Full Name: ________________________________

Andrew ID (print clearly!): ________________________________

15-213/18-213, Spring 2013

Final Exam Review Session
Wednesday, May 8, 2013

Instructions:

• Make sure that your exam has 27 pages and is not missing any sheets, then write your full name and Andrew login ID on the front.

• This exam will be closed book. You may not use any electronic devices. You may use two single-sided page of notes that you bring to the exam. However, for today you can just ask us for the answers.

• Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.

• The exam has a maximum score of 100 points.

• The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Your Score</th>
<th>Possible Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 9. (15 points):

Structure alignment

Consider the following C struct.

```c
struct st1_t {
    char a;
    char b;
    char c;
};

struct st2_t {
    st1_t d;
    st1_t e;
    st1_t *f;
    short g;
    char h;
    double i;
    long j;
};
```

A. Show how the st1_t struct above would appear on a 32 bit Linux system. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Page 24 of 33
B. Show how the `st2_t` struct above would appear on a 32 bit Linux system. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
| | | | | | | | | +---+
| | | | | | | | | +---+
| | | | | | | | | +---+
| | | | | | | | | +---+
| | | | | | | | | +---+
| | | | | | | | | +---+
```

Page 25 of 33
Problem 2. (10 points):  
Floating point encoding. In this problem, you will work with floating point numbers based on the IEEE floating point format. We consider two different formats:

**Format A:** 8-bit floating point numbers:
- There is one sign bit $s$. $s = 1$ indicates negative numbers.
- There are $k = 4$ exponent bits. The bias is $2^{k-1} - 1 = 7$.
- There are $n = 3$ fraction bits.

**Format B:** 9-bit floating point numbers:
- There is one sign bit $s$. $s = 1$ indicates negative numbers.
- There are $k = 4$ exponent bits. The bias is $2^{k-1} - 1 = 7$.
- There are $n = 4$ fraction bits.

1. How would you represent positive infinity using **format A**?
   
   Binary representation for positive infinity: ________________

2. How would you represent $\sqrt{-100}$ using **format B**?
   
   Give an example binary representation: ________________

3. For formats A and B, please write down the binary representation and the corresponding values for the following (use round-to-even):

<table>
<thead>
<tr>
<th>Description</th>
<th>Format A binary</th>
<th>Format A value</th>
<th>Format B binary</th>
<th>Format B value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0 0000 000</td>
<td>0</td>
<td>0 0000 0000</td>
<td>0</td>
</tr>
<tr>
<td>Largest normalized value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest positive number</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negative one</td>
<td></td>
<td>$-1$</td>
<td></td>
<td>$-1$</td>
</tr>
<tr>
<td>2.625</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 3. (6 points):
Arrays. Consider the C code below, where H and J are constants declared with `#define`.

```c
int array1[H][J];
int array2[J][H];

void copy_array(int x, int y) {
    array2[x][y] = array1[y][x];
}
```

Suppose the above C code generates the following x86-64 assembly code:

```assembly
# On entry:
#   %edi = x
#   %esi = y
#
# copy_array:
#    movslq %esi,%rsi
#    movslq %edi,%rdi
#    movq  %rdi, %rax
#    salq  $4, %rax
#    subq  %rdi, %rax
#    addq  %rsi, %rax
#    leaq  (%rsi,%rsi,4), %rsi
#    leaq  (%rdi,%rsi,2), %rsi
#    movl  array1(%rsi,%rsi,4), %edx
#    movl  %edx, array2(%rax,4)
#    ret
```

What are the values of H and J?

H =

J =
Problem 2. (14 points):

Stack discipline.

Consider the following C code and assembly code for two mutually recursive functions:

