

Full Name: _____

CS 351 Fall 2009

Final Exam

December 9, 2009

Instructions:

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

1 (/10) :
2 (/12) :
3 (/12) :
4 (/12) :
5 (/12) :
TOTAL (/58) :

Problem 1. (10 points):

Multiple choice. For each question, circle the letter corresponding to the **single best** answer.

1. What policy decision may result in artificial external fragmentation?
 - (a) Quadruple word block alignment
 - (b) Deferred coalescing
 - (c) Address-based free list ordering
 - (d) Next-fit free block search
2. Which of the following is *not* a UNIX I/O system call?
 - (a) `pipe`
 - (b) `socket`
 - (c) `rio_readn`
 - (d) `select`
3. Which characteristic of well written code motivates the use of large cache blocks?
 - (a) temporal locality
 - (b) spatial locality
 - (c) small function size
 - (d) minimal coupling
4. What is a reasonable justification for using multiple processes as a concurrency model for an application *instead of* multiple threads?
 - (a) it is easier to share data amongst processes
 - (b) process context switches are more efficient
 - (c) thread concurrency is unpredictable
 - (d) a single misbehaved thread may crash the entire application
5. Which is the best argument *against* using multiplexed I/O for a concurrent server?
 - (a) the lack of true parallelism
 - (b) the inability to use file descriptors without blocking
 - (c) implicit barriers to data sharing
 - (d) the need to synchronize parallel control flows

Problem 2. (12 points):

Assume the following setup of virtual memory and cache:

- Memory is byte addressable
- Virtual addresses are 16 bits wide
- Physical addresses are 12 bits wide
- The page size is 128 bytes (2^7 bytes)
- The TLB is 2-way set associative, with 8 total lines
- The cache is 4-way set associative, with 4-byte blocks and 8 total lines

The TLB, cache, and partial page table contents are shown below:

TLB			
Index	Tag	Valid	PPN
0	78	1	0F
	4F	0	14
1	36	1	19
	60	0	0D
2	46	0	1F
	37	1	1D
3	68	1	15
	57	1	00

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	13	0	0A	07	1
01	04	0	0B	16	0
02	0C	1	0C	1D	0
03	0E	0	0D	10	0
04	19	0	0E	05	1
05	15	1	0F	0B	1
06	00	0	10	1E	1
07	17	0	11	01	1
08	12	0	12	06	1
09	1A	0	13	02	0

Cache						
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	164	0	CB	E2	0B	1F
	19E	1	BA	59	D8	3B
	115	1	C9	70	38	66
	076	0	56	49	97	37
1	092	0	68	FF	28	27
	19E	0	E8	54	FC	B1
	073	1	02	F8	0B	AF
	1A3	0	AF	76	60	D8

Based on the description above, answer the questions on the following two pages.

Part I (4 points)

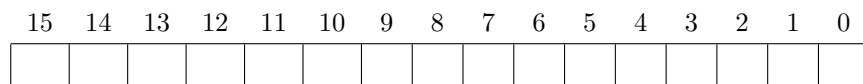
The diagram below shows the format of a virtual address. Label it with the fields (indicate their positions) that would be used to determine the following:

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:

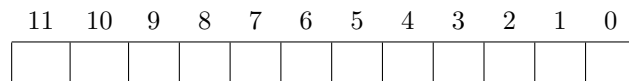
PPO : The physical page offset

PPN : The physical page number

CO : The cache block offset

CI : The cache set index

CT : The cache tag

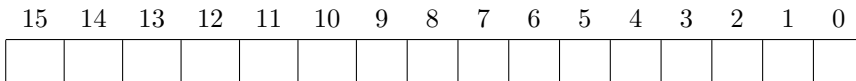


Part II (8 points)

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: **0x051F**

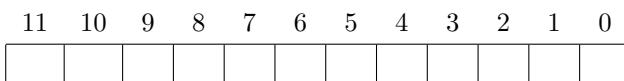
A. Virtual address format



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: **0x6CF0**

A. Virtual address format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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

