Joachim Grimen Westgaard

Assignment 4

1 File systems

1. Name two factors that are important in the design of a file system.

When designing a file system, we need it to be (among other factors) fast, flexible, persistent, and reliable. We can achieve a fast file system by striving for *spatial locality*, where we store blocks (that are accessed together) ideally sequentially.

Flexibility is achieved through the designing the file system as a 'jack-of-all-trades'. It must allow applications to share data and tackle all sorts of file-manipulation (whether the file is small, large, never written to, randomly accessed and so on...).

Persistence is achieved through the file system being able to maintain and update data and its structures on the storage devices that we use (e.g. the connected SSD). This is to ensure that the data is not lost if a system crash occurs.

Reliability is achieved through the file system being able to important data for an extended period and defend them in the cases of system crashes and storage hardware malfunctions.

2. Name some examples of file metadata.

A file's metadata includes information such as a file's access time, owner ID, permissions, size, creation time, last modified time and whether the file is a directory or not.

2 Files and directories

1. Fast File System (FFS)

a. Explain the difference between a hard link and a soft link in this file system. What is the length of the content of a soft link file?

A hard link is when different directories map different path names to the same file number, while a soft link is a link that *points* to another file number. In other words, a hard link points a filename to data on a storage device, while a soft link points a filename to another filename which then points to data on the storage device.
A soft link can point to a directory, while a hard link is not. This is to avoid recursiveness within the file system.

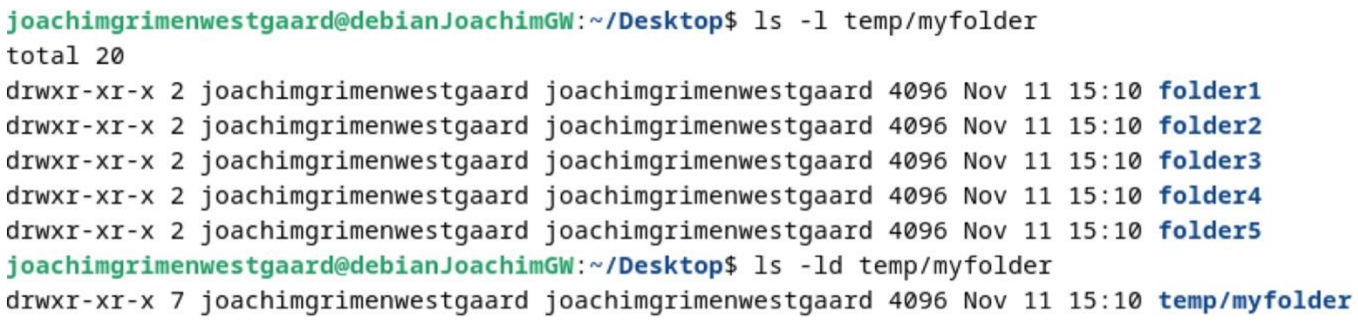
The size of a soft link is the same as the length of the path of the original file.

b. What is the minimum number of references (hard links) for any given folder?

The minimum number of references is two. It has a hard link to its own 'i-number' (can be read with 'ls -i' in the terminal, and to the entry '.'. Every subdirectory inside of a directory will have a hard link to its parent directory called '..'.
To see the hidden folder '.' and its i-number, we can use the command 'ls -ldi'.

In other words, the minimum is 2, where one belongs to '.' and one belongs to itself.

c. Consider a folder /tmp/myfolder containing 5 subfolders. How many references (hard links) does it have? Try it yourself on a Linux system and include the output. Use ls -ld /tmp/myfolder to view the reference count (hint, it's the second column in the output).



```
joachimgrimenwestgaard@debianJoachimGW:~/Desktop$ ls -l temp/myfolder
total 20
drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder1
drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder2
drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder3
drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder4
drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder5
joachimgrimenwestgaard@debianJoachimGW:~/Desktop$ ls -ld temp/myfolder
drwxr-xr-x 7 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 temp/myfolder
```

We can see that there are 7 hard links in the folder '/temp/myfolder' which has 5 empty subdirectories inside. These 7 hard links belong to each one of the 'folderX' subdirectories, while the remaining 2 belong to the '.' and itself.

d. Explain how spatial locality is acheived in a FFS.

Spatial locality is in an FFS achieved through the implementation of 'locality heuristics'. Locality heuristics are the embodiments of certain policies that we employ to achieve a high percentage of sequential ordering in our file system.