```c
int even(unsigned int n) 0x080483e4 <even+0>: push %ebp
{ 0x080483e5 <even+1>: mov %esp,%ebp
   if(!n) 0x080483e7 <even+3>: sub $0x8,%esp
   { 0x080483ea <even+6>: cmpl $0x0,0x8(%ebp)
      return 1;
   } 0x080483ee <even+10>: jne 0x80483f9 <even+21>
   return odd(n - 1);
} 0x080483fc <even+24>: sub $0x1,%eax
0x080483ff <even+27>: movl $0x1,-0x4(%ebp)
0x08048402 <even+30>: jmp 0x804840a <even+38>
return odd(n - 1);
} 0x0804840f <odd+0>: push %ebp
{ 0x08048410 <odd+1>: mov %esp,%ebp
   if(!n) 0x08048412 <odd+3>: sub $0x8,%esp
   { 0x08048415 <odd+6>: cmpl $0x0,0x8(%ebp)
      return 0;
   } 0x08048419 <odd+10>: jne 0x8048424 <odd+21>
   return even(n - 1);
} 0x08048422 <odd+19>: jmp 0x8048435 <odd+38>
0x08048424 <odd+21>: mov 0x8(%ebp),%eax
0x0804842d <odd+30>: call 0x80483e4 <even>
0x08048432 <odd+35>: mov %eax,-0x4(%ebp)
0x08048435 <odd+38>: mov -0x4(%ebp),%eax
0x08048438 <odd+41>: leave
0x08048439 <odd+42>: ret
```

Imagine that a program makes the procedure call `even(3)`. Also imagine that prior to the invocation, the value of `%esp` is `0xffff1000`—that is, `0xffff1000` is the value of `%esp immediately before the execution of the call instruction.`
A. Note that the call `even(3)` will result in the following function invocations: `even(3)`, `odd(2)`, `even(1)`, and `odd(0)`. Using the provided code and your knowledge of IA32 stack discipline, fill in the stack diagram with the values that would be present immediately before the execution of the `ret` instruction for `odd(0)`. Cross out each blank for which there is insufficient information to complete.

```
+-----------------------------------+    | 0xffff1004
|                                   |    | 0xffff1000
|                                   |    | 0xffff0ffc
|                                   |    | 0xffff0ff8
|                                   |    | 0xffff0ff4
|                                   |    | 0xffff0ff0
|                                   |    | 0xffff0fec
|                                   |    | 0xffff0fe8
|                                   |    | 0xffff0fe4
|                                   |    | 0xffff0fe0
|                                   |    | 0xffff0fdc
|                                   |    | 0xffff0fd8
|                                   |    | 0xffff0fd4
|                                   |    | 0xffff0fd0
|                                   |    | 0xffff0fcc
|                                   |    | 0xffff0fc8
|                                   |    | 0xffff0fc4
|                                   |    | 0xffff0fc0
```

B. What are the values of `%esp` and `%ebp` immediately before the execution of the `ret` instruction for `odd(0)`?
Problem 5. (6 points):
Loops. Consider the following x86-64 assembly function:

```
loopy:
    # a in %rdi, n in %esi
    movl  $0, %ecx
    movl  $0, %edx
    testl %esi, %esi
    jle   .L3
.L6:
    movslq %edx,%rax
    movl  (%rdi,%rax,4), %eax
    cmpl  %eax, %ecx
    cmovl %eax, %ecx
    addl  $1, %edx
    cmpl  %ecx, %esi
    jg    .L6
.L3:
    movl  %ecx, %eax
    ret
```

Fill in the blanks of the corresponding C code.

- You may only use the C variable names \( n \), \( a \), \( i \) and \( x \), not register names.
- Use array notation in showing accesses or updates to elements of \( a \).

```c
int loopy(int a[], int n)
{
    int i; 
    int x = _____;

    for(i = ____________; ____________; ____________) {

        if (____________)
            x = ____________;
    }
    return x;
}
```
Problem 3. (10 points):

Consider the following x86-64 assembly function, called foo.

