Exploring the Filesystem

Icon made by Freepik from www.flaticon.com

File systems

Simple File System Simulator

Files

Everything in Linux is a represented as a file

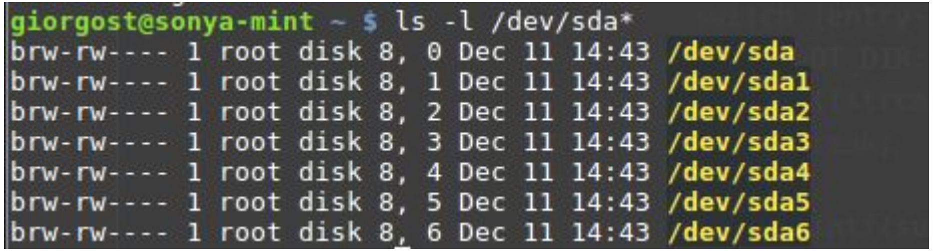
- - : regular file.
- d : **directory**.
- c : **character device** file.
- b : **block device** file.
- s : local socket file.
- p : named pipe.
- l : symbolic link.

Filesystems cont.

tmpfs - File system on the ram. Common for temporary storage

ext4 - Journaling/log base file system. Very common as the main fs for linux

Drives are actually represented as files which accepts blocks of data or characters:



```
giorgost@sonya-mint ~ $ ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 Dec 11 14:43 /dev/sda
brw-rw---- 1 root disk 8, 1 Dec 11 14:43 /dev/sda1
brw-rw---- 1 root disk 8, 2 Dec 11 14:43 /dev/sda2
brw-rw---- 1 root disk 8, 3 Dec 11 14:43 /dev/sda3
brw-rw---- 1 root disk 8, 4 Dec 11 14:43 /dev/sda4
brw-rw---- 1 root disk 8, 5 Dec 11 14:43 /dev/sda5
brw-rw---- 1 root disk 8, 6 Dec 11 14:43 /dev/sda6
```

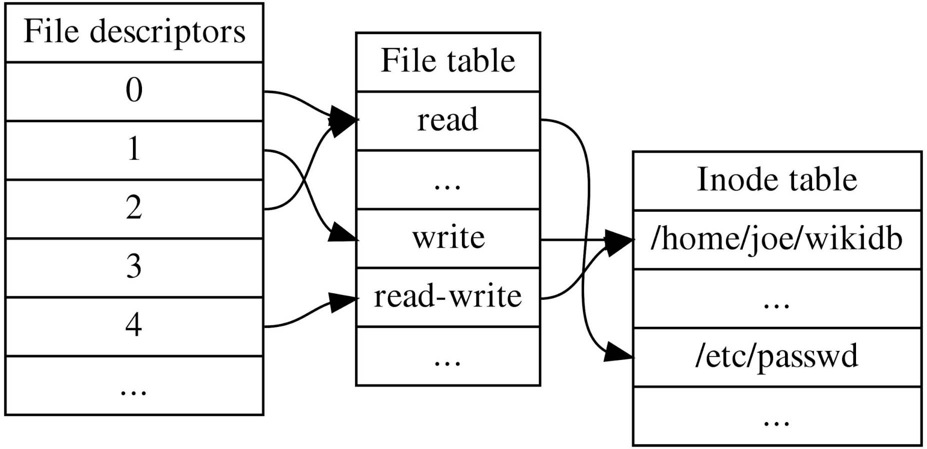
Inodes

Why is called an Inode?
“*In truth, I don't know either….. The **I** probably refers to **Index***” - *Dennis Ritchie*

The data structure that is actually written to disk

Inodes contains:

- Device ID
- Inode number
- Filemode (permissions)
- Links
- UID / GID
- Device ID
- Size
- Timestamps
- I/O Blocksize
- Number of blocks



By Qwertyus - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38970813

Vnodes

Virtual (in memory) representation of inode

Provide file system transparency: We no longer have to consider which type of file system the machine is using

