# CMPT 300: Operating Systems
# School of Computing Science
# Fall 2015, Section D1

# Assignment #4: A Distributed Flock of Lyrebirds
# Due Date: Thursday, December 3, 2015

## Assignments in this Course:

All four assignments in CMPT 300 this term will be related to each other. If you were unable to complete assignment #3 and would like a solution to build upon for this assignment, you may purchase an assignment for a penalty of -20% (i.e., 20/100) off your mark for this assignment. You cannot arbitrarily adopt another student's (or anyone else's) code; that is considered plagiarism. This purchased assignment is not guaranteed to be bug free, but we will provide a solution of reasonable quality. Please contact your instructor if you would like to do this. Of course, read this assignment description first.

**Cleaning Up Your Processes**

When using `fork()` (and related functions) for the first time, it is easy have bugs that leave processes on the system, even when you logout of the workstation. It is **your** responsibility to clean up (i.e., kill) extraneous processes from your workstation before you logout. Learn how to use the **ps** and **kill** (and related) commands.

Marks will be deducted if you leave processes on a workstation after you logout.

## Overview:

In this assignment, you will be extending and improving the `lyrebird` program from Assignment #3. The format of the input is changing in order to make `lyrebird` more flexible and to allow for some new behavior. You will also have to make additions to the internal structure of `lyrebird`.

The `lyrebird` program from assignment #3 had the ability to use multiple cores through the use of child processes, and had the ability to communicate between them by passing messages. However, it still had the limitation that it could only make use of hardware resources on a single machine. As the number of encrypted tweets increases, we want the ability to take advantage of multiple machines to do the decryption. For this task, we will develop a distributed system that uses a client-server architecture to (a) centralize the decryption of tweets over a number of workstations, (b) to centralize the output of all activities of all processes, and (c) to coordinate their clean exit.

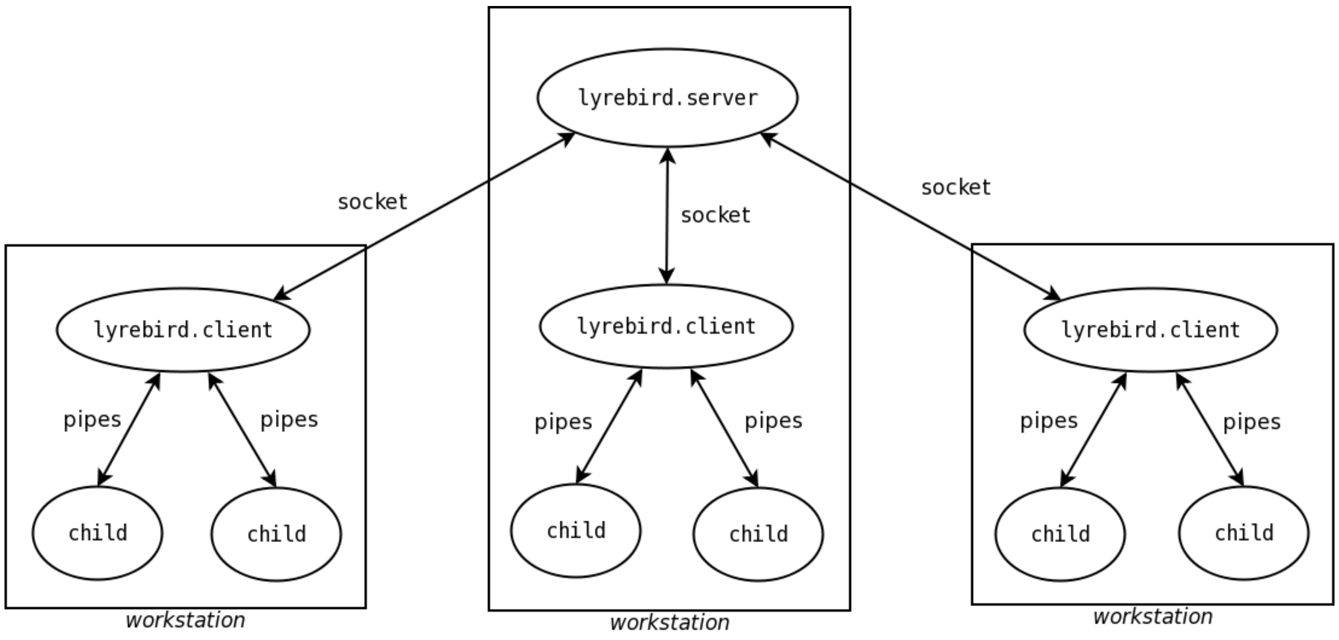
In this assignment, you will be creating *two* different programs:

