Full Name: ____________________________

CS 351 Fall 2012

Final Exam

December 7th, 2012

Instructions:

- This exam is closed-book, closed-notes.
- Calculators are not permitted.
- Write your full name on the front and make sure your exam is not missing any sheets.
- Write your answers legibly in the space provided for each problem. If you make a mess, clearly indicate your final answer.
- Good luck!

Score
Problem 1 (/20) :
Problem 2 (/8) :
Problem 3 (/8) :
Problem 4 (/8) :
Problem 5 (/12) :
Problem 6 (/10) :
Problem 7 (/9) :
Problem 8 (/10) :
TOTAL (/85) :

Problem 1. (20 points):

Please circle the *single best* answer to each of the following questions.

1. By "hiding" open file metadata behind file descriptors, the operating system makes it so that processes *cannot*:
   (a) share read/write positions within the same file
   (b) access the same file simultaneously
   (c) access a file without explicitly calling `open` on it
   (d) access file data/metadata except by going through the kernel

2. One of the guarantees offered by the pipe IPC facility is that:
   (a) all writes to the pipe are *atomic*
   (b) no more than two processes can be reading/writing to the same pipe simultaneously
   (c) so long as the pipe can be written to, readers will not receive an EOF
   (d) writes to the pipe will return an error until at least one reader is blocking

3. In which of the following situations is it not a good idea to be using the buffered `stdio.h` I/O functions?
   (a) when reading from a pipe
   (b) when writing to a block special file
   (c) when reading and writing from/to a regular file
   (d) when reading and writing from/to a character special file

4. Which of the following interprocess communication mechanisms is best suited for the **custom synchronization** of multiple processes?
   (a) shared memory
   (b) semaphores
   (c) named pipes
   (d) file locks

5. In a cache which resides at the uppermost level of the memory hierarchy (i.e., just below the CPU/registers), we prioritize:
   (a) improving the hit rate
   (b) minimizing the hit time
   (c) the implementation of complex replacement policies
   (d) high amounts of associativity

6. Which of the following policies is most likely to be adopted together with *write-back*?
   (a) write-through
   (b) write-around
   (c) write-allocate
   (d) no-write-allocate

7. Which of the following caching techniques most directly leverages good **spatial locality**?
   (a) increasing the block size
   (b) increasing the total number of lines
   (c) increasing associativity
   (d) implementing LRU

8. Which of the following will likely result in an *increase* in the size of per-process page tables?
   (a) decreasing the size of a page
   (b) installing more physical memory
   (c) decreasing the size of virtual addresses
   (d) using a multi-level page table (instead of single-level one)

9. While buddy-system based allocators can reduce the amount of required per-block metadata and simplify searching and freeing, they are not often used in practice. Why?
   (a) they are computationally intractable
   (b) they significantly reduce malloc/realloc throughput
   (c) they can result in large amounts of internal fragmentation
   (d) they cannot be used together with other allocation mechanisms

10. When it comes to optimizing a given dynamic memory allocator implementation, the hardest element to quantify and predict (and therefore improve) is typically:
    (a) the quantity of space lost due to boundary tags
    (b) the quantity of space lost due to alignment/padding
    (c) the asymptotic (big O) runtime of conducting a best-fit search
    (d) the amount of external fragmentation

The following two programs are failed attempts by a student to get a parent and child process to communicate — the child is intended to run the `/bin/ls` program (which produces a directory listing to stdout) and the parent is to run the `/usr/bin/wc` program (which counts the characters, words, and lines in its stdin and prints the counts to stdout).

Note that the `O_CREAT` flag to the `open` system call indicates that the named file is to be created if it doesn't already exist.

```
/* program A */
#define PROG1 "/bin/ls"
#define PROG2 "/usr/bin/wc"
#define SHARED_FILE "shared.txt"

main()
{
    int fd;
    if (fork() == 0) {
        /* child */
        fd = open(SHARED_FILE, O_WRONLY | O_CREAT);
        dup2(fd, 1);
        execl(PROG1, PROG1, NULL);
    } else {
        /* parent */
        fd = open(SHARED_FILE, O_RDONLY | O_CREAT);
        dup2(fd, 0);
        execl(PROG2, PROG2, NULL);
    }
}
```

```
/* program B */
#define PROG1 "/bin/ls"
#define PROG2 "/usr/bin/wc"

main()
{
    int fds[2];
    pipe(fds);
    if (fork() == 0) {
        /* child */
        dup2(fds[1], 1);
        execl(PROG1, PROG1, NULL);
    } else {
        /* parent */
        dup2(fds[0], 0);
        execl(PROG2, PROG2, NULL);
    }
}
```

Problem 2. (8 points):

Refer to the programs on the previous page to answer the following questions.

