Sprint 2 Report
Team StuckOverflow

touch Command Functionality

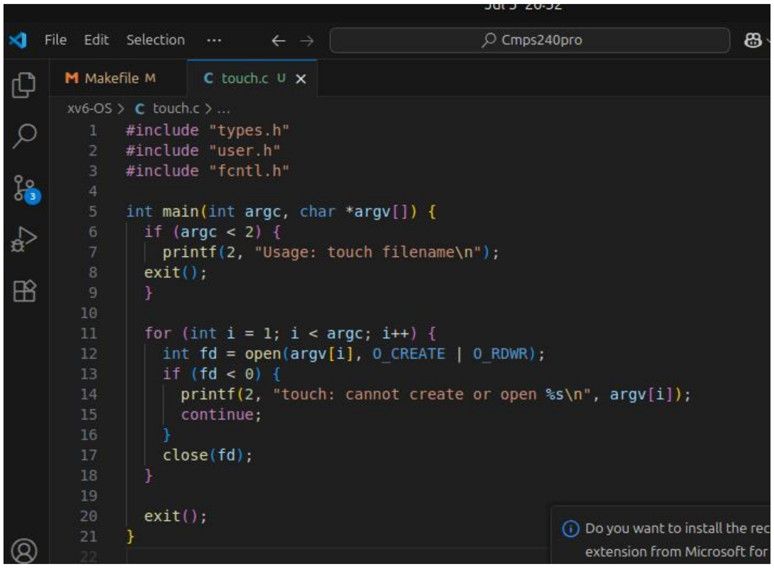
Objective.

Implement a touch command that creates a new empty file or updates the timestamp of an existing file. This replicates basic touch behavior in UNIX-like systems and enhances usability in xv6 by supporting file creation directly from the shell.

Steps Taken.

1. Created a new user program

- File path: user/touch.c
- Implemented a program that:
    - Accepts one or more filenames as command-line arguments.
    - Calls open() with O_CREATE | O_RDWR to either create or update the file.
    - Closes the file descriptor after creation.
    - Skips and prints an error message if the file cannot be created



```c
#include "types.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf(2, "Usage: touch filename\n");
  exit();
  }

  for (int i = 1; i < argc; i++) {
    int fd = open(argv[i], O_CREATE | O_RDWR);
    if (fd < 0) {
      printf(2, "touch: cannot create or open %s\n", argv[i]);
      continue;
    }
    close(fd);
  }

  exit();
}
```

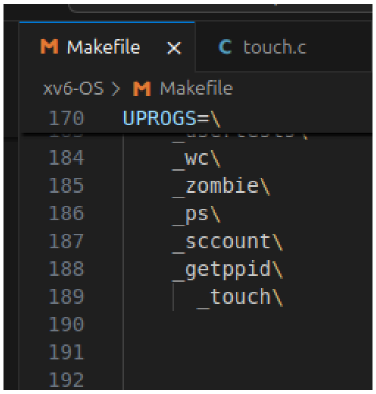
Do you want to install the rec
extension from Microsoft for

2. Modified Makefile to include the new program

- Opened the top-level Makefile.
- Located the line starting with

  UPROGS = \
  _touch\

- This ensured that the touch program gets compiled and included in fs.img.

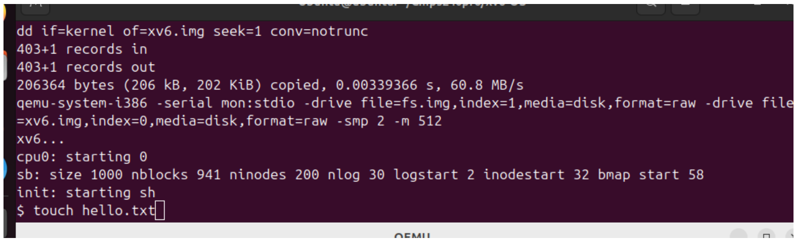


```
UPROGS=\
    _wc\
    _zombie\
    _ps\
    _sccount\
    _getppid\
      _touch\
```