```assembly
foo:  # rdi = t,  rsi = v
    pushq  %r12
    pushq  %rbp
    pushq  %rbx

.LCFI2:
    movq  %rdi, %rbx
    movq  %rsi, %r12
    testq  %rdi, %rdi
    je   .L3
    movl  (%rsi), %ebp
    cmpl  24(%rdi), %ebp
    jne  .L12
    jmp   .L5
.L7:
    cmpl  %ebp, 24(%rbx)
    jne  .L12
.L5:
    leal  1(%rbp), %edx
    movq  16(%rbx), %rax
    addl  (%rax,%rdx,4), %ebp
    movl  %ebp, %eax
    jmp   .L8
.L12:
    movq  %r12, %rsi
    movq  (%rbx), %rdi
    call   foo
    testl  %eax, %eax
    je   .L9
    movl  %ebp, %eax
    jmp   .L8
.L9:
    movq  8(%rbx), %rbx
    testq  %rbx, %rbx
    jne  .L7
.L3:
    movl  $0, %eax
.L8:
    popq  %rbx
    popq  %rbp
    popq  %r12
    ret
```
Fill in the blanks of the corresponding C code.

- The function used the data structure "Node" as defined below:

```c
struct Node {
    struct Node *left;
    struct Node *right;
    unsigned int *value;
    unsigned int index;
};
```

- You may use only the C variable names that are defined, not the register names.

