Andrew login ID (please print in capital letters): | | | | | | | | | | | | |

Full Name: ____________________

Recitation Section: ____________________

15-213/18-243, Spring 2009
Final Exam

Tuesday, May 12, 2009

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–H) on the front.
- The exam has a maximum score of 200 points.
- This exam is OPEN BOOK. You may use any books or notes you like. No calculators or other electronic devices are allowed.

Virtual Memory	1 (20):
System-Level IO	2 (15):
Cache Memories	3 (20):
Signals	4 (15):
Assembly	5 (15):
Network Programming	6 (10):
Floating Point	7 (15):
Dynamic Memory 1	8 (10):
Dynamic Memory 2	9 (10):
Stack 1	10 (20):
Stack 2	11 (30):
Multithreading	12 (20):
	TOTAL (200):

Problem 1. (20 points):

VM On a Boat

In this question you will perform a virtual to physical address translation for a hypothetical virtual memory architecture.

The specifications for the system are as follows:

- Virtual addresses are 16 bits in length
- The page size is 64 bytes
- The system operates on a two level page table structure, which is organized as follows:
  - The page directory has 16 entries, each of which is 2 bytes in length
  - Each page table has 64 entries, each of which is 2 bytes in length
- Each page directory entry encodes the address of the page table in its upper bits, and the lowermost bit is a valid bit, where P = 1 indicates that the page table is present.
- Each page table entry encodes the physical page number in its upper bits, and the lowermost bit is a valid bit, where P = 1 indicates that the page frame is present.

Below is a memory dump of various regions of memory. The left column of each table stores the address, and the right column stores the value at that address.

Address	Value
`0x0200`	`0x1401`
`0x0206`	`0x1481`
`0x020c`	`0x1501`
`0x0212`	`0x1581`
`0x0224`	`0x1600`
`0x0228`	`0x1681`
`0x022c`	`0x1700`
`0x0230`	`0x1781`
`0x1408`	`0x3201`
`0x1410`	`0x3301`
`0x1420`	`0x3400`
`0x1440`	`0x3501`
`0x1480`	`0x3600`
`0x1488`	`0x3701`
`0x1490`	`0x3800`
`0x14a0`	`0x3901`
`0x14c0`	`0x3a01`
`0x1500`	`0x3b01`
`0x1508`	`0x3c00`
`0x1510`	`0x3d01`
`0x1520`	`0x3e01`
`0x1540`	`0x3f01`
`0x1580`	`0x4001`

Address	Value
`0x1588`	`0x4101`
`0x1590`	`0x4200`
`0x15a0`	`0x4301`
`0x15c0`	`0x4401`
`0x1600`	`0x4501`
`0x1604`	`0x4600`
`0x1612`	`0x4701`
`0x1624`	`0x4801`
`0x1648`	`0x4900`
`0x1680`	`0x4a01`
`0x1684`	`0x4b00`
`0x1692`	`0x4c01`
`0x16a4`	`0x4d00`
`0x16c8`	`0x4e01`
`0x1704`	`0x4f01`
`0x1712`	`0x5001`
`0x1724`	`0x5101`
`0x1748`	`0x5201`
`0x1784`	`0x5301`
`0x1792`	`0x5401`
`0x17a4`	`0x5501`
`0x17c8`	`0x5600`
`0xdead`	`0xbeef`

Part I

Process 1 is trying to read at **virtual address `0x683B`**. The page directory base register for this process stores the value `0x0200`. Answer the following questions. Use the space below for your calculations and working. **To facilitate the awarding of partial credit, please note down any memory addresses looked up, and the values they contained.**

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

1. What is the address of the page directory entry?
2. What is stored in the page directory entry?
3. What is the address of the page table entry?
   OR The page table is not present (circle if true)
4. What is the physical address accessed?
   OR There was a page fault (circle if true)

Part II

Process 2 is trying to write to virtual address `0x44A3`. The page directory base register for this process stores the value `0x0220`. Answer the following questions. Use the space below for your calculations and working. **To facilitate the awarding of partial credit, please note down any memory addresses looked up, and the values they contained.**

