15-213 Introduction to Computer Systems

Final Exam

May 3, 2006

Name: Model Solution

Andrew User ID: fp

Recitation Section: _____

- This is an open-book exam.
- Notes and calculators are permitted, but not computers.
- Write your answers legibly in the space provided.
- You have 180 minutes for this exam.

	Problem	Max	Score
Assembly Language	1	**15**	
Out-of-Order Execution	2	**20**	
Pointer Arithmetic	3	**10**	
Caching	4	**20**	
Signals	5	**15**	
Semaphores	6	**30**	
Servers	7	**20**	
System-Level I/O	8	**20**	
	Total	**150**	

1. Assembly Language (15 points)

Consider the following declaration of a binary search tree data structure.

```
struct TREE {
  int data;
  struct TREE* left;
  struct TREE* right;
};
```

Describe the layout of this structure on an x86-64 architecture.

1. (3 pts) If a is the address of the beginning of the structure, then
   - a is the address of the `data` field,
   - $a + 8$ is the address of the `left` field, and
   - $a + 16$ is the address of the `right` field.

   Give your address calculations in bytes in decimal.
2. (1 pts) a will be aligned to 0 modulo 8.
3. (1 pts) The total size of a TREE structure will be 24 bytes.

Next we consider the tree as a binary search tree, where elements to the left of a node are smaller and elements to the right of a node are larger than the data stored in the node. The following function checks whether a given integer x is stored in the global tree `root`.

```
typedef struct TREE tree;
tree* root;

int member(int x) {
  tree* t = root;
  while (t != NULL) {
    if (x == t->data)
      return 1;
    if (x < t->data)
      t = t->left;
    else
      t = t->right;
  }
  return 0;
}
```

This function might compile to the following piece of assembly code, omitting some code alignment directives and three lines for you to fill in.

```
member:
        movq    root(%rip), %rax
        testq   %rax, %rax
        je      .L9
.L14:

        _____________________________
        je      .L13
        jle     .L5
        movq    8(%rax), %rax
.L11:
        testq   %rax, %rax
        jne     .L14
.L9:

        _____________________________
        ret
.L5:
        movq    16(%rax), %rax
        jmp     .L11
.L13:

        _____________________________
        ret
```

4. (4 pts) Complete the following table, associating C variables with machine registers or assembly expressions.

C variable	Assembly expression
`x`	**`%edi`**
`root`	**`root(%rip)`**
`t`	**`%rax`**
return value	**`%eax`**

5. (6 pts) Fill in the missing three lines of assembly code.

```
member:
        movq    root(%rip), %rax
        testq   %rax, %rax
        je      .L9
.L14:
        cmpl    %edi, (%rax)
        je      .L13
        jle     .L5
        movq    8(%rax), %rax
.L11:
        testq   %rax, %rax
        jne     .L14
.L9:
        xorl    %eax, %eax
        ret
.L5:
        movq    16(%rax), %rax
        jmp     .L11
.L13:
        movl    $1, %eax
        ret
```

2. Out-of-Order Execution (20 points)

We continue the code from the previous problem

1. (15 pts) On a machine with pipelining and out-of-order execution as the machines used in this course, the efficiency of the inner loop can be improved by the use of conditional move instructions. Rewrite the code between `.L14` and `.L9` by using only instructions from the original program, ordinary move instructions, and one or more of the following conditional move instructions.

`cmovl`	S, D
`cmovle`	S, D
`cmove`	S, D
`cmovge`	S, D
`cmovg`	S, D

   As usual, S stands for the source, D for the destination, and the suffix `l`, `le`, `e`, `ge`, `g` has the same meaning as for conditional branches.

```
.L14:
        cmpl    %edi, (%rax)        # compare t->data:x
        je      .L13                # if = then return 1
        cmovl   16(%rax), %rsi      # if < then temp = t->right
        cmovg   8(%rax), %rsi       # if > then temp = t->left
        movq    %rsi, %rax          # t = temp
        testq   %rax, %rax
        jne     .L14
```

   (Solution lines are commented. Note that replacing lines 3–5 by

```
        cmovl   16(%rax), %rax      # if < then t = t->right
        cmovg   8(%rax), %rax       # if > then t = t->left
```

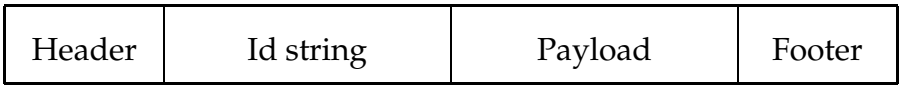
   does not work, since the arguments to a conditional move are fetched from memory even if the condition is false. The `cmovg` will therefore lead to a segfault if `cmovl` stores `NULL` in `%rax`.)

