COMP30023 Project 2
Email client

Out date: 26 April 2024
Due date: No later than **5pm on Thursday 16 May, 2024** AEST

Weight: 15% of the final mark

1 Project Overview

The aim of this project is to familiarize you with socket programming. Your task is to write a simple email client that downloads and parses email from a standards-compliant IMAP server.

Writing network code relies on conformance to standards, and so part of this project is to look up the relevant standards (Requests for Comments, RFCs). You may also refer to online tutorials; if you do, then please mention them in the comments of your code.

Your email client must be written in C or, **if you get prior permission**, in Rust. Submissions that do not compile and run on a cloud VM may receive zero marks.

Blue text in this document is hyperlinked to resource material.

2 Project Details

Your task is to design and code a simple email client.

2.1 Program execution / command line arguments

The program should be be launched using the command:

> `fetchmail`
> `-u` <*username*> `-p` <*password*> *[*`-f` <*folder*>*] [*`-n` <*messageNum*>*] [*`-t`*]*
> <*command*> <*server_name*>
>
> Where <*command*> may be one of: `retrieve`, `parse`, `mime`, or `list`.

The options beginning with "-" can occur in any order, including between and/or after the 2 positional arguments. The arguments in square brackets are optional.

All assessed output must be printed to `stdout`.
If you want to output debugging information, use `stderr`.

2.2 Logging on

Version 4rev1 of the IMAP protocol is specified in RFC 3501 and the below just describes one possible way that it can be used to implement this project.

The basic steps to retrieving an email are:

- Create an IPv6 socket (or fall back to IPv4 if IPv6 is not supported) on port 143 (or a TLS socket on port 993 if you are doing the extension task and `-t` was specified) to the host named on the command line.
- Log in to the IMAP server using:

tag `LOGIN` *username password*

where *username* and *password* are as specified on the command line.

Each command is prefixed with an identifier, or *tag* (such as `A01`). The tags should all be different within one session. These are used for the server to report errors or completion of commands.

Each line ends with the standard internet end-of-line sequence `\r\n` (not the standard C end-of-line).

If this succeeds, you will receive a string starting with the tag, followed by a single space, (a case-insensitive) `OK`, and another single space. For more details, refer to the `response`, `response-done`, and `response-tagged` rules of the formal syntax.

If it fails, you should print the string `Login failure\n` to `stdout` and exit with status 3.

You may assume that the `LOGINDISABLED` capability will not be advertised, and do not need to support RFC5738: IMAP Support for UTF-8.

- Tell the system which folder you want to read from:

  tag `SELECT` *folder*

  If this succeeds, you will receive some untagged response lines starting with "`*`", followed by a string starting with the tag followed by `OK` .

  If the folder doesn't exist, print the string `Folder not found\n` to `stdout` and exit with status 3.

  You should read from the folder `INBOX` if no folder is specified on the command line.

The next steps depend on which command is being executed.

2.3 Retrieve

If the command is `retrieve`, fetch the email, such as with the command:

tag `FETCH` *messageNum* `BODY.PEEK[]`

If `messageNum` (the message sequence number) was not specified on the command line, then fetch the **last added** message in the folder (Hint: Take a look at the `seq-number` rule in the formal syntax).

If this succeeds, the response will be the contents of the email, including the headers.

For more details, refer to the `response`, `message-data`, and `msg-att-static` rules of the formal syntax.

If message with sequence number *messageNum* does not exist, then print the string `Message not found\n` to `stdout` and exit with status 3 (apply this to `parse` and `mime` also).

Once you have retrieved the email, print the raw email to `stdout` and exit with status 0.

2.4 Parse

If the command is `parse`, print the following `\n`-delimited output to `stdout` for the fetched email:

From: *mailbox-list*

To: *address-list*

Date: *date-time*

Subject: *unstructured*

You may get this information from the raw email, or by issuing more specific IMAP commands such as `BODY.PEEK[HEADER.FIELDS (FROM)]` or `ENVELOPE` that explicitly fetch header fields.

Note that a header field body may be split over multiple lines using RFC 5322's "folding" syntax. You must unfold such lines before printing, so that each header is printed as a single line.