Inode nr 0 - null inode

Inode nr 1 - Bad records

Inode nr 2 - The root in the file system

This “restarts” for each file system

Soft & Hard Links

Inodes contain:

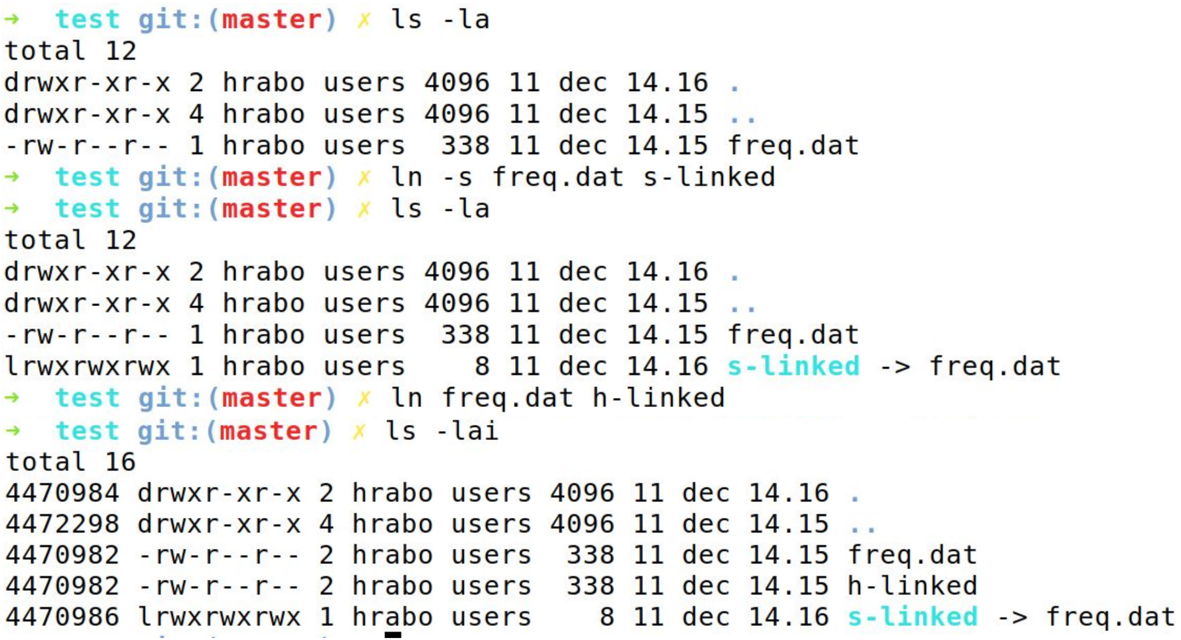
- **(number of) Links**

Softlinked

- A inode that points to a name in the filesystem

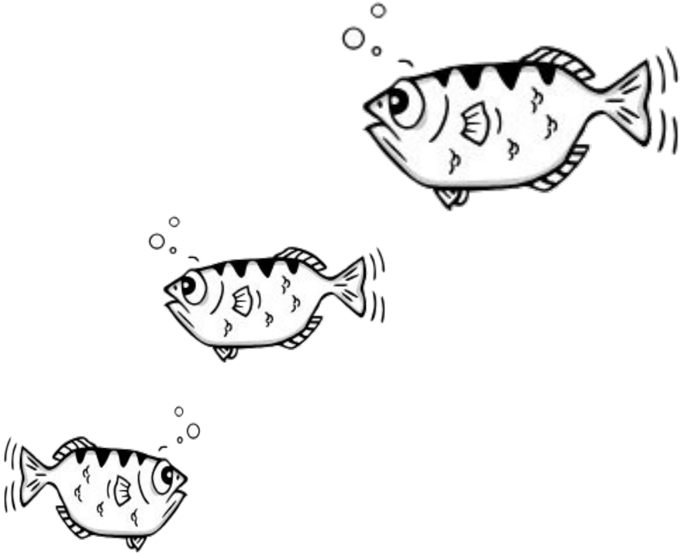
Hardlinked

- Just another entry in the directory table



```
➜  test git:(master) ✗ ls -la
total 12
drwxr-xr-x 2 hrabo users 4096 11 dec 14.16 .
drwxr-xr-x 4 hrabo users 4096 11 dec 14.15 ..
-rw-r--r-- 1 hrabo users  338 11 dec 14.15 freq.dat
➜  test git:(master) ✗ ln -s freq.dat s-linked
➜  test git:(master) ✗ ls -la
total 12
drwxr-xr-x 2 hrabo users 4096 11 dec 14.16 .
drwxr-xr-x 4 hrabo users 4096 11 dec 14.15 ..
-rw-r--r-- 1 hrabo users  338 11 dec 14.15 freq.dat
lrwxrwxrwx 1 hrabo users    8 11 dec 14.16 s-linked -> freq.dat
➜  test git:(master) ✗ ln freq.dat h-linked
➜  test git:(master) ✗ ls -lai
total 16
4470984 drwxr-xr-x 2 hrabo users 4096 11 dec 14.16 .
4472298 drwxr-xr-x 4 hrabo users 4096 11 dec 14.15 ..
4470982 -rw-r--r-- 2 hrabo users  338 11 dec 14.15 freq.dat
4470982 -rw-r--r-- 2 hrabo users  338 11 dec 14.15 h-linked
4470986 lrwxrwxrwx 1 hrabo users    8 11 dec 14.16 s-linked -> freq.dat
```

GNU Debugger (*GDB*)

