Andrew login ID: _____

Full Name: _____

Section: _____

15-213/18-243, Fall 2010

Exam 2

Tuesday, November 9. 2010

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID, full name, and section on the front.
- This exam is closed book, closed notes, although you may use a single 8 1/2 x 11 sheet of paper with your own notes. You may not use any electronic devices.
- 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 67 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

1 (10):	
2 (06):	
3 (12):	
4 (09):	
5 (09):	
6 (12):	
7 (09):	
TOTAL (67):	

Problem 1. (10 points):

Multiple choice on a variety of topics. Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10

1. For the Unix linker, which of the following accurately describes the difference between global symbols and local symbols?

   (a) There is no functional difference as to how they can be used or how they are declared.

   (b) Global symbols can be referenced by other modules (files), but local symbols can only be referenced by the module that defines them.

   (c) Global symbols refer to variables that are stored in `.data` or `.bss`, while local symbols refer to variables that are stored on the stack.

   (d) Both global and local symbols can be accessed from external modules, but local symbols are declared with the "static" keyword.

2. Consider the following C variable declaration

   ```
   int *(*f[3])();
   ```

   Then `f` is

   (a) an array of pointers to pointers to functions that return int

   (b) a pointer to an array of functions that return pointers to int

   (c) a function that returns a pointer to an array of pointers to int

   (d) an array of pointers to functions that return pointers to int

   (e) a pointer to a function that returns an array of pointers to int

   (f) a pointer to an array of pointers to functions that return int

   Hint: Recall from the K&R book that `[]` and `()` have higher precedence than `*`.

3. Which of the following is true concerning dynamic memory allocation?

   (a) External fragmentation is caused by chunks which are marked as allocated but actually cannot being used.

   (b) Internal fragmentation is caused by padding for alignment purposes and by overhead to maintain the heap data structure (such as headers and footers).

   (c) Coalescing while traversing the list during calls to malloc is known as immediate coalescing.

   (d) Garbage collection, employed by calloc, refers to the practice of zeroing memory before use.

For the next 3 questions, consider the following code running on a 32-bit Linux system. Recall that `calloc` zeros the memory that it allocates.

```
int main()
{
    long a, *b, c;
    char **p;

    p = calloc(8, sizeof(char)); /* calloc returns 0x1dce1000 */

    a = (long) (p + 0x100);
    b = (long *) (*p + 0x200);
    c = (int) (b + 0x300);

    printf("p=%p a=%x b=%p c=%x\n", p, a, b, c);

    exit(0);
}
```

4. When `printf` is called, what is the hex value of variable a?
   (a) Can't tell
   (b) 0x1dce1100
   (c) 0x1dce1400
   (d) 0x1dce1800

5. When `printf` is called, what is the hex value of variable b?
   (a) Can't tell
   (b) 0x1dce1200
   (c) 0x1dce2000
   (d) 0x200
   (e) 0x800

6. When `printf` is called, what is the hex value of variable c?
   (a) Can't tell
   (b) 0x1dce2a00
   (c) 0x1dce4400
   (d) 0xc00
   (e) 0xe00

7. Which of the following is **not** a default action for any signal type?

   (a) The process terminates.

   (b) The process reaps the zombies in the waitlist.

   (c) The process stops until restarted by a SIGCONT signal.

   (d) The process ignores the signal.

   (e) The process terminates and dumps core.

8. A system uses a two-way set-associative cache with 16 sets and 64-byte blocks. Which set does the byte with the address `0xdeadbeef` map to?

   (a) Set 7

   (b) Set 11

   (c) Set 13

   (d) Set 14

9. When it succeeds, `execve` is called once and returns how many times?

   (a) 0

   (b) 1

   (c) 2

   (d) 3

10. When it suceeds, `fork` is called once and returns how many times?

    (a) 0

    (b) 1

    (c) 2

    (d) 3

Problem 2. (6 points):

Linking. Consider the executable object file `a.out`, which is compiled and linked using the command

```
unix> gcc -o a.out main.c foo.c
```

and where the files `main.c` and `foo.c` consist of the following code:

```
/* main.c */
#include <stdio.h>

static int a = 1;
int b = 2;
int c;

int main()
{
    int c = 3;

    foo();
    printf("a=%d, b=%d, c=%d\n", a, b, c);
    return 0;
}
```

```
/* foo.c */
int a, b, c;

void foo()
{
    a = 4;
    b = 5;
    c = 6;
}
```

What is the output of `a.out`?