```c
int foo( __________________ t , unsigned int * v) {
    if ( t == _________ )
        return 0;
    if( ______________________________ ) {
        return _________________________________________________;
    }
    return ( _____________________________________________?
        ______________________________________________ :
        ______________________________________________ );
}
```
Problem 12. (12 points):

Address translation. This problem deals with virtual memory address translation using a multi-level page table, in particular the 2-level page table for a 32-bit Intel system with 4 KByte pages tables. Assume all processes are running under Supervisor mode. The following diagrams are direct from the Intel System Programmers guide and should be used on this problem:
The contents of the relevant sections of memory are shown on this page. All numbers are given in hexadecimal. Any memory not shown can be assumed to be zero. The Page Directory Base Address is 0x0045d000.

For each of the following problems, perform the virtual to physical address translation. If an error occurs at any point in the address translation process that would prevent the system from performing the lookup, then indicate this by circling FAILURE and noting the physical address of the table entry that caused the failure.

For example, if you were to detect that the present bit in the PDE is set to zero, then you would leave the PTE address in (b) empty, and circle FAILURE in (c), noting the physical address of the offending PDE.

<table>
<thead>
<tr>
<th>TLB</th>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Tag</td>
<td>Frame Number</td>
</tr>
<tr>
<td>0</td>
<td>0x03506</td>
<td>0x98f8a</td>
</tr>
<tr>
<td></td>
<td>0x27f4a</td>
<td>0x34abe</td>
</tr>
<tr>
<td>1</td>
<td>0x1f7ee</td>
<td>0x95dbc</td>
</tr>
<tr>
<td></td>
<td>0x2a064</td>
<td>0x72954</td>
</tr>
<tr>
<td>2</td>
<td>0x1f7f0</td>
<td>0x95ede</td>
</tr>
<tr>
<td></td>
<td>0x2005d</td>
<td>0xaa402</td>
</tr>
<tr>
<td>3</td>
<td>0x3fc2e</td>
<td>0x2029e</td>
</tr>
<tr>
<td></td>
<td>0x3df82</td>
<td>0xff644</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>000c3020</td>
<td>345ab236</td>
</tr>
<tr>
<td>000c3080</td>
<td>345ab237</td>
</tr>
<tr>
<td>000c332f</td>
<td>08e4523f</td>
</tr>
<tr>
<td>000c3400</td>
<td>93c2ed98</td>
</tr>
<tr>
<td>000c3cbc</td>
<td>34abd237</td>
</tr>
<tr>
<td>000c3ff0</td>
<td>93c2ed99</td>
</tr>
<tr>
<td>000c4020</td>
<td>8e56e237</td>
</tr>
<tr>
<td>000c432f</td>
<td>33345237</td>
</tr>
<tr>
<td>000c4400</td>
<td>43457292</td>
</tr>
<tr>
<td>000c4cbc</td>
<td>385ed293</td>
</tr>
<tr>
<td>000c4ff0</td>
<td>c3726292</td>
</tr>
<tr>
<td>0045d000</td>
<td>000c3292</td>
</tr>
<tr>
<td>0045d028</td>
<td>000c4297</td>
</tr>
<tr>
<td>0045d032</td>
<td>0df2a292</td>
</tr>
<tr>
<td>0045d0a0</td>
<td>000c3297</td>
</tr>
<tr>
<td>0045d3ff</td>
<td>0df2a236</td>
</tr>
<tr>
<td>0045d9fc</td>
<td>0df2a237</td>
</tr>
<tr>
<td>0df2a000</td>
<td>deded000</td>
</tr>
<tr>
<td>0df2a080</td>
<td>bc3de239</td>
</tr>
<tr>
<td>0df2a3fc</td>
<td>000c4296</td>
</tr>
<tr>
<td>0df2a4a0</td>
<td>00324236</td>
</tr>
<tr>
<td>0df2a4fc</td>
<td>df72c9a6</td>
</tr>
<tr>
<td>0df2b080</td>
<td>01f008c3</td>
</tr>
<tr>
<td>0df2bff0</td>
<td>000c5112</td>
</tr>
</tbody>
</table>
1. Read from virtual address 0x9fd28c10. Scratch space:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) (TLB Hit) Physical address is: \(0x\) 

OR

(b) Physical address of PDE: \(0x\) 

(c) Physical address of PTE: \(0x\) 

(d) (SUCCESS) The physical address accessed is: \(0x\) 

OR

(Failure) The physical address of the table entry causing the failure is: \(0x\)
2. Read from virtual address 0x0d4182c0. Scratch space:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) (TLB Hit) Physical address is: 0x <br>
OR <br>
(b) Physical address of PDE: 0x <br>
(c) Physical address of PTE: 0x <br>
(d) (SUCCESS) The physical address accessed is: 0x <br>
   OR <br>
   (FAILURE) The physical address of the table entry causing the failure is: 0x
3. Read from virtual address 0x0a32fcd0.
Scratch space:

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16
    15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

(a) (TLB Hit) Physical address is: 0x

OR

(b) Physical address of PDE: 0x

(c) Physical address of PTE: 0x

(d) (SUCCESS) The physical address accessed is: 0x

OR

(FAILURE) The physical address of the table entry causing the failure is: 0x
IA32 Two-Level Page Tables

IA32 Two-Level Page Tables: PDE Flags

Page Directory Entry

IA32 Two-Level Page Tables: PTE Flags

Page Table Entry
Problem 8. (10 points):
The Curse of Abalienation!!

For this question, we will be looking at the 32-bit libc implementation of malloc.

- The libc implementation uses an 8 byte alignment of the payload areas.
- The libc implementation uses the following layout for free blocks:

  | header | prev | next | payload | footer |
  | 4 bytes | 4 bytes | 4 bytes | arbitrary size | 4 bytes |

  Where prev, next and footer are stored inside the space for the payload.

- The libc implementation uses the following layout for allocated blocks:

  | header | payload |
  | 4 bytes | arbitrary size |

Your friend, Harry Q. Bovik, is taking 15-123, where one of the assignments is to write a linked list implementation of a dictionary. Harry is experiencing a strange bug where his dictionary works on everything except for 12 letter words, on which it generates a Segmentation Fault. After some debugging you find that it also doesn’t work on words of size 20 and 28 (you don’t test any further).

Here is Harry’s addWordDict method:

