**Andrew login ID:** _____

**Full Name:** _____

# CS 15-213, Fall 2004

# Exam 2

Tuesday November 16, 2004

**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 74 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 use a calculator, but not a laptop, PDA, or any wireless devices. Good luck!

| |
|---|
| 1 (10): |
| 2 (12): |
| 3 (12): |
| 4 (10): |
| 5 (6): |
| 6 (12): |
| 7 (12): |
| TOTAL (74): |

## Problem 1. (10 points):

This problem tests your understanding of programming in the presence of signals – in particular, race conditions. The questions are based on the library below which maintains a sorted list of integers.

The programs are compiled without optimization: the machine instructions executed are a straightforward translation of the C code. In particular, statements are never reordered.

Remember that `malloc` is not guaranteed to zero the memory it returns.

```
1  struct list {
2      struct list *next;
3      int value;
4  };
5  static struct list global_header = { NULL, 0 };

7  /* Find the largest datum < value */
8  static struct list *find_prev(int value) {
9      struct list *prev = &global_header;
10     while(prev->next) {
11         if(prev->next->value >= value) break;
12         prev = prev->next;
13     }
14     return prev;
15 }

17 /* Add 'value' into the list */
18 void add_datum(int value) {
19     struct list *prev = find_prev(value);
20     struct list *node = malloc(sizeof(*node));
21     struct list *next = prev->next;
22
23     if(!node) return;
24
25     prev->next = node;
26     node->next = next;
27     node->value = value;
28 }

30 /* Remove the first datum >= 'value' */
31 void remove_datum(int value) {
32     struct list *prev, *node, *next;
33
34     prev = find_prev(value);
35     node = prev->next;
36     if(!node) return;
37     next = node->next;
38
39     prev->next = next;
40     free(node);
41 }
```

```
43  /* Count the number of data in the list */
44  int count(void) {
45      struct list *node = global_header.next;
46      int cnt = 0;
47      while(node) {
48          cnt++;
49          node = node->next;
50      }
51      return cnt;
52  }
```

---

A.

```
1   void handler(int sig) {
2       printf("we have %d elements\n", count());
3   }
4   int main() {
5       char buf[256];
6       signal(SIGUSR2, handler);
7
8       while(fgets(buf, sizeof(buf), stdin)) {
9           add_datum(atoi(buf));
10      }
11      return 0;
12  }
```

Does this have a race condition that can cause the program to crash? Why or why not? (2 points)

B.

```
1   void handler(int sig) {
2       printf("we have %d elements\n", count());
3   }
4   int main() {
5       char buf[256];
6       int n = 100; while(n > 0) { add_datum(n--); }
7       signal(SIGUSR2, handler);
8
9       while(fgets(buf, sizeof(buf), stdin)) {
10          remove_datum(atoi(buf));
11      }
12      return 0;
13  }
```

Does this have a race condition that can cause the program to crash? Why or why not? (2 points)

C.

```
1  void handler(int sig) {
2      add_datum(random());
3  }
4  int main() {
5      char buf[256];
6      signal(SIGUSR2, handler);
7
8      while(fgets(buf, sizeof(buf), stdin)) {
9          remove_datum(atoi(buf));
10     }
11     return 0;
12 }
```

This code has a race condition. While it won't cause a crash, it could cause a memory leak. Why? (2 points)

D. Explain one way to avoid all the crashes and undesired behaviour in all the examples above. (4 points)

## Problem 2. (12 points):

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system with the following parameters:

- Virtual addresses are 20 bits wide.
- Physical addresses are 18 bits wide.
- The page size is 4096 bytes.
- The TLB is 2-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal**.

