable declarations – your code should be straightforward enough that I can figure out any variables. After you write the line of code, explain to me exactly why it has to spill the register.

Useful Prototypes:

```
void memcpy(void *to, void *from, int numbytes);
void strcpy(char *to, char *from);
void strcat(char *to, char *from);
char *strdup(char *string_to_duplicate)
```

```
int *a(int ***x, int *y)
{
  int i;
  int **b;

  i = *y;
  b = *x;
  return b[i];
}
```

Question 4

Please give me the jassem assembly code for the C procedure to the right.
As always, your compiler should not optimize.

Question 5

Below, I present for you the contents of memory at the time that procedure **a()** above is called.
At that point, the frame pointer and the stack pointer both equal `0xfff40c`.

Please answer the following questions when **a()** is just about to return. Your answers should be numbers in hexadecimal. I don't want to see answers like "[fp]". I want numbers.

A: What is the address of **x**?
B: What is the address of **y**?
C: What is the address of **i**?
D: What is the address of **b**?
E: What is the address of **(*x)**?
F: What is the value of **(*x)**?
G: What is the value of **(**x)**?
H: What is the value of **b**?
I: What is the value of **i**?
J: What does **a()** return?
K: What is the value of ***(b[i])**?
L: How many bytes of local variables are in the procedure that called **a()**?
M: When **a()** returns, to what value will the program counter be set?
N: When **a()** returns, to what value will the frame pointer be set?
O: When **a()** returns, to what value will the stack pointer be set?
P: There are three stack frames on the stack.
What are their three frame pointers?

0xfff400	0x0
0xfff404	0x0
0xfff408	0x0
0xfff40c	0x0
0xfff410	0xfff41c
0xfff414	0x1054
0xfff418	0xfff42c
0xfff41c	0xfff428
0xfff420	0xfff448
0xfff424	0x10b4
0xfff428	0x2
0xfff402c	0xfff440
0xfff430	0xaa
0xfff434	0xbb
0xfff438	0xcc
0xfff43c	0x0
0xfff440	0xfff434
0xfff444	0xfff438
0xfff448	0xfff430

Question 6

You are programming an embedded device whose only stable storage is a 32MB solid state storage device (otherwise known as an SSD). The interface to this device is as follows. The device is partitioned into 8192 pages, each of which is 4096 bytes. There are two global variables that you get to use. The first is a 4096-byte buffer, which is in the global variable **ssd_buf** (its type is **char ***). It holds the contents of one of these pages. The second is **ssd_bufid**, which is the page number of the page in **ssd_buf**. This will be a number between 0 and 8191 (or -1 if there is no page in **ssd_buf**).

There are two system calls that you can use with respect to the SSD:

- **void load_ssd_page(int pagenumber)**: This will read the specified page from the SSD into the memory pointed to by **ssd_buf,** and it will set **ssd_bufid** to **pagenumber**.
- **void flush_ssd_page()**: This writes the page in **ssd_buf** to page **ssd_bufid** on the SSD.

Both of these do nothing if given a bad **pagenumber**, or if **ssd_bufid** is not between 0 and 8191.

Your company has hired a summer intern, whose sole job was to write two procedures:

- `void ssd_read(char *buf, int size, int ssd_address)`
- `void ssd_write(char *buf, int size, int ssd_address)`

These allow the users to treat the SSD as a big block of 32MB, and treat **ssd_address** as a pointer into that block. For example, if you wanted to write the five characters "CS360" into the middle of the first page, starting at byte 100, you would call `ssd_write("CS360", 5, 100);` If you wanted to write them to the beginning of the second page, you would call `ssd_write("CS360", 5, 4096).` And if you wanted the "CS" to go into the last two bytes of the first page, and the "360" to into the first three bytes of the second page, you could call `ssd_write("CS360", 5, 4094).` That's convenient, isn't it?

The summer intern was, in my opinion, a little lazy, and wrote the following code:

```
void ssd_read(char *buf, int size, int a)
{
  int i;

  for (i = 0; i < size; i++) {
    load_ssd_page((a+i) / 4096);
    buf[i] = ssd_buf[(a+i)%4096];
    flush_ssd_page((a+i) / 4096);
  }
}
```

```
void ssd_write(char *buf, int size, int a)
{
  int i;

  for (i = 0; i < size; i++) {
    load_ssd_page((a+i) / 4096);
    ssd_buf[(a+i)%4096] = buf[i];
    flush_ssd_page((a+i) / 4096);
  }
}
```

At least it works. However, it is so slow that your embedded device is unusable. Your manager is worried that the SSD itself is too slow, and is thinking of bidding new companies for faster SSD's, but just in case it's not the SSD, she has hired you to to see if you can speed up **ssd_read()** and **ssd_write()**. Your job on this exam is to explain to me all of the reasons why these two procedures are too slow – both major reasons and minor reasons. Then, you need to rewrite **ssd_read()** (just **ssd_read()** --You don't have to write **ssd_write()**). Rewrite it so that it has none of the problems of the previous **ssd_read()**. One hint (and this is just one hint, because I'm thinking you might not think of it otherwise) – you should consider using **memcpy()** in your implementation, and then think about why.