Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Spring 2003

Exam 2

April 10, 2003

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 77 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. You may not use a calculator, laptop or other wireless device. Good luck!

1 (8):
2 (10):
3 (9):
4 (20):
5 (8):
6 (6):
7 (8):
8 (8):
TOTAL (77):

Problem 1. (8 points):

Here's a familiar function that returns the nth fibonacci number:

```
int fibo(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    if (n == 2)
        return 1;
    return fibo(n-1) + fibo(n-2);
}
```

This implementation is way too slow. Which of the following are the **two main** contributing factors? (Circle the correct answer)

(a) There are two recursive calls per iteration.

(b) There are too many if checks before the actual recursion.

(c) The overhead of a recursion (calling functions) is too high as compared to the actual work done.

(d) The recursive calls only goes down by 1 or 2 each round. (i.e. $n-1$ and $n-2$)

You have been hired recently to create the world's fastest fibonacci number generator to be run on a **Intel Pentium III** just like our fish machines.

You wrote your first draft as shown below. Assume n takes only natural numbers.

```
int fibo2(int n) {
    int x = 0;
    int y = 1;
    int tmp, i;
    if (n < 2)
        return n;

    for (i = 2; i <= n; ++i) {
        tmp = y;
        y = x + y;
        x = tmp;
    }
    return y;
}
```

This program gave a terrific improvement over the recursive implementation. However, unsatisfied with the result, you seek to further improve the program. Consider the following two alternatives, `fibo3` and `fibo4`. Assume n to be very large (e.g. > 10000).

```
int fibo3(int n) {
    int x = 0;
    int y = 1;
    int i;
    if (n < 2)
        return n;

    for (i = 2; i <= n; i+=2) {
        x = x + y;
        y = x + y;
    }
    if (i == n+1)
        return y;
    return x;
}
```

Is `fibo3` faster than `fibo2`? **Yes** **No**

Why? Please give short answers (one sentence and/or indicative keywords will suffice)

Here’s another one:

```
int fibo4(int n) {
    int p = 0;    int q = 1;    int r = 1;    int s = 2;
    int k = 3;    int l = 5;    int m = 8;    int o = 13;

    int i;
    if (n < 2)  return n;
    if (n == 2) return r;
    if (n == 3) return s;
    if (n == 4) return k;
    if (n == 5) return l;
    if (n == 6) return m;
    if (n == 7) return o;

    for (i = 8; i <= n; i+=8) {
        p = m + o;
        q = o + p;
        r = p + q;
        s = q + r;
        k = r + s;
        l = s + k;
        m = k + l;
        o = l + m;
    }

    if (i == n+1) return o;
    if (i == n+2) return m;
    if (i == n+3) return l;
    if (i == n+4) return k;
    if (i == n+5) return s;
    if (i == n+6) return r;
    if (i == n+7) return q;
    return p;
}
```

Is `fibo4` faster than `fibo3`?   **Yes**   **No**

Why? Please give short answers (one sentence and/or indicative keywords will suffice)

Problem 2. (10 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 12 bits wide.
- The cache is 4-way set associative, with a 2-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 0–1* 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 0–1	Tag	V	Bytes 0–1	Tag	V	Bytes 0–1	Tag	V	Bytes 0–1
0	30	1	4E 47	4B	0	1A D6	77	1	5A B3	EA	0	0D C3
1	09	1	F8 88	AF	1	CA 4A	6C	0	8B 58	47	1	3A 17
2	80	1	4B 59	3B	0	84 0D	A6	1	B4 5B	EE	1	FF 75
3	C4	1	77 CF	77	0	61 DC	3A	1	6B D5	C3	0	1A 9F
4	0D	0	87 2A	66	1	CE 64	7D	1	4E AF	D6	0	89 29
5	E3	1	30 E8	3F	1	1E E8	B4	0	E2 5F	84	1	59 C0
6	60	0	93 2B	35	0	56 46	D4	1	64 CD	FE	0	CA 98
7	A7	1	B2 9B	1B	0	1F 0E	35	1	9E 44	08	0	04 12

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:

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

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

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: `EE4`

A. Physical address format (one bit per box)

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

B. 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: `B4A`

A. Physical address format (one bit per box)

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

B. 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

Problem 3. (9 points):

This problem tests your understanding of memory bugs. Each of the code sequences below may or may not contain memory bugs. The code all compiles without warnings or errors. If you think there is a bug, please circle **YES** and indicate the type of bug from the list below of memory bugs. Otherwise, if you think there are no memory bugs in the code, please circle **NO**.

