Name	

UW CSE 351, Summer 2013
Final Exam Solution

9:40am - 10:40am, Friday, 23 August 2013

Instructions:

- Make sure that your exam is not missing any of the 11 pages, then write your full name and UW student ID on the front.
- **Read over the entire exam before starting to work on the problems!** The last page is a reference page that you may tear off to use during the exam; it does not have to be turned in.
- Do not spend too much time on a problem if there are other easy problems that you haven't solved yet.
- Feel free to use the backs of pages for working on problems, but *write your answers in the space provided below each problem*. If you make a mess, clearly indicate your final answer. Be sure to answer all parts of all questions.
- For free-form answers, keep you answers brief and feel free to use short phrases. ***Full sentences NOT required.***
- **No books, notes, or electronic devices may be used during the exam.** You may not communicate with other students during the exam, but please ask the instructor if you need clarification for some problem.
- Thanks for a fun quarter of 351! Cue the final exam music!

Part	Awarded	Possible
Part 1		**20**
Part 2		**32**
Part 3		**18**
Part 4		**30**
Total		**100**

1 Concepts (20 points)

Answer the following questions ***briefly***. ***Full sentences NOT required.***

(a) (5 points) List the two important illusions that the process abstraction provides to programs. For each illusion, list a mechanism involved in its implementation.

1. Logical control flow: the process executes as if it has complete control over the CPU. The OS implements this by interleaving execution of different processes via context-switching (exceptional control flow...).
2. Private linear address space: the process executes as if it has access to a private contiguous memory the size of the virtual address space.

(b) (5 points) One purpose of virtual memory is to allow programs to use more memory than is available in the physical memory, by storing some parts on disk transparently. Name one *other* useful things that can be done with the virtual memory system.

There a number of possibilities. Popular choices included:
*1. Sharing of a single physical page in multiple virtual address spaces (*e.g.*, shared library code).*
*2. Memory protection mechanisms (*e.g.*, page-granular read/write/execute permissions or protecting one processes memory from another).*

(c) (5 points) Does a TLB (Translation Lookaside Buffer) miss always lead to a page fault? Why or why not?

No. The TLB caches page table entries. After a TLB miss, we do an in-memory page table lookup. A page fault occurs if the page table entry is invalid. (Some people confused the presence of a mapping in the page table with the presence of a page in cache, but these are not strongly linked.)

(d) (5 points) Name one difference between Java references and C pointers.

There are a number of differences. The most popular included:
1. C allows pointer arithmetic; Java does not.
2. C pointers may point anywhere (including the middles of memory objects); Java references point only to the start of objects.
3. C pointers may be cast arbitrarily (even to non-pointer types); casts of Java references are checked to make sure they are type-safe.

2 Cache Flow (32 points)

You are interviewing for a lucrative position with Accelerated Throughput Memories, Inc. (ATM), a company known for its fast cache hardware. Answer the following questions to get the job.

ATM is evaluating two cache designs for machines with 32-bit physical addresses. The Fast Data and Instruction Cache (FDIC) and the Super-Efficient Cache (SEC) both use the conventional *write-back* and *write-allocate* policies and a true *least-recently-used* replacement policy. The (partial) geometry of these caches is as follows:

Name	Cache Size (bytes)	Block/Line Size (bytes)	Sets	Set Associativity
FDIC	1024	32	*16*	2
SEC	*512*	16	8	4

(a) (2 points) Fill in the number of sets in the FDIC cache and the size (in bytes) of the SEC cache in the table above. Use decimal notation. (The powers of 2 and exponent rules on page 11 may be useful.)

(b) (6 points) Write the number of tag bits, set index bits, and block/line offset bits for each cache in the table below. Recall the physical address size is 32 bits. (The powers of 2 and exponent rules on page 11 may be useful.)

Cache	Number of Bits		
	Tag	Set Index	Block/Line Offset
FDIC	*23*	*4*	*5*
SEC	*25*	*3*	*4*

(c) (4 points) Briefly, why does *write-back* often have better performance than *write-through*? ***Full sentences NOT required.***

Write-back stores updated values back to the next level of the memory hierarchy only when a cache line/block is evicted. Write-through stores back to the next level on every write operation. Write-back often has better performance because of locality *of write operations: more than one write to the cache line is likely to occur before it gets evicted, thus reducing the number of next-level memory operations. You had to mention locality in addition to the policies to get full credit.*

Part 2 continues on the next page.

(d) You will calculate the data-cache ***miss rates*** of two code sections on each of the two caches.

