CSCI 4061: Introduction to Operating Systems, Fall 2023
Project #2: Data Deduplication

Instructor: Abhishek Chandra

Intermediate submission due: **11:59pm (CDT), Oct. 20, 2023**

Final submission due: **11:59pm (CDT), Oct. 27, 2023**

1. Background

Data deduplication is a specialized method of data compression aimed at eliminating duplicate copies of repeating data in storage. Instead of storing identical data instances multiple times, deduplication retains a single copy of the data and references to that original copy for subsequent identical data instances. It is widely utilized in backup systems to minimize storage of repetitive data, cloud storage platforms to optimize space when multiple users upload identical files, virtualization environments to reduce redundancy in virtual machine images, and email systems to store single instances of common content or attachments. This technique enhances storage efficiency and reduces data transfer requirements across various domains.

In this project, you will leverage the process functions fork(), exec() and wait() along with file, directory, and pipe operations, to carry out data deduplication within a file system hierarchy. This will involve finding and removing duplicate files and replacing them with symbolic (soft) links. You will use directory syscalls opendir(), readdir() and closedir() for traversing directories, system-level file I/O read() and write() as well as pipe() for data transfer between processes, symlink() for creating symbolic links and dup2() for redirection.

2. Project Overview

For this project, you'll construct a hierarchical process tree, where processes are either associated with files (leaf processes) or folders (non-leaf processes). Each process creates a set of pipes for communication with its children. Non-leaf processes pass directory entry paths to their children and collect filenames and file hashes from their children, while leaf processes compute file hash values and relay them, with file paths, to their parent processes. The root process then identifies and handles file duplicates, replacing them with symbolic links to the original. Additionally, the root process will read these symbolic links, redirect standard output to a new file, write the full path as well as the content of the symbolic link file to text file.

