Andrew login ID:______________________________________

Full Name:______________________________________

Recitation Section:______________________________________

CS 15-213, Spring 2008
Final Exam

Tue. May 6, 2008

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–H) on the front.
- 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 72 points.
- This exam is OPEN BOOK. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- Good luck!

1 (12):	
2 (9):	
3 (10):	
4 (7):	
5 (8):	
6 (6):	
7 (6):	
8 (8):	
9 (6):	
TOTAL (72):	

Problem 1. (12 points):

The following problem concerns virtual memory and the way virtual addresses are translated into physical addresses. Below are the specifications of the system on which the translation occurs.

- The system is a 16-bit machine - words are 2 bytes.
- Memory is byte addressable.
- The maximum size of a virtual address space is 64KB.
- The system is configured with 16KB of physical memory.
- The page size is 64 bytes.
- The system uses a two-level page tables. Tables at both levels are 64 bytes (1 page) and entries in both tables are 2 bytes as shown below.

In this problem, you are given parts of a memory dump of this system running 2 processes. In each part of this question, one of the processes will issue a single memory operation (read or write of one byte) to a single virtual address (as indicated in each part). Your job is to figure out which physical addresses are accessed by the process if any, or determine if an error is encountered.

Entries in the first and second level tables have in their low-order bits flags denoting various access permissions.

15		2	1	0
Page Table Base Address				P

Page Directory Entry

15		2	1	0
Page Address		U	W	P

Page Table Entry

- P = 1 ⇒ Present
- W = 1 ⇒ Writable (applies both in kernel and user mode)
- U = 1 ⇒ User-mode

The contents of relevant sections of memory is shown on the next page. All numbers are given in **hexadecimal**.

Address	**Contents**
0118	2381
0130	2101
0160	2281
018E	1581
019C	1201
01B8	1A01
120A	2701
1214	27C1
1228	2741
158A	25C1
1594	2541
15A8	2501
1A0A	2041
1A14	20C1
1A28	2081
2106	3FC7
210C	3A47
2118	3587
2286	3107
228C	3447
2298	3007
2386	33C7
238C	3887
2398	3247

For the purposes of this problem, omitted entries have contents = 0.

Process 1 is a process in **user** mode (e.g. executing part of `main()`) and has page directory base address `0x0100`.
Process 2 is a process in **kernel** mode (e.g. executing a `read()` system call) and has page directory base address `0x0180`.

For each of the following memory accesses, first calculate and fill in the address of the page directory entry and the page table entry. Then, if the lookup is successful, give the physical address accessed. Otherwise, circle the reason for the failure and give the address of the table entry causing the failure. You may use the 16-bit breakdown table if you wish, but you are not required to fill it in.

1. Process 1 writes to `0xC1B2`.
   Scratch space:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(a) Address of PDE: 0x

(b) Address of PTE: 0x

(c) The result of the address translation is (write NONE if the translation does not result in a valid address): 0x

(d) The result of the access is (circle EXACTLY one):

success / page not present / page not writable / illegal non-supervisor access

2. Process 2 writes to `0x728F`. Scratch space:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(a) Address of PDE: 0x

(b) Address of PTE: 0x

(c) The result of the address translation is (write NONE if the translation does not result in a valid address): 0x

(d) The result of the access is (circle EXACTLY one):

success / page not present / page not writable / illegal non-supervisor access

```
If it's there and you can see it - it's real.  If it's not there and you
can see it - it's virtual.  If it's there and you can't see it - it's
transparent.  If it's not there and you can't see it - you erased it!
- IBM poster explaining virtual memory, 1978.
```

Problem 2. (9 points):

Consider a 12-bit IEEE floating-point representation with:

- 1 sign bit
- 4 exponent bits (therefore the bias $B = 2^{4-1} - 1 = 7$)
- 7 mantissa bits

Fill in **all** the blanks in the following table. In the process of converting some numbers to their bit representations, you might have to round up or down. If you do, put the rounded value in the "Rounded Value" column. If you didn't have to round, put a line through that row's "Rounded Value" cell. You should use "round to even" when rounding is needed.