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

1. What is the address of the page directory entry?
2. What is stored in the page directory entry?
3. What is the address of the page table entry?
   OR The page table is not present (circle if true)
4. What is the physical address accessed?
   OR There was a page fault (circle if true)

Problem 2. (15 points):

File Descriptor Mania

Suppose the file `./file1.txt` has the following contents:

```
aabbccdd
```

And we have the following C files compiled to `./program1` and `./program2`, respectively.

```
/*
 *  Program1
 */
#include <unistd.h>
#include <fcntl.h>

int main()
{
  int pid, fd_x, fd_y, fd_z;
  char buf[8];

  fd_x = open("file1.txt", O_RDWR);
  fd_y = open("file1.txt", O_RDWR);
  fd_z = open("file1.txt", O_RDWR);

  read(fd_x, buf, 2);
  read(fd_y, buf+2, 4);

  if ((pid = fork()) == 0) {
    dup2(fd_x, STDOUT_FILENO);
    dup2(fd_y, STDIN_FILENO);
    execl("program2", "program2", NULL);
  }

  wait(NULL);

  read(fd_y, buf+6, 2);
  write(fd_z, buf+6, 2);
  write(fd_x, buf+4, 2);
  write(fd_x, buf+2, 2);

  close(fd_x);
  close(fd_y);
  close(fd_z);
}
```

```
/*
 *  Program2
 */
#include <unistd.h>
#include <fcntl.h>

int main()
{
  char buf[2];

  read(STDIN_FILENO, buf, 2);
  write(STDOUT_FILENO, buf, 2);
}
```

What is the contents of `file1.txt` after `./program1` executes? Assume that reads and writes are not cached.

Problem 3. (20 points):

We consider a 128 byte data cache that is 2-way associative and can hold 4 doubles in every cache line. A double is assumed to require 8 bytes.

For the below code we assume a cold cache. Further, we consider an array `A` of 32 doubles that is cache-aligned (that is, `A[0]` is loaded into the first slot of a cache line in the first set). All other variables are held in registers. The code is parameterized by positive integers `m` and `n` that satisfy `m*n = 32` (i.e., if you know one you know the other).

Recall that miss rate is defined as $\frac{\#\text{misses}}{\#\text{accesses}}$.

```
float A[32], t = 0;
for(int i = 0; i < m; i++)
  for(int j = 0; j < n; j++)
    t += A[j*m + i];
```

Answer the following:

1. How many doubles can the cache hold?
2. How many sets does the cache have?
3. For `m = 1`:
   (a) Determine the miss rate.

   (b) What kind of misses occur?

   (c) Does the code have temporal locality with respect to accesses of `A` and this cache?

4. For `m = 2`:

   (a) Determine the miss rate.

   (b) What kind of misses occur?

5. For `m = 16`:

   (a) Determine the miss rate.

   (b) What kind of misses occur?

   (c) Does the code have spatial locality with respect to accesses of `A` and this cache?

Problem 4. (15 points):

You may have taken 15-251 and learned that there is no oracle for the halting set, meaning it's impossible to write a program that will determine if an other arbitrary program will halt for a given input. A 213 TA, Punter Hitelka, is determined to disprove this using a program called autolab that makes students do such determinations for credit. Congrats, you are the guinea pig!

For this problem, you must determine if the following code halts or not, then tell us why. By halt, we mean that the **parent** process eventually exits. You do not need to tell us if any child processes are maintained as zombies.

Write your answer in the blank space below each of the three code blocks.

When grading this problem, we will only read the first 30 words of each response, so keep your answers clear and concise!