1. A client `lyrebird` (which is a modified form of the `lyrebird` program from the previous assignment). You can safely assume that each workstation will have *at most one* `lyrebird` client running at any time. A `lyrebird` client will use `fork()` to create child processes to

perform decryption tasks across multiple cores on the workstation. The name of executable for the client must be `lyrebird.client`.

2. A server `lyrebird` (which will contain portions of the `lyrebird` program from the previous assignment). You can safely assume that *only one* `lyrebird` server will be running at any time. The `lyrebird` server centralizes the execution of all decryption tasks and logs the success or failure of each task. The server also is responsible for the clean exit of its client processes. The name of the executable for the server must be `lyrebird.server`.

The `lyrebird.client` processes are controlled by the `lyrebird.server`. Only the client communicates (using Internet sockets) with the server; only the client communicates (using pipes) with the child processes that are created to do the decryption. The child processes should *never* communicate directly with the server. This means we have a hierarchy of processes (see example in the figure below) whereby (a) the server is at the top, running on one machine, (b) one client process runs per workstation, and (c) many children per client process may be running on a workstation. (a) can only communicate with (b) using sockets, (b) can communicate with both (a) and (c) using sockets for (a) and pipes for (c), and (c) can only communicate with (b) using pipes.



Only the `lyrebird` server reads the configuration file and that config file has a similar format as in Assignment #2 (note that this means the configuration file will not specify a scheduling algorithm. More on that below). Client `lyrebird` processes are given the filename information by communicating with the server process using an Internet socket. The `lyrebird` client is responsible for creating child processes on the workstation on which it does decryption.

The server process also keeps track of the success and failure of each decryption task. To do this, the child processes should send messages through the client to the server describing the success or failure of the decryption tasks. It is the server that outputs those messages to a log file. Use of `stdout` and `stderr` in all processes should be limited to startup & exit messages (described below), and in extreme cases where exceptional situations prohibit the program from continuing its normal execution.

**Standard Comment About Design Decisions**

Although many details about this assignment are given in this description, there are many other design decisions that are left for you to make. In those cases, you should make reasonable design decisions (e.g., that do not contradict what we have said and do not significantly change the purpose of the assignment), document them in your source code, and discuss them in your report or README file. Of course, you may ask questions about this assignment (for example, on the mailing list) and we may choose to provide more information or provide some clarification. However, the basic requirements of this assignment will not change.

## Input/Output and Behavior Specification (Lyrebird Server):

The server process is started before any clients are started. The server program `lyrebird.server` takes exactly two command-line arguments: the path to a configuration file, and the path to a log file. An example of how the lyrebird server is started from the command line is:

```
$ ./lyrebird.server config_file.txt log_file.txt
```

When the lyrebird.server starts up, it must output its process ID, the IP address of the workstation it is running on, and the port number used by the server to accept new connections (see discussion on Internet sockets below). That information should be outputted in the following format:

```
[Mon Nov  2 23:04:31 2015] lyrebird.server: PID 345 on host
199.60.156.47, port 2309
```

The server is responsible for three main tasks:

1. The server should accept Internet socket connections to client processes regularly throughout its execution. The lyrebird client processes can connect to the server at any time and the number of clients connected to the server can vary across time. The server should record, in its log file, when a client connects to and disconnects from the server process.

2. The server should read the configuration file and assign decryption tasks to clients in a first come first served order. Note that we will not use round robin scheduling in this assignment because clients can show up and exit (as a result of error) at any time. The assignment of a decryption task to a client should be recorded in the log file.

