

Systems Software Report CA1

DT282-4

BSc in Computer Science (International)

**Jennifer Nolan**

**C16517636**

School of Computer Science

TU Dublin – City Campus

**13th March 2020**

**Table of Contents**

*Functionality Checklist* 3

*Feature 1 - System Architecture including makefile* 4

*Feature 2 - Daemon (Setup/ Initialisation/ Management)* 4

*Feature 3 - Daemon (Implementation)* 4

*Feature 4 - Backup Functionality* 5

*Feature 5 - Transfer Functionality* 5

*Feature 6 - Lockdown folder for Backup / Transfer* 5

*Feature 7 - Reporting (IPC)* 5

*Feature 8 - Logging and Error Logging* 5

*Conclusion* 6

# *Functionality Checklist*

|  |  |  |
| --- | --- | --- |
| ***Feature*** | ***Description*** | ***Implemented*** |
| F1 | System Architecture including makefile | Yes |
| F2 | Daemon (Setup/Initialisation/Management) | Attempted |
| 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

Link to Video:

<https://youtu.be/KvqAaYowJNc>

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:

A close up of a logo

Description automatically generated

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Jennifer Nolan

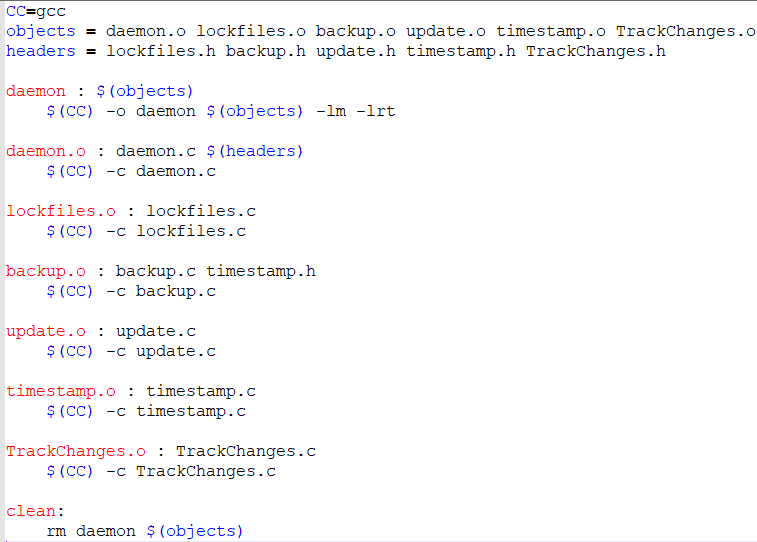
13th March 2020

# *Feature 1 - System Architecture including makefile*

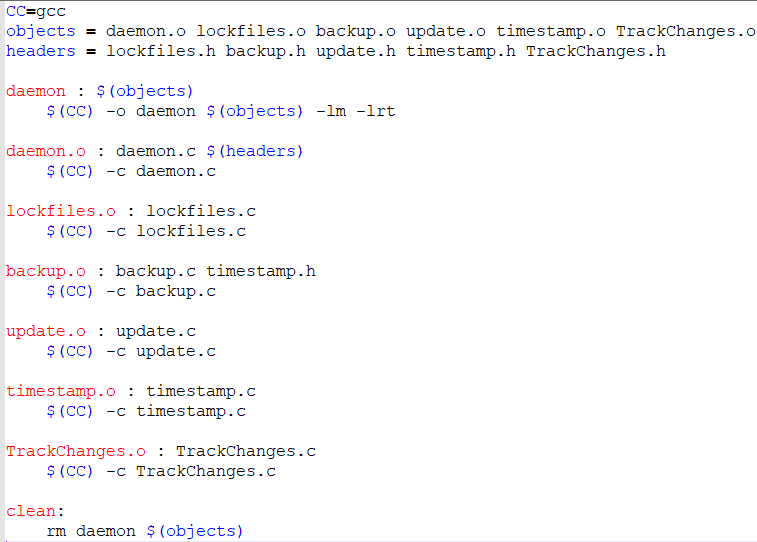
The system created for this assignment consists of a daemon that is run in the background. This daemon executes a backup and update of the live site every night at a predefined time, i.e. midnight. Once the time reaches midnight the files are locked and the backup and update of files are all carried out within the daemon in the background. When it is not time for the backup or update to run the daemon is kept running in the background.

The Separation of Concerns principle can be seen in this assignment. The Separation of Concerns (SoC) and Single Responsibility Principle (SRP) are closely related to one another. The main purpose of both of these principles is to make sure various class behaviours are not used in the same code. Essentially, this involves encapsulating code and behaviour to allow for code reuse. In the case of this assignment all of the major processes carried out, i.e. updates, backups and file locking, are all implemented using their own functions and header files which can be accessed by any program that needs to use them. This means that not only can the daemon process make use of these functions for updating and backing up, but the same functions can be used by the on-demand program for the same purposes. This essentially allows for the same code to be implemented in two different programs.

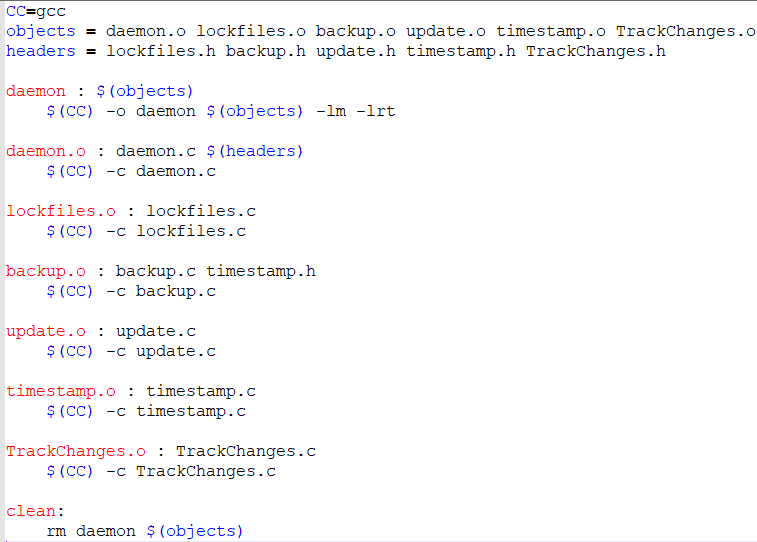
The code exert below demonstrates how the make file for this assignment was implemented to compile the program files. All of the object files were specified under the object variable and all of the header files needed to compile the daemon process are stored in the header variable.