1. Does this program terminate? Justify.

```
void main()
{
  int *x = malloc(sizeof(int));
  int cpid = fork();

  *x = 1;

  if(cpid == 0) {
    *x = 0;
  }

  while(*x)
    continue;

  return 0;
}
```

2. Does this program terminate? Justify.

```
void handler(int signum)
{
  exit(1);
}

void main()
{
  int cpid;
  sigset_t s;

  sigaddset(&s, SIGUSR1);

  sigprocmask(SIG_BLOCK, &s, NULL);

  signal(SIGUSR1, handler);

  if((cpid = fork()) == 0)
  {
    printf("I’m a child");
    while(1)
      continue;
  }

  sigprocmask(SIG_UNBLOCK, &s, NULL);

  printf("I’m on a boat!");
  kill(cpid, SIGUSR1);
  waitpid(cpid, NULL, 0);
}
```

3. Does this program terminate? Justify.

```
void sig_kill_handler(int signum)
{
  printf("I’m not gonnaaa stoopppppp\n");
  while(1)
    continue;
}

void main()
{
  int cpid;

  signal(SIGKILL, sig_kill_handler);

  if((cpid = fork()) == 0)
  {
    printf("Looping Forever\n");
    while(1) continue;
  }
  else
  {
    kill(cpid, SIGKILL);
    waitpid(cpid, NULL, 0);
  }
}
```

Problem 5. (15 points):

Following is a series of **three** C snippets with associated disassemblies. Each snippet contains **one** or **zero** errors. If there is an error, circle it and provide a **brief** explanation of why it is wrong in the space below the code. If there is no error, state that there is no error. Note that the error (if one exists) is in the C → assembly translation, not in the logic or behavior of the C code.

Please write your answers only on this page.

```
int squareNumber(int x) {
  return (x * x);
}

08048344 <squareNumber>:
 8048344:       55                      push   %ebp
 8048345:       89 e5                   mov    %esp,%ebp
 8048347:       8b 45 04                mov    0x4(%ebp),%eax
 804834a:       0f af c0                imul   %eax,%eax
 804834d:       5d                      pop    %ebp
 804834e:       c3                      ret
```

```
int fourth(char *str) {
  return str[3];
}

0804834f <fourth>:
 804834f: 55                    push   %ebp
 8048350: 89 e5                 mov    %esp,%ebp
 8048352: 8b 45 08              mov    0x8(%ebp),%eax
 8048355: 83 c0 03              add    $0x3,%eax
 8048358: 0f be 00              movsbl (%eax),%eax
 804835b: 5d                    pop    %ebp
 804835c: c9                    leave
 804835d: c3                    ret
```

```
int unrandomNumber() {
  return 4;
}

0804835e <unrandomNumber>:
 804835e: 55                    push   %ebp
 804835f: 89 e5                 mov    %esp,%ebp
 8048361: a1 04 00 00 00        mov    0x4,%eax
 8048366: 5d                    pop    %ebp
 8048367: c3                    ret
```

Problem 6. (10 points):

Crummy Networks

Error Handling

Below is some code for a concurrent echo server. We have left out the error handling code for the three functions `socket`, `send`, and `recv`. These blanks are marked with

```
/**** WRITE CODE BELOW *******/


  <and you have to fill in here>


/***** WRITE CODE ABOVE *****/
```