Not all emails have a `Subject` header; if it does not, then respond as if the subject were "`<No subject>`". Not all emails have a `To` header; if not, print an empty string in place of the value (with no space after colon).

Note that header field names are case insensitive.

Your mail client does not need to support RFC 6532: Internationalized Email Headers, or Obsolete Header Field Syntax (but should not crash, if they are encountered; code should never crash).

Note that a line in the body can look just like a header line. How can you tell them apart?

2.5 MIME

If the command is `mime`, then the program should decode Multimedia Internet Mail Extension (MIME) messages as follows, and print the plain ASCII version of the message.

MIME encoding (RFC 2045, RFC 2046) is used to encode attachments, and also to allow both an HTML and plain text version of an email. In this project, it is introduced by headers of the form:

```
MIME-Version:  1.0
Content-type:  multipart/alternative; boundary="boundary parameter value"
```

Your program is expected to match at the top level of a message:

- The `MIME-Version` header, with value of `1.0` (There will be no "comment strings".)
- The `Content-Type` header, with `multipart/alternative` media type, and `boundary` parameter

The body of the email will then contain body parts separated by boundary delimiter lines like:

CRLF --boundary-parameter-value CRLF

where the *boundary parameter value* matches the value in the `Content-Type` header of the top-level message.

The final block is ended with a line:

CRLF --boundary-parameter-value--

The standard allows quite a general syntax for body parts, such as a range of encodings. You are not expected to handle all these options.

Your program is expected to match and print the first `UTF-8` `text/plain` body part:

- `Content-Type:  text/plain`, with `charset=UTF-8`
- `Content-Transfer-Encoding:  quoted-printable | 7bit | 8bit`

Note that headers may be folded, and parameter (`boundary` and `charset`) values may be quoted with double quotes. The actual values do not contain any quotes, so the values can be obtained by simply ignoring any double quote characters on the line.

Note that header field names, parameter names, charset parameter value, Content-Transfer-Encoding mechanism, Content-Type media type and subtype are *case-insensitive*.

The RFC allows multipart entities to be nested, but that is *not* required by this project.

If the `mime` command is given, the output should consist only of the body area of the the first `utf-8` `text/plain` body part that was matched, with no further decoding.

If matching fails, your program should print a sensible error message and exit with status 4.

2.6 List

If the command is `list`, then print to `stdout` the subject lines of all the emails in the specified folder, sorted by message sequence number. If the mailbox is empty, print nothing and exit with status 0.

The subject lines should be unfolded so that each is a single line, but does not need be decoded in any other way.

Output the message sequence number, followed by a colon and a space, followed by the body of the subject header (*not* the initial `Subject:` or any additional leading or trailing whitespace). For example:

```
1: Recent Canvas Notifications
2: New in COMP30023: Mst Practise Exam
3: <No subject>
```

You can use the SEARCH command from RFC 3501 section 6.4.4, or the FETCH command from RFC 3501 section 6.4.5 with the range `1:*`, or any other method you choose.

You may assume that the contents of the selected mailbox will not be changed during this `list` operation.

2.7 Stretch Goal – TLS using OpenSSL

For the stretch goal marks (included in the 15 marks), use transport layer security (TLS) instead of a plain TCP socket when the `-t` option is specified on the command line.

This task is only for those aiming to get 15/15 for the assignment. If you think the assignment is too big, then focus on the non-stretch goals.

If setting up the connection fails, provide meaningful output to `stdout` and then exit with status 2.

If implementing in C, you may find the following OpenSSL functions helpful.

```
SSL_library_init                    SSL_new
SSL_load_error_strings              SSL_set_mode
OpenSSL_add_all_algorithms          SSL_CTX_free
SSL_CTX_new                         BIO_*
```

These are explained at [https://www.openssl.org/docs/manmaster/man3].

You may also find the instructions at [https://wiki.openssl.org/index.php/SSL/TLS_Client] useful. Make sure that you understand (and document) any code that you use.