Answer: `a=____, b=_____, c=_____`

Problem 3. (12 points):

Cache Operation In this problem, you are asked to simulate the operation of a cache. You can make the following assumptions:

- There is only one level of cache
- Physical addresses are 8 bits long ($m = 8$)
- The block size is 4 bytes ($B = 4$)
- The cache has 4 sets ($S = 4$)
- The cache is direct mapped ($E = 1$)

(a) What is the total capacity of the cache? (in number of data bytes)

(b) How long is a tag? (in number of bits)

(c) Assuming that the cache starts clean (all lines invalid), please fill in the following tables, describing what happens with each operation. Addresses are given in both hex and binary for your convenience.

Operation	Set index?	Hit or Miss?	Eviction?
load 0x00 $(0000\ 0000)_2$	0	miss	no
load 0x04 $(0000\ 0100)_2$		miss	
load 0x08 $(0000\ 1000)_2$			
store 0x12 $(0001\ 0010)_2$	0		
load 0x16 $(0001\ 0110)_2$			
store 0x06 $(0000\ 0110)_2$			
load 0x18 $(0001\ 1000)_2$	2		
load 0x20 $(0010\ 0000)_2$			
store 0x1A $(0001\ 1010)_2$			

Problem 4. (9 points):

Signals. Consider the following three different snippets of C code. Assume that all functions and procedures return correctly and that all variables are declared and initialized properly. Also, assume that an arbitrary number of SIGINT signals, and only SIGINT signals, can be sent to the code snippets randomly from some external source.

For each code snippet, circle the value(s) of `i` that could possibly be printed by the `printf` command at the end of each program. *Careful: There may be more than one correct answer for each question. Circle all the answers that could be correct.*

Code Snippet 1:

```
int i = 0;

void handler(int sig) {
  i = 0;
}

int main() {
  int j;

  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  exit(0);
}
```

1. Circle possible values of `i` printed by snippet 1:

   1. 0
   2. 1
   3. 50
   4. 100
   5. 101
   6. Terminates with no output.

Code Snippet 2:

```
int i = 0;

void handler(int sig) {
  i = 0;
}

int main () {
  int j;
  sigset_t s;

  signal(SIGINT, handler);

  /* Assume that s has been
     initialized and declared
     properly for SIGINT */

  sigprocmask(SIG_BLOCK, &s, 0);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  sigprocmask(SIG_UNBLOCK, &s, 0);
  printf("i = %d\n", i);
  exit(0);
}
```

2. Circle possible values of `i` printed by snippet 2:

   1. 0
   2. 1
   3. 50
   4. 100
   5. 101
   6. Terminates with no output.

Code Snippet 3:

```
int i = 0;

void handler(int sig) {
  i = 0;
  sleep(1);
}


int main () {
  int j;
  sigset_t s;

  /* Assume that s has been
     initialized and declared
     properly for SIGINT */

  sigprocmask(SIG_BLOCK, &s, 0);
  signal(SIGINT, handler);
  for (j=0; j < 100; j++) {
    i++;
    sleep(1);
  }
  printf("i = %d\n", i);
  sigprocmask(SIG_UNBLOCK, &s, 0);
  exit(0);
}
```

3. Circle possible values of `i` printed by snippet 3:

   1. 0
   2. 1
   3. 50
   4. 100
   5. 101
   6. Terminates with no output.

Problem 5. (9 points):

Unix process control. Consider the C program below. For space reasons, we are not checking error return codes, so assume that all functions return normally.

Part 1

```
int main () {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("9");
            exit(1);
        }
        else
            printf("5");
    }
    else {
        pid_t pid;
        if ((pid = wait(NULL)) > 0) {
             printf("3");
        }
    }
    printf("0");
    return 0;
}
```

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program.

