COMP1521

W8 - Files

Overview

Admin

Files

File operations

Admin

Assignment 2 released

W7 weekly test due on thursday (today)

W8 weekly test released today

W8 labs due next monday

Assignment 1 style marks probably some time in W9 (no promises)

- Style marking is 90% finished as of this morning, but releasing the marks may take some time

Admin: Assignment 2

General advice:

- Complete this week's lab before starting the assignment (the last two exercises will be very helpful)
- Write lots of helper functions
- I recommend using structs to represent the TABI/TBBI/TCBI records in memory
- Start early (and if you want help from tutors, come to help sessions as early as you can)
- We will cover all content needed for the assignment today

Admin: Assignment 2

Ideas for helper functions:

- Read little endian number from file
- Write little endian number to file
- Read a byte from a file and exit with an error if no byte could be read
- Find the size of a file
- Check if the nth bit of a number is 1 or 0
- Convert file permissions between string and bit representation

Using structs:

- Instead of having one big function where you simultaneously read from TABI and write to TBBI, have three functions:

Read TABI to struct, convert TABI struct to TBBI struct, write TBBI struct to output file.

Files

What is a UNIX file

A file is a stream of data

This includes:
- ascii text
- c program
- compiled executable
- a directory (a small amount of data describing the contents of the directory)
- a network connection (socket; a stream of data received over the internet)
- peripherals (the stream of data received from your keyboard)
- a symbolic link to another file (the stream of data in the other file)

The philosophy is that “everything is a file”

What is a UNIX file

Some special files that will be relevant are called “stdin”, “stdout” and “stderr”.

These files represent the console (they are typically connected to the console).

stdin represents the user input in the console.

stdout represents regular program output to the console

stderr represents error program output to the console

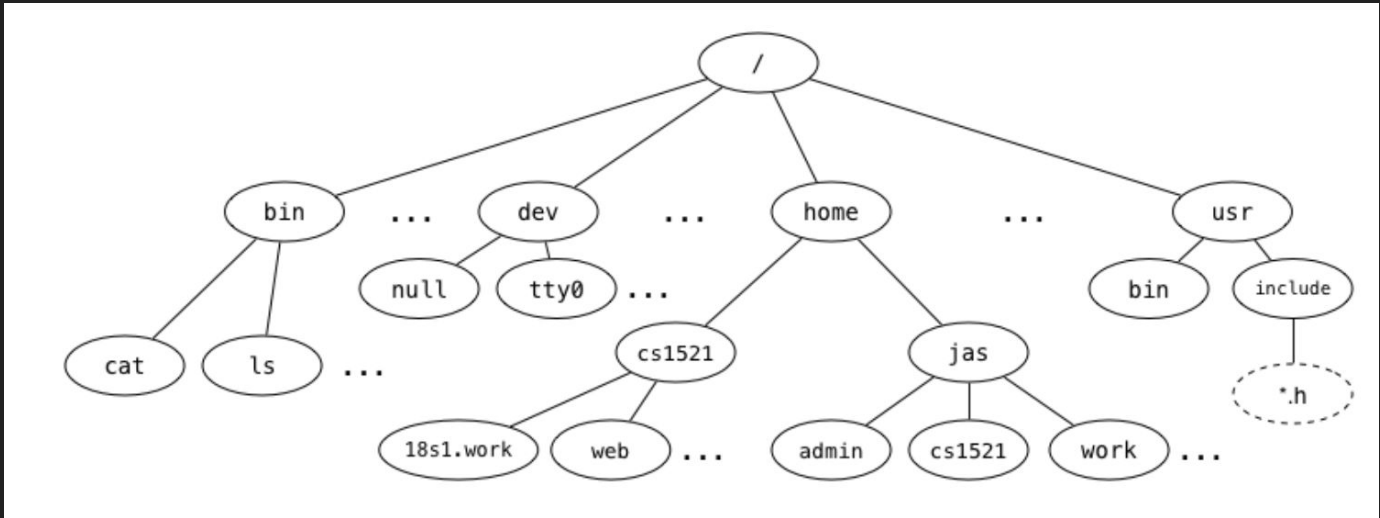
We can interact with these “files” the same way we interact with any other file.