```c
int addWordDict(dictionary * dict, char * word){
    int result;
    char * wordCopy;
    if (dict == NULL){
        return ERR_NULL_DICT;
    }
    if(word == NULL){
        return WARN_INVALID_ARGUMENT;
    }
    /*add the word */
    /*We’re going to make a copy of the word because the word buffer gets reused. This wordCopy will get free’d when we remove the word from the dictionary. */
    wordCopy = (char *)malloc((strlen(word)) * sizeof(char));
    strcpy(wordCopy,word);
    result = addItemLL(((dict)->wordList),(void*) wordCopy);
    dict->count = ((dict)->wordList)->count; /*update the count */
    return result;
}
```
1. What is wrong with Harry's addWordDict method?

2. Why does this code work on words of sizes other than 12, 20, 28... but not on these sizes? (Be as detailed as possible)
Problem 3. (20 points):

We consider a 128 byte data cache that is 2-way associative and can hold 4 doubles in every cache line. A double is assumed to require 8 bytes.

For the below code we assume a cold cache. Further, we consider an array $A$ of 32 doubles that is cache-aligned (that is, $A[0]$ is loaded into the first slot of a cache line in the first set). All other variables are held in registers. The code is parameterized by positive integers $m$ and $n$ that satisfy $m \cdot n = 32$ (i.e., if you know one you know the other).

Recall that miss rate is defined as $\frac{\#\text{misses}}{\#\text{accesses}}$.

```c
float A[32], t = 0;
for(int i = 0; i < m; i++)
    for(int j = 0; j < n; j++)
        t += A[j*m + i];
```

Answer the following:

1. How many doubles can the cache hold?
2. How many sets does the cache have?
3. For $m = 1$:
   (a) Determine the miss rate.
   (b) What kind of misses occur?
   (c) Does the code have temporal locality with respect to accesses of $A$ and this cache?
4. For $m = 2$:
   
   (a) Determine the miss rate.

   (b) What kind of misses occur?

5. For $m = 16$:

   (a) Determine the miss rate.

   (b) What kind of misses occur?

   (c) Does the code have spatial locality with respect to accesses of $A$ and this cache?
Problem 10. (10 points):

Concurrency, races, and synchronization. Consider a simple concurrent program with the following specification: The main thread creates two peer threads, passing each peer thread a unique integer thread ID (either 0 or 1), and then waits for each thread to terminate. Each peer thread prints its thread ID and then terminates.

Each of the following programs attempts to implement this specification. However, some are incorrect because they contain a race on the value of myid that makes it possible for one or more peer threads to print an incorrect thread ID. Except for the race, each program is otherwise correct.

You are to indicate whether or not each of the following programs contains such a race on the value of myid. You will be graded on each subproblem as follows:

- If you circle no answer, you get 0 points.
- If you circle the right answer, you get 2 points.
- If you circle the wrong answer, you get −1 points (so don’t just guess wildly).

A. Does the following program contain a race on the value of myid?  Yes  No

```c
void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    Free(vargp);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i, *ptr;

    for (i = 0; i < 2; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], 0, foo, ptr);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```
B. Does the following program contain a race on the value of myid?  

Yes  No

```c
void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid[i], NULL, foo, &i);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
}
```

C. Does the following program contain a race on the value of myid?  

Yes  No

```c
void *foo(void *vargp) {
    int myid;
    myid = (int)vargp;
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid[i], 0, foo, i);
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
D. Does the following program contain a race on the value of `myid`?  
Yes  No

```c
sem_t s; /* semaphore s */

void *foo(void *vargp) {
    int myid;
    P(&s);
    myid = *((int *)vargp);
    V(&s);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    sem_init(&s, 0, 1); /* S=1 INITIALLY */

    for (i = 0; i < 2; i++) {
        Pthread_create(&tid[i], 0, foo, &i);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

E. Does the following program contain a race on the value of `myid`?  
Yes  No

```c
sem_t s; /* semaphore s */

void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    V(&s);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    sem_init(&s, 0, 0); /* S=0 INITIALLY */

    for (i = 0; i < 2; i++) {
        Pthread_create(&tid[i], 0, foo, &i);
        P(&s);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
Problem 10. (6 points):

Signals. Consider the following two different snippets of C code. Assume all functions return without error, no signals are sent from other processes, and printf is atomic.

**Code Snippet 1:**

