OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
DUBLIN
TECHNOLOGICAL UNIVERSITY DUBLIN

Systems Software Report CA1

DT228
BSc in Computer Science

Aron O'Neill
C16466214

School of Computer Science
TU Dublin – City Campus

13/03/20

Table of Contents

Table of Figures

Functionality Checklist

Feature	***Description***	***Implemented***
F1	System Architecture including makefile	Yes
F2	Daemon (Setup/Initialisation/Management)	Yes
F3	Daemon (Implementation)	Yes
F4	Backup Functionality	Yes
F5	Transfer Functionality	Yes
F6	Lockdown folder for Backup / Transfer	Yes
F7	Reporting (IPC)	Yes
F8	Logging and Error Logging	Yes

Have you included a video demo as part of the assignment: Yes or No
Link to Video: please paste link here

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

[illegible]

Aron O'Neill

13/03/20

Feature 1 - System Architecture including makefile

Detailed description of the system architecture choices made.
How Separation of Concerns (SoC) and Single Responsibility Principle (SRP) was followed.

Architecture Diagram.

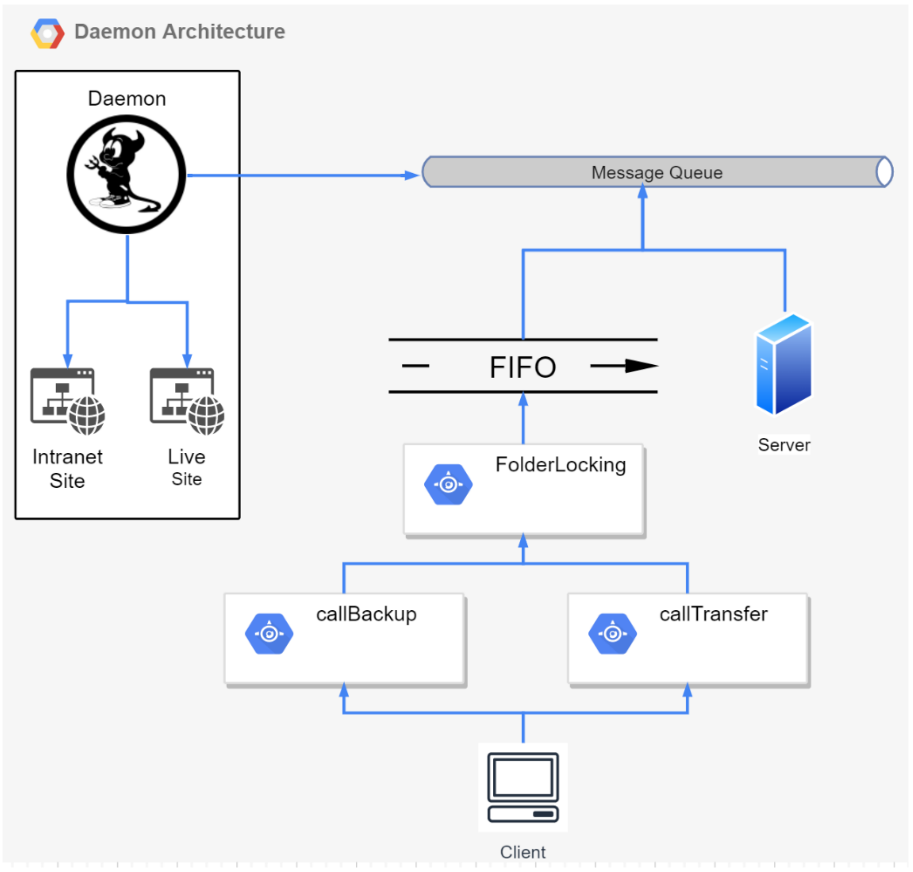


Figure 1: Daemon Architecture

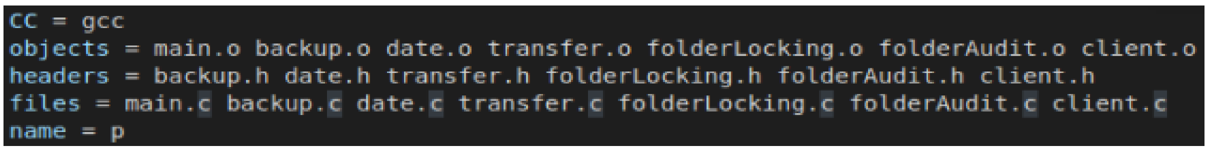
The system architecture diagram followed in this project is as shown in figure 1. Here, it can be seen how the Client creates a queue which is then received by the server. Following this, the server writes to the daemon using the message queue as a medium for the server to receive it.

