THE UNIVERSITY OF SYDNEY

COMP2017 / COMP9017 Assignment 1

Full assignment due: May 24th, 11:59pm AEST (Week 12 Friday)

Milestone due: May 12th, 11:59pm AEST (Week 10 Sunday)

This assignment is worth 15% of your final assessment

Task Description

In this assignment, you will be implementing your own virtual filesystem. You will first program a library that simulates virtual file operations on a virtual disk. After this, you will extend the functionality of the library using the Filesystem in Userspace (FUSE) module in Linux to enable it to behave as a real filesystem.

Your assignment will also be tested for performance. You will need to implement a multithreaded solution and consider the synchronisation of parallel operations to ensure that your filesystem behaves correctly.

Milestone

To ensure you are making regular progress on this assignment, you must have achieved a minimum level of functionality in your submission by May 12th, 11:59pm AEST (Week 10 Sunday) to receive a portion of the marks. See the Submission section at the end of this document for further details.

Working on your assignment

Staff may make announcements on Ed (`https://edstem.org`) regarding any updates or clarifications to the assignment. You can ask questions on Ed using the assignments category. Please read this assignment description carefully before asking questions. Please ensure that your work is your own and you do not share any code or solutions with other students.

You can work on this assignment using your own computers, lab machines, or Ed. However, it must compile and run on Ed and this will determine the grade. It is important that you continually back up your assignment files onto your own machine, flash drives, external hard drives and cloud storage providers. You are encouraged to submit your assignment while you are in the process of completing it to receive feedback and to check for correctness of your solution.

For Part 3 of this assignment, the FUSE kernel module and library need to be installed. This is not available on Ed. For Part 3, we recommend using your own Linux computer, a Linux virtual machine,

or the lab machines. We are using FUSE version 2.9 for this assignment. Please ensure that if you use FUSE on your own machine, that version 2.9 is installed (not version 3.0 or higher). Version 3.0 or higher of FUSE is NOT compatible. Minor version number differences such as 2.9.7 vs 2.9.9 are not an issue.

If you use a Linux computer or virtual machine, you generally need to install the "fuse" package and also the libfuse development library. On Ubuntu, these packages are **`fuse`** and **`libfuse-dev`**, which you can install by **`sudo apt-get install fuse libfuse-dev`**. Make sure you don't install **`fuse3`** or **`libfuse3-dev`** which correspond to version 3. Once you have fuse installed, you can check the version you have with **`fusermount -V`**, which should show a 2.9.x version.

In other distributions such as Fedora, the FUSE package may be called **`fuse2`**. The development library may be called **`fuse-devel`**. Post on Ed if you are having any difficulties installing the required packages.

The School lab machines should have the required packages installed. Make sure you boot into Linux.

Introduction

Low-level storage devices, such as hard drives, are presented to the operating system as an array of fixed-size blocks of data. Filesystems are (usually) parts of the operating system that present an abstraction of this raw data. It is the filesystem's job to internally organise these blocks, and present a consistent interface of "files" (and directories) to user programs, via system calls (such as **`read()`**, **`write()`**, **`open()`**).

