Andrew login ID: _____

Full Name: _____

CS 15-213, Spring 2004

Exam 2

April 8, 2004

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID 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 79 points and a total of **17** pages.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. You may not use a calculator, laptop or other wireless device. Good luck!

1 (16):	
2 (10):	
3 (8):	
4 (12):	
5 (9):	
6 (10):	
7 (14):	
TOTAL (79):	

Problem 1. (16 points):

The following problem concerns basic cache lookups.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are 13 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 32 total lines.

In the following tables, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

4-way Set Associative Cache									
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	84	1	ED 32 0A A2	9E	0	BF 80 1D FC	10	0	EF 09 86 2A	E8	0	25 44 6F 1A
1	18	1	03 3E CD 38	E4	0	16 7B ED 5A	02	0	8E 4C DF 18	E4	1	FB B7 12 02
2	84	0	54 9E 1E FA	84	1	DC 81 B2 14	48	0	B6 1F 7B 44	89	1	10 F5 B8 2E
3	92	0	2F 7E 3D A8	9F	0	27 95 A4 74	57	1	07 11 FF D8	93	1	C7 B7 AF C2
4	84	1	32 21 1C 2C	FA	1	22 C2 DC 34	73	0	BA DD 37 D8	28	1	E7 A2 39 BA
5	A7	1	A9 76 2B EE	73	0	BC 91 D5 92	28	1	80 BA 9B F6	6B	0	48 16 81 0A
6	8B	1	5D 4D F7 DA	29	1	69 C2 8C 74	B5	1	A8 CE 7F DA	BF	0	FA 93 EB 48
7	84	1	04 2A 32 6A	96	0	B1 86 56 0E	CC	0	96 30 47 F2	91	1	F8 1D 42 30

Part 1

The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

O The block **o**ffset within the cache line
I The cache **i**ndex
T The cache **t**ag

12	11	10	9	8	7	6	5	4	3	2	1	0
T	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**I**	**I**	**I**	**O**	**O**

Part 2

For the given physical address, indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs. If there is a cache miss, enter "-" for "Cache Byte returned".

Physical address: `0x0D74`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0
0	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**0**

Physical memory reference

Parameter	Value
Cache Offset (CO)	0x **00**
Cache Index (CI)	0x **05**
Cache Tag (CT)	0x **6B**
Cache Hit? (Y/N)	**N (dirty)**
Cache Byte returned	0x **-**

Physical address: `0x0AEE`

Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0
0	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0**

Physical memory reference

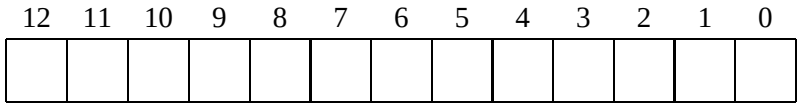
Parameter	Value
Cache Offset (CO)	0x **02**
Cache Index (CI)	0x **03**
Cache Tag (CT)	0x **57**
Cache Hit? (Y/N)	**Y**
Cache Byte returned	0x **FF**

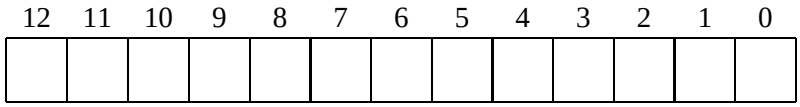
Part 3

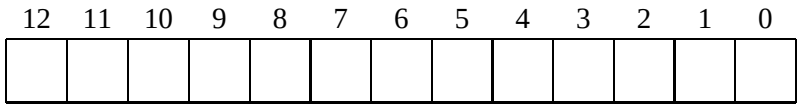
For the given contents of the cache, list all of the hex physical memory addresses that will hit in Set 7. To save space, you should express contiguous addresses as a range. For example, you would write the four addresses `0x1314`, `0x1315`, `0x1316`, `0x1317` as `0x1314--0x1317`.

