CSCI 4061: Introduction to Operating Systems, Fall 2023
Project #1: Merkle Tree Implementation

Instructor: Abhishek Chandra

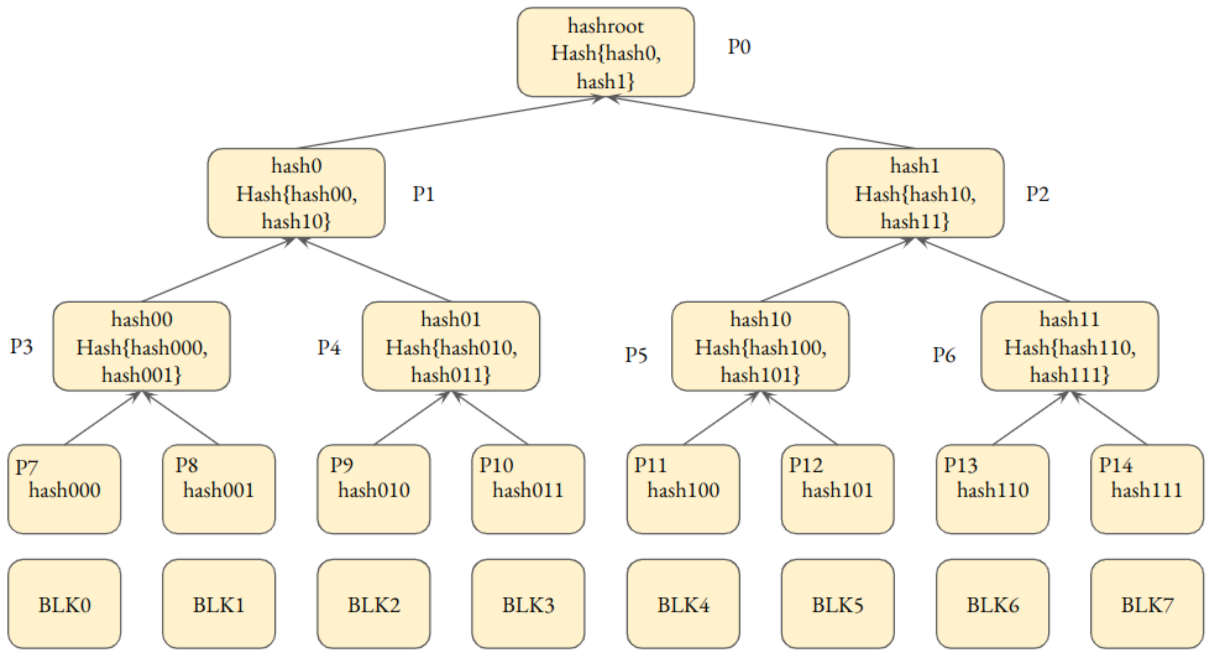
Intermediate submission due: **11:59pm (CDT), Oct. 4, 2023**
Final submission due: **11:59pm (CDT), Oct. 11, 2023**

1. Background

A Merkle tree is a hash tree in which every "leaf" (node) is labeled with the cryptographic hash of a data block, and every node that is not a leaf is labeled with the cryptographic hash of the labels of its child nodes. Merkle trees allow for efficient data verification. Demonstrating that a leaf node is a part of a given binary Merkle tree is logarithmic with respect to the number of leaf nodes in the tree. Merkle trees are typically used to ensure that files duplicated across different computers are the same. Hash trees, such as Merkle trees, are seen in software systems such as Bitcoin, Git, and Cassandra.

In this project, you'll be using the functions fork(), exec(), and wait() to parallelize building a binary Merkle tree. The data blocks hashed by the tree will be a single file split up into many "chunks". You'll use basic file I/O functions (fopen(), fread(), fwrite()) to split up this file and to read/write data to intermediate files throughout the project.

2. Project Overview



hashroot Hash{hash0, hash1}
P0
hash0 Hash{hash00, hash10}
P1
hash1 Hash{hash10, hash11}
P2
P3
hash00 Hash{hash000, hash001}
P4
hash01 Hash{hash010, hash011}
P5
hash10 Hash{hash100, hash101}
P6
hash11 Hash{hash110, hash111}
P7 hash000
P8 hash001
P9 hash010
P10 hash011
P11 hash100
P12 hash101
P13 hash110
P14 hash111
BLK0
BLK1
BLK2
BLK3
BLK4
BLK5
BLK6
BLK7

Figure 1: Example Merkle tree

In general, the iterative process for building a binary Merkle tree looks something like this:

1. Divide the file data evenly into fixed size chunks
2. Hash the data blocks to form the leaf nodes of the tree
3. Combine leaf nodes two at a time and hash their combined hash values to get the nodes at the next level up in the tree (towards the root)
4. Repeat step 3 until you get to a single root hash node

See Figure 1 above for what a completed Merkle tree looks like.