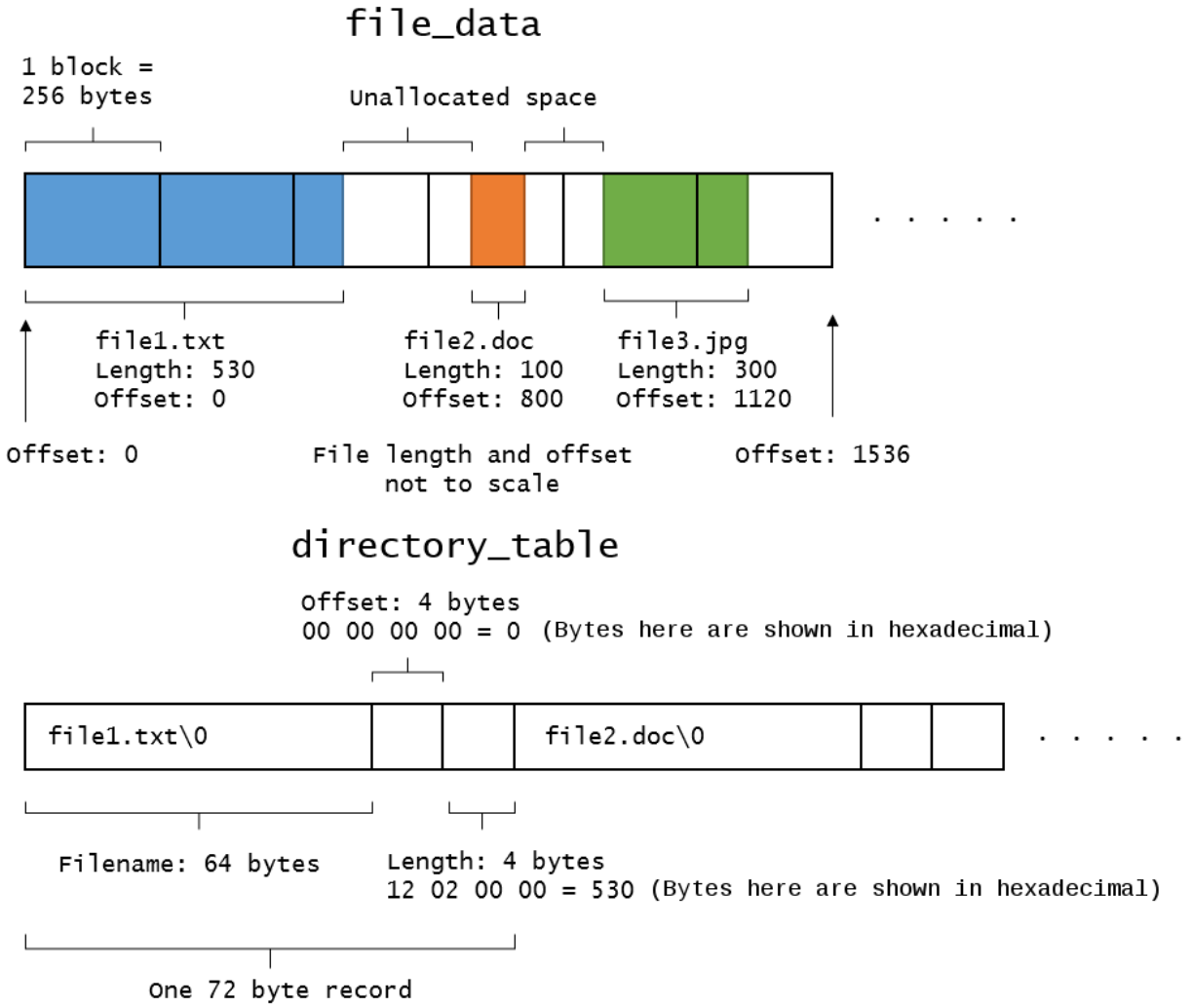
For this reason, filesystems are typically kernel space code. However, in this assignment you will only require the user space programming which you have been learning. As the last part of the assignment, you will use the special FUSE library and module which allows your user space filesystem to link into kernel code and present actual files as part of the real Linux directory tree.

Real filesystems utilise storage devices as mentioned - such as a hard drive, or solid state drive, accessed over a variety of interfaces such as USB. For this assignment, your virtual filesystem will use 3 real files as a simulated storage device, and store the data of your "virtual files" in these.