The client creates the queue using the files 'callTransfer.c', 'callBackup.c' and 'folderLocking.c'. These files are coded in a FIFO format and are implemented pipes to communicate between processes This FIFO format is in a FIFO format using a buffer to write and receive the appropriate call. The client communicates to the server which in turn communicates to the daemon; this is executed using pipes.

Separation of Concerns (SoC) is the principle whereby its main purpose is to ensure that different concerns are no located in the same code base. Similarly, the Single Responsibility Principle states that aspects of a problem are in fact wo separate responsibilities and therefore should be implemented in separate classes.

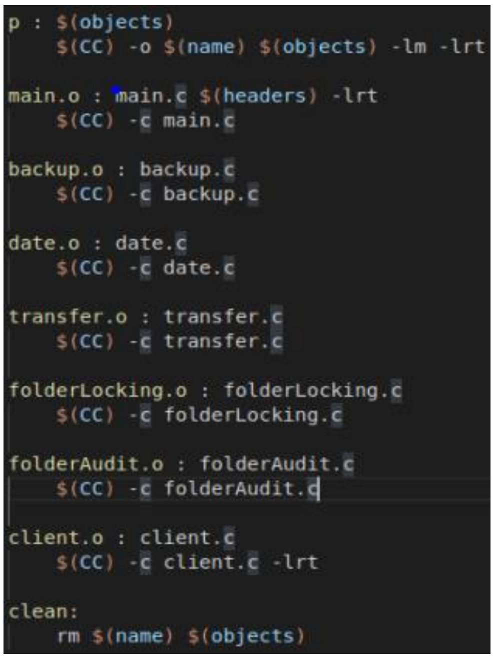
This was accomplished using a makefile and the make command. Implementing a makefile is significantly beneficial for the developer as it enables them to compile a program in a very specific sequence. Thus, adhering to the Separation of Concerns and Single Responsibility Principle.

The makefile shown on the right, contains the sequence for creating the main executable. This executable consists of the necessary object files which were compiled from the defined C source files as displayed in the figure below.



```
CC = gcc
objects = main.o backup.o date.o transfer.o folderLocking.o folderAudit.o client.o
headers = backup.h date.h transfer.h folderLocking.h folderAudit.h client.h
files = main.c backup.c date.c transfer.c folderLocking.c folderAudit.c client.c
name = p
```

Figure 2: makefile objects, headers & files



```
p : $(objects)
	$(CC) -o $(name) $(objects) -lm -lrt

main.o : main.c $(headers) -lrt
	$(CC) -c main.c

backup.o : backup.c
	$(CC) -c backup.c

date.o : date.c
	$(CC) -c date.c

transfer.o : transfer.c
	$(CC) -c transfer.c

folderLocking.o : folderLocking.c
	$(CC) -c folderLocking.c

folderAudit.o : folderAudit.c
	$(CC) -c folderAudit.c

client.o : client.c
	$(CC) -c client.c -lrt

clean:
	rm $(name) $(objects)
```

Figure 3: makefile

Feature 2 - Daemon (Setup/ Initialisation/ Management)

Detailed description of the daemon setup:
Startup Script and init process
Daemon control options