2. (5 pts) Explain why the code with conditional moves can be more efficient than the original code produced by the compiler.

   In general, it will be very difficult for the processor to predict whether the `jle` instruction in the original program will branch or not. Assuming the tree is balanced, it will be wrong roughly half the time, paying a high mis-prediction penalty. The code with the conditional moves does not pay this overhead, at the small cost of one additional move instruction.

3. Pointer Arithmetic (10 points)

A desparate student decided to write a dynamic memory allocator for an x86-64 machine in which each block has the following form:



where

- *Header* is a 4 byte header
- *Id string* is an 8 byte string
- *Payload* is arbitrary size, including padding
- *Footer* is a 4 byte footer

Assume the student wants to print the Id string with the following function

```
void print_block(void *bp) {
  printf("Found block ID: %s\n", GET_ID(bp));
}
```

where `bp` points to the beginning of the payload and is aligned to 0 modulo 8. Circle each of the following letters A–J for which the macros will correctly print the id string.

A **yes** `#define GET_ID(bp)  ((char *)(((long)bp)-8))`

B **no** `#define GET_ID(bp)  ((char *)(((int)bp)-8))`

C **no** `#define GET_ID(bp)  ((char *)(((char)bp)-8))`

D **yes** `#define GET_ID(bp)  ((char *)(((long *)bp)-1))`

E **yes** `#define GET_ID(bp)  ((char *)(((int *)bp)-2))`

F **yes** `#define GET_ID(bp)  ((char *)(((char *)bp)-8))`

G **yes** `#define GET_ID(bp)  ((char *)(((char**)bp)-1))`

H **no** `#define GET_ID(bp)  ((char *)(((char**)bp)-2))`

I **no** `#define GET_ID(bp)  ((char *)(((char**)bp)-4))`

J **no** `#define GET_ID(bp)  ((char *)(((char**)bp)-8))`

4. Caching (20 points)

Assume the following situation.

- All caches are fully associative, with LRU eviction policy.
- The cache is write-back, write-allocate.
- All caches are empty at the beginning of an execution.
- Variables i, j, and k are stored in registers.
- A float is 4 bytes.

The function `mm_ijk` multiplies two $N \times N$ arrays A and B and puts the result in R. For simplicity, we assume R is initialized to all zeros.

```
void mm_ijk (float A[N][N], float B[N][N], float R[N][N]) {
  int i,j,k;
  for (i = 0; i < N; i++)
     for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
           R[i][j] += A[i][k] * B[k][j];
}
```

1. (6 pts) Consider the executions of `mm_ijk` with N=2 and N=4 or a 64-byte fully associative LRU cache with 4-byte lines (the cache holds 16 lines). Fill in the table below with the number of cache misses caused by accesses to each of the arrays A, B, and R, assuming that the argument arrays are 16-byte aligned.

N	A	B	R
2	**4**	**4**	**4**
4	**16**	**64**	**16**

2. (6 pts) Now suppose we consider the previous experiment on a 64-byte fully associative LRU cache with 16 byte lines (the cache holds 4 lines). Fill in the table below with the number of cache misses due to each array, assuming that the argument arrays are 16 byte aligned.

N	A	B	R
2	**1**	**1**	**1**
4	**4**	**64**	**4**

3. (5 pts) Even if R is initialized to all zeros, and even if the program is single-threaded and no signals occur, after the execution of the `mm_ijk` function, R will not necessarily contain the product of A and B. Give a concrete counterexample.

If

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \text{and} \qquad B = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

then $A \cdot B = A$. But if R starts on the second row of B (in C, `R = B+1` for `float B[2][2]`) writing the results into $R[0,0]$ and $R[0,1]$ will overwrite $B[1,0]$ and $B[1,1]$, which eventually yields

$$R = \begin{bmatrix} 2 & 2 \\ 3 & 3 \end{bmatrix}$$

and $R \neq A = A \cdot B$.
Since we are tied to a machine, we interpret *product* as a floating point operation. If we tried to find a difference from the mathematically correct answer we would also have to consider overflow, underflow, and rounding error.

