ICT374 Major Assignment Two

Introduction (Back to Beginning)

This is a group assignment. Each group consists of two students who are working together to complete the assignment. You must confirm your group's composition no later than the end of Week 6 by email to your lecturer.

One of the purposes of this assignment is to build up team-work skills, including communication, planning, task setting and monitoring, negotiation and conflict resolution, as one of the broad aims of the unit is to develop team-work skills. The mark for each student in a group is calculated using the group mark and that student's percentage of contributions to the assignment. For details see Group Declaration in ICT374 Project Declaration sheet.

Each student is required to team up with another student enrolled in this unit to work on this assignment. Team work skill is worth 20% of the assignment. In the rare cases where a student completes the assignment alone without a partner, the maximum mark he or she could achieve is 80%, because the team-work skills could not be assessed.

This assignment consists of a single project to design and implement a significant piece of software on Unix systems using C.

There are three standard projects for you to choose from. You are only required to complete one of the three projects.

Alternatively you may propose your own project. However it must have a similar level of technical contents and be of a similar level of technical difficulty. The project must be approved by the Unit Coordinator. If you intend to do your own project you should discuss it with your Unit Coordinator and submit a written proposal to your unit coordinator. You can go ahead with your proposal only if it is formally approved by your unit coordinator.

The normal size of assignment group is two. However, in some rare circumstances, a group may be allowed to have more than two persons. However, this requires the formal approval from the unit coordinator. You must submit a written proposal of your project to your unit coordinator and the proposal must be approved by the unit coordinator. The scope of the project, the level of difficulty, the required technical skills and the amount of work must be justifiable for the size of the group.

Regardless of the project you select, as a general requirement, you must not use the function `system` in the implementation of your project.

The three standard projects are listed below. Detailed requirements for each project are given in the later sections of this document.

1. **Project 1 - A Simple Unix Shell**

The project requires the use of more Unix systems facilities and other functions than the other two projects. It also demands good C programming skills in designing the command line parser. Therefore the project is considered to be more difficult to implement than the other two and this will be taken into consideration during assessment. A lot can be learnt about operating systems by doing this project.

2. **Project 2 - A Simple File Transfer Protocol**

   The project requires three pieces of work: a network protocol specification, a client program that implements the client side of the protocol, and a server program that implements the server side of the protocol. As beginners in network programming, most students tend to overlook the importance of correct protocol design and specification. They often write a "protocol" after they have written up the client and the server programs. This is akin to building a palace without an architectural plan and then drawing up a blue print after the palace is already erected (or more likely after the palace is crumbled). Correct specification of the communication protocol is required in this project, and you can expect to be marked down significantly if your protocol specification is not complete or correct or not corresponding to what you have implemented! Having said that, programming wise, it is relatively straightforward to code once the network protocol is *correctly* specified.

3. **Project 3 - A Simple HTTP Client and A Simple HTTP Server**

   This project is similar to Project 2 in some aspects. However it requires you to learn a non-trivial, *existing* network protocol, HTTP, and to thoroughly understand it, at least in the parts that are required by this project. Therefore you will need to make a lot of efforts to conduct an independent research and self-learning before you can even start to write a single line of code. Once you understand the protocol it is not too difficult (but can be tedious) to write a client and a server that is logically sound but inefficient. However, it is a challenge to design and implement an HTTP client and an HTTP server that are both correct to the protocol specification and also efficient!

The project will be marked out of 100 marks, of which 10% is attributed to compliance to the Documentation Requirements. To achieve good marks you must strictly adhere to the Documentation Requirements.

In presenting your work, you must document the test cases and test outputs for each feature required by the project. The test output can be in the form of terminal output. You may copy the text from the terminal output and paste it to your Word document (always include the command line). You should format the terminal output using a monospaced font so that it is clearly distinguishable from the rest of texts in your document. It is even better if you place the terminal output in a text box so that it is clearly separated from other texts in your document. But do not modify your test output unless it is too long and repetitive - heavy penalties will be applied if test output is modified to make your work appear better than what it really is!

Marking will be based on the number of working features demonstrated in your documentation. If a feature works and you have provided test evidence that the feature works in the form of test cases and test outputs, and your explanation of the test is convincing, you will get the marks allocated to the feature. Otherwise you will not get marks for that feature even if you have implemented it.

