CS-202 Week 4

Lecture 1

Memory virtualization mechanism: Paging

Better than segmentation

Key idea

Divide the address space into fixed-size chunks, called pages
Manage physical memory in fixed-size chunks called frames
An address space is a map: {pages -> frame}
Apply protection at page level

The allocated physical memory can be non-contiguous but appear contiguous in virtual memory

Page size

A page is the minimal unit of an address space
Should be small enough to minimize internal fragmentation (4KB-16KB)

Mapping

Map is stored in memory

Address representation

Two components of virtual addresses

Virtual page number: Number of pages in virtual address
Page offset: Size of the page

Address translation

MMU translates from virtual address to physical address:

- High order bits -> page number
- Map page number to frame number using a page table
- Low order bits -> page size

Page Tables

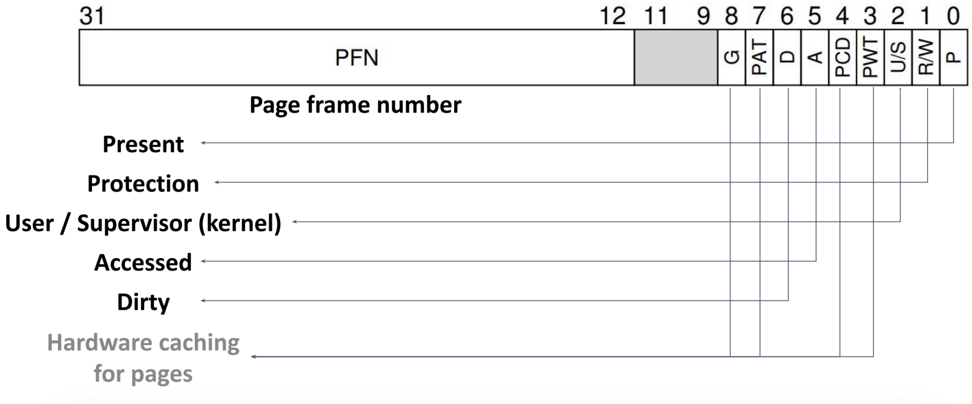
Stores virtual to physical address translations

- Stores address translations for each virtual pages in the address space
- Allows MMU to know where in physical memory each virtual page resides
  Every process has one page table
  The page table resides in memory and managed by the OS

The pointer to the page table is stored in a special register (page-table base register (PTBR / %cr3))
Value of PTBR is saved and restored in process control block upon context switch

Page table entries

- Index i corresponds to virtual page number (VPN)
- Contains page fram (PFN)
- Additional contains information bits:
    - Present bit: Indicates whether the address translation is valid
    - Protection bits: A page can have read/write/execute permission
        - Can be used to protect read-only pages like Text (Process memory segment)
    - U/S bit: accessible by user-level code (or only kernel code)
    - Dirty bit: Whether a page has been modified
    - Access/Reference bit



Organising PTE

Linear page tables

Take up to much space

Multi-level page tables

Only allocate a metadata for fraction of address space that is needed by process
(Re watch)

Translation look-aside buffer (TLB)

A cache of recent virtual address to physical address mappings
TLB is inside the CPU

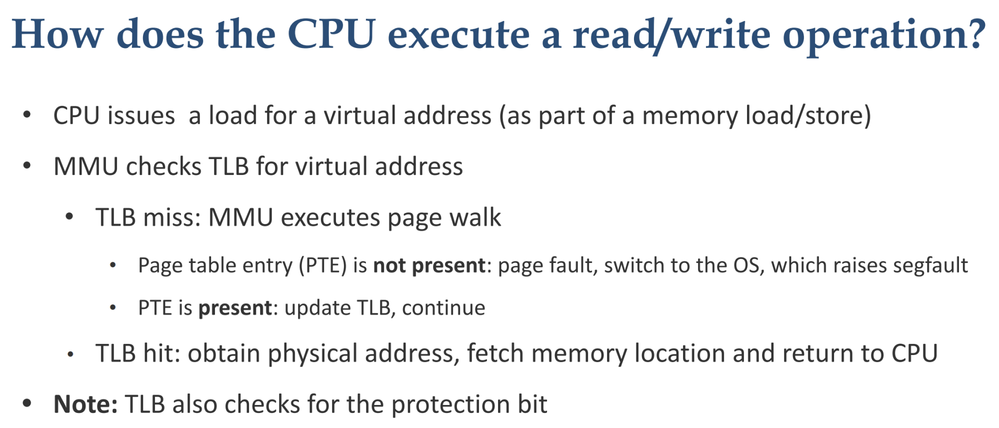
Translating virtual address to physical address:

1. MMU first looks up TLB
2. If TLB hit: physical address can be directly used
3. Only if TLB miss: MMU "walks" page table

TLB misses are **expensive** (multiple memory accesses)
Locality of reference helps to have a high hit rate

TLB entries may become invalid on context switch and change of page tables

TLB is NOT is memory, but rather a special circuit



How does the CPU execute a read/write operation?

