

# CSE351 Spring 2012 – Final Exam (6 June 2012)

---

Please read through the entire examination first! We designed this exam so that it can be completed in 90 minutes and, hopefully, this estimate will prove to be reasonable.

There are 5 problems for a total of 170 points. The point value of each problem is indicated in the table below and at every part of every problem. 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 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*                    

ID#: \_\_\_\_\_

<b>Problem</b>	<b>Max Score</b>	<b>Score</b>
1	30	<b><i>30</i></b>
2	30	<b><i>30</i></b>
3	30	<b><i>30</i></b>
4	30	<b><i>30</i></b>
5	50	<b><i>50</i></b>
<b>TOTAL</b>	<b>170</b>	<b><i>170</i></b>

## 1. Data Representation (30 points)

Draw the memory layout for the following struct (assume a 32-bit architecture and alignment to 8-byte boundaries – in other words, all structs must be a multiple of 8 bytes – with any padding at the end of the struct rather than between fields):

```
typedef struct xform {
    int i[2][2];
    float * factor;
    int color;
    int x;
} xform;
```

a) (10 points) Draw an int as , a pointer as , and wasted space as a shaded or hashed box .

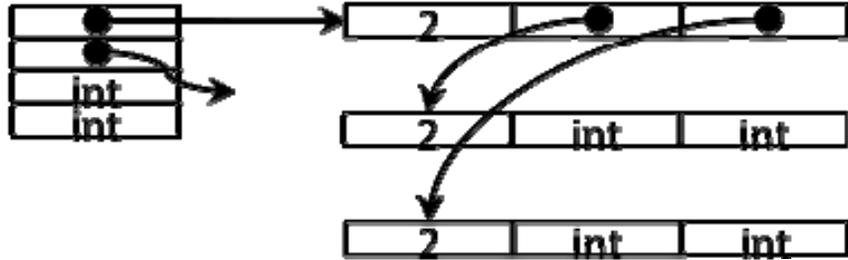


b) (5 points) How much fragmentation is there for this data structure? Recall that fragmentation is the percentage of wasted memory due to alignment considerations, in other words:

$$\frac{\text{wasted bytes due to alignment for one element}}{\text{total size of one struct} + \text{wasted bytes due to alignment for one element}} * 100\%$$

$$\frac{4}{28 + 4} * 100\% = \underline{12.5\%}$$

c) (15 points) Redraw the memory that Java would use. Remember that in Java, arrays are always 1-dimensional and use pointers to point to the next level (make sure your pointers in the diagram go to the right place). You can use non-contiguous parts of memory as needed.



## 2. Puzzlers for Pointers, Addresses, and Values (30 points)

Back to C. A memory has the following contents (in little-endian format):

Variable	Address	Bytes	Final Value of Bytes
A	0x08000000	00 00 00 08	0c 00 00 08
B	0x08000004	04 00 00 08	
C	0x08000008	fe ff ff ff	00 00 00 00
D	0x0800000c	ff ff ff ff	02 00 00 00
E	0x08000010	00 00 00 00	05 00 00 00
F	0x08000014	01 00 00 00	18 00 00 08
G	0x08000018	02 03 04 05	
H	0x0800001c	33 35 31 00	04 00 00 08

Given the following declarations (assuming a 32-bit architecture):

```
int *A, *B;
float C, E, G;
int D, F;
typedef struct xform { int i[2][2];
                      float * factor;
                      int color;
} xform;
xform *H;
```

Fill in columns for the address (in hex) that is changed in each statement and the value (in hex) to which it is changed. **NOTE: The statements are executed in sequence and changes made to memory apply in the following lines.**

C Statement (5 points each)	Address (hex)	Value (hex)
A = B + 2;	0x08000000	0x0800000c
C = (float) (*A + F);	0x08000008	0x00000000
H = (xform *) &B;	0x0800001c	0x08000004
H->factor = &E + 2;	0x08000014	0x08000018
D = (int) *((char *) (H->factor));	0x0800000c	0x00000002
H->i[(D >> 1)][1] = D + 3;	0x08000010	0x00000005

### 3. Processes and Virtual Memory (30 points)

Answer the following questions with just a couple of sentences.

a) (4 points) What are the two principal abstractions that make the process model easy for programmers?

- 1) *Each process thinks it is the only one running on the processor.*
- 2) *Each process thinks it has the full memory address space at its disposal.*

b) (6 points) When creating a child process as a result of a fork, what does the operating system copy from the parent process to give to the child process? Is there anything that is done differently for parent and child?

*It creates an exact copy of the parent process's entire memory (including stack, heap, page tables, etc.), instruction pointer register, and condition codes. The child process sees the fork return a 0 while the parent process sees the fork return the process ID of the child.*

c) (6 points) Are pointers in C and references in Java the same thing? Give an example to illustrate your answer.

