Machine Problem 4 - File System

CSIE3310 - Operating Systems
National Taiwan University

Sky-blue correction: 2025/05/15 23:00
Orange correction: 2025/05/18 22:20

Total Points:	100
Release Date:	May 13
Due Date:	May 26, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	Fri. 14:00-16:00 before the due date, CSIE Building B04

Contents

1 Overview

In this MP, you will implement three common file system features in xv6: symbolic links, access control, and RAID simulation. Successfully completing this MP requires a solid understanding of xv6's file system interface. Therefore, you are strongly encouraged to carefully read *Chapter 8: File System* in the xv6 book before getting started.

2 Environment Setup

1. Download mp4.zip from NTU COOL. Unzip it, and enter.

```
#!/bin/bash
$ unzip mp4.zip
$ cd mp4
```

2. Pull the docker image from the Docker Hub.

```
$ docker pull ntuos/mp4
```

3. In the mp4 directory, use docker run to enter the container.

```
$ docker run -it -v "$(pwd)":/home/xv6 -w /home/xv6 ntuos/mp4
```

3 Problem I - Access Control & Symbolic links (40 pts)

3.1 Introduction

Access control is typically used to manage file access permissions for each user. Although xv6 does not implement user accounts, access control is still valuable for preventing unintended access to files—for example, accidental writes to executables. Additionally, while xv6 currently supports only hard links, symbolic links are often preferred because they can link to directories and span across different file systems.

In this problem, you will add two features to xv6: access control and symbolic links. Specifically, you will implement two commands: `symln` and `chmod`, along with two system calls: `sys_symlink` and `sys_chmod`. You will also modify the `ls` command to display file permissions correctly, and update the `open` system call to handle symbolic links appropriately.

3.2 Implementation Overview

Although symbolic links and access control are conceptually independent, integrating both features into xv6 requires careful consideration of their interactions. For example, the behavior of `chmod` with symbolic links, `ls` with symbolic links, and `ls` in the presence of restricted read permissions must be handled correctly. These interactions are detailed in the specification below.

In our test cases, we will call the following commands: `ls`, `chmod`, `symln` and the `open` system call to verify your implementation.

Note that we have already provided the `symlink` system call and the `symln` command for you. **However, you are responsible for adding the `chmod` system call and the `chmod` command to xv6 on your own.**

3.3 Specification

To help you better understand the expected behavior of your implementation, we have divided the specification of this problem into several parts.

3.3.1 Part I: Basic Usage of symln

- The format of the `symln` command is:

  ```
  $ symln old new
  ```

  Which create a symbolic link called `new` pointing to the file/directory `old`.
- Make sure that your symbolic link works on both files and directories.
- We have defined a new file type `T_SYMLINK` in `kernel/stat.h` which has the value 4. Your `ls` command should correctly show that a symbolic link has the file type 4. Here is an example:

```
$ mkdir orig
$ symln orig new
$ ls
...
orig            1 22 32 rw
new             4 23 9 rw
```

- You do not need to check whether the links form a cycle or not. It is guaranteed that symbolic links in test cases will not form any cycle.
- Your system call should return `0` on success and `-1` when error occurs, just like `link` or `unlink`.
- It is guaranteed that `old` always exists in the test cases. However, if `new` already exists, `symln` must fail by giving the following error message:

```
$ symln orig new
~~symln~~ symlink orig new: failed
```

- Whether `symln` successes or not should not be affected by the read/write permission of `target`.

3.3.2 Part II: Basic Usage of `chmod` and Access Control

- You only have to implement read and write permission.
- The format of `chmod` command is:

```
$ chmod [-R] (+|-)(r|w|rw|wr) file_name|dir_name
```

  Where `-R` flag apply `chmod` recursively to each file and subdirectory, `+r` means adding the read permission, `-rw` means removing both the read and write permission, .etc. Note that `rw` is exactly the same as `wr`, it is listed for clarity, and your code should handle both cases.