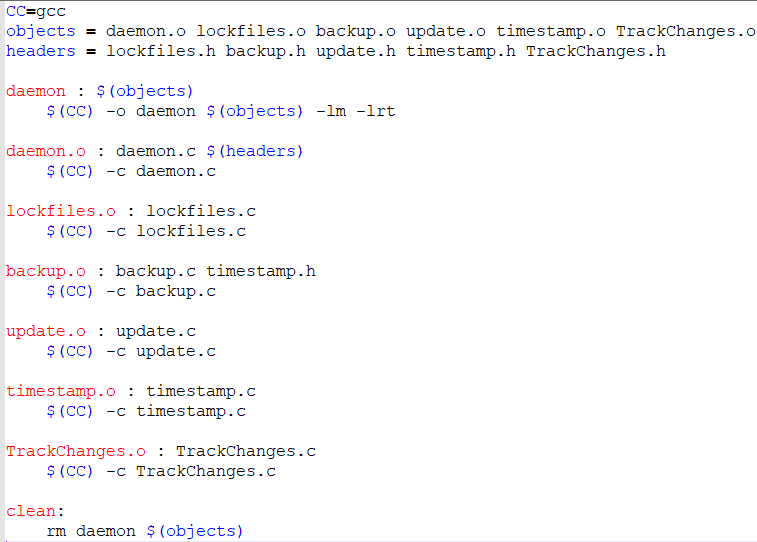
From there, the daemon executable program is compiled, as shown in the image below. This uses the object files from the other processes that are implemented through the daemon process.



After the daemon executable program is compiled all of the individual .c programs are compiled to create the equivalent .o files.



Lastly, an option to clean or remove the previously compiled programs was implemented, as shown in the code below.



# *Feature 2 - Daemon (Setup/ Initialisation/ Management)*

For this part of the assignment an attempt was made to get the daemon process to automatically run using the init script shown below.

The below program was trying to implement an automatic start, stop or reload to the daemon process using a script stored in the /etc/init.d file path. However, the program was unable to run due to the following error that was received.

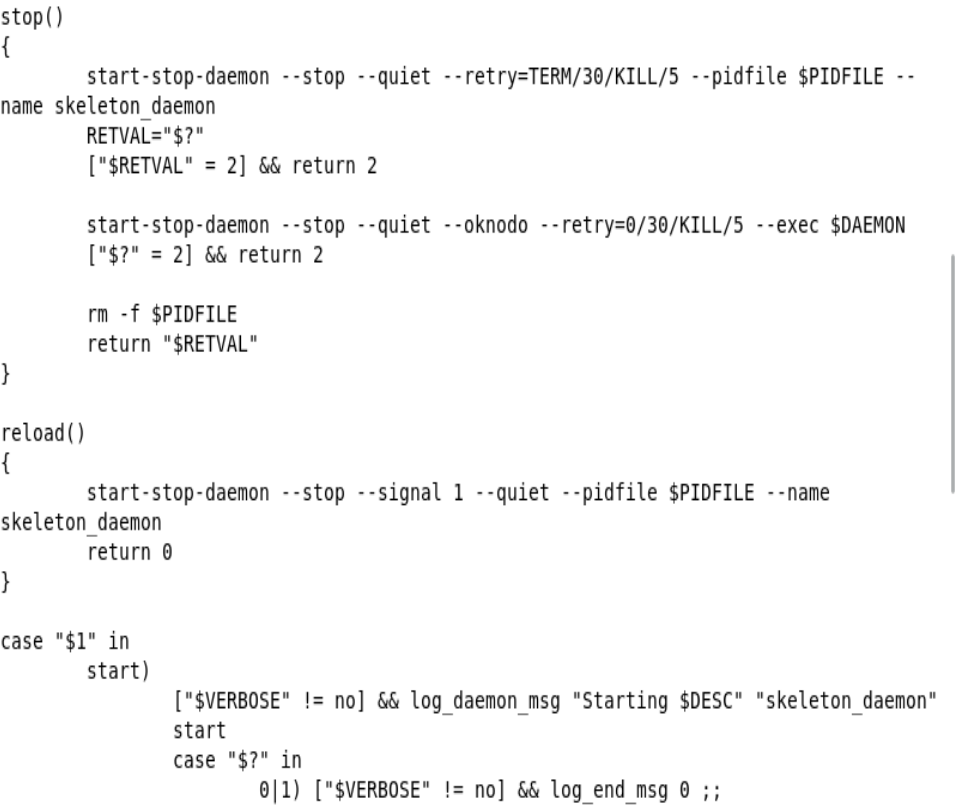


With regards to the script used to automatically run the daemon process, firstly the parameters were set, as shown in the image below.