*Pointers in C can point to any memory location. Pointers in Java can only point to the start of an object (struct). As an example, in Java we can't create a pointer to a field within an object.*

d) (6 points) What are some types of information that need to be explicitly saved by the operating system when switching between processes? You can assume you have a write-through cache. Put a check mark in the “Yes” or “No” column.

	Yes	No
Register contents	✓	
Condition codes	✓	
Program instruction pointer	✓	
Stack memory		✓
Cache contents		✓
TLB entries		✓

e) (4 points) In a virtual memory system with 32-bit virtual addresses and 4KB page sizes, how large is the page table? How many pages will it require just to store this table?

*32 – log(4K) = 32-12 = 20 bits, thus we need 2<sup>20</sup> or 1M entries. With each entry being 4 bytes, we’ll need 4MB of page storage or 1K pages of 4K bytes each.*

f) (4 points) If a cache has a total storage capacity of 1024 bytes and is organized into 8 byte blocks, how many sets does it have if it is 2-way set associative? If addresses are 16 bits, how many bits are used for the tag?

*C = B \* S \* E where C is cache size, B is block size, S is number of sets, and E is the degree of associativity (E =1 for a direct mapped cache). In this case, by substitution, we have that 1024 = 8 \* S \* 2. Solving for S, we know we have 64 sets. Thus, 6 bits are used to specify the set, while 3 bits are used to specify the byte. In a 16-bit architecture, that leaves 16 – 6 – 3, or 7 bits for the tag.*

#### 4. Caches and Virtual Memory (30 points)

A system implements virtual memory with a translation look-aside buffer (TLB – 16 entries, 2-bytes lines, 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.

TLB

Set	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

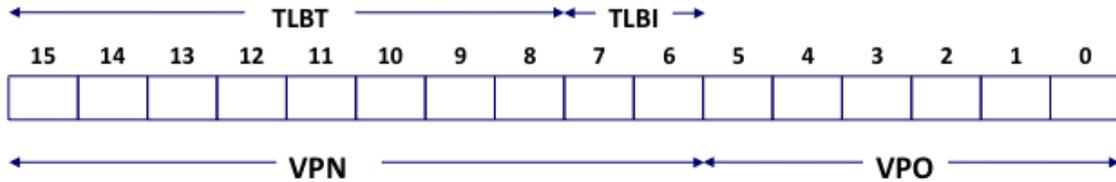
Page Table

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

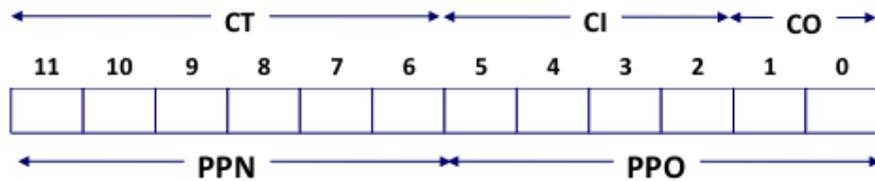
Cache

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) (5 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).



b) (5 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).



c) (20 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).

Virtual Address	Physical Address	Value	TLB Miss?	Page Fault?	Cache Miss?
0x036A	0xB6A	0xDA	N	N	N
0x013F	ND	ND	Y	Y	ND
0x0722	0x0A2	ND	N	N	Y
0x0000	0xA00	0x99	Y	N	N

## 5. Call Stacks and Memory Allocation (50 points)

In this problem you will implement three debugger commands for a 32-bit x86 Linux machine. Since this is a 32-bit machine, remember that all pointers and registers have 32 bits, and remember that the stack looks like this:

```

    [   ...   ]
    [  args   ]  Higher Addresses
    [ ret addr ]
%ebp-> [ old %ebp ]
    [ local vars ]  Lower Addresses
    [   ...   ]
```

a) (20 points) First we will implement an equivalent of gdb's "backtrace" command. This command prints information about each stack frame, including the current stack frame and everything up to the call to main(). For example, if we interrupt the code on the left with a breakpoint at the line marked with an arrow and then invoke the "backtrace" command, our debugger might print the output shown on the right:

```

1   int foo(int x) {
2 =>   x += 6;
3   return x;
4   }
5   int main(int argc, char *argv[]) {
6     int y = foo(5);
7   }
```

(gdb) backtrace  
in foo() at line 2  
in main() at line 6

We have started the implementation for you. Your job is to complete the printBacktrace() function given two functions to call: printCallerStackFrame() to print each frame on the stack, and isMain() to determine when you've reached the main stack frame (which is the last stack frame you should print).

```
void printCallerStackFrame(int eip, int ebp);
bool isMain(int eip);

void printBacktrace(int eip, int ebp) {
    printCallerStackFrame(eip, ebp);

    // ----- Your code here -----

    while (!isMain(eip)) {
        eip = *((int*)ebp + 1);
        ebp = *(int*)ebp;
        printCallerStackFrame(eip, ebp);
    }

    // -----
}
```

b) (10 points) Your machine comes with a simple memory allocator that uses an implicit free list. Each block in the heap has a 32-bit header that contains the size of the block along with a “used” bit that is set when the block is allocated. The size is always aligned to 8 bytes, and the “used” bit is stored in the least-significant bit of the header.