- Consider data accesses only. Instructions are handled by a separate cache.
- There is only one level of data cache in the system.
- For each combination of code and cache, the cache starts empty.
- Both code sections use the following declarations:

```
int values[4][128];
int i, j;
```

- All data except arrays are stored in registers; accesses to them never affects the cache.
- The `values` array starts at address 0x0.
- The system's page size is 4096 bytes.
- `sizeof(int) == 4`

Showing calculations or explanations for your answers to the following four questions is ***NOT required****, but could help achieve partial credit if your answers are incorrect.*

(i) (8 points) What is the miss rate of the following code on the FDIC cache? on the SEC cache? Write your answers as fractions.

```
// code section A
int prod = 1;
for (i = 0; i < 8; i++) {
  for (j = 0; j < 128; j++) {
    prod = prod * values[i % 4][j];
  }
}
```

FDIC miss rate $= \frac{1}{8}$ SEC miss rate $= \frac{1}{4}$

Note that this nested loop accesses every element in the array twice, since i *ranges from 0 through 7 and is used as an index* `i % 4`*:* $i = 0..3, i = 4..7$*. However, the array is of size 2048 bytes = 4 * 128 * sizeof(int), which is larger than either cache, so on the second run through the array, none of the needed lines will be there, since later accesses to other lines evict them. The sizes and associativities of FDIC and SEC thus make no difference.*

FDIC has 32-byte cache lines, which fit 8 ints: every 8th access is a miss; the following 7 accesses hit on the same cache line. SEC has 16-byte cache lines, which fit 4 ints: every 4th access is a miss; the following 3 accesses hit on the same cache line.

(ii) (2 points) The three types of cache misses are cold/compulsory misses, conflict misses, and capacity misses. What type or types of cache misses occur in *both* caches when executing code section A?

Both cold/compulsory misses (the caches starts empty) and capacity misses occur. Capacity misses occur on the second access to each array element because the array (the working set) is larger than the cache.

Part 2 continues on the next page.

(iii) (8 points) What is the miss rate of the following code on the FDIC cache? on the SEC cache? Write your answers as fractions.

```
// code section B
for (i = 0; i < 128; i++) {
  values[0][i] = values[1][i] + values[2][i] + values[3][i];
}
```

FDIC miss rate $= \frac{1}{1}$ SEC miss rate $= \frac{1}{4}$

Here associativity comes into play. Each loop iteration accesses all 4 elements in a column of the array. For both caches, each of these 4 is in a separate cache line and all 4 of these lines (at addresses differing by 512) map to the same set in the cache.

Regardless of what order we perform the accesses within an iteration, this means that, in FDIC, with an associativity of 2, the latter two accesses will always force the cache lines for the earlier two accesses to be evicted. On the next iteration, the former are needed first, but are not present, so each will take and miss and they will will evict the latter two to be placed in the cache, and so on. All accesses will miss.

SEC's associativity of 4 means that the cache lines for all 4 elements in an iteration can live in the cache simultaneously. Thus, after a cold miss for each, they are each reused on the next 3 iterations (3 hits each) and then never used again.

(iv) (2 points) The three types of cache misses are cold/compulsory misses, conflict misses, and capacity misses. What single type of cache misses accounts for the majority of misses in the FDIC cache when executing code section B?

Conflict misses, as discussed above. Each cache line is used 8 times, but every access misses. The first miss on each line is a cold miss; subsequent misses (7 each) are conflict misses. These are not capacity misses because the working set size is small (4 cache lines = 128 bytes), much smaller than the cache. We never return to an element after the one iteration that uses it; we never return to a cache line after the 8 contiguous loop iterations that use it.

3 Virtual Mystery (18 points)

You may detach this page for reference. You do not need to turn it in.

We just dug up a dusty old computing system that uses virtual memory, but we could not find much information about the paging system. We know from the ISA that the machine has 16-bit words and uses 16-bit virtual addresses and we discovered via some tinkering that it uses 14-bit physical addresses. *Note that* `sizeof(short) == 2` *bytes.*

To learn more about the virtual memory system, we ran the following program in a process P. It uses `valloc`, which is just like `malloc`, but allocates the payload to be page-aligned. Assume that all local variables are stored in registers; only the array is stored in memory.

```
short* numbers = (short*)valloc(1024 * sizeof(short));
short i;
for (i = 0; i < 1024; i++) {
  numbers[i] = i;
}
// examine physical memory here...
```

We verified that no disk accesses occurred during process P's execution, so we know all of the virtual pages P used must be mapped to physical pages. Just before the program finished, we used a special hardware tool to dump the contents of *physical* memory. Here is part of what we found. (Note these are all hexadecimal values!)