- We do not set the format of the `chmod` system call, you are free to make your choice. We will call the `chmod` command in the test cases and we expect your `chmod` user program to call your self-designed `chmod` system call.
- Newly created files or directories (e.g. `mkdir`) have the default permission of both readable and writable (i.e. `rw`).
- It is guaranteed that `-R` only appears before permission, and only one file/directory will be given in a `chmod` command. Therefore, the following commands will NOT appear in our test cases: `chmod +w -R abc_dir`, `chmod -rw abc.txt def.sh`
- If the command format is incorrect, print the following error message:

```
$ chmod -R +o test_dir
Usage: chmod [-R] (+|-)(r|w|rw|wr) file_name|dir_name
```

- If the target file/directory does not exist, or cannot be opened due to lack of read permission, print the following error message:

```
$ chmod +w d1/d2/f
chmod: cannot chmod d1/d2/f
```

  Note: `chmod +w d` does not need the read permission of `d`, while `chmod +w d/f` does, because you have to read `d` to know if there is a `f` inside.

- **Special Consideration**: Try the following commands in your terminal (on your Ubuntu or macOS system, not in xv6):

```
$ mkdir testDir
$ touch testDir/a testDir/b
$ ls -al testDir  # check current permission
```

```
$ chmod -R 000 testDir
chmod: cannot read directory 'testDir': Permission denied
```

The `chmod` command fails because it first removes the read permission from `testDir`, and then attempts to descend into the directory to modify the permissions of `a` and `b`. However, since `testDir` is no longer readable, this operation is immediately blocked. Interestingly, the command would succeed if the execution order were reversed.

The reason most UNIX-like systems behave this way is due to the UNIX philosophy: favoring simplicity over convenience. Recursive operations like `chmod -R` typically follow a top-down traversal strategy. Now that you have the opportunity to implement your own version of `chmod`, you should handle this edge case more robustly.

Please ensure that your `chmod` implementation is not blocked in the following edge cases:

1. When the `-R` flag is present, the target directory has read permission, and `chmod` is used to recursively remove the `r` permission from it.
2. When the `-R` flag is present, the target directory lacks read permission, and `chmod` is used to recursively ~~add `r` or `w` permission~~ add `r` permission to it.

Hint: If you view the directory structure as a tree graph, in which traversal order(s) can you successfully perform `chmod -R -r` on a readable directory and `chmod -R +r` on an unreadable directory?

3.3.3 Part III: The Output of `ls`

- The output format of the `ls` command is:

```
[file_name]      [type] [inum] [size] [mode]
```

  In the `mode` field, `r` indicates read permission, `w` indicates write permission, and `-` represents the absence of a permission. The read permission (`r`) always appears before the write permission (`w`). Refer to the example in the Guidance section for more details.

  Note that our tests will only verify the correctness of the `file_name`, `type`, and `mode` fields. The `inum` and `size` fields may contain any number, but they must be included in the output.

- Listing a file or directory with `ls` requires that the target file or directory itself has read permission. However, listing the contents of a directory does not require any specific permissions on the files or subdirectories within it. For example, if `d` is a directory, then `ls d` requires `d` to be readable. The entry `d/f` will be shown as long as `d` has read permission, regardless of the permissions set on `d/f`.

- `ls` should output the following error message if the directory/file has no read permission:

```
$ ls f
ls: cannot open f
```

3.3.4 Part IV: About the Symbolic Links

- The permission of a symbolic link itself does not have any meaning and should always have `rw` permission.
- Applying `chmod` to a symbolic link always changes the permission of the target file/directory but not the link itself.
- Here we discuss how `ls` works on a symbolic link (Note: These behaviors might differ a little from the real UNIX kernel since we do not implement the execute permission mode):
  1. If `x` is a symbolic link to a file `f`, `ls x` should give the detail of x itself, not `f`.
  2. If `x` is a symbolic link to a directory `d`, `ls x` should list the content of `d`.
  3. If `x` is a symbolic link to a file `f`, and `f` does not have the `r` permission, `ls x` should not be affected. However, if `f` is <u>inside of any directory that has no `r` permission</u>, `ls x` should fail.
  4. If `x` is a symbolic link to a directory `d`, `ls x` should fail when `d` has no `r` permission or `d` is inside a directory that has no `r` permission.