| TLB | | | |
|---|---|---|---|
| Index | Tag | PPN | Valid |
| 0 | 16 | 13 | 1 |
| | 1B | 2D | 1 |
| 1 | 10 | 0F | 1 |
| | 0F | 1E | 0 |
| 2 | 1F | 01 | 1 |
| | 11 | 1F | 0 |
| 3 | 03 | 2B | 1 |
| | 1D | 23 | 0 |
| 4 | 06 | 08 | 1 |
| | 0F | 19 | 1 |
| 5 | 0A | 09 | 1 |
| | 1F | 20 | 1 |
| 6 | 02 | 13 | 0 |
| | 18 | 12 | 1 |
| 7 | 0C | 0B | 0 |
| | 1E | 24 | 0 |

| Page Table | | | | | |
|---|---|---|---|---|---|
| VPN | PPN | Valid | VPN | PPN | Valid |
| 00 | 17 | 1 | 10 | 26 | 0 |
| 01 | 28 | 1 | 11 | 17 | 0 |
| 02 | 14 | 1 | 12 | 0E | 1 |
| 03 | 0B | 0 | 13 | 10 | 1 |
| 04 | 26 | 0 | 14 | 2D | 0 |
| 05 | 13 | 1 | 15 | 1B | 0 |
| 06 | 0F | 1 | 16 | 31 | 1 |
| 07 | 10 | 1 | 17 | 12 | 0 |
| 08 | 1C | 0 | 18 | 23 | 1 |
| 09 | 25 | 1 | 19 | 04 | 0 |
| 0A | 31 | 0 | 1A | 0C | 1 |
| 0B | 16 | 1 | 1B | 2B | 1 |
| 0C | 01 | 1 | 1C | 1E | 0 |
| 0D | 15 | 1 | 1D | 3E | 1 |
| 0E | 0C | 0 | 1E | 27 | 1 |
| 0F | 14 | 0 | 1F | 18 | 1 |

## Part 1

1. The diagram below shows the bits of a virtual address. Please indicate the locations of the following fields by placing an 'X' in the corresponding boxes of that field's row. For example, if the virtual page offset were computed from the 2 most significant bits of the virtual address, you would mark the 'O' (offset) column as shown:

| | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| O | X | X | | | | | | | | | | | | | | | | | | |

*O* The virtual page **o**ffset
*N* The virtual page **n**umber
*I* The TLB **i**ndex
*T* The TLB **t**ag

| | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| O | | | | | | | | | | | | | | | | | | | | |
| N | | | | | | | | | | | | | | | | | | | | |
| I | | | | | | | | | | | | | | | | | | | | |
| T | | | | | | | | | | | | | | | | | | | | |

2. The diagram below shows the format of a physical address. Please indicate the locations of the following fields by placing an 'X' in the corresponding boxes of that field's row.

*O* The physical page **o**ffset
*N* The physical page **n**umber

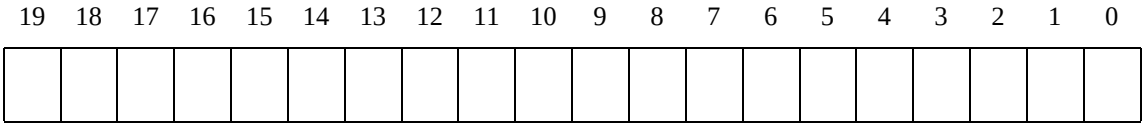
| | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| O | | | | | | | | | | | | | | | | | | |
| N | | | | | | | | | | | | | | | | | | |

## Part 2

For the given virtual addresses, please indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs. If there is a page fault, enter "-" for "PPN" and leave the physical address blank.