Partial Contents of Physical Memory							
Address	**Contents**	**Address**	**Contents**	**Address**	**Contents**	**Address**	**Contents**
0x0a00	0x0010	0x0a20	0x0000	0x0a40	0x000e	0x0a60	0x0040
0x0a02	0x0011	0x0a22	0x0001	0x0a42	0x00e0	0x0a62	0x0041
0x0a04	0x0012	0x0a24	0x0002	0x0a44	0x0e00	0x0a64	0x0042
0x0a06	0x0013	0x0a26	0x0003	0x0a46	0xe000	0x0a66	0x0043
0x0a08	0x0014	0x0a28	0x0004	0x0a48	0x0e00	0x0a68	0x0044
0x0a0a	0x0015	0x0a2a	0x0005	0x0a4a	0x00e0	0x0a6a	0x0045
0x0a0c	0x0016	0x0a2c	0x0006	0x0a4c	0x000e	0x0a6c	0x0046
0x0a0e	0x0017	0x0a2e	0x0007	0x0a4e	0x00e0	0x0a6e	0x0047
0x0a10	0x0018	0x0a30	0x0008	0x0a50	0x0e00	0x0a70	0x0048
0x0a12	0x0019	0x0a32	0x0009	0x0a52	0xe000	0x0a72	0x0049
0x0a14	0x001a	0x0a34	0x000a	0x0a54	0x0e00	0x0a74	0x004a
0x0a16	0x001b	0x0a36	0x000b	0x0a56	0x00e0	0x0a76	0x004b
0x0a18	0x001c	0x0a38	0x000c	0x0a58	0x000e	0x0a78	0x004c
0x0a1a	0x001d	0x0a3a	0x000d	0x0a5a	0x00e0	0x0a7a	0x004d
0x0a1c	0x001e	0x0a3c	0x000e	0x0a5c	0x0e00	0x0a7c	0x004e
0x0a1e	0x001f	0x0a3e	0x000f	0x0a5e	0xe000	0x0a7e	0x004f

Assume that, except for physical addresses `0x0a40-0x0a5f`, the physical memory shown above represents part of the `numbers` array in process P.

Part 3 continues on the next page.

(a) (6 points) Given these assumptions and the contents of physical memory, what is the ***largest*** page size the system could be using? *Briefly* explain your reasoning. ***Full sentences NOT required.*** *NOTE: questions (b) and (c) do not depend on this answer.*

The `numbers` *array is allocated contiguously and filled with successive short integer values starting at zero. We can see parts of the array in physical memory and we know element zero must be page-aligned due to* `valloc`*. Elements 0-15 appear at physical addresses 0x0a20-0x0a3f, so 0x0a20 must be page aligned. It's largest power-of-2 divisor is 32, so pages must not be bigger than 32 bytes (but they could be smaller). Elements 16-31 appear at physical addresses 0x0a00-0x0a1f and 64-79 appear at 0x0a60-0x0a7f. The longest contiguous run of elements is 16 elements, or 32 bytes, thus pages can be no larger than 32 bytes.*

Some people derived this answer via other (broken) means (e.g., arithmetic on address sizes, some things that I could not decipher). Others suggested 2^{14} *as the maximum page size, since pages must be small enough to fit in physical memory. Both of these answers received partial credit. Note that* 2^{16} *is not a reasonable page size since not even one page can fit in physical memory. (In fact,* 2^{14} *has problems too, though more subtle: space is needed in memory for the page table – can't use it all for storage!).*

(b) (4 points) Assume that the system actually uses 16-byte pages and an 8-way set-associative TLB with 32 total entries. Label the bits of a virtual address below as they are used to determine the following:

VPO Virtual Page Offset: *16-byte pages require* $16 = 2^4$ *offsets, so 4 bits.*

VPN Virtual Page Number: *address size minus VPO bits*

TI TLB Index: *A 32-entry 8-way associative TLB has* $32/8 = 4 = 2^2$ *sets, so 2 bits.*

TT TLB Tag: *VPN bits minus TI bits*

VPN												*VPO*			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TT										*TI*					

(c) (8 points) Assume 16-byte pages and assume the `numbers` array begins at *virtual* address 0x00004000. Using the contents of physical memory, reconstruct as many valid entries in process P's page table as you can. Write page table entries in any order, using the notation $VPN \rightarrow PPN$ to mean that the page table entry for virtual page number VPN is valid and maps to physical page number PPN. Write page numbers in hexadecimal.