A. After running program A multiple times, the student observes that sometimes the output produced by the `/usr/bin/wc` program indicates that the file is empty. Why?

   (a) the parent process is running before the child process
   (b) the child process is running before the parent process
   (c) the child process is deleting the file created by the parent
   (d) the system is incorrectly buffering the output of the parent

B. How would you go about fixing program A?

   (a) create a file lock in the child on `SHARED_FILE`
   (b) swap the code in the `if` and `else` clauses
   (c) add `wait(NULL)` in the child just before calling `open`
   (d) add `wait(NULL)` in the parent just before calling `open`

C. Upon running program B, the student observes that there is no output whatsoever. Why?

   (a) the pipe was created improperly
   (b) the child process is running before the parent process
   (c) the child process is blocking indefinitely
   (d) the parent process is blocking indefinitely

D. How would you go about fixing program B?

   (a) add `close(fds[1])` just before calling `fork`
   (b) add `close(fds[0])` in the child just before calling `exec`
   (c) add `close(fds[1])` in the parent just before calling `exec`
   (d) add `wait(NULL)` to the parent just before calling `exec`

Problem 3. (8 points):

Consider the following function which takes pointers to three non-overlapping arrays (`arr1`, `arr2`, `arr3`) of word-sized elements, and the number (`n`) of elements in each to be processed:

```
int foo (int *arr1, int *arr2, int *arr3, int n)
{
    int i, accum;

    for (i=0; i<n; i++)   /* loop #1 */
        accum += arr1[i] + arr2[i];

    for (i=0; i<n; i++)   /* loop #2 */
        accum += arr2[i] * arr3[i];

    return accum;
}
```

For A-C, assume a direct-mapped cache with 64 total lines, each containing a 4-word block.

A. What is the *best possible* ***hit rate*** that can be achieved over the arrays accessed while executing loop #1? Assume the cache was "cold" before `foo` was called.

(a) 12.5% (1/8)	(b) 25% (1/4)
(c) 50% (1/2)	(d) 75% (3/4)

B. Following the execution of loop #1 what is the *best possible* ***hit rate*** that could be achieved during the execution of loop #2? Again assume that the cache was "cold" beforehand.

(a) 37.5% (3/8)	(b) 50% (1/2)
(c) 87.5% (7/8)	(d) 100%

C. What is the *worst possible* average hit rate that might be achieved over loops #1 and #2?

(a) 0%	(b) 6.25% (1/16)
(c) 25% (1/4)	(d) 75% (3/4)

D. Suppose we switch over to using a 4-way set associative cache with each line containing a 8-word block. What would be the new worst case hit rate over loops #1 and #2?

(a) 12.5% (1/8)	(b) 25% (1/4)
(c) 50% (1/2)	(d) 87.5% (7/8)

Problem 4. (8 points):

Consider a system that uses 42-bit virtual addresses and has 4GB (2^{32} bytes) of physical DRAM installed. Paging is used as a virtual memory implementation, and the page size is 16KB (2^{14} bytes). Memory is byte-addressed, and the word size is 8 bytes.

A. What is the maximum amount of data each process can store in its address space?

(a) 2^{32} bytes (4GB)
(b) 2^{35} bytes (32GB)
(c) 2^{42} bytes (4TB)
(d) 2^{56} bytes (64PB)

B. What is the total number of pages in a process's virtual address space?

(a) 2^{10} pages
(b) 2^{18} pages
(c) 2^{28} pages
(d) 2^{32} pages

C. Given single-level page tables with word-sized page table entries, how large is the size of each per-process page table?

(a) 2^{21} bytes (2MB)
(b) 2^{24} bytes (16MB)
(c) 2^{31} bytes (2GB)
(d) 2^{42} bytes (4TB)

D. What would be the *average* amount of internal fragmentation arising from the described virtual memory setup?

(a) 2^{7} bytes (128 bytes)
(b) 2^{13} bytes (8KB)
(c) 2^{24} bytes (16MB)
(d) 2^{29} bytes (512MB)

Problem 5. (12 points):

The questions on the next three pages are based on the cache and memory setup described below:

- Memory is byte addressable
- Virtual addresses are 16 bits wide
- Physical addresses are 12 bits wide
- The page size is 32 bytes (2^5 bytes)
- The TLB is 4-way set associative, with 8 total lines
- The cache is 4-way set associative, with 2-byte blocks and 8 total lines

The current TLB, cache, and partial page table contents are given here:

TLB			
Index	Tag	Valid	PPN
0	27B	1	30
	2BA	1	5D
	255	0	1A
	2F5	1	47
1	2D7	0	03
	0EF	1	58
	037	1	2C
	355	1	5E

