Assignment 2: a file synchroniser

version: 1.0.0 last updated: 2024-07-12 11:00:00

NOTE:

A video explaining the assignment can be found here.

An additional help video for subset 2 onwards is available at this link.

Contents

Aims

- building a concrete understanding of file system objects;
- practising C, including bitwise operations and robust error handling;
- understanding file operations, including input-output operations on binary data

The Task

The `rsync` utility is a useful and popular tool which efficiently transfers files between computers. In this assignment you will be implementing **rbuoy**, which is a simplified version of **rsync**.

To copy a file from a sending computer to a receiver, it would theoretically be sufficient to just naïvely send over the entire contents (and possibly metadata) of the file.

However, if the receiver already has an older version of the file which is very close to the sender's version (or even an identical copy!), then a large amount of redundant data is being transmitted. With slow networks or large file sizes this can translate to a unnecessary waiting and cost.

Both the real **rsync** utility and the **rbuoy** utility that you'll be implementing in this assignment avoid unnecessary data transfer by only sending the chunks of a file which differ between sender and receiver. The **rbuoy** algorithm takes place over four stages:

1. **Stage 1**: the sender constructs a *Type A Buoy Index* (**TABI**) file containing a record for each file the sender wants to send. Each record contains metadata about the file, as well as a hash for each block in the file (see the subset 1 description for more information).

2. **Stage 2**: the receiver uses the **TABI** file to construct a *Type B Buoy Index* (**TBBI**) file containing a record for each **TABI** record. The **TBBI** file contains information about which blocks the receiver already has an up-to-date copy of (see the subset 2 description for more information).
3. **Stage 3**: the sender uses the *Type B Buoy Index* file to construct a *Type C Buoy Index* (**TCBI**) file containing a record for each **TBBI** record. The **TCBI** file contains the contents of the blocks which the receiver did not have an up-to-date copy of (see the subset 3 description for more information).
4. **Stage 4**: the receiver uses the **TCBI** file to reconstruct an up-to-date copy of the files it is receiving. (see the subset 4 description for more information).

The first four subsets of this assignment correspond to implementing each of these stages for a given list of files. The fifth subset involves adding support for directories.

The real **rsync** utility is able to transfer files over a network to a remote computer; where the sender would be one computer and the receiver would be a different computer. It can also transfer files locally, where the 'sender' and 'receiver' are two different directories on the same computer. In this assignment, you will only be implementing the local version of **rbuoy**, where the sender and receiver are two different directories on the same computer.

Getting Started

Create a new directory for this assignment, change to this directory, and fetch the provided code by running

```
$ mkdir -m 700 rbuoy
$ cd rbuoy
$ 1521 fetch rbuoy
```

If you're not working at CSE, you can download the provided files as a zip file or a tar file.

This will give you the following files:

rbuoy.c	is the only file you need to change: it contains partial definitions of four functions, *stage_1*, *stage_2*, *stage_3*, and *stage_4*, to which you need to add code to complete the assignment. You can also add your own functions to this file.
rbuoy_main.c	contains a *main*, which has code to parse the command line arguments, and which then calls one of *stage_1*, *stage_2*, *stage_3*, and *stage_4*, depending on the command line arguments given to rbuoy. *Do not change this file.*
rbuoy.h	contains shared function declarations and some useful constant definitions. *Do not change this file.*
rbuoy_provided.c	contains the *hash_block* function; you should call this function to calculate hashes for subset 1. *Do not change this file.*
rbuoy.mk	contains a Makefile fragment for rbuoy.
rbuoy_hash_block.c	contains the source code for the `1521 rbuoy-hash-block` helper utility which we have provided you. You may find it useful to look at this code to better understand how the *hash_block* function can be used. *Do not change, attempt to compile with, or submit this file.*

You can run *make* to compile the provided code; and you should be able to run the result.

```
$ make
dcc   rbuoy.c rbuoy_main.c rbuoy_provided.c -o rbuoy
$ ./rbuoy
Usage: ./rbuoy [--stage-1|--stage-2|--stage-3|--stage-4]
```

You may optionally create extra `.c` or `.h` files. You can modify the provided Makefile fragment if you choose to do so.

You should run `1521 rbuoy-examples` to get a directory called `examples/` full of test files and example Buoy Index files to test your program against.

```
$ 1521 rbuoy-examples
$ ls examples
aaa  bbb  ccc  tabi  tbbi  tcbi
```

Subset 1

To complete subset 1, you need to complete the provided `stage_1` function.

The `stage_1` function should create a *TABI* file at the specified output path, based on a given array of filenames.

The *TABI* file should contain the appropriate header, as outlined in the format of the TABI file section below.

It should then produce a TABI record for each file in the given array of in_filenames.

```
$ 1521 rbuoy-examples
$ cd examples/aaa
$ ls
emojis.txt  empty  fizz  fractal_bin  little_endian_shorts  long_path  lyrics.txt  short.txt
$ ../../rbuoy --stage-1 ../out.tabi emojis.txt empty
$ 1521 rbuoy-show ../out.tabi
Field name         Offset        Bytes                    ASCII/Numeric
-------------------------------------------------------------------
magic              0x00000000    54 41 42 49              chr TABI
num records        0x00000004    02                       dec 2
============================== Record   0 ==============================
pathname len       0x00000005    0a 00                    dec 10
pathname           0x00000007    65 6d 6f 6a 69 73 2e 74  chr emojis.t
                   0x0000000f    78 74                    chr xt
num blocks         0x00000011    03 00 00                 dec 3
hashes[0]          0x00000014    90 30 e3 14 6e e7 0a 90  chr .0..n...
hashes[1]          0x0000001c    91 90 5c 46 fc 07 b3 93  chr ..\F....
hashes[2]          0x00000024    8c ec 01 86 4c dc 63 af  chr ....L.c.
============================== Record   1 ==============================
pathname len       0x0000002c    05 00                    dec 5
pathname           0x0000002e    65 6d 70 74 79           chr empty
num blocks         0x00000033    00 00 00                 dec 0
```

HINT:

Use *fopen* to create the TABI file for writing. You should overwrite the file if it already exists.

