Full Name:_____

Andrew ID:_____

Recitation Section:_____

CS 15-213, Spring 2001

Exam 2

April 17, 2001

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew 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 70 points.
- This exam is OPEN BOOK. You may use any books or notes you like. You cannot, however, use any computers, calculators, palm pilots, ... Good luck!

1:
2:
3:
4:
5:
6:
TOTAL:

Problem 1. (12 points):

In this problem, you will compare the performance of direct-mapped and 2-way associative caches with various program fragments.

To keep calculations simple count the number of cache misses caused only by accesses to array elements. Also assume that

- All caches have a size of 128 bytes and 16-byte cache-lines.
- The arrays are stored in row-major order.
- `array_1, array_2` and `grid` begin at a 16-byte aligned addresses.
- The cache is empty at the beginning of an execution.
- Variables `i`,j are stored in registers and thus, an access to these variables does not change the cache content and does not cause a cache miss.
- Calls to malloc do not change the cache content and do not cause a cache miss.

A. What is the minimum and maximum number of cache misses incurred in any execution of the following code with a direct mapped cache?

```
int *array_1;
int *array_2;
array_1 = (int *) malloc (8*sizeof(int));
array_2 = (int *) malloc (8*sizeof(int));

for (i = 0; i < 8; ++i)
  array_1[i] = 0;
for (i = 0; i < 8; ++i)
  array_2[i] = 0;

for (i = 0; i < 8; ++i)
  array_1[i] = 1;
for (i = 0; i < 8; ++i)
  array_2[i] = 1;
```

The minimum number of cache misses = ...
The maximum number of cache misses = ...

B. What is the minimum and maximum number of cache misses incurred in any execution of the code in part A with a 2-way associative cache?

The minimum number of cache misses = ...
The maximum number of cache misses = ...

C. How many cache misses are incurred in an execution of the following code with a direct mapped cache?

```
int grid[8][8];

for (i=0; i<8; i++){
    for (j=0; j<8; j++)
        grid[i][j] = j;
}
```

The number of cache misses with a direct mapped cache = ...

D How many cache misses are incurred in an execution of the following code with a direct mapped cache?

```
int grid[8][8];

for (i=0; i<8; i++)
    for (j=0; j<8; j++)
        grid[j][i] = i;
```

The number of cache misses with a direct mapped cache = ...

Problem 2. (12 points):

Part 1

Consider a computer with a 12-bit address space and a two level cache. Both levels use a LRU replacement policy. The parameters of the caches are as follows:

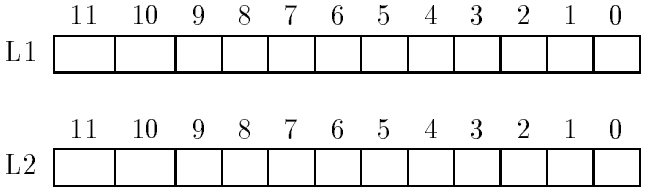
- **L1:** 32 bytes, 2-way set associative, 8-byte cache lines.
- **L2:** 512 bytes, 2-way set associative, 16-byte cache lines.

The boxes below represent the bit-format of a physical address. In each box, for each cache (L1 and L2), indicate which field that bit represents (it's possible that a field doesn't exist). Here are the fields:

O: Byte offset within the cache line

I: The cache (set) index

T: The cache tag



The table below shows a trace of memory accesses (loads) made by the processor. For each access specify whether it is a level 1 cache hit (Ll), a level 2 cache hit (L2), or a miss (M). If the access is a hit, specify which previous access (by line number) loaded the value into the cache. **Assume that initially all cache lines are invalid.**

Load No.	Hex Address	Binary Address	L1, L2 or M	Which line loaded?
1	b57	1011 0101 0111		
2	c55	1100 0101 0101		
3	f74	1111 0111 0100		
4	b50	1011 0101 0000		
5	b5c	1011 0101 1100		
6	f72	1111 0111 0010		
7	b7a	1011 0111 1010		
8	159	0001 0101 1001		
9	c50	1100 0101 0000		
10	b7f	1011 0111 1111		

Suggestion: Work out the L1 hits before dealing with the L2 hits.

Problem 3. (12 points):

You are given four groups of statements relating to memory management and garbage collection below. In each group, only one statement is true. Your task is to mark the statement that is true.