A. `93050` Y N

B. `53090` Y N

C. `50930` Y N

D. `39500` Y N

E. `59300` Y N

Part 2

Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int i = 0;

int main () {
    int j;
    pid_t pid;

    if ((pid = fork()) == 0) {
        for (j = 0; j < 20; j++)
            i++;
    }
    else {
        wait(NULL);
        i = -1;
    }

    if (i < 0)
        i = 10;

    if (pid > 0)
        printf("Parent: i = %d\n", i);
    else
        printf("Child: i = %d\n", i);

    exit(0);
}
```

What are the outputs of the two `printf` statements?

`Parent: i = _____`

`Child : i = _____`

Problem 6. (0xc points):

Address translation. This problem deals with virtual memory address translation using a multi-level page table, in particular the 2-level page table for a 32-bit Intel system with 4 KByte pages tables. The following diagrams are direct from the Intel System Programmers guide and should be used on this problem:

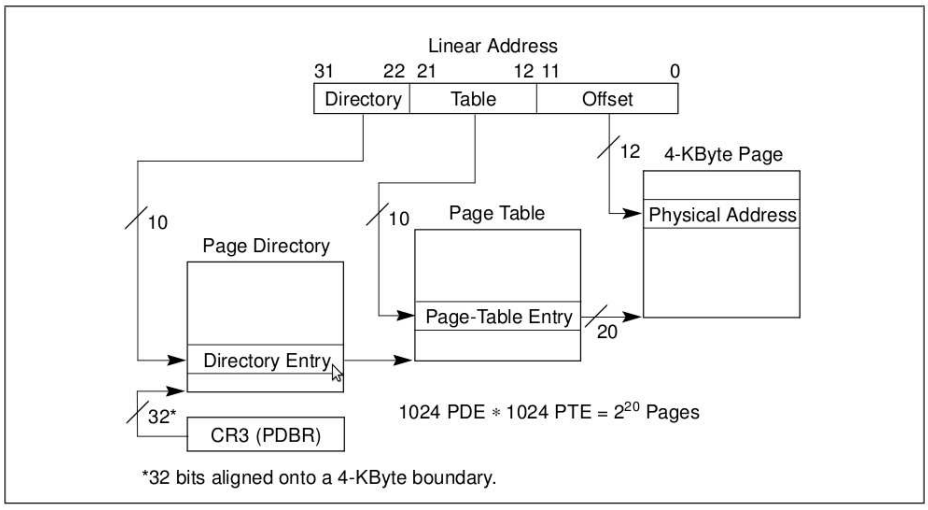

Figure 3-12. Linear Address Translation (4-KByte Pages)

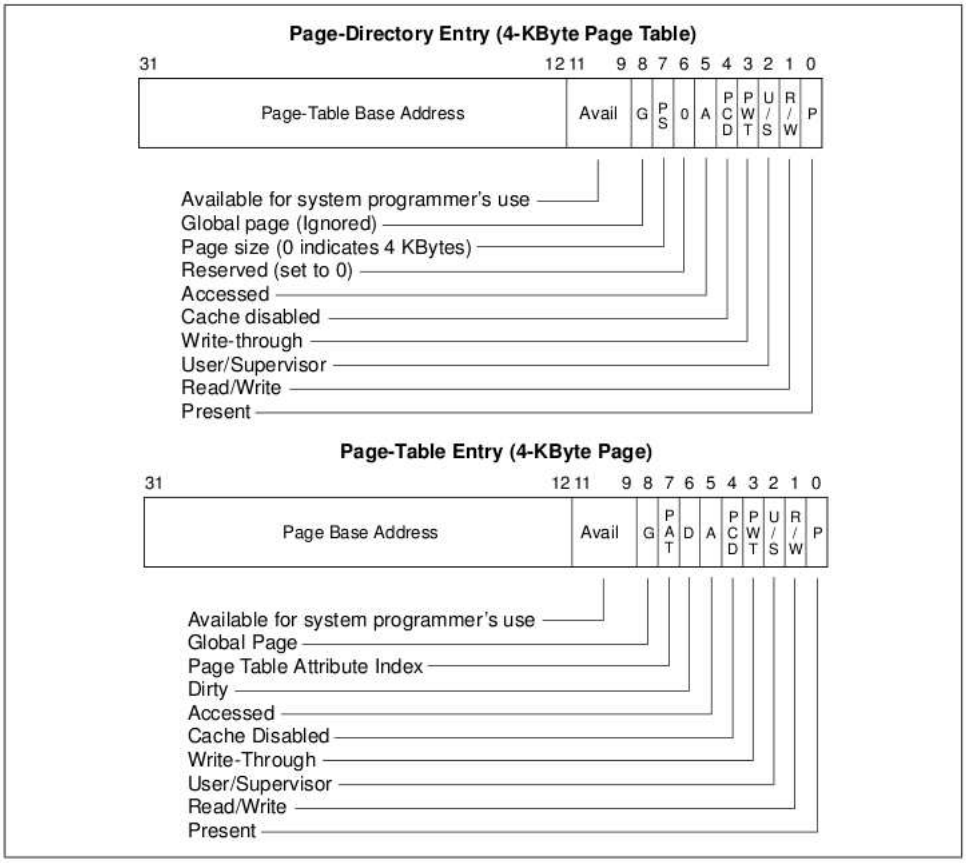

Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

The contents of the relevant sections of memory are shown on this page. All numbers are given in **hexadecimal**. Any memory not shown can be assumed to be zero. The Page Directory Base Address is `0x0c23b000`.

For each of the following problems, perform the virtual to physical address translation. If an error occurs at any point in the address translation process that would prevent the system from performing the lookup, then indicate this by circling FAILURE and noting the physical address of the table entry that caused the failure.

For example, if you were to detect that the present bit in the PDE is set to zero, then you would leave the PTE address in (b) empty, and circle FAILURE in (c), noting the physical address of the offending PDE.

Address	Contents
00023000	beefbee0
00023120	12fdc883
00023200	debcfd23
00023320	d2e52933
00023FFF	bcdeff29
00055004	8974d003
0005545c	457bc293
00055460	457bd293
00055464	457be293
0c23b020	01288b53
0c23b040	012aab53
0c23b080	00055d01
0c23b09d	0FF2d303
0c23b274	00023d03
0c23b9fc	2314d222
2314d200	0fdc1223
2314d220	d21345a9
2314d4a0	d388bcbd
2314d890	00b32d00
24AEE520	b58cdad1
29DE2504	56ffad02
29DE4400	2ab45cd0
29DE9402	d4732000
29DEE500	1a23cdb0

1. Read from virtual address `0x080016ba`.
   Scratch space:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