Bugs:

1. buffer overflow error
2. memory leak
3. dereference of possibly bad pointer
4. incorrect use of free
5. incorrect use of realloc
6. misaligned access to memory
7. Other memory bug

A.
```
typedef struct _stackelem* StackElem;

struct _stackelem {
        void* ptr;
        StackElem next;
};

typedef struct _stack* Stack;
struct _stack {
        StackElem top;
};

void* popptr(Stack s)
{
        void* ret;
        if (s == NULL) return NULL;
        if (s->top != NULL) {
                ret = s->top->ptr;
                s->top = s->top->next;
                return ret;
        }
        return NULL;
}
```

NO **YES** Type of bug(s): ____________________

B.
```
void readints(int* count, int* vals)
{
        int i;

        scanf("%d\n", count);
        vals = malloc(*count * sizeof(int));
        for (i = 0; i<*count; i++) {
                scanf("%d\n", vals+i);
        }
}

void caller(void)
{
        int count;
        int* vals;

        /* some stuff */
        readints(&count, vals);
        /* some more stuff */

        /* all done with inputs */
        free(vals);
}
```

NO **YES** Type of bug(s): ____________________

C.
```
struct IntList {
        int val;
        struct IntList* next;
};

struct IntList*
addToList(struct IntList* old, int v)
{
        struct IntList* newone;

        newone = malloc(sizeof(struct IntList));
        newone->val = v;
        newone->next = old;
        return newone;
}
```

NO **YES** Type of bug(s): ____________________

Problem 4. (20 points):

Imagine that you are implementing a console driver. The job of a console driver is to take characters and output them to the screen. The contents of the console is controlled by a region of main memory (memory mapped I/O). Each character on the console is represented in this region by a pair of integers. The first number in this pair is simply the character itself. The second integer controls the foreground and background colors used to draw the character. These integer pairs are stored in row major order. For your console, there will be 25 rows of 80 characters each.

We define the following structure:

```
struct consoleChar
{
  int character;
  int color;
}

struct consoleChar buffer[25][80];
```

To simplify the reasoning about this problem, you can assume the following:

- `sizeof(int) = 4`
- Buffer begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array buffer. All variables are stored in registers.
- All code is compiled with -O0 flag (no optimizations).

Note: Be careful about the number of accesses per loop.

An obvious thing that we might to do is to clear the console - this requires writing a black colored 'space' to all positions of the console. Consider the following algorithm to clear the console:

```
for(row=0; row<25; row++)
  for(col=0; col<80; col++)
  {
    buffer[row][col].character = ' ';
    buffer[row][col].color = 0x00;
  }
```

Assume your cache is 1024-byte direct-mapped data cache with 64-byte lines. What is the cache miss rate?

Assume your cache is 1024-byte 4-way set associative using an LRU replacement policy with 32-byte lines. What is the cache miss rate?

Now, we change the algorithm a little - instead of clearing the screen by rows, we will do so by columns. Thus we change the algorithm to be the following:

```
for(col=0; col<80; col++)
  for(row=0; row<25; row++)
  {
    buffer[row][col].character = ' ';
    buffer[row][col].color = 0x00;
  }
```

Assume your cache is 512-byte direct-mapped data cache with 64-byte lines. What is the cache miss rate?

Assume your cache is 1024-byte 2-way set associative using an LRU replacement policy with 64-byte lines. What is the cache miss rate?

Another thing we might want is to count how many non-blank characters we have. Consider the following algorithm:

```
for(row=0; row<25; row++)
  for(col=0; col<80; col++)
  {
    cChar = buffer[row][col].character;
    cColor = buffer[row][col].color;
    if (cChar != ’ ’ || cColor != 0x00)
      nonBlankCounter++;
  }
```

Assume your cache is 128-byte direct-mapped data cache with 16-byte lines. What is the cache miss rate?

