Operating System

Memory Management

Address Space

- Memory of a process is divided into segments. This way of arranging memory is called **segmentation**
- **Logical address space**: **Each process has its own address space**, and it can reside in any part of the **physical memory**. Therefore the process may be cut up and not starting from 0x0000_0000 actually!
- user-space memory is about:
    1. Where does the address point to?
    2. Is the memory address valid or allocated?
    3. Is the permission granted?

Code & Constants

- A program is an executable file
- A process is **not bounded** to one program code.
- A program is an executable file
- Codes and constants are both **read-only**

Data Segment & BSS (Block Started by Symbol)

- A **static variable** is treated as the same as a **global variable** and Only the compiler cares about the difference

Difference	Data Segment	BSS
Property	Containing *initialized* global and static variables	Containing *uninitialized* global and static variables
Location	Lower address	Higher address
Size	Has the required space	Is just a bunch of symbols. The space is *not yet allocated* to the process which will be allocated once it starts executing

- Everything in a computer has a limit: **On a 32-bit Linux system, the user-space addressing**

  space is around 3GB

Stack

- Stores
  - all the local variables
  - all function parameters
  - program arguments
  - environment variables
- When the stack shrinks, the memory is **not** returned to the OS
- A function can ask the CPU to read and to write **anywhere** in the stack, not just the "zone" belonging to the running function!

Heap

- **Dynamic**: not defined at compile time
- **Allocation**: only when you *ask* for memory, you would be *allocated* the memory (Lazy allocation)
- Is it possible to run **OOM** (Out of Memory)?
  - Use `memset` when doing the malloc
- External & Internal Fragmentation
  - External fragmentation
    - The heap memory looks like a map with many holes
    - It is the source of inefficiency because of the **unavoidable search for suitable space**
    - The memory wasted because **external fragmentation is inevitable**
  - Internal fragmentation
    - Payload is smaller than allocated block size
    - Padding for alignment
    - Placement policy
      - Allocate a big block for small request (would cause this)

Segmentation Fault

- When you are accessing a piece of memory that is not allowed to be accessed, then the OS returns you an error called – segmentation fault.
- (WikiPedia) In computing, a **segmentation fault** (often shortened to **segfault**) or **access violation** is a fault, or failure condition, raised by hardware with memory protection, notifying an operating system (OS) the software has attempted to access a restricted area of memory (a memory

access violation).
- The memory in a process is separated into **segments**. So, when you visit a segment in an illegal way, then...**segmentation fault**.

Mass Storage

Disk Structure

- Cylinder, track, sector
- Use:
  - Address mapping
  - Bad block management
    - Maintain a list of bad blocks (initialized during low-level formatting) and preserve an amount of spare sectors
    - **Sector sparing/forwarding:** replace a bad sector logically with one spare sector (spare sectors in each cylinder + spare cylinder in order to prevent invalidate disk scheduling algorithm problem)
    - **Sector slipping:** remap to the next sector (data movement is needed)
  - Disk Formatting
    - **Step 1: Low-level formatting/physical formatting**
      - Divide into sectors
      - Fills the disk with a special data structure for each sector (data area(512B), header and trailer (sector number & ECC))
      - Done at factory, used for testing and initializing
    - **Step 2: How to use disks to hold files after shipment?**
      - FS
      - Raw disk

Disk Scheduling

- First-come, first-served (**FCFS**)
- Shortest-seek-time-first (**SSTF**): Choose the request closest to the current head position
- **SCAN** (**Elevator algorithm**): Starts at one end, moves toward the other end and reverses
- **LOOK**: Goes only as far as the final request
- **C-SCAN**: Circular scan back and forth
- **C-LOOK**

RAID

- Purpose :
    - In the past, combine small and cheap disks as a **cost-effective** alternative to large and expensive disks
    - Nowadays
        - Higher performance
        - Higher reliability via redundant data
        - Larger storage capacity
- RAID-0: Block level stripping, no redundancy
- RAID-1: Data mirroring
- RAID-01: First stripping, then mirroring
- RAID-10: On the contrary of 👆
- RAID-4: Parity generation: Each parity block is the XOR value of the corresponding data disks $A_p = A_1 \otimes A_2 \otimes A_3$
    - RMW (read modify write) $A'_p = A_p \otimes A_1 \otimes A'_1$
    - RRW (read reconstruct write) $A'_p = A_3 \otimes A'_2 \otimes A'_1$
    - Problem: **Imbalance**
        - Disk bandwidth are not fully utilized
            - Parity disk will not be accessed under normal mode
            - Parity disk may become the bottleneck
- RAID-5: One parity per stripe
    - Key difference: Uniform parity distribution
- RAID-6: 2 parities

File System

Programmer's perspective

- What are stored inside a storage device?
    - File
    - Directories
    - Interfaces/Operations
- Layout
    - what are stored inside the device
    - Where the stored things are
    - The set of FS operations defines how the OS should work with the FS layout. In other words, OS knows the FS layout and works with that layout.
- There are **two basic things** that are stored inside a storage device, and are common to all existing file systems: File and Directories
- How does a FS store data into the disk? – That is, the **layout** of file systems.