Use *fputc* and/or *fwrite* to write bytes to the *TABI* file.

Use *fgetc* and/or *fread* to read bytes from the input files.

Use *stat* to get the size of each input file. In particular, you may find the `st_size` field of the `struct stat` useful. You may find *inode* to be a useful source of documentation for the `struct stat` fields - note that filesystem blocks are *not* relevant to this assignment, and shouldn't be confused with the blocks in a TABI record.

The provided `number_of_blocks_in_file` function will determine the number of blocks required for a TABI record, given the size of the file in bytes.

Use C bitwise operations such as `<<` `&` and `|` to combine bytes into little endian integers. You may find it useful to write a helper function to do this, as you will need to do this in later subsets.

Make sure you understand the format of the TABI file.

To compute the hashes field, you will need to open and read from the file, and for each block use the provided `hash_block` function.

Think carefully about the functions you can construct to avoid repeated code.

NOTE:

TABI files do not necessarily end with `.tabi`. This has been done with the provided example files purely as a convenience.

You may assume any paths in `in_filenames` are either regular files or do not exist.

Subset 2

To complete subset 2, you need to complete the provided `stage_2` function.

The stage 2 function receives a path to an input *TABI* file and a path to an output *TBBI* file.

The *TBBI* file should contain the appropriate header, as outlined in the format of the TBBI file section below.

It should then produce a TBBI record for each file in the given *TABI* file.

```
$ # [continued from subset 1 example]
$ cd ../bbb
$ ../../rbuoy --stage-2 ../out.tbbi ../out.tabi
$ 1521 rbuoy-show ../out.tbbi
Field name         Offset        Bytes                    ASCII/Numeric
-------------------------------------------------------------------------
magic              0x00000000    54 42 42 49              chr TBBI
num records        0x00000004    02                       dec 2
============================== Record   0 ==============================
pathname len       0x00000005    0a 00                    dec 10
pathname           0x00000007    65 6d 6f 6a 69 73 2e 74  chr emojis.t
                   0x0000000f    78 74                    chr xt
num blocks         0x00000011    03 00 00                 dec 3
matches[0]         0x00000014    a0                       bin 10100000
============================== Record   1 ==============================
pathname len       0x00000015    05 00                    dec 5
pathname           0x00000017    65 6d 70 74 79           chr empty
num blocks         0x0000001c    00 00 00                 dec 0
```

NOTE:

Remember that stage 2 will typically be invoked in a different working directory to the directory in which stage 1 was invoked.

HINT:

You will need to detect invalid TABI files being supplied to stage 2, and handle them appropriately. You may find it handy to refer to the section on error handling below.

Use C bitwise operations such as `<<` and `|` to construct the matches field.

You may find the provided `num_tbbi_match_bytes` function to be helpful.

Subset 3

In subset 3, you will need to complete the provided `stage_3` function, you will need to produce a *TCBI* file given a *TBBI* file as input.

The *TCBI* file should contain the appropriate header, as outlined in the format of the TCBI file section below. It should also contain a *TCBI* record for each file in the given *TBBI* file, containing the data for the blocks the receiver didn't already have an up-to-date copy of.

```
$ # [continued from subset 2 example]
$ cd ../aaa
$ ../../rbuoy --stage-3 ../out.tcbi ../out.tbbi
$ 1521 rbuoy-show ../out.tcbi
Field name         Offset        Bytes                    ASCII/Numeric
-----------------------------------------------------------------------
magic              0x00000000    54 43 42 49              chr TCBI
num records        0x00000004    02                       dec 2
============================= Record   0 ==============================
pathname len       0x00000005    0a 00                    dec 10
pathname           0x00000007    65 6d 6f 6a 69 73 2e 74  chr emojis.t
                   0x0000000f    78 74                    chr xt
file type          0x00000011    2d                       chr -
owner perms        0x00000012    72 77 2d                 chr rw-
group perms        0x00000015    72 2d 2d                 chr r--
other perms        0x00000018    2d 2d 2d                 chr ---
file size          0x0000001b    01 02 00 00              dec 513
num updates        0x0000001f    01 00 00                 dec 1
(0) block num      0x00000022    01 00 00                 dec 1
(0) update len     0x00000025    00 01                    dec 256
(0) update data    0x00000027    54 68 65 20 73 65 63 6f  chr The seco
                   0x0000002f    6e 64 20 62 6c 6f 63 6b  chr nd block
                [... omitted for brevity ...]
                   0x00000117    73 20 61 73 74 65 72 69  chr s asteri
                   0x0000011f    73 6b 20 2d 2d 3e 20 2a  chr sk --> *
============================= Record   1 ==============================
pathname len       0x00000127    05 00                    dec 5
pathname           0x00000129    65 6d 70 74 79           chr empty
file type          0x0000012e    2d                       chr -
owner perms        0x0000012f    72 77 2d                 chr rw-
group perms        0x00000132    72 2d 2d                 chr r--
other perms        0x00000135    2d 2d 2d                 chr ---
file size          0x00000138    00 00 00 00              dec 0
num updates        0x0000013c    00 00 00                 dec 0
```

HINT:

- You may find the *stat* system call to be useful here - in particular, the `st_mode` field of the `struct stat` supplied.