3. The server should read messages from the client regularly throughout its execution. Any messages regarding the success or failure of the decryption tasks should be recorded in the log file.

The configuration file has the same format as in assignment #2 (meaning that the specification of scheduling algorithm is *not* part of the configuration file). The contents of the configuration file are pairs of file locations (i.e. file names with, optionally, its full path specified), with two file locations per line. The first file location in each line should correspond to a file containing encrypted tweets. The second file location should correspond to the file where the decrypted tweets will be saved. For example, the contents of `config_file.txt` may look like:

```
./encrypted_tweets.txt ./decrypted_tweets.txt
/home/userid/more_tweets.txt /home/userid/output.txt
```

where `./encrypted_tweets.txt` and `/home/userid/more_tweets.txt` are files containing the encrypted tweets, while `./decrypted_tweets.txt` and `/home/userid/output.txt` are the files where the corresponding decrypted tweets will be saved. As with previous assignments, you can safely assume that each string in the configuration file (i.e. every file location) is a maximum of 1024 characters long. There is no limit on the number of lines (i.e. encrypted files) in the configuration file. Note that the files containing the encrypted tweets will have the same format as in the previous assignments.

The log file is used to keep track of the execution of all the decryption tasks. The executions of the server and the clients are logged using messages similar to those used in assignments 2 and 3. An example of a log file is:

```
[Mon Nov  2 23:04:36 2015] Successfully connected to lyrebird client
199.60.147.56.
[Mon Nov  2 23:04:37 2015] The lyrebird client 199.60.147.56 has
been given the task of decrypting
./a1-tst/testdata/t10-et-File-with-10000-tweets.txt.
[Mon Nov  2 23:04:37 2015] The lyrebird client 199.60.147.56 has
been given the task of decrypting
./a1-tst/testdata/t6-et-Single-Tweet-20-chars.txt.
[Mon Nov  2 23:04:37 2015] The lyrebird client 199.60.147.56 has
been given the task of decrypting
./a1-tst/testdata/t2-et-Assignment-Example.txt.
[Mon Nov  2 23:04:37 2015] The lyrebird client 199.60.147.56 has
successfully decrypted
./a1-tst/testdata/t10-et-File-with-10000-tweets.txt in process 1102.
[Mon Nov  2 23:04:37 2015] The lyrebird client 199.60.147.56 has
been given the task of decrypting ./bar.
[Mon Nov  2 23:04:37 2015] The lyrebird client 199.60.147.56 has
successfully decrypted
./a1-tst/testdata/t6-et-Single-Tweet-20-chars.txt in process 1103.
[Mon Nov  2 23:04:37 2015] The lyrebird client 199.60.147.56 has
successfully decrypted
./a1-tst/testdata/t2-et-Assignment-Example.txt in process 1104.
[Mon Nov  2 23:04:40 2015] The lyrebird client 199.60.147.56 has
encountered an error: Unable to open file ./bar in process 1102.
[Mon Nov  2 23:04:40 2015] lyrebird client 199.60.147.56 has exited
successfully.
```

Note that the log file should contain five different types of files:

1. Connection Messages: When a client connects to the server, the server should record the IP address of the client in the log file. The message should also contain a timestamp. An example of a connection message is:

```
[Mon Nov  2 23:04:36 2015] Successfully connected to lyrebird
client 199.60.147.56.
```

2. Dispatch Messages: When the server sends a decryption task to a client, the server should record the task assignment in the log file. Decryption tasks are sent out in first come first served order. The dispatch message should contain a timestamp, the IP address of the client receiving the task, and the name of the file to be decrypted. An example of a dispatch message is:

```
[Mon Nov  2 23:04:37 2015] The lyrebird client 199.60.147.56
has been given the task of decrypting
./a1-tst/testdata/t2-et-Assignment-Example.txt.
```