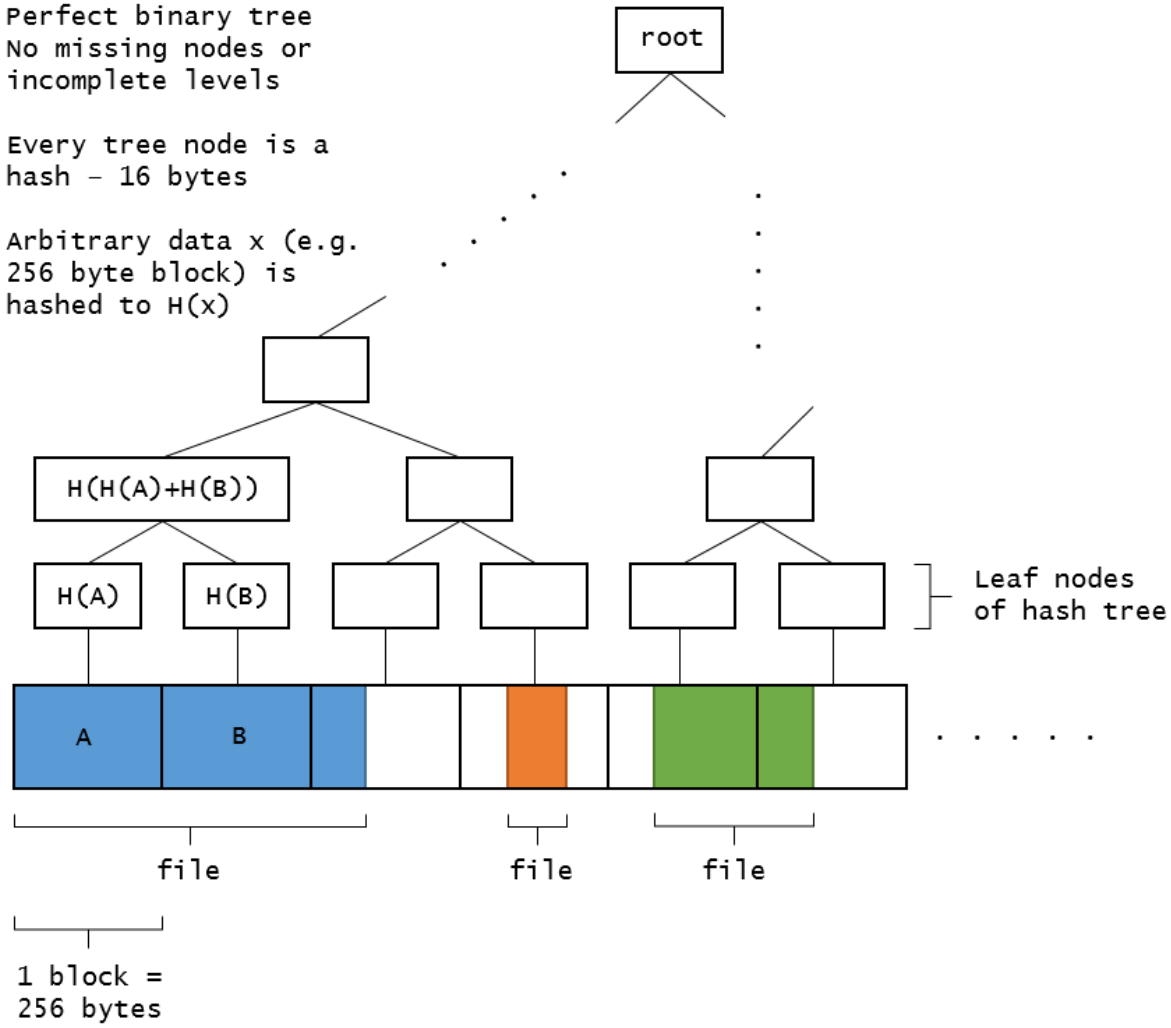
The **`file_data`** file contains the contents of all virtual files stored. Files will be stored as a contiguous series of bytes starting at various offsets in the **`file_data`** file. Note that **`file_data`** contains only the contents of your virtual files and not their filenames. The **`file_data`** file consists of 2^n blocks, each of size 256 bytes. **`n`** is a positive integer with maximum value 24. Files do not have to start or end on block boundaries. Files may span multiple blocks or parts of blocks, but are always a contiguous section of bytes. There may be two or more files contained within one block if they are all less than 256 bytes in size.

The **`directory_table`** file maps filenames to where the corresponding virtual file is stored in **`file_data`**. It consists of a series of 72 byte records. Each record first contains a 64 byte region for a null terminated string that represents the filename. A filename that starts with the null byte is considered to have been deleted. This is followed by the offset and length fields, both of which are 4 bytes, which are unsigned integer values stored in little-endian format. These represent the position of the first byte of the file from the beginning of **`file_data`** in bytes and the length of the file in bytes respectively. The maximum number of records the **`directory_table`** file can contain is 2^{16}. Your filesystem will not support any directories, links, users, owners, permissions, etc.