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

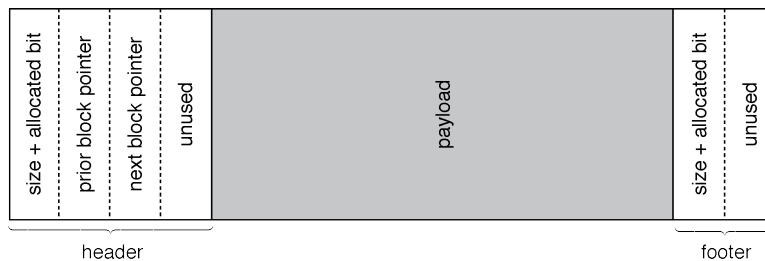
D. Physical memory reference

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

Problem 3. (12 points):

Consider an allocator that uses an explicit (circular) free list. Each block of memory is double-word aligned, and is prefaced by a header that occupies four words of memory: the first word contains the total block size (including payload and boundary tags) and allocated bit, and the second and third words contain pointers to the beginning of the prior and next free blocks (when the block is free). The fourth word in the header is unused. A footer is also present for each block, and consists of two words: the first contains the size and allocated bit and the second is unused.

The diagram below shows the structure of a block:



Answer the following questions based on this allocator, assuming a word size of **4 bytes**. If you need additional space, use the back of this sheet.

1. Given an incoming malloc request for 100 bytes, what is the minimum size of a free block than can be used to satisfy the request?
2. Given a pointer `curr` of type `(void *)` to the head of a free block, how would you obtain a pointer to the beginning of the next free block in the explicit list?

```
void *next =
```

3. Describe the steps needed to free an allocated block using immediate coalescing and a LIFO free list maintenance policy.
4. It is proposed that the footer be dropped from each block and deferred coalescing be adopted so as to reduce internal fragmentation. Is this a good idea? Why or why not?

Problem 4. (12 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 use their LSBs for the allocated bit and the rest of the bits for storing the size of the block. The allocator performs immediate coalescing, and splits blocks during allocation whenever possible (zero payload is not allowed).

Starting with the heap on the left below, show the resulting heap following each successive call to `malloc` and `free`. Use a *best-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 <code>malloc(6)</code> :	After <code>malloc(23)</code> :	After <code>free(0x8000504C)</code> :				
Addr.	Data	Addr.	Data	Addr.	Data	Addr.	Data
0x80005068	0x00000014	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	0x00000014	0x...5058		0x...5058		0x...5058	
0x...5054	0x00000011	0x...5054		0x...5054		0x...5054	
0x...5050	---	0x...5050		0x...5050		0x...5050	
0x...504c	---	0x...504c		0x...504c		0x...504c	
0x...5048	0x00000011	0x...5048		0x...5048		0x...5048	
0x...5044	0x00000010	0x...5044		0x...5044		0x...5044	
0x...5040	---	0x...5040		0x...5040		0x...5040	
0x...503c	---	0x...503c		0x...503c		0x...503c	
0x...5038	0x00000010	0x...5038		0x...5038		0x...5038	
0x...5034	0x00000015	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	0x00000015	0x...5024		0x...5024		0x...5024	
0x...5020	0x00000024	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	0x00000024	0x80005000		0x80005000		0x80005000	

Problem 5. (12 points):

Briefly (in a sentence or two) describe the purpose and semantics of each of the following system calls in the context of a program that implements a network server. When significant, you should describe the value returned. (8 points)

- socket:

- bind:

- listen:

- accept:

After explaining these system calls to a friend, she proceeds to write the following code for a very simple echo server . It doesn't work, however – what's wrong, and how would you fix it? Note that the echo function isn't shown, but you can assume it's correct. (4 points)

```
int main() {
    int fd_skt, clen;
    struct sockaddr_in saddr, caddr;
    fd_skt = socket(AF_INET, SOCK_STREAM, 0);           // 1. socket
    saddr.sin_family      = AF_INET;
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);
    saddr.sin_port        = htons(80);
    bind(fd_skt, (struct sockaddr *)&saddr, sizeof(saddr)); // 2. bind
    listen(fd_skt, SOMAXCONN);                          // 3. listen
    clen = sizeof(caddr);
    accept(fd_skt, (struct sockaddr *)&caddr, &clen);    // 4. accept
    echo(fd_skt);
    close(fd_skt);
}
```