CS 451/551 Midterm Solution

Date: Mar 23, 2016 **Max Points**: 100

Open book, open notes. No electronic devices

90 Minutes

Please justify all your answers.

Important Reminder As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

1. You are given a file of fixed-size records of size `REC_SIZE`. These records can be compared using the following function:

```
/** Returns < 0, 0, >0 depending on whether the record
 *  pointed to by recP1 is less than, equal, greater than
 *  the record pointed to by recP2.
 */
int recCmp(const void *recP1, const void *recP2);
```

The quick sort function from `stdlib.h` has the following prototype:

```
/** Sort array of nElements elements of size elementSize
 *  in ascending order (ordered by comparison function pointed
 *  to by compar).  The comparison function returns < 0, 0, > 0
 *  depending on whether the element pointed to by its
 *  first argument is less than, equal, greater than the
 *  element pointed to by its second argument.
 */
void qsort(void *base, size_t nElements, size_t elementSize,
           int (*compar)(const void *, const void *));
```

Describe how you would use `qsort()` to sort the specified file of records in **non-ascending** order assuming that:

a) No more than two records may be in memory at a time.

b) The amount of memory used by the program may be proportional to the number of records in the file (but less than the size of the file).

It is sufficient to provide a detailed description, actual code is not required. *20-points*

The basic idea is to use an in-memory array of record indexes as proxies for the records in the file. This array meets the restrictions of (b).

If the file contains `nRecords` records, then the array `indexes` will have `nRecords` entries and initialized such that `indexes[i] = i`.

The routine qsort will be called on this array: `qsort(indexes, nRecords, sizeof(int), indexCompare)` where the comparison function `indexCompare` will work as follows:

```
static int indexCompare(void *ip1, void *ip2) {
  int index1 = *(int *)ip1;
  int index2 = *(int *)ip2;
  seek to record index1 in file and read into rec1;
  seek to record index2 in file and read into rec2;
  int compareRecs = recCmp(&rec1, &rec2);
  return -compareRecs;
}
```

Note the negation in the `return` statement. That is because the problem specification requires the records to be sorted in non-ascending order. Alternately, the arguments to `recCmp()` could be swapped.

After `qsort()` completes, the `indexes[]` array reflects the sorted order of the records. Hence to complete simply:

a) Open a destination file.

b) For `i` in `0 ... nRecords - 1`, read record with index `indexes[i]` into memory from the source file (seeking as necessary) and append it to the destination file.

c) Rename the destination file to the source file.

This method is not particularly efficient when the file is located on magnetic disks as it will do a lot of seeking causing a lot of head movement which will result in very slow disk I/O.

2. Given the following declaration:

```
int *(*f)(void (*)(), int (*)(int *));
```

describe the type of `f` precisely. *15-points*

`f` is a pointer to a function taking 2 arguments:

1. A pointer to a function taking an unspecified number of arguments and returning no results.

2. A pointer to a function taking a single argument which is a pointer to an `int` and returning an `int`.

The return value of the function pointed to by `f` is a pointer to an `int`.

3. User `u1` belongs to primary group `g1` and supplementary group `g2`, and user `u2` belongs to primary group `g2` and supplementary group `g1`. Neither `u1` nor `u2` are super-users, nor do they belong to any supplementary groups other than those explicitly listed above. Given the following `ls -l` listing:

```
-rwxr-xr--    1 u1       g3           14096 Mar  8 22:38 exec1
-rwsr--r-x    1 u2       g2           44096 Mar 10 01:36 exec2
-r--r--rw-    1 u1       g1            4012 Mar 10 01:12 data1
-rw--w-r--    1 u2       g2            8222 Mar  8 17:13 data2
```

fill in the following matrix with a `Y`, `N` or `-` depending on whether the execution specified by the row can (`Y`) or cannot (`N`) make the access specified by the column (R denotes *read*, W denotes *write*), or such access does not make sense (`-`). Please remember to justify your answers. *15-points*

	`data1` R	**`data1` W**	**`data2` R**	**`data2` W**
`u1` runs `exec1`				
`u2` runs `exec1`				
`u1` runs `exec2`				
`u2` runs `exec2`				

When `u1` runs `exec1`, the effective uid is `u1` with GIDs `g1` and `g2`. Hence because its effective UID matches the owner UID of `data1` it can read but not write `data1`. Because its effective GID matches the owner GID of `data2`, it can write but not read `data2`.