Note that the test server will have a self-signed certificate, and so you must add the root certificate to your trust store.

```
openssl s_client -showcerts -connect <host>:993 <<< "Q"
# Manually save the root certificate to a file named ca.crt

sudo cp ca.crt /usr/local/share/ca-certificates/
sudo update-ca-certificates
```

If you implement this extension, you can test your implementation of `fetchmail` against the IMAP server of an actual email provider. It is suggested that you create a disposable account for this purpose.

For Gmail, You will need to get a Google app password, following the instructions at [https://support.google.com/accounts/answer/185833]:

- Log in to the Google Account.
- Select '2-Step Verification' from Security.
- After completing 2-Step Verification, select 'App passwords' at the bottom of the page.
- After generating, get a 16-character code that you can use at the authentication step.

3 Marking Criteria

The marks are broken down as follows:

Task # and description	Marks
1. Correctly requests email message	2
2. Displays raw email (`retrieve`)	2
3. Displays headers (`parse`)	2
4. Displays `text/plain` version of the email (`mime`)	2
5. Lists email (`list`)	2
6. Safety	2
7. Build quality	1
8. Quality of software practices	1
Stretch goal. Transport layer security (TLS)	1

Code that does not compile and run on cloud VM will usually be awarded zero marks for parts 1–6. Use the Git CI infrastructure to ensure your submission is valid. Your submission will be tested and marked with the following criteria:

Task 1. Correctly requests email message Your code correctly establishes a connection, navigates to the specified folder and makes *at least one* of the requests (retrieve, parse, mime or list).

Task 2. Displays raw email Your code downloads the complete message, which may require multiple `read()` calls. This should not result in any memory overflow problems, even if a large message is returned, by using dynamic memory allocation (`malloc`). If dynamic memory allocation fails, the code can print an error message and exit.

Task 3. Displays headers Your code correctly identifies the appropriate headers and displays them in the correct order. For this task, you may assume that the listed headers will follow the limits defined in RFC 5322.

Task 4. Displays `text/plain` version of the email Your code correctly parses any MIME headers and displays only the first `text/plain` version of the email.

Task 5. Displays list of email Your code correctly displays the list of email which is present in the selected mailbox.

Task 6. Safety Your code should be robust to things like invalid command line arguments, failed connections, improperly formatted mail.

Your code should not allow injection of IMAP commands through malicious command line arguments or email contents.

Network code should *never* crash, even if the command line is invalid or the hosts on the other side behave poorly. The sorts of bugs that cause involuntary termination, such as segmentation faults (memory errors) also introduce security vulnerabilities. You must not implement any handlers for program error signals (such as `SIGSEGV`). It is OK to print an error message and abort the program.

If interaction with the server fails at any point, an informative error message should be displayed. The program may either attempt to continue, or exit cleanly.

Task 6 covers segmentation faults, but code that crashes with a segmentation fault may be marked down in other tasks too.

For the purpose of assessment, use the following predefined status codes:

1. Command-line parsing error, argument validation failure (if implemented)
2. Connection initiation errors
3. Unexpected IMAP or server responses (e.g. unexpected disconnect, random characters)
4. Parse failure (e.g. Expected header is missing (optional), `text/plain` part not found)
5. Other errors

You do not need to apply a timeout; if the server becomes unresponsive, your code is allowed to hang.

Task 7. Build quality

- The repository must contain a `Makefile` that produces an executable named "`fetchmail`", along with all source files required for compilation. Place the `Makefile` at the root of your repository, and ensure that running `make` places the executable there too.
- Make sure that all source code is committed and pushed.
- `make clean && make -B && ./fetchmail <...arguments>` should execute the submission.
- Compiling using "`-Wall`" should yield no warnings (C).
  Compiling using "`cargo build`" should yield no warnings (Rust).
  Do not suppress any default warnings inline.

- Running `make clean` should remove all object code and executables.
- Do not commit `fetchmail` or other executable files. Scripts (with `.sh` extension) are exempted.

The mark calculated for "Build quality" will be visible on CI.

If this fails for any reason, you will be told the reason, and be allowed to resubmit (with the usual late penalty). If it still fails, you will get 0 for Tasks 1–6 and the stretch goal. Test this by committing regularly, and checking the CI feedback. (If you need help, ask on the forum.)