For this project, you'll be starting "from the top" when building the binary Merkle tree. Starting with the root process (P0 in Figure 1, root node in the Merkle tree), each process (non-leaf nodes in Merkle tree) will create two child processes to compute the hash of its left subtree and right subtree. It will then use these hash values to create its own hash. This will occur recursively until the leaf nodes of the Merkle tree are reached. The leaf processes (P7 - P14 in Figure 1) will simply hash their associated data blocks and exit.

3. Coding Details

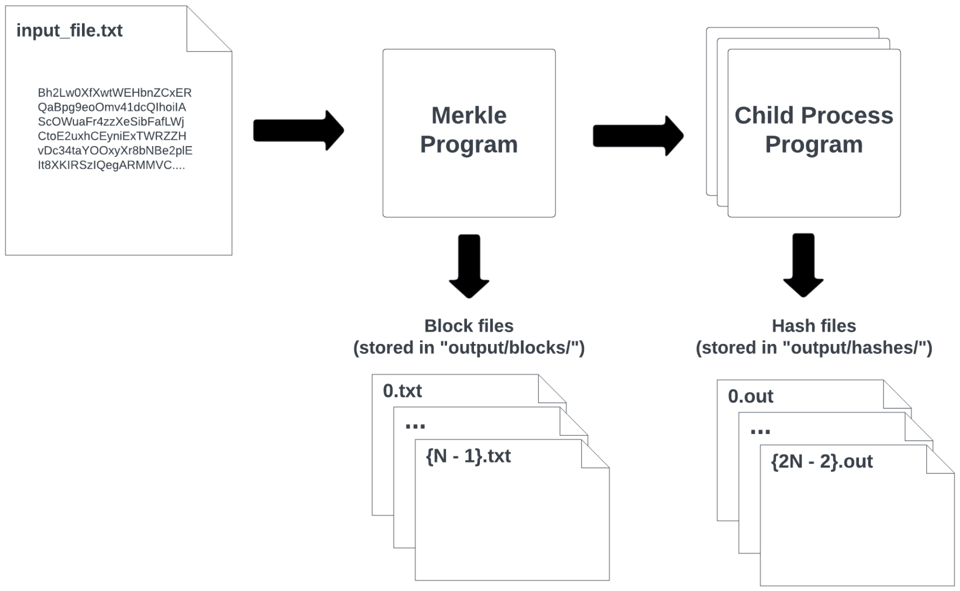


Figure 2: General flow of program and data

3.1 Partitioning Input Data into N files

The input to the Merkle program (src/merkle.c) will be a text file (input_file.txt in the example above) and N. The text file is the data you'll be constructing a Merkle tree for and N is the number of blocks (files) that you'll split the input file into. N will also be the number of leaf nodes in the Merkle tree (so N should be a power of 2). After validating the input, the first step is to implement partition_file_data(), which resides in src/utils.c. This function splits the input text file into N roughly equal sized files and places them in the folder "output/blocks/". The formula for how much data each file should contain is:

- floor(size(input.txt) / N) for 0.txt, 1.txt, …, {N-2}.txt
- floor(size(input.txt) / N) + size(input.txt) % N for {N-1}.txt

See Figure 3 below for an example of how this function partitions the data into the files. The files should be named 0.txt, 1.txt, ..., {N-1}.txt.