4. (3 pts) Under which additional assumption is `mm_ijk` correct?

We assume that there is no aliasing between A and R and between B and R. There is no harm if A and B are aliased, since we don't write to these arrays.

5. Signals (15 points)

For each code segment below, give the **largest** value that could be printed to stdout. Remember that when the system executes a signal handler, it blocks signals of the type currently being handled (and no others).

```
/* Version A */
int i = 0;

void handler(int s) {
  if (!i) {
    kill(getpid(), SIGINT);
  }
  i++;
}

int main() {
  signal(SIGINT, handler);
  kill(getpid(), SIGINT);
  printf("%d\n", i);
  return 0;
}
```

1. (5 pts) **Largest** value for version A: 2.

```
/* Version B */
int i = 0;

void handler(int s) {
  if (!i) {
    kill(getpid(), SIGINT);
    kill(getpid(), SIGINT);
  }
  i++;
}

int main() {
  signal(SIGINT, handler);
  kill(getpid(), SIGINT);
  printf("%d\n", i);
  return 0;
}
```

2. (5 pts) **Largest** value for version B: 2.

```
/* Version C */
int i = 0;

void handler(int s) {
  if (!i) {
    kill(getpid(), SIGINT);
    kill(getpid(), SIGUSR1);
  }
  i++;
}

int main() {
  signal(SIGINT, handler);
  signal(SIGUSR1, handler);
  kill(getpid(), SIGUSR1);
  printf("%d\n", i);
  return 0;
}
```

3. (5 pts) **Largest** value for version C: 4.

6. Semaphores (30 points)

We would now like to use binary search trees with (long) integer data to represent sets of integers.

```
struct TREE {
  long int data;
  struct TREE* left;
  struct TREE* right;
};
typedef struct TREE tree;
tree* root = NULL;       /* tree is initially empty */

int member(long int x);  /* returns 1 if x is in *root, 0 otherwise */
void insert(long int x); /* insert x into tree at *root */
```

We assume that the functions `member` and `insert` are straightforward (as `member` in an earlier problem) and are not thread-safe. In this problem we explore how to write wrappers so they can be used in a thread-safe manner.

We would like to allow any number of concurrent readers (via the `member` function), since simultaneous reading the data structure is safe. On the other hand, we would like to ensure that a writer (via the `insert` function) has exclusive access to the data structure (no other writers or readers allowed).

To implement this, we use one global variable, `readers`, which counts the number of currently executing readers and two semaphores: r to ensure mutually exclusive access to the `readers` variable, and w to ensure that a writer has exclusive access to the data structure stored in the `root` variable.

```
int readers = 0;
sem_t w;
sem_t r;
```

1. (15 pts) Below is the skeleton of the wrapper functions. Fill in the missing lines from the following selection. Note that you may need some commands more than once while others may remain unused.

   ```
   P(&r);
   V(&r);
   P(&w);
   V(&w);
   readers++;
   readers--;
   if (readers == 1) P(&w);
   if (readers == 1) V(&w);
   if (readers == 0) P(&w);
   if (readers == 0) V(&w);
   ```

```
int member_safe(long int x) {
  int b;
  P(&r);                    /* wait for read lock */
  readers++;                /* increment readers */
  if (readers == 1) P(&w);  /* wait for write lock if first reader */
  V(&r);                    /* release read lock */
  b = member(x);
  P(&r);                    /* wait for read lock */
  if (readers == 1) V(&w);  /* release write lock if last reader */
  readers--;                /* decrement readers */
  V(&r);                    /* release read lock */
  return(b);
}

void insert_safe(long int x) {
  P(&w);                    /* wait for write lock */
  insert(x);
  V(&w);                    /* release write lock */
}
```

(Solution lines are commented. There are some other correct solutions with the given commands.)

2. (5 pts) Which code is needed to initialize the semaphores. Circle all needed initialization.

   - `sem_init(&r, 0, 0);` Yes No **No**
   - `sem_init(&r, 0, 1);` Yes No **Yes**
   - `sem_init(&w, 0, 0);` Yes No **No**
   - `sem_init(&w, 0, 1);` Yes No **Yes**

3. (5 pts) Note that it is possible for a write request never to succeed by having certain read patterns. Precisely explain the circumstances under which this may happen.

> If the write request arrives when there is at least one reader, and from then on there is at least one reader at any point in time, then the write lock is never released and the writer does not get a turn.