Task 8. Quality of software practices Factors considered include:

- **Proper use of version control**, based on the regularity of commit and push events, their content and associated commit messages (e.g., repositories with a single commit and/or non-informative commit messages will lose 0.5 marks).
- **Quality of code**, based on the choice of variable names, comments, formatting (e.g. consistent indentation and spacing), and structure (e.g. abstraction, modularity).
- **Proper memory management**, based on the *absence* of memory errors and memory leaks.

4 Submission

All code must be written in C or Rust (e.g., it should not be a C wrapper over code in another language). You may use the standard library of the chosen programming language.

You *must not* use or adapt any code or libraries relating to IMAP or Internet Messages.

You may link with the `openssl` library present on the system, or use any of the `libc`, `openssl`, `openssl-sys`, and `openssl-probe` crates *for the purpose of implementing TLS or (optionally) SASL Authentication only*. `openssl` must be dynamically linked. The `openssl-src` crate must not be included as a sub-dependency.

For Rust submissions, any invocation of `cargo build` must include the `--frozen`, `--offline`, and `--release` options. A `vendor` rule must be defined in your `Makefile`, to vendor crates prior to offline compilation. The `vendor` directory must be present in `.gitignore`, and must not be committed at any stage of the `git` history. The `vendor` directory must not be removed by `make clean`. `Cargo.lock` files must be committed. Unless otherwise approved, your crate should not rely on any custom build scripts. You may assume that the latest stable version of Rust will be installed in the build environment.

You can reuse the code that *you wrote* for your other *individual* projects if you clearly specify when and for what purpose you have written it (e.g., the code and the name of the subject, project description and the date, that can be verified if needed) in `references.txt`.

If you import code from somewhere else, within the collaboration policy, there should be a commit that does nothing but import that code, with a commit message saying "importing code from [reference]". You should then customise the imported code in later commits.

GitHub The use of `GitHub` is mandatory. Your submission will be assessed based using the code in your Project 2 repository (`proj2-⟨usernames...⟩`) under the subject's organization.

We strongly encourage you to commit your code at least once per day. Be sure to `push` after you `commit`. This is important not only to maintain a backup of your code, but also because the git history may be considered for matters such as special consideration, extensions and potential plagiarism. Proper use of `git` will have a positive effect on the mark you get for quality of software practices.

Submission To submit your project, please follow these steps carefully:

1. Push your code to the repository named `proj2-⟨usernames...⟩` under the subject's organization, https://github.com/feit-comp30023-2024.

   Executable files (that is, all files with the executable bit that are in your repository other than `.sh` files) **will be removed** before marking. Hence, ensure that none of your source files have the executable bit.

Ensure your code **compiles and runs on the provided VMs**. Code that does not compile or produce correct output on VMs will typically receive very low or 0 marks.

2. **Submit the full 40-digit SHA1 hash** of the commit you want us to mark to the **Project 2 Assignment** on the LMS.

   You are allowed to update your chosen commit by resubmitting the LMS assignment as many times as desired. However, only the last commit hash submitted to the LMS before the deadline (or approved extension) will be marked without a late penalty.

3. Ensure that the commit that you submitted to the LMS is correct and accessible from a fresh clone of your repository. An example of how to do this is as follows:

   ```
   git clone git@github.com:feit-comp30023-2024/proj2-<usernames...> proj2
   cd proj2 && git checkout <commit-hash-submitted-to-lms>
   ```

   Please be aware that we will only mark the commit submitted via the LMS. It is your responsibility to ensure that the submission is correct and corresponds to the commit you want us to mark.

Late submissions Late submissions will incur a deduction of 2 marks per day (or part thereof). For example, a submission made 1 hour after the deadline is considered to be 1 day late and carries a deduction of 2 marks.

We strongly encourage you to allow sufficient time to follow the submission process outlined above. Leaving it to the last minute usually results in a submission that is a few minutes to a few hours late, or in the submission of the incorrect commit hash. Either case leads to late penalties.