---

3. Tested touch Command Inside xv6

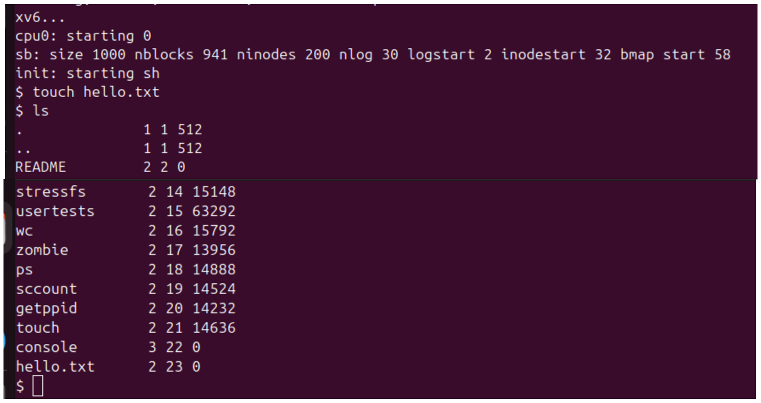
$ touch hello.txt



```
dd if=kernel of=xv6.img seek=1 conv=notrunc
403+1 records in
403+1 records out
206364 bytes (206 kB, 202 KiB) copied, 0.00339366 s, 60.8 MB/s
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file
=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ touch hello.txt
```

$ ls
...
Hello.txt

When using ls:



```
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ touch hello.txt
$ ls
.              1 1 512
..             1 1 512
README         2 2 0
stressfs       2 14 15148
usertests      2 15 63292
wc             2 16 15792
zombie         2 17 13956
ps             2 18 14888
sccount        2 19 14524
getppid        2 20 14232
touch          2 21 14636
console        3 22 0
hello.txt      2 23 0
$ 
```

head Command Functionality

Objective.

The objective of this functionality is to implement a user-level head command in xv6, which prints the first N lines of a given file. This enhances file interaction capabilities within xv6 by allowing users to preview file contents quickly, similar to the standard Unix head command.

Steps Taken.

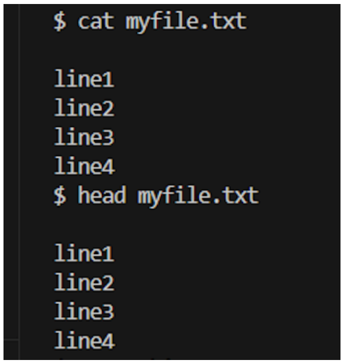
1. **Created a new user program:** Developed head.c, a new user-level program that reads a specified file and outputs the first N lines to the console. The program accepts one mandatory argument (filename) and one optional argument (number of lines to print). If the number of lines is not specified, it defaults to printing the first 10 lines.
2. **Code overview:** The program opens the target file and reads it in chunks. It counts newline characters to determine when to stop printing. Input validation is included to handle cases such as missing filename, invalid line count, or file open errors. The program exits gracefully on error, providing clear messages.
3. **Modified the Makefile:** Added _head\ to the UPROGS variable in the Makefile to ensure head.c is compiled and included in the xv6 user programs during the build process.

Testing

To ensure the correctness and robustness of the head command, the following test cases were performed inside the xv6 environment:

Default behavior: print first 4 lines.

- Created a file myfile.txt with 4 lines using shell input redirection.
- Ran head myfile.txt without specifying the number of lines.
- **Expected output:** First 4 lines of the file printed.



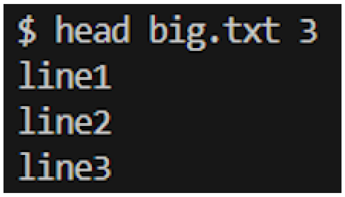
```
$ cat myfile.txt

line1
line2
line3
line4
$ head myfile.txt

line1
line2
line3
line4
```

Created another file with more lines(12 lines) called big.txt and tested custom number of lines.

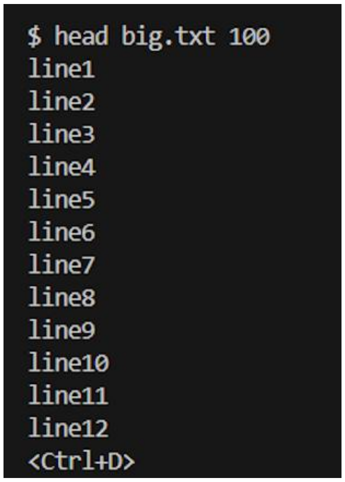
- Ran head big.txt 3 to print only the first 3 lines.
- Expected output: Lines 1, 2, and 3 of the file printed.



```
$ head big.txt 3
line1
line2
line3
```

Number of lines greater than file length.

