Full Name:_______________________________________

Andrew ID:_______________________________________

Recitation Section:_______________________________________

CS 15-213, Spring 2001

Final Exam

May 11, 2001

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew 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 99 points
- This exam is OPEN BOOK. You may use any books or notes you like. You cannot, however, use any computers, calculators, palm pilots, ... Good luck!

1:
2:
3:
4:
5:
6:
7:
8:
9:
TOTAL:

Problem 1. (6 points):

The pseudocode on the next page is meant to implement a Mark and Sweep garbage collector. However, there's a bunch of holes that have been left by a (very) lazy programmer. It's up to you to fill in the blank portions of the code with the correct pseudocode. Place the **letter** corresponding to the correct pseudocode for each blank line.

Some notes on functions used:

- `mark(node)` - Simply marks a node as visited
- `free_node(node)` - Frees a node and performs any necessary book-keeping (e.g. coalescing)
- `find_root_pointers(s)` - Finds root pointers and places them on stack s
- `IS_PTR(x)` - Evaluates to 1 if the value of `x` lies in the program's heap

Here are the choices available to you:

(a) `push(ptr, stack)`	(b) `if(node is marked) { continue }`
(c) `node = pop(stack)`	(d) `find_root_pointers(stack)`
(e) `free_node(node)`	(f) `mark(node)`

```
void garbage_collect()
{
  stack_t stack;  /* Working set of pointers */
  node_t  node;
  ptr_t   ptr;
  if(no memory allocated yet) { return; }

  1. ____________
  while(not_empty(stack)) {

    2. ______________

    3. ______________

    4. ______________
    for each ptr in node's data block {
      if(IS_PTR(ptr)) {

        5. ______________
      }
    }
  }
  for each unmarked node {

    6. ____________
  }
}
```

Problem 2. (12 points): Please fill in the blanks in the following C function whose assembly code is given below.

```
int foo (_______ x, _______ y)
{
    if (________)
        return _____________;
    else
        return _________________________________________;
}
```

```
foo:
       pushl %ebp
       movl %esp,%ebp
       subl $20,%esp
       pushl %ebx
       movl 8(%ebp),%ebx
       movl 12(%ebp),%eax
       testl %eax,%eax
       je .L18
       addl $-8,%esp
       decl %eax
       pushl %eax
       leal 4(%ebx),%eax
       pushl %eax
       call foo
       addl (%ebx),%eax
       jmp .L21
       .p2align 4,,7
.L18:
       movl (%ebx),%eax
.L21:
       movl -24(%ebp),%ebx
       movl %ebp,%esp
       popl %ebp
       ret
```

Problem 3. (12 points):

This problem tests your understanding of the memory layout of C structures and unions in IA-32/Linux assembly code. (Recall that in Linux, 8 byte primitive data types must be only 4-byte aligned). Consider the data structure declarations below. (Note that this is a single declaration which includes several data structures; they are shown horizontally to fit on the page.)

```
struct a {                  struct b{                   union u{
    short s;                    short s;                    int i;
    char arr[3];                int i;                      short s;
    int i;                      char arr[3];                char arr[3];
};                          };                          };


char q1(char *ptr); char q2(char *ptr); char q3(char *ptr); char q4(char *ptr);

int main()
{
    struct a one;
    struct b two;
    union u three;
    struct b four[4];

    four[2].arr[1] = one.arr[1] = two.arr[1] = three.arr[1] = 5;

    /* we omit the casts in the following line for clarity and brevity */
    printf("%d %d %d %d\n",q1(&one), q2(&two), q3(&three), q4(&four));
}
```

Given the above code, your task is the fill in the blanks of each function definition such that the output of the program is "5 5 5 5". You can assume that all uninitialized values are 0.

```
char q1(char *ptr)
{
    return *(ptr + _____ );
}

char q2(char *ptr)
{
    return *(ptr + _____ );
}

char q3(char *ptr)
{
    return *(ptr + _____ );
}

char q4(char *ptr)
{
    return *(ptr + _____ );
}
```

Problem 4. (12 points):

Assume we are given a processor without a floating-point unit, and we want to implement single-precision floating-point numbers directly. Below is a routine `double(x)` that is supposed do double the value of `x` (i.e. `x * 2`). It is supposed to act according to the IEEE specification.