Read Documentation Requirements for further and more detailed requirements regarding the test documentation.

The project requires a significant amount of work on research, design and implementation. It is advised that you should start the project as soon as possible.

Deadline and Penalty for Late Submission (Back to Beginning)

The submission deadline of this assignment is specified in the Unit Information Page of the Unit LMS.

Assignments submitted on or before the deadline will be marked out of 100%.

Late submission of the assignment will have 10% of the group's raw mark deducted per day, unless an application for extension of submission deadline is granted. A submission that is late by more than 10 days or after the last teaching week (Week 14) will not be accepted.

Applications for extension of your assignment deadline can only be made via email to the Unit Coordinator, normally prior to the specified due date of the assignment. If an extension is granted (also by email), you must attach a copy of the email to your submission (see Documentation Requirements). Applications for extension by phone or in person do not count, even if granted. The above policy will be rigorously enforced.

Project 1: A Simple Unix Shell (Back to Beginning)

Design and implement a simple UNIX shell program using the grammar specified in the later part of this section. Please allow for at least 100 commands in a command line and at least 1000 arguments in each command.

In addition to the above, the following are required:

1. ***Reconfigurable shell prompt (default %)***

   The shell must have a shell built-in command `prompt` for changing the current prompt. For example, type the following command

   ```
   % prompt john$
   ```

   should change the shell prompt to `john$`, i.e., the second token of the command.

2. ***The shell built-in command `pwd`***

   This command prints the current directory (also known as *working directory*) of the shell process.

3. ***Directory walk***

   This command is similar to that provided by the Bash built-in command `cd`. In particular, typing the command without a path should set the current directory of the shell to the home directory of the user.

4. ***Wildcard characters***

   If a token contains wildcard characters `*` or `?` the token is treated as a filename. The wildcard characters in such a token indicate to the shell that the filename must be expanded. For example the command

   ```
   % ls *.c
   ```

   may be expanded to `ls ex1.c ex2.c ex3.c` if there are three matching files `ex1.c ex2.c ex3.c` in the current directory.

   You may implement this feature using the C function `glob`.

5. ***Standard input and output redirections > and <***

For example:

```
% ls -lt >  foo
```

would redirect the standard output of process `ls -lt` to file `foo`. Similarly in the following command,

```
% cat <  foo
```

the standard input of process `cat` is redirected to file `foo`.

6. ***Shell pipeline*** **|**

   For example:

   ```
   % ls -lt | more
   ```

   the standard output of process `ls -lt` is connected to the standard input of process `more` via a pipe.

7. ***Background job execution***

   For example:

   ```
   % xterm &
   ```

   The commannd line starts command `xterm` in the background (i.e., the shell will not wait for the process to terminate and you can type in the next command immediately). The following command line

   ```
   % sleep 20 &  ps -l
   ```

   starts command `sleep 20` and immediately execute command `ps -l` without waiting for command `sleep 20` to finish first.

8. ***Sequential job execution***

   For example the command line

   ```
   %  sleep 20 ; ps -l
   ```

   starts command `sleep 20` first, and wait for it to finish, then execute command `ps -l`.

9. ***The shell environment***

   The shell should inherit its environment from its parent process.

10. ***The shell built-in command*** `exit`

    Use the built-in command `exit` to terminate the shell program.

The behaviour of the above commands (except `prompt`) should be as close to those of the Bash shell as possible. In addition, your shell should not be terminated by CTRL-C, CTRL-\, or CTRL-Z.

Finally you must not use any existing shell program to implement your shell (for example by calling a shell through the function `system`). That would defeat the purpose of this project.

In the above, commands such as `ls`, `cat`, `grep`, `sleep`, `ps` and `xterm` are used as examples to illustrate the use of your shell program. However your shell must be able to handle *any*

command or executable program. Note the commands `prompt`, `pwd`, `cd` and `exit` should be implemented as shell builtins, not as external commands.

The syntax and behaviour of the built-in commands `pwd`, `cd` and `exit` should be similar to the corresponding commands under Bash shell.

A major part of this shell is a command line parser. Please read the this note for suggestions on implementing the parser.