   (a) Physical address of PDE: 0x

   (b) Physical address of PTE: 0x

   (c) (SUCCESS) The physical address accessed is: 0x

   OR

   (FAILURE) The physical address of the table entry causing the failure is: 0x

2. Read from virtual address `0x9fd28c10`. Scratch space:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(a) Physical address of PDE: 0x

(b) Physical address of PTE: 0x

(c) (SUCCESS) The physical address accessed is: 0x

OR

(FAILURE) The physical address of the table entry causing the failure is: 0x

Problem 7. (9 points):

Unix I/O. You are given two text files `file_1.txt` and `file_2.txt`. Each of them contains exactly a single word without any whitespace or special characters. Their contents are shown in the table below.

File name	File contents
file_1.txt	`file`
file_2.txt	`descriptors`

You are also given two programs (headers omitted), both of which are independent of one another, i.e., the order of execution does not matter. They use the simple and familiar UNIX system functions to perform file I/O operations.

For each program, determine what will be the output to `stdout`, based on the file contents as shown above. Assume that after each program finish execution, the file contents will be reset to that shown above. Also, assume that all system calls will succeed and the files are in the same directory as the two programs.

Program 1: (4 points)

```
/* buf is initialized to be all zeroes */
char buf[20] = {0};

int main(int argc, char* argv[]) {
    int fd1, fd2 = open("file_1.txt", O_RDONLY);

    fd1 = dup(fd2);
    read(fd2, buf, 3);
    close(fd2);
    read(fd1, &buf[3], 1);

    printf("%s", buf);

    /* Don't worry about file descriptors not being closed */
    return 0;
}
```

output to `stdout` from Program 1:	

Program 2: (5 points)

```
/* buf is initialized to be all zeroes */
char buf[20] = {0};

int main(int argc, char* argv[]) {
    pid_t pid;
    int fd1, fd2, fd3 = open("file_2.txt", O_RDONLY);

    read(fd3, buf, 1);
    fd1 = dup(fd3);

    if ((pid = fork()) > 0){
        waitpid(pid, NULL, 0);
        read(fd1, &buf[1], 2);
    } else {
        fd2 = open("file_2.txt", O_RDONLY);
        read(fd1, &buf[1], 1);
        read(fd2, &buf[2], 2);
    }

    printf("%s", buf);

    /* Don't worry about file descriptors not being closed */
    return 0;
}
```

output to `stdout` from Program 2:	