Why do we need files

- File provides a long-term information storage.
- File is also a shared object for processes to access concurrently.
- A unique pathname lead to the file's attributes and its content, which are usually stored **separately**

File permissions

- **First field:** File/director
- **2nd /3rd /4th fields (3 bits each):** controls read/write/execute
- for the file owner/file’s group/others (e.g., 111:7,110:6)

Opening a File

1. The process supplies a path name to the OS.
2. The OS looks for the **file attributes** of the target file in the disk.
3. The disk returns the file attributes.
4. The OS then associates the attributes to a number and the number is called the **file descriptor**.
5. The OS returns the file descriptor to the process.
6. Opening a file only involves the **pathname** and the **attributes of the file**, instead of the file content!

Read From Opened Files

1. The process supplies a file descriptor to the OS.
    1. A file descriptor is just an **array index** for each process to locate its **opened files**.
2. The OS reads the file attributes and uses the stored attributes to **locate the required data**. (In the OPEN FILE TABLE !)
3. The disk returns the required data. -- File data is stored in a fixed size **cache in the kernel**.
4. The OS fills the **buffer provided by the process** with the data. Write data to the userspace buffer.

Read System Call

1. Check whether the end of the file is reached or not. [Comparing **size** and **file seek**.]
    1. File attributs: Name, Identifier (*Unique tag (a number which identifies the file within the FS)*), Type, Location, Size, Owner, Permission, Access, creation, modification time, etc.
    2. Runtime Attributes: reading position count, etc
2. Reading data.
3. File data is stored in a fixed size cache in the kernel.
4. Write data to the userspace buffer.

Write System call

1. Write data to the kernel buffer.
2. According to the data length,
    1. change in file size, if any (File attributes)
    2. change in the file seek (Runtime attributes)
3. The call returns
4. The buffered data will be flushed to the disk from time to time.

Directories

- It's a file
- Whether it has file attributes is FS-dependent
- It must have file content

Locate a File Using Pathname

> *Suppose that the process wants to open the file "**/bin/ls**".*

1. The process then supplies the OS the unique pathname "**/bin/ls**".
2. The OS retrieves the directory file of the root directory '**/**'.
3. The disk returns the directory file.
4. The OS looks for the name "bin" in the directory file.
5. If found, then the OS retrieves the directory file of "**/bin**" using the information of the file attributes of "**bin**".
6. The OS looks for the name "**ls**" in the directory file "**bin**". If found, then the OS knows that the file "**/bin/ls**" is found, and it starts the previously-discussed procedures to open the file "**/bin/ls**"

Creation and Deletion

- **File creation == Update of the directory file**
    - "`touch text.txt`" will only create the directory entry, and there is no allocation for the file content.
- Removing a file is just delete the information in Directory file.
    - Note that we are not ready to talk about de-allocation of the file content yet.

File System Layout

Trial 1.0: The Contiguous Allocation

- Drawbacks:
    - External Fragmentation (We have enough space, but there is no holes that I can satisfy the request.) Therefore we need to move files to clean up enough space.

    - When a file need to grow, it may also do not have enough space.
    - Used suitable for read-only cases

Trial 2.0: The Linked List Allocation

- Drawbacks:
    - A Complicated Root Directory: need to store all sequences of a file, e.g. 7-18 & 26-27 & 45 & ...

Trial 2.1: The Linked List

- Characters:
    - Borrow 4 bytes from eack block, to write the next block number into
    - Root directory become a REAL linked list, yet the file size should be stored since the last block might not be fully used
    - Free Space is extra stored
- Merits:
    - **External** fragmentation problem is solved.
    - Files can grow and shrink freely.
    - Free block management is easy to implement.
- Drawbacks:
    - **Random access** performance problem.
    - Internal Fragmentation

Trial 2.2: the FAT

- All the information about the next block #s are centralized, and it is called FAT.
- The random access problem can be eased by keeping a **cached version of FAT** inside the kernel.
- If this table is partially kept on the cache, then **extra I/O requests** will be generated in locating the next block #.

Task: read "ubuntu.iso" sequentially.

1. Look for the first block # of the file. (In its directory)
2. Read the file allocation table to determine the location of the next block. (In FAT)
3. The process stops until the block with the "**next block # = 0**".

Meaning of the numbers in `FAT12, FAT16, FAT32`

- The main difference among all the versions of FAT FS-es is the **cluster address size**. (12, 16, 32(28?) bytes of address length)

Trial 3.0: The Index-Node Allocation

- The Heart:
    - Structure of index node

- Usage of indirect blocks
- **How large** files can be supported? (block size = 2^x bytes, address length = 4 bytes)
    - Direct blocks: $n_0 \times 2^x$
    - Indirect blocks: $n_1 \times 2^{x-2} \times 2^x = n_1 \times 2^{2x-2}$
    - Double indirect blocks: $n_2 \times 2^{x-2} \times 2^{x-2} \times 2^x = n_2 \times 2^{3x-4}$
    - Triple indirect blocks: $n_3 \times 2^{x-2} \times 2^{x-2} \times 2^{x-2} \times 2^x = n_3 \times 2^{4x-6}$

Other Parts of the FS

Root Directory and Sub-directories

- Directory files are also files, so they also have their inodes

File system information and partitioning

- FS info is a set of important, FS-specific data
- **Solution**: The workaround is to save those information on the device.
    - FAT & NTFS: Boot Sector
    - Ext: Superblock
- A disk partition is a logical space...
    - A file system must be stored in a partition.
    - An operating system must be hosted in a partition.