Midterm Exam #2 Apr 6, 2017

COMS W3157 Advanced Programming Prof. Jae Woo Lee
Spring 2017 Columbia University

About this exam:

- There are 3 problems totaling 100 points:

  Problem 1: 40 points
  Problem 2: 20 points
  Problem 3: 40 points

- Assume the following programming environment:

  All programs are built and run on Ubuntu Linux 16.04, 64-bit version, where sizeof(int) is 4 and sizeof(int *) is 8.

  All library function calls and system calls are successful. For example, you can assume malloc() does not return NULL.

  For all program code in this exam, assume that all the necessary #include statements are there even if they are not shown.

  If this exam refers to lab code, assume the versions provided by Jae, i.e., skeleton code and solutions.

  When writing code, avoid using hardcoded numbers as much as possible. Hardcoded numbers make your program error prone, less extensible, and less portable. For example, using "sizeof(int *)" instead of "8" will make it correct for both 32-bit and 64-bit environments.

What to hand in and what to keep:

- At the end of the exam, you will hand in only the answer sheet, which is the last two pages (one sheet printed double-sided) of this exam booklet. You keep the rest of the exam booklet.

- Make sure you write your name & UNI on the answer sheet.

- Please write only your final answers on the answer sheet. Verbosity will only hurt your grade because, if we find multiple answers to a question, we will cherry-pick the part that will result in the LOWEST grade. This policy ensures that a shotgun approach to solving a problem is never rewarded. Please make sure you cross out clearly anything that you don’t consider your final answer.

- Before you hand in your answer sheet, please copy down your answers back onto the exam booklet so that you can verify your grade when the solution is published in the mailing list.

Good luck!

```
+-------------------------------------------------------------------+
|  PLEASE DO NOT OPEN THIS EXAM BOOKLET UNTIL YOU ARE TOLD TO DO SO!  |
+-------------------------------------------------------------------+
```

References

FILE *fopen(const char *filename, const char *mode)

  - Opens a file, and returns a FILE* that you can pass to other
    file-related functions to tell them on which file they should operate.

  - "r" open for reading (file must already exist)
    "w" open for writing (will trash existing file)
    "a" open for appending (writes will always go to the end of file)

    "r+" open for reading & writing (file must already exist)
    "w+" open for reading & writing (will trash existing file)
    "a+" open for reading & appending (writes will go to end of file)

  - returns NULL if file could not be opened for some reason

int fseek(FILE *file, long offset, int whence)

  - Sets the file position for next read or write. The new position,
    measured in bytes, is obtained by adding offset bytes to the position
    specified by whence. If whence is set to SEEK_SET, SEEK_CUR, or
    SEEK_END, the offset is relative to the start of the file, the current
    position indicator, or end-of-file, respectively.

  - returns 0 on success, non-zero on error

size_t fread(void *p, size_t size, size_t n, FILE *file)

  - reads n objects, each size bytes long, from file into the memory
    location pointed to by p.

  - returns the number of objects successfully read, which may be less than
    the requested number n, in which case feof() and ferror() can be used to
    determine status.

size_t fwrite(const void *p, size_t size, size_t n, FILE *file)

  - writes n objects, each size bytes long, from the memory location pointed
    to by p out to file.

  - returns the number of objects successfully written, which will be less
    than n when there is an error.

char *fgets(char *buffer, int size, FILE *file)

  - reads at most size-1 characters into buffer, stopping if newline is read
    (the newline is included in the characters read), and terminating the
    buffer with '\0'

  - returns NULL on EOF or error (you can call ferror() afterwards to find
    out if there was an error).

int fputs(const char *str, FILE *file)

  - writes str to file.
  - returns EOF on error.

