Andrew login ID:_____

Full Name:_____

CS 15-213, Fall 2005

Exam 2

Tuesday Nov 22, 2005

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 51 points.
- 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. No electronic devices are allowed. Good luck!

1 (04):
2 (09):
3 (10):
4 (10):
5 (10):
6 (08):
TOTAL (51):

Problem 1. (4 points):

This problem tests your understanding of how Linux represents and shares files. You are asked to show what each of the following programs prints as output:

- Assume that file `infile.txt` contains the ASCII text characters "`123456`".
- The system function `int dup(int oldfd)` is a variant of `dup2` that copies descriptor `oldfd` to the lowest-numbered unused descriptor, and then returns the index of the new descriptor. For example, suppose that the lowest-numbered unused descriptor is 5. Then `newfd = dup(3)` copies descriptor 3 to descriptor 5, returns the integer value 5, and assigns it to variable `newfd`.

A.

```
1  int main() {
2      int fd1, fd2;
3      char c;
4
5      fd1 = open("infile.txt", O_RDONLY, 0);
6      fd2 = open("infile.txt", O_RDONLY, 0);
7
8      read(fd1, &c, 1);
9      read(fd2, &c, 1);
10
11     printf("c = %c\n", c);
12     exit(0);
13 }
```

c = _____

B.

```
1  int main() {
2      int fd1, fd2;
3      char c;
4
5      fd1 = open("infile.txt", O_RDONLY, 0);
6      fd2 = dup(fd1);
7
8      read(fd1, &c, 1);
9      read(fd2, &c, 1);
10
11     printf("c = %c\n", c);
12     exit(0);
13 }
```

c = _____

Problem 2. (9 points):

This problem will test your knowledge of process control and signals. Each program below produces a single output line each time it runs. However, because of non-determinism in the scheduling of processes and signals, each run may produce different output lines. You are asked to list all possible output lines.

A. (3 pts) Assume that `main()` calls the following function `test()` exactly once.

```
void test(void)
{
  if ( fork() == 0 )
  {
      printf(‘‘0’’);
      exit(0);
  }
  printf(‘‘1’’);
}
```

List all possible output lines:

B. (3 pts) Assume that `main()` calls the following function `test()` exactly once.

```
void test(void)
{
  int status;
  int counter = 0;

  if ( fork() == 0 )
  {
    counter++;
    printf(‘‘%d’’, counter);
    exit(0);
  }
  wait(&status);

  if ( fork() == 0 )
  {
    counter++;
    printf(‘‘%d’’, counter);
    exit(0);
  }
  wait(&status);
}
```

List all possible output lines:

C. (3 pts) Assume that `main()` calls the following function `test()` exactly once. Assume that `sigusr1_handler()` is installed as the signal handler for SIGUSR1 and that `block_all_signals()` uses `sigprocmask()` to block all signals.

```
void sigusr1_handler(int n)
{
  printf(''1'');
  exit(0);
}

void test(void)
{
  int i, status;

  for (i = 0; i < 3; i++)
  {
    pid_t pid = fork();
    if ( pid == 0 )
    {
      block_all_signals();
      printf(''0'');
      exit(0);
    }
    kill( pid, SIGUSR1 );
    wait(&status);
  }
}
```

List all possible output lines:

Problem 3. (10 points):

The following problem concerns basic cache lookups.

- The memory is byte addressable.
- Memory accesses are to **1-byte words**
- Physical addresses are 13 bits wide.
- The cache is 4-way set associative, with a 4-byte block size and 8 sets.

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	F0	1	ED 32 0A A2	8A	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	0	25 44 6F 1A
1	BC	0	03 3E CD 38	A0	0	16 7B ED 5A	BC	1	8E 4C DF 18	E4	1	FB B7 12 02
2	BC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E
3	BE	0	2F 7E 3D A8	C0	1	27 95 A4 74	C4	0	07 11 6B D8	BC	0	C7 B7 AF C2
4	7E	1	32 21 1C 2C	8A	1	22 C2 DC 34	BC	1	BA DD 37 D8	DC	0	E7 A2 39 BA
5	98	0	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	BC	1	48 16 81 0A
6	38	0	5D 4D F7 DA	BC	1	69 C2 8C 74	8A	1	A8 CE 7F DA	38	1	FA 93 EB 48
7	8A	1	04 2A 32 6A	9E	0	B1 86 56 0E	CC	1	96 30 47 F2	BC	1	F8 1D 42 30

Part 1

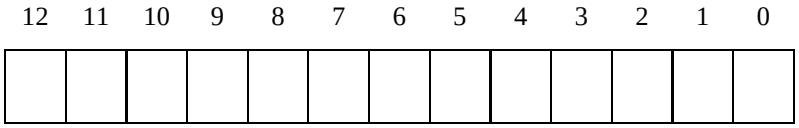
A) Warmup problem

What is the (data) size of this cache in bytes?