Please fill in those blanks with appropiate error handling code. Do not print out any messages, just modify the control flow.

```
#include <stdio.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

#define BUFF_SIZE 512
#define SERVER_PORT 15213

char buffer[BUFF_SIZE];

void * handleConnection(void *);

int main(){
    int server_sock;
    struct sockaddr_in serverAddr, clientAddr;
    pthread_t tid;

    /*ignore the SIGPIPE signal*/
    signal(SIGPIPE,SIG_IGN);

    /*open server_socket */
    if((server_sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))<0){
        /**** WRITE CODE BELOW *******/






        /***** WRITE CODE ABOVE *****/
    }

    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddr.sin_port = htons(SERVER_PORT);
    serverAddr.sin_family = AF_INET;
    if(bind(server_sock,(struct sockaddr *)&serverAddr,
            sizeof(struct sockaddr)<0)){
        /*handle bind failing*/
        exit(-1);
   }
```

```
    if(listen(server_sock,15)<0){
        /*handle listen failing*/
        exit(-1);
    }
    while(1){
        int client_socket;
        size_t clientLen = sizeof(struct sockaddr);

        if((client_socket = accept(server_sock,(struct sockaddr *)&clientAddr,
            &clientLen))<0){
            /*handle failing of accept*/
            continue;
        }
        pthread_create(&tid,NULL,handleConnection,(void *) client_socket);
    }
}

/*handle data on this socket*/
void * handleConnection(void * sock){
    int socket = (int) sock;
    int recvSize;

    pthread_detach(pthread_self());

    do{
        if((recvSize = recv(socket,buffer,BUFF_SIZE,0))<0){
            /***** WRITE CODE BELOW*******/





            /***** WRITE CODE ABOVE ******/
        }

        if(send(socket,buffer,recvSize,0)<0){
            /***** WRITE CODE BELOW *******/





            /****** WRITE CODE ABOVE *****/
        }
    }while(recvSize >0);
    /*once the code reaches this point, we have received 0 bytes from the recv
     * call*/
    close(socket);
    pthread_exit(NULL);
}
```

Network Bugs

Assuming all system calls succeed (and therefore the error handling code you wrote in part A is never executed), please locate the 2 logic bugs in this code and describe them. (A logic bug is one where the programmer misunderstood the way in which their program will execute and will produce unwanted behavior under certain input conditions).

Bug 1

Bug 2

Problem 7. (15 points):

You've been asked to design the floating-point unit for Harry Q. Bovik's new microprocessor. Harry is sure that he wants to use the IEEE standard for floating-point numbers, but he isn't sure of some of the other design parameters. He has some questions for you:

If floats are represented with 12 bits, with 1 bit for the sign, 6 for the exponent, and 5 for the fraction:

1. What is the largest non-infinite number representable?

2. What is the smallest positive number representable?

3. What does the number 255 round to using this format?

Harry is concerned about precision in his system. He'd like to be able to represent positive integers up to 255 without having to round.

What is the least number of fraction bits and the least number of exponent bits to make this possible? (Note that the total number of bits may change)

4. Number of Fraction Bits:

5. Number of Exponent Bits:

Harry decided to extend his floating point format by one bit.

First he wonders about the effect on the range (defined as the difference between the smallest and largest representable finite number). Which of the following is true?

6. The range will be increased

   (a) By adding the bit to the fraction bits
   (b) By adding the bit to the exponent bits
   (c) By both
   (d) By neither

Next he worries about the rounding error. Which of the following is true?

7. The rounding error for all numbers remains unchanged or is reduced

   (a) By adding the bit to the fraction bits
   (b) By adding the bit to the exponent bits
   (c) By both
   (d) By neither

Problem 8. (10 points):

The Curse of Abalienation!!

For this question, we will be looking at the 32-bit libc implementation of malloc.

- The libc implementation uses an 8 byte alignment of the payload areas.
- The libc implementation uses the following layout for free blocks:

header (4 bytes)	prev (4 bytes)	next (4 bytes)	payload (arbitrary size)	footer (4 bytes)

  Where `prev, next` and `footer` are stored inside the space for the payload.

- The libc implementation uses the following layout for allocated blocks:

header (4 bytes)	payload (arbitrary size)

Your friend, Harry Q. Bovik, is taking 15-123, where one of the assignments is to write a linked list implementation of a dictionary. Harry is experiencing a strange bug where his dictionary works on everything except for 12 letter words, on which it generates a Segmentation Fault. After some debugging you find that it also doesn't work on words of size 20 and 28 (you don't test any further).