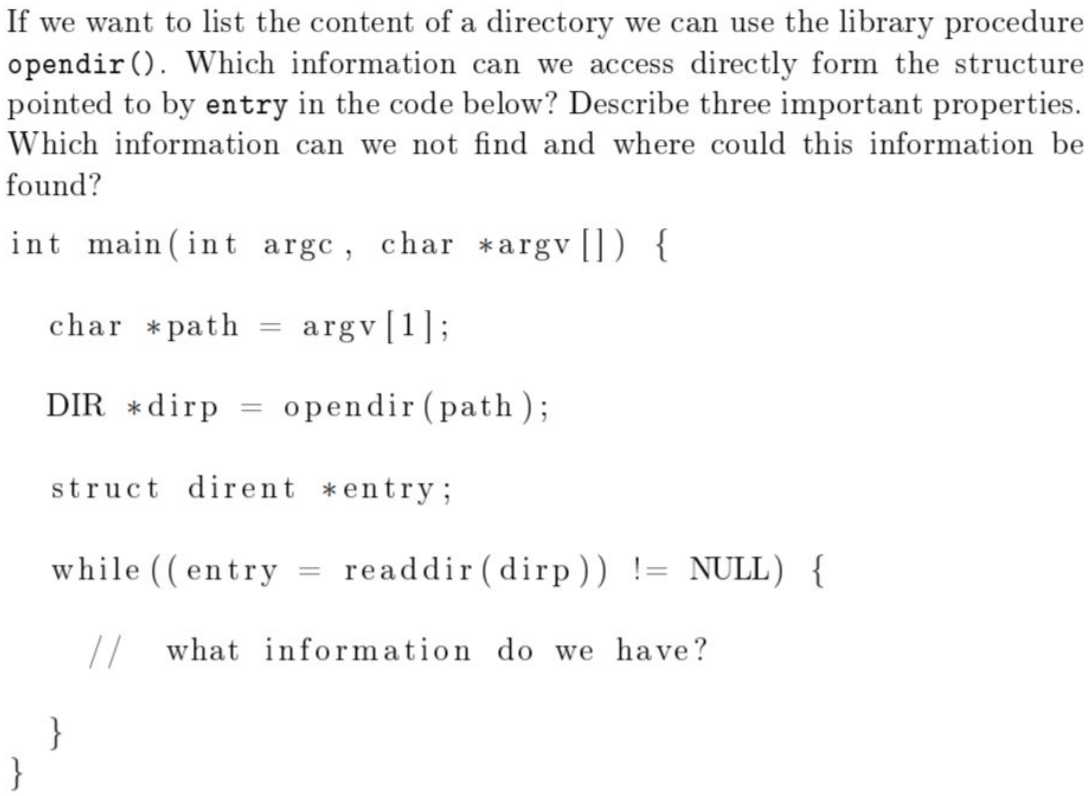


- The “low level” way of debugging C code in Linux
- Caches errors and provides more information than segmentation faults when the program crashes
- Is often integrated as the debugger in IDEs
- Compiling for GDB

```
gcc program.c -o program -g[gdb]
```

Exam Questions

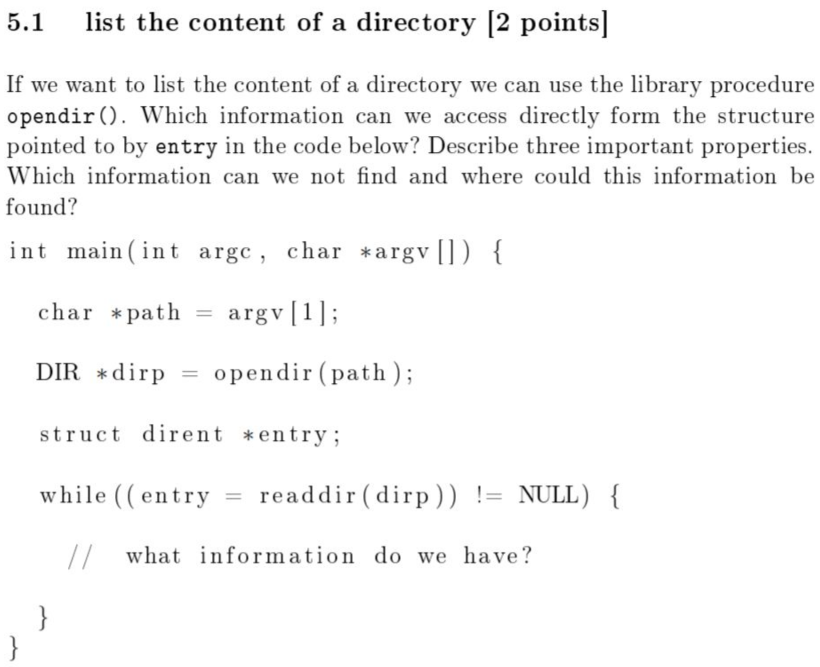
Exam question 1



If we want to list the content of a directory we can use the library procedure `opendir()`. Which information can we access