Answer: **0x109C – 0x109F, 0x123C – 0x123F**

The following templates are provided as scratch space:





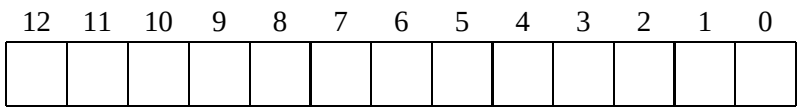


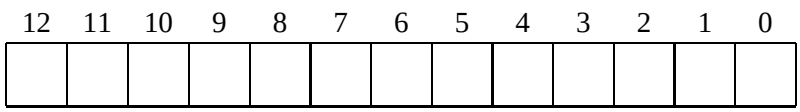
Part 4

For the given contents of the cache, what is the probability (expressed as a percentage) of a cache hit when the physical memory address ranges between `0x1080` - `0x109F`. Assume that all addresses are equally likely to be referenced.

Probability = **50** %

The following templates are provided as scratch space:





Problem 2. (10 points):

This problem requires you to analyze the behavior of the program below which transposes the $N \times N$ int-matrix A.For this problem, $N = 4$. For this problem, you should assume that the loop variables **`x`** and **`y`** are kept in registers and do not cause memory accesses. Likewise, the temporary variable **`t`** which is used to exchange two array elements is also stored in a register and does not cause any load/store from/to the memory system or the caches.

```
 1    #define N 4  /* Array size */
 2  ...
 3    int A[N][N] = {0};
 4  ...
 5   {   int x, y;
 6  ...
 7       for (y = 0; y < N; y++) {
 8         for (x = y + 1; x < N; x++) {
 9           int t;
10           t = A[y][x];
11           A[y][x] = A[x][y];
12           A[x][y] = t;
13         }
14      }
15  ...
16   }
```

You are supposed to analyze how this program will interact with a simple cache. The cache line size is **`2*sizeof(int)`**. The cache is cold when the program starts. Further more, the array A is aligned so that the first two elements are stored in the same cache line.

You are supposed to fill out the tables below. For each load (ld) and store (st) operation to an element of the array A, you should indicate if this operation misses (**M**) or hits (**H**) in the cache. Note that this program does not touch the diagonal array elements.

1. The cache is direct mapped and has two (2) lines:

	ld=**M** st=**M**	ld=**M** st=**H**	ld=**H** st=**H**
ld=**M** st=**M**		ld=**M** st=**H**	ld=**H** st=**H**
ld=**M** st=**H**	ld=**M** st=**H**		ld=**M** st=**M**
ld=**M** st=**H**	ld=**M** st=**H**	ld=**M** st=**M**	

2. The cache is 2-way set-associative and has one set. It uses the least recently used (LRU) replacement policy:

	ld=**M** st=**H**	ld=**M** st=**H**	ld=**H** st=**H**
ld=**M** st=**H**		ld=**M** st=**H**	ld=**H** st=**H**
ld=**M** st=**H**	ld=**M** st=**H**		ld=**M** st=**H**
ld=**M** st=**H**	ld=**M** st=**H**	ld=**M** st=**H**	

Problem 3. (8 points):

Consider the following C programs. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Assume that `printf` is unbuffered and that each call to `printf` executes atomically.

```
/* ecf.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAXLEN 32

int main(int argc, char *argv[])
{
   int i, status, limit;
   char num[MAXLEN+1];
   pid_t pid;

   limit = atoi(argv[1]);

   for(i = 0; i < limit; i++) {
      if((pid = fork()) == 0) {
         snprintf(num, MAXLEN, "%d", i+10);
         execl("./kid", "./kid", num, NULL);
      }
      if(i == (limit - 2)) {
         waitpid(pid, &status, 0);
         printf("%d\n", status);
      }
   }

   return 0;
}

-----------------------------------------
/* kid.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("%d\n", atoi(argv[1]));
   return 0;
}
```

Assume that ecf.c is compiled into ecf and kid.c is compiled into kid in the same directory. List **all** possible outputs (note the newlines) of the following command:

```
[user@host directory]$ ./ecf 3
```

```
10   11   11   11
11   10    0    0
 0    0   10   12
12   12   12   10
```

Problem 4. (12 points):

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 18 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 1024 bytes.
- The TLB is 4-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal**.

