Fall Semester 2016

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 140 minutes (9:00 AM – 11:20 AM) to complete your exam. Be wise in managing your time. Good luck.

Question 1	/ 25
Question 2	/ 20
Question 3	/ 10
Question 4	/ 15
Question 5	/ 15
Question 6	/ 15

Total	/ 100

Name: Student ID:

1. (25 points) Simple questions

(a) While the most popular page size is 4KB in practice, modern desktop/server CPUs support larger page sizes such as 2MB and 1GB as well. Such a large page size is called “hugepage”. What is the benefit of a hugepage? (5 points)

(b) Many programs exhibit some sort of locality of reference in memory access such as temporal locality and spatial locality. Briefly explain why the locality of reference matters in performance. (5 points)

(c) When multiple processes run on a machine, they share the CPU by taking turns. When does one process switch to another? Give two examples where a process context switches to another process. (5 points)

Name: Student ID:

(d) Thrashing refers to an undesirable situation that continually swaps between memory and disk. Give a concrete program example that would exhibit the thrashing behavior. (5 points)

(e) In the following code, (1) what kind of exceptions do you see? (2) How can you make the program _not_ crash? (You cannot modify func() nor avoid calling func() in main(), but you can add some code before calling func()). No need to write any real code, but be specific about your approach. (5 points)

```
#include <stdio.h>
void func(void) {
    char *str = “hello World\n”;
    *str = ‘H’;
     printf(“%s”, str);
 }

 int main() {
     func();
     return 0;
 }
```

Name: Student ID:

2. (20 points) Key-value hash table once again

```
enum {BUCKET_COUNT = 1024};
struct Node {
   const char *key;
   int value;
   struct Node *next;
};
struct Table {
   struct Node *array[BUCKET_COUNT];
};

struct Table *TableCreate(void)
{
   struct Table *t;
   t = (struct Table*)calloc(1, sizeof(struct Table));
   return t;
}

int TableInsert(struct Table *t, const char *key, int value)
{
   int h;
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   if (p == NULL) return -1;
   if ((p->key = strdup(key)) == NULL) {
      free(p);
      return -1;
   }
   h = hash(key);
   p->value = value;
   p->next = t->array[h];
   t->array[h] = p;
   return 0;
}
```

Name: Student ID:

(a) Fill in the body of the following function. (10 points)

- int TableSum(struct Table *t) calculates the sum of values of all (key, value) items stored at Table t.
- It returns the sum of values. In case of an error, return -1.
- You may use any C runtime library functions. No need to comment your code.
- You do not need to worry about the overflow of the sum.

```
int TableSum(struct Table *t)

{









}
```

(b) How do you improve the performance of TableSum() above as we insert and delete many (key, value) items over time? Briefly explain your approach (5 points)

c) You want to limit the total number of items stored at the table. When you insert a new item, you first check if the total number of items is below X. If so, you can insert the new item. If the total number is X, you need to remove the least recently used (LRU) item from the table and insert the new item into the table. What would you do to implement this LRU-based item replacement? Briefly explain your approach (explain extra data structures and algorithms for keeping track of the LRU property). (5 points)

Name: Student ID:

3. (10 points) Programming with fork()

(a) How many ‘A’ do you see when you call test1()? (Assume every fork() succeeds.) (5 points)

```
void test1(void)
{
     int i;
     for (i = 0; i < 5; i++) {
        fork();
     }
     printf(“A”);
}
```

(b) How many ‘A’ do you see when you call test2()? (Assume every fork() succeeds.) (5 points)

```
void test2(void)
{
     if (fork() != 0) {
        if (fork() == 0)
            fork();
     } else {
        fork();
     }
     printf(“A”);
}
```

Name: Student ID:

4. (15 points) Redirecting stdin and stdout
   The following code implements the command line,

   $ grep KyoungSoo < score > result

   In order to complete the program, you need to fill out these two functions.

   1. `int RedirectStdin(const char *filename);`

      Redirects stdin to filename. That is, after calling this function, reading from fd 0 would read from the file, “filename”. Returns 0 if successful or return -1.

   2. `int RedirectStdout(const char* filename);`

      Redirects stdout to filename. That is, after calling this function, writing to fd 1 would write to the file, “filename”. Returns 0 if successful or return -1.

   Be meticulous about resource management and error handling. (15 points)

```
int RedirectStdin(const char *filename);
int RedirectStdout(const char* filename);

int main(void)
{
  char *args[] = {“grep”, “KyoungSoo”, NULL};
  if (RedirectStdin(“score”) < 0) {
      fprintf(stderr, “could not read from score\n”);
      exit(-1);
  }

  if (RedirectStdout(“result”) < 0) {
      fprintf(stderr, “could not open result\n”);
      exit(-1);
  }
  execvp(“grep”, args);
  return 0;
}
```

Name: Student ID:

```
int RedirectStdin(const char *filename)
{









}
int RedirectStdout(const char* filename)
{









}
```

Name: Student ID:

5. (15 points) IA-32 Assembly language programming

Translate the following C code into IA-32 assembly language code. (15 points)