1. (a) In a buddy system, up to 50% of the space can be wasted due to internal fragmentation.
   (b) The first-fit memory allocation algorithm is slower than the best-fit algorithm (on average).
   (c) Deallocation using boundary tags is fast only when the list of free blocks is ordered according to increasing memory addresses.
   (d) The buddy system suffers from internal fragmentation, but not from external fragmentation.

2. (a) Using the first-fit algorithm on a free list that is ordered according to decreasing block sizes results in low performance for allocations, but avoids external fragmentation.
   (b) For the best-fit method, the list of free blocks should be ordered according to increasing memory addresses.
   (c) The best-fit method chooses the largest free block into which the requested segment fits.
   (d) Using the first-fit algorithm on a free list that is ordered according to increasing block sizes is equivalent to using the best-fit algorithm.

3. (a) The advantage of a reference counting garbage collector, when compared to a mark-and-sweep collector, is that it does not fail to free memory that is no longer in use.
   (b) Copying garbage collection is often faster than mark-and-sweep garbage collection (ignoring the cost of copying) because instead of touching all of the allocated blocks, it touches only the ones that are still reachable.
   (c) Copying garbage collection is helpful for reducing internal fragmentation.
   (d) In reference counting, decrementing is a constant time operation.

4. Mark-and-sweep garbage collectors are called conservative if
   (a) they coalesce freed memory only when a memory request cannot be satisfied,
   (b) they treat everything that looks like a pointer as a pointer,
   (c) they perform garbage collection only when they run out of memory,
   (d) they do not free memory blocks forming a cyclic list.

Problem 4. (12 points):

This problem tests your understanding of memory bugs. Each of the four parts below contains a snippet of C code which may or may not contain memory errors. The code compiles with no warnings or errors. If you think that there are no bugs in the code, then simply circle **NO**. Otherwise, please circle **YES** and write the letter(s) that correspond to the bug in the code.

Possible bugs:

A buffer overflow error

B memory leak

C dereference uniniatilized pointer

D incorrect use of address of operator

E may dereference NULL

F frees unallocated memory

G misaligned memory access

```
#include <stdlib.h>

typedef struct snode {
    void *ptr;
    struct snode *next;
} stack_node;

typedef struct stack_s {
    stack_node *head;
} stack_t;
```

a.

```
/* initializes stack to be an empty stack */
void init(stack_t *stack) {
    stack->head = NULL;
}

void gc() {
    stack_t **stack;
    init(*stack);
    /* more code */
}
```

YES (write corresponding letter(s))

NO (no bugs)

b.

```
/* Puts a stack_node with the value ptr
   onto the top of the stack.  You may assume
   the passed arguments are valid and the call
   to malloc does not fail.
*/
void push(stack_t *stack, void *ptr) {
    stack_node *new =
     (stack_node*)malloc(sizeof(stack_node));
    new->ptr = ptr;
    new->next = stack->head;
    stack->head = new;
}
```

YES (write corresponding letter(s))

NO (no bugs)

c.

```
/* removes the stack_node at the top of the stack
   and returns the value that was contained within it.
   You may assume the passed argument is valid.
*/
void *pop(stack_t *stack) {
    if(stack->head) {
       void *ret = stack->head->ptr;
       stack->head = stack->head->next;
       return ret;
    }
    return NULL;
}
```

YES (write corresponding letter(s))

NO (no bugs)

d.

```
/* returns a nonzero value if the stack isn't empty.
   You may assume the passed argument is valid.
*/
int non_empty(stack_t *stack) {
    return (int)stack->head->ptr;
}
```

YES (write corresponding letter(s))

NO (no bugs)

Problem 5. (10 points):

Suppose that the following C program is run on a singple processor machine.

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

int cnt = 0;

int main(void)
{
  if (fork()==0){
    cnt++;
    fork();
    cnt++;
  }
  cnt++;
  printf("%d",cnt);
  return 0;
}
```

List all possible outputs of this program.

Consider the following threaded program.

