CSE351 Autumn 2010 Final Exam (December 13, 2010)

Please read through the entire examination first! We designed this exam so that it can be completed in 100 minutes and, hopefully, this estimate will prove to be reasonable.

There are 7 problems worth a total of 100 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided. If you need more space (you shouldn't), you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. Do NOT use any other paper to hand in your answers. If you have difficulty with part of a problem, move on to the next one. They are mostly independent of each other.

The exam is CLOSED book and CLOSED notes. Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Name: ____Sample Solution______

Problem	**Max Score**	**Score**
1	12	
2	10	
3	10	
4	12	
5	24	
6	20	
7	12	
TOTAL	**100**	

1. Caches (12 points)

When images are printed on paper four different inks are used to generate colors: cyan, magenta, yellow, and black (CMYK). Software that deals with color printing needs to be able to represent images as arrays of these colors. For this problem we would like to determine the efficiency of the following code on a machine with a 2048-byte direct-mapped data cache with 32-byte blocks. We are given the following data definitions:

```
struct point_color {
  int c;
  int m;
  int y;
  int k;
};

struct point_color grid[16][16];
int i, j;
```

Assume the following:

- sizeof(int) == 4
- grid begins at memory address 0
- The cache is initially empty
- The only memory accesses are to the entries of array grid. Variables i and j are stored in registers

What is the cache hit ratio when the following code is executed?

```
for (i = 0; i < 16; i++) {
  for (j = 0; j < 16; j++) {
    grid[i][j].c = 0;
    grid[i][j].m = 0;
    grid[i][j].y = 0;
    grid[i][j].k = 0;
  }
}
```

Include enough details so we can follow your calculations. Please try to simplify your answers, but it's ok if you need to save time by leaving some of them as fractions.

Write

Your

Answer

On the

Next

Page …

1. Caches (cont.)

Write your answer here.

(This is problem 6.39 from the textbook)

Each cache block holds 2 16-byte point_color structs

The array is 16*16*16 = 4096 bytes. The cache holds half of that.

Since we are scanning in row-major order, each miss (for a grid[i][j].c int value) is followed by 7 hits (the rest of that struct and the entire following one).

The hit ratio, then, is 7/8 = 87.5%.

2. Caches and Programming (10 points)

Three programmers are debating how best to write the code to transpose a matrix in a C software package that will run on a variety of different machines. The exact details of the different computers aren't known. What is known is that all of them have caches and virtual memory. The matrix is much too large to fit entirely in the caches, but it is small enough to fit in main memory, so virtual memory paging is not an issue. The array is a normal C array of `doubles` stored in row-major order.

Alice says that this code will do the best overall job:

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
```

Bob says that this is all wrong and that the performance will be much better if it's written this way:

```
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i++) {
    temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
```

Chris says it doesn't matter. The code will perform about the same either way.

Who's right and why? Give a brief technical justification for your answer.

Chris is right – the order doesn't matter. Each iteration of either inner loop references both A[i][j] and A[j][i], so there is one group of references going through the array sequentially and another that is jumping by a complete row at a time. Both sets of nested loops will have essentially the same reference pattern and cache behavior.

3. Disk Performance (10 points)

The Round Number Disk Manufacturing Company specializes in manufacturing disks designed to produce simple calculations on CSE exams. Their CSE351 Special has the following performance characteristics:

> Rotational speed: 6000 rpm (100 revolutions per second)
> Average seek time: 5 ms.
> Number of sectors (data blocks) per track: 1000
> Sector size: 500 bytes

The disk has only two surfaces on a single platter. Tracks on both surfaces can be accessed without moving the head once it is in position after a seek.

Recall that the time needed to read data from a disk is the time needed to position the head over the first block (average rotational latency and seek time) plus the time for the data to move under the head once it is positioned.

a) What is the best expected time in milliseconds (ms) needed to read 1,000,000 bytes from the disk, assuming that the data is organized sequentially on the disk with the best possible mapping of logical blocks to disk sectors.

If the blocks are ordered sequentially, it takes 1 seek and 1 rotational latency to access the first block, followed by two complete rotations of the disk to transfer all the data. Total time is

> **5 ms seek + 5ms latency + 2 * 10ms/rotation = 30 ms.**

b) What is the expected time in milliseconds (ms) needed to read the same 1,000,000 bytes from the disk if the data blocks are scattered randomly on the disk?

We still need 20 ms. to transfer all the data, even though the data is scattered over the disk. The difference this time is that we potentially have a seek + latency delay to access each individual block. We need to read 2,000 blocks. The total time is:

> **2000 * (5 ms seek + 5 ms latency) + 20 ms transfer**
>
> **= 2000 * 10 ms + 20 ms**
>
> **= 20.02 sec.**

(Quite a bit worse than sequential transfer.)

4. Linking and Relocation (12 points)

Here is the code and relocation information for a small function named f. Lines have been numbered for reference below