How are files organized?

Files are contained in a tree-like structure of directories.

The root directory (/) contains subdirectories home, dev, usr, etc

Those subdirectories can contain more subdirectories, as well as files.



How do we refer to files

A file can be referred to by its full “path” from the root directory

E.G: /import/kamen/4/z55555555/example.txt (absolute path)

We can also use a relative path, relative to the directory we are currently inside.

For example, if I am in the directory `/import/kamen/4/z55555555/`, then I can just write `./example.txt` or even `example.txt` to refer to the above file.

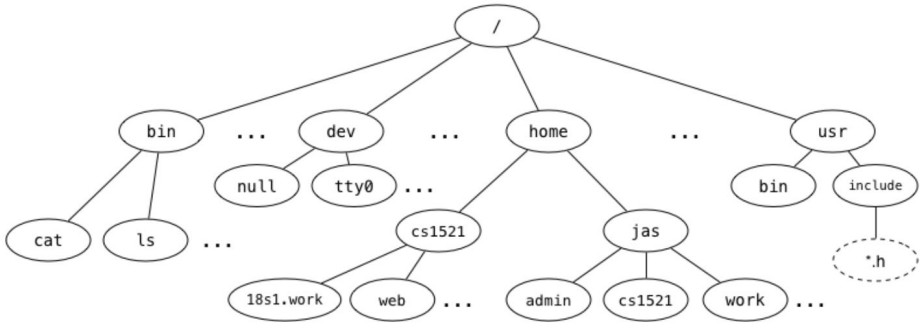
`.` is a special directory path that always refers to the current directory

`..` is a special directory path that always refers to the parent directory

`~` is a special directory path that always refers to the absolute path of the home directory

Tutorial Q1

1. We say that the Unix filesystem is *tree-structured*, with the directory called `/` as the root of the tree, e.g.,



Answer the following based on the above diagram:

a. What is the full pathname of COMP1521's `web` directory?

b. Which directory is `~jas/../..` ?

c. Links to the children of a given directory are stored as entries in the directory structure. Where is the link to the parent directory stored?

d. What kind of filesystem object is `cat` ?

e. What kind of filesystem object is `home` ?

f. What kind of filesystem object is `tty0` ?

g. What kind of filesystem object is a symbolic link? What value does it contain?

h. Symbolic links change the filesystem from a tree structure to a graph structure. How do they do this?

File operations

Low level file operations

At the lowest level, files can be read and modified using syscalls (Linux syscalls, as opposed to MIPS syscalls).

The level above that are the C functions open, close, read and write. These functions call the syscalls themselves.

The level above that (the functions we actually use) are the functions fopen, fclose, and a family of functions that read and write to files (fgetc, fread, fputc, fwrite, etc).

These functions call the lower level open/close/read/write functions, which in turn call the syscalls to actually do the operations.

Opening files

A file can be opened with the function `fopen`.

The function takes an argument `path`: the name of the file to open.

It takes another argument `mode`: a string representing the mode to open the file

“r” = read, “w” = write, “a” = append, “r+” = read and write.

The function returns a file pointer (FILE*), which we can use to edit and inspect the file with other functions.

A file can have multiple (FILE*) objects at the same time.

```
FILE *
fopen(const char * restrict path, const char * restrict mode);
```

File pointers

File pointers refer to specific files.

They also store information about how the file has been opened (ie, read or write), as well as where in the file the pointer is looking (1st byte, 2nd byte, etc).

Typically, a file pointer is initialised to point to the first byte of a file (with “r” or “w”).

If instead we use “a”, the pointer is initialized to point just past the last byte of the file, so that we can “append” to the file when we write.

Opening in mode “w” will delete all bytes in the file.

If you wish to write in the middle of the file and preserve the bytes currently stored, use “r+” mode.

Writing to files

We can write one byte to a file using the `fputc` function.

The function takes an int c (which should contain a single byte value), and a FILE* stream to write to.

It then writes the byte to where the file pointer is looking, and moves the file pointer forwards to the next byte.

The function returns the character written on success, or EOF if an error occurred.

```
int
fputc(int c, FILE *stream);
```

Writing to files

We can write multiple bytes to a file at once using `fwrite`.