Subset 4

So far, we've created several types of rbuoy indices files in order to communicate the current state of the receiver's files to the sender, and to communicate updated blocks from the sender to the receiver. In this subset, you will need to complete the provided `stage_4` function, which will be invoked with a *TCBI* file as input. You will then need to apply the changes described in the *TCBI* file to the receiver's files. This includes updating the contents of the receiver's files, and creating any new files that are required. You will also need to update the mode of the receiver's files such that the permissions match those described in the *TCBI* file.

```
$ # [continued from subset 3 example]
$ cd ../bbb
$ ../../rbuoy --stage-4 ../out.tcbi
$ diff ../aaa/empty ../bbb/empty # identical
$ diff ../aaa/emojis.txt ../bbb/emojis.txt # identical
$ # we have now synchronised `empty` and `emojis.txt` from aaa/ to bbb/
```

HINT:

- You may find *chmod* and *fseek* to be useful here.

Subset 5

Subset 5 requires you to add support for directories. You will need to update your `stage_1`, `stage_2`, `stage_3` and `stage_4` implementations to complete subset 5:

- In `stage_1`, if the value of `num_in_filenames` is zero, then you should create a `TABI` file containing the contents of the entire current working directory. When `num_in_filenames` is non-zero you can still make the assumption that all paths in `in_filenames` are either regular files or don't exist.

When creating a *TABI* file for the current directory, you should include a record for every directory, as well as every file. Records for directories should have their number of blocks as zero. The record for a parent directory should be placed in the *TABI* file before any records for files or sub-directories in that parent directory. Apart from that restriction, you may choose any order for records in the generated *TABI* file.

- In `stage_2`, a record with a path which is a directory for the receiver should result in all match bits being set to zero.
- In `stage_3`, a record with a path which is a directory for the sender should be treated as an empty file. That is, the number of blocks should be checked to be zero, and a record with no updates should be generated. Note that the file type of the mode should be a `d` rather than a `-`.
- In `stage_4`, you should create directories for directory records if they do not exist. You should also set the correct permissions for directories. If a record for a file has the path of an existing directory, or a record for a directory has the path of an existing file, then you should output an appropriate error message and exit with status 1.

Additionally, you must add checks in `stage_2`, `stage_3` and `stage_4` to detect if any paths referenced in the input rbuoy indices reference files outside the current working directory. When that occurs, you should output an appropriate error message and exit with status 1. In real code, it is important that untrusted user input such as paths cannot be used to do damage to the wider system. You may assume that if any initial segment of the path exits the current working directory then the whole path will exit the current working directory.

You are encouraged to use the reference implementation to check that your understanding of the above subset 5 requirements are correct.

HINT:

You may find *opendir*, *readdir*, *closedir* to be useful here.

Error handling

Error checking is an important part of this assignment. Automarking will test error handling.

Error messages should be one line (only) and be written to `stderr` (not `stdout`).

rbuoy should `exit` with status 1 after an error.

You do not have to free memory or close files before exiting in the event of an error.

rbuoy should check all file operations for errors.

As much as possible match the reference implementation error messages exactly.

The reference implementation uses *perror* to report errors from file operations and other system calls.

It is not necessary to remove files and directories already created or partially created when an error occurs.

You may leave any created rbuoy indices in an indeterminate state.

Where multiple error messages could be produced, *rbuoy* may produce any one of the error messages.

In stages 2, 3, and 4 you **cannot** assume that the input rbuoy indices are in a valid format. If your program is given an invalid Buoy Index file, you must output an appropriate error message to `stderr` and `exit` with status 1.

During automarking to be awarded marks for the error handling tests you'll need to also have passed a sufficient proportion of the non-error tests for that subset.

Reference implementation

A reference implementation is a common, efficient, and effective method to provide or define an operational specification; and it's something you will likely work with after you leave UNSW.

We've provided a reference implementation, `1521 rbuoy`, which you can use to find the correct outputs and behaviours for any input:

```
$ 1521 rbuoy-examples
$ cd examples
$ cd aaa
$ 1521 rbuoy --stage-1 ../out.tabi short.txt
$ 1521 rbuoy-show ../out.tabi
Field name         Offset        Bytes                    ASCII/Numeric
-------------------------------------------------------------------
magic              0x00000000    54 41 42 49              chr TABI
num records        0x00000004    01                       dec 1
============================ Record   0 ==============================
pathname len       0x00000005    09 00                    dec 9
pathname           0x00000007    73 68 6f 72 74 2e 74 78  chr short.tx
                   0x0000000f    74                       chr t
num blocks         0x00000010    01 00 00                 dec 1
hashes[0]          0x00000013    15 b8 4c 98 fe c3 b7 d6  chr ..L.....
```

Every concrete example shown below is runnable using the helper utilities; run `1521 rbuoy` instead of `./rbuoy`.

The command `1521 rbuoy-show <name of index>` display the contents of *TABI*, *TBBI* and *TCBI* files in a human readable format. It is useful for understanding the output of both the reference implementation and your own implementation.

Where any aspect of this assignment is undefined in this specification, you should match the behaviour exhibited by the reference implementation. Discovering and matching the reference implementation's behaviour is deliberately a part of this assignment.

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

Helper utilities

Alongside `1521 rbuoy-show`, which was used above, we have also provided you two additional utilities - `1521 rbuoy-dump-blocks` and `1521 rbuoy-hash-block`. These utilities have been provided to assist you in understanding the requirements of the assignment, and to help you debug your program.