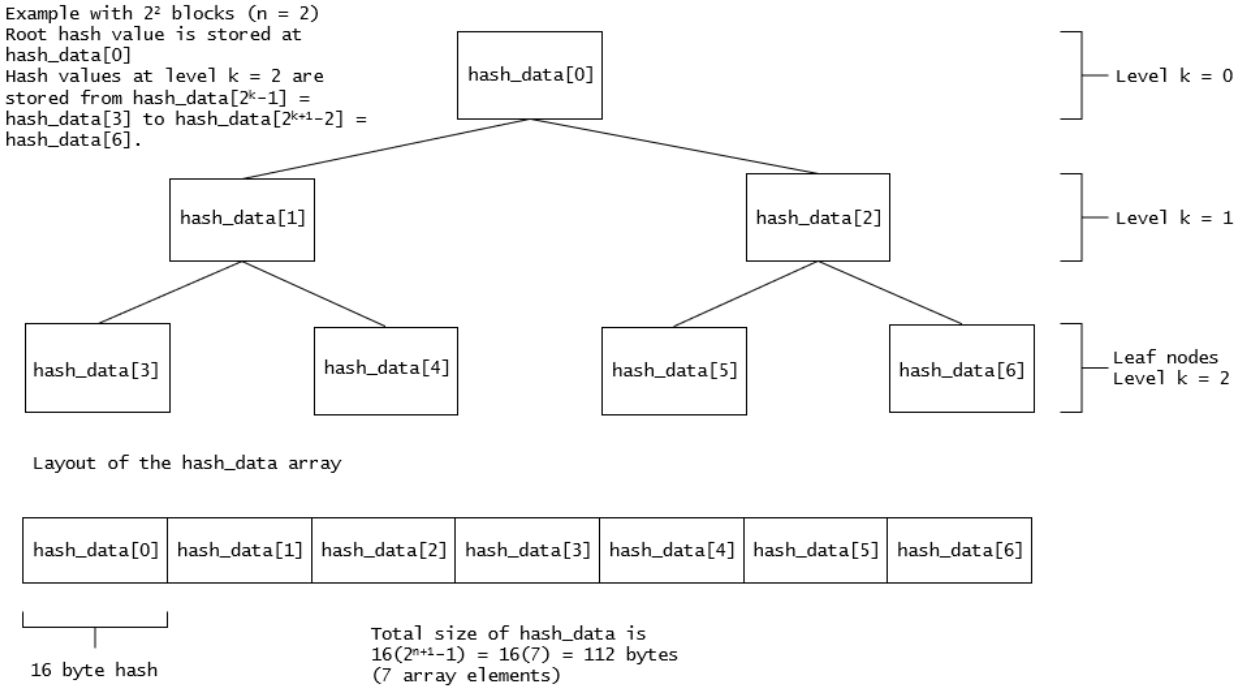
The layout of an example **`file_data`**, with the corresponding **`directory_table`**, is shown in the figure below.



The **hash_data** file contains verification data for your virtual filesystem. You will implement verification in Part 2. You will construct a Merkle hash tree (https://en.wikipedia.org/wiki/Merkle_tree) over your 256-byte blocks from **file_data**. You will implement the simplest form in this assignment, a perfect binary tree of nodes which each contain a hash (all levels of the tree are fully complete with nodes). Every node in the tree, except the leaves, has 2 children. All leaves exist at the same level. A hash function $H(x)$ takes arbitrary input data x and returns a fixed length output h, which is referred to as the hash of that input data. Each leaf node of your tree corresponds to one block of data in **file_data** and stores its hash value. Every other node in your tree stores the following hash of its 2 child nodes: if its child nodes contain hashes $H(a)$ and $H(b)$ (a corresponding to the smaller offset from the start of the file), the parent node stores value $H(H(a) + H(b))$ where $+$ means to concatenate the two hash values. This continues up the Merkle tree to obtain the hash value of the root node. The figure below demonstrates conceptually how the Merkle hash tree is organised on top of the **file_data** example from the first figure.



You will implement a variant of a Fletcher hash function, which will produce a 16 byte hash. Pseudocode is included below. The hash tree is stored in flat array form in **`hash_data`**. The root hash value is stored at **`hash_data[0]`**. Hash values at level **`k`** in the tree (counting from the root node at level 0) will be stored low to high in the array **`hash_data`** index $2^k - 1$ to $2^{k+1} - 2$ inclusive. For a **`file_data`** containing 2^n blocks, the **`hash_data`** therefore has a file size of $16 \times (2^{n+1} - 1)$ bytes. The figure below demonstrates an example of how a Merkle tree would be stored in **`hash_data`** for the case of $n = 2$.