The submission date is determined **solely** by the date in which the LMS assignment was submitted. Forgetting to submit via the LMS or submitting the wrong commit hash will result in a late penalty that will apply regardless of the commit date.

We will not give partial marks or allow code edits for either known or hidden cases without applying a late penalty (calculated from the deadline).

Extension policy: If you believe you have a valid reason to require an extension, please fill in the Project 2 extension request form available on the LMS *at the earliest opportunity*, which in most instances should be well before the submission deadline. Extensions **will not be** considered otherwise. Requests for extensions are not automatically granted and are considered on a case-by-case basis. You are required to submit supporting evidence such as a medical certificate. In addition, your `git` history should illustrate the progress made on the project up to the date of your request.

If you have a chronic condition or an AAP, please *complete an extension request early*, even if you hope not to need it. This will allow us to spend more time helping people near the deadline instead of doing paperwork. If you need special consideration, an extension may not be the best way to help you, especially if you would need a long extension. When you apply, please think of other things we could do to help you to submit on time.

5 Testing

You will have access to several test cases (via an IMAP server – see Ed) and their expected outputs. However, these test cases are *far from exhaustive*; they are mainly to avoid misinterpretation of the specification. Designing and running your own tests is a part of this project. Your code will be assessed on these cases other cases that you haven't seen before. The unseen cases are not "trick" cases, but are chosen to reflect the fact that real world programming tasks do not come with an exhaustive list of test cases.

Project 2 Repository: The project skeleton and sample outputs are available from:
`feit-comp30023-2024/project2`.

Continuous Integration Testing: To provide you with feedback on your progress before the deadline, we will set up a Continuous Integration (CI) pipeline on `GitHub` with the same set of test cases.

Though you are strongly encouraged to use this service, the usage of CI is not assessed, i.e., we do not require CI tasks to complete for a submission to be considered for marking.

The requisite `ci.yml` file has been provisioned and placed in your repository, but is also available from the `.github/workflows` directory of the `project2` repository linked above.

6 Team Work

Both team members are expected to contribute equally to the project. If this is not the case, please approach the head tutor or lecturer to discuss your situation. In cases in which a student's contribution is deemed inadequate, the student's mark for the project will be adjusted to reflect their lack of contribution. We will look at git history when making such an assessment.

7 Getting help

Please see Project 2 Help module on LMS.

8 Collaboration and Plagiarism

You may discuss this project abstractly with your classmates but what gets typed into your program must be the work of your group, not copied from anyone else. Do **not** share your code and do **not** ask others to give you their programs. The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, point out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See https://academicintegrity.unimelb.edu.au for more information.

Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. You should *not* post your code to any public location (e.g., GitHub) until final subject marks are released.

If you use a **small** amount of code not written by you, you must attribute that code to the source you got it from (e.g., a book or Stack Exchange) in both the comments and the git commit messages.

Do **not** post your code on the subject's discussion board Ed, except in a **Private** thread.

Plagiarism policy: You are reminded that all submitted project work in this subject is to be your own work, as a pair or individual. Automated similarity checking software will be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the pair or student concerned.

Using git properly is an important step in the verification of authorship. We should see the stages of your code being written, not just the finished product.

AI software such as ChatGPT can generate code, but it will not earn you marks. You are allowed to use tools like ChatGPT, but if you do then you must strictly adhere to the following rules.

1. Have a file called AI.txt
2. That file must state the query you gave to the AI, and the response it gave
3. You will only be marked on the *differences* between your final submission and the AI output.

   If the AI has built you something that gains you points for Task 1, then you will not get points for Task 1; the AI will get all those points.

   If the AI has built you something that gains no marks by itself, but you only need to modify five lines to get something that works, then you will get credit for identifying and modifying those five lines.
4. If you ask a generic question like "How do I convert an integer to network byte order?" or "What does the error 'implicit declaration of function parse_command_line' mean?" then you will not lose any marks for using its answer, but please report it in your AI.txt file.

If these rules seem too strict, then do not use the AI tools.

These issues are new, and this may not be the best policy, but it is this year's policy. If you have suggestions for better rules for *future* years, please mention them on the forum.

Good luck!