When `u2` attempts to run `exec1`, neither the UID or GIDs match the user and group GIDs on `exec1`. Hence the file permissions for `other` apply and since `other` does not have execute permission, `u2` does not have permission to execute `exec1`.

Note that the setuid bit will be used to change effective uids after a process has passed access checks based on its real uid. Hence when `u1` attempts to run `exec2`, it starts up with real uid `u1` and effect gids `g1` and `g2`. It's uid `u1` does not match the uid `u2` of the owner of `exec2`, but one of its gids `g2` matches the group of `exec2`. Hence the access decision is based on the group permissions for `exec2`; since the group permissions deny execute access, `u1` cannot execute `exec2`.

`u2` runs `exec2` with effective UID `u2` and GIDs `g1` and `g2`. Hence it can read but not write `data1` (via its group permissions), and can both read and write `data2` (via its owner permissions).

	`data1` R	**`data1` W**	**`data2` R**	**`data2` W**
`u1` runs `exec1`	Y	N	N	Y
`u2` runs `exec1`	-	-	-	-
`u1` runs `exec2`	-	-	-	-
`u2` runs `exec2`	Y	N	Y	Y

4. List bugs and inadequacies in the following program which purports to print out the average of all the doubles read from the binary files specified by its command-line arguments.

```
int main(char *argv[], int argc) {
  double sum = 0.0;
  int n = 0;
  for (int i = 0; i < argc; i++) {
    FILE *f = fopen(argv[i], "r");
    double d;
07      while (!feof(f)) {
08        if (fread(d, sizeof(double), 1, f) != 1) {
09          perror("fread"); exit(0);
10        }
11        sum += d; n++;
12      } //while
13    } //for
14    printf("%g\n", sum/n);
15  } //main
```

You may assume that all required header files have been included. *15-points* The problems follow (with line numbers in brackets):

1. The arguments to `main()` are reversed [`01`].

2. The iteration over the arguments includes the name of the program. The iteration should start with `i = 1` [`04`].

3. The files are binary, hence `fopen()` should use `rb` as its second argument [`05`].

4. The `fread()` function should have a pointer to a buffer as its first argument, but the code passes a `double`. Hence the first argument should be `&d` [`08`].

5. When an `fread()` error occurs, the program exits with status `0`. Normally `0` the termination status of a successful program run; hence the exit status should be some non-zero value.

6. If the file is empty, then [`14`] will do a *divide-by-0‘ which will most likely abort the program.

7. There is no return value for the program hence its exit status may be garbage on pre-C99 compilers.

8. The `FILE` stream `f` is not closed.

5. Write a function with the following specification:

```
/** Fill in the 10-character string pointed to by perms[] with
 *  the ‘ls -l‘ style permissions string for the file
 *  specified by file descriptor fd.
 *
 *  Specifically, perms[0] should be set to ’d’ if fd
 *  specifies a directory, ’l’ if fd specifies a symlink, ’b’
 *  if fd specifies a block special file, ’c’ if fd specifies
 *  a character special file, ’p’ if fd specifies a FIFO, ’s’
 *  if fd specifies a socket, ’-’ if fd specifies a ordinary
 *  file.
 *
 *  perms[1] ([4], [7]) should be set to ’r’ if the file
 *  specified by fd is readable by its owner (group, other),
 *  ’-’ otherwise; perms[2] ([5], [8]) should be set to ’w’ if
 *  the file specified by fd is writable by its owner (group,
 *  other), ’-’ otherwise; perms[3] ([6]) should be set to ’s’
 *  if the file specified by fd is executable by its owner
 *  (group) and the setuid-bit (setgid-bit) is set, as ’x’ if
```

```
         *  the file specified by fd is executable by its owner
         *  (group) but the setuid (setgid) bit is not set, as '-'
         *  otherwise; perms[9] should be set to 'x' if the file
         *  specified by fd is executable by other, '-' otherwise.
         *  The function should NUL-terminate the perms[] string.
         *
         *  The function should return 0 if there is no error;
         *  otherwise it should return the ‘errno‘ of the first error
         *  encountered.
         */
        int getFilePermissions(int fd, char perms[11]);
```