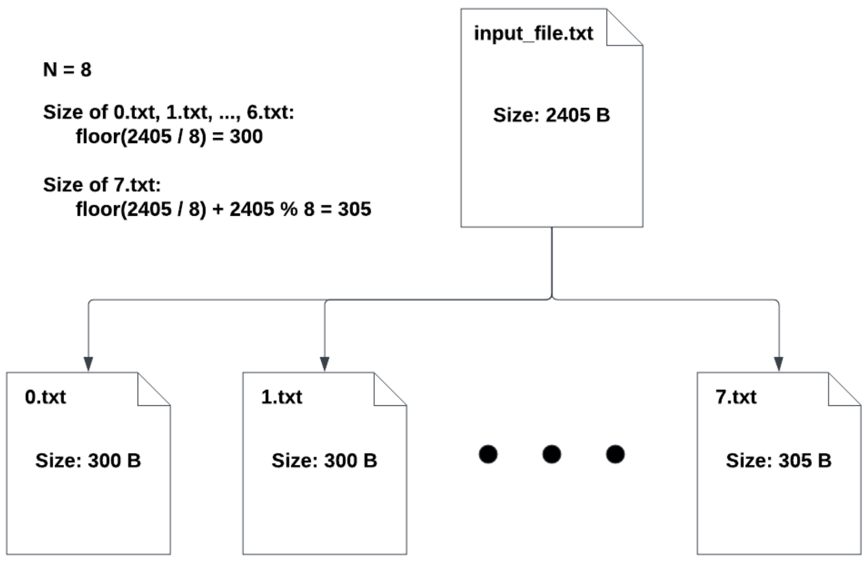


Figure 3: Example splitting of the input text file

Note: The bytes in the generated files should maintain the order from input_file.txt (i.e. concatenating the files 0.txt, 1.txt, …, {N-1}.txt in order should give you back the file input_file.txt)

3.2 Creating Merkle Process Tree and Hashing Files

After creating the N block files, you'll spawn the first child process (root of Merkle tree). Each child process that gets spawned will have a unique ID (**Note: This ID is not the pid of the actual process**). The first child process will have an ID of 0 since it is the root. Look at Figure 4 (in the top left) below to see how subsequent child process IDs are determined.

The child process that gets created will run the executable associated with src/child_process.c. This program will be responsible for generating the hashes in the Merkle tree (See Figure 2) and outputting them to files in the folder "output/hashes/" with the name <ID>.out, where ID is the ID of the node associated with the hash value. Thus the "output/hashes/" folder should have 2N - 1 files after the Merkle tree has been fully constructed. This program will handle two cases:

1. Current process is a leaf process
2. Current process is not a leaf process

3.2.1 Leaf Process

If the current process is a leaf process, the program should hash a block file and write that hash to its hashes file ("output/hashes/<leaf_ID>.out"). The block file that the leaf process should hash will be determined by its ID. The furthest left leaf node (ID = N-1) should hash 0.txt, next furthest left leaf node (ID = N) should hash 1.txt, and so on until {N-1}.txt is hashed by the furthest right leaf node (ID = 2N - 2). **The leaf process should exit the program after writing this data to the hash file. If you fail to do this, you'll have infinite recursion and your computer will feel the pain (trust me 😔).**

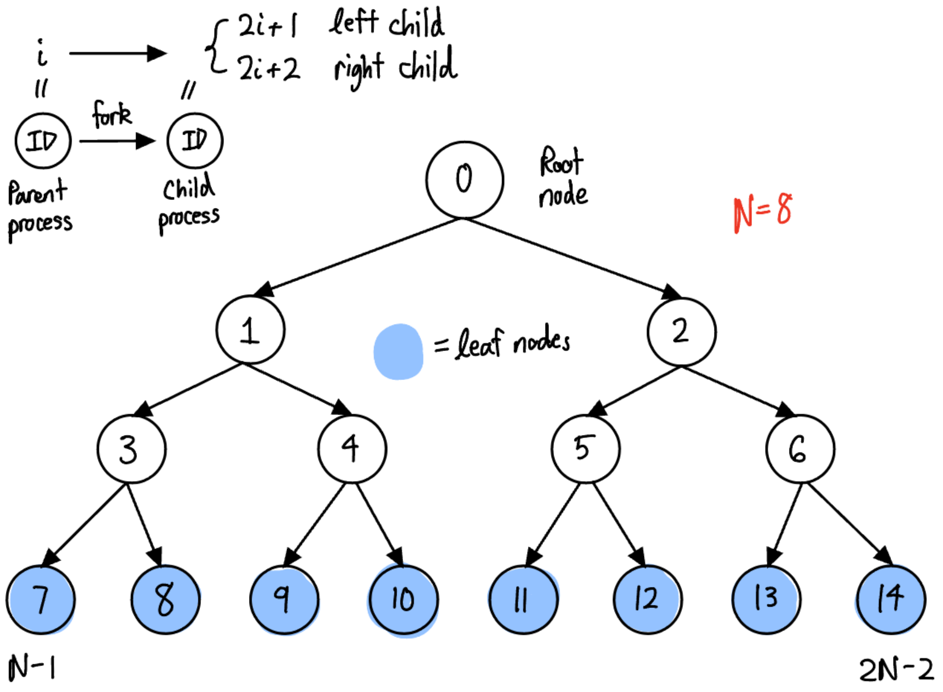


Figure 4: Merkle tree process IDs for N=8

3.2.2 Non-Leaf Process

If the current process is not a leaf process, the program should spawn two child processes and have them run ./child_process using exec() with the new child ID parameter (See Figure 4 above for determining new child ID parameter). The current process (parent) should then wait() for these child processes to complete. After the child processes complete, the current process can read the hash values from the files its children output and write the combined hash to its own output/hashes/<ID>.out file.