3.3.5 Part V: `open` Mode Check

- We provide a new flag, `O_NOACCESS`, defined in `kernel/fcntl.h`. When the `open` system call is invoked with this flag:
  1. It should return a file descriptor (`fd`) for which the underlying file is neither readable nor writable.
  2. The file metadata must remain accessible via the `fstat` system call.
  3. If the specified path is a symbolic link, the link should not be followed. Otherwise, when `O_NOACCESS` is not set, symbolic links must always be followed.

  The `O_NOACCESS` flag is useful for retrieving metadata (e.g., via `stat`, `fstat`) on files that do not grant any read or write permission, such as when listing the contents of a directory using `ls`.
- Make sure that your `open` system call denies when the provided flags does not meet the file's/directory's permission. e.g. if `f` does not have the `r` permission, `open(f, O_RDONLY)` should fail.
- In our test cases, we will call your `open` system call with some mode flags. While you are not asked to validate the mode flags in `open`, we will follow the policy below when calling `open()` in our test cases.
  1. Exactly one from `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_NOACCESS` will be chosen.
  2. `O_TRUNC` will be used along with write permission.
  3. `O_NOACCCESS` will be used alone.
  4. A directory, or a symbolic link to a directory, will only be opened with either `O_RDONLY` or `O_NOACCESS`.

3.4 Guidance

Since we understand that this problem is quite complicated, we provide overall steps to help you complete this problem. Feel free to follow your own way if you have other ideas.

1. **Implement `symlink`**: We recommend you to first complete the basic logic of symbolic link, since it is rather simple compared to other tasks.
   - You should think about how and where to store the target file/directory in the symbolic link. Maybe the inode data block is a good idea.
2. **Add modes**: We have defined 3 mode constants in `kernel/stat.h`: `M_READ`, `M_WRITE`, and `M_ALL`. Add a new `mode` field in appropriate places to store mode as part of the metadata of a file.
   - Hint1: Consider adding `mode` in `struct inode`, `struct stat`...
   - Hint2: Can you arbitrarily modify the size of `struct inode`? Which field is a good trade-off to be replaced?
3. **Initialize modes**:
   - Go to `iupdate`, `ilock`, `stati` and `ialloc` in `kernel/fs.c` to set the mode field correctly.
   - Go to `mkfs/mkfs.c` to modify `ialloc` to initialize the mode of each file. We encourage you assign `r-` mode to executables and `rw` to other files to follow the convention, but we will not check this in our test cases.
4. **Modify `ls`**: Go to `user/ls.c` to make sure that `ls` command shows the correct permissions at the end of each line:

```
$ ls test1
.              1 22 96 rw
..             1 1 1024 rw
a              2 23 3 rw
b              2 24 3 rw
```

5. **Implement `chmod`**:
   - Complete the system call `sys_chmod(...)` in `kernel/sysfile.c`. You have to add this system call on your own. Possible files to be modified to add a system call include `kernel/syscall.c`, `kernel/syscall.h`, `user/usys.pl`, ...

- Create a user program `user/chmod.c` (don't forget to modify `Makefile` as well) to make the `chmod` command available.

6. **Check permission**: Modify the `open` system call to ensure that `open` succeeds only if the provided `omode` is allowed by the file's permission. ~~Also, remember to validate the `omode`, and fails when the given mode is illegal~~ (You do not have to validate whether mode flag is legal or not, as mentioned in the last bullet point of **Part V: `open` Mode Check**). Test your implementation with `ls`, `chmod`, `cat` .etc.

3.5 Test

This problem accounts for 40 points, where 20 points are public test cases. Each test case consists of the following steps:

1. Run `gen` command to generate some directory structures.
2. Run the commands: `symln`, `chmod` and `ls` arbitrary times in arbitrary order and collect the output.
3. Run `mp4_1` to test your `open` and `fstat` system call.

