Fall term 2012
KAIST EE209 Programming Structures for EE

Final exam

Thursday Dec 20, 2012

Student's name: ____

Student ID: ____

The exam is closed book and notes. Read the questions carefully and focus your answers on what has been asked. You are allowed to ask the instructor/TAs for help only in understanding the questions, in case you find them not completely clear. Be concise and precise in your answers and state clearly any assumption you may have made. All your answers must be included in the attached sheets. You have 120 minutes to complete your exam. Be wise in managing your time.

Scores

Question 1	/10
Question 2	/20
Question 3	/15
Question 4	/10
Question 5	/10
Question 6	/15
Total	/80

1. Briefly explain following terms or answer the question (10 pt)

(a) Virtual memory (2pt)

Ans) It refers to the virtual address space of a process that is mapped to physical memory space by address translation. Virtual memory gives the illusion that each process owns its independent memory space while actual physical memory is shared by multiple processes.

(b) Page table (2pt)

Ans) It refers to per-process table that maps a virtual memory page to a physical memory page. It is maintained by the kernel to support virtual memory per process.

(c) External fragmentation (2pt)

Ans) It refers to the interspersed memory fragments that are too small to be allocated to satisfy a large memory allocation request. In the worst case, a memory allocation request could fail even if the sum of free fragments could satisfy the request because they are scattered around.

(d) Trap (2pt)

Ans) It refers to a synchronous exception that arises as part of a system call. A trap happens when the control flow has to go to the kernel mode to execute a system call function.

(e) What is the difference between read(int fd, void* buf, size_t count) (system call) and fread(void* ptr, size_t size, size_t nmemb, FILE *stream) (C library function)? (2 pt)

Ans) read() incurs context switching to the kernel mode while fread() could satisfy the request by returning buffered data. In case the buffered data is not enough, it issues the read() system call to read the data from the kernel.

2. Fun programming (20 pt)

a) Assume a[99] has 99 distinct natural numbers from 1 to 100. Fill out the following function that finds the missing number out of 1 to 100 that does not exist in a[99]. Be concise and be cautious about memory use (including the stack usage) (5pt)

```
int find_missing(int a[99])
{

    int max = 5050; /* sum of 1 to 100 */
    int i;

    for (i = 0; i < 99; i++)
        sum -= a[i];

    return sum;

}
```

b) The function stack grows from high to low address in the Intel architecture. Assume the stack could grow low to high or high to low address in general. Write a function that returns 1 if the stack grows from high to low address, 0 if it grows from low to high address. You can add another function if you want. (5pt)

```
Ans)

void *dummy_func(void)
{
        int i;
        return (void *)&i;
}

int does_stack_grow_to_low_address(void)
{

     int i;
     return (dummy_func() - (void *)&i < 0) ? 1 : 0;

}
```

c) What does this function do? (5pt)
```
typedef struct node {
 void *data;
 struct node *next;
} Node;

int f(Node *p)
{
  Node *p1, *p2;

  p1 = p2 = p;
  if (p == NULL) return 0;
  do {
     p1 = p1->next;
     p2 = p2->next;
     if (p2 == NULL)
        return 0;
     p2 = p2->next;
   } while (p1 != NULL && p2 != NULL && p1 != p2);

  if (p1 == NULL || p2 == NULL)
      return 0;

  return 1;
}
```
Original Ans) It detect whether the linked list has any loop (returns 1) or not (returns 0). Note that one pointer moves to the next node while another pointer skips the next node and move to the next node of the immediate next node. If there's no loop, one of the two pointers will end up at the NULL pointer, but if there's a loop, these two pointers will eventually meet.

But the problem is flawed without the boldface statement. So, everyone gets full points for this problem.

d) What's the output of the second printf() (printf("parent ..."))? If it's called multiple times, show every possible output (5pt).
```
int g = 1;

int main(void)
{
 int k = 1;

 if (fork() == 0) {
        k++; g++;
        printf("child k+g=%d\n", k+g);
 }
 g--;
 k += 2;
 wait(NULL);
 printf("parent k+g=%d\n", k+g);
 return 0;
}
```
Ans) parent k+g=3,   parent k+g=5

3. The following assembly code was generated by gcc209 by compiling a simple C function (named f). It takes one signed integer parameter and returns a signed integer value. You may assume the passed-in parameter is in the range of 1 to 40. (15pt)

```
file    "f.c"
        .text
        .globl  f
        .type   f, @function
f:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $20, %esp
        cmpl    $1, 8(%ebp)
        jg      .L2
        movl    $1, %eax
        jmp     .L3
.L2:
        movl    8(%ebp), %eax
        decl    %eax
        movl    %eax, (%esp)
        call    f
        movl    %eax, %ebx
        movl    8(%ebp), %eax
        subl    $2, %eax
        movl    %eax, (%esp)
        call    f
        addl    %ebx, %eax
.L3:
        addl    $20, %esp
        popl    %ebx
        popl    %ebp
        ret
```

(a) At label .L2, what do 4(%ebp) and 8(%ebp) have? (4pt, 2pt each)

Ans) 4(%ebp) => Old EIP,   8(%ebp) => the integer parameter

(b) What's the value of f(5)? (4pt)

Ans) f(5) = 8

(c) Write the equivalent code in C. (5pt)

Ans) f() calculates a Fibonacci number of the parameter.

```
int f(int n)
{
  if (n <= 1) return 1;
  return f(n-1) + f(n-2);
}
```