**Virtual address**: `0xD8AC3`

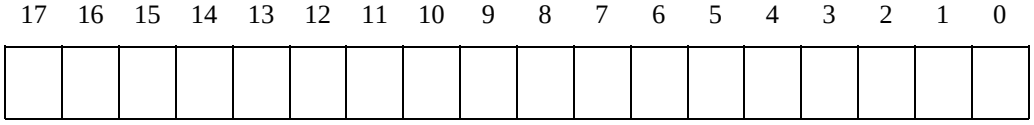
1. Virtual address (one bit per box)




2. Address translation

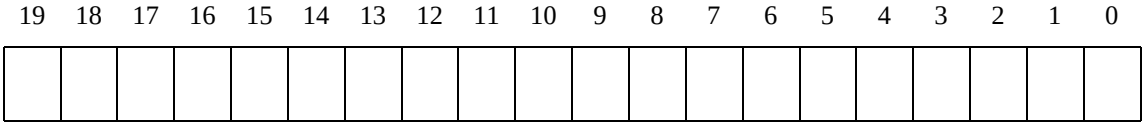
| Parameter | Value | Parameter | Value |
|---|---|---|---|
| VPN | 0x | TLB Hit? (Y/N) | |
| TLB Index | 0x | Page Fault? (Y/N) | |
| TLB Tag | 0x | PPN | 0x |

3. Physical address(one bit per box)




**Virtual address**: `0x1665D`

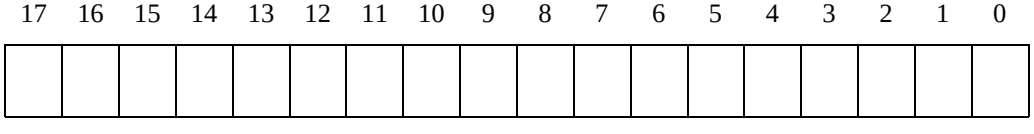
1. Virtual address (one bit per box)




2. Address translation

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

3. Physical address(one bit per box)



## Problem 3. (12 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
  int i = 0;
  pid_t pid1, pid2;

  if((pid1 = fork()) == 0) {
    i++;
    if((pid2 = fork()) == 0) {
      i++;
      printf("i: %d\n", ++i);
      exit(0);
    }
    printf("i: %d\n", i);
  }
  else {
    if(waitpid(pid1, NULL, 0) > 0) {
      printf("i: %d\n", ++i);
    }
  }
  printf("i: %d\n", ++i);
  exit(0);
}
```

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

| | | | | | |
|---|---|---|---|---|---|
| i: 1 | i: 1 | i: 1 | i: 3 | i: 1 | i: 1 |
| i: 3 | i: 1 | i: 2 | i: 3 | i: 2 | i: 2 |
| i: 2 | i: 2 | i: 3 | i: 3 | i: 1 | i: 3 |
| i: 1 | i: 3 | i: 2 | i: 1 | i: 2 | i: 4 |
| i: 2 | | i: 3 | i: 2 | i: 3 | i: 2 |
| | | i: 2 | | | i: 3 |

## Problem 4. (10 points):

This problem tests your understanding of the the cache organization and performance. To simplify your reasoning, you may assume the following:

1. `sizeof(int) = 4`
2. x begins at memory address 0 and is stored in row-major order.
3. The cache is initially empty.
4. The only memory accesses are to the entries of the array x. All variables are stored in registers.
5. All code is compiled with -O0 flag (no optimizations).

Consider the following C code:

```
int x[2][128];
int i;
sum = 0;