Additionally, we guarantee that in each test case:

1. The length of any file/directory's name is less than or equal to 13.
2. We will not `cd` to other directory. All commands are run at the root directory.
3. We will not `ls` the root directory. We will only `ls` the directories/files/symbolic links we create.
4. The depth of a path will be at most 5, e.g. `/d1/d2/d3/d4/f`.
5. We will only create regular directories and plain-text files in the test cases, no executables will be created. (, and recall that all newly created files/directories should have the default permission `rw`.)
6. Your xv6's output requested by **Problem II** will not affect your grade in **Problem I**. We will ignore all output lines starting with `"BW_DIAG"` or `"BW_ACTION"`.

You can test how many points you would get in public test cases by running `make grade` inside the docker container. We will test your submission in the same way with some private test cases.

4 Problem II - RAID 1 Simulation (60 points)

RAID 1, or disk mirroring, is a technique used to increase data reliability by writing identical data to two separate disks simultaneously. If one disk fails, the data can still be retrieved from the other disk.

This assignment involves implementing a simulation of RAID 1 at the block I/O layer (specifically within the buffer cache functions) in xv6. The goal is to ensure data redundancy during writes and to implement a basic read fallback mechanism in case the primary disk block is unavailable (simulated failure).

4.1 Provided Environment & Configuration

You will start with a modified xv6 environment where:

- **Disk Size:** The total virtual disk size (`FSSIZE`) has been set to 4096 blocks.
- **RAID 1 Layout:** The disk is logically divided into two parts for mirroring:
  - **Disk 0 (Primary):** Uses physical blocks 0 through 2047. All standard filesystem operations target logical blocks within this range (Logical Block Numbers - 0 to 2047).
  - **Disk 1 (Mirror):** Uses physical blocks 2048 through 4095. This disk serves as a mirror copy of Disk 0.
- **Constants:** Relevant constants are defined (likely in `kernel/param.h` or included via headers):
  - `LOGICAL_DISK_SIZE` (=2048): The size of the logical filesystem view (and Disk 0).
  - `DISK1_START_BLOCK` (=2048): The starting physical block number for the mirror disk (Disk 1).
- **Target File:** Your primary modifications will be within `kernel/bio.c`.

- **Testing Mechanism:** The instructor's kernel includes a mechanism to **simulate** read failures on Disk 0 to test your fallback logic (details in Part 2).

4.2 Specification

We split the specification into three main implementation parts:

4.2.1 Part I: Mirrored Writes (Modify `bwrite`)

- **Requirement:** All write operations intended for the logical filesystem must be mirrored. When the kernel requests to write data to a logical block `lbn` (where `0 <= lbn < LOGICAL_DISK_SIZE`), your code must ensure this data is physically written to **both** Disk 0 and Disk 1.
- **Implementation Target:** Modify the `bwrite(struct buf *b)` function in `kernel/bio.c`.
- **Details:**
  - The incoming buffer `b` contains the data to be written, and `b->blockno` initially holds the target **physical block number**.
  - Calculate the corresponding physical block number for Disk 0 (`pbn0`) and Disk 1 (`pbn1 = pbn0 + DISK1_START_BLOCK`).
  - **Crucial:** Before each call to `virtio_disk_rw`, you must temporarily set `b->blockno` to the correct physical block number (`pbn0` or `pbn1`) that `virtio_disk_rw` needs. **After both** physical writes are complete, you must restore `b->blockno` back to its original **block number** before `bwrite` returns. Failure to restore `b->blockno` will corrupt the buffer cache.

4.2.2 Part II: Write Fallback logic (Modify `bwrite`)

In this part of the assignment, you will modify the `bwrite` function in `kernel/bio.c`. The goal is to make `bwrite` respond to "simulated disk failure" and "simulated block failure" by printing specific messages showing the intended write actions (attempting or skipping).