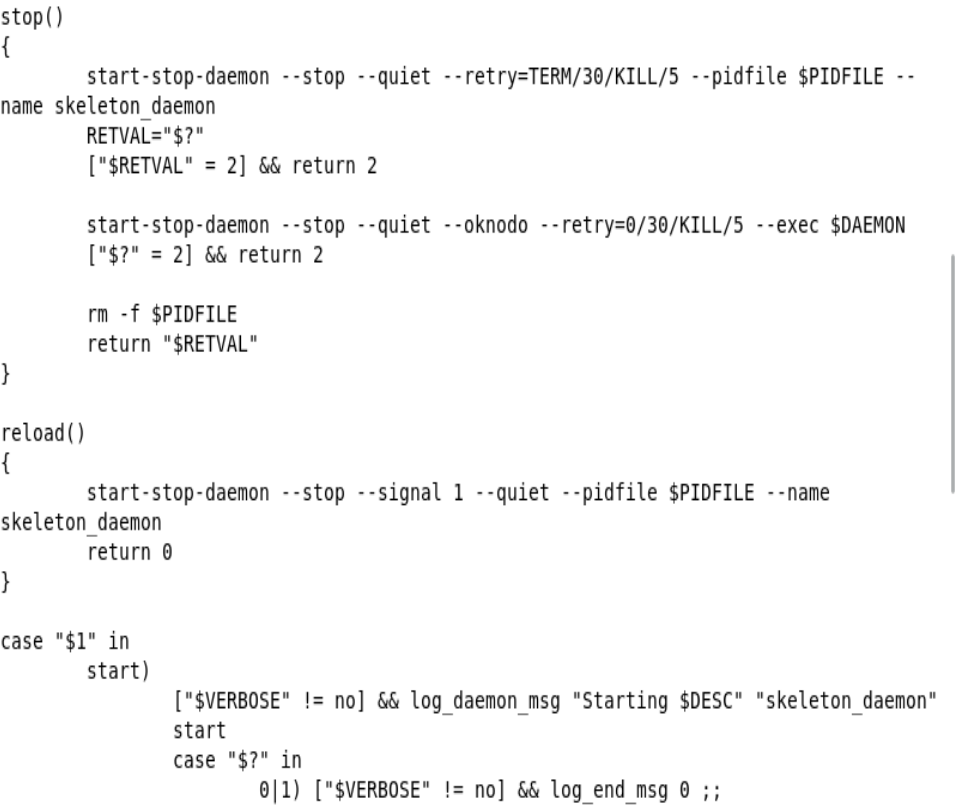


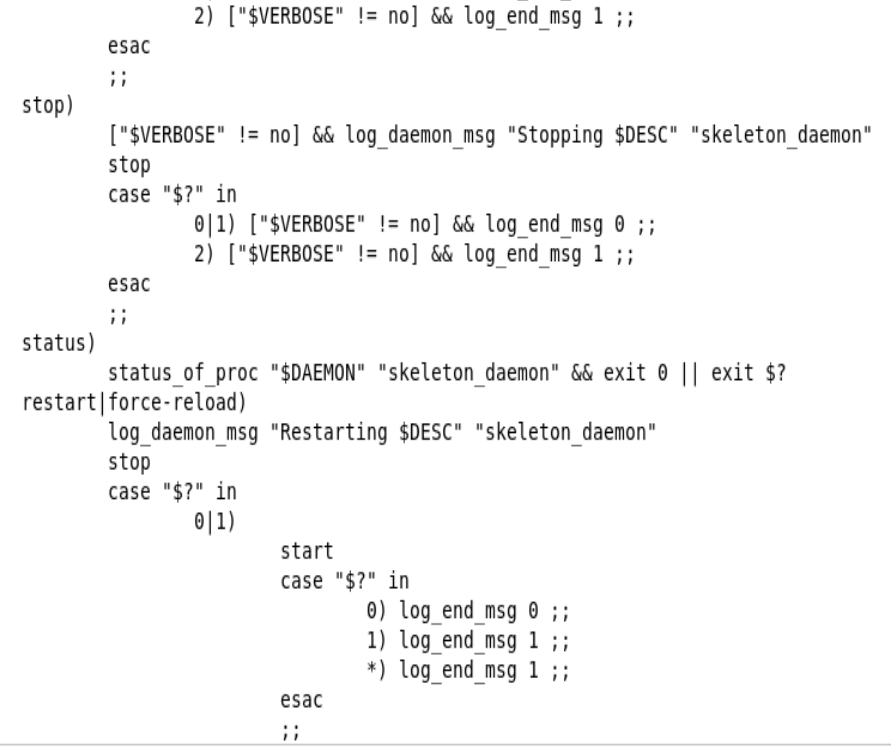
Next the start function, shown in the image below, that would automatically start the daemon was implemented. The same format used to start the daemon process is also used to stop and reload the daemon, with the term start being replaced with stop or reload.





After the above functions, that will determine the running of the daemon process, are declared the situations in which these functions are run are determined. For this a switch statement is used. This switch statement will determine, depending on the value entered, which function will be run, i.e. start, stop or reload.