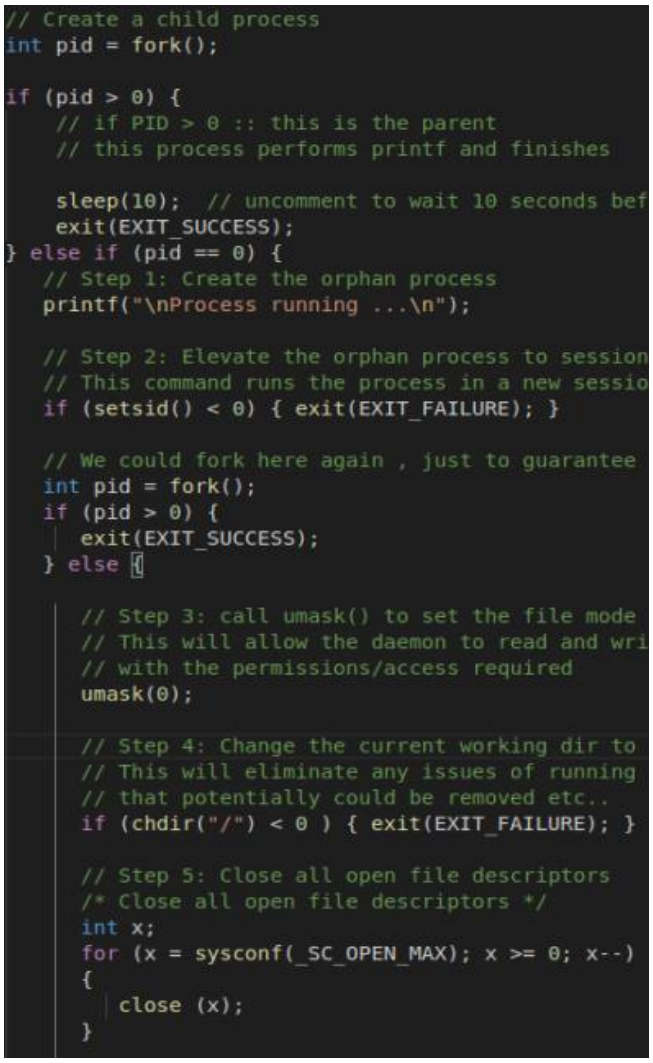
In order to correctly create a daemon, one must follow a certain order of steps.

First, the orphan process is created using the 'fork()' command, before being elevated to session leader, to loose controlling TTY. This is implemented using the 'setsid()' system call.

After this, 'unmask(0)' is called to set the file mode creation mask to 0. Thus, enabling the daemon to read and write files with the required permissions.

Following this, the current working directory is changed to root. Implementing this eliminates any issues of running on a mounted drive, that potentially could be removed.

Finally, all open file descriptors are closed using the 'close()' command.



```c
// Create a child process
int pid = fork();

if (pid > 0) {
    // if PID > 0 :: this is the parent
    // this process performs printf and finishes

    sleep(10);  // uncomment to wait 10 seconds bef
    exit(EXIT_SUCCESS);
} else if (pid == 0) {
   // Step 1: Create the orphan process
   printf("\nProcess running ...\n");

   // Step 2: Elevate the orphan process to session
   // This command runs the process in a new sessio
   if (setsid() < 0) { exit(EXIT_FAILURE); }

   // We could fork here again , just to guarantee
   int pid = fork();
   if (pid > 0) {
      exit(EXIT_SUCCESS);
   } else {

      // Step 3: call umask() to set the file mode
      // This will allow the daemon to read and wri
      // with the permissions/access required
      umask(0);

      // Step 4: Change the current working dir to
      // This will eliminate any issues of running
      // that potentially could be removed etc..
      if (chdir("/") < 0 ) { exit(EXIT_FAILURE); }

      // Step 5: Close all open file descriptors
      /* Close all open file descriptors */
      int x;
      for (x = sysconf(_SC_OPEN_MAX); x >= 0; x--)
      {
         close (x);
      }
```

Figure 4: Daemon Setup

The start-up script starts with the definition of the 12am as the time in which the intranet site is first backed up, then transferred to the live site. Following this, a watchfile is used to track any changes made to the defined folder of '/var/www/html'. Throughout the start-up and init processes, messageQueue(), openlog() and syslog() are used to log appropriate messages as to how each function performed. The message 'Started auditing' is then displayed to the server when a watch has successfully been placed on the folder.

For this project, the client requested that a singleton pattern be followed so that only a single copy of the daemon should be running at a time. This eliminates the possibility of multiple instances carrying out duplicate operations. The propose of a singleton pattern is that it will restrict us to one copy of the daemon running at a time. This was implemented using the file, 'singleton.c' and header 'singleton.h' and is called before the daemon enters a running state.