You need not show the inclusion of required header files. *20-points*

```
int
getFilePermissions(int fd, char perms[11])
{
  struct stat statBuf;
  if (fstat(fd, &statBuf) != 0) return errno;
  mode_t mode = statBuf.st_mode;
  perms[0] =   S_ISDIR(mode) ? 'd'
             : S_ISCHR(mode) ? 'c'
             : S_ISBLK(mode) ? 'b'
             : S_ISFIFO(mode) ? 'p'
             : S_ISLNK(mode) ? 'l'
             : S_ISSOCK(mode) ? 's' : '-';
  static int masks[] = {
    S_IRUSR, S_IWUSR, S_IXUSR,
    S_IRGRP, S_IWGRP, S_IXGRP,
    S_IROTH, S_IWOTH, S_IXOTH,
  };
  for (int i = 0; i < sizeof(masks)/sizeof(masks[0]); i++) {
    perms[i + 1] = (mode & masks[i]) ? "rwx"[i%3] : '-';
  }
  if (perms[3] == 'x' && (mode & S_ISUID)) perms[3] = 's';
  if (perms[6] == 'x' && (mode & S_ISGID)) perms[6] = 's';
  perms[10] = '\0';
  return 0;
}
```

The above array and loop abstracts out the commonality for the different types of users. For solutions done during the exam it is sufficient that the handling of a single type of user be specified.

6. Discuss the validity of the following statements: *15-points*

   a) If a setuid program is executed, then a necessary condition for the executing program to read a file is that the owner of the program have read access to the file.

   b) The output of `printf()` will always be line-buffered.

   c) A expression like `scanf("%d", i)` is always incorrect.

d) If you do a `fprintf()` to a `FILE` stream, followed subsequently by a `write()` to the file descriptor underlying the same stream, then it is possible that the output of the `fprintf()` appears after the output corresponding to the `write()`.

e) If a successful `close(0)` is immediately followed by a successful call to `open()`, then the return value of `open()` is guaranteed to be `0`.

The answers follow:

a) It is possible that the program obtains read access to the file using group or other permissions. Hence the condition is not **necessary** and the statement is false.

b) The output of `printf()` will be line-buffered by default, since `stdout` is line-buffered by default. However, it is possible to change this default buffering of `stdout` using `setbuf()` or `setvbuf()`. Hence, in general, the statement is false.

c) The statement

   A expression like `scanf("%d", i)` is almost always incorrect.

   would be more accurate. The `scanf()` could be correct if `i` has been given a value representing a `int *` expression or #`define`'d to be a macro which expands to an expression giving an address. Another situation when an expression like the above may be correct is that the `scanf()` does not refer to the `scanf()` defined in `<stdio.h>`, but to some other library function (where passsing a `int` as the 2nd argument (instead of a `int *`) is correct. Such a program is pretty obfuscated, but still possible.

d) This statement is true. The `fprintf()` could be going to a stdio buffer which may not be getting flushed out until after the `write()` statement.

e) This statement is almost always true. If there are no errors, the file descriptor returned by `open()` is always guaranteed to be the lowest numbered unused descriptor which in this case should be `0`. One situation where this may not hold is if the program is multi-threaded and another thread does an explicit or implicit `open()` between the call to `close(0)` and the call to `open()` (in such situations, the atomicity guaranteed by functions like `dup2()` can be useful).