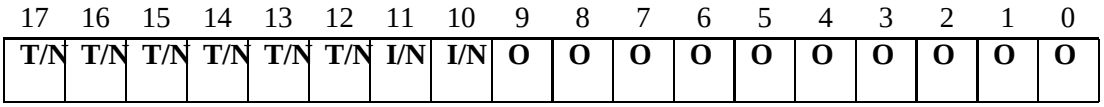
TLB			
Index	Tag	PPN	Valid
0	05	13	0
	1E	14	1
	10	0F	1
	0F	1E	0
1	1F	01	1
	11	1F	0
	03	2B	1
	1D	23	0
2	06	08	1
	0F	19	1
	0A	09	1
	1F	20	1
3	03	13	0
	13	12	1
	0C	0B	0
	2E	24	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	2D	0
05	13	1	15	1B	0
06	0F	1	16	0C	0
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	01	0	1A	0C	1
0B	16	1	1B	12	1
0C	01	1	1C	1E	0
0D	15	1	1D	0E	1
0E	0C	0	1E	27	1
0F	14	0	1F	18	1

Part 1

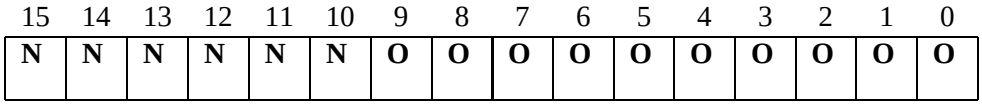
1. The diagram below shows the format of a virtual address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

   O The virtual page **o**ffset
   N The virtual page **n**umber
   I The TLB **i**ndex
   T The TLB **t**ag



2. The diagram below shows the format of a physical address. Please indicate the following fields by labeling the diagram: (If a field does not exist, do not draw it on the diagram.)

   O The physical page **o**ffset
   N The physical page **n**umber



Part 2

For the given virtual addresses, please indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs. If there is a page fault, enter "-" for "PPN" and leave the physical address blank.

Virtual address: `0x0718F`

1. Virtual address (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x**1C**	TLB Hit? (Y/N)	**No**
TLB Index	0x**0**	Page Fault? (Y/N)	**Yes**
TLB Tag	0x**07**	PPN	0x**–**

3. Physical address(one bit per box)

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

Virtual address: `0x04AA4`

1. Virtual address (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x**12**	TLB Hit? (Y/N)	**No**
TLB Index	0x**2**	Page Fault? (Y/N)	**No**
TLB Tag	0x**04**	PPN	0x**0E**

3. Physical address(one bit per box)

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

Problem 5. (9 points):

This problem tests your understanding of Unix signals.
Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Assume that printf is unbuffered, and that latency in receiving signals is negligable.

```
/* signal.c */

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

sigset_t s1;
sigset_t s2;
int i;

void handler1(int sig);
void handler2(int sig);
void func0();
void func1();
void func2();

int main(int argc, char** argv)
{
    int n = atoi(argv[1]);

    signal(SIGINT, handler1);
    signal(SIGTSTP, handler2);

    sigemptyset(&s1);
    sigaddset(&s1, SIGINT);
    sigemptyset(&s2);
    sigaddset(&s2, SIGTSTP);

    if(n == 0)
        func0();
    else if(n == 1)
        func1();
    else if(n == 2)
        func2();

    return 0;
}
```

```
/* signal.c (continued) */

void handler1(int sig)
{
    int j;
    printf("hello\n");
    for(j = 0; j < 3; j++)
        kill(0, SIGTSTP);
}

void handler2(int sig)
{
    printf("greetings\n");
}

void func0()
{
    sigprocmask(SIG_BLOCK, &s1, NULL);
    for(i = 0; i < 5; i++)
        kill(0, SIGINT);
    sigprocmask(SIG_UNBLOCK, &s1, NULL);
}

void func1()
{
    sigprocmask(SIG_BLOCK, &s2, NULL);
    for(i = 0; i < 5; i++)
        kill(0, SIGINT);
    sigprocmask(SIG_UNBLOCK, &s2, NULL);
}

void func2()
{
    for(i = 0; i < 5; i++)
    {
        sigprocmask(SIG_BLOCK, &s1, NULL);
        kill(0, SIGINT);
        sigprocmask(SIG_BLOCK, &s2, NULL);
        sigprocmask(SIG_UNBLOCK, &s1, NULL);
        sigprocmask(SIG_UNBLOCK, &s2, NULL);
    }
}
```