Definition of Command Line Syntax

The following is the formal definition of the command line syntax for the shell, defined in Extended BNF:

```
< command line >      ::=   < job >
                          | < job > '&'
                          | < job > '&' < command line >
                          | < job > ';'
                          | < job > ';' < command line >

< job >               ::=   < command >
                          | < job > '|' < command >

< command >           ::=   < simple command >
                          | < simple command > '<' < filename >
                          | < simple command > '>' < filename >

< simple command >    ::=   < pathname >
                          | < simple command >  < token >
```

An informal definition plus additional explanations of the syntax is given below:

1. A ***command line*** consists of one or several ***jobs*** separated by the special character "&" and/or ";". The last ***job*** may be followed by the character "&" or ";". If a ***job*** is followed by the character "&", then it should be executed in the background.

2. A ***job*** consists of one or more ***command***s separated by pipeline characters "|";

3. A ***command*** is either a ***simple command*** or a ***simple command*** followed by an input redirection (< filename) or an output redirection (> filename);

4. A ***simple command*** consists of a single ***pathname*** followed by zero or more tokens;

5. The following five characters are the ***special characters***: &, ;, |, < , > ;

6. The ***white space characters*** are defined to be the space character and the tab character;

7. A ***token*** is either a special character or a string that does not contain space characters or special characters. In this project we do not consider quoted strings. Therefore if single quote or double quote characters appear in a string, they are treated just like any other non-special characters without its usually special meaning;

8. ***Tokens*** must be separated by one or more white spaces;

9. A ***pathname*** is either a file name, or an absolute pathname, or a relative pathname. Examples of pathnames are **grep**, **/usr/bin/grep**, **bin/grep** and **./grep**;

10. A command line must end with a newline character.

Project 2: A Simple File Transfer Protocol (Back to Beginning)

Design and implement a simple network protocol that can be used to download files from a remote site and to upload files to a remote site, and a client and a server programs that communicate using that protocol. The protocol should use TCP as its transport layer protocol. The server must be able to serve multiple client requests simultaneously. For simplicity, do not consider any authentication process in this project, hence the server will provide its service to any client with the right site address and port number.

To simplify project marking, please name your server program `myftpd` with the following command line syntax (here the letter `d` in `myftpd` stands for daemon, since the server should run as a daemon process:

```
myftpd [ initial_current_directory ]
```

The server process maintains a current directory. Its initial value should be the one inherited from its parent process unless the optional

```
initial_current_directory
```

is given. In the latter case, the user supplied path should be used to set the initial current directory of the server. This can be done using the function `chdir`. A client can use the `cd` command to change the (child) server's current directory later.

The client program should be named `myftp` with the following command line syntax:

```
myftp [ hostname | IP_address ]
```

where the optional *`hostname`* (*`IP_address`*) is the name (address) of the remote host that provides the `myftp` service. If the *`hostname`* or *`IP_address`* is omitted, the local host is assumed.

After the connection between the client and the server is established, the client should display a prompt `>` and wait for one of the following commands:

- `pwd` - to display the current directory of the server that is serving the client;
- `lpwd` - to display the current directory of the client;
- `dir` - to display the file names under the current directory of the server that is serving the client;
- `ldir` - to display the file names under the current directory of the client;
- `cd directory_pathname` - to change the current directory of the server that is serving the client; Must support "." and ".." notations.
- `lcd directory_pathname` - to change the current directory of the client; Must support "." and ".." notations.
- `get filename` - to download the named file from the current directory of the remote server and save it in the current directory of the client;
- `put filename` - to upload the named file from the current directory of the client to the current directory of the remove server.
- `quit` - to terminate the `myftp` session.

The `myftp` client should repeatedly display the prompt and wait for a command until the `quit` command is entered.

This project consists of three components:

1. ***Protocol Specification:***

   You must specify a protocol for communication between the `myftp` client and the `myftp` server. This protocol will become the *sole* reference to which the client program and the server program can be separately implemented. This means that, by referring to the protocol, the client and server can be implemented independent of each other. The design and implementation of the client (server) should not depend on (1) what strategy and algorithms were used to implement the server (client), (2) what programming language were used to implement the server (client), and (3) what operating system the server (client) is running on. The protocol must be complete, i.e., it contains all the necessary information required by both parties to communicate, such as the format and sequence of data exchanges between the client and the server, the transport layer protocol (TCP or UDP) used to deliver the data, and the server port number. However it should not contain anything that unnecessarily constrains the implementation of the client or the server. For example, there is no point to limit the implementation language to a particular programming language in the protocol, or to require the client or the server to be implemented with a particular data structure.