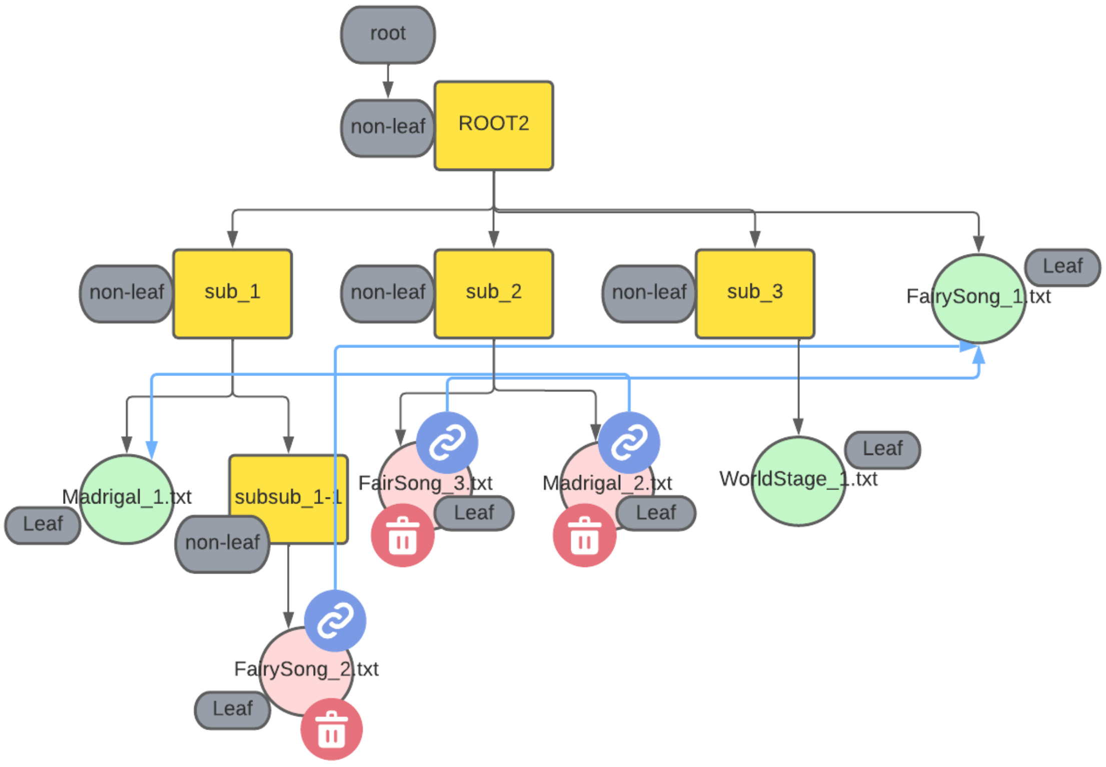


Figure 1: Directory Structure

2.1 Process Tree Creation

- **Types of Processes**:
    - **Leaf Process**: Associated with a file. This process will not spawn any child processes.
    - **Non-leaf Process**: Associated with a folder. This process is responsible for spawning child processes for each item (folder or file) within the directory it represents.
    - **Root Process**:This process is responsible for spawning one child (a non-leaf process), aggregating all file paths and hashes, deleting duplicates, and creating symbolic links.
- **Example**:
    - As shown in Figure 1, given a root directory ./root_directories/root with three folders:
        - The root process creates one non-leaf process (P1) for the root folder.
        - The non-leaf process P1 browses the root folder which contains 3 subfolders. So 3 non-leaf processes are created as children of P1.

        - For `sub_1` which contains `Madrigal_1.txt file` and `subsub_1-1 folder`:
            - The process for `sub_1` spawns:
                - A leaf process for `Madrigal_1.txt`.
                - Another non-leaf process for `subsub_1-1 folder` which further spawns a leaf process for `FairSong_2..txt`

2.2 Inter-process Communication

Every process will establish a pipe to communicate with each child process. For example, if a non-leaf process creates three children, it will construct three distinct pipes. Here's the workflow of the process tree creation and IPC:

- **Root Process:**
    - Parse the command line arguments
    - Create the first non-leaf process to start the traversing process.
- **Non-leaf Process:**
    - Create a pipe for each child process.
    - Pass the complete path of each item in the current folder to its associated child process as an argument.
    - Collect all children's pipe messages, relaying the leaves' filenames as well as file hashes for all the files within its subtree. Send this information to its parent process through the pipe shared with its parent.
- **Leaf Process:**
    - Compute the hash value of the file's content.
    - Send this hash value, along with the file's full path, to the parent process through the pipe.
- **Workflow**:
    - As shown in Figure 1, the workflow is as follows:
        - Root process forks the first non-leaf process (P1) and creates the first pipe between the root and P1. It passes its full path and pipe's write-end as arguments to P1.
        - Each non-leaf process recursively creates non-leaf processes and leaf-processes according to the directory structure. For each pair of parent and child processes, one pipe should be created and the parent shall pass its full path and pipe's write-end as arguments to the child.
        - Each leaf-process calculates the hash of the leaf file content and writes it to the write-end of the pipe shared with its parent.
        - The non-leaf process shall wait for all its children and aggregate ALL pipe messages through the read_ends of ALL pipes shared with its children.

2.3 Duplicate File Handling and Symbolic Links

Using the hash values and file paths sent to the root process:

- The root process identifies unique files and pinpoints duplicates.
- It deletes the duplicate files, replacing them with symbolic links pointing to the remaining copy.
- Note: files are identified by the hashes of their contents. We numbered the files with the same content to help with debugging. For example, FairSong_1.txt and FairSong_2.txt should be detected as duplicate files. The file with a **smaller** index should be retained. Thus, FairSong_1.txt is the retained copy and FairSong_2.txt should be deleted and replaced with a symbolic link

2.4 Redirection

The root process performs the following steps:

- Reads the content of symbolic links which is the full path of the file that the symbolic link points to.
- Redirects and writes this full path as well as the content of the symbolic link to a new file within the root directory. For example, the output for each symbolic link should be like *./root_directories/root1/sub_2/Madrigal_2.txt →* *./root_directories/root1/sub_1/Madrigal_1.txt*, where Madrigal_2.txt is the symbolic link while Madrigal_1.txt is the file that Madrigal_2.txt points to.

3. Coding Details

3.1 Creating a process tree

The process tree starts with the root process. The root process will take the name of the target directory as input. Because the given directory is a non-leaf folder, the root process will create the first non-leaf process. It should first prepare the pipe for further communication, then fork() the non-leaf process and use the exec() function to execute the non-leaf program, passing the arguments. The read end of the pipe should be used by the parent process to aggregate the file hashes, and the write end of the pipe should be used by the child process to enable communication with the parent.