`1521 rbuoy-dump-blocks` takes a file as input and splits it into 256 (`BLOCK_SIZE`) byte blocks, and outputs it to `stdout` either in hex format or raw bytes. This is useful for ensuring that your program is correctly splitting files into blocks.

```
$ 1521 rbuoy-dump-blocks ---raw examples/aaa/emojis.txt
=== block 0 ===
This file should be broken up by your program into three blocks: the
first 256 bytes spans lines one to four (and includes the newline on line
four), the second 256 bytes is from line 5 to the asterisk (inclusive), and
the final block is only 1 byte long!

=== block 1 ===
The second block started on this line. Now for an assortment of emoji:
✨ ✨ ✨ 1️⃣ 5️⃣ 2️⃣ 1️⃣ ✨ ✨ ✨
📚 🎓 📈 📈 💾 📀 💿 🖥 💻 🚀 🌌 🤯 🎉 🥳
The last character of this block is this asterisk --> *
=== block 2 ===
a
[... no newline after output ...]
$ 1521 rbuoy-dump-blocks --hex examples/aaa/emojis.txt
=== block 0 ===
    54 68 69 73 20 66 69 6c 65 20 73 68 6f 75 6c 64
    20 62 65 20 62 72 6f 6b 65 6e 20 75 70 20 62 79
    20 79 6f 75 72 20 70 72 6f 67 72 61 6d 20 69 6e
    74 6f 20 74 68 72 65 65 20 62 6c 6f 63 6b 73 3a
    20 74 68 65 0a 66 69 72 73 74 20 32 35 36 20 62
    79 74 65 73 20 73 70 61 6e 73 20 6c 69 6e 65 73
    20 6f 6e 65 20 74 6f 20 66 6f 75 72 20 28 61 6e
    64 20 69 6e 63 6c 75 64 65 73 20 74 68 65 20 6e
    65 77 6c 69 6e 65 20 6f 6e 20 6c 69 6e 65 0a 66
    6f 75 72 29 2c 20 74 68 65 20 73 65 63 6f 6e 64
    20 32 35 36 20 62 79 74 65 73 20 69 73 20 66 72
    6f 6d 20 6c 69 6e 65 20 35 20 74 6f 20 74 68 65
    20 61 73 74 65 72 69 73 6b 20 28 69 6e 63 6c 75
    73 69 76 65 29 2c 20 61 6e 64 0a 74 68 65 20 66
    69 6e 61 6c 20 62 6c 6f 63 6b 20 69 73 20 6f 6e
    6c 79 20 31 20 62 79 74 65 20 6c 6f 6e 67 21 0a

=== block 1 ===
    54 68 65 20 73 65 63 6f 6e 64 20 62 6c 6f 63 6b
    20 73 74 61 72 74 65 64 20 6f 6e 20 74 68 69 73
    20 6c 69 6e 65 2e 20 4e 6f 77 20 66 6f 72 20 61
    6e 20 61 73 73 6f 72 74 6d 65 6e 74 20 6f 66 20
    65 6d 6f 6a 69 3a 0a e2 9c a8 20 e2 9c a8 20 e2
    9c a8 20 31 ef b8 8f e2 83 a3 20 35 ef b8 8f e2
    83 a3 20 32 ef b8 8f e2 83 a3 20 31 ef b8 8f e2
    83 a3 20 20 e2 9c a8 20 e2 9c a8 20 e2 9c a8 0a
    f0 9f 93 9a 20 f0 9f 8e 93 20 f0 9f 93 88 20 f0
    9f 93 88 20 f0 9f 92 be 20 f0 9f 92 bd 20 f0 9f
    92 bf 20 f0 9f 96 a5 ef b8 8f 20 f0 9f 92 bb 20
    f0 9f 9a 80 20 f0 9f 8c 8c 20 f0 9f a4 af 20 f0
    9f 8e 89 20 f0 9f a5 b3 0a 54 68 65 20 6c 61 73
    74 20 63 68 61 72 61 63 74 65 72 20 6f 66 20 74
    68 69 73 20 62 6c 6f 63 6b 20 69 73 20 74 68 69
    73 20 61 73 74 65 72 69 73 6b 20 2d 2d 3e 20 2a

=== block 2 ===
    61
```

Additionally, `1521 rbuoy-dump-blocks` is able to only output a single block, specified by the `--index` option. For example, to only output the first block of the file `examples/aaa/emojis.txt` as hex, you would run:

```
$ 1521 rbuoy-dump-blocks --index 0 --hex rbuoy/examples/aaa/emojis.txt
=== block 0 ===
    54 68 69 73 20 66 69 6c 65 20 73 68 6f 75 6c 64
    20 62 65 20 62 72 6f 6b 65 6e 20 75 70 20 62 79
    20 79 6f 75 72 20 70 72 6f 67 72 61 6d 20 69 6e
    74 6f 20 74 68 72 65 65 20 62 6c 6f 63 6b 73 3a
    20 74 68 65 0a 66 69 72 73 74 20 32 35 36 20 62
    79 74 65 73 20 73 70 61 6e 73 20 6c 69 6e 65 73
    20 6f 6e 65 20 74 6f 20 66 6f 75 72 20 28 61 6e
    64 20 69 6e 63 6c 75 64 65 73 20 74 68 65 20 6e
    65 77 6c 69 6e 65 20 6f 6e 20 6c 69 6e 65 0a 66
    6f 75 72 29 2c 20 74 68 65 20 73 65 63 6f 6e 64
    20 32 35 36 20 62 79 74 65 73 20 69 73 20 66 72
    6f 6d 20 6c 69 6e 65 20 35 20 74 6f 20 74 68 65
    20 61 73 74 65 72 69 73 6b 20 28 69 6e 63 6c 75
    73 69 76 65 29 2c 20 61 6e 64 0a 74 68 65 20 66
    69 6e 61 6c 20 62 6c 6f 63 6b 20 69 73 20 6f 6e
    6c 79 20 31 20 62 79 74 65 20 6c 6f 6e 67 21 0a
```

We have also provided a `1521 rbuoy-hash-block` command that reads up to 256 (`BLOCK_SIZE`) bytes from standard input and outputs the 64-bit hash of the data as a hex string, using the same `hash_block` function as provided for the assignment. We've also provided you the source code for this command in rbuoy_hash_block.c for your reference.