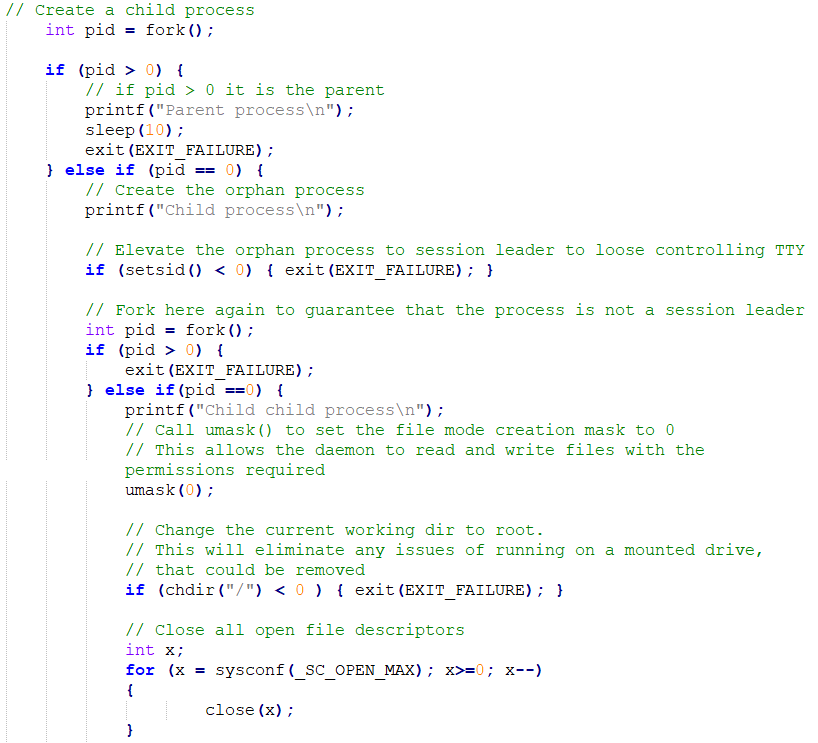




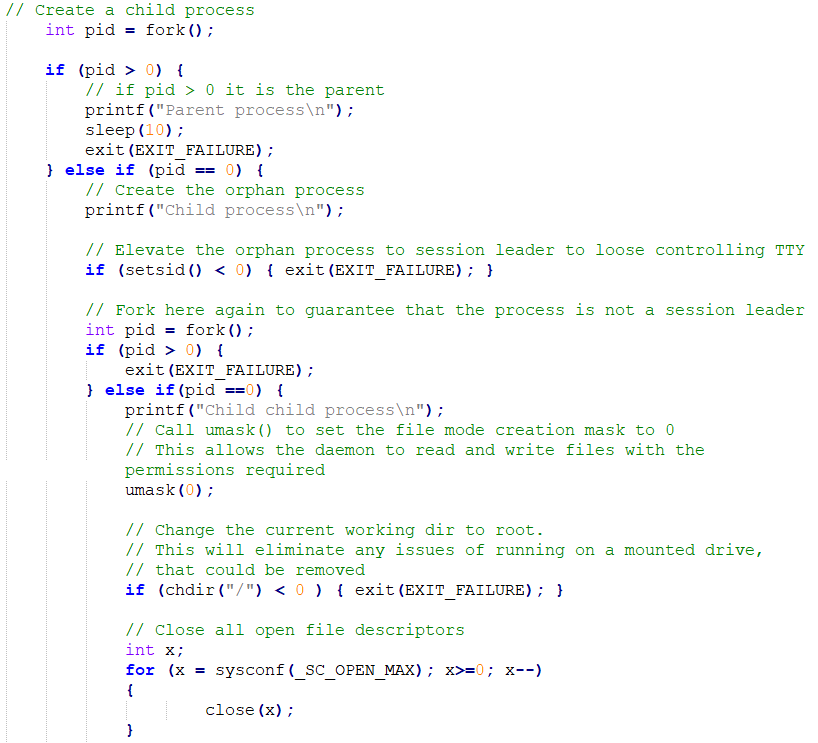


# *Feature 3 - Daemon (Implementation)*

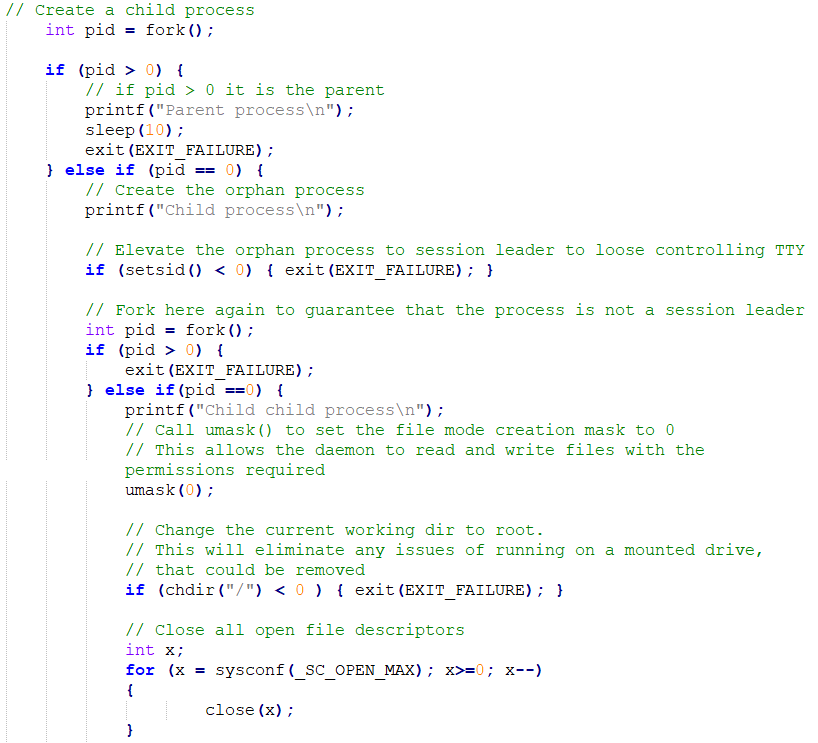
Feature 3 of this assignment consisted of the daemon implementation. The daemon implemented for this assignment uses the same set up as the labs conducted in week 4. In order to create a daemon that allows processes to run in the background the following steps needed to be implemented. Firstly, the process was forked to create a parent and child process, as shown in the image below. The parent process then exits, and the child process is left as an orphan process.