- **Requirement:** The actual disk write process in **xv6** is quite complex, involving the logging system (`log.c` with its `commit()` and `install_trans()` functions) and the underlying `virtio_disk_rw` function. To allow you to focus on the decision-making logic of RAID 1 under failure conditions, this assignment simplifies certain aspects:
  - You are **not required to** modify `virtio_disk_rw` itself or prevent data from actually being written to the disk by the low-level driver when a failure is simulated.
  - Your core task is: when the `bwrite` function is called, it must correctly identify the simulated failure state (set by the `force_read_error_pbn` or the `force_disk_fail_id`) and **print specific messages showing whether it would attempt a write or skip a write to PBN0 and PBN1.**
- **Simulated Failure Paramters (Controlled by TAs):** Your `bwrite` function will need to be aware of and use the following global kernel variables, which will be set by the TA's test programs.
  1. `int force_disk_fail_id`:
     - `-1`: Normal operation, no simulated disk failure.
     - `0`: Simulated Disk 0 (the primary logical disk, PBN0s) as failed.
     - `1`: Simulated Disk 1 (the mirror logical disk, PBN1s) as failed.
  2. `int force_read_error_pbn`:
     - `-1`: Normal operation, no simulated specific block failure.
     - `PBN`: If this value matches the *current* `PBN0` that `bwrite` is about to operate on, it simulates a block-level write failure for that specific `PBN0`.
- **Details:** When `bwrite` is called (where `b->blockno` is the block number from the file system), which is `PBN0` for your RAID layer:
  1. Calculate `pbn0` and `pbn1`.
  2. Read the current values of the global simulation variables `force_disk_fail_id` and `force_read_error_pbn`.

3. **Print diagnostic message:** Display the current PBN0, calculated PBN1, and the current state of the simulation flags (force_disk_fail_id and whether force_read_error_pbn matches the current pbn0). Use a clear format for this message:

```
printf(
    "BW_DIAG: PBN0=%d, PBN1=%d, sim_disk_fail=%d, sim_pbn0_block_fail=%d\n",
    pbn0, pbn1, fail_disk, pbn0_fail_or_not);
```

   – pbn0: The physical block number to write on Disk 0.

   – pbn1: The physical block number to write on Disk 1.

   – fail_disk: Current simulated fail disk (must be -1, 0 or 1).

   – pbn0_fail_or_not: If pbn0 is simulated as failure. (must be 0 or 1).

4. **Action for PNB0 (Disk 0) Write:**

   – If Disk 0 is simulated as failed (via force_disk_fail), bwrite should not attempt the virtio_disk_rw for PBN0. Print a specific "skip" message:

```
printf(
    "BW_ACTION: SKIP_PBN0 (PBN %d) due to simulated Disk 0 failure.\n",
    pbn0);
```

     * pbn0: The physical block number to write on Disk 0.

   – Else, if the current PBN0 is simulated as a "bad block" for writing (via force_read_error_pbn), bwrite should also not attempt the virtio_disk_rw for PBN0. Print a specific "skip" message:

```
printf(
    "BW_ACTION: SKIP_PBN0 (PBN %d) due to simulated PBN0 block "
           "failure.\n",
    pbn0);
```

   – Otherwise (PBN0 is clear for writing), bwrite should attempt the write, and print an "attempt" message:

```
printf(
    "BW_ACTION: ATTEMPT_PBN0 (PBN %d).\n",
    pbn0);
```

5. **Decision and Action for PBN1 (Disk 1) Write:**

   – If Disk 1 is simulated as failed (via force_disk_fail), bwrite should not attempt the virtio_disk_rw for PBN1. Print a specific "skip" message:

```
printf(
    "BW_ACTION: SKIP_PBN1 (PBN %d) due to simulated Disk 1 failure.\n",
    pbn1);
```

   – Otherwise (Disk 1 is clear for writing; note that force_read_error_pbn only directly targets PBN0 for simulation purposes in this assignment – if PBN0 fails at a block level, the write to PBN1 should still be attempted unless Disk 1 itself is failed), bwrite should attempt the write, and print an "attempt" message:

```
printf(
    "BW_ACTION: ATTEMPT_PBN1 (PBN %d).\n",
    pbn1);
```

6. **Restore b->blockno:** At the end of all operations, you must restore b->blockno to the original incoming block number (which was PBN0) to ensure Buffer Cache consistency.