for (i = 0; i < 128; i ++){
    sum += x[0][i] * x[1][i];
}
```

**Case 1**

1. Assume your cache is a 512-byte direct-mapped data cache with 16-byte cache blocks. What is the cache **miss rate**? (3 pts)

miss rate = _____%

2. If the cache were twice as big, what would be the miss rate? (2 pts)

miss rate = _____%

**Case 2**

1. Assume your cache is 512-byte 2-way set associative using an LRU replacement policy with 16-byte cache blocks. What is the cache miss rate? (3 pts)

miss rate = _____%

2. Will larger **cache size** help to reduce the miss rate? (Yes / No) (1 pt)

3. Will larger **cache line** help to reduce the miss rate? (Yes / No) (1 pt)

## Problem 5. (6 points):

True/False:

1. A linker needs access to the source code to determine what global variables are referenced by a code file.
2. After a fork occurs, the memory updates made by the child process can affect the behavior of the parent process.
3. After a fork occurs, the file reads made by the child process can affect the behavior of the parent process.
4. When the `kill` function is invoked, a termination signal will be sent to the indicated process.
5. The availability of pointer casting in C makes it impossible for a garbage collector to identify all inaccessible data.
6. Implementing the free list as a doubly linked list makes it possible to implement the `free` operation in $O(1)$ time while still doing block coalescing.

## Problem 6. (12 points):

This problem tests your understanding of basic cache operations. A frequent operation in image processing programs is to convolve a large array (= the image) with a small, constant matrix (= the kernel). Filter operations such as sharpening, blurring, edge-enhancement can be implemented by choosing the kernel elements. In the simplest case, the kernel is a 3x3 matrix.

For this problem, you should assume a very large $N \times N$ image array of `unsigned short`'s, where $N$ happens to be an integral power of 2. The inner loop uses two int-pointers $src$ and $dst$ that scan the image array. There are two arrays: $src$ is scanning the source image while $dst$ is pointing to the resulting image after the convolution kernel has been applied. Thus the inner loop looks like this:

```
...
   unsigned short *src, *dst;
...
  { int new_pix;

    /* compute one pixel */
    new_pix  = src[     0] * A;

    new_pix += src[   - N] * B;
    new_pix += src[-1    ] * B;
    new_pix += src[ 1    ] * B;
    new_pix += src[   + N] * B;

    new_pix += src[-1 - N] * C;
    new_pix += src[ 1 - N] * C;
    new_pix += src[-1 + N] * C;
    new_pix += src[ 1 + N] * C;

    /* save the new pixel */
    *dst = new_pix;