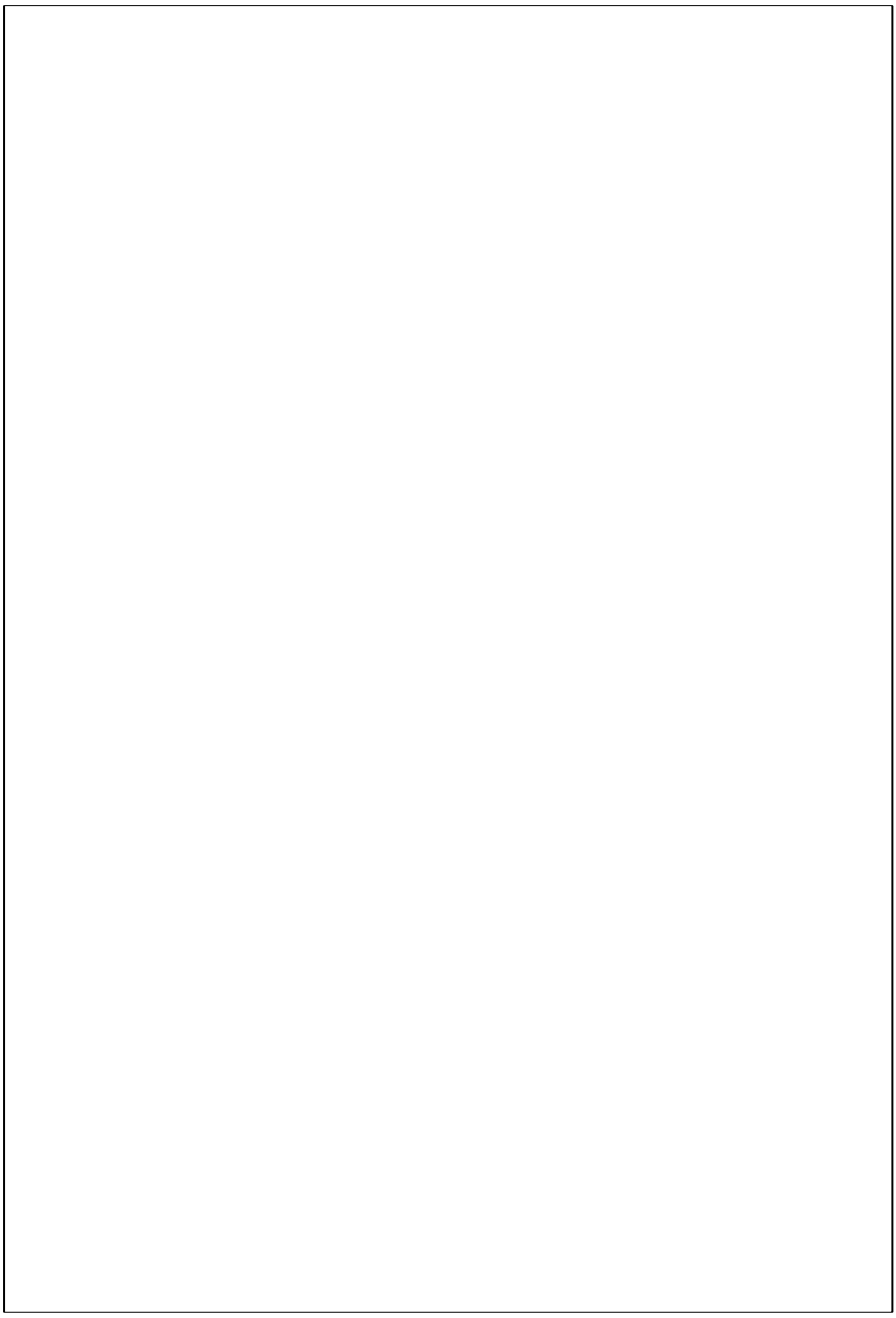
C code:

```
int CountCapitalLetters(char *p)
{
   int count = 0;
   while (*p != 0) {
      if (isupper((int)*p)) count++;
      p++;
   }
   return count;
}
```

Assembly language code: Please fill in the code after “pushl %ebp”.

```
CountCapitalLetters:
  pushl %ebp
```

Name: Student ID:



6. (15 points) Writing simplified ‘tee’

tee is a simple tool that reads from standard input and writes to standard output and files. We write simplified tee that copies whatever is read from standard input to standard output as well as to a “single” file provided as a command line parameter by a user. When the (optional) file name is not provided, the program does not need to write to a file (just need to copy standard input to standard output). Here is an example:

```
$ ls
a
b
c
$ ls | tee x
a
b
c
x
$ cat x
a
b
c
x
```

Here are some guidelines for writing the program.

(1) Don’t worry about including header files.

(2) Whenever there is an error, print out a proper error message to stderr and call exit(-1) to stop the program.

(3) These function prototypes might be useful

- ssize_t read(*int fd, void *buf, size_t count*);

On success, it returns the number of bytes read (zero indicates end of file). It is not an error if this number is smaller than the number of bytes requested. On error, -1 is returned.

- ssize_t write(*int fd, void *buf, size_t count*);

  On success, it returns the number of bytes written. The return value could be smaller than count. On error, it returns -1.

- FILE* fopen(*const char *path, const char *mode*);
- FILE* fclose(*FILE *stream*);
- size_t fwrite(*void *ptr, size_t size, size_t nmemb, FILE *stream*);

  fwrite() writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr. On success, it returns the number of items written. On error, it returns a short item count (or zero).

```
int main(int argc, const char** argv)
{
```

Name: Student ID:

```
  return 0;
}
```