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	0A	1	08	7F	1
01	2F	0	09	7D	0
02	62	0	0A	0D	0
03	78	0	0B	13	0
04	1D	0	0C	6C	1
05	77	1	0D	6F	0
06	65	1	0E	24	1
07	3D	1	0F	23	0

Cache				
Index	Tag	Valid	Byte 0	Byte 1
0	0D3	1	2D	E3
	0E0	0	5F	61
	36F	1	85	15
	162	1	81	28
1	379	0	8E	20
	218	0	67	43
	13A	0	16	8D
	1EA	1	BA	30

And for your convenience, here's a decimal ↔ hex ↔ binary lookup table:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Part I (4 points)

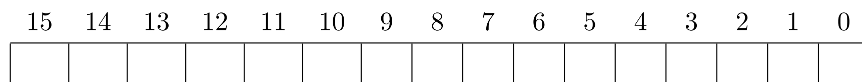
The diagram below shows the format of a virtual address. Label it with the fields (indicate their positions) that would be used to determine the following:

VPO : The virtual page offset

VPN : The virtual page number

TLBI : The TLB index

TLBT : The TLB tag



The diagram below shows the format of a physical address. Label it with the fields (indicate their positions) that would be used to determine the following:

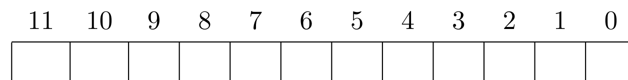
PPO : The physical page offset

PPN : The physical page number

CO : The cache block offset

CI : The cache set index

CT : The cache tag

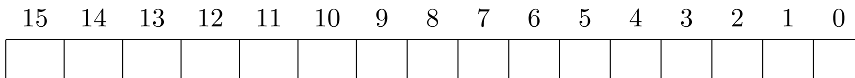


Part II (8 points)

For the virtual address on this and the following page, fill in the diagrams and tables in parts A-D using the address translation and cache structures described. If a page fault occurs, you should leave parts C and D blank.

Virtual Address: **0x00EB**

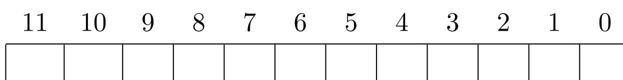
A. Virtual address format



B. Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

C. Physical address format

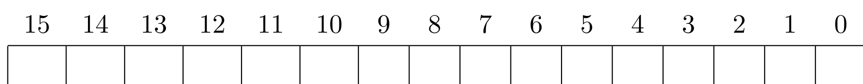


D. Physical memory reference

Parameter	Value
Cache offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Virtual Address: **0x0DE8**

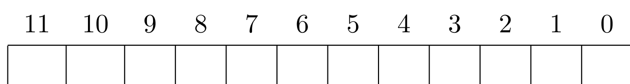
A. Virtual address format



B. Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

C. Physical address format



D. Physical memory reference

Parameter	Value
Cache offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Problem 6. (10 points):

Given the following sequence of calls to `malloc` and `free`:

```
void *p1, *p2, *p3, *p4, *p5, *p6, *p7;
p1 = malloc(40);
p2 = malloc(85);
p3 = malloc(30);
free(p2);
p4 = malloc(20);
free(p3);
p5 = malloc(100);
free(p1);
free(p4);
p6 = malloc(10);
p7 = malloc(50);
```

A. What is the maximum aggregate payload (P_{max})?

(a) 100 (b) 125

(c) 160 (d) 335

B. Given a final heap size of 1024 bytes, what is the peak memory utilization?

(a) $P_{max}/1024$ (b) $P_{max}/(1024 - 335)$

(c) $P_{max}/(P_{max} + 1024)$ (d) $1024 - P_{max}$

Next, consider an explicit-list based dynamic memory allocator with the following properties:

- blocks and payloads are word-aligned, with a word size of 8 bytes
- all blocks have a header and footer word used to store the block size & allocated bit
- free blocks additionally have next and prior pointers for free-list maintenance
- allocated blocks that have no room for payload are not permitted

C. What is the minimum size of a block?

(a) 8 bytes (b) 16 bytes

(c) 32 bytes (d) 40 bytes

D. Given a payload request of 50 bytes, what is the minimum block size to be allocated?

(a) 56 bytes (b) 72 bytes

(c) 82 bytes (d) 88 bytes

E. How much space is lost to internal fragmentation for the block described in part D?

(a) 8 bytes (b) 14 bytes

(c) 22 bytes (d) 32 bytes

Problem 7. (9 points):

Consider an allocator that uses an implicit free list. Each memory block is word-aligned (4-byte words), with word-sized boundary tags that store the block size and use their LSB for the allocated bit. The allocator performs immediate coalescing, and splits blocks whenever doing so results in two blocks with non-zero payload. When splitting, the "lower" part of a block is allocated.