Problem [1] (40 points)

Consider the following shell session:

```
$ ./mdb-lookup my-mdb
lookup:
   1: {one} said {111}
   2: {two} said {222}
   3: {three} said {333}
   4: {four} said {444}

lookup: ^C

$ ./mdb-swap.sh my-mdb 4 1

$ ./mdb-lookup my-mdb
lookup:
   1: {four} said {444}
   2: {two} said {222}
   3: {three} said {333}
   4: {one} said {111}

lookup: ^C

$ ./mdb-swap.sh my-mdb 2 3

$ ./mdb-lookup my-mdb
lookup:
   1: {four} said {444}
   2: {three} said {333}
   3: {two} said {222}
   4: {one} said {111}

lookup: ^C

$ cat mdb-swap.sh
#!/bin/sh

# Three command line arguments:
#   $1 is the mdb file
#   $2 is the first record number
#   $3 is the second record number

./mdb-read  $1 $2 > ./mdb-swap-$$
./mdb-read  $1 $3 | ./mdb-write $1 $2
./mdb-write $1 $3 < ./mdb-swap-$$
rm -f ./mdb-swap-$$
```

The mdb-lookup program is used to display the content of the "my-mdb" database file containing four records. The mdb-swap.sh script takes three command line arguments, the mdb file name and two record numbers, and modifies the mdb database file to swap the positions of the two records.

As you can see, the mdb-swap.sh script calls two programs, mdb-read and mdb-write, to do its job. Implement mdb-read and mdb-write on the answer sheet. See the next page for hints and requirements.

Problem [1] continued: hints and requirements

- Note that the mdb-lookup program prints record numbers starting from 1 -- that is, the very first mdb record in the database file has the record number 1, not 0.

- You may find the following function useful: int atoi(const char *str); The man page says, "The atoi() function converts the initial portion of the string pointed to by str to int representation."

- For brevity, please do not write any error checking code in this exam. Assume that the input database file exists, the given record numbers are all valid, and all library function calls will succeed.

- However, mdb-read and mdb-write must work with ANY mdb database file, not just the "my-mdb" file shown in the shell session.

- Write as little code as you can. Unnecessary code will lose points.

- Do NOT declare any new variables other than the ones that are already declared in the skeleton code on the answer sheet.

- Please get the syntax right. At this point, I expect that you can write correct C code without compiler. Incorrect syntax will lose points.

Problem [2] (20 points): Consider the following shell session:

```
$ ls -al
total 8
drwx------  2 jae jae 4096 Apr  5 11:17 .
drwx--x--x 19 jae jae 4096 Mar 20 19:28 ..

$ # Recall that you can use the sed command to match and replace
$ # strings which are read from the standard input. For example:

$ echo "AP SUCKS" | sed s/SU/RO/
AP ROCKS

$ mkfifo mypipe

$ # At this point, I logged in to the same machine from another
$ # terminal window, cd into the same directory, and ran a command.
$ # And then I switched back to this window and ran the following:

$ nc 127.0.0.1 50000
AP SUCKS
AP ROCKS
WHAT?
WHAT?
AP REALL SUCKS!
AP REALL ROCKS!
^C
```

Write on the answer sheet the command that I ran in the other terminal window before I ran "nc 127.0.0.1 50000".

Problem [3] (40 points)

Consider the following program, mdb-msg-cat.c:

```
    struct MdbRec {
        char name[16];
        char  msg[24];
    };

    // Opens a mdb file. Exits the program if the file does not exist.
    FILE *open_mdb(const char *mdb_file_name); // Implementation not shown.

    // Reads a mdb record. Returns 1 if a record is read; returns 0 on EOF.
    int read_mdb(FILE *mdb, struct MdbRec *r); // Implementation not shown.

    void f(FILE *mdb, void *unused)
    {
        struct MdbRec r[1];
        while (read_mdb(mdb, &r[0])) {
            printf("%s,", r[0].msg);
        }
    }

    int main(int argc, char **argv)
    {
        FILE *mdb = open_mdb("my-mdb");

        f(mdb, NULL);

        printf("\n"); fclose(mdb); return 0;
    }
```

Here is the result of running the program:

```
    $ ./mdb-lookup my-mdb
    lookup:
       1: {one} said {111}
       2: {two} said {222}
       3: {three} said {333}
       4: {four} said {444}

    lookup: ^C

    $ ./mdb-msg-cat
    111,222,333,444,
```

Each of the five parts of this problem, (3.1) - (3.5), replaces the function f() of mdb-msg-cat.c. Determine the behavior of the new versions.

