Spring Semester 2017

KAIST EE209

Programming Structures for Electrical Engineering

Final Exam

Name: _____

Student ID: _____

This exam is open 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. You have 165 minutes (1:00 PM – 3:45 PM) to complete your exam. Be wise in managing your time. Good luck.

Question 1	/ 25
Question 2	/ 20
Question 3	/ 15
Extra credit	/ 10
Question 4	/ 20
Question 5	/ 20

Total	/ 110

Name: Student ID:

Read before you start: The code snippets in the exam omit error handling due to space constraint. In real-world code, you should handle all errors properly. Please ***assume all system call functions succeed*** for the purpose of the exam problems.

1. (25 points) Multiple processes

```
int main()
{
   int x = 3;
   if (fork() != 0)
      printf("x=%d\n", ++x);
   printf("x=%d\n", --x);
   return 0;
 }
```

(a) Give all possible output sequences of the code above. (5 points)

⇨ 432, 243, 423 (new line (\n) after each character as well)

```
void funfork(int n) {
   int i;
   for (i = 0; i < n; i++)
      fork();
   printf("hi\n");
   exit(0);
}
```

(b) In the code above, how many lines of output will you see when funfork() is called? Give the number in terms of n (n >=1). (5 points)

⇨ 2^n

Name: Student ID:

```
int main() {
   if (fork() == 0) {
     printf("a"); fflush(stdout);
   } else {
     printf("b"); fflush(stdout);
     waitpid(-1, NULL, 0);
   }
   printf("c"); fflush(stdout);
   exit(0);
}
```

(c) What are the possible output sequences of the code above? Write all possible output sequences. (5 points)

⇨ acbc, abcc, bacc

Name: Student ID:

```
void end(void)
{  printf("2"); fflush(stdout); }


int main()
{
   if (fork() == 0) atexit(end);
   if (fork() == 0) {
      printf("0"); fflush(stdout);
   } else {
      printf("1"); fflush(stdout);
   }
   exit(0);
}
```

(d) Which of the following is possible for the output of the code above? Note that atexit() takes a pointer to a function, and adds it to a list of functions (initially empty) that will be called (in the reverse order of registration) when the exit() function is called. (5 points)

A. 112002
B. 211020
C. 102120
D. 122001
E. 100212

⇨ A, C, E

(e) Briefly describe what a zombie process is (2points). What should a programmer need to do to remove zombie processes? (3 points) (2+3=5 points)

⇨ A zombie process is a process that has finished execution but has not been reaped by its parent yet. One should call the wait (or waitpid) system call to get the return status of a child process (reaping). A typical way to implement it is to catch the SIGCHILD signal and have code like this

```
while (waitpid(-1, NULL,0) > 0));
```

You'll get full points even if you don't mention the SIGCHILD signal, and the code above.

Name: Student ID:

2. (20 points) Signal Programming

```
static void SigIntHandler(int sig)
{
   printf("caught SIGINT! \n");
}

int main()
{
   void (*pf)(int);
   pf = signal(SIGINT, SigIntHandler);
   assert(pf != SIG_ERR);
   while (1) {
     sleep(1);
     printf("catching a signal is fun!\n");
   }
   exit(0);
}
```

(a) Signal programming is so fun, so I wrote the following code to test the catching SIGINT signal. I compiled and ran the code, and confirmed that it runs well by showing "caught SIGINT!" whenever I typed in Ctrl+C. But now, I realize that I cannot kill the process with Ctrl+C any more since it has set up a signal handler for it. Describe how to kill the process without resorting to a nonsensical way such as turning off the machine. Describe the steps **in detail.** (5 points)

⇨ There could be multiple answers, and we will give full points for any reasonable answer.

Open up another shell. Find the process id with ps command in the new shell. Then, 'kill -9 pid' should kill the process.