This is almost exactly like the implicit list we saw in lecture, with one difference: every block starts with a special 4-byte magic word that is always 0xdeadbeef. Every call to malloc() ensures that the magic word is set to 0xdeadbeef. The format of a block is shown below:

```
[ magic word ] (should be 0xdeadbeef)
[ header     ] (size and TAG_USED bit)
[ payload    ]
[ ...        ]
```

We will implement a “blockinfo” command. This command takes one argument, a pointer “x”, and then prints information about the block pointed to by “x”. Namely, it prints the size of the block and whether or not the block is currently allocated. You can assume “x” always points to the payload area of an object. Your job is to finish the implementation of printBlockInfo(). You need to compute “size”, which is the size of the block, and “alloc” which is 1 if the block is allocated, and 0 otherwise. We have provided macros that will be useful. Remember to cast your pointers correctly!

```
#define SIZE(x) ((x) & ~7)
#define TAG_USED 1

void printBlockInfo(void *x) {
    int size;
    int alloc;

    // ----- Your code here -----

    int header = *((int*)x - 1);
    size = SIZE(header);
    alloc = header & TAG_USED

    // -----

    printf(" Block size: %d\n", size);
    printf(" Block allocated?: %d\n", alloc);
}
```

c) (10 points) Lastly, we will implement a “memcheck” command that will scan the heap and ensure that the magic word is set to 0xdeadbeef for every block in the heap. Your job is to finish implementing the memcheck() function below. It should return “true” if all blocks have a valid magic word, and “false” otherwise. The heap ends with a special block of size 0.

```
struct BlockInfo {
    int magic;
    int header;
};

define MAGIC 0xdeadbeef

bool memCheck(BlockInfo *firstBlock) {
    BlockInfo *block;

    block = firstBlock;

    // ----- Your code here -----

    while (SIZE(block->header) != 0) {
        if (block->magic != MAGIC)
            return false;
        block = (BlockInfo*)((char*)block + SIZE(block->header));
    }
    return true;

    // -----
}
```

d) (10 points) What kind of bugs does “memcheck” help find in the memory allocator? What kind of bugs does “memcheck” help find in user programs? Describe an example of each type of bug. Also, describe a bug “memcheck” does not find.

*In the memory allocator, “memcheck” ensures there are no unaccounted for areas of memory, in other words, they are all well-formed blocks with 0xdeadbeef at the front and a size (until the last block of size 0 is encountered). In user programs, “memcheck” finds evidence of buffer overflow bugs – overwriting past the end of a block (as in a buffer overflow). “memcheck” does not catch bugs where free blocks are not properly coalesced.*

## REFERENCES

### **Powers of 2:**

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = .5$
$2^2 = 4$	$2^{-2} = .25$
$2^3 = 8$	$2^{-3} = .125$
$2^4 = 16$	$2^{-4} = .0625$
$2^5 = 32$	$2^{-5} = .03125$
$2^6 = 64$	$2^{-6} = .015625$
$2^7 = 128$	$2^{-7} = .0078125$
$2^8 = 256$	$2^{-8} = .00390625$
$2^9 = 512$	$2^{-9} = .001953125$
$2^{10} = 1024$	$2^{-10} = .0009765625$

### **Data Sizes:**

$$1 \text{ KB} = 2^{10} \text{ B}$$

$$1 \text{ MB} = 2^{20} \text{ B}$$

$$1 \text{ GB} = 2^{30} \text{ B}$$

### **Binary-Hex conversion:**

$$0x0 = 0b0000$$

$$0x1 = 0b0001$$

$$0x2 = 0b0010$$

$$0x3 = 0b0011$$

$$0x4 = 0b0100$$

$$0x5 = 0b0101$$

$$0x6 = 0b0110$$

$$0x7 = 0b0111$$

$$0x8 = 0b1000$$

$$0x9 = 0b1001$$

$$0xA = 0b1010$$

$$0xB = 0b1011$$

$$0xC = 0b1100$$

$$0xD = 0b1101$$

$$0xE = 0b1110$$

$$0xF = 0b1111$$