```
1)     00000000 <f>:
2)        0: 55                          push   %ebp
3)        1: 89 e5                       mov    %esp,%ebp
4)        3: 83 ec 10                    sub    $0x10,%esp
5)        6: a1 0c 00 00 00              mov    0xc,%eax
6)                                 7: R_386_32 stuff
7)        b: a3 00 00 00 00              mov    %eax,0x0
8)                                 c: R_386_32 data
9)       10: c7 45 f8 fc ff ff ff        movl   $0xfffffffc,-0x8(%ebp)
10)                                13: R_386_32 stuff
11)      17: b8 11 00 00 00              mov    $0x11,%eax
12)      1c: c9                          leave
13)      1d: c3                          ret
```

When this code is loaded, the linker assigned the following absolute addresses for the external symbols in the .code and .data sections of the program, and loads the bytes of function f starting at the address given here (0x08048394):

```
f           0x08048394
stuff       0x08049634
data        0x08049648
```

The above code contains three relocated references. For each relocated reference, give the line number in the above code of the instruction that is being relocated, the runtime memory address of the relocated data, and the relocated value stored at that address.

Line #	Relocated Data Address	Relocated Data Value
5	**0x0804839b**	**0x08049640**
7	**0x080483a0**	**0x08049648**
9	**0x080483a7**	**0x08049630**

5. Virtual Memory (24 points)

Our system implements virtual memory with a translation look-aside buffer (TLB – 16 entries, 4-way set associative), page table (PT – 1024 page entries, only the first 16 shown below), and memory cache (16 entries, 4-byte lines, direct-mapped). The initial contents are shown in the tables below. The virtual address is 16 bits (4 hex digits) while the physical address is 12 bits (3 hex digits); the page size is 64 bytes.

(You can remove this page for reference while answering the rest of this question.)

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	33	1	08	–	0	06	–	0	03	11	1
3	07	–	0	03	0D	1	0A	34	1	02	–	0

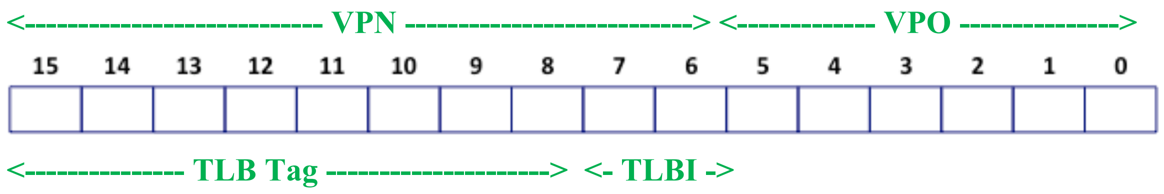
Page Table

VPN	PPN	Valid	VPN	PPN	Valid
000	28	1	008	13	1
001	–	0	009	17	1
002	09	1	00A	33	1
003	02	1	00B	–	0
004	–	0	00C	–	0
005	16	1	00D	2D	1
006	–	0	00E	11	1
007	–	0	00F	0D	1

Cache

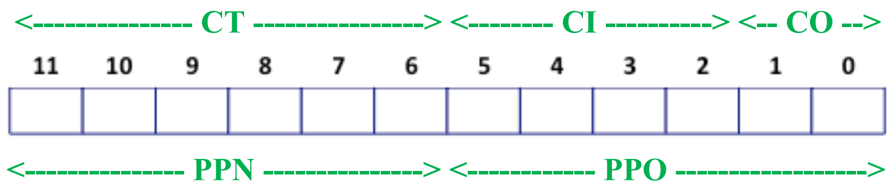
Index	Tag	Valid	B0	B1	B2	B3	Index	Tag	Valid	B0	B1	B2	B3
0	28	1	99	1F	23	11	8	11	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

a) (2 points) Outline the bits corresponding to each of the components of the virtual address, namely, the virtual page number (VPN), the virtual page offset (VPO), the TLB set index (TLBI), and the TLB tag value (TLBT).



<---------------------------- VPN ---------------------------> <------------- VPO --------------->
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<--------------- TLB Tag ---------------------> <- TLBI ->

b) (2 points) Outline the bits corresponding to each of the components of the physical address, namely, the physical page number (PPN), the physical page offset (PPO), the cache set index (CI), the cache tag value (CT), and the cache byte offset (CO).



<--------------- CT ----------------> <-------- CI ----------> <-- CO -->
11 10 9 8 7 6 5 4 3 2 1 0
<--------------- PPN ---------------> <------------ PPO ------------------>

c) (12 points) Determine the returned values for the following sequence of accesses and specify whether a TLB miss, page fault, and/or cache miss occurred. In some cases, it may not be possible to determine what value is accessed or whether there is a cache miss or not. For these cases, simply write ND (for Not Determinable). If it matters, assume the accesses occur in this sequence and each access updates the tables as needed.

Virtual Address	Physical Address	Value	TLB Miss?	Page Fault?	Cache Miss?
0x01B9	ND	ND	Y	Y	ND
0x0363	0xb63	ND	N	N	Y
0x0353	0xb53	ND	N	N	Y
0x072B	0x0AB	ND	N	N	Y

d) (5 points) One of our customers is running the following code snippet to initialize every entry in a byte array to 0x00:

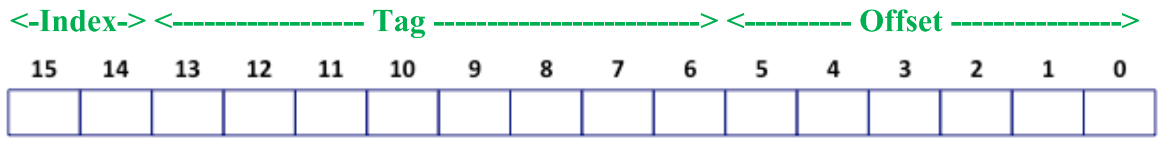
```
char array[32][32];

