Assignment 1: Pushy

version: 1.0 last updated: 2024-03-06 12:00

Aims

This assignment aims to give you

- practice in Shell programming generally
- a clear concrete understanding of Git's core semantics

Note: the material in the lecture notes will not be sufficient by itself to allow you to complete this assignment. You may need to search on-line documentation for Shell, Git, etc. Being able to search documentation efficiently for the information you need is a *very* useful skill for any kind of computing work.

Introduction

You are going to implement **Pushy**, a simple but powerful subset of the version control system Git.

Your task in this assignment is to write 10 shell scripts named `pushy-init` `pushy-add` `pushy-commit` `pushy-log` `pushy-show` `pushy-rm` `pushy-status` `pushy-branch` `pushy-checkout` `pushy-merge` .

Each of these script implements a simplified version of the corresponding Git command.

Git is a *very* complex program that has many individual commands. You only have to implement simplified equivalents of some core commands.

You are given a number of simplifying assumptions which make your task much easier.

Interestingly, large parts of early versions of Git were implemented in Shell.

Reference implementation

Many aspects of this assignment are not fully specified in this document; instead, you must match the behaviour of a reference implementation.

For example, your script `pushy-add` should match the behaviour of `2041 pushy-add` exactly, including producing the same error messages.

Provision of a reference implementation is a common method to provide or define an operational specification, and it's something you will likely need to do after you leave UNSW.

Discovering and matching the reference implementation's behaviour is deliberately part of the assignment.

While the code in the reference implementation is fairly straightforward, reverse-engineering its behaviour is obviously not so simple, and is a nice example of how coming to grips with the precise semantics of an apparently obvious task can still be challenging.

If you discover what you believe to be a bug in the reference implementation, report it in the class forum. We may fix the bug, or indicate that you do not need to match the reference implementation's behaviour in this case.

Pushy Commands

Subset 0

Subset 0 commands must be implemented in POSIX-compatible Shell.

See the **Permitted Languages** section for more information.

pushy-init

The `pushy-init` command creates an empty Pushy repository.

`pushy-init` should create a directory named `.pushy` , which it will use to store the repository.
It should produce an error message if this directory already exists, or cannot be created.

You should match this, and other error messages exactly. For example:

```
$ ls -d .pushy
ls: cannot access '.pushy': No such file or directory
$ ./pushy-init
Initialized empty pushy repository in .pushy
$ ls -d .pushy
.pushy
$ ./pushy-init
./pushy-init: error: .pushy already exists
```

`pushy-init` may create initial files or directories inside `.pushy` .

You do not have to use a particular representation to store the repository.

You do not have to (and should not) create the same files and directories inside `.pushy` as the reference implementation.
You can create whatever files or directories inside `.pushy` you wish.

Do not store information outside `.pushy`

pushy-add *filenames...*

The `pushy-add` command adds the contents of one or more files to the "**index**".

Files are added to the repository in a two-step process. The first step is adding them to the index.

You will need to store files in the index somehow in the `.pushy` sub-directory.
For example, you might choose store them in a sub-directory of `.pushy`.

Only ordinary files in the current directory can be added.
You can assume filenames start with an alphanumeric character (`[a-zA-Z0-9]`) and will only contain alpha-numeric characters, plus `.`, `-` and `_` characters.

The `pushy-add` command, and other Pushy commands, will not be given pathnames with slashes.

pushy-commit -m *message*

The `pushy-commit` command saves a copy of all files in the index to the repository.

A message describing the commit must be included as part of the commit command.

Pushy commits are numbered sequentially: they are not hashes, like Git. You must match the numbering scheme.

You can assume the commit message is ASCII, does not contain new-line characters, and does not start with a `-` character.

pushy-log

The `pushy-log` command prints a line for every commit made to the repository.
Each line should contain the commit number and the commit message.

pushy-show *[commit]:filename*

The `pushy-show` should print the contents of the specified *filename* as of the specified *commit*.
If *commit* is omitted, the contents of the file in the index should be printed.

You can assume the commit, if specified, will be a non-negative integer.

Subset 0 examples

```
$ ./pushy-init
Initialized empty pushy repository in .pushy
$ echo line 1 > a
$ echo hello world >b
$ ./pushy-add a b
$ ./pushy-commit -m 'first commit'
Committed as commit 0
$ echo  line 2 >>a
$ ./pushy-add a
$ ./pushy-commit -m 'second commit'
Committed as commit 1
$ ./pushy-log
1 second commit
0 first commit
$ echo line 3 >>a
$ ./pushy-add a
$ echo line 4 >>a
$ ./pushy-show 0:a
line 1
$ ./pushy-show 1:a
line 1
```

Subset 1

Subset 1 is more difficult. You will need to spend some time understanding the semantics (meaning) of these operations, by running the reference implementation, or researching the equivalent Git operations.