The key here is to recognize where parts of the array live in physical memory by the element contents (and its correspondence with index, i.e.*,* `numbers[i] = i`*). The array starts at virtual address 0x4000, so the zero element, stored in physical address 0x0a20, establishes a mapping. Since page offsets are 4 bits, you can just drop the least-significant hex digit from an address to get the page number. I tried to give credit for answers to (c) that matched up with incorrect VPO/VPNs in (b). Some people listed just the mappings of the top of each column. There are 6 recoverable mappings: 0x400* $\rightarrow$ *0x0a2, 0x401* $\rightarrow$ *0x0a3, 0x402* $\rightarrow$ *0x0a0, 0x403* $\rightarrow$ *0x0a1, 0x408* $\rightarrow$ *0x0a6, 0x409* $\rightarrow$ *0x0a7*

4 Memory Allocation and Movement (30 points)

You may detach this page for reference. You do not need to turn it in.

Some garbage collectors for languages like Java move allocated objects from one address in memory to another address in memory to defragment the heap. To do this safely, they must update all references to moved objects to refer to the object's new address. Movement is generally unsafe in languages like C, because it is not always possible to determine exactly what values are pointers.

Suppose we are using a special dialect of C that makes it possible to find all pointers and move allocated blocks safely. We would like to extend a memory allocator like the one in Lab 5 with the ability to move allocated blocks in memory to reduce fragmentation.

Movement Algorithm Given an `old` allocated block to move and a `new` allocated block of sufficient size into which to move the `old`, we must (1) copy the payload of the `old` block into the `new` block, (2) replace all pointers that point to the `old` payload with pointers to the `new` payload, and, finally, (3) free the `old` block.

Your Tasks You will complete three small C functions to help find and replace pointers to the `old` payload with pointers to the `new` payload. **Minor coding mistakes will not affect your score since you do not have a compiler at hand. Use pseudocode if short on time.**

Provided Code and Environment You may use the following functions and definitions in your code. Most should be familiar from Lab 5. The machine has a 64-bit word and address size. Blocks are 8-byte aligned. When allocated, blocks contain an 8-byte header, storing their size and allocation tags, and a payload. You will only need to use the headers of blocks when they are free.

We added functions that return the `heap_start()` and `heap_end()` addresses. We also added the magic `holds_pointer` function, which determines whether or not a given memory location holds a pointer to some other memory location.

```
#define WORD_SIZE 8
// Unscaled pointer arithmetic (same as UNSCALED_POINTER_ADD ...)
#define P_ADD(x,y) ((void*)((char*)(x) + (char*)(y)))
#define P_SUB(x,y) ((void*)((char*)(x) - (char*)(y)))
#define TAG_USED 1
#define SIZE(x) ((x) & ~3)
struct BlockInfo {
  size_t sizeAndTags;
  // other fields not needed today
};
typedef struct BlockInfo BlockInfo;

// Returns the address of the first block in the heap.
BlockInfo* heap_start();
// Return the address of the special end word of the heap.
BlockInfo* heap_end();
// Return 1 if memory at addr holds a pointer, 0 otherwise.
int holds_pointer(void* addr);
```

Part 4 continues on the next page.

(a) (12 points) Implement a function `update_heap` that takes pointers to the `old` and `new` versions of the moved block and scans over the heap block by block from `heap_start()` to `heap_end()`, calling `update_block` on every *allocated* block to replace any `old` pointers with `new` pointers. Do not allocate or free any blocks. Do not change headers or footers. Do not use the free list. *Our solution adds 5 lines; 2 lines hold single curly braces.*

```
// Implemented in later question.
void update_block(BlockInfo* b, BlockInfo* old, BlockInfo* new);

void update_heap(BlockInfo* old, BlockInfo* new) {
  BlockInfo* b;   // A pointer for iterating over heap blocks.
  // -- YOUR CODE HERE ------------------------------------

  // I ignored casting issues and improper uses of void*
  // (even in my own solution) unless they dropped bits or would
  // compile and do (improperly) scaled pointer arithmetic.

  // Iterate over the heap, block by block.
  for (b = heap_start(); b < heap_end();
       b = P_ADD(b, SIZE(b->sizeAndTags))) {
    // Update each allocated block.
    if (b->sizeAndTags & TAG_USED) {
      update_block(b, old, new);
    }
  }
}
```

(b) (8 points) Implement a function `points_into` that takes a pointer, `ptr`, a starting address, `start`, and a length (in bytes), `nbytes`. Return 1 if `ptr` points to any address in the `nbytes`-long range of addresses starting at `start`. Return 0 otherwise. Assume `start` is at least `nbytes` before the end of the address space. *Our solution adds one (long) line.*

