COMP1521

TUTORIAL 9

FILE PERMISSIONS (AGAIN) > ENVIRONMENT VARIABLES > UTF-8 (UNICODE) > RECURSIVE FILE TRAVERSAL

FILES PERMISSIONS (2)

Modes

d means directory
- means regular file

User/Owner

Group

Everyone else

d {r w x}{r - x}{r - x}

{1 1 1}{1 0 1}{1 0 1}

{755}

Each of these permissions can be represented in a set of 3 octal values within the st_mode of the stat struct of a file

- Which you can use the chmod program in your terminal: chmod 755 file.exe
- Or the function in your programs: chmod(file_name, 755)

Modes bit masks

Can find a more detailed information about the file metadata @ https://man7.org/linux/man-pages/man7/inode.7.html

#define	value	Description
S_IFDIR	0040000	Directory
S_IFREG	0100000	Regular file
S_IRUSR	0000400	Owner has read permission
S_IWUSR	0000200	Owner has write permission
S_IXUSR	0000100	Owner has execute permission
S_IRGRP	0000040	Group has read permission
S_IWGRP	0000020	Group has write permission
S_IXGRP	0000010	Group has execute permission
S_IROTH	0000004	Others have read permission
S_IWOTH	0000002	Others have write permission
S_IXOTH	0000001	Others have execute permission

Example: q1.c

Write a C program, which is given 1 or more arguments that are file path names, if the file is publically-writeable, change the permission to not publically-writeable, leaving other permissions unchanged

ENVIRONMENT VARIABLES

What is an environment variable?

- When a program is run, it is passed a set of environment variables
  - These are things such as PATH (path to the program), HOME (users current home directory) and many more
- They are an array of strings, of the form ‘name=value’ that are null terminated
- We access these through the global variable environ
  - getenv() – get a specific environment variable e.g getenv(“HOME”)
  - setenv() – set or alter an environment variable
  - extern char **environ – access the global array e.g environ[1] is the “HOME” variable

Example: q4.c

Write a C program, which prints the contents of the file $HOME/.diary to stdout

UNICODE

What is Unicode? Why does it exist?

- Modern computers use Unicode to represent text, but why did it replace ascii?
- Well simply there is so many possible characters we need to represent, and ascii values use 1 byte and are not sufficient, whereas Unicode can represent characters up to 4 bytes in total
  - While we have mathematical expressions to represent numbers using two's compliment and IEE754
- We use a lookup table for character encoding:
  - This table ranges from 0x0000 to 0x10FFFF

So how do we represent Unicode values

- If Unicode values have a range from 0x0000 to 0x10FFFF, this means we need 21 bits to represent the values.
  - Should we just use a 32-bit number to store all Unicode values? Why or why not?

- No, it is not space efficient and we would have a large amount of redundant memory
  - To represent an ascii character we just need 1 byte, so why don't we just use 1 byte
  - When we need more data, we can use 2, 3 or 4-byte Unicode representations

- And that's why we have UTF-8

UTF-8

- UTF-8 is a variable length Unicode encoding, allowing us to represent a larger range of characters without redundant memory

# bytes	# bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	0xxxxxxx	-	-	-
2	11	110xxxxx	10xxxxxx	-	-
3	16	1110xxxx	10xxxxxx	10xxxxxx	-
4	21	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

- To find the number of codepoints in each Unicode character we can use bit masks to extract if it will be a 1,2,3 or 4 length Unicode character:
  - 1 byte – Ensure the first bit is 0
    - (byte & 0x80) = 0 or in binary (byte & 0b10000000) = 0
  - 2 byte – Ensure the first 2 bits are 1, and the third bit is 0
    - (byte & 0xE0) = 0xC0 or in binary (byte & 0b11100000) = 0b11000000
  - 3 byte – Ensure the first 3 bits are 1, and the fourth bit is 0
    - (byte & 0xF0) = 0xE0 or in binary (byte & 0b11110000) = 0b11100000
  - 4 byte - Ensure the first 4 bits are 1, and the fifth bit is 0
    - (byte & 0xF8) = 0xF0 or in binary (byte & 0b11111000) = 0b11110000
  - Continuation byte – is correct we confirm the first bit is 1 and the second bit is 0
    - (byte & 0xC0) = 0x80 or in binary (byte & 0b11000000) = 0b10000000

UTF-8 Bit masks

	Binary Bitmasks	Hex Bitmasks
1 Byte	01011101 & 10000000 = 00000000	01011101 & 0x80 = 0
2 Byte	11010010 & 11100000 = 11000000	11010010 & 0xE0 = 0xC0
3 Byte	11101001 & 11110000 = 11100000	11101001 & 0xF0 = 0xE0
4 Byte	11110111 & 11111000 = 11110000	11110111 & 0xF8 = 0xF0
Continuation	10100101 & 11000000 = 10000000	10100101 & 0xC0 = 0x80