3. Success Messages: When the client has successfully decrypted an encrypted file, the server should record that event in the log file. The success message should contain a timestamp, the IP address of the client that decrypted the file, the process ID of the child process that did the decryption, and the name of the decrypted file. An example of a success message is:

```
[Mon Nov  2 23:04:37 2015] The lyrebird client 199.60.147.56
has successfully decrypted
./a1-tst/testdata/t6-et-Single-Tweet-20-chars.txt in process
1103.
```

4. Failure Messages: When the client, or one of its children, runs into an error and is unable to complete a decryption task, that failure should be reported in the log file. The failure message should contain a timestamp, the IP address of the client, the process ID of the process that failed, and the name of the file that could not be decrypted successfully. An example of a failure message is:

```
[Mon Nov  2 23:04:40 2015] The lyrebird client 199.60.147.56
has encountered an error: Unable to open file ./bar in process
1102.
```

5. Disconnect Messages: When a client disconnects from the server, the server should record that action in the log file. The disconnection message should contain a timestamp, the IP address of the client, and whether the disconnection was expected or unexpected. An example of an expected disconnection message is:

```
[Mon Nov  2 23:04:40 2015] lyrebird client 199.60.147.56 has
disconnected expectedly.
```

Once all the decryption tasks have been successfully completed, the server should send a message to each client telling them to exit. Once the server has told each client to exit, the server should wait to hear from each client that they have successfully exited and record any remaining messages from the clients to the log file. Finally, the server should print the following to the terminal and exit:

```
[Mon Nov  2 23:04:41 2015] lyrebird server: PID 345 completed its
tasks and is exiting successfully.
```

## Input/Output and Behavior Specification (Lyrebird Client):

After the server process has been started, client processes can be started on a number of workstations. The client process takes exactly two arguments which tell the client where the server is located: The IP address and the port number of the server. For example, if the server is started as shown in the example above, then each client would be started from the command line using:

```
$ ./lyrebird.client 199.60.156.47 2309
```

When the client process starts up, it must make a socket connection with the server using the information provided on the command line. This socket connection will remain open for the entire lifetime of the client process and it is from this connection that the client finds out which files to decrypt. Note that the clients *never* read the configuration file or write to the log file directly. When a `lyrebird.client` has successfully connected to the server, it should print out the following to the terminal:

```
[Sat Nov  7 21:33:02 2015] lyrebird client: PID 6940 connected to
server 199.60.147.56 on port 35341.
```

Once a `lyrebird.client` has started running, it should create its child processes as well as *read and write* pipes to communicate with each child process. As in assignment #3, there should be `N-1` child processes created, where `N` is the number of CPU cores on the client machine.

The client then communicates with the server across the socket connection to get decryption tasks. Those tasks are forwarded to the client's child processes who perform the decryption. The client should also read messages from its child processes and communicate the success and failure of decryption tasks to the server. The format of the messages passed over the socket and pipe connections is entirely up to you.

At some point, the client will receive a message from the server to stop executing. When this happens, the client should tell its children to exit cleanly and continue to forward messages from the children to the server until the children have exited. Once all the child processes have exited successfully, the client should send a message to the server saying that the client is exiting successfully. Finally, the client should print the following to the terminal and exit:

```
[Sat Nov  7 21:33:06 2015] lyrebird client: PID 6940 completed its
tasks and is exiting successfully.
```

## Error Handling

Note that a successful run of the client and server processes should only result in the output specified above. You may, of course, insert `printf()` output for your own debugging purposes, but those debugging statements should not appear when running the submitted version of your assignment.

The one exceptional situation where you are allowed to print something to the terminal that is not mentioned above is when an error occurs that forces the exit of one of your `lyrebird` processes. In that situation, it is acceptable to print out an error message. That error message should include a timestamp, the process ID of the process that encountered the error, and an informative description of what the error is.

In a situation where an error occurs in the client or server processes, the process should first try to recover from the error and if it cannot, the process should exit cleanly. In the server process, a clean exit involves making sure that all the client processes exit cleanly first and that any messages from the clients are correctly received and handled (or at least as well as can be done under the circumstances). In the client process, a clean exit is equal to what was described in assignment #3: the client process should make sure that its children have exited and should continue to forward any messages from the children to the server.