    dst++;
    src++;
  }
...
```

This code utilizes the fact that many 3x3 kernels have rotational symmetry and have only 3 distinct values $A, B, C$. You should assume these values are kept in three registers. Likewise, the pointers $src$ and $dst$ are also stored in registers, as are the variables needed to control the inner loop. The target machine is fairly old and uses a write-through cache with no-write-allocate policy. Therefore, you do *not* need to worry about the write operations to the destination image array.

Each cache line on Harry's machine holds 4 `unsigned shorts` (8 Bytes). The cache size is 16 KBytes, which is too small to hold even one row of the image. Hint: each row has $N$ elements, where $N$ is a power of 2.

Figure 1 shows how this filter is scanning the source image array. The thick vertical bars represent the boundaries of cache lines: four consecutive horizontal squares are one cache line. The convolution kernel is represented by the 9 squares that are not marked with an X.

The 2 kernels shown in Figure 1 represent 2 successive iterations (case A and B) through the inner loop. The $src$ pointer is incremented one cell at a time and moves from left to right in these pictures.

You shall mark each of the 9 squares those with either a 'H' or a 'M' indicating if the corresponding memory read operation hits (H) or misses (M) in the cache. Cells that contain an X do not belong to the convolution step that is being computed and you should not mark these.

**Part 1**

In this part, assume that the cache is organized as a direct mapped cache. Please mark the left column in Figure 1 with your answer. The right column may be used as scratch while you reason about your answer. We will grade the left column only.

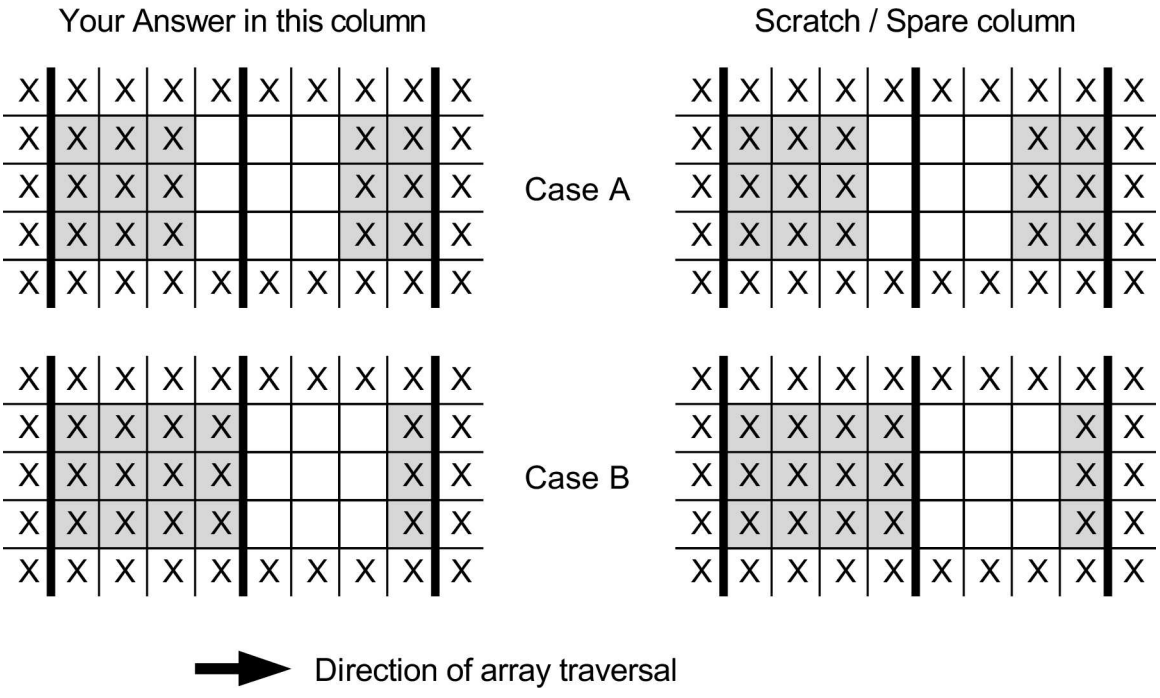



Figure 1: Convolution with a direct mapped cache

**Part 2**

In this part, assume a 3-way, set-associative cache with true Least Recently Used replacement policy (LRU). As in Part 1 of this question, please provide your answer by marking the empty squares of the left column in Figure 2 with your solution.

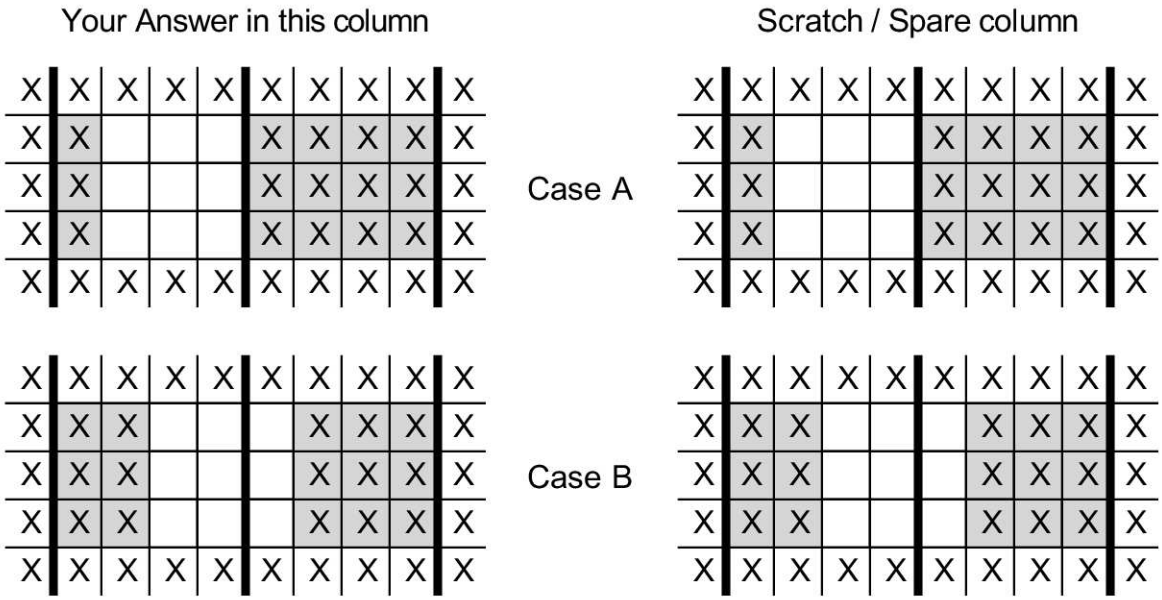



Figure 2: Convolution with a set associative cache

## Problem 7. (12 points):

This problem tests your knowledge of block coalescing.

Harry Q. Bovik is trying to write his `malloc` implementation. Assume his allocator does following:

- Uses an implicit list of blocks.
- All headers, footers, and pointers are 4 bytes in size
- All memory blocks have a size of at least 16 bytes: (one header + one footer + payload of at least 8).
- All memory blocks have their *total* size rounded up to a multiple of 8.
- Allocated blocks consist of a header, a payload, and a footer.
- All free block coalescing happens in the `free()` function.
- The heap contains prologue and epilogue blocks which are allocated in the `malloc` initialization function (guaranteed to be called before any call to `malloc()` or `free()`). The prologue and epilogue are never freed although the epilogue is shifted appropriately when the heap is extended.