Pseudocode for hash function

```
function fletcher_hash(uint32_t * data, length): //Consider data 4 bytes at a time
    uint64_t a = 0; //Treat 4 byte blocks as little-endian integers
    uint64_t b = 0; //Store in uint64_t to ensure modulus is applied correctly
    uint64_t c = 0;
    uint64_t d = 0;
    for i = 0 ... length-1:
        a = (a + data[i]) mod 2^32-1;
        b = (b + a) mod 2^32-1;
        c = (c + b) mod 2^32-1;
        d = (d + c) mod 2^32-1;
    uint8_t hash_value[16] = concatenate a, b, c, d //4x4 = 16 bytes in total
    //Note: when you concatenate a, b, c, d you must treat them as uint32_t
    //That is, treat them as 4 byte little endian integers, even though
    //we suggest you use uint64_t for the calculations
    return hash_value
```

If data passed into your hash function is not a multiple of 4 bytes in length, pad it to a multiple of 4 bytes with zero bytes.

Part 1

Implement the following functions for your virtual filesystem in **myfilesystem.c**. This file has been included in the scaffold provided. Do not write any **main()** function; your code will be tested by directly calling the functions you implement.

Notes for all functions

For all functions accepting a filename, truncate the filename to 63 characters if the parameter exceeds this length. If this would result in an error (such as a duplicated filename), return the specified error code.

You can assume that test cases will only test your function on a single error at a time.

```
void * init_fs(char * f1, char * f2, char * f3, int n_processors);
```

This function you provide will be called first and once before any further operations are performed. You need to initialise any data structures and perform any preparation you wish. Return a pointer to a memory area (**void ***) where you can store data you wish to use during filesystem operations. For all other functions, this pointer will be passed in so you can access your data as **void * helper**. As parameters, you are provided with the filenames of the 3 files that represent your virtual storage device, as described above in the introduction, as well as an indication of how many processors you have access to. **f1** is the filename of your **file_data** file, **f2** is **directory_table** and **f3** is **hash_data**. You have the opportunity to setup threads or processes at this point to use in further operations, or you can create threads/processes during each individual operation. Your program needs to be ready to handle any number of consecutive filesystem operation calls after **init_fs()** completes.

```
void close_fs(void * helper);
```

This function will be called once and at the end of all operations. You need to clean up any data structures, threads etc. that you created in **init_fs**, including deallocating any dynamic memory.

```
int create_file(char * filename, size_t length, void * helper);
```

Create a file with the given filename and size **length** bytes, and fill it with zero bytes. You must place your file at the smallest offset where there is at least **length** bytes of contiguous space. Place the directory filename entry in the first available slot in the **directory_table**. If there are no suitably sized contiguous spaces, try to create the file again after repacking (see below). Return 0 if the file is successfully created. Return 1 if the filename already exists. Return 2 if there is insufficient space in the virtual disk overall.

```
int resize_file(char * filename, size_t length, void * helper);
```

Resize the file with the given filename to the new **length**. If and only if the new size would not fit in the current space that the file occupies, you need to repack (see below) the other files and attempt to find the first contiguous space. If you are increasing the size of the file, fill the new space with zero bytes. If you are decreasing the size of the file, truncate the end of the file. Return 0 if the file is successfully resized. Return 1 if the file does not exist. Return 2 if there is insufficient space in the virtual disk overall for the new file size.

```
void repack(void * helper);
```

Your **file_data** might have holes, areas of unallocated space between files, which could result

from deleting or resizing files. When this function is called, move the offsets of files such that they are all positioned as far as possible to the left (low offsets) with no unallocated space between them. Do not change the order of the files. Note that you might have to perform repacking as part of other functions. The figure below demonstrates the result of a repacking operation performed on the example **file_data** from the figures in the introduction.


```
int delete_file(char * filename, void * helper);
```

Delete a file with the given filename. You do not need to modify any data in the **file_data** file. Return 0 if the file is successfully deleted and 1 if an error occurs, such as the file not existing.

```
int rename_file(char * oldname, char * newname, void * helper);
```

Rename a file with filename **oldname** to **newname**. Return 0 if the file is successfully renamed and 1

if an error occurs, such as the file not existing.

```
int read_file(char * filename, size_t offset, size_t count, void * buf,
void * helper);
```

Read **count** bytes from the file with given filename at **offset** from the start of the file. Store them into **buf**. Return 0 if successfully completed. If the file does not exist, return 1. If the provided **offset** makes it impossible to read **count** bytes given the file size, return 2.

```
int write_file(char * filename, size_t offset, size_t count, void * buf,
void * helper);
```