directly form the structure pointed to by `entry` in the code below? Describe three important properties. Which information can we not find and where could this information be found?

```
int main(int argc, char *argv[]) {

  char *path = argv[1];

  DIR *dirp = opendir(path);

  struct dirent *entry;

  while((entry = readdir(dirp)) != NULL) {

    // what information do we have?

  }
}
```

Exam question 1



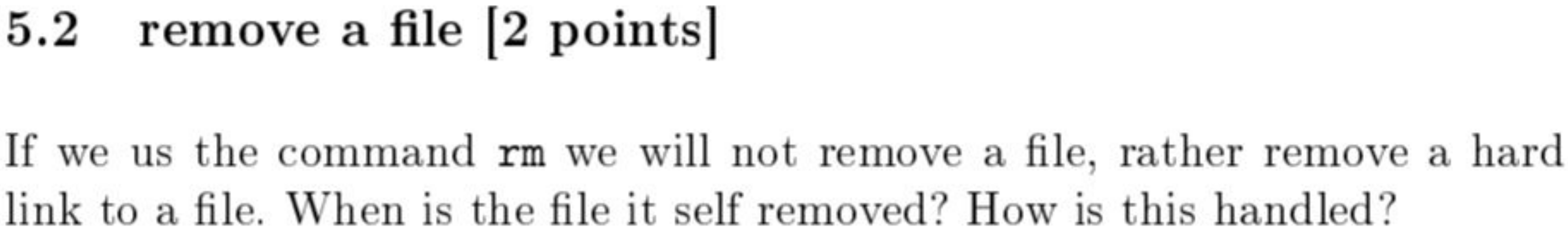
5.1 list the content of a directory [2 points]

