CS 3423 Test 2

1. **[30]** Write a C **function** with prototype:

   ```
   int delrecs(char *fdb, int first, int num, int rsz);
   ```

   This function will delete the block of `num` records starting at record `first` from file whose name is given by `fdb`. Notice that the records at the end of the file must be moved up to fill the positions of the deleted records and the file shortened. You may assume that all records exist in the files.

   A record is merely a consecutive block of bytes of size **rsz**. The record indexing starts at 0 so record 0 consists of bytes 0 through $rsz - 1$, record 1 consists of bytes rsz through $2rsz - 1$, and so on. The function must both open and close the files and reads/writes **must (NOTE MUST)** be record sized. If successful, **delrecs()** will return a 0 else **delrecs()** will return -1. The function must use low-level I/O except for printing error messages and you may assume that $rsz \leq 4096$.

2. **[25]** There are constant attacks on your machine consisting of thousands of ssh login attempts. This has no chance of working since you don't use word-based passwords on your system but they irritate you and fill up your log files making it next to impossible to read your logs. You have written a program, `sshreject whitelist` which will read from stdin and if it finds 5 failed login attempts without a success from the same IP (not in the file whitelist) will reject the route. Thus your machine becomes a "black hole" to that IP. However your program has to read the log file `/var/log/auth.log` which is periodically rotated. The `tail` program can be told to ignore the rotation and continue reading from the new `auth.log` file. Thus this could be done on the command line with the command:

   ```
   tail -F /var/log/auth.log | sshreject myips
   ```

   Write a single C program that will implement the above.

3. **[25]** Write a C `program` which will fork 33 children assigning each process a unique index from 0 to 32. After creation the original process (process 0) will write his **pid** (not index) to each child process. Each child process (processes 1, ..., 32) will read the message from process 0 and those with an **odd pid** will write a message back to process 0 that consists of the received pid and the child's pid. Process 0 will read all the messages written to him and will print on stdout a string similar to `"process 23456 received message from 98765\n"`.

4. **[25]** Write a `function`, called `xmsg()`, which will read **all** messages from the pipe whose **read file descriptor** is passed to the function. Each message will contain the name of an executable and a single argument. The function will cause the execution of the executable with the single argument and return -1 on error or 0 on reaching an end-of-file on the pipe. The function cannot wait for the executable nor do we want to create zombies. The function `xmsg()` has the following prototype:
   `int xmsg(int fd);`
   and the message structure is:

   ```
   struct command
     {
       char cmd[64];      /* Null terminated name of executable */
       char arg[64];      /* Single Argument to command         */
     };
   ```