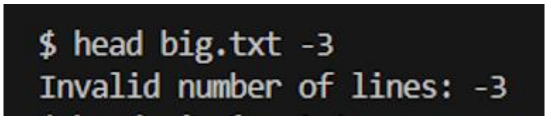
- Ran head big.txt 100 requesting more lines than file contains.
- Expected output: Entire file (12 lines) printed without crashing or errors.



```
$ head big.txt 100
line1
line2
line3
line4
line5
line6
line7
line8
line9
line10
line11
line12
<Ctrl+D>
```

Invalid number of lines.

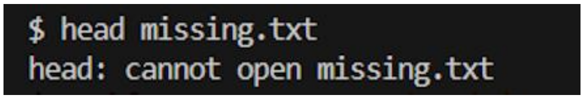
- Ran head big.txt -3 with a negative line count.
- Expected output: Error message Invalid number of lines: -3.



```
$ head big.txt -3
Invalid number of lines: -3
```

File does not exist.

- Ran head missing.txt with a non-existent file.
- Expected output: Error message head: cannot open missing.txt.



```
$ head missing.txt
head: cannot open missing.txt
```

find command.

Goal.

To implement a find command that takes path & file name and searches recursively starting from the path to find the file. If found, it prints the full path to it.

find <path> <target>

Flow.

1. Program will take the path from the user and try to open it using open() , which returns a file descriptor fd to us.
2. Through the fd , the program uses it to check the metadata of the file using fstat and copies it inside the struct stat st .
    1. If it is a file, compare it to the target and outputs the full path if it is the target or error otherwise.
    2. If it is a directory, we move to step 3.
3. Appends the directory to the current buffer and loop over its content using read() function and copies it inside the directory entry de .
4. It searching for the folder by recursively calling find() again.

Steps.

1. created find.c file to implement the command in it.
    1. included types.h , user.h → since it is a user program
    2. included stat.h → to access the struct stat to get the stat of the files (mostly needed is the file type).
    3. included fs.h → to access the struct dirent to access the inode number and the name.
2. Implemented find(char* path, char *target, int *found);
    1. arguments:
        1. path → directory we want to search (taken from argv[1]).
        2. target → file we want to find (taken from argv[2]).

        3. found → flag that returns to the main function: The goal of this flag is in case the search is completed and the file is not found → then we know that the flag was never changed.
    2. created:
        1. fd integer to get file descriptor after opening the file.
        2. buf & p → the buffer to returned path and its pointer.
        3. st → struct of type struct stat to get the stat for the file.
        4. de → struct of type struct dirent to get the directory entry in case the file descriptor is a directory
    3. tries to open the file and stores the results in fd or returns if error (incorrect directory or error).
    4. tries to stat the file and stores the results in st or returns if error.
    5. In case the fd points to a file, it:
        1. executes the file name from the current path.
        2. compares it to the target & if found → changes the flag found to 1.
        3. closes the file and returns.
    6. Else → fd points to a directory.
        1. check if the path fits the buffer size.
        2. append it to the buffer.
        3. start reading the content using read() .
            1. it skips empty inodes (inum == 0 & the default subdirectories . & ..).
            2. append the current subdirectory/file to the path.
        4. finally, recursively calls find() .

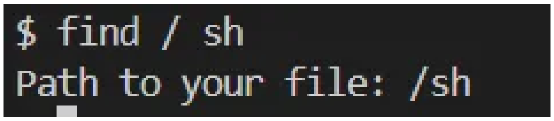
3. in main(int argc, char *argv[]);
    1. 3 arguments should be passed (taken from the terminal):
        1. find → the command itself.
        2. argv[1] → path to search in.
        3. argv[2] → file to search for.
    2. then call find()

Testing.

Default Behavior.

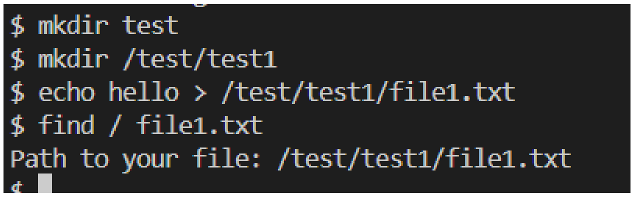
Tested for it twice -> finding a regular folder & creating my own folder.