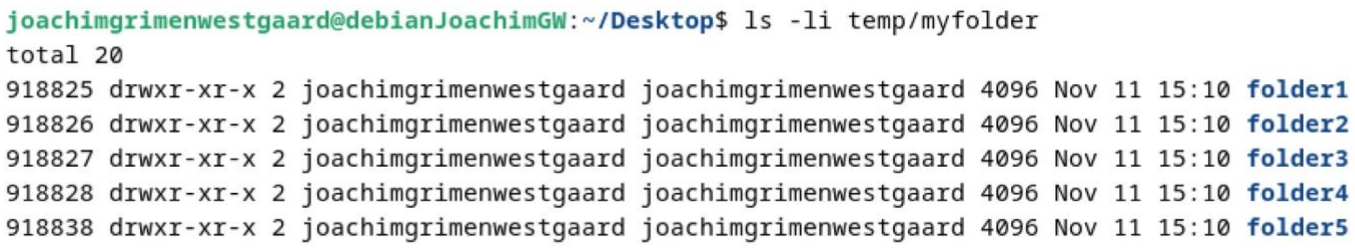
For example, we might want files inside of the same directory to be stored sequentially on our storage device. We can achieve this by ensuring that we either store the files that we want sequentially, sequentially when first written, or we could take advantage of periodic defragmentation. When we defragment, we can move the written files around on the storage device, to increase the amount of sequential data (that we wanted to be sequential in the first place but decided to fix later down the road).

Basically, spatial locality refers to the principal of that if a memory location is accessed, there is reason to believe that memory locations nearby will be accessed soon. Therefore, by actively making sure that related data (through folders e.g.) can be accessed in sequence, we can speed up access times on the storage device.

We employ these policies when storing data on our storage device, due to our ability to locate the empty/free spaces on the storage device through 'free space maps' that we can retrieve using directories and index structures in our file system.

In the case of an FFS, we can strive to ensure good allocations of i-numbers, such that they correspond to the actual file layout. If we for example consider the subfolders inside of '/temp/myfolder', we would want these to have i-numbers that are close together.

We can see that this is indeed the case in our example:



```
joachimgrimenwestgaard@debianJoachimGW:~/Desktop$ ls -li temp/myfolder
total 20
918825 drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder1
918826 drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder2
918827 drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder3
918828 drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder4
918838 drwxr-xr-x 2 joachimgrimenwestgaard joachimgrimenwestgaard 4096 Nov 11 15:10 folder5
```

2. NTFS

a. Explain the differences and use of resident versus non-resident attributes in NTFS.

In FFS we have the 'inode array', while we in the NTFS have the 'master file table' (MFT). Inside the MFT, we store 'MFT records' that are of size 1KB each. Each of these records store a sequence of variable-size 'attribute records'. In NFTS, we store

both data and metadata inside of these MFT records, both of which are considered attributes of a file.
Some of these attributes can be too large to fit inside an MFT record. This is where we differentiate between attributes being either a 'resident' or a 'non-resident'. A resident attribute stores its data directly inside of the MFT record, while the non-resident stores pointers to a 'data extent' where its data is stored instead.

We use the resident attributes to store data directly on the MFT record, while the non-resident attributes are stored in a special 'extension' (where we instead of data, store the pointers to the extensions inside the MFT record).

b. Discuss the benefits of NTFS-style extents in relation to blocks used by FAT or FFS.

Extents can be of variable size. This leads to the possibility of having a file with a 'shallow' index tree, because these individual extents can be long/large. This also leads to us having a less fragmented file (as we would have with a deeper index tree).

3. Explain how copy-on-write (COW) helps guard against data corruption.

COW ensures that the new version of a file is written to a free space of the storage device, and that this operation is finished (like a strict transaction) before the file has its 'i-node' updated. This minimizes the possibility of data corruption, as we do not update the file piece by piece, but rather as a single step (transaction).

3 Security

1. Authentication

a. Why is it important to hash passwords with a unique salt, even if the salt can be publicly known?

When we hash the string "hello", we will get a hashed string that will look the same as the next "hello" that gets hashed. This means that we do not have any inherent uniqueness for any string. However, by adding "salt" to the hashing function (a random input added to the hash function to randomize the final hashed string) we get inherent uniqueness even for two strings that both are "hello" to begin with.
By applying the salt, we avoid the danger of a "rainbow table" being able to maliciously retrieve our password.