4. (5 pts) Propose a simple solution to avoid this problem so that every write request is eventually honored (in words—no need to show code).

> For example, all requests could be entered into a queue so that read request arriving after a write request must wait. Some care must be taken to ensure that the queued read requests can still execute simultaneously and not in sequence.

7. Servers (20 points)

We now write a server which maintains a do-not-call list of telephone numbers. Telemarketers who call numbers in this list are subject to heavy fines. Our protocol is much simpler than HTTP. A client connects to the server and then sends either

- `+`x`\n` to add the phone number x to the do-not-call list, or
- `?`x`\n` to query whether the number x is in the do-not-call list, whereupon the server responds with `0\n` (not in the list) or `1\n` (in the list).

Here, a phone number x is just a 10-digit number (no spaces or parentheses). The server spawns a separate thread for each connection. We assume that `serve(connfd)` reads one line of input from `connfd`, parses it, and responds if appropriate according to the protocol above. For conciseness, the code below does not check return codes for system calls.

```
void* thread(void* vargp) {
  int connfd = *((int *) vargp);
  pthread_detach(pthread_self());
  serve(connfd);
  close(connfd);
  return NULL;
}

int main() {
  int listenfd, connfd;
  pthread_t tid;

  /* semaphores are initialized here */
  ...

  listenfd = open_listenfd(15213);
  while (1) {
    connfd = accept(listenfd, NULL, NULL);
    pthread_create(&tid, NULL, thread, &connfd);
  }
}
```

1. (10 pts) Even if `serve` is thread-safe, the code above exhibits a race condition. Explain this race condition in detail.

> The `connfd` variable resides on the stack frame of the `main` function. When the function `thread` is called, we have a race condition between the next call to `accept` (which would overwrite the contents of `&connfd`) and the first assignment in the `thread` which would dereference that location.

2. (10 pts) Correct the code above to avoid the race condition. Your code should not cast integers to pointers or pointers to integers (which would be in poor taste).

```
void* thread(void* vargp) {
  int connfd = *((int *) vargp);
  free(vargp);  /* new */
  serve(connfd);
  close(connfd);
  return NULL;
}

int main() {
  int listenfd;
  int* connfdp;  /* new */
  pthread_t tid;

  ...

  listenfd = open_listenfd(15213);
  while (1) {
    connfdp = malloc(sizeof(int));               /* new */
    *connfdp = accept(listenfd, NULL, NULL);     /* modified */
    pthread_create(&tid, NULL, thread, connfdp); /* modified */
  }
}
```

(Lines new or modified in the solution are annotated. Important is that the `connfd` is stored in the heap, and that this space is freed by the thread.)

8. System-Level I/O (20 points)

We continue the example from the previous question. To complete the `serve` function below, you should use

```
void    rio_readinitb(rio_t* rp, int fd);
ssize_t rio_readlineb(rio_t* rp, char* usrbuf, size_t maxlen);
ssize_t rio_writen(int fd, char* usrbuf, size_t len);
```

from the robust I/O package. The function `long atol(char* s)` converts the beginning of the string `s` to a long int.

1. (10 pts) Complete the following function `serve`.

```
void serve(connfd) {
  long int phone_number;
  rio_t rio;
  char buf[16];   /* big enough for char, number, newline, 0 */

  rio_readinitb(&rio, connfd);  /* initialize read buffer */
  if (rio_readlineb(&rio, buf, 16) != 0) {  /* read line into buf */
    if (buf[0] == '?') {
      phone_number = atol(buf+1);
      if (member_safe(phone_number))
        rio_writen(connfd, "1\n", 2);   /* number listed, reply 1 */
      else
        rio_writen(connfd, "0\n", 2);   /* number not listed, reply 0 */
    }
    if (buf[0] == '+') {
      phone_number = atol(buf+1);
      insert_safe(phone_number);
    }
  }
}
```

(Solution lines are commented.)

2. (5 pts) HTTP 1.1 allows for multiple client requests and and server responses on a single connection. Explain why.

> Establishing connections requires expensive system calls and communication and is slow. Keeping a connection open can significantly speed up multiple interactions.

3. (5 pts) We would like our protocol to support multiple interactions per connection. Show how to change the the code for the `serve` function above to handle multiple requests per connection. [Hint: you don't have to change much.]

> Change the main `if` to a `while`!