Write **count** bytes to the file with given filename, at the given **offset** from the start of the file, reading data from **buf**. If the current filesize is too small, you should dynamically increase the size of the file as required to fit the new data. This may require you to repack the other files to fit the new data (place the file you are writing in the new first contiguous space). Return 0 if successfully completed. If the file does not exist, return 1. If **offset** is greater than the current size of the file, return 2. If there is insufficient space in the virtual disk overall to fit the data to write, return 3.

```
ssize_t file_size(char * filename, void * helper);
```

Return the file size of the file with given filename. If there is an error, such as the file not existing, return -1.

Note on synchronisation

All functions that you write in Part 1 and 2 are blocking. You must perform the operations on your virtual storage files (**file_data**, **directory_table**, **hash_data**) synchronously, meaning you must perform any modifications to those files before returning from your function. Testing code will check the state of the virtual storage files between function calls, not just after **close_fs()** is called.

The testing code that calls your filesystem library functions may be multithreaded. This means that after initialisation, your library functions may be called concurrently. For example, it may request creation of a file whilst a previous read function is still proceeding. You must implement synchronisation primitives to ensure your filesystem can perform such requests correctly. This means ensuring that functions that may modify the same region of data do not interleave such modifications with each other. For example, if two calls to **write_file** to the same region of a file are called simultaneously, you need to ensure that one of the calls is synchronised to only occur after the other is fully completed. It is the user of your filesystem's responsibility to ensure that they synchronise their simultaneous writes correctly such that they do not overwrite each other, but it is your responsibility to ensure that if simultaneous writes are issued, that they are performed sequentially.

For performance reasons, you may also wish to multithread internally within your own filesystem functions. This is separate from your code being tested concurrently.

Part 2

In this section, you need to extend the functionality of functions written in Part 1, and implement a number of new functions. These new functions will also only be called after **`init_fs`** is called and rely on the **`void * helper`** returned from **`init_fs`** to refer to your virtual filesystem. You will implement a Merkle hash tree, storing the data in **`hash_data`** as described in the introduction.

```
void fletcher(uint8_t * buf, size_t length, uint8_t * output);
```

This function does not read or write to your virtual filesystem. You are provided with **`length`** bytes of input data pointed to by **`buf`**. Compute the fletcher hash as described in the introduction on this data, storing it into **`output`**. You may assume **`output`** points to exactly 16 bytes of memory to store the hash.

If you use this function in other functions in your assignment, you are responsible for ensuring that **`output`** has sufficient space to store the hash output in these cases.

```
void compute_hash_tree(void * helper);
```

Compute the entire Merkle hash tree for **`file_data`** and store it in **`hash_data`** according to the layout described in the introduction. Make sure that you read and write these files in binary mode.

```
void compute_hash_block(size_t block_offset, void * helper);
```

Compute the hash for the block at given **`block_offset`** in **`file_data`**, and update all affected hashes in the Merkle hash tree, writing changes to **`hash_data`**. Note that **`block_offset`** counts blocks and not bytes, and is therefore a positive integer less than 2^{24}, since this is the maximum number of blocks in **`file_data`**. The first block has **`block_offset`** equal to 0.

Modifications to Part 1 functions

You need to update the following functions from Part 1.

`create_file(), resize_file(), repack(), write_file()`

These functions can update **`file_data`**. Write code so that they automatically also update **`hash_data`** with the new hashes corresponding to the new data.

`read_file()`

Write code so that this function automatically verifies the hashes of blocks that it reads from **`file_data`**. This verification must proceed recursively up the hash tree (i.e. check that parent hash values are correct all the way to the root). Assume that the root hash value is correct. If you determine that a verification check has failed, **`read_file()`** must return 3.

Part 3

Implement this part in **myfuse.c**. This file has been included in the scaffold provided. You will need to call your library functions from Part 1 and 2. You will need to write a **main()** function for this part.

How do filesystems work? How does FUSE work?

In Linux, as you will be familiar with, all files are presented under a directory tree rooted at **/**. Different devices, such as a USB stick, contain filesystems. These filesystems are mounted (i.e. made accessible) under the root directory tree. The root directory tree itself is also a mounted filesystem, generally stored on the internal computer hard drive where the operating system resides.

