Innlevering-04.md

OBL4 - Operating Systems

1. File systems

1.1 Two important factors in file system design

1. **Data integrity and reliability** – Ensuring that data is stored and retrieved accurately, and that the file system can recover from crashes or corruption.
2. **Performance and efficiency** – Optimizing access speed, storage utilization, and scalability to handle different workloads and hardware.

1.2 Examples of file metadata

Metadata is a form of *data about the data* which i like to call it. This usually includes:

- **Filename** – The name of the file.
- **File size** – The amount of data stored in the file.
- **Timestamps** – Information such as creation time, last modification time, and last access time.
- **Permissions** – Access control information specifying who can read, write, or execute the file.
- **Owner and group** – The user and group that own the file.
- **File type** – Indicates whether the file is a regular file, directory, symbolic link, etc.
- **Location on disk** – Pointers to the physical blocks where the file's data is stored.

2. Files and directories

2.1 Fast File System (FFS) - ext4

a) Hard link vs soft link

A **hard link** is a direct reference to the actual data on disk; multiple hard links to a file are indistinguishable and remain valid as long as at least one exists. Deleting one hard link does not affect access to the file as long as other hard links remain. In contrast, a **soft link** (symbolic link) is a special file that points to another file or directory by path. If the target is deleted or moved, the soft link becomes broken and no longer provides access to the data.

b) Minimum number of hard links for a folder

the minimum number is `2` : one link is for the directory's name in its parent, and the other is the `.` entry (self-reference) inside the directory itself.

Example: `<dirname>/.` with the `.` refers to itself

c) Reference count for /tmp/myfolder with 5 subfolders

In Linux, every directory keeps a count of how many hard links point to it. The reference count for a directory is always at least 2: one for the directory itself (`.`) and one for its entry in the parent directory. Each time you add a subfolder, that subfolder's `..` entry points back to the parent, increasing the parent's reference count by 1.

Now lets test it out physically:

```
mkdir -p /tmp/myfolder && for i in 1 2 3 4 5;
do mkdir -p /tmp/myfolder/sub$i;
done && ls -ld /tmp/myfolder
```

Simple for loop to create subfolders, `-p` makes parent directories as needed, `-ld` lists detailed directory information

Explanation

5 subfolders result in 7 hard links because 2 links are always present: one for the directory itself, and one for the `.` entry. Each subfolder adds 1 more link via its `..` entry pointing back to the parent.

`2 + 5 = 7` hardlinks; `5` subfolders `2` self + parent

with the included `ls -ld /tmp/myfolder` we will get `drwxrwxr-x 7` showing the `7` hard links as expected.

d) Spatial locality in FFS

Lets answer what `spartial locality` is first: spartial locality refers to the tendency of programs to access data locations that are physically close to each other. This is important for performance because accessing contiguous blocks of data is faster than accessing scattered blocks due to reduced seek times and caching efficiency.

In `FFS` (*Fast File System*), spatial locality is achieved by organizing files and directories in a way that related data is stored close together on the disk. This minimizes the distance the read/write head must travel, leading to faster access times and improved overall performance.

Analogy in reallife: Imagine a group of students in a school: students taking the same subject (like Math) are seated together in the same classroom, while students in different subjects (like History or Science) are in nearby rooms. If a student needs to quickly collaborate with classmates or switch between related subjects, having everyone close by saves time and effort—just like how FFS stores related files and directories close together on disk to speed up access.

2.2 NTFS (Windows file system) - Flexible tree with extents

To explain the answer to this questions, we must know about NTFS and MFT (Master File Table) entries in NTFS.

- NTFS is a file system developed by Microsoft that uses a Master File Table (MFT) to manage files and directories.
- Each file or directory is represented by an MFT entry, which contains metadata about the file, including its attributes.

a) Resident vs non-resident attributes

- **Resident attributes** are stored directly within the MFT entry itself. This is typically used for small files or metadata that can fit within the limited space of the MFT entry. Since the data is stored in the MFT, access to resident attributes is faster because it avoids additional disk reads.
- **Non-resident attributes**, on the other hand, are stored outside the MFT entry in separate clusters on the disk. This is used for larger files that cannot fit within the MFT entry. The MFT entry contains pointers to the locations of these non-resident attributes on the disk. Accessing non-resident attributes requires additional disk reads, which can be slower than accessing resident attributes.

Concrete explanation:

- Resident attributes stored directly in MFT.
- Non-resident attributes stored outside MFT with pointers.

b) Benefits of NTFS-style extents

NTFS-style extents offer several benefits:

1. **Reduced fragmentation**: By allocating contiguous blocks of disk space for files, extents minimize fragmentation, leading to improved read/write performance.
2. **Efficient storage management**: Extents allow for dynamic allocation of disk space, which can lead to better utilization of available storage.
3. **Faster file access**: Since extents can represent large contiguous areas of storage, accessing files can be faster due to reduced seek times.
4. **Scalability**: Extents can handle large files and volumes more efficiently, making NTFS suitable for modern storage needs.