If we want to list the content of a directory we can use the library procedure `opendir()`. Which information can we access directly form the structure pointed to by `entry` in the code below? Describe three important properties. Which information can we not find and where could this information be found?

```
int main(int argc, char *argv[]) {

  char *path = argv[1];

  DIR *dirp = opendir(path);

  struct dirent *entry;

  while((entry = readdir(dirp)) != NULL) {

    //  what information do we have?

  }
}
```

Answer: We can find name, type and inode number directly in this directory entry. The rest of the information is available in the inode.

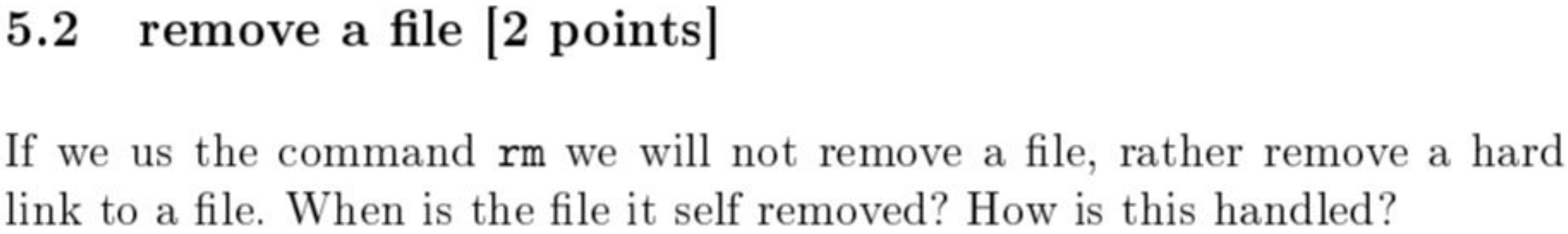
Exam question 2



5.2 remove a file [2 points]

If we us the command `rm` we will not remove a file, rather remove a hard link to a file. When is the file it self removed? How is this handled?

Exam question 2

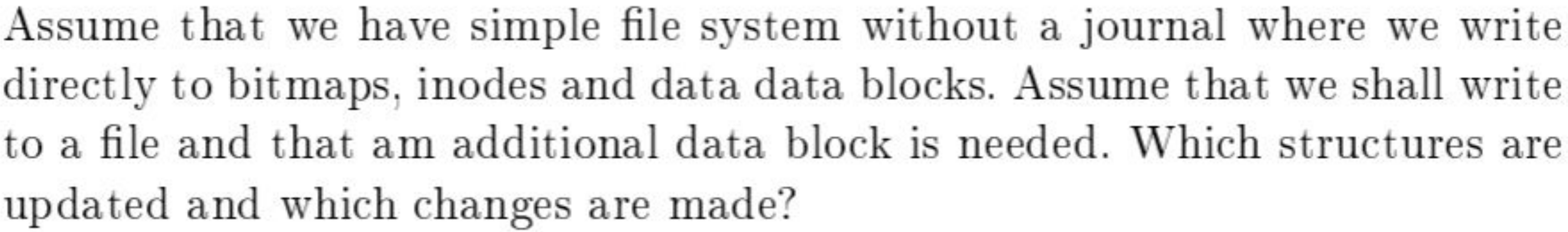


5.2 remove a file [2 points]

If we us the command `rm` we will not remove a file, rather remove a hard link to a file. When is the file it self removed? How is this handled?