Note the assessment scheme recognises this difficulty.

Subset 1 commands must be implemented in POSIX-compatible Shell.

See the **Permitted Languages** section for more information.

pushy-commit *[-a]* -m *message*

`pushy-commit` can now have a `-a` option,
which causes all files already in the index to have their contents from the current directory added to the index before the commit.

pushy-rm *[--force] [--cached] filenames...*

`pushy-rm` removes a file from the index, or, from the current directory and the index.

If the `--cached` option is specified, the file is removed only from the index, and not from the current directory.

`pushy-rm`, like `git rm`, should stop the user accidentally losing work, and should give an error message instead if the removal would cause the user to lose work. You will need to experiment with the reference implementation to discover these error messages. Researching `git rm`'s behaviour may also help.

The `--force` option overrides this, and will carry out the removal even if the user will lose work.

pushy-status

`pushy-status` shows the status of files in the current directory, the index, and the repository.

There are many different cases to consider for `pushy-status`.
You will need to experiment with the reference implementation to find them all.

Subset 1 examples

```
$ ./pushy-init
Initialized empty pushy repository in .pushy
$ touch a b c d e f g h
$ ./pushy-add a b c d e f
$ ./pushy-commit -m 'first commit'
Committed as commit 0
$ echo hello >a
$ echo hello >b
$ ./pushy-commit -a -m 'second commit'
Committed as commit 1
$ echo world >>a
$ echo world >>b
$ echo hello world >c
$ ./pushy-add a
$ echo world >>b
$ rm d
$ ./pushy-rm e
$ ./pushy-add g
$ ./pushy-status
a - file changed, changes staged for commit
b - file changed, changes not staged for commit
```

Subset 2

Subset 2 is extremely difficult. You will need to spend considerable time understanding the semantics of these operations, by running the reference implementation, and/or researching the equivalent Git operations.

Note the assessment scheme recognises this difficulty.

Subset 2 commands must be implemented in POSIX-compatible Shell.

See the **Permitted Languages** section for more information.

pushy-branch *[-d] [branch-name]*

`pushy-branch` either creates a branch, deletes a branch, or lists current branch names.

If *branch-name* is omitted, the names of all branches are listed.

If *branch-name* is specified, then a branch with that name is created or deleted,
depending on whether the `-d` option is specified.

pushy-checkout *branch-name*

`pushy-checkout` switches branches.

Note that, unlike Git, you can not specify a commit or a file: you can only specify a branch.

pushy-merge (*branch-name*|*commit-number*) -m *message*

`pushy-merge` adds the changes that have been made to the specified branch or commit to the index, and commits them.

Subset 2 examples

```
$ ./pushy-init
Initialized empty pushy repository in .pushy
$ seq 1 7 >7.txt
$ ./pushy-add 7.txt
$ ./pushy-commit -m commit-1
Committed as commit 0
$ ./pushy-branch b1
$ ./pushy-checkout b1
Switched to branch 'b1'
$ sed -Ei 's/2/42/' 7.txt
$ cat 7.txt
1
42
3
4
5
6
7
$ ./pushy-commit -a -m commit-2
Committed as commit 1
$ ./pushy-checkout master
```

If a file has been changed in both branches `pushy-merge` produces an error message.

Note: if a file has been changed in both branches `git` examines which lines have been changed and combines the changes if possible. Pushy doe not do this, for example:

```
$ ./pushy-init
Initialized empty pushy repository in .pushy
$ seq 1 7 >7.txt
$ ./pushy-add 7.txt
$ ./pushy-commit -m commit-1
Committed as commit 0
$ ./pushy-branch b1
$ ./pushy-checkout b1
Switched to branch 'b1'
$ sed -Ei 's/2/42/' 7.txt
$ cat 7.txt
1
42
3
4
5
6
7
$ ./pushy-commit -a -m commit-2
Committed as commit 1
$ ./pushy-checkout master
```

Testing

Autotests

As usual, some autotests will be available:

```
$ 2041 autotest pushy pushy-*
...
```

You can also run only tests for a particular subset or an individual test:

```
$ 2041 autotest pushy subset1 pushy-*
...
$ 2041 autotest pushy subset1_13 pushy-*
...
```

If you are using extra Shell files, include them on the autotest command line.

Autotest and automarking will run your scripts with a current working directory different to the directory containing the script. The directory containing your submission will be in `$PATH` .

You will need to do most of the testing yourself.