For each of the stared and numbered lines in the code specify whether it is correct (C) or incorrect (I). Use the corresponding numbered slots below. If it is incorrect, you must either fix it (only if the fix can be made in one line of code), or explain in one sentence why it is wrong. When judging the correctness of each line assume that all previous lines are correct. **A point will be deducted if you say it is correct and it is not.**

```
*1   int exp_bias = 128;

     float double(float x) {

       /* mask out everything but the sign bit */
*2     int sign = x & (1 << 32);

       /* get the biased exponent */
*3     int exp = (x >> 23) && 0xFF;

       /* get the fractional part */
*4     int frac = x & ((1 << 23) - 1);

       /* check for NAN */
*5     if (exp == 255 && frac != 0)
*6         return x;   /* return x if NAN */

       /* check of denormalized */
*7     if (exp == -bias) {
*8        frac = frac << 1;   /* double the value */
          /* check if overflow into a normalized number */
*9        if (frac >= (1 << 23)) {
*10          frac = frac & ((1 << 23) - 1); /* fix fractional */
*11          exp = 1;                 /* change exponent */
          }
       }
       else {   /* normalized */
*12       exp = exp + 1;
          /* check if infinity */
*13       if (exp > 255) {
*14         exp = 255;   /* set infinity exponent */
*15         frac = 0;    /* set infinity fractional */
          }
       }
       return sign | (exp << 23) | frac;
     }
```

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

11.

12.

13.

14.

15.

Problem 5. (12 points):

This problem will test your understanding of stack frames. You are given the following declarations on an x86 architecture:

```
struct color {
  char r,g,b;
};

struct vertex {
  int xyz[3];
  struct color c;
};

struct vertex get_light_source()
{
  /* code that is irrelevant for question */
}

void calc_shading(struct vertex *v0, struct vertex *v1, struct vertex *v2,
                  int r, int s, struct vertex light_source, ...)
{
  /* code that is irrelevant for question */
}

void shade_triangle(struct vertex *v0, struct vertex *v1, struct vertex *v2)
{
  struct vertex ls = get_light_source();
  struct color rgb;
  calc_shading(...,...,...,...,...,...,...);
  /* code that is irrelevant for question */
}
```

On the next page, you have the diagram of the stack immediately before the call assembly instruction for `calc_shading()` in `shade_triangle()` is executed. Argument `v2` given to `shade_triangle()` is stored at `0xbffff6e0`. You can make the following assumptions:

- The stack grows towards smaller addresses.
- The alignment of variables on the stack corresponds to the alignment of variables on the heap.
- The allocation order of local variables on the stack corresponds to their definition order in the source code.
- The compiler does not insert any additional unused space on the stack apart from unused space required for alignment restrictions of variables.
- No registers (apart from `%ebp`) are being saved on the stack.

You are free to add comments into the third column of the table below but those comments will not be graded.

Address	Numeric Value	Comments/Description
0xbffff6e0	0xbffff750	
0xbffff6dc	0xbffff740	
0xbffff6d8	0xbffff730	
0xbffff6d4	0x080483ff	
0xbffff6d0	0xbffff6f0	
0xbffff6cc	0xbffff568	
0xbffff6c8	0x00000224	
0xbffff6c4	0x00000034	
0xbffff6c0	0x00000005	
0xbffff6bc	0xa3c8530a	
0xbffff6b8	0xbffff6bc	
0xbffff6b4	0xbffff568	
0xbffff6b0	0x00000224	
0xbffff6ac	0x00000034	
0xbffff6a8	0x00000005	
0xbffff6a4	0x00000003	
0xbffff6a0	0x00000005	
0xbffff69c	0xbffff730	
0xbffff698	0xbffff740	
0xbffff694	0xbffff750	