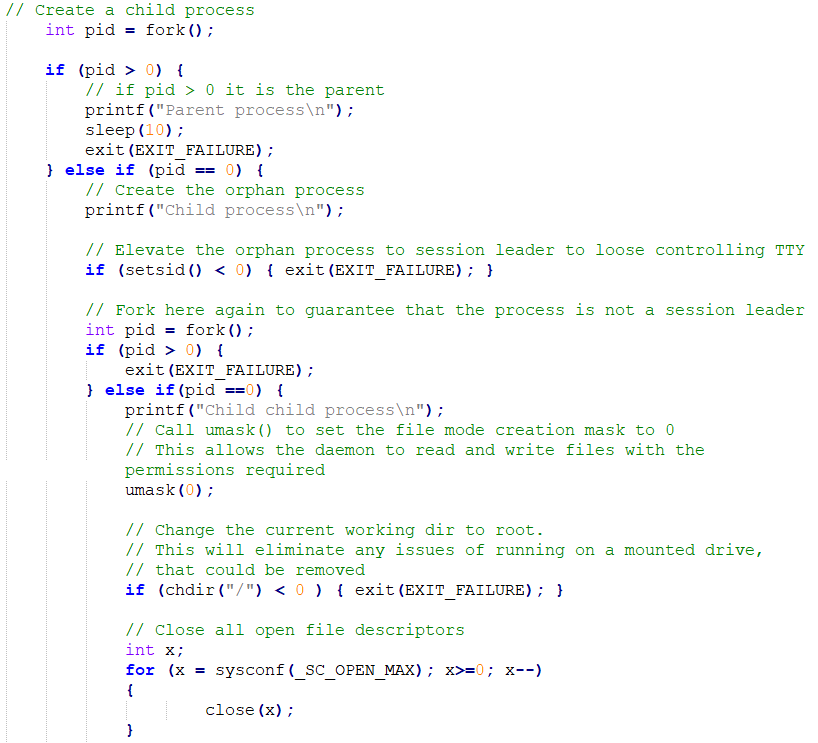
Following on from this the orphan process is elevated to session leader in order to lose the controlling terminal, as shown in the image below.



After this the orphan process is then also forked. This second fork results in, what was previously the child process, becoming the parent process of the new child process. The parent process, like the previous parent process, exits, leaving the child process as the final process running, as shown in the image below.



This new child process is where the daemon is implemented and is where the rest of the functions within this assignment are called, so that they run in the background. The child process is set to have a file mode creation of 0, as shown in the image below. This allows the child process to read and write to files. Next, the working directory of the process is changed to root. This removes any possible issues of running the process on a mounted drive. Lastly, all of the open file descriptors are closed.



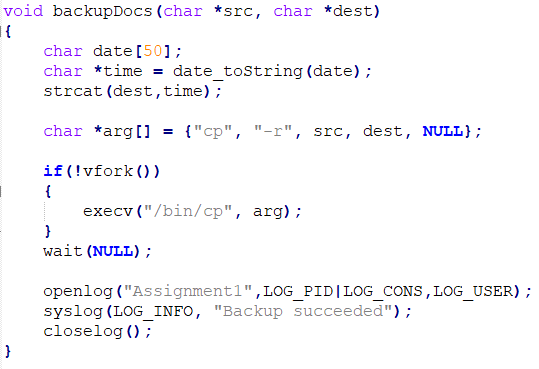
The above-mentioned steps to create a daemon allow the daemon to run the backup and update processes in the background every night at midnight.

# *Feature 4 - Backup Functionality*

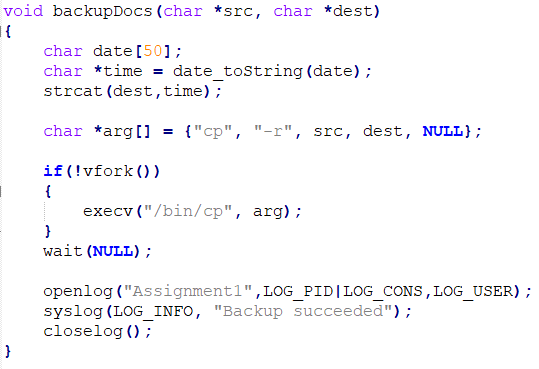
The backup functionality is called, using the function call below, from the daemon when the backup time, midnight, is reached. The backupDocs() function takes in the file path for the live file that needs to be backed up to the backup file, which is the second file path that is passed to the backupDocs() function.



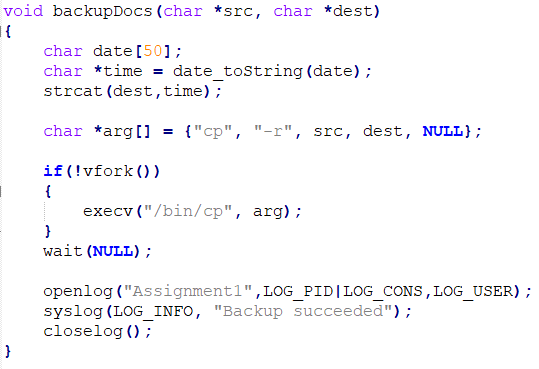
The following processes are used to implement the backup functionality required for this project. Firstly, in the backupDocs() function the date the backup takes place is obtained and appended to the end of the backup file path, as shown in the code snippet below. This allows for the different backed up file versions to be discerned from one another by date.



Next, the command that is going to copy the live file to the backup folder is placed in a character array to be executed next using execv(). This command, shown in the image below, is then executed in a child process created using vfork(). The vfork() function was chosen because it is useful when it is used to immediately issue an execv() command.



Once the copy command is executed and the live site is backed up to the backup folder wait(NULL) is called. Wait(NULL) is used to prevent a zombie process from being created. Lastly, as shown below, the backup process is logged and accounted for using the syslog() function.



# *Feature 5 - Transfer Functionality*

The update, or transfer, functionality is implemented once the live site have been backed up. The updateSite() function, shown below, takes in the paths that the function will be updating from and to, similarly to the backupDocs() function. Essentially the updateSite() function is updating the live site file with the contents of the intranet file.



The updateSite() function begins by gathering the time and date the update occurs, shown in the code below. This information is to be used later for the update log.



Next in the updateSite() function, and similarly to the backupDocs() function, the command that is going to copy and replace the content of the live site file with the content of the intranet site file is placed in a character array to be executed next. This command, like the command used for the backup, is executed using vfork(), shown in the image below. Also included in this part of the code is the file handling to open the update log file which will store the date and time of when an update occurs.



The copy command is executed, using the above code, and the live site is updated with the contents of the intranet site file. Once again, as shown in the code below, wait(NULL) is implemented to prevent a zombie process from being created.