- **Important Note on Real RAID 1 Behavior:** Although this assignment simplifies write failure handling by focusing on your decision logic and printf messages (and not requiring you to prevent the underlying `virtio_disk_rw` from actually succeeding if it's called), it's crucial to understand that:
  - In a real, robust RAID 1 system, when a write to one disk or block fails, the system's goal is to **preserve data integrity and availability on at least one copy**.
  - This means if Disk 0 or PBN0 fails a write, a real RAID 1 system **would still try its best to successfully write the data to the healthy Disk 1 or PBN1**, and vice-versa. The write operation to the logical block is generally considered successful if at least one mirror copy is successfully written.
  - The simplification in this assignment is to allow you to focus on implementing the correct conditional logic for attempting or skipping writes based on the simulation flags.

4.2.3 Part III: Read Fallback Logic (Modify `bread`)

- **Requirement:** Implement a read fallback mechanism. Normally, reads should be served from Disk 0. However, if a read attempt on Disk 0 fails (this failure will be **simulated** during testing), your code must automatically attempt to read the data from the corresponding block on Disk 1.
- **Implementation Target:** Modify the `bread(uint dev, uint blockno)` function in `kernel/bio.c`.
- **Details:**
  - The incoming blockno argument represents the **block number** that the filesystem layer wants to read (this will be a value between 0 and `LOGICAL_DISK_SIZE - 1`, corresponding to a block on Disk 0). The `bget` function will provide a buffer `b` associated with this LBN.
  - Your primary logic modification should occur within the part of `bread` that handles reading data from the disk (typically inside the `if(!b->valid || force cache misses logic)`.
  - **Normal Path:** Calculate `pbn0 = blockno`. Attempt to read physical block `pbn0` using the underlying I/O function.
  - **Fallback Path Trigger (Simulation):** To allow testing, the instructor's kernel contains a global variable `extern int force_disk_fail_id` and `extern int force_read_error_pbn`. Your `bread` implementation **must** check if the current failure disk is equal to disk 0, or if the physical block it is about to read from Disk 0 (`pbn0`) is equal to the current value of `force_read_error_pbn`.
  - **Fallback Action:** If `simulate_error` is true, skip the read from `pbn0`, calculate `pbn1`, and attempt to read from `pbn1` instead.
  - **Return Value:** `bread` should return the buffer `b` containing the correct data, whether it came from `pbn0` (normal path) or `pbn1` (fallback path).
  - **Error Handling:** You are **not required** to handle real hardware I/O errors reported by `virtio_disk_rw`. The focus is on correctly implementing the fallback logic based on the `force_disk_fail_id` or `force_read_error_pbn` simulation.

4.3 Files to Modify

- Primarily: `kernel/bio.c`
- You might need to include the headers you need.

4.4 Test

This problem accounts for 60 points, where 30 points are public test cases (disk failure), other 30 points for private test cases (block failure).

1. Correctness of mirrored writes (data identical on PBN0 and PBN1). We will check it using `raw_read` system call in our test instead of bread. `Raw_read` will directly read the data in the block.
2. Check the required printed message mentioned above while forcing failure on bwrite.

3. Correctness of the read fallback logic when triggered by the `force_read_error_pbn` or `force_disk_fail_id` simulation. We will use `raw_read` to modify the data in the failure block or failure disk, so be sure you will return the right data.

5 Bonus Report

We encouraged you to help other students. Please describe how you helped other students here. You should make the descriptions as short as possible, but you should also make them as concrete as possible (e.g., you can screenshot how you answered other students' questions on NTU COOL). Please note that you will not get any penalty if you leave it empty here. Please also note that this bonus is not for you to do optimization, so we will not release the grading criteria and the grades. Regarding the final letter grades, it is very likely that this does not help - you will get promoted to the next level only if you are near the boundary of levels and you have significant contributions.