   Before you attempt this project, you should first complete the first two exercises in Lab 11. You may also read the Trivial FTP Protocol given in Appendix 17A of the textbook (Stallings: Chapter 17 of Online Chapters). Note TFTP uses UDP, while `myftp` protocol must use TCP.

2. ***Client Program:***

   You need to implement a client program according to the `myftp` protocol. Note that the client should never rely on any undocumented internal tricks in the server implementation. Apart from the `myftp` protocol, you should not make any other assumption about the server program in your client program.

3. ***Server Program:***

   You need to implement a server program according to the `myftp` protocol. Note that the server should never rely on any undocumented internal tricks in the client implementation. Apart from the `myftp` protocol, you should not make any other assumption about the client program in your server program.

Note the following additional requirements/explanations:

- The name of the client executable program must be `myftp`. The name of the server executable program must be `myftpd`.

- To avoid potential conflict in the use of the server's ``well known'' port number when several students use the ceto server, you should use the TCP port allocated to you as the default server listening port.

- Tests should show the cases where both the client and the server are on the same machine as well as on different machines (you may want to use ceto.murdoch.edu.au to do the testing, however ***you must kill your server at the end of the test***). They should also show that more than one client can obtain the service at the same time. You may use the command `script` to record the terminal I/O. When debugging your network program on a standalone machine (e.g., on your home Linux), you may use `localhost` as the ``remote'' host name.

- Tests should show that the program can transfer not only small files (e.g., 0, 10 and 100 bytes) but also large files (at least several mega bytes), and not only text files but also binary files. In addition, you must show that the transferred file and the original file are identical not only in their sizes but also in their contents (use `diff` command).

- The specification of the protocol is an important part of this question. Please provide a full specification of the protocol in a section separated from the client and server implementations. You should complete this specification before starting the implementation of the client and server programs.

- Please note that the server must be implemented as a daemon and must log all of its interactions with the clients.

Project 3: A Simple HTTP Client and a Simple HTTP Server (Back to Beginning)

This project involves an independent study of an existing protocol named HTTP (HyperText Transfer Protocol), version 1.1, and then design and implement a simple HTTP client that is able to communicate with *any* HTTP server, and design and implement a simple HTTP server that is able to provide simple services requested by *any* HTTP client.

Please note that a thorough understanding of the HTTP protocol is an important part of this project. Unless you are confident that you are able to gain a thorough understanding of the protocol, you should not select this project as your assignment.

This project has three components:

1. An essay of at least three pages long, explaining your understanding of the HTTP protocol. The essay should provide details of the protocol that are relevant to the project.
2. Design and implementation of an HTTP client program using C, see the detailed requirements below.
3. Design and implementation of an HTTP server program using C, see the detailed requirements below.

The HTTP Client

The client program must be able to request resources using the GET method from any HTTP server, such as Apache HTTP Server, Microsoft IIS and your own HTTP server. It must also be able to handle HEAD and TRACE methods. However it is not required to render HTML or any graphics or to manipulate HTML links. Content from an HTTP server can be displayed in the same format as you receive it on the terminal.

The client should be named `myhttp` with the following command line syntax:

```
myhttp [ -m  <method> ]  [ -a ]  <url>
```

The command takes an url and send an HTTP request message to the server and receives the response from the server. If there is no `-a` option, the program displays the content of the response. Otherwise it displays entire response message including both the headers and the content.

By default, the client uses GET method. But if the command line includes an `-m` option, the client should either use HEAD method or TRACE method: if the option is `-m head`, the client should use HEAD method; If the option is `-m trace`, the client should use TRACE method.

The HTTP Server

The server program must be able to perform the following tasks:

- serve regular files in response to GET requests from any HTTP client, such as Mozilla Firefox, Google Chrome, Microsoft Internet Explorer as well as your own HTTP client.
- handle directory requests. If the url points to a directory file and the url has a trailing slash, returns the index file in the directory (in the order of "index.html", "index.htm",

and "default.htm"). If the directory does not contain a recognized index file, serve the directory listing instead.
- support redirection when the url points to a directory but without trailing slash.
- disallow backtracking beyond the document root using ".." in url.
- handle HEAD method and TRACE method from any client.
- input a MIME type file containing a list of MIME type definition and handle the content accordingly.