As well as that, the update log file, opened previously is appended to with the time and date of the update that just occurred. The update log contains logs that are formatted as shown below. All updates will be accounted for in this single file as they can be easily distinguished from their dates.



Lastly, as part of the update functionality, the update process is logged and accounted for using the syslog() function, shown in the image below.

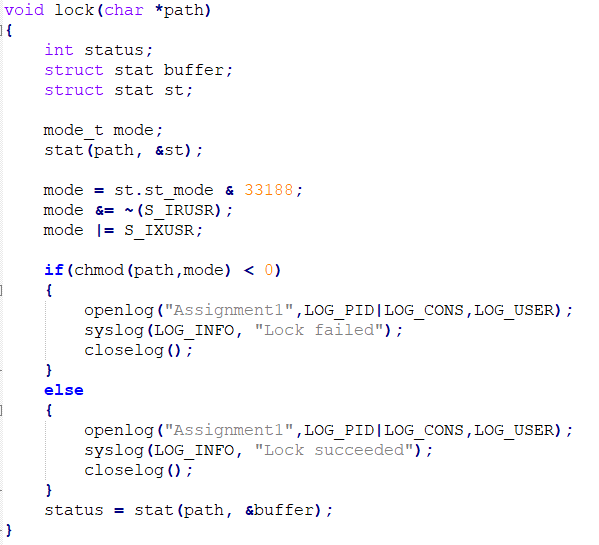


# *Feature 6 - Lockdown folder for Backup / Transfer*

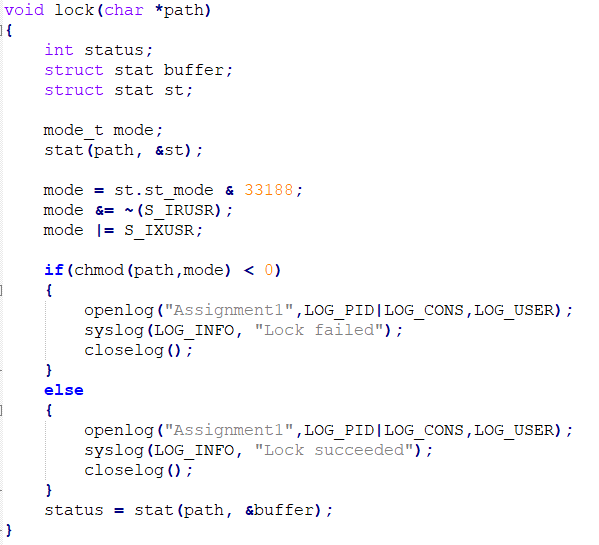
The locking of files functionality is conducted before any file backups or updates occur. Similarly, to the update and backup functions, to implement the lock functionality the live file path first needs to be passed to the lock() function, as shown below.



The lock() function uses the past path to lock the live site file during backups and updates. This means that the live files permissions are changed and it cannot be modified during the backup or update. The lock() function, shown in the code below, essentially disables write permissions to the specified file to prevent users from modifying it.



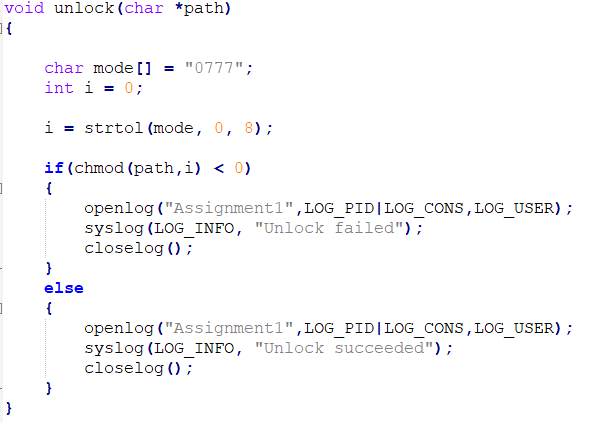
As shown in the image below, if the lock is successful at locking the file it is logged as such in the system log. The same process is followed if the lock fails to lock the file.



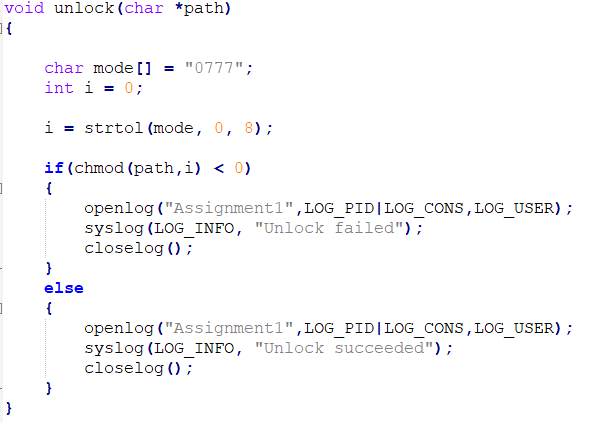
The unlock functionality follows the same procedure as the lock functionality. To implement the unlock functionality the live file path first needs to be passed to the unlock() function, as shown below.



The unlocking of the live file path is conducted after the file is backed up and updated. This allows the past file path’s permissions to be modified again. The unlock() function, as shown below, essentially reenables the file write permissions to enable users to edit the file again.

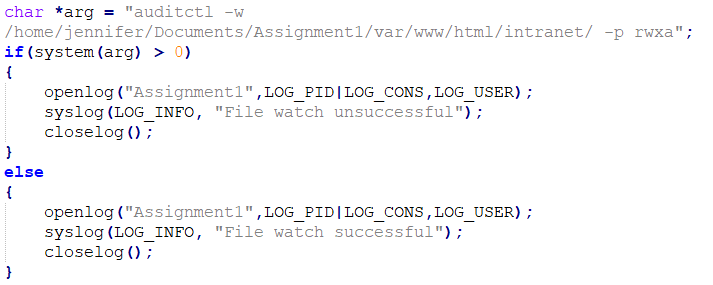


As shown in the image below, if the unlock is successful at unlocking the file it is logged as such in the system log. The same process is followed if the unlock fails to unlock the file.