Name: Student ID:

```
int counter = 0;
static void SigIntHandler(int sig)
{
   counter--;
}

int main()
{
   void (*pf)(int);
   int i;

   pf = signal(SIGINT, SigIntHandler);
   assert(pf != SIG_ERR);
   for (i = 0; i < 100; i++)
     sleep(1);
     counter++;
   }
   printf("count = %d\n", counter);
   exit(0);
}
```

(b) Now that I figured out how to kill the process whenever I want, I went on to write another toy program that increments a counter by one for each one second while it decrements the counter by one whenever I type in Ctrl+C. When I ran the code, and it looked working fine for almost all time, but in a very rare case, it shows an incorrect counter value at the end. Explain the bug in the code above. (5 points)

⇨ Global variable (counter) is accessed both by the main function and by the signal handler. They could experience race condition.

(c) Rewrite the portion of the code to fix the bug in (b). (10 points)

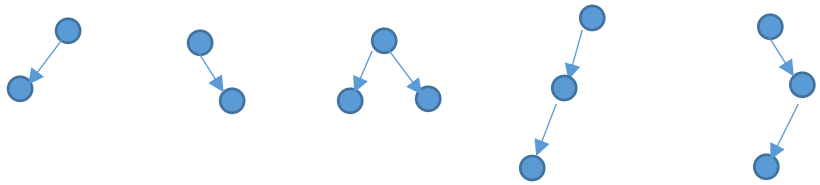
⇨ The loop in the main function could be rewritten like this. Boldface fonts are the added code.

```
void (*pf)(int);
int i;
sigset_t pset;

pf = signal(SIGINT, SigIntHandler);
assert(pf != SIG_ERR);

sigemptyset(&pset);
sigaddset(&pset, SIGINT);
for (i = 0; i < 100; i++)
    sleep(1);
    sigprocmask(SIG_BLOCK, &pset, NULL);
    counter++;
    sigprocmask(SIG_UNBLOCK, &pset, NULL);
}
```

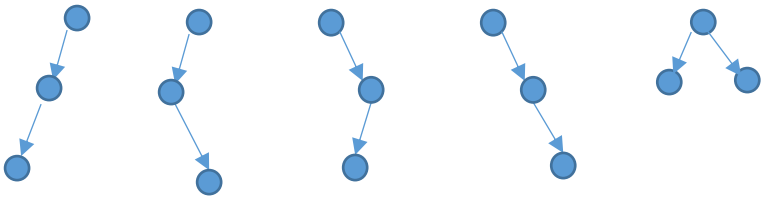
Name: Student ID:

3. (15 points, and 10 points for extra credit = total 25 points)
Counting the number of different binary trees.

A binary tree (in data structure) consists of nodes where each node can have up to two pointers (e.g., left and right pointers) to other nodes (called children nodes, and the node that points to a child node is called a parent node of the child node) and there is no cycle among the nodes if one follows the pointers. All nodes should be reachable by following the pointers from a root node. A root node is a node that does not have a parent node. The followings are examples of different binary trees. A circle is a node and an arrow represents a pointer. A pointer can be either left or right pointer.



Let's say Cn represents the number of different binary trees with n nodes. For example, C1 = 1 and C2 = 2. Assume C0 = 1. C3 = 5, as shown below.



(a) Describe Cn in relation with Ck where 0 <= k < n (5 points)

$$\Rightarrow C_n = \sum_{k=0}^{n-1} C_k * C_{n-1-k}$$

Name: Student ID:

(b) Write the function that calculates Cn where 0<= n <= 35. For these n, long (signed 64-bit integer) is enough to represent Cn (no overflow). (10 points for (b-1), and 10 points for (b-2) as extra credit)

(b-1) Write a recursive version. Hint: it would require less than 10 lines of code (10 points)

```
long Cn(int n)
{
  long result = 0;
  int i;

  assert(n >=0 && n <= 35);
  if (n == 0) return 1;

  for (i = 0; i < n; i++)
     result += Cn(i) * Cn(n-1-i);
  return result;
}
```

Name: Student ID:

(b-2) (extra credit) A naïve recursive function could be very slow for large n. One way to optimize is to have the function remember the intermediate Ck values (k < n) while calculating Cn (e.g., in a static array in the function) and use the stored values for any future queries. This strategy is called dynamic programming in general. Write a version that exploits dynamic programming – again, you can assume n is between 0 and 35. Hint: actually, only a few lines of code need to be added to the naïve recursive version. (10 points)

```
long Cn(int n)
{
  long result = 0;
  int i;
  static long cn[36] = {0}; /* initialize them to 0 */

  assert(n >=0 && n <= 35);

   /* return the value if we already know it */
  if (cn[n] != 0) return cn[n];

  if (n == 0) {cn[0] = 1; return 1;}
  for (i = 0; i < n; i++) {
     result += Cn(i) * Cn(n-1-i);
  }
  cn[n] = result; /* remember the value for n */
  return result;
}
```