Now we add another feature to our driver - scrolling. We implement it via the following algorithm:

```
for(row = 0; row < 24; row++)
  for(col = 0; col < 80; col++)
  {
    buffer[row][col].character = buffer[row+1][col].character;
    buffer[row][col].color = buffer[row+1][col].color;
  }
```

Assume your cache is 640-byte direct-mapped data cache with 640-byte lines. What is the cache miss rate?

Assume your cache is 1280-byte 2-way set associative using an LRU replacement policy with 640-byte lines. What is the cache miss rate?

Problem 5. (8 points):

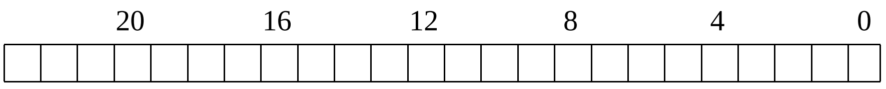
The following problem concerns various aspects of virtual memory.

Part I.

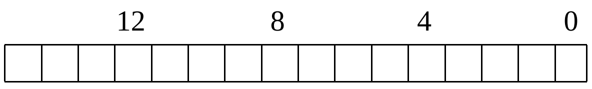
For this part only, the following are attributes of the machine that you will need to consider:

- Memory is byte addressable
- Virtual Addresses are 24 bits wide
- Physical Addresses are 16 bits wide
- Pages are 1KB
- Each Page Table Entry contains:
  - Physical Page Number
  - Valid Bit, Read Only bit (1 for Read Only, 0 for Read/Write)

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

The Xbox video game console comes configured with a Pentium III 733MHz processor with 64MB of RAM. Games running on the Xbox console do not make use of a virtual memory subsystem, because they are performance sensitive. Give two aspects of a virtual memory subsystem that would cause the designers to forego using virtual memory at all. **Note:** Short, simple answers are fine, but be clear as to how performance suffers.

Problem 6. (6 points):

This problem tests your understanding of Unix process control.
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.

```
int main()
{
    int i = 1;

    printf("%d",i);

    if(fork() == 0)
    {
        i++;
        printf("%d",i);
        i++;
    }
    else
    {
        i+=3;
        printf("%d",i);
        wait(NULL);
    }

    printf("%d",i);
}
```

List all possible outputs of this program.

Problem 7. (8 points):

This question tests your understanding of signals and signal handlers.
Assume that all functions and procedures return correctly. If the `pid` argument of `kill` equals 0, then the `sig` argument is sent to every process in the process group of the current process. Assume that printf is unbuffered.

```
void handler(int sig)
{
    printf("hello"\n");
}

int main()
{
    int pid;
    setpgid(0,0);
    signal(SIGCHLD, handler);
    if((pid = fork()) == 0)
    {
        kill(0, SIGCHLD);
        printf("bonjour\n");
        exit(0);
    }
    else
    {
        waitpid(pid, NULL, 0);
    }
    printf("hola\n");
}
```

Draw an **X** through any column which does not represent a valid possible output of this program.

hello	hello	hello	hello	hello	hello
hello	hello	hello	hello	bonjour	bonjour
bonjour	bonjour	hello	hola	hola	hello
hello	hola	bonjour	bonjour		hello
hola		hola			hola

Problem 8. (8 points):

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

A. Where would a conservative garbage collect for C find the root set of pointers?

B. What advantage does a mark-and-sweep collector have over a reference-counting collector?

C. What could go wrong when the following two files are linked together?

```
int x;
int y;
foo(int a) {
        x=a;
        y=a*a;
}
```

```
static int y;
double x;
bar(int b) {
        x = (double)b;
        y = b * b;
}
```

Answer the following questions by circling the correct answer:

D. When finding a free block, the segregated free-list algorithm aproximates best-fit algorithm in memory usage, but has a time complexity similar to the first-fit algorithm.

True False

E. In the call `realloc(p, 128)`, current value of `p` does not have to be the result of a previous malloc or realloc.

True False

F. A genius programmer in 213 found a way to eliminate all internal fragmentation by eliminating the need for a header and a boundary tag. The resulting malloc package achieves 100% peak memory utilization.

True False

G. Deferred coalescing often performs better than immediate coalescing, but can result in an increase in the amortized time for a free operation.

True False

H. Deferred coalescing reduces false fragmentation

True False