3.3 Visualizing Merkle Tree

After your original child process finishes in src/merkle.c, you now have all of the Merkle tree hashes in the "output/hashes/" folder. Using these files, we have written code that will output the hashes to a file "output/visualization.txt" to show the tree structure and hash values. Using the example from **Figure 4**, It will look something like this:

— — — — — — — — — — — — — — — —

```
Node 0: 3a1e7b9c0f2d84e7
        Node 1: f6b8e1f335a39b6e
                Node 3: 7a4d22c4df34583a
                        Node 7: 3a14a0a16d207c2d
                        Node 8: e78754cc738b8ce1
                Node 4: 72c2fb8c1e9d8033
                        Node 9: edf53e3a4c9d207c
                        Node 10:  c4f4b13aa504f62f
        Node 2: d6efce83f8ad269e
                Node 5: 3b38e7b3ff07bf08
                        Node 11: 0f23471a6e0b5e8a.
                        Node 12: a501a4d1c742a2d9
                Node 6: 43e86193211c1c13
                        Node 13: 1b1a7d46eb586da5u
                        Node 14: aa09d102b4f12caf
— — — — — — — — — — — — — — — —
```

The expected visualization files for the test cases will be located in the “expected/” folder. See 6. Testing for info on how to easily compare your files to the expected ones.

4. Compilation Instructions

You can create all of the necessary executable files with

Command Line

```
$ make all
```

If you want to see if your code works for the intermediate solution, you can use

Command Line

```
$ make inter
```

This simply avoids printing out the visualization.txt file. This Makefile target uses a debug macro.

Running the program can be accomplished with

Command Line

```
$ ./merkle input/T1.txt 8
```

where input/T1.txt is the file containing the data to be hashed and 8 is the number of data blocks (files) to split the input file into.

The data blocks (files) will go in “output/blocks/” and the node hashes will go in “output/hashes/”. The files in output/hashes/ are named after their node’s ID. Remember that the root node has an ID of 0 and each node in the tree (except for the leaf nodes) has children with IDs:

- Left: (2 * i + 1)
- Right: (2 * i + 2)

where i is the ID of the parent node.

5. Project Folder Structure

Please strictly conform with the folder structure that is provided to you. **Your conformance will be graded.**

Project structure	**Contents (initial/required contents[1])**
include/	.h header files (hash.h, print_tree.h, sha256.h, utils.h)
lib/	.o library files (hash.o, print_tree.o, sha256.o)
src/	.c source files (merkle.c, child_process.c, utils.c)
input/	.txt input files (T1.txt, T2.txt, T3.txt)
expected/	.txt visualization files (T1.txt, T2.txt, T3.txt)
*output/	program results
Makefile	file containing build information and used for testing/compiling
README.md	file containing info outlined in 8. Submission Details
*merkle	executable file created during compilation
*child_process	executable file created during compilation

*** These files should not be included in your submission**

The files hash.o, print_tree.o, and sha256.o are precompiled for you and should not be modified/deleted. They were compiled on a Linux CSELabs machine and are platform dependent, so you will have to run your code on a CSELabs machine or equivalent Linux computer.

[1] This content is required at minimum, but adding additional content is OK as long as it doesn’t break the existing code.

6. Testing

The Makefile contains three tests (T1, T2, T3). After running “make all” or “make inter”, you can run these test cases like such:

Command Line

```
$ make t1
```

They will use the associated test input files (T1.txt, T2.txt, T3.txt) in the input folder to generate output with your code. The expected “output/visualization.txt” for these tests is in the expected/ folder. You can run a comparison with the “diff” command to see if your output matches the expected output:

Command Line

```
$ make t1
$ diff output/visualization.txt expected/T1.txt
```

The diff command will tell you what specific lines don’t match between the two files, so you can debug specific nodes that don’t have the correct hash. You can also check if the hashes of your data are correct by using SHA-256 calculators online (https://xorbin.com/tools/sha256-hash-calculator).