Number	Bit representation	Rounded Value
32.125		
$-\frac{71}{2048}$		
-255.25		
	`0 1111 0000000`	———-
	`0 0000 0110000`	———-
	`0 1001 1111111`	———-

Problem 3. (10 points):

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

```
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:

```
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.

```
int foo( __________________ t ,  unsigned int * v) {

  if ( t == _________ )
      return 0;

  if( ______________________________ ) {

    return _________________________________________________;

  }


  return ( ____________________________________________?

           ____________________________________________ :

           ____________________________________________ );
}
```

Problem 4. (7 points):

Consider the following C code and disassembly of function foo.

```
int main()
{
  char *src = "some string";
  char dest[20];

  foo(44, src, dest);

  return 0;
}

void foo(int arg1, char *arg2, char*arg3)
{
  while(*arg2)
    *(arg3++) = *(arg2++) + arg1;
}
```

```
0x00001fc5 <foo+0>: push   %ebp
0x00001fc6 <foo+1>: mov    %esp,%ebp
0x00001fc8 <foo+3>: sub    $0x8,%esp
0x00001fcb <foo+6>: jmp    0x1fe7 <foo+34>
0x00001fcd <foo+8>: mov    0xc(%ebp),%eax
0x00001fd0 <foo+11>: movzbl (%eax),%edx
0x00001fd3 <foo+14>: mov    0x8(%ebp),%eax
0x00001fd6 <foo+17>: add    %eax,%edx
0x00001fd8 <foo+19>: mov    0x10(%ebp),%eax
0x00001fdb <foo+22>: mov    %dl,(%eax)
0x00001fdd <foo+24>: lea    0xc(%ebp),%eax
0x00001fe0 <foo+27>: incl   (%eax)
0x00001fe2 <foo+29>: lea    0x10(%ebp),%eax
0x00001fe5 <foo+32>: incl   (%eax)
0x00001fe7 <foo+34>: mov    0xc(%ebp),%eax
0x00001fea <foo+37>: movzbl (%eax),%eax
0x00001fed <foo+40>: test   %al,%al
0x00001fef <foo+42>: jne    0x1fcd <foo+8>
0x00001ff1 <foo+44>: leave
0x00001ff2 <foo+45>: ret
```

While debugging the above code, you open a GDB session and examine the stack at the entry point to foo.

```
(gdb) break foo
Breakpoint 1 at 0x1fcb

(gdb) run
Breakpoint 1, 0x00001fcb in foo ()

(gdb) x/40w $esp
0xbffff8f0: 0x00000000 0x00000009 0xbffff938 0x00001fba
0xbffff900: 0x0000002c 0x00001ff3 0xbffff918 0x8fe005bc
0xbffff910: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff920: 0x00000000 0x00000000 0x8fe0154b 0x00001ff3
0xbffff930: 0x00000000 0xbffff9cc 0xbffff95c 0x00001f5e
0xbffff940: 0x00000001 0xbffff964 0xbffff96c 0xbffff9cc
0xbffff950: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff960: 0x00000001 0xbffff9f4 0x00000000 0xbffffa11
0xbffff970: 0xbffffa4a 0xbffffa66 0xbffffa77 0xbffffa87
0xbffff980: 0xbffffac1 0xbffffaf6 0xbffffb0f 0xbffffb3b
```

Using the above information, please fill in the addresses for the following objects. Objects that correspond to variables are written in bold. Do not write the **values** of any of the items in the table, write only their **addresses**.