```
#include <stdio.h>
#include <pthread.h>

int cnt = 0;

void *count(void *arg){
  printf("%d\n",cnt);
  return NULL;
}

int main(void)
{
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, count, NULL);
  pthread_create(&tid2, NULL, count, NULL);
  cnt++;
  pthread_join(tid1,NULL);
  cnt++;
  pthread_join(tid2,NULL);
  return 0;
}
```

List all possible outputs of the first thread (the one whose thread ID is stored in `tid1`).

List all possible outputs of the second thread (the one whose thread ID is stored in `tid2`).

Problem 6. (12 points):

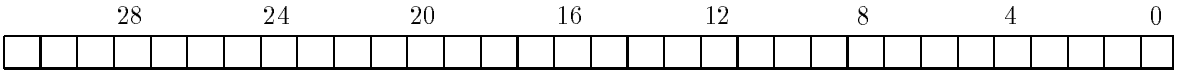
The following problem concerns various aspects of virtual memory.

Part I.

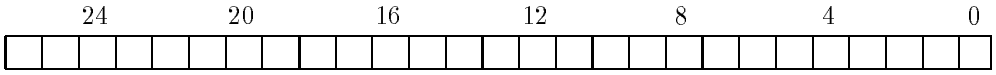
For this part only, the following are attributes of the machine that you will need to consider:

- Memory is byte addressable
- Virtual Addresses are 32 bits wide
- Physical Addresses are 27 bits wide
- Pages are 16KB
- Each Page Table Entry contains:
  - Physical Page Number
  - Valid Bit, Read Protection Bit, and Write Protection Bit

A. The box below shows the format of a virtual address. Indicate the bits used for the VPN (Virtual Page Number) and VPO (Virtual Page Offset).



B. The box below shows the format for a physical address. Indicate the bits used for the PPN (Physical Page Number) and PPO (Physical Page Offset)



C. **Note:** For the questions below, answers of the form 2^i are acceptable. Also, please note the units of each answer

How much *virtual* memory is addressable?

_____ bytes

How much *physical* memory is addressable?

_____ bytes

How many bits is each Page Table Entry?

_____ bits

How large is the Page Table?

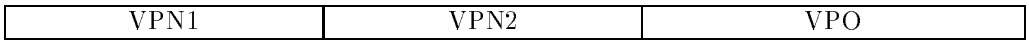
_____ bytes

Part II

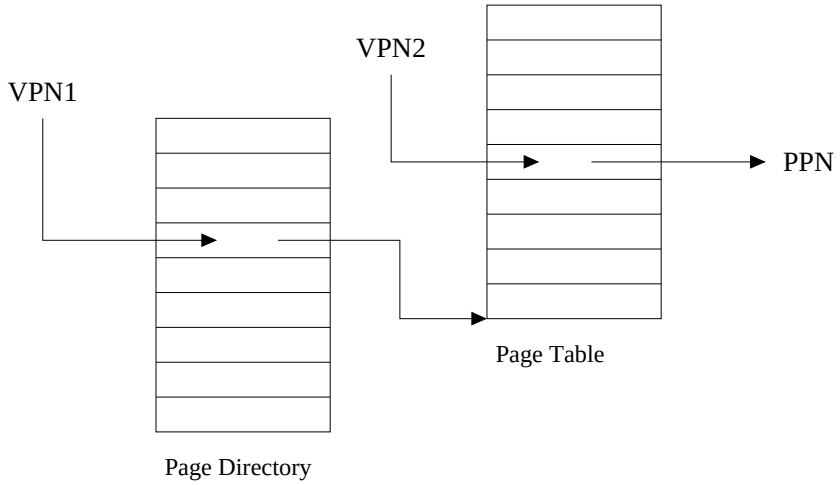
(4 points)

In this part, we will analyze the use of a 2-level page table, in which the entries of the first level table (the Page Directory) tell us the (physical) address in memory of a second page table. This second table's entries finish the address translation by mapping from a second virtual page number to a physical page number.

A virtual address on a machine with a 2-level page table looks like:



While address translation looks like:



For this part, assume the following:

- The machine uses 32-bit addresses, with VPN1 and VPN2 being 10 bits wide and VPO being 12 bits wide
- Page Table Entries and Page Directory Entries are 4 bytes

A. Assume there is a single task running on the system. The task's heap area is allocated in the *physical* range `0x0A000 - 0x19fff`. The task's stack area is allocated in the *physical* range `0x04000 - 0x05fff`. How much memory is in use strictly by the Page Directory and Page Tables?

B. Name one benefit of using 2-level page tables.