You can combine these commands to check the hash of any given block of an input file, for example:

```
$ 1521 rbuoy-dump-blocks --index 0 --raw examples/aaa/emojis.txt | 1521 rbuoy-hash-block
900ae76e14e33090
```

It is important to use the `--raw` option and specify a block index in order to produce the expected hash for that block.

Formats of rbuoy indices

The rbuoy indices emitted by your implementation must follow the exact format produced by the reference implementation.

Type A Buoy Index format

When a sender wants to send files, it first creates a *TABI* file. This file contains a record for each file that is going to be sent. In each record is the pathname of the file, the number of blocks in the file (computed by `number_of_blocks_in_file`), and the hash of each block in the file.

A *TABI* file consists of a header, followed by 0 or more records. The format of the header is:

name	length	type	description
magic number	4 B (byte)	characters sequence	The magic number for *TABI* files, which is the sequence of bytes 0x54, 0x41, 0x42, 0x49 (ASCII `TABI`).
number of records	1 B (byte)	unsigned, 8-bit	The number of records in this *TABI* file.

The *TABI* header is followed by the specified number of records. Each *TABI* record has the following format:

name	length	type	description
pathname length	2 B (byte)	unsigned, 16-bit, little-endian	The length of the pathname of this record.
pathname	*pathname-length*	character sequence	The pathname of the file of this record. It is *not* nul-terminated.
number of blocks	3 B (byte)	unsigned, 24-bit, little-endian	The number of 256-byte blocks in the sender's version of the file (the final block may be shorter than 256 bytes).
hashes	8 B (byte) × *num-blocks*	sequence of unsigned, 64-bit, little-endian integers	The hashes the sender has computed for their version of the file (using the `hash_block` function), with one 64-bit hash for each block.

An example *TABI* file, displayed using `1521 rbuoy-show` :

```
$ 1521 rbuoy-examples
$ cd examples
$ 1521 rbuoy-show tabi/my_text_files.tabi
Field name         Offset        Bytes                    ASCII/Numeric
-------------------------------------------------------------------
magic              0x00000000    54 41 42 49              chr TABI
num records        0x00000004    03                       dec 3
============================== Record   0 ==============================
filename len       0x00000005    09 00                    dec 9
filename           0x00000007    73 68 6f 72 74 2e 74 78  chr short.tx
                   0x0000000f    74                       chr t
num blocks         0x00000010    01 00 00                 dec 1
hashes[0]          0x00000013    15 b8 4c 98 fe c3 b7 d6  chr ..L.....
============================== Record   1 ==============================
filename len       0x0000001b    0a 00                    dec 10
filename           0x0000001d    65 6d 6f 6a 69 73 2e 74  chr emojis.t
                   0x00000025    78 74                    chr xt
num blocks         0x00000027    03 00 00                 dec 3
hashes[0]          0x0000002a    90 30 e3 14 6e e7 0a 90  chr .0..n...
hashes[1]          0x00000032    91 90 5c 46 fc 07 b3 93  chr ..\F....
hashes[2]          0x0000003a    8c ec 01 86 4c dc 63 af  chr ....L.c.
============================== Record   2 ==============================
filename len       0x00000042    05 00                    dec 5
filename           0x00000044    65 6d 70 74 79           chr empty
num blocks         0x00000049    00 00 00                 dec 0
```

The above example shows that the sender is sending three files: `short.txt`, `emojis.txt`, and `empty`. The file `short.txt` has one block of data (so its length must be between 1 and 256), and that block has a hash `0xd6b7c3fe984cb815`.

The second file `emojis.txt` has 3 blocks, so its length must be between 513 and 768. The first block (bytes at indices 0..255) hashes to `0x900ae76e14e33090`, the second block (bytes and indices 256..511) has a hash of `0x93b307fc465c9091` and the final block (bytes from index 512 to the end of the file) has a hash of `0xaf63dc4c8601ec8c`.

The final record is for the file named `empty`. Since it has zero blocks it must be, as its name suggests, empty.

Type B Buoy Index format

After a receiver receives a *TABI* file, it responds with a *TBBI* file, containing information about which blocks the receiver already has a copy of. A *TBBI* file contains a header, followed by zero or more records. The format for the header is:

name	length	type	description
magic number	4 B (byte)	characters sequence	The magic number for *TBBI* files, which is the sequence of bytes 0x54, 0x42, 0x42, 0x49 (ASCII `TBBI`).
number of records	1 B (byte)	unsigned, 8-bit	The number of records in this *TBBI* file.

Following the *TBBI* header are the records. The receiver creates one record for each record in the *TABI* file. Each *TBBI* record has the following format:

name	length	type	description
pathname length	2 B (byte)	unsigned, 16-bit, little-endian	The length of the pathname of this record.
pathname	*pathname-length*	character sequence	The pathname of the file of this record. It is *not* nul-terminated.
number of blocks	3 B (byte)	unsigned, 24-bit, little-endian	The number of blocks in the **sender's** version of the file. This is the same value as the number of blocks in the *TABI* file.

name	length	type	description
matches	ceil(*num-blocks* ÷ 8) (`num_tbbi_match_bytes`)	bit sequence	A sequence of bits, with a single bit for each hash in the *TABI* file. For each hash in the *TABI* file, the receiver computes the hash for the corresponding block in their own copy of the file. If the two hashes match, then the corresponding match bit is a 1. Otherwise (if the hashes don't match, there is no corresponding block because the receiver's file is too small, or the file doesn't exist) the corresponding bit is 0. This means that if the file doesn't exist for the receiver, all the bits in the `matches` field will be 0. The first bit is the most significant bit of the first byte. In the case where the number of blocks is not a multiple of 8, the last byte of the `matches` field is right-padded with 0 bits.