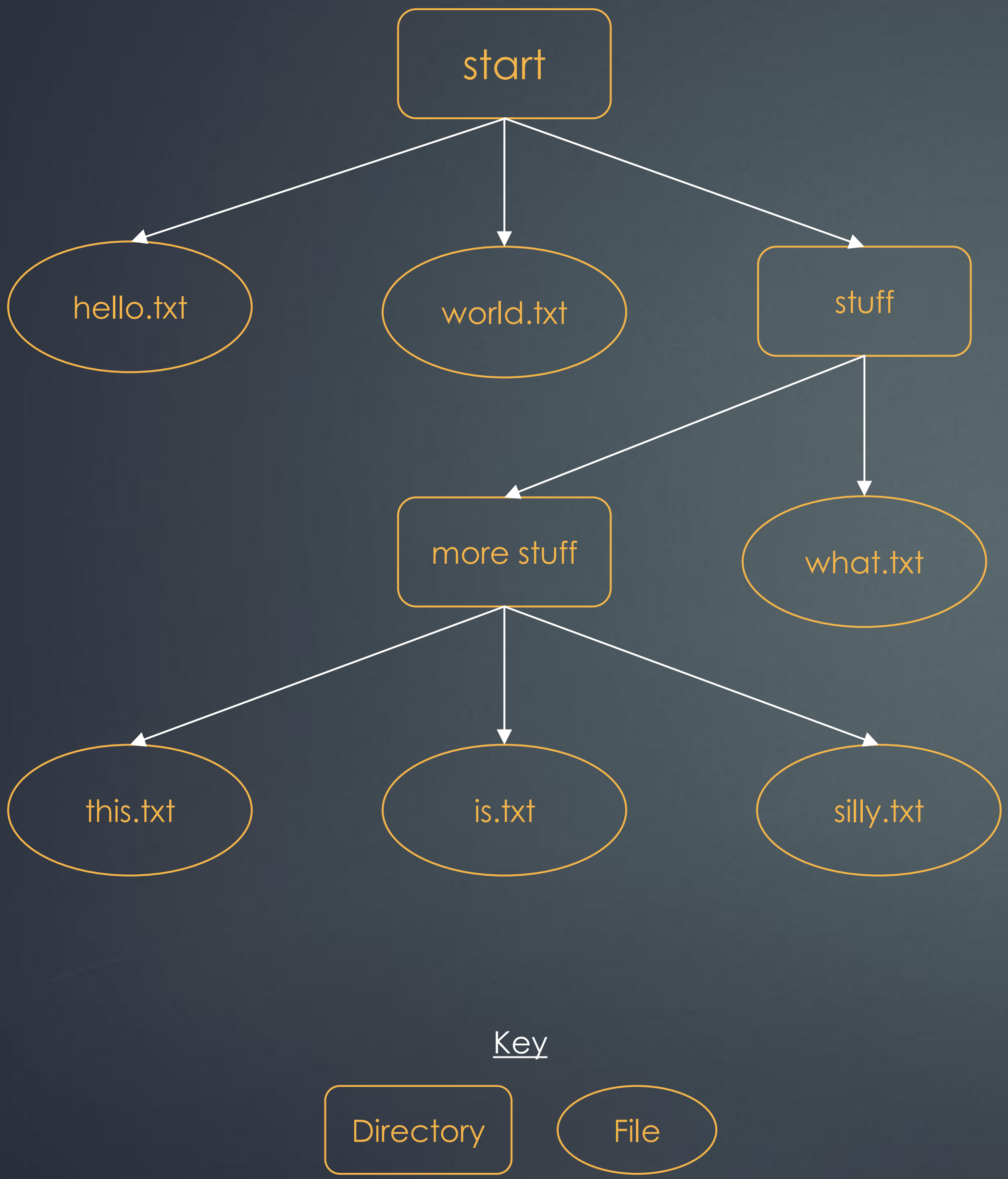
The leading bits in black are the only bits we care about for checking validity, and size of code points, all other bits in the top values represent the actual Unicode value

Example q8.c

- Write a program that reads a null-terminated UTF-8 string as a command line argument and counts how many Unicode characters (code points) it contains, assume all codepoints in the string are valid.

RECURSIVE FILE TRAVERSAL

Directories



Psuedocode for a program to print each file within a directory, assume we get next files from left to right

```
fn traverse(char *path)
  dir = open_directory(cur_location)
  print("Directory: " + path)

  while (file = get_next_file(dir)):
      if (IS_DIRECTORY(file)):
          if (file.path == "." || file.path == ".."):
              continue
          traverse(file.path)

      print("Regular: " + file.path)

  }
```

1. Directory: start
2. Regular: hello.txt
3. Regular: world.txt
4. Directory: stuff
5. Directory: more_stuff
6. Regular: this.txt
7. Regular: is.txt
8. Regular: silly.txt
9. Regular: what.txt

How to do this in C

- We can open/close a directory just like a file

```
DIR *dir = opendir(path);
```

- We can also open each entry within a directory, and use a loop to traverse through each entry

```
struct dirent *entry;
while((entry = readdir(dir))) {
   /// rest of your traversal
}
```

- From the dirent struct we can get the full path name by concatenating it with the directory path we are within (path + entry->d_name)

```
char file_path[MAX_PATH_LEN];
snprintf(file_path, MAX_PATH_LEN, "%s/%s", path, entry->d_name);
// this appeds the file onto the directory path, useful for further traversal
```

Example: directory_traversal.c

- Write a C program to traverse a directory, and print out the directory names, and the files within, printing when you enter a directory, and when you encounter a file.

FIN