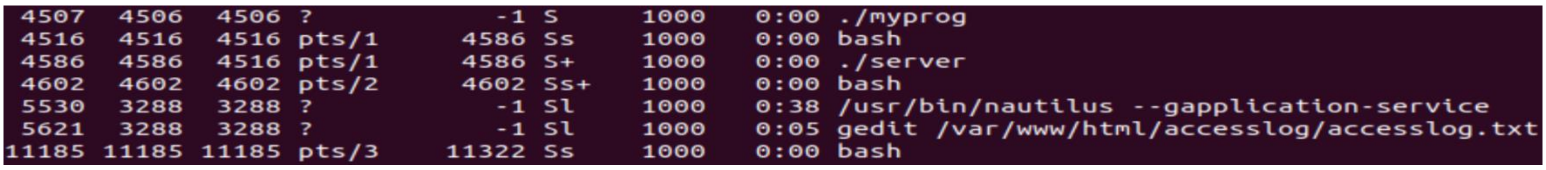
Feature 3 - Daemon (Implementation)

Detailed description of the process followed to create the background process.

The daemon is implemented using a makefile which is used to set the order in which applications are compiled and ran. The order at which this project follows can be seen in the makefile or in figure 3 above. The main is used to create the daemon process using the steps outlined in detail in feature 2 and once created, the daemon is executed as a background process.

A daemon was created for this project by creating the orphan process, elevating it to session leader, calling unmask() to allow the daemon read/write permissions and changing the current working directory to root. Following this, all open file descriptors are closed using the 'close()' command.

The background processes in a UNIX environment can be shown in the terminal by running the command, 'ps -jx'. A sample of the background process running on my computer can be seen in the diagram below.



```
 4507  4506  4506 ?          -1 S     1000   0:00 ./myprog
 4516  4516  4516 pts/1    4586 Ss    1000   0:00 bash
 4586  4586  4516 pts/1    4586 S+    1000   0:00 ./server
 4602  4602  4602 pts/2    4602 Ss+   1000   0:00 bash
 5530  3288  3288 ?          -1 Sl    1000   0:38 /usr/bin/nautilus --gapplication-service
 5621  3288  3288 ?          -1 Sl    1000   0:05 gedit /var/www/html/accesslog/accesslog.txt
11185 11185 11185 pts/3   11322 Ss    1000   0:00 bash
```

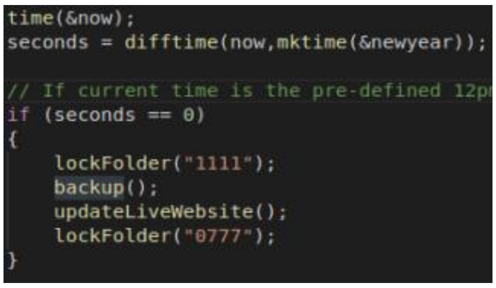
Figure 5: Background processes

Here, it can be seen how both, the application 'myprog', and the server, 'server' are running as background processes. Both of these processes can be killed by using their ID on the left. In order to stop the 'myprog' from running in the background, one must kill it.

Feature 4 - Backup Functionality

Detailed description of the backup implementation

A backup of the website is created at the client's specified time of midnight. This pre-defined time is set in the main to 0 to equal 12am. The daemon then calls backup() when the system's current time is 12am.
It achieves this by including 'backup.h' as a header and comparing the system's current time to that of 12am.

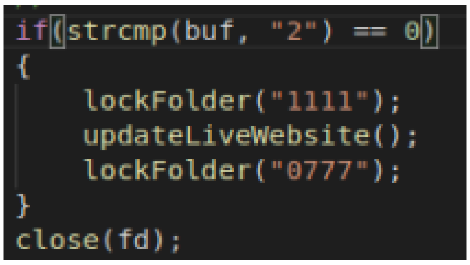


```
time(&now);
seconds = difftime(now,mktime(&newyear));

// If current time is the pre-defined 12p
if (seconds == 0)
{
    lockFolder("1111");
    backup();
    updateLiveWebsite();
    lockFolder("0777");
}
```

Figure 6: Compare current time with pre-defined 12am

This same backup() function is called after the daemon's buffer receives "1" from the user running 'callBackup.c'. This is executed in 'callBackup.c' using the write() command. At either of these times, the function backup() defined in 'backup.c' is executed.