for (int i = 0; i < 32; i++){
        for (int j = 0; j < 32; j++){
                array[j][i] = 0;
        }
}
```

How many TLB misses occur when this code runs on our system? (You should assume this code is executed on a cold system with an initially empty cache and empty TLB.)

16 initial misses. After that, all pages in the active set are in the cache.

e) (3 points) Bernie Bitkiller, the summer intern, suggested that it would be better if we used the virtual address bits differently to access the TLB. His idea is that the TLBI (TLB Index) bits should be the high-order bits of the virtual address starting at bit 15, then the TLBT (TLB Tag) bits next, and the VPO (virtual page offset) bits in the lower-order part of the address ending with bit 0. Basically he wants to interchange the positions of the TLBI and TLBT bits.



<-Index-> <------------------ Tag ------------------------> <---------- Offset ---------------->
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Suppose we execute code that has many sequential reads from memory. How will this design perform for those programs? What will the TLB hit/miss ratios be like and why?

There is an initial cold miss, then every subsequent read in the block is a hit – only the offset bits will be changing. The TLB hit ratio will be close to 1.

(Note: this is actually a little unexpected, since using the high-order bits for the index often leads to bad behaviors in caches. In this particular case it turns out not to be a problem.)

6. Memory Management (20 points)

One of our customers has an unusual request for an addition to our memory manager code. They would like to have the following function added:

```
void * max_alloc(size_t * size_found)
```

- max_alloc() locates the largest single block on the free list and allocates it to the caller. It is similar to malloc, except that the caller does not specify the size requested; instead the largest available free block is allocated.
- If the free list is empty when max_alloc() is called, it returns null as the function result and does not use the pointer parameter size_found.
- Otherwise, if there is at least one block on the free list, max_alloc finds the largest such block, removes it from the free list, and returns a pointer to the payload part of the block as the function result. It also stores the payload size in bytes in the variable referenced by the pointer size_found. (i.e., this function has two outputs – a pointer to the allocated block and the size of the block. The pointer is the value returned by the function; the size is stored in the memory location referenced by the parameter.)
- If two or more blocks on the free list have the maximum size, the function removes only one of them from the free list and allocates it. (i.e., you may break ties any way you wish)

Write C-like pseudo-code for this function, using the explicit free list from lab 7. Be sure to adhere to the specification above. Your pseudo-code should be close to C (with comments as needed), but don't worry about syntax details if you don't remember them exactly.

Here is the relevant code from our explicit free list implementation for reference. You may call any of the functions below. You may also remove this page and the next one from the exam if you wish. Write your answer on the page after that.