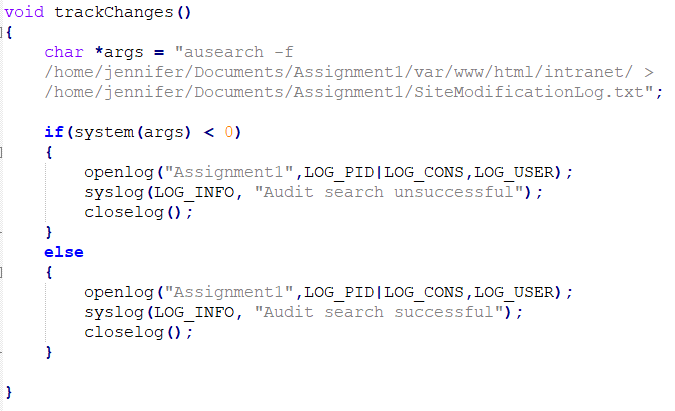
# *Feature 7 - Reporting (IPC)*

The auditd functionality was used within this assignment to place a watch on the intranet file. This watch is used to monitor any edits or updates that have been made to the intranet file. The following code snippet demonstrates how the watch was placed on the intranet file path to monitor for any updates. Firstly, the command, that is used to set the watch on the intranet file, is stored in a character string to be run next. The defined command is then executed using the system() function, as auditd requires root privileges to run. If the watch was successfully placed on the file then it is logged as successful in the system log. Whereas if the watch was not placed on the file successfully then it is logged as unsuccessful in the system log.

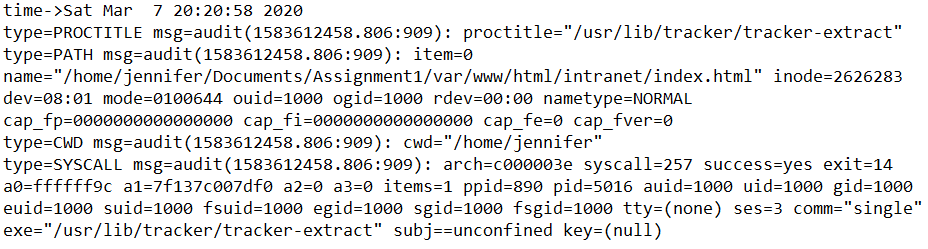


Once the above watch is placed on the intranet file auditd keeps a record of any changes or updates made to the watched file, i.e. the intranet file. These edit records can be viewed using the ausearch command provided by auditd. The implementation of the ausearch command can be seen in the TrackChanges() function in this assignment. This function is implemented after the live site is backed up and before the live site is updated with the contents of the intranet file in the daemon process.

The following code exert shows how the auditd ausearch functionality was utilized to provide a log of the changes made to the watched intranet site.



In the above code the ausearch command, used to retrieve the information about the changes made to the intranet site file, is stored in a character string. This command is then run on the system. Once this command is executed it retrieves the information for all the edits made to the intranet site file and stores these in the SiteModificationLog.txt file each distinguishable by the date and time they were captured. The executed command provides a log with the following format.

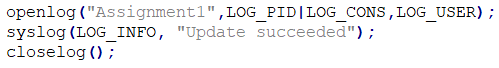


If the log retrieval for the watch placed on the intranet file was successful then it is logged as successful in the system log. Whereas if the log retrieval was not successful then it is logged as unsuccessful in the system log.

In order for the above commands to run in the daemon process root privileges are required. This involved adding rules for the user to the sudoers file using the visudo editor to edit the sudoers configuration file. The following rule was added to the sudoers file to enable the user to run with root privileges: jennifer ALL=(ALL:ALL) NOPASSWD:ALL. Once this rule was set the above commands included in the daemon process could be run, using root privileges, with the following command: sudo -u jennifer /home/jennifer/Documents/Assignment1/daemon.

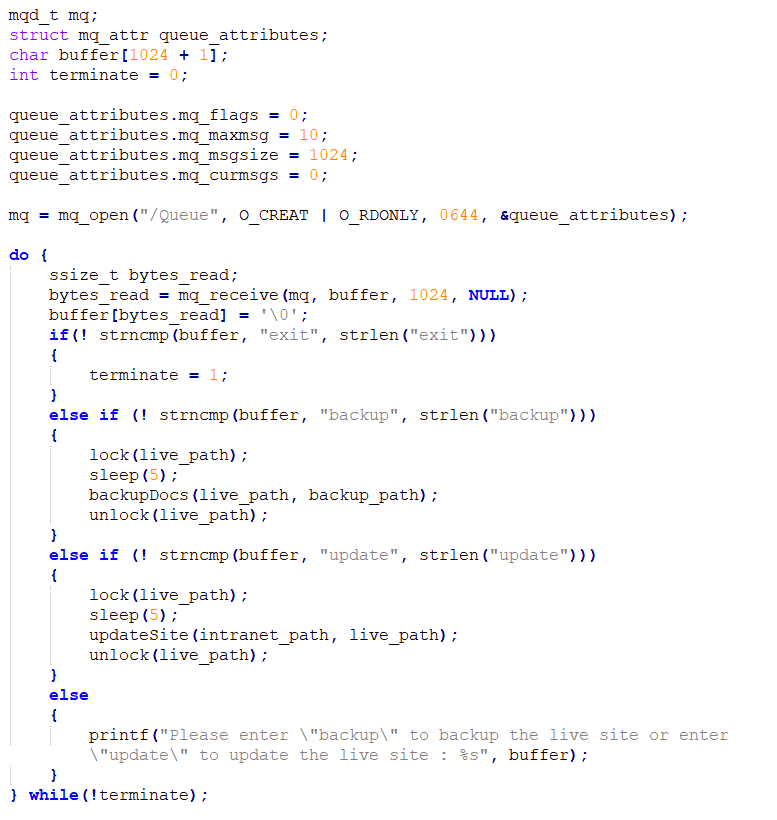
# *Feature 8 - Logging and Error Logging*