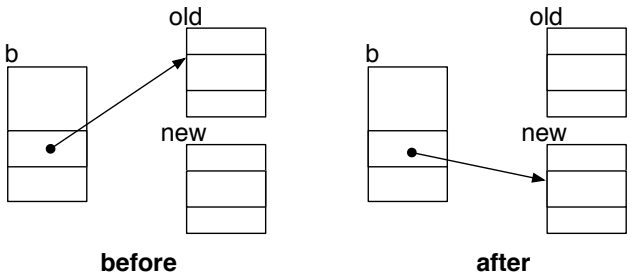
```
int points_into(void* ptr, void* start, size_t nbytes) {
  // -- YOUR CODE HERE ------------------------------

  // I ignored casting issues and improper uses of void*
  // (even in my own solution) unless they dropped bits or would
  // compile and do (improperly) scaled pointer arithmetic.

  return ptrByte >= startByte && ptrByte < P_ADD(startByte, nbytes);
```

Part 4 continues on the next page.

(c) (10 points) Implement a function `update_block` that takes a pointer to a block, `b`, and pointers to blocks `old` and `new`, the source and destination of the movement operation. The function should update all pointers stored in the payload of `b` that point to the `old` payload so they point to the `new` payload. The provided code iterates over the payload of `b`, word by word. Your job is to update a word in the payload if it holds a pointer to an address in the payload of the `old` block (see `holds_pointer` and `points_into`), replacing it with a pointer to the corresponding location in the `new` block. Assume the `old` and `new` blocks are the same size. *Our solution adds 3 lines (4 lines when wrapped to fit).*

Example 1 If `old`'s payload starts at 0x8000 and `new`'s payload starts at 0xc000, and a pointer in the payload of `b` points to 0x8008, then this pointer should be replaced with a pointer to 0xc008.

Example 2




```
void update_block(BlockInfo* b, BlockInfo* old, BlockInfo* new) {
  // Start address and size of the old payload
  void*  oldPayloadStart = P_ADD(old, WORD_SIZE);
  size_t oldPayloadSize  = SIZE(*((size_t*)old)) - WORD_SIZE;
  void*  newPayloadStart = P_ADD(new, WORD_SIZE);
  void** slot;  //  pointer for iterating over payload words
  for (slot = P_ADD(b, WORD_SIZE);
       slot < P_ADD(b, SIZE(b->sizeAndTags));
       slot = P_ADD(slot, WORD_SIZE)) {
  // -- YOUR CODE HERE ------------------------------------
    // I ignored casting issues and improper uses of void*
    // (even in my own solution) unless they dropped bits or would
    // compile and do (improperly) scaled pointer arithmetic.

    // If memory at address slot is used store a pointer
    // and the pointer stored there points into the old payload
    // then update the pointer.
    // Excessive or insufficient dereferencing were common problems.
    if (holds_pointer(slot)
        && points_into(*slot, oldPayloadStart, oldPayloadSize)) {
      // Old pointer may point anywhere in old payload.
      // Point to corresponding offset into new payload.
      *slot = P_ADD(newPayloadStart, P_SUB(*slot, oldPayloadStart));
    }
  }
}
```

5 Just for Fun

Write a pun using 351 material.

I won't give away the puns suggested, in order to let their creators enjoy maximum effect when using them on unsuspecting 351 students.

6 Reference

You may detach this page for reference. You do not need to turn it in.

Hexadecimal

Hex	Binary	Decimal
0x0	0000	0
0x1	0001	1
0x2	0010	2
0x3	0011	3
0x4	0100	4
0x5	0101	5
0x6	0110	6
0x7	0111	7
0x8	1000	8
0x9	1001	9
0xa	1010	10
0xb	1011	11
0xc	1100	12
0xd	1101	13
0xe	1110	14
0xf	1111	15

Powers of Two

2^0 = 0x0001 = 1	2^7 = 0x0080 = 128
2^1 = 0x0002 = 2	2^8 = 0x0100 = 256
2^2 = 0x0004 = 4	2^9 = 0x0200 = 512
2^3 = 0x0008 = 8	2^{10} = 0x0400 = 1024
2^4 = 0x0010 = 16	2^{11} = 0x0800 = 2048
2^5 = 0x0020 = 32	2^{12} = 0x1000 = 4096
2^6 = 0x0040 = 64	2^{13} = 0x2000 = 8192

$2^a * 2^b = 2^{a+b}$
$2^a / 2^b = 2^{a-b}$