In addition, your server should run as a daemon and it should be able to serve multiple client requests simultaneously. The server should be named `myhttpd` with the following command line syntax:

```
myhttpd [ -p  <port number> ]
        [ -d  <document root> ]
        [ -l  <log file> ]
        [ -m  <file for mime types> ]
        [ -f  <number of preforks> ]
```

where

- *`<port number>`* is an optional port number for the server. If the optional port number is not supplied, your server should use TCP port 8000.

  The default port for HTTP is TCP 80. However on Unix, only the root can use ports that are below 1024. To run your server with port 80, you must logon as the root (which is not possible in some machines such as ceto.murdoch.edu.au). This is the reason we use port 8000 instead of 80 as the default port.

  When testing your server on a shared machine such as ceto, you should avoid using port 8000, as there may be clashes if two students use the same port at the same time on that machine. To avoid potential conflict you should use the TCP port allocated to you if you test your server on ceto.

- *`<document root>`* is an optional directory path representing the document root of the server under which all files are stored. Without this option, you document root is taken as the current directory of the server.

  When testing your server, make sure that under your document root, there are regular files as well as directories. Some of these directories contain the index file `index.html`. Other directories do not contain this index file. Your should test both cases.

- *`<log file>`* is an optional path of the log file, to which the server will send its logs. Without this option, your log file should be `./myhttdp.log`, i.e., file `myhttpd.log` under the server's current directory.

  Your server must add one line into the log file for each client request, giving the date and time of the request, the HTTP method in the request, the originating host, the resource request and status code returned to the client.

- *`<file for mime types>`* is an optional file containing a list of mime types that the server recognises. Without this option, your server would only recognise the following mime types: `text/plain`, `text/html`, `image/jpeg`, and `image/gif`.

  In testing your server, you must at least test the above file types plus two additional file types.

- *`<number of preforks>`* is an optional number of preforks. Without this option, the number of preforks is fixed at 5.

Preforks mean that the server (parent server) will always maintain a pool of idle child processes (child servers), so that when a client request arrives, the parent server can immediately hand it to one of the idle child servers. In this way the client can get speedier service rather than waiting for the parent server to create a new child process before it can be served. Note that it takes a long time to create a new child process.

It is not a trivial task to design an efficient web server that can serve many client requests simultaneously and quickly. You may want to consider using `clone` systems call to create child processes to reduce the overhead. However if you choose to do so, you must be careful with the global variables as they are shared between the parent and all child processes!

Please note that the server `myhttpd` can take any combination of the above five command line options, not necessarily one option each time. For example, we may run the server with two options as shown below:

```
%  myhttpd  -p 12345  -l /tmp/hong.log
```

To complete this project you will have to carry out research on issues such as web server, HTTP protocol, MIME type, etc. There are numerous Internet sites providing the relevant materials. A good starting point is Wikipedia.

Documentation Requirements (Back to Beginning)

Your assignment must be submitted in the form of a single zip archive file (or a tar file) via the Unit LMS. The zip file must be named in the following format:

```
A2_<your first name>_<your family name>.zip
```

For example, the student, John Smith, would name his tar file as: `A2_John_Smith.zip`.

Your zip archive must contain the following files:

- One PDF document named "Assignment2.pdf". See the detailed requirements of this document later in this section.
- All your C/C++source code, makefiles used to compile source code and to build the executable programs, Linux executable programs, test files, test output files, etc, that are related to your project.

You can create test output files by cut-and-pasting from your terminal output into a text file. Alternatively you may use command `script` to record terminal input and output (but you need to remove the control characters). Please do not modify the test output.

You must make sure that your program can be compiled and run in other machines. Do not use any non-standard feature or any feature that is only available in your machine. Do not hard code any directory path into your program otherwise your program may not run on someone else's machine.

The file "Assignment2.pdf"

This PDF file should contain the documents in the following order:

1. ***ICT374 Project Declaration:***

   The ICT374 Project Declaration form must be completed and signed by all members and included in the Word file.

2. ***Extension Granted:***

If you have been granted an extension, include the email from your Unit Coordinator after the declaration form. The extension does not count unless it is documented here.

3. ***List of Files:***

   List the name and purpose of each file included in the tar archive.

4. ***The project title and a brief description of the project:***

   You can cut-and-paste the project question from this web page to your file.