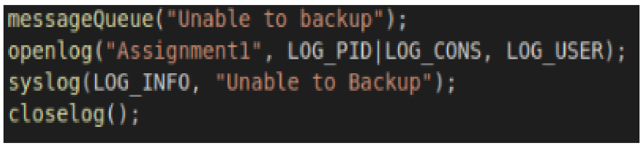


```
if(strcmp(buf, "2") == 0)
{
    lockFolder("1111");
    updateLiveWebsite();
    lockFolder("0777");
}
close(fd);
```

Figure 7: Listens for callBackup()

When the function is implemented, a source and destination path specified. This is defined using the command:
' char * path = "cp -R /var/www/html/intranet /var/www/html/Backup/"; '

The backup is then executed using a buffer and 'malloc' to allocate it the necessary space to copy the relevant folder. Following this, the buffer is concatenated with the current date (which is retrieved using 'date.c') to append its folder name with the current date. This allows each backup to be easilt maintained.



```
messageQueue("Unable to backup");
openlog("Assignment1", LOG_PID|LOG_CONS, LOG_USER);
syslog(LOG_INFO, "Unable to Backup");
closelog();
```

Figure 8: Queue server message

When the backup completes succesfully, the following messages get printed and logged to server. This concludes the backup functionality.

Feature 5 - Transfer Functionality

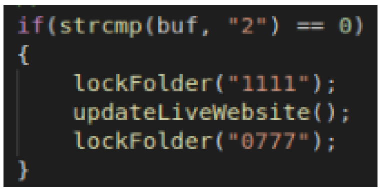
Detailed description of the transfer implementation

A transfer of the website is also created at the client's specified time of midnight. This pre-defined time is set in the main to 0 to equal 12am. The daemon then calls updateLiveWebsite() when the system's current time is 12am and after the backup() has been called. It achieves this by including 'transfer.h' as a header and comparing the system's current time to that of 12am.

Similar to that of backup(), updateLiveWebsite () is called after the daemon's buffer receives "0" from the user running 'callTransfer.c'. This is once again executed in 'callTransfer.c' using the write() command. Both of these cases result in updateLiveWebsite () being executed inside 'transfer.c'.

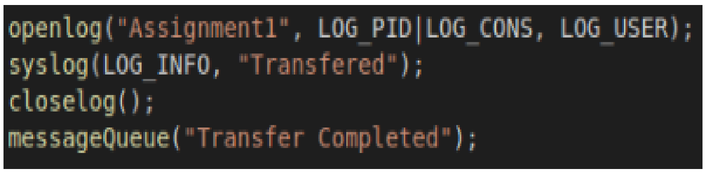
Just as the backup() function was implemented, updateLiveWebsite () defines a source and destination path. With this function, the bootstrap template stored in the 'intranet' folder on the server, gets copied into that of the 'live' folder. This is part of the core functionality of what the client requested and occurs after the backup at 12am. The command used to specify the path to copy is:
' char * path = "cp -R /var/www/html/intranet/ /var/www/html/live"; '

When the attempted transfer completes, the function logs the appropiate result to the server. Once the intranet site has been modified in any way, the transfer copies across the modified file or files. When completed succesfully, the following messages get printed and logged to server.



```
if(strcmp(buf, "2") == 0)
{
    lockFolder("1111");
    updateLiveWebsite();
    lockFolder("0777");
}
```

Figure 10: Listens for callTransfer()



```
openlog("Assignment1", LOG_PID|LOG_CONS, LOG_USER);
syslog(LOG_INFO, "Transfered");
closelog();
messageQueue("Transfer Completed");
```

Figure 9: Queue server message

Feature 6 - Lockdown folder for Backup / Transfer

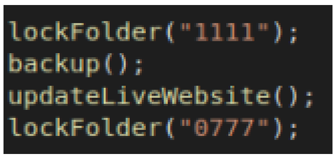
Detailed description of the lockdown functionality/implementation

The lockdown functionality within this project, is called through the main and is executed when the backup and transfer functionality is called. In each case, the lockFolder() function (defined in ‘folderLocking.c’) passes the permission it is assigning to the defined path of ‘/var/www/html/’. The commands used are called both before, and after the backup() or updateLiveWebsite() is called and are as follows:

‘lockFolder(“1111”);’ - (locks the folder’s permissions)
‘lockFolder(“0777”);’ - (unlocks the folder’s permissions)

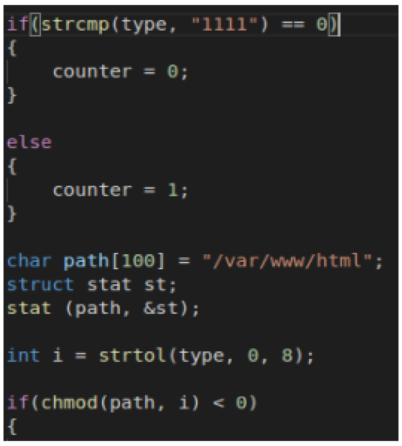
A counter is then used to carry out the appropriate code depending if the variable “1111” is passed as a parameter or not. This can be seen in the figure. Here, it can be also seen how “var/www/html” is defined as the path and is assigned the relevant permissions using “chmod”.