## Required Design:

As previously discussed, you must write two different programs: the server and the client. There is only one server process and it communicates with (potentially) many clients using Internet sockets. Specifically, the server should use a different socket for each client. Those sockets should be `AF_INET` domain stream sockets. The server and clients communicate via Internet sockets, which is why they can be executing on different Linux workstations. Note that the clients are not child processes of the server. In fact, the client and the server processes are completely unrelated processes and would not even have to be run by the same user (although for the purpose of this assignment, they will be run by the same user as you have no real means at your disposal for executing programs under two different user IDs). To setup a socket connection, you will need the following socket-related functions: `socket()`, `bind()`, `listen()`, `connect()`, `accept()`. If your client and server programs do not use sockets, you will receive a mark of zero for correctness.

When printing out the IP addresses, either to the terminal or the log file, note that the following are *not* acceptable IP address outputs: 0.0.0.0, 127.0.0.1 (these IP addresses have special meaning and a client may not always use these IP addresses to connect to the server).

Your client program *must* use `fork()` to create child processes. If your program does not use `fork()`, you will receive a mark of zero for correctness. You may also require functions like `getpid()`, `waitpid()`, and `sysconf()`.

Also, your client program must pass information between the client and its child processes using pipes. Two pipes should be created for each child process: one to pass messages from parent to child, and one to pass messages from child to parent. To take care of this task, you'll likely need to use the `pipe()`, `read()`, `write()` and `close()` functions. If your program does not use `pipe()`, you will receive a mark of zero for correctness.

For the scheduling in both the server and the client, you'll need the ability to monitor multiple pipes or sockets to see if any information has arrived. For this, you'll need the `select()` function.

As appropriate, you must use C memory allocation (e.g., `malloc(), free()`) and C file I/O functions (e.g., `fopen(), fscanf(), fclose()`). Because of the use of MEMWATCH (see below), you cannot use C++ streams, the Standard Template Library (STL), or the C++ standard library extensions (e.g., cannot use type/class `string`). Also, your TA may not have any expertise in C++ and therefore we cannot guarantee support for languages other than C.

You must write a Makefile for your program. When someone types `make`, your Makefile should build both executables `lyrebird.server` and `lyrebird.client`. When someone types `make clean`, your Makefile should remove the executables `lyrebird.server` and `lyrebird.client` (if any), all `.o` files (if any), `memwatch.log`, and all `core` files (if any).

It is IMPERATIVE that your program properly deallocates ALL dynamic memory in a correct fashion (i.e., using `free()`) before your program terminates, or else your assignment will LOSE marks. To check that your program properly allocates and deallocates ALL dynamic memory it uses, you must use the MEMWATCH package, as described in assignment #1. If your assignment is not properly compiled with MEMWATCH enabled, or if MEMWATCH reports that your memory allocation/deallocation was incorrect, then you will lose marks.

When developing and testing your program, make sure you clean up all of your `lyrebird` processes before you logout of a workstation. Marks will be deducted for `lyrebird` processes left on workstations.

## What to Hand In:

You will submit your assignment through http://courses.cs.sfu.ca. Your submission should be a `zip` archive file with the name `submit.zip`. The zip file should contain the following:

1. A **README** file (ASCII text is fine) for your assignment with: (1) your name, (2) student number, (3) SFU user name, (4) lecture section, (5) instructor's name, and (6) TA's name clearly labeled. All these items of information should also be part of **each file** that you submit (e.g., as a comment in your code files). The **README** file must also include a short description of your program, as well as a description of the relevant commands to build (e.g. `make all`) and how to execute your programs including command line parameters. As per the academic honesty guidelines, you should list your sources and the people you have consulted within this **README** file.

2. A report in HTML file format, in a file called **report.html**, describing the design, implementation, and testing of your assignment. The report should contain *no more than 750 words*. You do not need to repeat any information contained in this assignment description. I recommend you spend 25% of your report on an overview of your assignment, 50% on your design and implementation, and 25% on how you tested your program, and some concluding remarks. Note the emphasis on testing your program.