It takes a first argument `ptr`, which is an array of elements which should be written to the file.

The second argument `size` is the size in bytes of each element in the array.

The third argument `nitems` is the number of elements in the array.

The last argument `stream` is the file pointer to write to.

The function will write the objects specified in the array and then move the file pointer forward by the amount of bytes written.

The function returns the number of items successfully written to the file.

```
size_t
fwrite(const void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);
```

Portability of fwrite

fwrite should only be used with elements that have a size of 1 byte.

For larger elements, the endianness used to store the elements will depend on the machine. This can lead to inconsistent behaviour.

As such using multiple byte elements with fwrite is strongly discouraged.

Writing to files

One of the most useful functions for writing to files is `fprintf`.

`fprintf` has the exact semantics of `printf`, but with an extra first argument which specifies which file to print to.

fprintf writes a formatted string to a file, and moves the file pointer forward by the amount of bytes written.

fprintf is often used to output error messages, by printing formatted strings to the stderr file

```
fprintf(file, format, arg1, ..., argn)
FILE *file;
char *format;
```

Reading from files

We can read one byte from a file using the `fgetc` function.

The function takes one argument, a file pointer to read from.

The function will read one byte from the file, move the pointer to the next byte, and return the byte that it read inside an int type.

It is returned inside an int so the function is able to also return EOF (-1) if an error occurs.

```
int
fgetc(FILE *stream);
```

Reading from files

We can read multiple bytes from a file using `fread`, with semantics very similar to `fwrite`.

It takes a first argument `ptr`, which is an array of elements in which the data read from the file will be stored.

The second argument `size` is the size in bytes of each element in the array.

The third argument `nitems` is the number of elements in the array.

The last argument `stream` is the file pointer to read from.

The function will read the objects specified from the file into the array and then move the file pointer forward by the amount of bytes read.

The function returns the number of items successfully written to the file.

```
size_t
fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);
```

Portability of fread

For the same reason as with fwrite, we should not use elements larger than 1 byte since we don’t know if the file is necessarily formatted with the same endianness as the computer that is reading it.

Reading from files

Another sometimes useful function is `fgets`, which reads one line from a file.

The first argument `str` is an array where the bytes read will be stored.

The second argument `size` is the maximum amount of bytes to be read (to avoid overflow errors).

The third argument `stream` is the file pointer to read from.

The function returns a pointer to `str` on success, or NULL on error.

```
char *
fgets(char * restrict str, int size, FILE * restrict stream);
```

Moving the file pointer

The file pointer can be manually moved using `fseek`.

The first argument `stream` is the file pointer to move.

The second argument `offset` is a 64-bit integer (long) describing a distance in bytes.

The third argument `whence` can take values SEEK_SET, SEEK_CUR or SEEK_END, describing what the offset is from.

SEEK_SET takes the offset relative to the start of the file. SEEK_CUR takes the offset relative to the current position of the file pointer. SEEK_END takes the offset relative to the end of the file.

The function returns an 0 on success and -1 on error.

```
int
fseek(FILE *stream, long offset, int whence);
```

Checking where the file pointer is

We can use `ftell` to check where the file pointer is currently.

The function takes a file pointer as an argument and returns the zero-indexed byte position of the file pointer.

```
long
ftell(FILE *stream);
```

Closing the file

Closing a file ensures that all the reads/writes have been applied.

It happens automatically when your program terminates, but it is good practice to do it manually (a bit like malloc and free).

We can used `fclose` to close a file pointer, with the only argument being the file pointer `stream` to close.

```
int
fclose(FILE *stream);
```

File permissions

File permissions are stored in a file's metadata.