An example *TBBI* file, displayed using `1521 rbuoy-show`:

```
$ 1521 rbuoy-examples
$ cd examples
$ 1521 rbuoy-show tbbi/bbb_text_files.tbbi
Field name         Offset        Bytes                    ASCII/Numeric
-------------------------------------------------------------------------
magic              0x00000000    54 42 42 49              chr TBBI
num records        0x00000004    03                       dec 3
============================== Record   0 ==============================
pathname len       0x00000005    09 00                    dec 9
pathname           0x00000007    73 68 6f 72 74 2e 74 78  chr short.tx
                   0x0000000f    74                       chr t
num blocks         0x00000010    01 00 00                 dec 1
matches[0]         0x00000013    00                       bin 00000000
============================== Record   1 ==============================
pathname len       0x00000014    0a 00                    dec 10
pathname           0x00000016    65 6d 6f 6a 69 73 2e 74  chr emojis.t
                   0x0000001e    78 74                    chr xt
num blocks         0x00000020    03 00 00                 dec 3
matches[0]         0x00000023    a0                       bin 10100000
============================== Record   2 ==============================
pathname len       0x00000024    05 00                    dec 5
pathname           0x00000026    65 6d 70 74 79           chr empty
num blocks         0x0000002b    00 00 00                 dec 0
```

The above example file shows a response to the `tabi/my_text_files.tabi` *TABI* file. The first file, `short.txt` has only one block. Since the first bit of `matches` is a `0`, this means that either the first block of the receiver's version of `short.txt` didn't have a hash of `0xd6b7c3fe984cb815`, or the receiver didn't have the file `short.txt` at all. The remaining 7 bits are padding bits, and so are all zero.

The second file, `emojis.txt` has 3 blocks. The first of the 3 match bits is a `1`. This means that the the first block of the receiver's `emojis.txt` had a hash which matched the hash of the first block in the *TABI* record (0x900ae76e14e33090). The second bit is a 0, meaning that the receiver's second block didn't match the second hash in the *TABI* record. The third bit is a 1, so the third block did match. The remaining 5 bits are padding bits, and so are all zero.

The final file, `empty`, has zero blocks. Since `num_tbbi_match_bytes(0) == 0`, this means that there are no match bytes included in the *TBBI* file.

Type C Buoy Index format

After the sender receives the *TBBI* file, it responds with a *TCBI* file, containing the data for the blocks which the receiver didn't already have a copy of. A *TCBI* file contains a header, followed by zero or more records. The format for the header is:

name	length	type	description
magic number	4 B (byte)	characters sequence	The magic number for *TCBI* files, which is the sequence of bytes 0x54, 0x43, 0x42, 0x49 (ASCII `TCBI`).
number of records	1 B (byte)	unsigned, 8-bit	The number of records in this *TCBI* file.

Following the *TCBI* header are the records. The sender creates one record for each record in the *TBBI* file. Each *TCBI* record has two sections. The first section has the following format:

name	length	type	description
pathname length	2 B (byte)	unsigned, 16-bit, little-endian	The length of the pathname of this record.
pathname	*pathname-length*	character sequence	The pathname of the file of this record. It is *not* nul-terminated.
mode	10 B (byte)	characters	The type and permissions as a *ls*-like character array; e.g., `"-rwxr-xr-x"`. It is *not* nul-terminated.
file size	4 B (byte)	unsigned, 32-bit, little-endian	The size of the sender's version of the file, in bytes.
number of updates	3 B (byte)	unsigned, 24-bit, little-endian	The number of updates in this record.

The second section of the record contains the updates for that file. An update contains a block of data which the receiver needs. The number of updates for a record is equal to the number of non-padding 0 bits in the *TBBI* record. Each non-padding 0 bit creates an update. An update has the following format:

name	length	type	description
block index	3 B (byte)	unsigned, 24-bit, little-endian	The index of the block that this update is for. This is zero-indexed - the first block in a file has an index of 0, the second block has an index of 1, and so on.
update length	2 B (byte)	unsigned, 16-bit, little-endian	The number of bytes in the block that is being updated. For any block apart from the trailing block, this is equal to 256. But the final block in a file might be shorter than that.
update data	*update-length*	bytes	The block at *block-index* from the sender's version of the file.