2.3 Copy-on-write (COW) and data corruption

Copy-on-write (COW) is a technique used in file systems and memory management where data is not immediately copied when a modification is made. Instead, the original data remains unchanged until a write operation occurs. When a write operation is initiated, a new copy of the data is created, and the modifications are applied to this new copy. This approach helps to optimize resource usage and improve performance.

Analogy: Consider a shared document in a collaborative editing application. When multiple users are viewing the document, they all see the same version. If one user decides to make changes, instead of altering the original document, the application creates a new copy of the document for that user to edit. The original document remains intact for other users until the changes are finalized and saved.

3. Security

3.1 Authentication

a) Importance of hashing passwords with a unique salt

Lets first define what `salting` is: Salting is the process of adding a unique, random value (the salt) to a password before hashing it. This ensures that even if two users have the same password, their hashed passwords will be different due to the unique salt.

Then define `rainbow tables` which are precomputed tables used to reverse cryptographic hash functions, primarily for cracking password hashes.

Hashing passwords with a unique salt is crucial for security in several ways:

1. **Prevents rainbow table attacks**: A salt adds randomness to the password before hashing, making precomputed hash tables (rainbow tables) ineffective.
2. **Unique hashes for identical passwords**: Even if two users have the same password, the unique salt ensures that their hashed passwords are different, preventing attackers from identifying common passwords.
3. **Increases complexity**: Salting increases the complexity of the hashing process, making it more difficult for attackers to use brute-force methods to crack passwords.
4. **Protects against database breaches**: In the event of a database breach, salted hashes make it significantly harder for attackers to reverse-engineer the original passwords.

Reallife example where breach without salt happened: In 2012, LinkedIn suffered a major data breach where millions of user passwords were exposed. The passwords were hashed without any salt, making it easier for attackers to use rainbow tables to crack the passwords. This incident highlighted the importance of using unique salts in password hashing to enhance security.

b) Updating password database without direct permissions

To update a password database without direct permissions, one common method is to use a privileged service or process that has the necessary permissions to modify the database. This can be achieved through:

1. **Using a setuid program**: A setuid (set user ID) program runs with the privileges of the file owner, allowing a user to execute a program with elevated permissions temporarily.
2. **Inter-process communication (IPC)**: A user can send a request to a privileged service via IPC mechanisms (like sockets or message queues) to update the password database on their behalf.

Knowledge from fullstack (IDG2100) on web server, example: A web application have a backend service running with elevated privileges that handles password updates. When a user requests a password change through the web interface, the application sends the request to this backend service, where it includes the necessary authentication and authorization information (user specified or admin privelages), which then updates the password database securely without exposing direct write permissions to the user.

3.2 Software vulnerabilities

a) Problem with gets() and safer alternative

The `gets()` function is problematic because it does not perform bounds checking on the input it receives. This means that if a user inputs more data than the allocated buffer can hold, it can lead to a buffer overflow. Buffer overflows can overwrite adjacent memory, potentially allowing attackers to execute arbitrary code, crash the program, or manipulate data.

Example of bounder overflow with gets():

```
#include <stdio.h>
int main() {
    char buffer[10];
    printf("Enter some text: ");
    gets(buffer); // Unsafe function
    printf("You entered: %s\n", buffer);
    return 0;
}
```

If a user inputs more than 9 characters (plus the null terminator), it will overflow the buffer, leading to undefined behavior. such as overwriting the return address of the function, which can be exploited by attackers. You should use `fgets()` instead, which allows you to specify the maximum number of characters to read, preventing buffer overflows.

```
#include <stdio.h>
int main() {
    char buffer[10];
    printf("Enter some text: ");
    fgets(buffer, sizeof(buffer), stdin); // Safe function
    printf("You entered: %s\n", buffer);
    return 0;
}
```

b) Microkernel vs monolithic kernel security

First to define both kernel types:

- **Monolithic kernel**: A monolithic kernel is a single large process running entirely in a single address space. It includes all the core functionalities of the operating system, such as device drivers, file system management, and system calls.
- **Microkernel**: A microkernel is a minimalistic kernel that only includes the most essential functions, such as inter-process communication and basic scheduling.

Microkernels are generally considered more secure than monolithic kernels for several reasons:

1. **Reduced attack surface**: Microkernels have a smaller codebase running in kernel mode, which reduces the number of potential vulnerabilities that can be exploited by attackers.
2. **Isolation of services**: In a microkernel architecture, many services (like device drivers and file systems) run in user space rather than kernel space. This isolation means that if a service is compromised, it does not compromise the entire system.
3. **Easier to review and maintain**: The smaller size of the microkernel makes it easier for developers and security experts to carefully check (audit) the code for security vulnerabilities and maintain it, allowing for quicker identification and patching of security issues.
4. **Fault tolerance**: Since services run in user space, a failure in one service does not crash the entire system, enhancing overall system stability and security.