6 Submission

Please submit your `<student_id>.patch` to NTU COOL, and your bonus report to Gradescope.

6.1 Bonus Report

Submit your report to the Bonus Report section on Gradescope in one pdf file.

6.2 xv6

The submission of this MP use `diff` to generate a patch file. Note that the following commands are available on Linux and macOS only. For Windows users, there are 2 workarounds to run `diff`:

1. If you have installed Git for Windows, you should have **Git Bash** on your computer. You can run the commands below in Git Bash.
2. Install WSL, Window Subsystem for Linux, and run the commands below in WSL. You can refer to the slides of MP0 for the WSL installation guide.

If neither way is applicable, feel free to book a TA hour. We will try to help you with your issue.

6.2.1 Submission Format

In this MP, we only accept a patch file as your submission. Please generate the patch file by the instructions below:

1. Clean your workspace, and backup your progress in case you erase all your work accidentally.

```
root@2a6008d5c4dc:/home/xv6# make clean
root@2a6008d5c4dc:/home/xv6# # ctrl D to stop and leave container
user@host ~/os/mp4 $ cd ..
user@host ~/os $ cp -r mp4 mp4-bk
```

2. Rename you current `mp4` directory. Please make sure you do so, or it is very likely that your progress will be overwritten by `unzip`.

```
user@host ~/os $ mv mp4 mp4-new
```

3. Download `mp4.zip` **AGAIN** from NTU COOL and unzip it.

```
user@host ~/os $ unzip mp4.zip
```

4. Now you have `mp4-new`, the folder containing you work, and `mp4`, the original mp4.
5. Use `diff` to generate a patch file. You may add more `--excludes` to filter out unrelated files. And please do not mix up the order of `mp4` and `mp4-new`.

```
user@host ~/os $ diff -ruN \
--exclude='.git' \
--exclude='.vscode' mp4 mp4-new > <student_id>.patch
```

It is normal that `diff` exit with code 1. `diff` returns 0 if two files are the same and returns 1 if they are different. Please check the return code with `echo $?`

6. Upload `<student_id>.patch` to NTU COOL, e.g. `r13922001.patch`. The leading letter should be **lowercase**.

We will apply your patch and test your code using the following command:

```
$ cd mp4  # this is the original mp4
$ patch -p1 < ../<student_id>.patch
$ make grade
```

Furthermore, if your patch file creates/modifies/deletes files that are listed here, there will be a point penalty.

- Grading-related files (`user/gen.c`, `user/mp4_1.c`, `user/mp4_2_*test.c`, `gradelib.py`, `grade-mp4.py`)
- Object files, dependency files, or any file that can be removed by `make clean`.
- Other unrelated files, including but not limit to `.git`, `.vscode`, `.idea`, ...

Note: Any attempt to manipulate the grading process—such as modifying the behavior of `make grade` by tampering with the `grade` command in the `Makefile`—will result in severe consequences.

6.3 Grading Policy

- The total points of this MP is 100 points.
- You will get 0 points if we cannot apply your patch or compile your submission. You will also get 0 points if you submit anything other than a patch file, e.g. the whole `mp4.zip`.
- You will be deducted 10 points if the name of the patch is wrong. Using uppercase in the `<student_id>.patch` or not suffixing the file name with `.patch` are both considered wrong.
- If your submission is late for n days, your score will be $max(raw\_score - 20 \times \lceil n \rceil), 0)$. Note that you will not get any points if $\lceil n \rceil >= 5$.

7 References

[1] xv6, a simple Unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2024/xv6.html

[2] Docker: Empowering App Development for Developers. https://docs.docker.com/

[3] RISC-V: The Free and Open RISC Instruction Set Architecture. https://riscv.org/

[4] QEMU, the FAST! processor emulator. https://www.qemu.org/