Object	**Address**
src[0]	
dest[0]	
arg1	
arg2	
arg3	
caller's return address	
caller's saved base pointer	

Problem 5. (8 points):

The problem requires understanding how C code accessing structures, unions, and arrays is compiled. Assume the x86-64 conventions for data sizes and alignments.

```
#include "def.h"

typedef struct {
  int x[A][B];       /* Unknown constant A and B */
  double y;
} str1;

typedef struct{
  str1  data[B];    /* Unknown constant  B */
  int idx;
} str2;

typedef union{
  float t ;
  str2 S[3];
  str1 V;
} uni;

void setVal(str2  *p, double val)  {
  int i = p->idx;
  p->data[ i ].y = val;
}
```

You do not have a copy of the file def.h, in which constants A and B are defined, but you have the following x86-64 assembly code for the function setVal:

```
setVal:
        # rdi = p, rsi = val
        movslq  1728(%rdi),%rax
        leaq    (%rax,%rax,2), %rax
        salq    $6, %rax
        movq   %rsi, 184(%rax,%rdi)
        ret
```

Based on this code, determine the values of the two constants and the size of the union:

A = ________________

B = ________________

Size of uni = _______________

Problem 6. (6 points):

The 15-213 `fish machines` contain Intel Xeon `Nocona` processors. The L1 data cache organization is as follows.

- 16 kilobyte total size
- 4-way associative
- 64-byte line size
- write-through

Consider the function `vsum()` defined below, shown in C and in x86-64 assembly language. Assume that the array `a` begins at address 0x0000000001000000 and the array `b` begins right after `a`. Assume for each part of the question that the cache is "cold" (empty).

```
#define X 128
double a[X], b[X];

double vsum(double *v1, double *v2, int n)
{
  double s = 0.0;
  int i;

  for (i = 0; i < n; ++i)
    s += v1[i] + v2[i];
  return (s);
}
```

```
.globl vsum
vsum: # %rdi=v1, %rsi=v2, %edx=n
    testl   %edx, %edx
    xorpd   %xmm1, %xmm1
    jle     .L4
    xorpd   %xmm1, %xmm1
    xorl    %ecx, %ecx
    xorl    %eax, %eax
.L5:
    movsd   (%rdi,%rax,8), %xmm0
    addl    $1, %ecx
    addsd   (%rsi,%rax,8), %xmm0
    addq    $1, %rax
    cmpl    %edx, %ecx
    addsd   %xmm0, %xmm1
    jne     .L5
.L4:
    movapd    %xmm1, %xmm0
    ret
```

1. How many sets does the L1 cache contain?

`No of sets:   ___________`

2. What is the miss rate in the L1 cache if `vsum()` is invoked as `vsum(a, b, X)`?

`Miss rate:   ___________`

3. What is the miss rate in the L1 cache if `vsum()` is invoked as `vsum(a, a, X)`?

`Miss rate:   ___________`

Problem 7. (6 points):

Consider the following code:

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

int main() {
  char c;
  int file1 = open("buffer.txt", O_RDONLY);
  int file2;

  read(file1, &c, 1);
  file2 = dup(file1);
  read(file1, &c, 1);
  read(file2, &c, 1);
  printf("1 = %c\n", c);

  int pid = fork();
  if (pid == 0) {
    close(file1);
    file1 = open("buffer.txt", O_RDONLY);

    read(file1, &c, 1);
    printf("2 = %c\n", c);
    read(file2, &c, 1);
    printf("3 = %c\n", c);

    exit(0);
  } else {
    waitpid(pid, NULL, 0);

    printf("4 = %c\n", c);

    close(file2);
    dup2(file1, file2);

    read(file1, &c, 1);
    printf("5 = %c\n", c);
    read(file2, &c, 1);
    printf("6 = %c\n", c);
  }