For each commandline listed below, write the output which would result:

```
unix> ./signal 0
```

```


hello
greetings
greetings
greetings
```

```
unix> ./signal 1
```

```


hello
hello
hello
hello
hello
greetings
```

```
unix> ./signal 2
```

```


hello
greetings
hello
greetings
hello
greetings
hello
greetings
hello
greetings
```

Problem 6. (10 points):

Suppose the file `foo.txt` contains the text "`123456`", `bar.txt` contains the text "`abcdef`", and `baz.txt` does not yet exist. Examine the following C code, and answer the questions below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main() {
    int fd1, fd2, fd3, fd4, fd5, fd6;
    int status;
    pid_t pid;
    char c;

    /* foo.txt has "123456" */
    fd1 = open("foo.txt", O_RDONLY, 0);
    fd2 = open("foo.txt", O_RDONLY, 0);

    /* bar.txt has "abcdef" */
    fd3 = open("bar.txt", O_RDWR, 0);
    fd4 = open("bar.txt", O_RDWR, 0);

    /* baz.txt doesn't exist initially */
    fd5 = open("baz.txt", O_WRONLY | O_CREAT | O_TRUNC,
                          S_IRUSR | S_IWUSR); /* r/w */

    fd6 = dup(STDOUT_FILENO);
    dup2(fd5, STDOUT_FILENO);

    if ((pid = fork()) == 0) {
        dup2(fd3, fd2);

        read(fd3, &c, 1); printf("%c", c);
        write(fd4, "!@#$%^", 6);
        read(fd3, &c, 1); printf("%c", c);
        read(fd1, &c, 1); printf("%c", c);
        read(fd2, &c, 1); printf("%c\n", c);
        exit(0);
    }

    wait(NULL);
    read(fd1, &c, 1); printf("%c", c);
    fflush(stdout);

    dup2(fd6, STDOUT_FILENO);
    printf("done.\n");
    return 0;
}
```

A. What will the contents of `baz.txt` be after the program completes?

```
a@1#
2
```

B. What will be printed on `stdout`?

```
done.
```

Problem 7. (14 points):

Answer the following short answer questions with **no more** than 2 sentences.

A. What characteristics of a disk subsystem determine the time it takes to access a sector on disk?

Seek time, rotation speed, transfer time

B. What is the CPU-Memory gap? And, is it getting bigger or smaller over time?

The difference in time between the CPU clock rate and the dram access time. It is getting bigger.

C. Does the declaration "`static int x;`" generate an entry in the .o file? If so, what kind of entry would it be?

yes. A relocatable entry in the data segment.

D. List up to two positive reasons and two negative reasons for using a mark and sweep garbage collection system instead of an explicit memory allocator in a real-time system?

Postive: fast code development, fewer memory errors. Negatives: unpredictable pauses in system execution.

E. The following two files are linked together and the program is run.

```
main()
{
   int x;

   x  = 40;
   foo();
   printf("A: %d\n", x);
}

extern int x;

bar()
{
   x = 2 * x;
   foobar();
}

int x = 2;
```

```
static int x = 5;

foo()
{
   x = x + 10;
   bar();
}

foobar()
{
   extern int x;

   x+=3;
   printf("B: %d\n", x);
}
```

What is the output:

B: 18 A: 40

F. What is printed out in the following code:

```
#include <setjmp.h>

jmp_buf buf;
int x = 0;

main()
{
   switch (setjmp(buf))
   {
   case 0: x = 2;
           foo();
           x++;
   case 1: bar();
           x++;
   case 2: printf("A: %d\n", x);
   }
}
```

```
foo()
{
   x++;
   if (x < 10) foo();
   longjmp(buf, 1);
   x++;
   printf("B: %d\n", x);
}

bar()
{
   x++;
   longjmp(buf, 2);
   printf("C: %d\n", x);
}
```

Output:

A: 11