Use these tests to check your work, but know that they are not exhaustive and we will be using additional tests for grading. Feel free to add your own tests to the Makefile.

7. Assumptions / Notes

- 128B <= Input file size <= 128MB
- 1 <= N <= 128
- Input file size >= N
- The length of a file path will be < 1024 characters (bytes)
  - Use PATH_MAX when creating char arrays for file names
- All processes have access to the output/ folder
- You should be using fork(), wait(), and exec() to create the Merkle tree using a process tree
- If you are dynamically allocating memory, make sure to free it

8. Submission Details

There will be two submission periods. The intermediate submission is due 1 week before the final submission deadline. The first submission is mainly intended to make sure you are on pace to finish the project on time. The final submission is due ~2 weeks after the project is released.

8.1 Intermediate Submission

For the Intermediate submission, your task is to implement the partition_file_data() function in src/utils.c and come up with a plan on how you are going to implement the rest of the project.

One student from each group should upload a **.zip file** to Gradescope containing all of your project files. We'll be primarily focusing on utils.c and your README, which should contain the following information:

- Project group number
- Group member names and x500s
- The name of the CSELabs computer that you tested your code on
    - e.g. csel-kh1250-01.cselabs.umn.edu
- Any changes you made to the Makefile or existing files that would affect grading
- Plan outlining individual contributions for each member of your group
- Plan on how you are going to implement the process tree component of creating the Merkle tree (high-level pseudocode would be acceptable/preferred for this part)

The member of the group who uploads the .zip file to Gradescope should add the other members to their group after submitting.

8.2 Final Submission

One student from each group should upload a **.zip file** to Gradescope containing all of the project files. The README should include the following details:

- Project group number
- Group member names and x500s
- The name of the CSELabs computer that you tested your code on
    - e.g. csel-kh1250-01.cselabs.umn.edu
- Members' individual contributions
- Any changes you made to the Makefile or existing files that would affect grading
- Any assumptions that you made that weren't outlined in 7. Assumptions / Notes
- How you designed your program for creating the process tree (again, high-level pseudocode would be acceptable/preferred for this part)
    - If your original design (intermediate submission) was different, explain how it changed
- Any code that you used AI helper tools to write with a clear justification and explanation of the code (Please see below for the AI tools acceptable use policy)

The member of the group who uploads the .zip file to Gradescope should add the other members to their group after submitting.

Your project folder should include all of the folders that were in the original template. You can add additional files to those folders and edit the Makefile, **but make sure everything still works**. Before submitting your final project, run "make clean" to remove any existing output/ data and manually remove any erroneous files.

9. Reiteration of AI Tool Policy

Artificial intelligence (AI) language models, such as ChatGPT, may be used in a **limited manner for programming assignments** with appropriate attribution and citation. For programming assignments, you may use AI tools to help with some basic helper function code (similar to library functions). You must not use these tools to design your solution, or to generate a significant part of your assignment code. You must design and implement the code yourself. You must clearly identify parts of the code that you used AI tools for, providing a justification for doing so. You must understand such code and be prepared to explain its functionality. Note that the goal of these assignments is to provide you with a deeper and hands-on understanding of the concepts you learn in the class, and not merely to produce "something that works".

If you are in doubt as to whether you are using AI language models appropriately in this course, I encourage you to discuss your situation with me. Examples of citing AI language models are available at: libguides.umn.edu/chatgpt. You are responsible for fact checking statements and correctness of code composed by AI language models.

For this assignment: The use of AI tools for generating code related to the primary concepts being applied in the assignment, such as process tree management, process operations, and file I/O, **is prohibited**.

10. Rubric (tentative)

- [10%] README
- [10%] Intermediate submission
- [10%] Coding style: indentations, readability, comments where appropriate
- [50%] Test cases
- [10%] Correct use of fork(), wait(), and exec()
- [10%] Error handling — should handle system call errors and terminate gracefully

Additional notes:

- We will use the GCC version installed on the CSELabs machines to compile your code. Make sure your code compiles and runs on CSELabs.
- A list of CSELabs machines can be found at https://cse.umn.edu/cseit/classrooms-labs
    - Try to stick with the Keller Hall computers since those are what we'll use to test your code
- Helpful GDB manual. From Huang: GDB Tutorial From Kauffman: Quick Guide to gdb