Here is Harry's addWordDict method:

```
int addWordDict(dictionary * dict, char * word){
        int result;
        char * wordCopy;
        if (dict == NULL){
                return ERR_NULL_DICT;
        }
        if(word == NULL){
                return WARN_INVALID_ARGUMENT;
        }
        /*add the word */
        /*We're going to make a copy of the word because the word buffer
                gets reused. This wordCopy will get free'd when we remove
                the word from the dictionary. */
        wordCopy = (char *)malloc((strlen(word)) * sizeof(char));
        strcpy(wordCopy,word);
        result = addItemLL(((dict)->wordList),(void *) wordCopy);
        dict->count = ((dict)->wordList)->count; /*update the count */
        return result;
}
```

1. What is wrong with Harry's addWordDict method?

2. Why does this code work on words of sizes other than 12, 20, 28... but not on these sizes? (Be as detailed as possible)

Problem 9. (10 points):

Harry Q. Bovik is working on some code and needs your help. He is writing a malloc package with the intent that it should compile and run correctly on both x86 and x86-64 machines, but to keep things simple he's never allowing the heap to grow larger than 4GB, so he can use 4 byte headers. He's using the block layout shown below, which should look familiar to you.

```
+----------------------------------------------------------------+
| header (4 bytes) |     payload (varies)     | footer (4 bytes) |
+----------------------------------------------------------------+
```

When Harry gets to his `free` implementation, he decides to write a macro to abstract the pointer arithmetic details out of his code. The first thing he needs to do is determine a block's size given a pointer to the payload of that block, to be used like so:

```
void free(void *p) {
    int size = HEADER(p) & ~0x7;
    ...
}
```

Fill in the blanks in the table below, indicating with "Yes" or "No" whether each macro will perform correctly on either x86 or x86-64.

	x86	x86-64
`#define HEADER(p) (*(long *)((char **)(p) - 1))`		
`#define HEADER(p) (*(char *)((char *)(p) - 4))`		
`#define HEADER(p) (*(int *)((char *)(p) - 2))`		
`#define HEADER(p) (*(long *)((char *)(p) - 2))`		
`#define HEADER(p) (*(char *)((long *)(p) - 2))`		
`#define HEADER(p) (*(int *)((long *)(p) - 1))`		
`#define HEADER(p) (*(char *)((int *)(p) - 1))`		
`#define HEADER(p) (*(long *)((long *)(p) - 2))`		
`#define HEADER(p) (*(int *)((int *)(p) - 1))`		

Problem 10. (20 points):

Consider the following C program running on a 32-bit machine.

```
int fact (int n) {
  if (n == 1)
    return 1;

  return n * fact(n - 1);
}

int main (void) {
  int a = fact(2);
  return 0;
}
```

The assembly dump of the two functions is printed below.

```
8048344 <fact>:
8048344:   55                         push   %ebp
8048345:   89 e5                      mov    %esp,%ebp
8048347:   53                         push   %ebx
8048348:   83 ec 04                   sub    $0x4,%esp
804834b:   8b 5d 08                   mov    0x8(%ebp),%ebx
804834e:   b8 01 00 00 00             mov    $0x1,%eax
8048353:   83 fb 01                   cmp    $0x1,%ebx
8048356:   74 0e                      je     8048366 <fact+0x22>
8048358:   8d 43 ff                   lea    0xffffffff(%ebx),%eax
804835b:   89 04 24                   mov    %eax,(%esp)
804835e:   e8 e1 ff ff ff             call   8048344 <fact>
8048363:   0f af c3                   imul   %ebx,%eax
8048366:   83 c4 04                   add    $0x4,%esp
8048369:   5b                         pop    %ebx
804836a:   5d                         pop    %ebp
804836b:   c3                         ret

804836c <main>:
804836c:   55                         push   %ebp
804836d:   89 e5                      mov    %esp,%ebp
804836f:   83 ec 08                   sub    $0x8,%esp
8048372:   c7 04 24 02 00 00 00       movl   $0x2,(%esp)
8048379:   e8 c0 ff ff ff             call   8048344 <fact>
804837e:   b8 00 00 00 00             mov    $0x0,%eax
8048383:   c9                         leave
8048384:   c3                         ret
8048385:   90                         nop
```

Right before the execution of the call to `fact(2)` at line `0x8048379`, the value of `%esp` is `0xbfc5e4f0`, and the value of `%ebx` is `0xdeadbeef`.