For example, when you mount a USB, its contents become accessible under a directory known as a mountpoint, such as **/media/usb0**. (The mountpoint directory is generally used exclusively for mounting, but it could contain files which can be mounted over. This is beyond the scope of this assignment, as are many further complex details of **mount**).

When you perform any operation on a file or directory in Linux, such as **write()**, the system call is processed by the kernel. For most filesystems, such as FAT which is often used on USB storage devices, this operation reaches the relevant filesystem driver, which is responsible for creating the FAT data structures and issuing the relevant low-level USB commands to store raw data onto and read raw data from the USB device.

If the system call concerns a file that resides on a FUSE filesystem, the kernel passes the operation back to userspace, where it is processed by libfuse. FUSE filesystems are simply programs linked to libfuse, which takes care of receiving the kernel instruction and subsequently invokes callback functions that your program implements. These functions receive specific parameters and are expected to return data in a specified format, depending on what your filesystem is expected to achieve. The results are passed back into the kernel and back to the program that originally issued the system call on the file in question.

Example of FUSE callback function

An example of a FUSE callback function is:

```
int open(const char *, struct fuse_file_info *);
```

This directly corresponds to the **open()** system call. When **open()** is called on a file existing under a FUSE filsystem mountpoint, the FUSE filesystem program eventually has this function called, with the first parameter populated with the path of the target file relative to the mount point (and a **struct fuse_file_info** populated with various information/flags). It is important to note that what happens at this point is entirely dependent on what the goal of the filesystem is. For example, if this FUSE filesystem is implementing a network shared filesystem, then there would be code at this point implementing network connections and protocols. Regardless, libfuse expects a return code from the function - which is passed back to the kernel and the calling processas the **errno** which you should be familiar with. It is also important to note that the "file" in the FUSE filesystem may not be a real "file" at all. It could even be dynamically generated content on the fly; it depends entirely

on what the FUSE program implements. You should appreciate that "files" and "folders" are just interfaces into the operating system, and that they are simply an abstraction that is presented to the user.

What do I need to do?

Refer to the scaffold code **myfuse.c**. It contains basic code to get you started and indicates where to implement new code.

Ed does not support FUSE, so download the code to your own Linux computer, virtual machine, or lab computer. Make sure you have installed the required packages.

FUSE filesystems are just normal C programs linked to libfuse. The scaffold code can be compiled and run. Compile with:

```
gcc -o myfuse -std=gnu11 -lfuse myfuse.c
```

The key component required is a `struct fuse_operations`. This contains a set of function pointers that point to the callback functions you implement (such as **write()** described above in the example).

You pass `struct fuse_operations` into **fuse_main**, which stays running, accepting filesystem operations from the kernel and calling your callback functions. **fuse_main** also automatically interprets FUSE-specific command line options for you.

You can run (mount) your FUSE filesystem to a certain **mountpoint** with:

```
./myfuse -d mountpoint
```

Create a directory to serve as your mountpoint. The **-d** option is automatically interpreted by **fuse_main** to keep your program in the foreground and also provide debug output. Pass the **--help** option to see more usage information.

You can access your FUSE filesystem under **mountpoint** using standard Linux filesystem tools such as **ls**, or any other program of your choosing. If you have run your program with **-d**, you can see the callback functions being called.

You can explore the basic filesystem mounted by the scaffold code. Note how the output of commands such as **ls** corresponds to what is implemented.

To unmount your filesystem, you can use **fusermount -u mountpoint**, or simply terminate the program.

Your task is to implement the following callback functions as if they were operating on your virtual filesystem. Make the appropriate calls to your virtual filesystem library functions in Part 1 and 2, so that they are presented through the FUSE interface.

Implement: **getattr**, **unlink**, **rename**, **truncate**, **open**, **read**, **write**, **release**, **readdir**, **init**, **destroy**, **create**

In software development, you will be often required to program against third party APIs and libraries. To practice this skill, for this Part 3 of the assignment you will need to refer to the provided documentation for FUSE. This will tell you what the intent of each callback function is, and what data structures and return values it expects. You will also need to refer extensively to Linux system call manpages. There are also many other Internet resources that document FUSE. Be careful to avoid

academic honesty issues by citing your sources.