  return 0;
}
```

Assume that the disk file `buffer.txt` contains the string of bytes `source`. Also assume that all system calls succeed. What will be output when this code is compiled and run? You may not need all the lines in the table given below.

Output Line Number	**Output**
1st line of output	
2nd line of output	
3rd line of output	
4th line of output	
5th line of output	
6th line of output	
7th line of output	
8th line of output	
9th line of output	

Problem 8. (8 points):

This problem tests your understanding of pointer arithmetic, pointer dereferencing, and malloc implementation.

Harry Q. Bovik has decided to exercise his creativity and has created the most exotic dynamic memory allocator that the 213 staff has ever seen. The following is a description of Harry's block structure:

KEY	PAYLOAD	FTR

- KEY - Key of the block (4 bytes).
- PAYLOAD - Payload of the block (arbitrary size).
- FTR - Footer of the block (4 bytes).

Harry has decided to store a key in the beginning of each block instead of a header; Harry has a secret way of computing the size of the block's **payload** from the key. The size of the KEY field is 4 bytes.

Harry has also decided to store the size of a block's payload in the footer of the block. Since there is an 8-byte alignment requirement, the least significant of the 3 unused bits is used to indicate whether the block is free (0) or allocated (1).

Note that Harry is working on a 32-bit machine. You can assume the following:

- `sizeof(int) == 4 bytes`,
- `sizeof(char) == 1 byte`,
- `sizeof(short) == 2 bytes`,
- `sizeof(long) == 4 bytes`,
- and the size of any pointer (e.g., `char *`) is 4 bytes.

Your task is to help Harry get the correct key (using the function `get_key()`), by indicating which of the following implementations of the `GET_KEY` macro are correct. For each of the proposed solutions listed below, fill in the blank with either **Yes** for correct or **No** for incorrect.

```
/* get_key returns the key of a block.
    bp is pointing to the first byte of
    a block returned from Harry's malloc().
*/

#define GET_KEY(p)       ??

int get_key(void *bp) {
    return (int)(GET_KEY(bp));
}


/* A. */
#define GET_KEY(p)  (*(char *)((int *)(p) - 1))    ________

/* B. */
#define GET_KEY(p)  (*(int *)((short **)(p) - 2))   ________

/* C. */
#define GET_KEY(p)  (*(long *)((char *)(p) - 4))    ________

/* D. */
#define GET_KEY(p)  (*(long *)((long **)(p) - 1))   ________

/* E. */
#define GET_KEY(p)  (*(int *)((long)(p) - 4))       ________

/* F. */
#define GET_KEY(p)  (*(int *)((char)(p) - 4))      ________

/* G. */
#define GET_KEY(p)  (*(int *)((int **)(p) - 1))     ________

/* H. */
#define GET_KEY(p)  (*(short *)((short *)(p) - 2))   ________
```

Consider the code written by Harry Q. Bovik for the following problem.

```
pthread_mutex_t *count_l, *l_count;
int ref_count, tid_1, tid_2, tid_3, tid_4;

void *thread1(void *vargp) {
    tid_2 = pthread_self();
    pthread_mutex_lock(count_l);
    ref_count++;
    pthread_mutex_unlock(count_l);
    return(0);
}
void *thread2(void *vargp) {
    tid_1 = pthread_self();
    pthread_mutex_lock(count_l);
    pthread_kill(pthread_self(), SIGKILL);
    ref_count++;
    pthread_mutex_unlock(count_l);
    return(0);
}
void *thread3(void *vargp) {
    tid_3 = pthread_self();
    pthread_mutex_lock(count_l);
    pthread_mutex_lock(l_count);
    ref_count++;
    pthread_mutex_unlock(l_count);
    pthread_mutex_unlock(count_l);
    return(0);
}
void *thread4(void *vargp) {
    tid_4 = pthread_self();
    pthread_mutex_lock(l_count);
    pthread_mutex_lock(count_l);
    ref_count--;
    pthread_mutex_unlock(l_count);
    pthread_mutex_unlock(count_l);
    return(0);
}
void func1(void) {
    pthread_t t1,t2;
    pthread_create(&t1,NULL, thread1, NULL );
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t2, NULL);
    pthread_join(t1, NULL);
    exit(0);
}
void func2(void) {
    pthread_t t3, t4;
    pthread_create(&t4,NULL, thread4, NULL );
    pthread_create(&t3, NULL, thread3, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    exit(0);
}
```

Problem 9. (6 points):

Please assume that all necessary header files are included in the code and all system calls and accessory functions always succeed. You may assume that the locks have been initialized correctly in `main()`, which we do not show.

1. Bovik comes to you and complains that `func1()` seems to misbehave. Is he lying or is there something wrong with his code? Defend your answer in 1-3 sentences.

2. Bovik is complaining about `func2()` as well. He insists to your boss that this program sometimes hangs and your boss would like your opinion. Is Bovik lying again or is there a problem? Defend your answer in 1-3 sentences.