Please answer the following questions.

1. What is the value of `%ebp` before the call to `fact(2)`?

2. How many bytes does each stack frame of `fact()` use?

3. How many bytes of the stack are written to in total before `fact(2)` returns?

4. Fill in the values contained on the stack when the call returns. If the value at a particular memory address is not written to during the course of execution of the program, write a dash (–) in it. Give all values in hex.

Stack Address	Value
0xbfc5e4f0	0x00000002
0xbfc5e4ec	
0xbfc5e4e8	
0xbfc5e4e4	
0xbfc5e4e0	
0xbfc5e4dc	
0xbfc5e4d8	
0xbfc5e4d4	
0xbfc5e4d0	
0xbfc5e4cc	
0xbfc5e4c8	
0xbfc5e4c4	
0xbfc5e4c0	

Problem 11. (30 points):

Stack Smashing

You have recently taken an internship on the IT staff of a start-up founded by recent CMU graduates. In order to take advantage of his systems programming skills, the company chose CS superstar Harry Q. Bovik to head up the IT department.

Harry decided to run all of the network services off of a single Linux server priced at $3,235,430.00. However, since he was not an ECE student, Harry never learned much about computer security. Instead, he spent most of his time analyzing the arcane properties of splay trees and skip lists. Not surprisingly, Harry completely botched the security on his multi-million dollar server.

You have been assigned to manage one of the several services running on the server. You can not run the service executable directly. Instead, you must run a small C program that Harry wrote which asks you for a password and, upon inputting the correct password, runs the service executable with root privileges (using the setuid/setgid mechanism).

One of the first things you notice about Harry's program is that if you type in a password that is too long the program causes a segmentation fault. Using your knowledge from 213, you suspect Harry's program is vulnerable to a buffer overflow attack via the password input code. Being a good 213 student, you immediately attempt to exploit this vulnerability for fun and profit!

Unfortunately, you do not have access to the source code, but you can copy the 32-bit executable to your local machine and examine it using GDB. An assembly dump of the code is included on the following pages.

```
Dump of assembler code for function main:
0x080484b4 <main+0>:    push   %ebp
0x080484b5 <main+1>:    mov    %esp,%ebp
0x080484b7 <main+3>:    sub    $0x18,%esp
0x080484ba <main+6>:    and    $0xfffffff0,%esp
0x080484bd <main+9>:    mov    $0x0,%eax
0x080484c2 <main+14>:   add    $0xf,%eax
0x080484c5 <main+17>:   add    $0xf,%eax
0x080484c8 <main+20>:   shr    $0x4,%eax
0x080484cb <main+23>:   shl    $0x4,%eax
0x080484ce <main+26>:   sub    %eax,%esp
0x080484d0 <main+28>:   call   0x804851e <ckpass>
0x080484d5 <main+33>:   mov    %eax,0xfffffffc(%ebp)
0x080484d8 <main+36>:   cmpl   $0x0,0xfffffffc(%ebp)
0x080484dc <main+40>:   je     0x80484f6 <main+66>
0x080484de <main+42>:   movl   $0x80486f8,(%esp)
0x080484e5 <main+49>:   call   0x80483bc <_init+104>
0x080484ea <main+54>:   movl   $0x1,(%esp)
0x080484f1 <main+61>:   call   0x80483cc <_init+120>
0x080484f6 <main+66>:   movl   $0x8048707,0xfffffff8(%ebp)
0x080484fd <main+73>:   movl   $0x0,0x8(%esp)
0x08048505 <main+81>:   mov    0xfffffff8(%ebp),%eax
0x08048508 <main+84>:   mov    %eax,0x4(%esp)
0x0804850c <main+88>:   mov    0xfffffff8(%ebp),%eax
0x0804850f <main+91>:   mov    %eax,(%esp)
0x08048512 <main+94>:   call   0x804837c <_init+40>
0x08048517 <main+99>:   mov    $0x0,%eax
0x0804851c <main+104>:  leave
0x0804851d <main+105>:  ret
End of assembler dump.

Dump of assembler code for function ckpass:
0x0804851e <ckpass+0>:  push   %ebp
0x0804851f <ckpass+1>:  mov    %esp,%ebp
0x08048521 <ckpass+3>:  sub    $0x38,%esp
0x08048524 <ckpass+6>:  movl   $0x10,0x8(%esp)
0x0804852c <ckpass+14>: movl   $0x0,0x4(%esp)
0x08048534 <ckpass+22>: lea    0xffffffe8(%ebp),%eax
0x08048537 <ckpass+25>: mov    %eax,(%esp)
0x0804853a <ckpass+28>: call   0x80483dc <_init+136>
0x0804853f <ckpass+33>: lea    0xffffffe8(%ebp),%eax
0x08048542 <ckpass+36>: mov    %eax,(%esp)
0x08048545 <ckpass+39>: call   0x804839c <_init+72>
0x0804854a <ckpass+44>: lea    0xffffffe8(%ebp),%eax
0x0804854d <ckpass+47>: mov    %eax,0x4(%esp)
0x08048551 <ckpass+51>: lea    0xffffffd8(%ebp),%eax
0x08048554 <ckpass+54>: mov    %eax,(%esp)
0x08048557 <ckpass+57>: call   0x8048571 <hashpass>
0x0804855c <ckpass+62>: lea    0xffffffd8(%ebp),%eax
0x0804855f <ckpass+65>: movl   $0x80486e8,0x4(%esp)
0x08048567 <ckpass+73>: mov    %eax,(%esp)
0x0804856a <ckpass+76>: call   0x804838c <_init+56>
0x0804856f <ckpass+81>: leave
0x08048570 <ckpass+82>: ret
End of assembler dump.
```