5. ***Self diagnosis and evaluation:***

   Provide a statement giving

   a. a list of features that are fully functional.

   b. a list of features that are not fully functional (include features that are not fully implemented, or not implemented at all).

   This statement is important and essential. There will be a heavy penalty (>50%) if this statement is too vague, short of details and/or misleading.

6. ***Discussion of your solution:***

   In particular, you should emphasise any aspect of your solution that is novel or unusual, such as use of particular data structure or algorithm. You should also discuss technical choices available to you, the consideration behind your technical choice, and whether the result of your solution meets your expectation. You should highlight the strength and weakness of your solution and point out what improvement can be made to your solution.

7. ***Protocol (Project 2 and Project 3 only):***

   For Project 2 (A Simple File Transfer Protocol), you must include your protocol specification here. For Project 3 (A Simple HTTP Client and a Simple HTTP Server), you must include your essay explaining your understanding of the HTTP protocol here.

8. ***Test evidence:***

   When testing your program, you should turn on gcc warnings (`gcc -Wall ...`) and fix all warnings before submitting your program.

   You must provide sufficient number of test cases to demonstrate and convince your tutor that your project has met all requirements.

   For each test case, you should 1) explain the purpose of the test, 2) provide the test output including the command line used, and 3) give an explanation on what the test has achieved based on the test results.

   It is important that you present a sufficient number of test cases to convince your tutor that your solution works in all situations. Please note that although your tutor may test your program to verify the evidence presented in your documentation, it is not the responsibility of your tutor to test your program for the purpose of finding marks for you. It is up to individual student to mount a convincing case that the submitted solution works as required. You will be awarded marks based on the test evicence you have presented. Therefore if you do not provide test evidence you do not get marks regardless of whether your program works or not, or if the test evidence you have provided is not sufficient, your marks will be substantially reduced.

You should format your test output using fonts such as courier so that test output is clearly distinguishable from the rest of the document. You should not modify the test output unless it is too long and repetitive.

In presenting your test cases, do not use the tabular format that you may have been told to use in some other units. Simply present one test case after the other in a linear order.

9. ***Source code listing:***

   The source code must be properly indented, commented, and formatted. The source code included in this Word file must be identical to that of source code files included in the tar archive.

   You must also include the makefile for building your program.

Your tutor may mark your project directly on your Word file. The marked project and comments will be returned to you via Unit LMS.

The documentation requirements, as described above, will be strictly enforced. Your project will not be marked, or your marks will be significantly reduced, if you fail to adhere to the above requirements.

Policy on the Reuse of the Third Party Source Code (Back to Beginning)

Please read this section very carefully.

All students are encouraged to solve the problems independently with their own source code. I understand, however, that occasionally there may be justifiable reasons to re-use source code from a third party. Please note that if you have used one or more pieces of third party source code in your program (this includes the situations where you have made minor modifications to the third party source code), your project will be acceptable only if you have satisfied all of the following conditions:

- The third party source code is fully identified, including the page numbers and line numbers in your hard-copy project documentation, and
- The origin of the third party code is fully disclosed and acknowledged in your project submission, and
- The third party source code is fully commented in your program listing. All variables, functions and major control structures must be commented to show clearly that you understand the logic of the code, and
- The third party source code is less than 20% of your program (in terms of the number of lines), excluding the code specifically allowed to be used.

Failure to satisfy any one of the above conditions will result in 0 marks being awarded to your entire project.

The above policy will be rigorously enforced by the Unit Coordinator and Tutors.

Grievance with Assignment Marking (Back to Beginning)

Once you have received your marked assignment, if you have any grievance with the marking, you must raise it with your Unit Coordinator by email, within 7 days of returning of your marked assignment to the Unit LMS, or within the announced deadline in the Unit LMS, whichever is earlier. Otherwise the marks awarded to you will be final and will be used to calculate your final weighted average score for the unit.

Errata (Back to Beginning)

This page is subject to change based on errors found. Any errors found and corrected will be posted in the Unit Announcement page of the Unit LMS. If you print out a copy of this page, please follow the news in the Unit Announcements page for any updates.

Dr Hong Xie
Unit Coordinator for ICT374
Last update: Tuesday, 5 May 2020

Document author: Dr. H. Xie, Unit Coordinator, ICT374 Operating Systems and Systems Programming
Last Modified: Tuesday, 05-May-2020 17:48:15 AWST