3. Your source code file(s) for `lyrebird.server` and `lyrebird.client`, including all

header files. Do NOT submit any MEMWATCH files, as the TA will use his own fresh copy of that code, but the use of MEMWATCH should be enabled in your code and `Makefile`.

4. Your `Makefile`.

**NOTE:** Do **not** submit files or test data **not** described above. Only submit what is requested and what is required to compile your program (except, of course, the MEMWATCH files). When you submit your assignment, please make sure that we can unzip, make, and run your assignment without having to switch directories.

## Marking:

This assignment is worth 10% of your final mark in this course. **This is an individual assignment. Do not work in groups.** Review the course outline on this matter.

The assignment itself will be marked as follows: 20% for your report (clarity, technical accuracy, completeness, thoroughness of the testing, etc.), 50% for the correctness of the program when we test it using the CSIL Linux machines, using `gcc`, and 30% for the quality of the implementation (design, modularity, good software engineering, coding style, useful and appropriate comments, etc.).

Note that the correctness mark will be computed solely on how your program runs and not on what the code looks like (with the exception of the use of `fork(), socket(),` and `pipe()`). If your source code, **as submitted**, does not compile and run (using the submitted Makefile) on the CSIL Linux workstations using `gcc`, you will receive a mark of zero for correctness. Review the Course Outline on this matter.

When it comes to your quality of implementation mark, all that you have learned about good programming style and comments in your code will apply. Having correct code is important, but good style, design, and documentation are also important. We cannot provide an exhaustive list of what we will look for, but an incomplete list includes: a comment for each source code file, a comment for each procedure/function, a comment for each significant (global or local) variable, good choice of names/identifiers, proper modularity (e.g., do NOT put all/most of the code in `main()`), checking function return values for errors, etc.

**NOTE:** There are a number of programs that you can download off the Internet that provide similar functionality to what you are asked to implement for this assignment. We are familiar with them. Therefore, do **not** download these programs; write your own solution to this problem. Modifying someone else's program (including programs that you can download) is against the requirements of this assignment and is an Academic Offense. If you have any doubts about whether your actions are permissible or not, you should ask the instructor **before** proceeding.

## Hints:

- Keep in mind that pipes are uni-directional but that sockets are bi-directional. As a result, you need two pipes for the client-child communication, but only one socket for server-client communication. Also remember that once a socket/pipe is no longer needed, the best practice is to close that connection.

- When it comes to testing your Internet socket communication, I recommend running the client and server on the same machine first before running the two programs on separate machines. That way, you are less likely to bug anyone. Once you start testing across multiple machines:
  - Keep in mind that the file names listed in the config file have to be accessible *by both the client and the server*, so you may want to restrict this type of testing to the CSIL lab machines where all your files are on a server and visible to all CSIL machines.
  - You may want to use `ssh` to test across machines in the CSIL lab. If you do, please read this first: http://www.sfu.ca/computing/about/support/csil/how-to-remote-access-to-csil.html

- There is a lot of programming overhead in working with sockets, so you may want to learn about the following C/Unix library constructs: `ntohs(), htons(), inet_pton(), getnameinfo(), getsockname(), getifaddrs(), struct sockaddr_in.`

- Here are a couple of good tutorials on network programming using Internet sockets:
  - Jim Frost's tutorial: http://webdocs.cs.ualberta.ca/~paullu/C379/jim.frost.sockets.html
  - Beej's Guide to sockets: http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html

- Before you submit, make sure your `submit.zip` file works from within a fresh directory. That way, you make sure that `submit.zip` contains all the needed files for testing (with the exception of MEMWATCH, of course).

- Remember, make sure that your program does **not** produce any debugging or extraneous output during **normal** execution. Only the requested output should be generated. Marks will be deducted for incorrect and other unrequested output. That said, it is acceptable to have output to report an actual error.

- Remember to list whatever sources you use in your README file.

Further hints and clarifications may be given later on the course mailing list, if warranted. Be sure to read the emails on the mailing list on a regular basis.