1. `execl`, where are you?

   The `execl` function is a C library function which executes the specified program with the specified arguments. It has the following prototype:

   ```
   int execl(const char* program, char* arg0, ...);
   ```

   The argument `program` is a pointer to a string containing the full path name of the program's executable file. The `arg0` argument begins a variable argument list which is passed to the program. By convention, `arg0` is always set equal to `program`. The argument list is terminated by a null pointer.

   Harry has used this function to transfer control from his C program to the actual server executable if the password is correct.

   From your experience running the program, you deduce that the main function has a structure similar to the following:

   ```
   int main(int argc, char* argv[])
   {
     int result;

     result = ckpass();
     if (result != 0)
     {
       printf("Bad password!\n");
       exit(EXIT_FAILURE);
     }

     char* server = "/usr/bin/server";
     execl(server, server, NULL);

     return 0;
   }
   ```

   However, GDB was unable to determine which of the calls in main are actually to `execl`.

   Question: Using the above assembler dump of the main function, what is the address of the first instruction of `execl`?

2. Draw the stack.

   Make a drawing of the stack frame of `ckpass` and the argument-build and return address areas of the stack frame of `main` when it calls `ckpass`. You do not need to specify the memory addresses used by the stack, just give names to each value pushed onto the stack (e.g. return address, argument (n), saved %ebp, array of (n) bytes, etc.).

3. Dis-dis-assembling...

   In order to get a better understanding of where the buffer overflow occurs, you attempt to reconstruct the C source of the `ckpass` function.

   For your reference, the three unknown function prototypes are listed below:

```
0x80483dc <_init+136> -> memset(void* address, int val, int n);
```

   Sets `n` bytes starting at the given address to `val` (shortened to 8 bits).

```
0x804838c <_init+72> -> gets(char* buf);
```

   Reads in one line of any length from standard input, copying the string into `buf`.

```
0x804837c <_init+56> -> strcmp(const char* s1, const char* s2);
```

   Compares the strings `s1` and `s2` for equality. Returns zero if the strings are equal, nonzero otherwise.