(d) It looks that the assembly code is inefficient in stack memory usage. How would you change the code to reduce the unnecessary stack memory usage? (2pt)

Ans) "subl $20, %esp" uses unnecessary 16B more memory. This space is used to pass the parameter to its own function, so it suffices to have only 4 bytes.

```
subl $4, %esp
…

addl $4, %esp
```

4. Memory management (10 pt)

   a) List three kinds of cache misses and explain each of them. (3pt)

   Ans)

   - Compulsory(/cold) miss: happens if the memory item is first accessed
   - Capacity miss: happens if the cache is too small to hold previously-accessed items
   - Conflict miss: happens if an accessed item is mapped to the same cache block which is mapped by another item. If these items are accessed, they compete for the same cache block, leading to cache misses even if the cache has enough capacity.

   b) What is the “first fit” memory allocation strategy? Discuss the advantages and disadvantages of “first fit”? (3pt)

   Ans) “First fit” refers to the allocation policy that handles the memory allocation request as soon as the system finds a big enough free memory chunk in the free list that satisfies the request. It is simple and fast in terms of allocation, but it could lead to severe external fragmentation.

c) Write a program that shows the minimum memory allocable chunk unit including the header/footer overhead. (4pt)

Ans)
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 #define MAX 10

 char *p[MAX];
 int min = 1000000;
 int i;

 for (i = 0; i < MAX; i++) {
        p[i] = malloc(1);  /* 1 byte allocation */
        if (i > 0) {
          int t = p[i] - p[i-1];
          if (t < min)
               min = t;
        }
 }

 for (i = 0; i < MAX; i++)
        free(p[i]);


 printf("minimum chunk size = %d\n", min);
 return 0;

}
```

5. Exceptions and Process control (10pt)

(a) What is the difference between function calls and exceptions. List at least three. (3pt)

Ans)

1) Exceptions switch from user mode to kernel mode (privileged mode)
2) When an exception happens, the processor pushes data onto OS's stack instead of application program's stack
3) Processor pushes additional states (like values of all registers) onto stack
4) Flow control does not always return to the next instruction. It could return to the current instruction or it could never return.

(b) Process context refers to the state that each process maintains. What does the context consist of? List at least three. (3pt)

Ans)

1) Process state: new, ready, waiting, terminated
2) CPU registers: EIP, EFLAGS, EAX, EBX…
3) I/O status information: Open file descriptor table, I/O requests, …
4) Memory management information: page tables
5) Accounting information: Time limit, group ID, …
6) CPU scheduling information: process priority, queues, …

d) Explain each of the following system call functions in one sentence (4pt)

A. fork(): spawns a new identical process as the parent process

B. exec(): converts the current process to the given program

C. wait(): blocks/waits until one of its child processes ends

D. kill(): sends a signal to a process

6. Big integer operations (15pt)
   We are writing a library that can add and multiply two unsigned large integer values that cannot be represented by the C's built-in integer type (unsigned int, unsigned long, unsigned long long). Assume that the largest integer in the library can be represented by 32 * sizeof(unsigned int) bytes. u_int is typedef'ed to be unsigned int.

   For example, 0x11111111222222223333333344444444 can be represented by an integer array of size 4. That is

```
u_int a[16] = {0};

a[0] = 0x44444444
a[1] = 0x33333333
a[2] = 0x22222222
a[3] = 0x11111111
```

   represents 0x11111111222222223333333344444444. a[3] represents the most significant four bytes whereas a[0] represents the least significant four bytes.

   a) add_large() takes two unsigned big integers (a[16], b[16]) and write the sum to c[17]. Please fill out the function. (5pt)

```
void add_large(u_int a[16], u_int b[16], u_int c[17])
{

        int i;

        for (i = 0; i <16; i++)
            c[i] = 0;                   /* initialize the results to 0 */
        /* or   memset(c, 0, 17 * sizeof(u_int));   */

        for (i = 0; i < 16; i++) {
            u_int temp = a[i] + b[i];
            if (temp < a[i] || temp < b[i])
              c[i+1] = 1;                 /* carry from the lower elements */
            c[i] += temp;
  }
```

b) multiply_large() takes two unsigned big integers (a[16], b[16]) and writes the result of the multiplication of the values into c[32]. Please fill out the function. (10pt)

```
void multiply_large(u_int a[16], u_int b[16], u_int c[32])
{
     int i, j, k;


    /* initialization */
     for (i = 0; i < 32; i++)
         c[i] = 0;

    for (i = 0; i < 16; i++) {
          for (j = 0; j < 16; j++) {
              u_int sum, low, high;
             unsigned long long temp;
             int carry = 0;

             /* 32-bit multiplication would produce a 64-bit result */
             temp = ((unsigned long long)a[i]) * b[j];
             low = ((u_int *)&temp)[0];
             high = ((u_int *)&temp)[1];

             /* low order 32- bit addition and carry propagation */
             k = i + j;
             sum = c[k] + low;
             carry = (sum < c[k] || sum < low);    /* got a carry */
             c[k++] = sum;
             if (carry) {c[k]++; while (c[k] == 0) c[++k]++;}

             /* high order 32-bit addition and carry propagation */
             k = i+j +1;
             sum = c[k] + high;
             carry = (sum < c[k] || sum < high); /* got a carry */
             c[k++] = sum;
             if (carry) {c[k]++; while (c[k] == 0) c[++k]++;}
          }
     }
}
```