Answer: `C = _____ bytes`

B) 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:

CO The block offset within the cache line
CI The cache index
CT The cache tag



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".

Hint: Pay attention to those valid bits!

Physical address: `0x71A`

Physical address format (one bit per box)

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

Physical memory reference

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

Physical address: `0x16E8`

Physical address format (one bit per box)

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

Physical memory reference

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

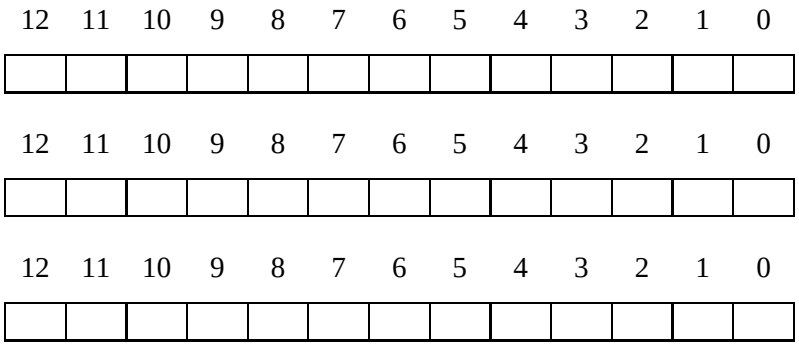
Part 3

For the given contents of the cache, list the eight hex physical memory addresses that will **hit in Set 2**.

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: _____

The following templates are provided as scratch space:



Problem 4. (10 points):

This problem requires you to analyze the cache behavior of two small segments of code that access an $N \times M$ integer matrix arr. For this problem, $N = 4$ and $M = 3$. Assume that the loop variables i and j, the accumulator variable sum, and the temporary variables $temp1$ and $temp2$ are all stored in registers.

```
#define N 4
#define M 3

int arr[N][M];

int arr_sum1()
{
  int i, j;
  int sum = 0;

  for (i=0; i<N; i+=2){
    for (j=0; j<M; j++){
      int temp1 = arr[i][j];
      int temp2 = arr[i+1][j];
      sum += (temp1 * temp2);
    }
  }

  return sum;
}
```

We are interested in how this function will interact with a simple cache. Assume that the cache is cold when the function is called and that the array has been initialized elsewhere. **The cache is direct-mapped with two sets and a block size of 8 bytes.** Assume that arr is aligned so that the first two elements are stored in the same cache block. Fill out the table below to indicate if the corresponding memory access in arr will be a hit (**H**) or a miss (**M**).

arr	Col 0	Col 1	Col 2
Row 0			
Row 1			
Row 2			
Row 3			

Now consider a slightly different function that accesses the same integer matrix arr. Again, assume **i**, **j**, **sum**, **temp1**, and **temp2** are all stored in registers.

```
#define N 4
#define M 3

int arr[N][M];

int arr_sum2()
{
  int i, j;
  int sum = 0;

  for(i=0; i<N/2; i++){
    for(j=0; j<M; j++){
      int temp1 = arr[i][j];
      int temp2 = arr[i+(N/2)][j];
      sum += (temp1 * temp2);
    }
  }

  return sum;
}
```

Once again, fill out the table below to indicate if the corresponding memory access in arr will be a hit (**H**) or a miss (**M**). Assume the cache is cold when the function is called and the array has already been initialized. **Like the previous problem, the cache is direct-mapped with two sets and a block size of 8 bytes.**

arr	Col 0	Col 1	Col 2
Row 0			
Row 1			
Row 2			
Row 3			

Problem 5. (10 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.

- Memory is byte addressable and memory accesses are to 4-byte words.
- The system is configured with 256MB of virtual memory.
- The system has only 64MB of physical memory.
- The page size is 4KB.
- The TLB is 2-way set associative with 8 total sets.