   Also, `0x80486cc` is the address of a string representing the correct password hash. It is declared as a global variable:

```
const char* good_hash = "...";
```

   Task: Fill in the `ckpass` function below:

```
int ckpass()
{
  char a[____];

  char b[____];

  memset(_________, _____, _____);

  gets(________);

  hashpass(b, a);

  return strcmp(________, good_hash);
}
```

4. Where's the exploit?

   Now that you have reconstructed the source to `ckpass`, BRIEFLY explain why this code has a buffer overflow vulnerability.

5. ...?

   Now that you have identified a stack buffer overflow vulnerability and have a clear picture of the stack, you need to figure how to exploit it for fun and profit (mostly profit...)! Since the system contains all of the data for the company, you are itching to modify payroll data in your favor. In order to do a complex task such as this, you have written a program to make the necessary modifications (`/home/213student/hax`). You just need to execute the program using root privileges.

   Fortunately, `execl`, the function Harry used to execute the server, will also execute your program – and you even know its address! Basically, given the documentation of `execl`, you need to execute an equivalent of the following code:

   ```
   char* hax = "/home/213student/hax";
   execl(hax, hax, NULL);
   ```

   Unfortunately, you are unable to insert executable code into your exploit string due to restrictions imposed by the kernel. You will have to use some other mechanism.

   Explain how you can call `execl` correctly by inputting a carefully designed password string. Be sure to describe how to build the arguments for `execl`! No more than 4 sentences should be necessary.

   HINT: `execl` does not return!

6. Profit!

   Show an implementation of your exploit by drawing a picture of the stack after your exploit code has been read in by `gets`. Also indicate where the stack pointer `%esp` is pointing right before `ckpass` returns. **Only draw the portion of the stack overwritten by your exploit code.**

   For reference, the return address pushed onto the stack when `main` calls `ckpass` is located at STACK address `0xffffce2c`. You may assume that this will not change between multiple executions of Harry's program.

Problem 12. (20 points):

Multithreading

For the entirety of this question, assume that the compiler performs no optimization and that all code runs on the processor exactly as written. Also assume that no library calls will fail.
Suppose we have a program as follows:

```
#include <stdio.h>
#include <pthread.h>

int i = 0;

void *do_stuff(void *arg __attribute__((unused))) {
    i++;
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, do_stuff, NULL);
    pthread_create(&tid2, NULL, do_stuff, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("%d\n", i);
    return 0;
}
```

Recall that because `i` is a global variable, `i++;` will compile to something like this:

```
  400728:       8b 04 25 40 10 60 00    mov    0x601040,%eax
  40072f:       83 c0 01                add    $0x1,%eax
  400732:       89 04 25 40 10 60 00    mov    %eax,0x601040
```

1. What are all possible outputs of this program? For each output, explain how the kernel could interleave execution of the two child threads to produce it.

Suppose we alter `do_stuff` to look as follows:

```
void *do_stuff(void *arg __attribute__((unused))) {
    int a;
    for (a = 0; a < 1000; a++)
        i++;
    return NULL;
}
```

Because the code is not optimized, there will be one load-increment-store sequence per iteration of the loop.

2. For each number, tell whether or not our program could output it, and briefly explain why or why not.

   - 2000
   - 1500
   - 2
   - 1

Your programming partner Harry Q. Bovik notices that your code has some race conditions, and draws up the following locking mechanism:

```
int locked = 0;
void lock() {
    while (locked == 1) {
        continue;
    }
    locked = 1;
}
void unlock() {
    locked = 0;
}
```

Because the load-increment-store sequence is the critical section of your program, you place a call to `lock()` immediately before the `i++` line, and a call to `unlock()` immediately after. However, running your supposedly-now-threadsafe program again, you discover that the output is still nondeterministic. Turns out your partner's clever locking scheme doesn't do a very good job protecting the critical section after all.

3. Give an execution sequence of two threads in the `lock()` function that would end up with both threads holding the lock at the same time.