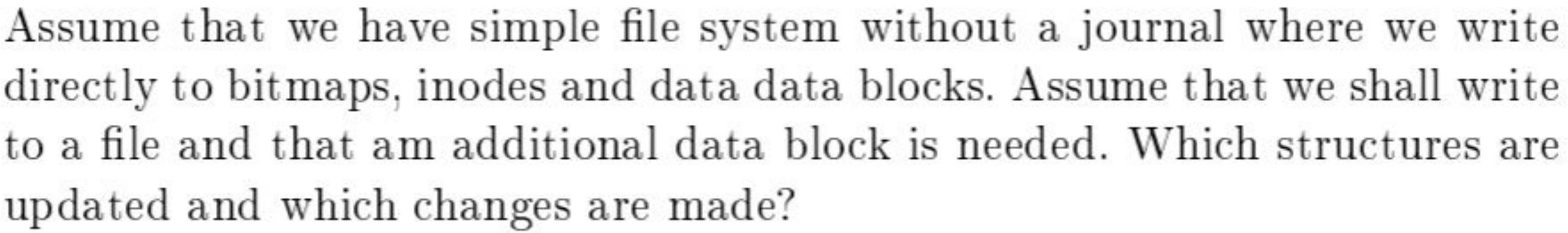
Answer: When using rm the inode is removed from the disk, while the actual data block remains. This data is only overwritten when it is needed later during the execution.

Exam question 3



Assume that we have simple file system without a journal where we write directly to bitmaps, inodes and data data blocks. Assume that we shall write to a file and that am additional data block is needed. Which structures are updated and which changes are made?

Exam question 3



Assume that we have simple file system without a journal where we write directly to bitmaps, inodes and data data blocks. Assume that we shall write to a file and that am additional data block is needed. Which structures are updated and which changes are made?

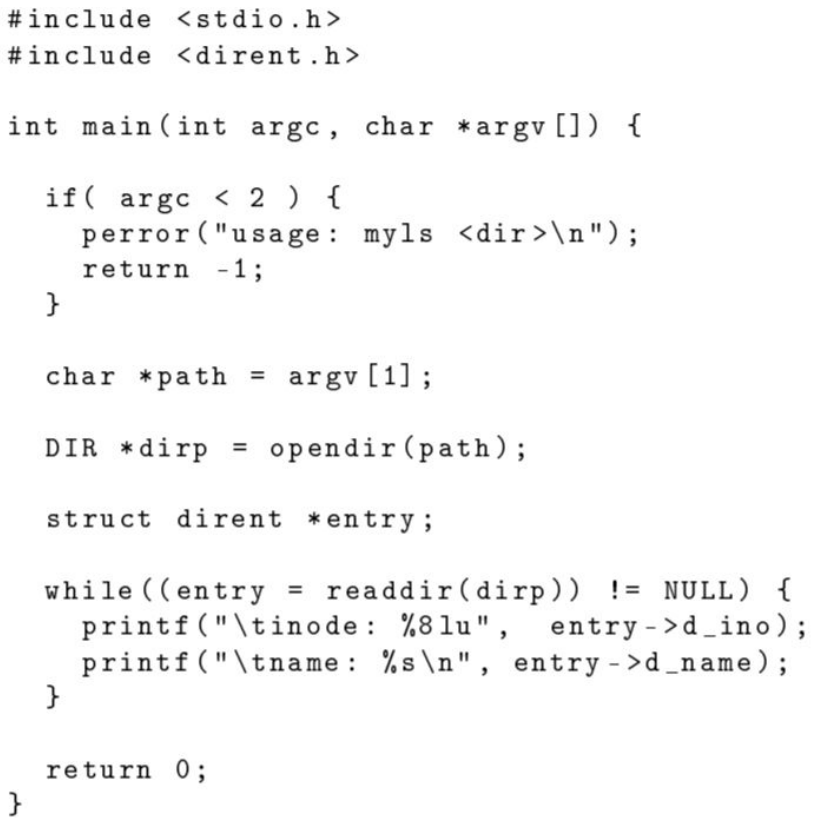
Answer:
We need to update the **bitmap** for used data blocks and mark the newly allocated data block as taken.
We also need to update the **Inode** for the file so that it includes the newly allocated data block.