1. searched for sh in /



```
$ find / sh
Path to your file: /sh
```

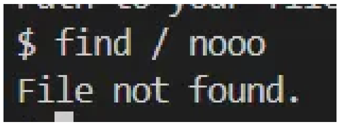
2. created directories and subdirectories.
    1. mkdir test
    2. mkdir /test/test1
    3. echo hello > /test/test1/file1.txt
    4. then searched for file1.txt in /



```
$ mkdir test
$ mkdir /test/test1
$ echo hello > /test/test1/file1.txt
$ find / file1.txt
Path to your file: /test/test1/file1.txt
$
```

File doesn’t exist.

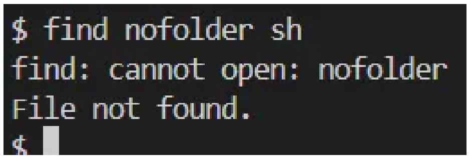
Searched for a file that doesn’t exist (searched for nooo inside /).



```
$ find / nooo
File not found.
```

Directory doesn’t exist.

Searched inside a folder that doesn’t exist.



```
$ find nofolder sh
find: cannot open: nofolder
File not found.
$
```

cp command.

Goal.

To implement a cp command that takes source & destination files to:

- Create a new destination file (if it doesn’t exist).
- Copies the content of the source into the destination

cp <source> <destination>

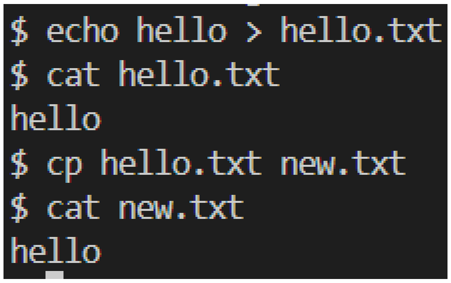
Flow & Steps.

1. in main(int argc, char *argv[]), 3 arguments should be passed (taken from the terminal):
    i. cp → the command itself.
    ii. argv[1] → source file.
    iii. argv[2] → destination file.
2. Tries to open source file using open() with mode O_RDONLY since it is only reading the source file and stores its file descriptor in fd1.
3. Tries to open destination file using open() with mode O_CREATE or O_RDWR and stores its file descriptor in fd2.
4. If succeeded, it reads the content in the source using read() with fd1 and stores it in common buffer buf.
5. Then, it writes it in the destination file using write() with fd2.

Testing.

Copying to a new file

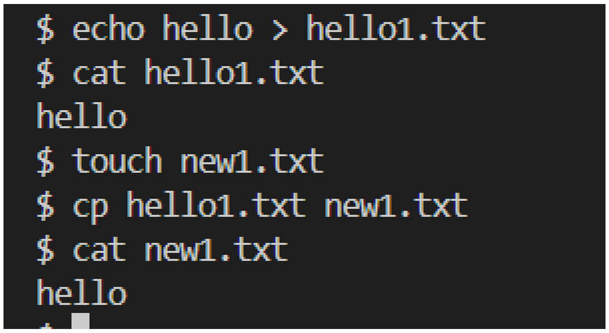
1. Created a file called hello.txt and added “hello” to it.
    echo hello > hello.txt
2. Copied the content of hello.txt to a new file called new.txt using cp command
    cp hello.txt new.txt
3. Used cat to read the content of the file hello.txt.



```
$ echo hello > hello.txt
$ cat hello.txt
hello
$ cp hello.txt new.txt
$ cat new.txt
hello
```

Copying to an existing file.

1. Created a file called hello.txt and added “hello” to it.
   echo hello1 > hello1.txt
2. Created a new empty file new1.txt using the touch command we worked on earlier
   touch new1.txt
3. Copied the content of hello1.txt in new1.txt using cp command.
   cp hello1.txt new1.txt
4. Read the content of the file new1.txt using cat command.
   cat new1.txt



```
$ echo hello > hello1.txt
$ cat hello1.txt
hello
$ touch new1.txt
$ cp hello1.txt new1.txt
$ cat new1.txt
hello
$
```