An example *TCBI* file, displayed using `1521 rbuoy-show`:

```
$ 1521 rbuoy-examples
$ cd examples
$ 1521 rbuoy-show tcbi/bbb_text_files.tcbi
Field name         Offset        Bytes                    ASCII/Numeric
-------------------------------------------------------------------
magic              0x00000000    54 43 42 49              chr TCBI
num records        0x00000004    03                       dec 3
============================ Record   0 ==============================
pathname len       0x00000005    09 00                    dec 9
pathname           0x00000007    73 68 6f 72 74 2e 74 78  chr short.tx
                   0x0000000f    74                       chr t
file type          0x00000010    2d                       chr -
owner perms        0x00000011    72 77 2d                 chr rw-
group perms        0x00000014    72 2d 2d                 chr r--
other perms        0x00000017    2d 2d 2d                 chr ---
file size          0x0000001a    40 00 00 00              dec 64
num updates        0x0000001e    01 00 00                 dec 1
(0) block num      0x00000021    00 00 00                 dec 0
(0) update len     0x00000024    40 00                    dec 64
(0) update data    0x00000026    54 68 69 73 20 74 65 78  chr This tex
                   0x0000002e    74 20 66 69 6c 65 20 68  chr t file h
                   0x00000036    61 73 20 73 69 78 74 79  chr as sixty
                   0x0000003e    20 66 6f 75 72 20 62 79  chr  four by
                   0x00000046    74 65 73 2c 20 74 77 65  chr tes, twe
                   0x0000004e    6c 76 65 20 77 6f 72 64  chr lve word
                   0x00000056    73 20 61 6e 64 20 6f 6e  chr s and on
                   0x0000005e    65 20 6c 69 6e 65 2e 0a  chr e line..
============================ Record   1 ==============================
pathname len       0x00000066    0a 00                    dec 10
pathname           0x00000068    65 6d 6f 6a 69 73 2e 74  chr emojis.t
                   0x00000070    78 74                    chr xt
file type          0x00000072    2d                       chr -
owner perms        0x00000073    72 77 2d                 chr rw-
group perms        0x00000076    72 2d 2d                 chr r--
other perms        0x00000079    2d 2d 2d                 chr ---
file size          0x0000007c    01 02 00 00              dec 513
num updates        0x00000080    01 00 00                 dec 1
(0) block num      0x00000083    01 00 00                 dec 1
(0) update len     0x00000086    00 01                    dec 256
(0) update data    0x00000088    54 68 65 20 73 65 63 6f  chr The seco
                   0x00000090    6e 64 20 62 6c 6f 63 6b  chr nd block
                   0x00000098    20 73 74 61 72 74 65 64  chr  started
                [... omitted for brevity ...]
                   0x00000170    6b 20 69 73 20 74 68 69  chr k is thi
                   0x00000178    73 20 61 73 74 65 72 69  chr s asteri
                   0x00000180    73 6b 20 2d 2d 3e 20 2a  chr sk --> *
============================ Record   2 ==============================
pathname len       0x00000188    05 00                    dec 5
pathname           0x0000018a    65 6d 70 74 79           chr empty
file type          0x0000018f    2d                       chr -
owner perms        0x00000190    72 77 2d                 chr rw-
group perms        0x00000193    72 2d 2d                 chr r--
other perms        0x00000196    2d 2d 2d                 chr ---
file size          0x00000199    00 00 00 00              dec 0
num updates        0x0000019d    00 00 00                 dec 0
```

The above example file shows a response to the `tbbi/bbb_text_files.tbbi` *TBBI* file.

Since the receiver indicated that it didn't have the first block of `shorts.txt`, a single update is sent containing the contents of that first block. Since the first block (which is also the last block) has a length of 64 bytes, the update length is also 64 bytes.

The second file, `emojis.txt` has 3 blocks. But since the receiver indicated it had the first and third block, only the second block needs to be sent across, so there is only one update. The second block (since it's not the final block) has a length of 256 bytes.

The final file, `empty`, does not require any updates (there are no non-padding zero bits in the *TBBI* file), so the number of updates is zero.

The file type (`-` for file and `d` for directory), as well as the permissions, are also included in every record, even if there are no updates.

Hashing and the hash_block function

A hash function is a function which takes a sequence of bytes and returns a fixed-length value called a hash. The hash is usually much smaller than the input, and is often used to verify that the input has not been modified without having to store the entire input. For example, if you download a file from the internet, you can verify that the file hasn't been corrupted by comparing the hash of the file you downloaded to the

hash of the file published by the author. If the hashes are the same, then the file is almost certainly the same as well, as hash functions are designed to produce different hashes for even slightly different inputs, and be very unlikely to produce the same hash for two given inputs.

The supplied `hash_block` function takes a sequence of bytes and returns a hash. The hash produced is a 64-bit integer regardless of the size of the input. You are not required to understand how the `hash_block` function works, but you are required to use it in your implementation of `rbuoy` to compute the hashes of blocks.

rbuoy vs rsync (optional extra information)

WARNING:

This section merely contains some extra information about the differences between the **rbuoy** algorithm and the real rsync algorithm. It's not necessary to know this to complete this assignment, nor is it in scope for this course. If you just want to work on the assignment, you can safely scroll down to the Assumptions and Clarifications section.

▶ Click here to view more

Assumptions and Clarifications

Like all good programmers, you should make as few assumptions as possible. If in doubt, match the output of the reference implementation.

- Your submitted code must be a single C program only. You may not submit code in other languages.
- You can call functions from the C standard library available by default on CSE Linux systems: including, e.g., `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `assert.h`, as well as any C POSIX libraries used in lectures or lecture slides such as `unistd.h`, `sys/types.h`, `sys/stat.h`, `fcntl.h`, `dirent.h`.
- We will compile your code with `dcc` when marking. Run-time errors from illegal or invalid C will cause your code to fail automarking (and will likely result in you losing marks).
- Your program must not require extra compile options. It must compile successfully with:

  ```
  $ dcc *.c -o rbuoy
  ```

- You may not use functions from other libraries. In other words, you cannot use the dcc `-l` flag.
- If your program prints debugging output, Make sure you disable any debugging output before submission. it will fail automarking tests.
- You may not create or use temporary files.
- You may not create subprocesses: you may not use *posix_spawn*, *posix_spawnp*, *system*, *popen*, *fork*, *vfork*, *clone*, or any of the `exec*` family of functions, like *execve*.
- *rbuoy* only has to handle ordinary files and directories.

  rbuoy does not have to handle symbolic links, devices or other special files.

  rbuoy will not be run in directories containing symbolic links, devices or other special files.

  rbuoy does not have to handle hard links.
- Outside of the cases of errors or early termination, *rbuoy* must make a reasonable attempt to free all memory it has allocated and close any open files.
- rbuoy will never need to delete any files.
- You may not make any assumptions based off file extensions.
- You must not assume that your program is being run on a system using little-endian byte ordering - you will be assessed on portability with respect to byte ordering.