Test Scripts

You should submit ten Shell scripts, named `test00.sh` to `test09.sh`, which run pushy commands that test an aspect of Pushy.

The `test??.sh` scripts do not have to be examples that your program implements successfully.

You may share your test examples with your friends, but the ones you submit must be your own creation.

The test scripts should show how you've thought about testing carefully.

You are only expected to write test scripts testing parts of Pushy you have attempted to implement. For example, if you have not attempted subset 2 you are not expected to write test scripts testing pushy-merge .

Permitted Languages

Your programs must be written entirely in POSIX-compatible shell.

Your programs will be run with *dash*, in `/bin/dash`. You can assume anything that works with the version of `/bin/dash` on CSE systems is POSIX compatible.

Start your programs with:

```
#!/bin/dash
```

If you want to run these scripts on your own machine — for example, one running macOS — which has *dash* installed somewhere other than `/bin`, use:

```
#!/usr/bin/env dash
```

You are permitted to use any feature `/bin/dash` provides.

On CSE systems, `/bin/sh` is the Bash (Bourne-again shell) shell: `/bin/sh` is a symlink to `/bin/bash`. Bash implements many non-POSIX extensions, including regular expressions and arrays. These *will not work* with `/bin/dash`, and you are not permitted to use these for the assignment.

You are not permitted to use Perl, Python or any language other than POSIX-compatible shell.

You are permitted to use only these external programs:

basename
bunzip2
bzcat
bzip2
cat
chmod
cmp
combine
cp
cpio
csplit
cut
date
dc
dd
df
diff
dirname
du
echo
egrep
env
expand
expr
false
fgrep
find
fold
getopt
getopts
grep
gunzip
gzip
head
hostname
ifne
less
ln
ls
lzcat
lzma
md5sum
mkdir
mktemp
more
mv
nl
patch
printf
pwd
readlink
realpath
rev
rm
rmdir
sed
seq
sha1sum
sha256sum
sha512sum
sleep
sort
sponge
stat
strings
tac
tail
tar
tee
test
time
top
touch
tr
true
uname
uncompress
unexpand
uniq
unlzma
unxz
unzip
wc
wget
which
who
xargs
xz
xzcat
yes
zcat

Only a few of the programs in the above list are likely to be useful for the assignment.

Note you are permitted to use built-in shell features including: `cd` , `exit` , `for` , `if` , `read` , `shift` and `while` .

If you wish to use an external program which is not in the above list, please ask in the class forum for it to be added.

You may submit extra shell files.

Assumptions/Clarifications

Like all good programmers, you should make as few assumptions as possible.

You can assume `pushy` commands are always run in the same directory as the repository, and only files from that directory are added to the repository.

You can assume the directory in which `pushy` commands are run will not contain sub-directories apart from `.pushy` .

You can assume where a branch name is expected a string will be supplied starting with an alphanumeric character ([a-zA-Z0-9]), and only containing alphanumeric characters plus '-' and '_'. In addition a branch name will not be supplied which is entirely numeric. This allows brnach names to be distinguished from commits when merging.

You can assume where a filename is expected a string will be supplied starting with an alphanumeric character ([a-zA-Z0-9]) and only containing alphanumeric characters, plus '.', '-' and '_' characters.

You can assume where a commit number is expected a string will be supplied which is a non-negative integer with no leading zeros. It will not contain white space or any other charcters except digits.

You can assume that `pushy-add` , `pushy-show` , and `pushy-rm` will be given just a filename, not pathnames with slashes.

You do not have to consider file permissions or other file metadata. For example, you do not have to ensure files created by a checkout command have the same permissions as when they were added.

You do not have to handle concurrency. You can assume only one instance of any `pushy` command is running at any time.

You can assume that only the arguments described above are supplied to `pushy` commands. You do not have to handle other arguments.

You should match the output streams used by the reference implementations. It writes error messages to stderr: so should you.

You should match the exit status used by the reference implementation. It exits with status 1 after an error: so should you.

You can assume the directory containing your scripts is in `$PATH` .

You can not assume the directory containing your scripts is the same as the repo.

Your scripts are always run in the directory containing the repository.
Autotests and automarking will put your scripts (pushy-init, pushy-add, ...) in a different directory to the repository. This may break scripts which run or source other scripts or file and assume they are in the current directory. Autotests and automarking will add the directory containing your scripts to `$PATH` . This allows you to access other scripts or files by just specifying their name. For example: `. library_functions.sh` will source the shell commands in `. library_functions.sh` even though it is another directory, because the directory has been added to `$PATH` .

Note running `. ./library_functions.sh` will break during autotests and automarking. There can be subtle problem related to directories, ask for help in the forum.