They are displayed to the terminal when using commands like `ls`:

```
-rw-r--r-- 1 z5420273 z5420273  534 Sep 19  2022 count.s
-rw-rw-r-- 1 z5420273 z5420273  856 Sep 19  2022 dynamic_load.s
```

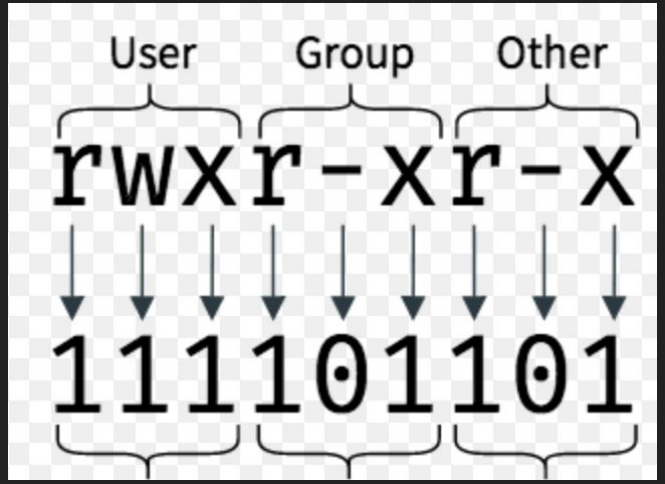
They are stored in a bitwise representation:

: ttttrwxrwxrwx

(first 4 bits represent file type)

File permissions can be read using `stat`.

File permissions can be modified using `chmod`.



Stat

`stat` takes two arguments: a pathname of the file to stat, and a stat struct object pointer to store the results in.

It will return an error code (non zero) on failure, or zero on success.

The information about file permissions and type can be accessed in the st_mode field.

There exist already defined bitmasks which can be used to inspect the object:
run “man inode”

```c
int main(int argc, char **argv) {
    struct stat s;
    char *pathname = argv[1];
    int code = stat(pathname, &s);
    if (code != 0) {
        perror("couldn't stat file");
        exit(1);
    }
    mode_t mode = s.st_mode;
}
```

Chmod

`chmod` takes two arguments: a `path` : the pathname of a file and a `mode`: the `mode_t` (just a number) which is formatted the same as the `s.st_mode` field from stat.

```
int main(int argc, char **argv) {
    char *pathname = argv[1];
    mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IROTH;
    int code = chmod(pathname, mode);
    if (code != 0) {
        perror("couldn't chmod file");
        exit(1);
    }
}
```

Managing low level errors

When errors occur in low level functions like these, they typically have a specific return value (like EOF or -1).

The function will also usually set a global variable called ERRNO to describe the specific error that occured.

In order to take advantage of this, we can use a function call `perror`.

When we detect a function returning something like EOF or -1, we know that some error occurred.

We can print to the terminal the specific error by simply calling `perror()`, which will read the ERRNO variable and print the appropriate message.

Tutorial Questions 3,4,5

3. What does *fopen* do? What are its parameters?

4. What are some circumstances when *fopen* returns NULL?

5. How do you print the specific reason that caused *fopen* to return `NULL` ?

Tutorial Questions 6,7,8

6. Write a C program, `first_line.c`, which is given one command-line argument, the name of a file, and which prints the first line of that file to `stdout`. If given an incorrect number of arguments, or if there was an error opening the file, it should print a suitable error message.
7. Write a C program, `write_line.c`, which is given one command-line argument, the name of a file, and which reads a line from `stdin`, and writes it to the specified file; if the file exists, it should be overwritten.
8. Write a C program, `append_line.c`, which is given one command-line argument, the name of a file, and which reads a line from `stdin` and appends it to the specified file.

Tutorial Questions 9,10,11, 12

9. Why should you not use *fgets* or *fputs* with binary data?

10. What does the following *printf* statement display?

```
printf ("%c%c%c%c%c%c", 72, 101, 0x6c, 108, 111, 0x0a);
```

11. How many different values can *fgetc* return?

12. Why are the names of *fgetc*, *fputc*, *getc*, *putc*, *putchar*, and *getchar* misleading?

Tutorial Q15

15. Consider a file of size 10000 bytes, open for reading on file descriptor `fd`, initially positioned at the start of the file (offset 0). What will be the file position after each of these calls to `lseek()`? Assume that they are executed in sequence, and one will change the file state that the next one deals with.

    a. `lseek(fd, 0, SEEK_END);`
    b. `lseek(fd, -1000, SEEK_CUR);`
    c. `lseek(fd, 0, SEEK_SET);`
    d. `lseek(fd, -100, SEEK_SET);`
    e. `lseek(fd, 1000, SEEK_SET);`
    f. `lseek(fd, 1000, SEEK_CUR);`