Name: Student ID:

4. (20 points) Page tables and pointers

(a) (5 points) On 32-bit OS with 4KB page size, how many page table entries does one process have at maximum (3 points)? How much memory is required for the page table of a process if one entry takes up 4 bytes ?(2 points)

⇨ 20 bits are used for virtual pages, so a page table can have 2^20 entries at maximum. The memory consumption for that is 4B x 2^20 = 4 MB.

(b) (5 points) On 64-bit OS/CPU, the number of page table entries could be huge and naïve memory allocation for a page table even for one process could be larger than the entire physical memory on a machine. How can you reduce the memory consumption for a page table for 64-bit virtual addresses? (Actually, you can assume that only 48 bits are used instead of the entire 64-bit address, like in Intel CPU)

⇨ One idea is a multi-level page table (CS book 9.6.3). With 48-bit address, 36 bits are used to access virtual pages. 36 bits are subdivided to 4 levels * 9-bits, and each 9 bit accesses one level of a page table. Each entry in a page table in one level could point to the next-level page table or NULL if not allocated. The page table in the last level (4^{th} level) has the physical page number. This way, you don't need to allocate page tables if they do not have any valid page mappings yet.

Name: Student ID:

(c) The following code scans an array of integers (n elements) and return a pointer to the first occurrence of val. What's wrong with the code? Fix the problem. (Numbers in the left are line numbers, and they are invisible to compilers) (5 points)

```
1 int *search(int *p, int n, int val)
2 {
3    int c = 0;
4   while (p && c < n && *p != val) {
5     p += sizeof(int); c++;
6    }
5    return (c < n) ? p : NULL;
6 }
```

⇨ p += sizeof(int) has to be rewritten as p++; Incrementing the pointer by one actually advances its address by sizeof(*p)

Name: Student ID:

(d) The following code creates n x m integer array, and return the pointer. Actually, it works well on 32-bit OS while it misbehaves on 64-bit OS. What's wrong with the code? Fix the problem. (Numbers in the left are line numbers, which is invisible to compilers) (5 points)

```
1 int **makearray(int n, int m)
2 {
3    int i;
4    int **A = (int **) malloc(n * sizeof(int));
5    for (i = 0; i < n; i++)
6       A[i] = (int *)malloc(m * sizeof(int));
7    return A;
8 }
```

⇨ Line 4 should be rewritten as

```
int **A = (int **) malloc(n * sizeof(int *));
```

⇨ That is, it should allocate the n * pointers instead of n * integers.

Name: Student ID:

5. (20 points) IA-32 Assembly language programming

Consider the following assembly code that is compiled from the C code below

```
…  # don't worry about prolog, etc.
# int loop(int x, int n): x is in %edi, n is in %esi
loop:
   movl %esi, %ecx
   movl $1, $edx
   movl $0, $eax
   jmp  .L2
.L3:
   movl %edi, %ebx
   andl %edx, $ebx
   orl   %ebx, $eax
   sall  %cl,  $edx
.L2:
   compl $0,  $edx
   jne  .L3
   …   # don't worry about epilog, etc.
   ret
```

C code:

```
int loop(int x, int n)
{
   int result = _______A________;
   int mask;
   for (mask = ____B____; mask ____C_____; mask = ___D____)
      result |= ______E_______;
  return result;
}
```

(a) Which registers hold program values x, n, result, and mask? (5 points)

- ⇨ x:edi, ebx
- ⇨ n: esi, ecx
- ⇨ result: eax
- ⇨ mask: edx

(b) Fill out underlined A and B in the C code above. (5 points)

- ⇨ A: 0
- ⇨ B: 1

(c) Fill out underlined C and D in the C code above. (5 points)

- ⇨ C: != 0
- ⇨ D: mask << n

(d) Fill out underlined E in the C code above. (5 points)

- ⇨ E: (x & mask)