You can assume arguments will be in the position and order shown in the usage message from the reference implementation. Other orders and positions will not be tested. For example, here is the usage message for `pushy-rm` :

```
$ 2041 pushy-rm
usage: pushy-rm [--force] [--cached] <filenames>
```

So, you assume that if the `--force` or `--cached` options are present, they come before all filenames, and if they are both present the `--force` option will come first.

Pushy error messages include the program name. It is recommended you use `$0` however it is also acceptable to hard-code the program name. The automarking and style marking will accept both.

Do not use the modification time of a file to determine whether it has changed. You must use the file contents.

Change Log

Version 1.0
(2024-03-06 12:00)

- Initial release

Assessment

Testing

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest pushy
```

`2041 autotest` will not test everything.
Always do your own testing.

Automarking will be run by the lecturer after the submission deadline, using a superset of tests to those `autotest` runs for you.

Submission

When you are finished working on the assignment, you must submit your work by running `give` :

```
$ give cs2041 ass1_pushy pushy-* test??.sh [any-other-files]
```

You must run `give` before **Week 7 Monday 11:59:59 2024 (midday)** to obtain the marks for this assignment. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

You can run `give` multiple times.
Only your last submission will be marked.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

You *cannot* obtain marks by emailing your code to tutors or lecturers.

You can check your latest submission on CSE servers with:

```
$ 2041 classrun check ass1_pushy
```

You can check the files you have submitted here.

Manual marking will be done by your tutor, who will mark for style and readability, as described in the **Assessment** section below. After your tutor has assessed your work, you can view your results here; The resulting mark will also be available via give's web interface.

Due Date

This assignment is due **Week 7 Monday 11:59:59 2024 (midday)** (2024-03-25 11:59:00).

The UNSW standard late penalty for assessment is 5% per day for 5 days - this is implemented hourly for this assignment.

Your assignment mark will be reduced by 0.2% for each hour (or part thereof) late past the submission deadline.

For example, if an assignment worth 60% was submitted half an hour late, it would be awarded 59.8%, whereas if it was submitted past 10 hours late, it would be awarded 57.8%.

Beware - submissions 5 or more days late will receive zero marks. This again is the UNSW standard assessment policy.

Assessment Scheme

This assignment will contribute 15 marks to your final COMP(2041|9044) mark

15% of the marks for assignment 1 will come from hand-marking. These marks will be awarded on the basis of clarity, commenting, elegance and style: in other words, you will be assessed on how easy it is for a human to read and understand your program.

5% of the marks for assignment 1 will be based on the test suite you submit.

80% of the marks for assignment 1 will come from the performance of your code on a large series of tests.

An indicative assessment scheme follows. The lecturer may vary the assessment scheme after inspecting the assignment submissions, but it is likely to be broadly similar to the following:

HD (85+)	All subsets working; code is beautiful; great test suite
DN (75+)	Subset 1 working; good clear code; good test suite
CR (65+)	Subset 0 working; good clear code; good test suite
PS (55+)	Subset 0 passing some tests; code is reasonably readable; reasonable test suite
PS (50+)	Good progress on assignment, but not passing autotests
0%	knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 FL for COMP(2041|9044)	submitting any other person's work; this includes joint work.
academic misconduct	submitting another person's work without their consent; paying another person to do work for you.

Intermediate Versions of Work

You are required to submit intermediate versions of your assignment.

Every time you work on the assignment and make some progress you should copy your work to your CSE account and submit it using the `give` command below. It is fine if intermediate versions do not compile or otherwise fail submission tests. Only the final submitted version of your assignment will be marked.

Attribution of Work

This is an individual assignment.

The work you submit must be entirely your own work, apart from any exceptions explicitly included in the assignment specification above. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted.

You are only permitted to request help with the assignment in the course forum, help sessions, or from the teaching staff (the lecturer(s) and tutors) of COMP(2041|9044).

Do not provide or show your assignment work to any other person (including by posting it on the forum), apart from the teaching staff of COMP(2041|9044). If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if that work was submitted without your knowledge or consent; this may apply even if your work is submitted by a third party unknown to you. You will not be penalized if your work is taken without your consent or knowledge.

Do not place your assignment work in online repositories such as github or anywhere else that is publicly accessible. You may use a private repository.

Submissions that violate these conditions will be penalised. Penalties may include negative marks, automatic failure of the course, and possibly other academic discipline. We are also required to report acts of plagiarism or other student misconduct: if students involved hold scholarships, this may result in a loss of the scholarship. This may also result in the loss of a student visa.

Assignment submissions will be examined, both automatically and manually, for such submissions.