Exam question 4

You are browsing you filesystem using the code displayed on the right.

Suddenly you get a segmentation fault.

What went wrong?



```
#include <stdio.h>
#include <dirent.h>

int main(int argc, char *argv[]) {

  if( argc < 2 ) {
    perror("usage: myls <dir>\n");
    return -1;
  }

  char *path = argv[1];

  DIR *dirp = opendir(path);

  struct dirent *entry;

  while((entry = readdir(dirp)) != NULL) {
    printf("\tinode: %8lu", entry->d_ino);
    printf("\tname: %s\n", entry->d_name);
  }

  return 0;
}
```

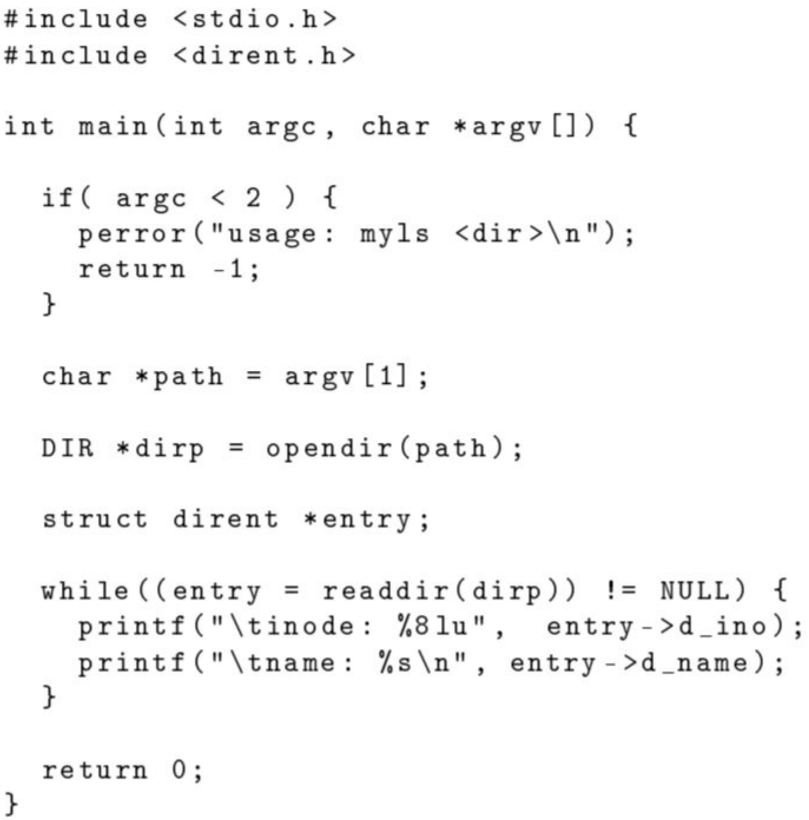
Exam question 4

You are browsing you filesystem using the code displayed on the right.

Suddenly you get a segmentation fault.

What went wrong?

Answer: The opendir function returns a null pointer if the user does not have permission to read the file.



```c
#include <stdio.h>
#include <dirent.h>

int main(int argc, char *argv[]) {

  if( argc < 2 ) {
    perror("usage: myls <dir>\n");
    return -1;
  }

  char *path = argv[1];

  DIR *dirp = opendir(path);

  struct dirent *entry;

  while((entry = readdir(dirp)) != NULL) {
    printf("\tinode: %8lu", entry->d_ino);
    printf("\tname: %s\n", entry->d_name);
  }

  return 0;
}
```