[Continued on the next page.]

Problem [3] continued

First, you need to pick one of GOOD, LEAK, or BAD for the new version of the program. Here are what they mean:

- GOOD: The resulting program runs without any valgrind error.
- LEAK: Valgrind reports that the resulting program has memory leak, but no other types of memory error is reported.
- BAD: Valgrind reports at least one memory error that is not a memory leak. In addition, there may or may not be memory leaks.

Second, you need to write down more information about the behavior of the program. What you need to write depends on what you picked:

- If you picked "GOOD" as your answer, you need to write one more thing:

  (A) the output of the program.

- If you picked "LEAK", you need to write two more things:

  (A) the number of bytes leaked;
  (B) the output of the program.

- If you picked "BAD", you need to write one more thing:

  (A) the description of the error, in 10 or fewer words. Your answer must clearly indicate where and what the problem was.

(3.1)
```
void f(FILE *mdb, void *unused)
{
    struct MdbRec *p;
    if (read_mdb(mdb, p))
        printf("%s", p->msg);
}
```

(3.2)
```
void f(FILE *mdb, void *unused)
{
    struct MdbRec r[1];
    if (read_mdb(mdb, r)) {
        f(mdb, NULL);
        printf("%s,", r->msg);
    }
}
```

[Continued on the next page]

Problem [3] continued

(3.3)
```
void f(FILE *mdb, void *unused)
{
    struct MdbRec r[1];
    if (read_mdb(mdb, r)) {
        f(mdb, NULL);
        printf("%s,", r->msg);
    } else {
        if (fork())
            fork();
    }
}
```

(3.4)
```
void g(struct MdbRec **pp)
{
    *pp = malloc(sizeof(struct MdbRec));
}

void f(FILE *mdb, struct MdbRec *r)
{
    g(&r);
    if (read_mdb(mdb, r)) {
        f(mdb, NULL);
        printf("%s,", r->msg);
    }
}
```

(3.5)
```
void f(FILE *mdb, void *unused)
{
    // Assume that the program is linked with libmylist.a.

    struct List list[1];
    initList(list);

    struct MdbRec *p = malloc(sizeof(struct MdbRec));
    while (read_mdb(mdb, p)) {
        addFront(list, p);
    }
    while (!isEmptyList(list)) {
        struct MdbRec *p = popFront(list);
        printf("%s,", p->msg);
    }
    free(p);
}
```

[blank page]

UNI: Name:

---

[1] The following skeleton code is for both mdb-read.c and mdb-write.c:

```
struct MdbRec { char name[16]; char msg[24]; };

int main(int argc, char **argv) {
    if (argc != 3) {
        fprintf(stderr, "usage: %s <mdb> <rec_num>\n", argv[0]);
        exit(1);
    }

    FILE *fp;   struct MdbRec R;   int num, num2;
    FILE *fp2;  struct MdbRec R2;  int i, j, k;

    // (1) You don't have to use every variable declared above.
    // (2) You are NOT allowed to declare any addtional variables.
```

(1.1) Finish the skeleton code above to implement mdb-read program:

(1.2) Finish the skeleton code above to implement mdb-write program:

COMS W3157 Spring 2017
Midterm #2
Answer Sheet, page 2 of 2

Your UNI: +---+---+---+---+---+---+---+---+
| | | | | | | | |
+---+---+---+---+---+---+---+---+

Your Name: _____

left->UNI: _____ right->UNI: _____

[2] Write the command line:

_____

[3]
(3.1)

_____

(3.2)

_____

(3.3)

_____

(3.4)

_____

(3.5)

_____