A. The given declaration of `calc_shading()` is missing the type of the seventh parameter. Determine this type.

B. List the parameters used in the call to `calc_shading()` in `shade_triangle()`, as they would appear in the source code.

```
calc_shading(      ,      ,      ,      ,      ,      ,      );
```

C. Give the current value of the frame pointer (`%ebp`).

D. Give the address (relative to the frame pointer) where `ls.c` is stored.

Problem 6. (12 points): In this problem, you will measure the cache efficiency of matrix multiplication. Assume that

- All caches are fully associative, LRU caches.
- The cache is empty at the beginning of an execution.
- Variables i,j,k are stored in registers and thus, an access to these variables does not change the cache content and does not cause a cache miss.
- For a given size `N` the arrays `A`,`B` and `R` are defined as follows.

```
float A[N][N];
float B[N][N];
float R[N][N];
```

- A float is 4 bytes.

Part I

The function `mm_ijk` multiplies two `N` × `N` arrays `A` and `B` puts the result in `R`. For simplicity, `mm_ijk` assumes that the result array `R` is initialized to all zeros.

```
void mm_ijk (float *A, float *B, float *R) {
   register 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. Consider the executions of `mm_ijk` with `N`= 2, and `N` = 4 on 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`.

N	A	B	R
2			
4			

2. Now suppose we repeat 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.

N	A	B	R
2			
4			

Part II

In the second part, you will measure the cache performance of `mm_kij`. We permute the `i`, j, k loop of `mm_ijk` to `k`, i, j to obtain the function `mm_kij` given below.

```
void mm_kij (float *A, float *B, float *R)  {
   register int i, j, k;

   for (k = 0; k < N; ++k)
       for (i = 0; i < N; ++i)
           for (j = 0; j < N;++j)
               R[i][j] += A[i][k] * B[k][j];
}
```

1. Consider the executions of `mm_kij` with `N`= 2, and `N` = 4 on 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`.

N	A	B	R
2			
4			

2. Now suppose we repeat the previous experiments now 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.

N	A	B	R
2			
4			

Problem 7. (9 points):

The following two problems deal with concurrency.

Signaling

Semaphores cannot only be used for mutual exclusion, but also for signaling. In the example below, give the correct initial value for the semaphore S.

Thread A:

```
int foo() {
  /* do something */

  /* wait for B to reach L */
  P(S);

  /* do something */
}
```

Thread B:

```
int bar() {
  /* do something */

  /* L: signal A to proceed */
   V(S);

  /* do something */
}
```

inital value of S: ______

Producer/Consumer

Below, you are given a solution for the producer/consumer problem, in which semaphores are used both for mutual exclusion and for signaling. Assume that there is space for N items in the buffer.

```
Producer thread:

while(1) {

  /* produce item */

  P(mutex);
  P(not_full);

  /* insert item into buffer */

  V(mutex);
  V(not_empty);
}
```

```
Consumer thread:

while(1) {

  P(not_empty);
  P(mutex);

  /* remove item from buffer */

  V(mutex);
  V(not_full);

  /* consume item */
}
```

This code can deadlock. Show how to fix it by moving one line of code (use an arrow to show were a line should move to).

Give the initial value of each semaphore (if there are multiple possibilities, choose the largest one):

A. `not_full`: _______

B. `not_empty`: _______

C. `mutex`: _______

Problem 8. (12 points):

This problem tests your understanding of the Berkeley Sockets API and the POSIX threads API. Consider the following client and server programs which implement a remote spell checking service. For simplicity we assume that the server always runs on 128.2.222.28 (muskie.cmcl.cs.cmu.edu) and the strings being spell checked will never exceed BUF_SIZE in length. The sections of the code that are in **bold** may or may not contain errors.

After carefully reading and understanding all of the code, your task is to circle the letter of the *single* statement that *best* describes what is wrong (if anything) with the section of code that corresponds to each question.

Client