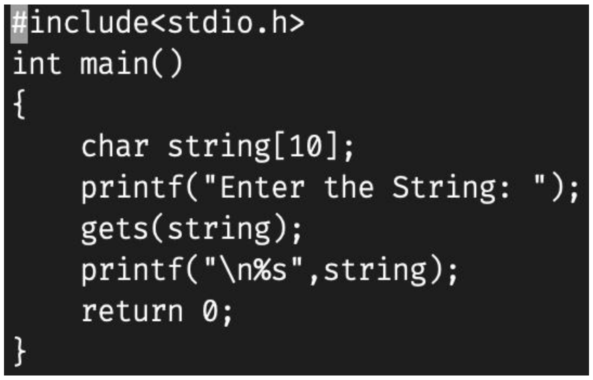
b. Explain how a user can use a program to update the password database, while at the same time does not have read or write permissions to the password database file itself. What are the caveats of this?

A user might run a program where the "setuid" permissions are altered to have root access. This program might now be able to access the password database, although the user itself does not have direct access to it. The program is sort of used as a middleman.
This highlights our need to be careful when assigning setuid-permissions, especially when considering the root permission.

2. Software vulnerabilities

a. Describe the problem with the well-known gets() library call. Name another library call that is safe to use that accomplishes the same thing.

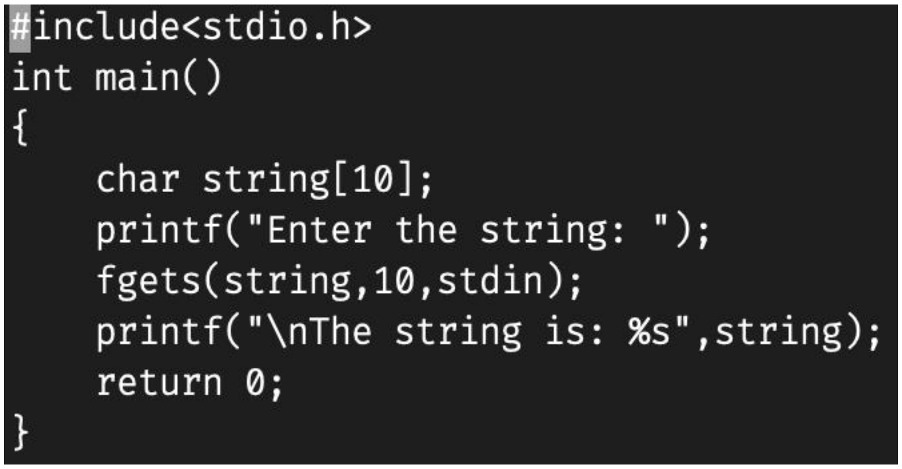
The function "gets()" only stops reading from "stdin" when it explicitly encounters a newline character ("\n"). It does not stop when it encounters an EOF-declaration, nor if it already has read the entire input it is given. Let us consider the following program:



```
#include<stdio.h>
int main()
{
    char string[10];
    printf("Enter the String: ");
    gets(string);
    printf("\n%s",string);
    return 0;
}
```

We get a warning both when compiling and running the program. If we decide to ignore these and input a string whose length is equal to 9 or more (excluding the newline character that also is considered when reading from stdin), the program will abort.

If we instead use "fgets()", like this:



```
#include<stdio.h>
int main()
{
    char string[10];
    printf("Enter the string: ");
    fgets(string,10,stdin);
    printf("\nThe string is: %s",string);
    return 0;
}
```

The program will only consider the first 9 (excluding the newline character) but will not abort. This example shows why gets() is less safe than fgets(), and why fgets() is preferred.

In other words, gets() does not verify its input before handling it, which might lead to the abortion error that we encountered. This also renders it less safe, as it might provide a way for a malicious attacker to corrupt our stack/heap (depending on where the buffer from gets() is stored).

b. Explain why a microkernel is statistically more secure than a monolithic kernel.

A microkernel is more modular than a monolithic kernel and will therefore be more resilient to system crashes. This can lead to a system crash not affecting the kernel at all, as the kernel memory space is reserved for kernel-specific services like IPC, memory management and scheduling.
The monolithic kernel has all its OS-related services running inside the same memory space (this includes both user and kernel services), which might lead to the kernel being more susceptible to damage if a system crash were to happen. This might prove itself to be a security risk.

This effectively means that a microkernel has less code being executed in kernel-mode than a monolithic kernel. We can consider the microkernel more secure due to this fact, as there will be a lower risk of executing malicious code in kernel-mode.