The system logging, an example of which is shown below, writes logs to the system. These logs are used to provide information on whether a function, for example the updateSite() function, has run successfully or not. The system logging is utilised in the programs associated with this assignment in the following way. Firstly, the log is opened and given a unique identifying name and id. Once the system log is opened the information that will be placed in the log is done using the syslog() function. Lastly, once the log is finished being edited it is closed

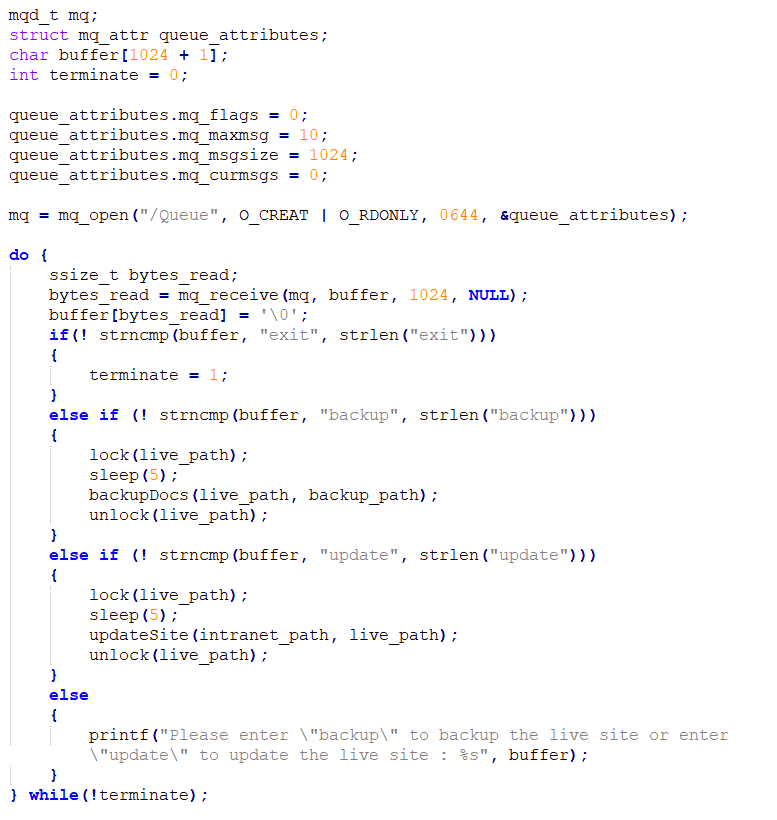


The message queue, which is the other form of logging used for this project, was used to implement the on-demand backup and update of the site. This was implemented separately from the rest of the project. This was done to allow the user to only have to run the server and client files once to update and backup the files on demand. This also means that the daemon that will backup and update the files every night is not tampered with.

The on-demand backup and update of the site were implemented with the use of a client and a server. The server, as shown in the figure below, sets up the message queue named “\Queue”.

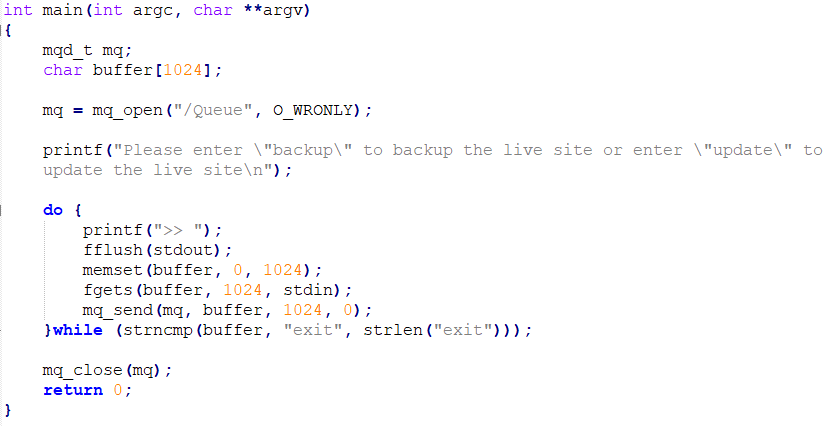


Once the message queue is set up the server running the queue waits for input from the client, as shown below. The server takes in the input from the client and, depending on the input, executes a process. If the client inputs the term “update” the updateSite() function, mentioned previously, is called and the live site file is updated with the contents of the intranet site file. If the client inputs the term “backup” the backupDocs() function, mentioned previously, is called and the live site file is backed up to the backup file. Lastly, if the client enters the term “exit” then both the client and server are closed.



The above-mentioned update and backup processes both apply the lock() function before they take place and the unlock() function after they take place.

The client process, shown below, is where the user can enter the commands to backup or update their site. The user can enter backup or update commands multiple times but can only exit the process by using the exit command once. These inputs from the client are the inputs, previously mentioned, that are taken in by the server to implement the backup and update processes.



The server program that runs the update and backup procedures on demand follow the same procedure as when they are carried out in the daemon process. In other words, the same functions are used by the on-demand program are the ones used in the daemon process. As well as that, the same logs are outputted with the on-demand process as with the daemon process.

# *Conclusion*

As can be seen from the above report, the majority of the criteria set out in this assignment has been met, in my opinion. The features that were implemented include the following:

* The locking and unlocking of files before a backup and update
* The backup of the live site to the backup file
* Tracking any modifications made to the intranet site
* Updating the live site with the contents of the intranet site
* The daemon implementation that allows processes to run in the background
* The file watching was placed on the intranet site file to log any modifications made to the intranet file
* System logs for either successful or unsuccessful programs were also taken

As well as the above-mentioned implemented features the automatic running of the daemon process using the /etc/init.d file was attempted. This however could not be fully implemented due to the error, mentioned previously in this report, not being able to be solved.