**A. Simple Header (4 points)**

Let the header store the size of the block in bytes. Since the block size is always a multiple of 8, the three least significant bits of the size are always 0. Harry decides to use these bits to store allocation info. The low order bit is set to 0 when the block is free and 1 when the block is allocated. To correctly access the size of the block's payload, he must mask the lower 3 bits to 0's. The footer value is set to match the header value.

Harry wants to make his `free()` function coalesce newly freed blocks with any and all adjacent free blocks.

How many memory reads must `free()` do to *decide whether to coalesce* this block with the ones around it? Assume that the only piece of data currently available is the base pointer of the block that is currently being freed. _____

What is the maximum number of writes that `free()` must do to create the final coalesced block? _____

*This question continues on the next page.*

**B. Packed Header (4 points)**

Harry's implementation works, but it is a little slow. Like the clever 213 student that he is, he decides to try and decrease the number of memory accesses made by his program by utilizing all three free bits in the header:

- The header consists of the payload size, in bytes, OR'ed with the three flags described below. This can be done since the payload size is always a multiple of 8. To access the size of the block's payload, you must mask the lower 3 bits to 0's.
- Let the lower three bits be $b_2$, $b_1$, and $b_0$, respectively with $b_0$ corresponding to the least significant bit of the header.
- $b_2$ indicates whether the previous block on the heap is allocated. This bit is 1 when the previous block is allocated and 0 when the previous block is free. Assume that there is a prologue block that is maintained, so the first allocatable block on the heap will have this bit set to 1.
- $b_1$ indicates whether the next block on the heap is allocated. This bit is 1 when the next block is allocated and 0 when the next block is free. Assume that there is an epilogue block that is maintained, so the last allocatable block on the heap will have this bit set to 1.
- $b_0$ indicates whether this block on the heap is allocated. This bit is 1 when this block is allocated and 0 when this block is free.

The footer value is set to match the header value.

How many memory reads must `free()` do to *decide whether to coalesce* this block with the ones around it? Assume that the only piece of data currently available is the base pointer of the block that is currently being freed. _____

What is the maximum number of writes that `free()` must do to create the final coalesced block? _____

**C. Custom Footer (4 points)**

Consider the header format specified in Part B. But now, instead of having the footer mirror the header value, Harry uses the footer to store a *pointer* to the header of the block.

How many memory reads must `free()` do to *decide whether to coalesce* this block with the ones around it? Assume that the only piece of data currently available is the base pointer of the block that is currently being freed. _____

What is the maximum number of writes that `free()` must do to create the final coalesced block? _____