```c
int main() {
    int pid = fork();
    if(pid > 0){
        kill(pid, SIGKILL);
        printf("a");
    }else{
        /* getpid() returns the pid of the parent process */
        kill(getppid(), SIGKILL);
        printf("b");
    }
}
```

**Code Snippet 2:**

```c
int a = 1;

void handler(int sig){
    a = 0;
}

void emptyhandler(int sig){
}

int main() {
    signal(SIGINT, handler);
    signal(SIGCONT, emptyhandler);

    int pid = fork();
    if(pid == 0){
        while(a == 1)
            pause();
        printf("a");
    }else{
        kill(pid, SIGCONT);
        printf("b");
        kill(pid, SIGINT);
        printf("c");
    }
}
```

For each code snippet write a Y next to an outcome if it could occur, otherwise write N.

<table>
<thead>
<tr>
<th>Snippet 1 Outcome</th>
<th>Possible? (Y/N)</th>
<th>Snippet 2 Outcome</th>
<th>Possible? (Y/N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing is printed.</td>
<td></td>
<td>Nothing is printed.</td>
<td></td>
</tr>
<tr>
<td>“a” is printed.</td>
<td></td>
<td>“ba” is printed.</td>
<td></td>
</tr>
<tr>
<td>“b” is printed.</td>
<td></td>
<td>“abc” is printed.</td>
<td></td>
</tr>
<tr>
<td>“ab” is printed.</td>
<td></td>
<td>“bac” is printed.</td>
<td></td>
</tr>
<tr>
<td>“ba” is printed.</td>
<td></td>
<td>“bca” is printed.</td>
<td></td>
</tr>
<tr>
<td>A process does not terminate.</td>
<td></td>
<td>A process does not terminate.</td>
<td></td>
</tr>
</tbody>
</table>
Problem 6. (10 points):

File I/O

The following problems refer to a file called numbers.txt, with contents the ASCII string 0123456789. You may assume calls to read() are atomic with respect to each other. The following file, read_and_print_one.h, is compiled with each of the following code files.

```c
#ifndef READ_AND_PRINT_ONE
#define READ_AND_PRINT_ONE
#include <stdio.h>
#include <unistd.h>

static inline void read_and_print_one(int fd) {
    char c;
    read(fd, &c, 1);
    printf("%c", c); fflush(stdout);
}
#endif
```

A. Consider the following code:

```c
#include "read_and_print_one.h"
#include <stdlib.h>
#include <fcntl.h>

int main() {
    int file1 = open("numbers.txt", O_RDONLY);
    int file2;
    int file3 = open("numbers.txt", O_RDONLY);
    file2 = dup2(file3, file2);

    read_and_print_one(file1);
    read_and_print_one(file2);
    read_and_print_one(file3);
    read_and_print_one(file2);
    read_and_print_one(file1);
    read_and_print_one(file3);

    return 0;
}
```
List all possible outputs of the above code.
B. Consider the following code:

```c
#include "read_and_print_one.h"
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    int file1;
    int file2;
    int file3;
    int pid;

    file1 = open("numbers.txt", O_RDONLY);
    file3 = open("numbers.txt", O_RDONLY);

    file2 = dup2(file3, file2);
    read_and_print_one(file1);
    read_and_print_one(file2);

    pid = fork();
    if (!pid) {
        read_and_print_one(file3);
        close(file3);
        file3 = open("numbers.txt", O_RDONLY);
        read_and_print_one(file3);
    } else {
        wait(NULL);
        read_and_print_one(file3);
        read_and_print_one(file2);
        read_and_print_one(file1);
    }

    read_and_print_one(file3);

    return 0;
}
```

List all possible outputs of the above code.