Starting with the heap on the left below, show the resulting heap following each successive call to `malloc` and `realloc`. Use a *first-fit* allocation policy that starts searching from the bottom of the heap shown. Note that you only need to show header/footer word contents.

Start state:		After `p = malloc(14):`		After `q = malloc(7):`		After `realloc(p, 35):`	
Addr.	Data	Addr.	Data	Addr.	Data	Addr.	Data
0x80005068	0x0000006C	0x80005068		0x80005068		0x80005068	
0x...5064	—	0x...5064		0x...5064		0x...5064	
0x...5060	—	0x...5060		0x...5060		0x...5060	
0x...505c	—	0x...505c		0x...505c		0x...505c	
0x...5058	—	0x...5058		0x...5058		0x...5058	
0x...5054	—	0x...5054		0x...5054		0x...5054	
0x...5050	—	0x...5050		0x...5050		0x...5050	
0x...504c	—	0x...504c		0x...504c		0x...504c	
0x...5048	—	0x...5048		0x...5048		0x...5048	
0x...5044	—	0x...5044		0x...5044		0x...5044	
0x...5040	—	0x...5040		0x...5040		0x...5040	
0x...503c	—	0x...503c		0x...503c		0x...503c	
0x...5038	—	0x...5038		0x...5038		0x...5038	
0x...5034	—	0x...5034		0x...5034		0x...5034	
0x...5030	—	0x...5030		0x...5030		0x...5030	
0x...502c	—	0x...502c		0x...502c		0x...502c	
0x...5028	—	0x...5028		0x...5028		0x...5028	
0x...5024	—	0x...5024		0x...5024		0x...5024	
0x...5020	—	0x...5020		0x...5020		0x...5020	
0x...501c	—	0x...501c		0x...501c		0x...501c	
0x...5018	—	0x...5018		0x...5018		0x...5018	
0x...5014	—	0x...5014		0x...5014		0x...5014	
0x...5010	—	0x...5010		0x...5010		0x...5010	
0x...500c	—	0x...500c		0x...500c		0x...500c	
0x...5008	—	0x...5008		0x...5008		0x...5008	
0x...5004	—	0x...5004		0x...5004		0x...5004	
0x80005000	0x0000006C	0x80005000		0x80005000		0x80005000	

Problem 8. (10 points):

Consider an implicit-list based allocator and memory architecture with the following properties:

- words are 4 bytes large (`int`s and addresses are word-sized)
- memory blocks are double-word (8-byte) aligned
- each block has a header and footer word, which stores its size and an allocated bit (0 for free, 1 for allocated)

The five functions that follow are defined to facilitate the implementation of `free`. The behavior of each function is explained in the comment that precedes it. Circle the letter next to the line of code that correctly completes each function.

```
/* given a pointer p to an allocated block, i.e., p is a
   pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the header of the block */
void *header(void* p)
{
  void *ptr;

  --------------------;
  return ptr;
}
```

(a) `ptr = p`

(b) `ptr = (char *)p + 4`

(c) `ptr = (int *)p - 1`

```
/* given a pointer to a valid block header or footer,
   returns the size of the block */
int size(void *hp)
{
  int result;

  --------------------;
  return result;
}
```

(a) `result = *hp & ~7`

(b) `result = (*(char *)hp & ~5) << 2`

(c) `result = *(int *)hp & ~7`

```
/* given a pointer p to an allocated block, i.e. p is
   a pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the footer of the block */
void *footer(void *p)
{
  void *ptr;

  --------------------;
  return ptr;
}
```

(a) `ptr = (char *)p + size(header(p)) - 8`

(b) `ptr = (char *)p + size(header(p)) - 4`

(c) `ptr = (int *)p + size(header(p)) - 2`

```
/* given a pointer to a valid block header or footer,
   returns the usage of the currect block,
   1 for allocated, 0 for free */
int allocated(void *hp)
{
  int result;

  --------------------;
  return result;
}
```

(a) `result = *(int *)hp & 1`

(b) `result = *(int *)hp & 0`

(c) `result = *(int *)hp | 1`

```
/* given a pointer to a valid block header,
   returns the pointer to the header of previous block in memory */
void *prev(void *hp)
{
  void *ptr;

  --------------------;
  return ptr;
}
```

(a) `ptr = (char *)hp - size((char *)hp - 4)`

(b) `ptr = (char *)hp - size((char *)hp - 4) + 4`

(c) `ptr = (int *)hp - size(hp)`