```
int main(int argc, char *argv[]) {
    int fd, num;
    char buf[BUF_SIZE];
    struct sockaddr_in serveraddr, clientaddr;
    char *msg = argv[1];
    if(strlen(msg) >= BUF_SIZE-1) return ERROR;
    if( (fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) return ERROR;
    serveraddr.sin_family = AF_INET;
    /* Question 1 */
    serveraddr.sin_addr.s_addr = ntohl(0x8002de9e); /* 128.2.222.158 */
    serveraddr.sin_port = ntohs(DEFAULT_PORT);

    if(connect(fd, (struct sockaddr*)&serveraddr, sizeof(serveraddr)) < 0)
      return ERROR;
    if((num = write(fd, msg, strlen(msg)+1)) < 0) return ERROR;

    /* Question 2 */
    while((num = read(fd, buf, BUF_SIZE-1)) > 0)
    {
      buf[num] = 0;
      printf("%s", buf);
    }

    return SUCCESS;
}
```

Question 1

a) The code is correct.
b) `ntohl` and `ntohs` should be changed to `htonl` and `htons` respectively.
c) `ntohl` and `ntohs` should be changed to `htons` and `htonl` respectively.
d) `ntohl` and `ntohs` should be changed to `ntohs` and `ntohl` respectively.
e) The arrow operator (`->`) should be used instead of the period (.).

Question 2

a) The code is correct.
b) The code is correct but the while loop is not needed.
c) The last character of the data won't print out.
d) The output is correct but the `while` loop never exits.
e) Both c and d are correct.

Server Part 1

```
int num_words_checked;

/* process_request takes a pointer to the client file descriptor.
   It reads the string to spell check, spell checks it, and returns
   the misspelled words to the client.
*/
void* process_request(void *arg)
{
    int len, num;
    int clientfd = *(int*)arg;
    char string[BUF_SIZE];

    /* Question 3 */
    if((num = read(clientfd, string, BUF_SIZE-1)) < 0)
      return NULL;

    /* Question 4 */
    num_words_checked += count_words(string);

    /* spell_check takes a string, mutates it to contain only
       the misspelled words, and returns the length of the
       new string.  For example, given "The cow goes asdf qtz"
       it would change the string to be only "asdf qtz" and
       return eight.
    */
    len = spell_check(string);
    write(clientfd, string, len);
    close(clientfd);
    free(arg);
    return NULL;
}
```

Question 3

a) The code is correct.
b) The `read` should be inside a loop that terminates when there is an error reading.
c) The `read` should be inside a loop that terminates when the end-of-file is reached.
d) There should be a mutex to protect string from concurrent accesses.
e) Both b and c are correct.

Question 4

a) The code is correct.
b) There should be a mutex to protect string from concurrent accesses.
c) There should be a mutex to protect `num_words_checked` from concurrent accesses.
d) Since `num_words_checked` is a global variable, the process_request thread doesn't have access to it.
e) Both b and c are correct.

Server Part 2

```
int main(int argc, char *argv[]) {
    int listenfd, clientlen, optval;
    struct sockaddr_in serveraddr, clientaddr;
    if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) return ERROR;
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
               (const void *)&optval , sizeof(int));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(DEFAULT_PORT);
    num_words_checked = 0;
    if(bind(listenfd, (struct sockaddr*)&serveraddr, sizeof(serveraddr)) < 0)
      return ERROR;
    if(listen(listenfd, 5) < 0) return ERROR;
    clientlen = sizeof(clientaddr);
    while(1)
    {
      int clientfd;
      pthread_t  thread;
      if((clientfd = accept(listenfd, (struct sockaddr*)&clientaddr, &clientlen)) < 0)
          return ERROR;

      /* Question 5 */
      pthread_create(&thread, NULL, process_request, (void*)&clientfd);

      pthread_detach(thread);

      /* Question 6 */
      pthread_join(thread, NULL);