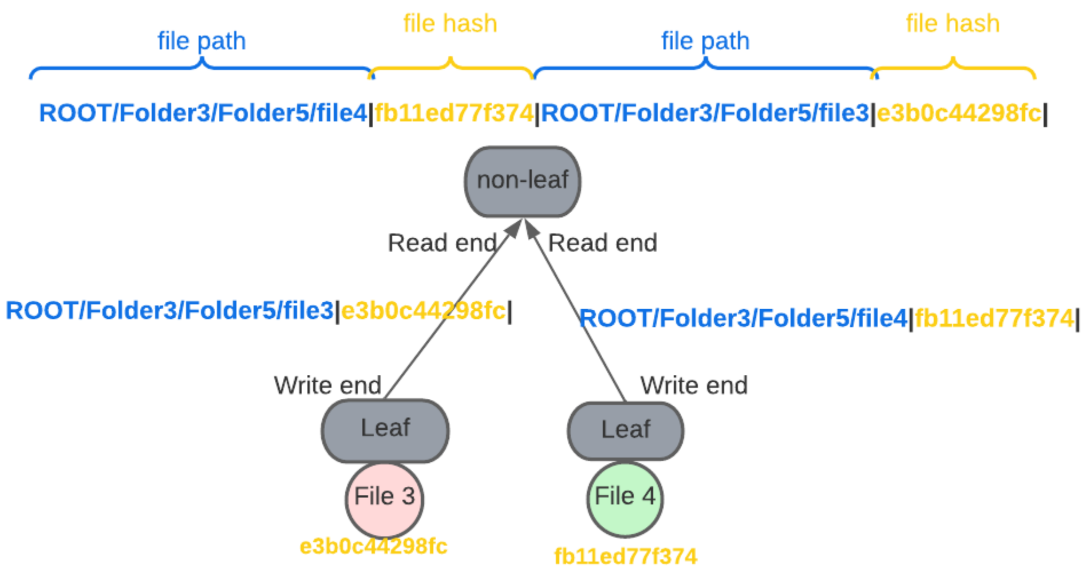

file path
file hash
file path
file hash
ROOT/Folder3/Folder5/file4|fb11ed77f374|ROOT/Folder3/Folder5/file3|e3b0c44298fc|
non-leaf
Read end
Read end
ROOT/Folder3/Folder5/file3|e3b0c44298fc|
ROOT/Folder3/Folder5/file4|fb11ed77f374|
Write end
Write end
Leaf
Leaf
File 3
File 4
e3b0c44298fc
fb11ed77f374

Figure 2: Illustration of Process Communication

3.1.1 Non-leaf Process

Each non-leaf process takes two arguments: the path to the target directory and the write end (file descriptor number) of the pipe. Each non-leaf process should iterate through its given directory. There are two cases during iteration:

1) The current entry is a directory -> Another non-leaf process should be created.
2) The current entry is a file -> A leaf process should be created.

In both scenarios, like the root process, it should begin by setting up the pipe for subsequent communication, followed by forking the child process and employing the exec() function to execute a program (non-leaf or leaf depending on the type of child process), transmitting the arguments to the child process.

3.1.2 Leaf Process

The leaf process is responsible for obtaining the file hash and communicating with its parent through the write end of the pipe. Each file is uniquely identified by its hash of contents. Similar to the PA1, we provide the function hash_data_block() to calculate the hash of file contents. In particular, the format of the content written to the write end of the pipe is **filename|hash|**. Please note that your intermediate submission is to implement the leaf process functionality.

3.2 Locating the duplicates

After traversing through the given root directory, the root process should construct a long string that contains all file names and file hashes obtained through the process tree. This root process should have collected a long string that contains all leaf file names as well as their hashes. You should partition the string to map the file name and file hashes by using the helper function parse_hash() that was provided in util.c. Then, you can iterate through all hashes to locate the duplicate hashes and their file names according to the hash.

Please note that all given file names end with numbers. You should always retain the file with the smallest file number.

3.3 Create symbolic links and delete duplicates

Delete all the duplicate files and create a symbolic link for each of them pointing to the unique file. You should always retain the file with the smallest file number.

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 test cases, you can use

Command Line

```
$ make inter
```

Running the program with various root directories can be accomplished with

Command Line

```
$ ./root_process root_directories/root1
```

Or you can use the make to run with a specific directory:

Command Line

```
$ make root1
```

Or you can use the make to run with all test cases:

```
Command Line
$ make final
```

5. Project Folder Structure

Please strictly conform to 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, sha256.h, utils.h)
lib/	.o library files (hash.o, sha256.o, utils.o)
src/	.c source files (root_process.c, noleaf_process.c, leaf_process.c)
root_directories/	Root directories used for testing
*output/	program results
expected/	expected output
Makefile	file containing build information and used for testing/compiling
README.md	file containing info outlined in 8. Submission Details
*root_process	executable file created during compilation
*leaf_process	executable file created during compilation
* nonleaf_process	executable file created during compilation

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

The files hash.o, sha256.o and utils.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.