The documentation for FUSE is provided as a zip file in Ed resources (under "Assignment"). A good place to start is **`fuse_8h.html`** (open with a web browser) and the documentation for **`fuse_main_real()`** - from there, jump to the documentation for **`fuse_main()`** and **`fuse_operations`**. We are using what is referred to as the "high-level API", you do not need to refer to any functions found in **`fuse_lowlevel.h`**. Warning: there is online official documentation for FUSE at https://libfuse.github.io/doxygen/index.html. Do not use this - it is automatically generated for FUSE version 3.x, which is incompatible with version 2.9 which you are using for the assignment.

Notes and Hints

You do not need to implement any other callback functions that FUSE supports other than those specified above.

The only field of **`struct fuse_file_info`** you need to read or write is **`fh`**. We will not test your code on other functionality such as **`direct_io`**.

You do not need to read or write any part of **`struct fuse_conn_info`**.

Whilst your filesystem does not support directories, you need to implement a basic **`readdir`** to list the files that it contains.

You can just pass 0 as the **`off_t`** argument to **`fuse_fill_dir_t`**.

When implementing functions, there will often be options to return data that is not applicable to your filesystem, such as the owner of files. You can return any sensible value here - it will not be checked. However, data that your filesystem has, such as the size of files, must be returned correctly.

Certain functions do not correspond directly to system calls; for instance, **`init`** and **`destroy`** are there should you wish to set up private data structures, etc.

You need to carefully read documentation to determine what is the appropriate **`errno`** to return for errors. For example, if the user specifies a non-existent file to open, you should return **`ENOENT`**.

If your filesystem encounters a verification error in the Merkle hash tree when reading a file, return **`EIO`**.

A number of data structures in FUSE are Linux-wide data structures and will not be found in FUSE documentation alone. For example, you can find the documentation on **`struct stat`** in the **`stat(2)`** manpage.

Part 4

There is a manual marking component of this assignment. This will assess your approach to memory management, process/threading management and synchronisation. It will also assess the style, layout and readability of your code.

You are also required to write test cases for Part 1 and 2 only. How comprehensively your test cases cover your submission and the assignment description will be assessed as part of your manual mark. Implement **`runtest.c`** in the scaffold so that it tests all your functions for Part 1 and 2. Provide your own input and output test files (**`file_data`**, **`hash_data`**, **`directory_table`** files - you must also test their contents after functions are executed).

There is minimal code in the scaffold to get you started. The way that you test your code is flexible. The only requirement is that **`runtest.c`** runs all your tests and provides some indication as to the success or failure of each test. Make sure you test all branches/source code lines in your tests. This coverage percentage does not directly translate into a mark but will be considered by your marker.

Whilst test cases only need to be written to cover Parts 1 and 2, note that the other components of manual marking will assess all components of your code, including Part 3 and your testing code itself.

Submission and Mark Breakdown

Submit your assignment on Ed. It must produce no compilation errors. Some test cases for correctness will automatically be performed. Note that testing code will not solely check the output of your program or its functions. The data that you write to your virtual filesystem files must also be correct.

Some test cases will be made available for download.

Your code will be compiled with the following options:

```
gcc -O0 -std=gnu11 -lm -lpthread -lfuse
```

The mark breakdown of this assignment follows (**15 marks total**).

- **7 marks** (total) for correctness, assessed by automatic test cases on Ed. Some test cases will be hidden and will not be available before or after the deadline. Correctness will be assessed for Parts 1, 2 and 3.
  - **3 marks** of these 7 correctness marks will be due for assessment at 11:59 pm AEST, Sunday May 12th as the Milestone. These will correspond to a subset of the test cases for Part 1 only. Test cases with names starting with `milestone` will count towards these marks. This will be assessed against your last submission before this Milestone time.
  - There will be no opportunity to regain these 3 marks after the Milestone.
- **3 marks** for performance. This will be assessed for Parts 1 and 2 only. You will receive at least 2 marks for a submission faster than a basic parallel solution. You will receive an additional 1 mark for a submission faster than a benchmark parallel solution. You must pass all correctness test cases before your submission will be assessed for performance.
- **5 marks** from manual marking. After the final assignment deadline, teaching staff will assess your submission as described in Part 4 and provide feedback. It will take into account the test cases that you write for your code. Manual marking of your code will be for your entire assignment. Your test cases only need to be written for Parts 1 and 2.

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment problem description or if your code is unnecessarily or deliberately obfuscated.

Academic Declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.