    }
}
```

Question 5

a) The code is correct.
b) Passing the address of `clientfd` to `process_request` is wrong since it creates a race condition.
c) Passing the address of `clientfd` to `process_request` is wrong since it results in a memory error when `process_request` frees `arg`.
d) Passing the address of `clientfd` to `process_request` is wrong since it results in a memory error when `process_request` tries to dereference `arg`.
e) Both b and c are correct.

Question 6

a) The code is correct.
b) The code is correct, but the join results in a loss of concurrency.
c) `pthread_join` will always return an error since `thread` is not joinable.
d) It is desirable to call `pthread_join` here, but first `thread` must be re-attached.
e) It would make more sense to use `fork` instead of threads.

Problem 9. (12 points):

Each of the following two programs is intended to have the same functionality – print "ouch!" once for each time the user types the character 'p'. For at least one of the programs, however, there is no guarantee that all of the processes or threads that it creates will be reaped. For each program, either indicate that the program does reap all of the processes and threads that it creates, or modify the program by adding or changing at most one line of code to ensure that it does.

```
/* program one */
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>

void handler(int sig);

int main()
{
  char c;
  signal(SIGCHLD,handler);
  while(1){
    c = getchar();
    if (c == EOF) exit(0);
    if (c == 'p'){
      if (fork()==0){
        printf("ouch!\n");
        exit(0);
      }
    }
  }
}

void handler (int sig)
{
  pid_t pid;
  int status;
  pid = waitpid(-1,&status,WNOHANG);
  return;
}
```

```
/* program two */
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

void *ouch(void *n);

int main()
{
  char c;
  pthread_t tid;

  while(1){
    c = getchar();
    if (c == EOF) exit(0);
    if (c == 'p'){
      pthread_create(&tid,NULL,ouch,NULL);
    }
  }
}

void *ouch(void *n)
{
  printf("ouch!\n");
  return NULL;
}
```

Problem 10. (0 points):

And now that you're finished with 15-213, take 5 minutes and read this poem in the vein of Dr. Seuss.

Bits Bytes Chips Clocks
Bits in bytes on chips in box.
Bytes with bits and chips with clocks.
Chips in box on ether-docks.

Chips with bits come. Chips with bytes come.
Chips with bits and bytes and clocks come.

Look, sir. Look, sir. read the book, sir.
Let's do tricks with bits and bytes, sir.
Let's do tricks with chips and clocks, sir.

First, I'll make a quick trick bit stack.
Then I'll make a quick trick byte stack.
You can make a quick trick chip stack.
You can make a quick trick clock stack.

And here's a new trick on the scene.
Bits in bytes for your machine.
Bytes in words to fill your screen.

Now we come to ticks and tocks, sir.
Try to say this by the clock, sir.

Clocks on chips tick.
Clocks on chips tock.
Eight byte bits tick.
Eight bit bytes tock.
Clocks on chips with eight bit bytes tick.
Chips with clocks and eight byte bits tock.

Here's an easy game to play.
Here's an easy thing to say....

If a packet hits a pocket on a socket on a port,
and the bus is interrupted as a very last resort,
and the address of the memory makes your floppy disk abort
then the socket packet pocket has an error to report!

If your cursor finds a menu item followed by a dash,
and the double-clicking icon puts your window in the trash,
and your data is corrupted cause the index doesn't hash,
then your situation's hopeless, and your system's gunna crash.

You can't say this? What a shame, sir!
We'll find you another game, sir.

If the label on the cable on the table at your house
says the network is connected to the button on your mouse,
but your packets want to tunnel on another protocol,
that's repeatedly rejected by the printer down the hall,
and your screen is all distorted by the side-effects of gauss,
so your icons in the window are as wavy as a souse,
then you may as well reboot and go out with a bang,
cause as sure as I'm a poet, the sucker's gunna hang!

When the copy of your floppy's getting sloppy on the disk,
and the microcode instructions cause unnecessary risc,
then you have to flash your memory and you'll want to RAM your ROM.
quickly turn off your computer and be sure to tell your mom!

- By Gene Ziegler -