# CS 3423 Final

1. **[20]** Suppose a Perl program, `attgen`, is invoked with the command:

```
attgen -t reactive file1 file2 file3
```

and suppose `file1` contains the lines "`dir1`", "`dir2`" and "`dir3`", while `file 2` contains "`dir4`" and "`dir5`", while `file3` contains "`dir6`".

At the start of the program:

(a) What is the list value of `@ARGV`?

(b) What is the value of `$#ARGV`?

(c) In the following what value is assigned to `$bat`:

```
$bat = @ARGV + 1;
```

(d) **After** the following statements have been executed,

```
$a = shift; -t
$b = shift; reactive
$c = shift; file1
$d = pop;  file2
unshift(@ARGV,$d); file2
$e = <>; dir6 dir6
```

i. What is the value of `$d`?

ii. What is the value of `$e`?

iii. What is the value of `$ARGV`?

iv. What is the list value of `@ARGV`?

(e) On the other hand if the following statements were to be executed first:

```
$a = shift; -t
$b = shift; reactive
$c = shift; file1
@d = <>; dir4 dir5 dir6
$e = shift @d;
```

i. What is the value of `$c`? file1

ii. What is the value of `@d`? dir4 dir5 dir6

iii. What is the value of `$e`? dir5

2. **[20]** Write a Perl script, `pgrep`, which will search all **TEXT** files which are **under** any of the directories whose names are in the files whose names are on the command line. The search is to search the contents of these files for the regular expression which is given as the first argument on the command line. For each match in each file, the pathname of the file is to be printed followed by a colon and then the matching line. A **sample** invocation:

```
pgrep 'ca*t+' file1 file2 file3
```

3. **[20]** Write a Perl script, called `inj`, which will read two files, inserting the second file after the given line number of the first file and then write the result to the third file. You may assume that the first file has sufficient lines. A sample invocation which will insert `file2` after line 50 in `file1` writing the result to `fileout`:

```
inj 50 file1 file2 fileout
```

4. **[20]** Write a perl script, `chext`, that will find all files under the given directory with a given extension and will change the extension of each of these file names to have the second given extension (using the unix mv command). Thus if you want to change all of the `*.c` files under `/src/myprog` to `*.cc` files you would use:

```
chext /src/myprog c cc
```

This would change the file `/src/myprog/src/file.c` to `/src/myprog/src/file.cc`.

5. **[25]** Write a C **function** with prototype:

```
int cprec(int num1, char *filedb1, int num2, char *filedb2, int rsz);
```

The function `cprec()` will copy the record number `num1` from the file `filedb1`, **inserting** the record into the file `filedb2` at record position `num2`. Pay careful note to the fact that you are to insert the record and **not** overwrite a record in `filedb2`. `filedb1` should remain unchanged.

A file can be considered to be a sequence of records of a fixed size, say `rsz`, where the first `rsz` bytes are to be considered record 0, the next `rsz` bytes are record 1, .... The records must be read/written one at a time but you may assume that they are less than 4096 in size. The function must also both open and close the files. If successful, `cprec` will return a 0 else `cprec` will return -1. The function must use low-level I/O.

6. **[25]** You are writing a grading program which prints the username of each user followed by his grades on the same line. However you want to also put the real name of the user on the same line. Since your system is using `ldap` you don't have a normal passwd file but you can use the `getent` function to get the equivalent of the contents of the passwd file. If you were on the command line then you could get the passwd line of the user `maynard` by executing

```
getent passwd | egrep maynard
```

Write a C **function** which, when passed the name of a user, will return a file descriptor to the read end of a pipe from which the program can read the user passwd line from `getent`. The pipe should contain only the single user line. A prototype

```
int getname(char *uname);
```

7. [25] Write a C program which will fork **num** (where num is a command-line argument) children. Every child with an odd pid (but not the original parent) will write his pid to all the other children (but not parent). Every child will then read every message written to him, one at a time, and then write each received pid along with his pid to the original parent using the following message structure.

```
struct message {
  pid_t my_pid;
  pid_t received_pid;
}
```

On receipt of each message, the parent process will print the received data to stdout in the following form:

```
Process 1234 received a message from Process 4567
```

You do not need to worry about pipes filling.

8. [25] Write a **function**, called `execpipe`, that will be passed two structures of the form:

```
struct command
  {
    char *cmd;       /* name of command */
    char *argv[10]; /* Null terminated argv list of arguments to command */
  };                 /* (including the command cmd) */
```

The function `execpipe()` with prototype:
`int execpipe(struct command cmd1, struct command cmd2);`
will fork and exec both commands, piping the output of the first command into the second command. The function **returns** 0 on success and -1 on error. Notice that this is a **function** and the original process expects a return!

9. [25] In the system that you are writing, many programs (called clients) will be executing and will need other programs, called servers, to process some data. In this version of the system each program will write to a well known fifo (named pipe) with the name "`/tmp/dispatch`". This fifo already exists in the file system. The dispatcher programs read messages from the fifo, exec the appropriate server after connecting the pipes. The server is expecting its data via stdin and will write its data to stdout. The dispatcher must be certain that the server's stdout is redirected to the return fifo named in the message and that the server's stdin is redirected to the data fifo named in the message. It is the clients responsibility to write the data to the data fifo. The format of the message is:

| **Data** | **Field (first byte - last byte)** |
|---|---|
| program name (null terminated) | 0-127 |
| return fifo name (null terminated) | 128-255 |
| data fifo name (null terminated) | 256-383 |

Write the dispatcher program (in C) for this system assuming that there may be several dispatcher programs executing at the same time. There should be no long-lived zombies created nor should the dispatchers wait for a potentially long time. The dispatchers should follow the current path (bad idea) to find the program when exec'ing.