The locking of folders was implemented into this project in order to adhere to the client’s demands. The client requested that the when the backup/transfer begins, no user should be able to modify site content and this is achieved using ‘folderLocking.c’ as described above.



```
lockFolder("1111");
backup();
updateLiveWebsite();
lockFolder("0777");
```

Figure 12: lockFolder() called in main



```
if(strcmp(type, "1111") == 0)
{
    counter = 0;
}

else
{
    counter = 1;
}

char path[100] = "/var/www/html";
struct stat st;
stat (path, &st);

int i = strtol(type, 0, 8);

if(chmod(path, i) < 0)
{
```

Figure 11: Check folder permissions & path

Feature 7 - Reporting (IPC)

Detailed description of how child processes communicate success/failure of tack to be completed to parent process etc….

In this project, the client communicates to the server using a message queue which operates as a linked list of messages. These sent messages are stored in the system kernel and can be differentiated using their unique identifier.

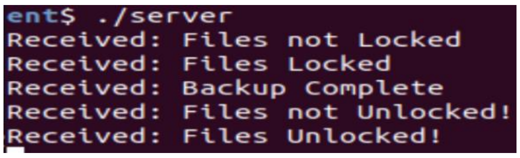
Figure 14 shows how "/MyQueue" was created with the option "0_WRONGLY" which enables the queue to send messages only. This is an appropriate option as the client only needs to send messages to the server. Once the queue has been created, the passed parameter, message, is then sent to the queue for the server to read.

Similar to the client, the server is implemented by the line:
"mq = mq_open("/MyQueue", 0_CREAT | 0_RDONLY, 0644, &queue_attributes);"

The option '0_CREAT' creates a message queue if it does not already exist, while the option, '0_RDONLY' opens the queue to receive messages only. The server then receives messages from the queue using the function 'mq_receive()' as shown in figure 14.

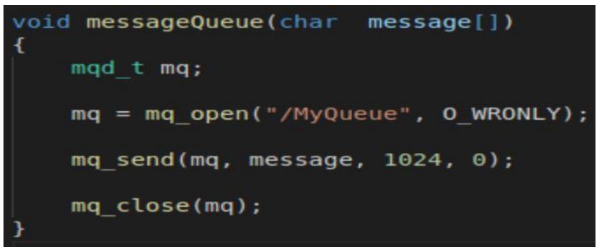
After the client has sent the message, mq_close() is used to close the queue. The server also closes it's queue after being terminated by a user.

The communication between the client and the server is then displayed on the server so that each function within the project can be traced. This can be seen in figure 12.



```
ent$ ./server
Received: Files not Locked
Received: Files Locked
Received: Backup Complete
Received: Files not Unlocked!
Received: Files Unlocked!
```

Figure 13: Server reporting



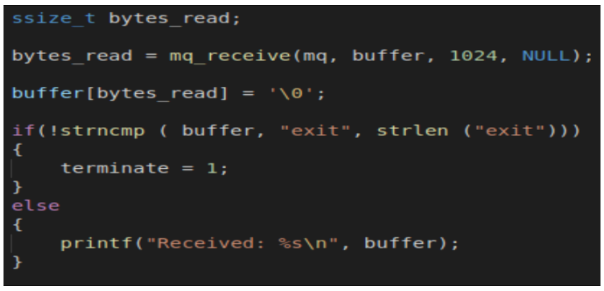
```
void messageQueue(char  message[])
{
    mqd_t mq;

    mq = mq_open("/MyQueue", O_WRONLY);

    mq_send(mq, message, 1024, 0);

    mq_close(mq);
}
```

Figure 15: Client creates Queue using mq_open()



```
ssize_t bytes_read;

bytes_read = mq_receive(mq, buffer, 1024, NULL);

buffer[bytes_read] = '\0';

if(!strncmp ( buffer, "exit", strlen ("exit")))
{
    terminate = 1;
}
else
{
    printf("Received: %s\n", buffer);
}
```

Figure 14: Server code to receive a message from "MyQueue"

Feature 8 - Logging and Error Logging

Detailed description of the error and logging functionality included in the code solution.