If you need clarification on what you can and cannot use or do for this assignment, ask in the class forum.

You are required to submit intermediate versions of your assignment. See below for details.

Subset weighting

The weighting of each subset in the performance mark is as follows:

- Subset 1: 40%
- Subset 2: 25%
- Subset 3: 15%
- Subset 4: 12%
- Subset 5: 8%

Change Log

Version 1.0.0

(2024-07-12 11:00:00)

- Initial release.

Version 1.1.0
(2024-07-19 10:00:00)

- Specified performance weight of each subset.

Assessment

Testing

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

```
$ 1521 autotest rbuoy [optionally: any extra .c or .h files]
```

You can also run autotests for a specific subset. For example, to run all tests from subset 1:

```
$ 1521 autotest rbuoy S1 [optionally: any extra .c or .h files]
```

Some tests are more complex than others. If you are failing more than one test, you are encouraged to focus on solving the first of those failing tests. To do so, you can run a specific test by giving its name to the `autotest` command:

```
$ 1521 autotest rbuoy S1_0 [optionally: any extra .c or .h files]
```

`1521 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.

WARNING:

Whilst we can detect errors have occurred, it is often substantially harder to automatically explain what that error was. As you continue into later subsets. the errors from `1521 autotest` will become less and less clear or useful. You will need to do your own debugging and analysis.

Submission

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

```
$ give cs1521 ass2_rbuoy rbuoy.c [optionally: any extra .c or .h files]
```

You must run `give` before **Week 10 Friday 20:00:00** 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:

```
$ 1521 classrun check ass2_rbuoy
```

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 10 Friday 20:00:00** (2024-08-02 20:00:00).

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 COMP1521 mark.

80% of the marks for assignment 2 will come from the performance of your code on a large series of tests.

20% of the marks for assignment 2 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.

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

100% for style	perfect style
90% for style	great style, almost all style characteristics perfect.
80% for style	good style, one or two style characteristics not well done.
70% for style	good style, a few style characteristics not well done.
60% for style	ok style, an attempt at most style characteristics.
≤ 50% for style	an attempt at style.

An indicative style rubric follows:

- Formatting **(6/20)**:
  - Whitespace (e.g. `1 + 2` instead of `1+2`)
  - Indentation (consistent, tabs or spaces are okay)
  - Line length (below 80 characters unless very exceptional)
  - Line breaks (using vertical whitespace to improve readability)
- Documentation **(8/20)**:
  - Header comment (with name and zID)
  - Function comments (above each function with a good description)
  - Descriptive variable names (e.g. `char *home_directory` instead of `char *h`)
  - Descriptive function names (e.g. `get_home_directory` instead of `get_hd`)
  - Sensible commenting throughout the code (don't comment every single line; leave comments when necessary)
- Elegance **(5/20)**:
  - Does this code avoid redundancy? (e.g. Don't repeat yourself!)
  - Are helper functions used to reduce complexity? (functions should be small and simple where possible)
  - Are constants appropriately created and used? (magic numbers should be avoided)
- Portability **(1/20)**:
  - Would this code be able to compile and behave as expected on other POSIX-compliant machines? (using standard libraries without platform-specific code)
  - Does this code make any assumptions about the endianness of the machine it is running on?

Note that the following penalties apply to your total mark for plagiarism:

0 for asst2	knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 FL for COMP1521	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 above. 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.

Assignment Conditions

- **Joint work** is **not permitted** on this assignment.

  This is an individual assignment. The work you submit must be entirely your own work: submission of work even partly written by any other person is not permitted.

  Do not request help from anyone other than the teaching staff of COMP1521 — for example, in the course forum, or in help sessions.

Do not post your assignment code to the course forum. The teaching staff can view code you have recently submitted with give, or recently autotested.

Assignment submissions are routinely examined both automatically and manually for work written by others.

Rationale: this assignment is designed to develop the individual skills needed to produce an entire working program. Using code written by, or taken from, other people will stop you learning these skills. Other CSE courses focus on skills needed for working in a team.

- The use of generative tools such as Github Copilot, ChatGPT, Google Bard is **not permitted** on this assignment.

  Rationale: this assignment is designed to develop your understanding of basic concepts. Using synthesis tools will stop you learning these fundamental concepts, which will significantly impact your ability to complete future courses.

- **Sharing, publishing, or distributing** your assignment work is **not permitted**.

  Do not provide or show your assignment work to any other person, other than the teaching staff of COMP1521. For example, do not message your work to friends.

  Do not publish your assignment code via the Internet. For example, do not place your assignment in a public GitHub repository.

  Rationale: by publishing or sharing your work, you are facilitating other students using your work. If other students find your assignment work and submit part or all of it as their own work, you may become involved in an academic integrity investigation.

- **Sharing, publishing, or distributing** your assignment work after the completion of COMP1521 is **not permitted**.

  For example, do not place your assignment in a public GitHub repository after this offering of COMP1521 is over.

  Rationale: COMP1521 may reuse assignment themes covering similar concepts and content. If students in future terms find your assignment work and submit part or all of it as their own work, you may become involved in an academic integrity investigation.

Violation of any of the above conditions may result in an academic integrity investigation, with possible penalties up to and including a mark of 0 in COMP1521, and exclusion from future studies at UNSW. For more information, read the UNSW Student Code, or contact the course account.