```
/* Alignment of blocks returned by mm_malloc. */
#define ALIGNMENT 8

/* Size of a word on this architecture. */
#define WORD_SIZE sizeof(void*)

/* Pointer to the first BlockInfo in the free list, the list's
   head. You can treat this as a BlockInfo*.   */
#define FREE_LIST_HEAD *((BlockInfo **)mem_heap_lo())

/* Minimum implementation-internal block size (to account for size
   header, next ptr, prev ptr, and boundary tag).  This is NOT the
   minimum size of an object that may be returned to the caller of
   mm_malloc. */
#define MIN_BLOCK_SIZE 4*WORD_SIZE
```

```
/* Macros for pointer arithmetic to keep other code cleaner. Casting
   to a char* has the effect that pointer arithmetic happens at the
   byte granularity (i.e. POINTER_ADD(0x1, 1) would be 0x2). (By
   default, incrementing a pointer in C has the effect of incrementing
   it by the size of the type to which it points (e.g. BlockInfo).) */
#define POINTER_ADD(p,x) ((char*)(p) + (x))
#define POINTER_SUB(p,x) ((char*)(p) - (x))

/* A BlockInfo contains information about a block, including the size
   and usage tags, as well as pointers to the next and previous blocks
   in the free list.

   Note that the next and prev pointers are only needed when the block
   is free. To achieve better utilization, mm_malloc would use the
   space for next and prev as part of the space it returns.

   +-------------+
   | sizeAndTags | <- BlockInfo pointers in free list point here
   +-------------+
   |    next     | <- Pointers returned by mm_malloc point here
   +-------------+
   |    prev     |
   +-------------+
   |    space    |
   |     ...     |
*/

struct BlockInfo {
/* Size of the block (log(ALIGNMENT) high bits) and tags for whether
   the block and its predecessor in memory are in use. */
int sizeAndTags;
   // Pointer to the next block in the free list.
struct BlockInfo* next;
   // Pointer to the previous block in the free list.
struct BlockInfo* prev;
};
typedef struct BlockInfo BlockInfo;

/* SIZE(x) selects just the higher bits of x to ensure that it is
   properly aligned. Additionally, the low bits of the sizeAndTags
   member are used to tag a block as free/used, etc. */
#define SIZE(x) ((x) & ~(ALIGNMENT - 1))

/* TAG_USED bit mask used in sizeAndTags to mark a block as used. */
#define TAG_USED 1

/* TAG_PRECEDING_USED is the bit mask used in sizeAndTags to indicate
   that the block preceding it in memory is used. (used in turn for
   coalescing) */
#define TAG_PRECEDING_USED 2

void insertFreeBlock(BlockInfo* freeBlock);
void removeFreeBlock(BlockInfo* freeBlock);
void coalesceFreeBlock(BlockInfo* oldBlock);
void * mm_malloc (size_t size);
void mm_free (void *ptr);
```

6. Memory Management (cont)

Implement max_alloc below

```
void * max_alloc(size_t * size_found){

// PLACE YOUR CODE HERE

   BlockInfo * current = FREE_LIST_HEAD;
   if (current == NULL)
      return NULL;

   BlockInfo * maxBlock;  // locn. and size of largest block
   int max = 0;

   while (current != NULL) {
      int size = SIZE(current->sizeAndTags);
      if (size > max) {
         max = size;
         maxBlock = block;
      }
   }

   removeFreeBlock(maxBlock);
   *size_found = max - WORD_SIZE;

   return (void *)POINTER_ADD(maxBlock, WORD_SIZE);
}
```

Notes: Many solutions called mm_alloc with the max size after searching the free list. That "works" so we didn't deduct points for it, but it is not a good solution since it searches the free list a second time.

We didn't worry about 8-byte alignment in the solutions as long as they correctly managed the difference between pointers and sizes of blocks vs. pointers and sizes of the payload area.

7. Processes and Concurrency (12 points)

Suppose we have the following program:

```
int main()
{
      printf("0");
      if (fork() != 0) {
            printf("1");
            fork();
            printf("2");
      } else {
            printf("3");
      }
      exit(0);
}
```

List all of the possible outputs that can occur when this program is run. If it helps, feel free to draw a graph showing the various processes that are created.

The original process prints 0 – 1 – 2 and the second child process prints 2 either before or after the original process prints 2. So those two processes always produce the output sequence 0 – 1 – 2 – 2.

The first child process prints 3. That can happen any time after the first process prints 0. So the possible output sequences are

```
0 3 1 2 2
0 1 3 2 2
0 1 2 3 2
0 1 2 2 3
```