Logging is an extremely important aspect of any project as it proves beyond doubt who modified the data, what was modified, and at what time it occurred. As a result, 'Auditd' was integrated into this project.

It is responsible for watching the defined path for any changes and logging the appropriate data once this directory has been altered. Therefore, a watch was put on the desired folder in the main by using the line:

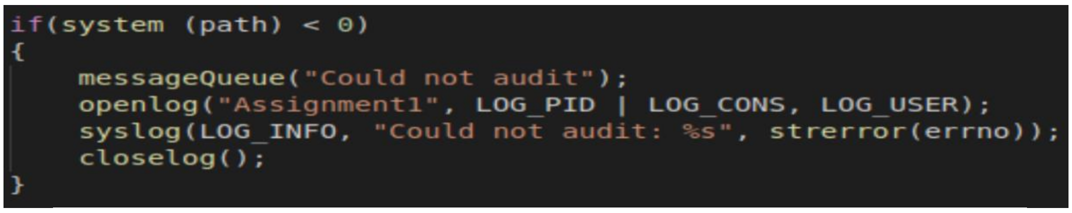
'char * watchFile = "auditctl -w /var/www/html -p rwxa";'

The necessary error logging and logging can be seen in figure 16 below where messageQueue(), openlog() and syslog() are implemented so that the appropriate logs are recorded.

The function 'folderAudit()' is then called at the end of the main and subsequently executes its function as defined in 'folderAudit.c' Here the text file, accesslog.txt, is created/appended when there has been a change to the folder defined above. This is achieved using the line:

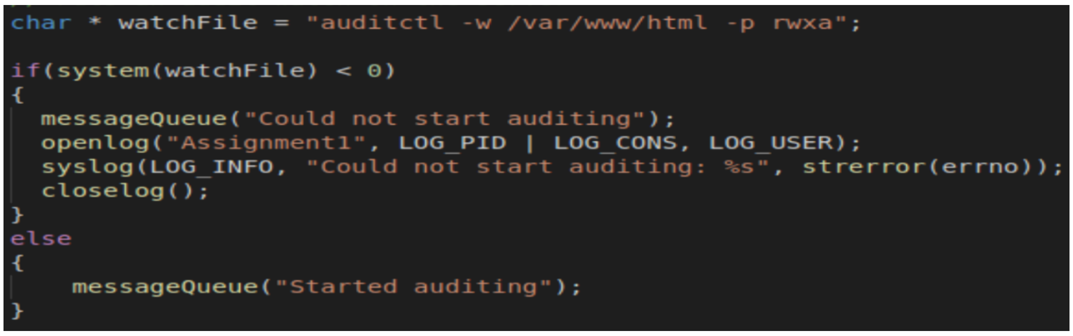
'char * path = "ausearch -f /var/www/html/ > /var/www/html/accesslog/accesslog.txt";'

If the audit fails, a series of logs are displayed to both, the server, and the system implementing it. These are shown in the figure 15 below.



```
if(system (path) < 0)
{
    messageQueue("Could not audit");
    openlog("Assignment1", LOG_PID | LOG_CONS, LOG_USER);
    syslog(LOG_INFO, "Could not audit: %s", strerror(errno));
    closelog();
}
```

Figure 16: folderAudit.c logging



```
char * watchFile = "auditctl -w /var/www/html -p rwxa";

if(system(watchFile) < 0)
{
  messageQueue("Could not start auditing");
  openlog("Assignment1", LOG_PID | LOG_CONS, LOG_USER);
  syslog(LOG_INFO, "Could not start auditing: %s", strerror(errno));
  closelog();
}
else
{
    messageQueue("Started auditing");
}
```

Figure 17: main.c logging

Conclusion

Summary of the implementation and achievement

The implemented project achieved most, if not all the client's requests and can now be used as a template for other projects. It also helped to enhance my technical skills in a Linux environment.

The benefit of achieving the transfer and backup functionality is that it significantly shortens development time needed for source management. Not only this, but each change made to the intranet site, gets logged in the file, 'accesslog.txt'. Each entry to this file is also time-stamped and logged with the username of whom made the change.

This project was therefore a worthwhile experience and one that I will potential implement for future web development projects.