The contents of the TLB and the relevant sections of the page tables are shown below. In the following tables, **all numbers are given in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	0003	03EB	1
	1A10	0D46	0
1	0107	0AD3	1
	1D01	052F	0
2	0106	0D4E	1
	1213	01D3	0
3	0301	005C	0
	0A0B	0231	1
4	1211	0819	1
	0108	03D8	0
5	011F	0218	1
	1102	0A4A	1
6	1FE7	0263	1
	103F	0FAF	0
7	0211	030D	0
	10D2	0310	0

Page Table		
VPN	PPN	Valid
0837	1457	0
0838	0D31	0
0839	0AD3	1
0840	035A	1
0841	16F3	0
0842	0D4E	1
0843	031D	1
0844	078F	1
0845	0487	1
0846	0819	0
0847	136B	1
0848	04BA	1
0849	0D33	0
0850	0061	0
0851	0328	0
0852	0F42	0

A. Part 1 - Warmup questions

(a) How many bits are needed to represent the virtual address space?

(b) How many bits are needed to represent the physical address space?

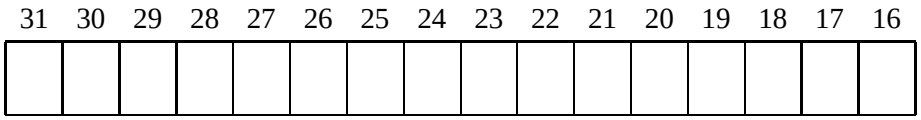
(c) What is the total number of page table entries?

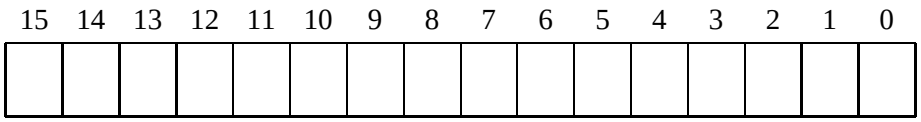
B. Part 2 - Virtual memory address translation

(a) Please step through the following address translation. You may indicate a page fault by entering a '–' for Physical Page Number and Physical Address.

Parameter	Value
Virtual Address Accessed:	0x844044
Virtual Page Number:	0x
TLB Index:	0x
TLB Tag:	0x
TLB Hit or Miss:	
Physical Page Number:	0x
Physical Address:	0x

Please use the layout below as scratch space if necessary.





(b) Please step through the following address translation. You may indicate a page fault by entering in '-' for Physical Page Number and Physical Address.

Parameter	Value
Virtual Address Accessed:	0x839108
Virtual Page Number:	0x
TLB Index:	0x
TLB Tag:	0x
TLB Hit or Miss:	
Physical Page Number:	0x
Physical Address:	0x

Please use the layout below as scratch space as necessary.

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

Performance Evaluation

The following code evaluates a polynomial of degree `degree` over `x` for a set of coefficients `coeff`.

```
/* Unroll 2x */
data_t peval2(data_t coeff[], data_t x, int degree)
{
    data_t result = 0;
    int i;

    /* Unroll by 2X */
    for (i = degree; i >= 1; i -= 2) {
        /* Version 1 Main Loop Computation */
        result = (((result * x) + coeff[i]) * x) + coeff[i-1];
    }

    /* Finish off remaining element(s) */
    for (; i >= 0; i -= 1) {
        result = result * x + coeff[i];
    }
    return result;
}
```

The code uses loop unrolling to compute two terms of the polynomial per iteration. It uses a technique known as *Horner's Rule* to reduce the number of multiplications and additions. Data type `data_t` can be defined to different types using a `typedef` declaration.

Running on an Intel Pentium 4, and with `data_t` defined to be `float`, this code achieves a CPE (Cycles Per Element) of 12.0. Considering that this machine has a latency of 7 cycles for single-precision multiplication, and 5 for single-precision addition, we can see that the CPE is dictated by these latencies, the data dependencies in the main loop (the line labeled "`Version 1 Main Loop Computation`"), and the fact that each iteration computes two "elements".

Problem 6. (8 points):

Below are four alternative versions of the main loop computation. For each version, write down the CPE that will result. The possible choices are: 6.0, 8.5, 9.5, and 12.0. You can determine the answer knowing only the operation latencies and the data dependencies that each line entails. **You can ignore other factors such as the issue time and the number of functional units.**

A. Version 2

```
result = (((result * x) * x) + (coeff[i] * x)) + coeff[i-1];
```

CPE = _____

B. Version 3

```
result = ((result * (x * x)) + (coeff[i] * x)) + coeff[i-1];
```

CPE = _____

C. Version 4

```
result = ((result * x) * x) + ((coeff[i] * x) + coeff[i-1]);
```

CPE = _____

D. Version 5

```
result = (result * (x * x)) + ((coeff[i] * x) + coeff[i-1]);
```

CPE = _____