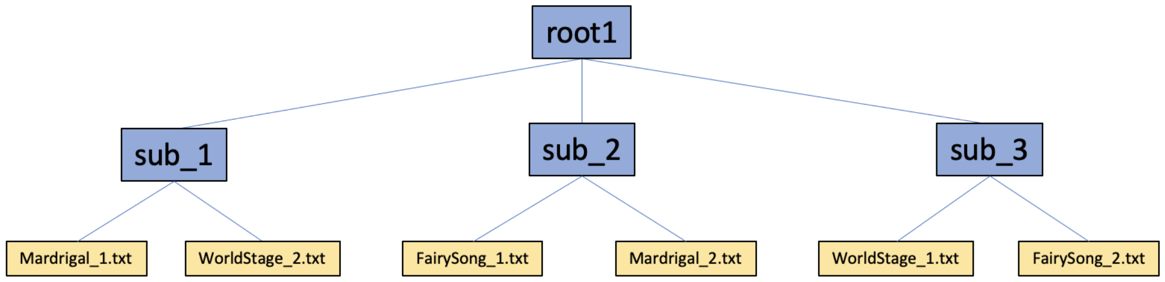
6. Testing

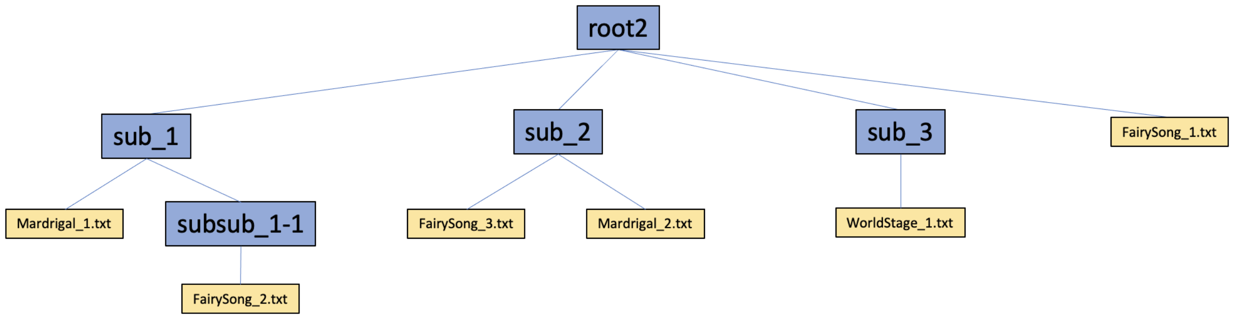
The Makefile contains three tests (root1, root2, root3). After running “make all”, you can run these test cases like such:

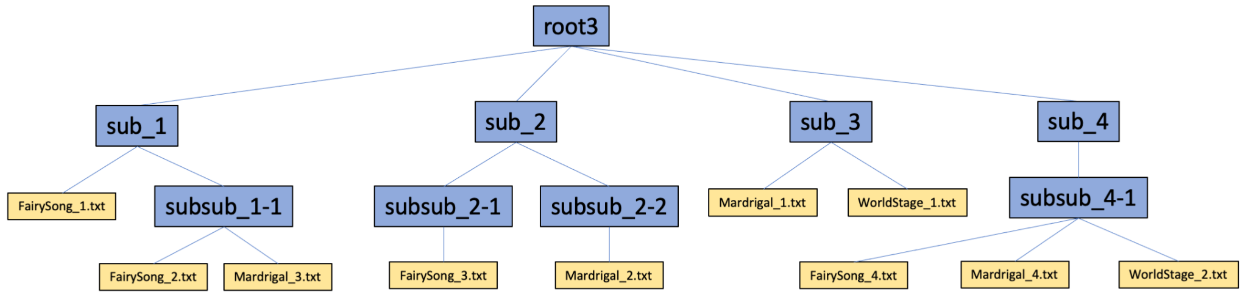
Command Line

```
$ make root1
```

The hierarchy of the 3 root directories are as below:







7. Assumptions / Notes

- Root directory is not empty
- 1<= Total number of files <= 10
- All processes have access to the root_directories/ folder
- You should be using fork(), wait(), and exec() to create a process tree
- If you are dynamically allocating memory, make sure to free it
- **Always initialize your buffer**
    - e.g. memset(buffer, 0, BUFSIZE);

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 leaf process functionality(leaf_process.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 *.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 construct the pipes and inter-process communication. (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. Only one member in a group should upload.

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. Only one member in a group should upload.

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, file I/O, directory operations, pipes and redirection, and links, **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(), pipe(), exec(), and write()/read()
- [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