- CPU issues a load for a virtual address (as part of a memory load/store)
- MMU checks TLB for virtual address
    - TLB miss: MMU executes page walk
        - Page table entry (PTE) is **not present**: page fault, switch to the OS, which raises segfault
        - PTE is **present**: update TLB, continue
    - TLB hit: obtain physical address, fetch memory location and return to CPU
- **Note:** TLB also checks for the protection bit

When assigning PTBR, TLB is wiped.

How the OS leverages paging

fork()

Better fork() implementation using **copy-on-write**

- Keep a reference count of the number of mapping of a frame
- fork(): duplicate PTE tree and mark PTE "read-only" in parent and child trees; increment reference count of all frames

Swapping

Used when main memory is not sufficient.
Stores unused pages in special file or designated region

Lecture 2

IOstack

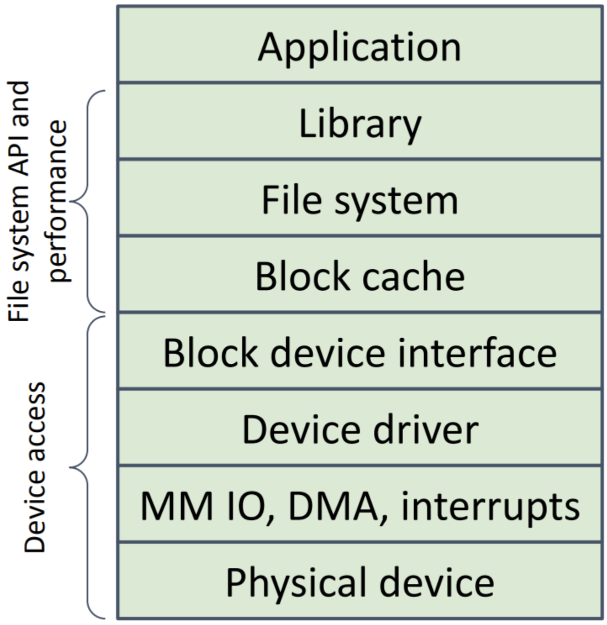
File system abstraction
File system API
File system implementation

Purpose of a file system

Given: Set of persistent blocks from storage device
Goal: Manage these blocks efficiently

General IO Abstraction stack

IO systems are accessed through a series of layered abstractions



The system call interface

Standardized, OS-independent, library

- more user friendly than system call
- fopen, fread, fwrite, fseek, etc...
- Operate on FILE * streams
- With additional buffering options (setvbuff)

Unix/Linux system calls

- open, read, write, lseek
- Operate on file descriptors

File System Abstraction

Two main components

The File abstraction

A file is named collection of related information that is recorded in secondary storage.
Or, a linear persistent array of bytes

Components

- **Data**: what a user or app puts in it
  - Array of bytes
- **Metadata**: Information added and managed by the OS
  - Size, owner, security information, modification time, etc...

3 perspectives

OS view: Inode and device number (persistent ID)

Low-level **unique** ID assigned to the file by the file system

- Inodes are unique for a file system but not globally
- Recycled after deletion
- Inode number is constant, doesn't change upon file modification
  - Its content can change
  An inode contains metadata of a file
- Permissions, length, access times
- Location of data blocks and indirection blocks
  Each file has exactly one associated inode

Inode table:

- Storage space is split into inode table and data storage
- Files are statically allocated
- Require inode number to access file content

User view: Path (human readable)

Each file has a human readable format: **file name**
Files are organized into hierarchies of directories: **pathname**

- A file name is unique locally to a directory; a full pathname is globally unique
  Modern file systems mostly use untyped files: array of bytes
- File is a sequence of bytes
- OS/file system does neither understand nor care about contents

Path to inode
A special file (directory) stores mapping between file names and inodes
Extend to hierarchy: Mark if a file maps to a regular file
Inode does NOT contain the file name

Directories:

- Each directory is a file (stored like regular files)
- Flag in the Inode separates directories from files
- Flag restricts API to processes (e.g., cannot write to a directory)
- Contains array of {filename, Inode}

Multiple filenames can link to same inode

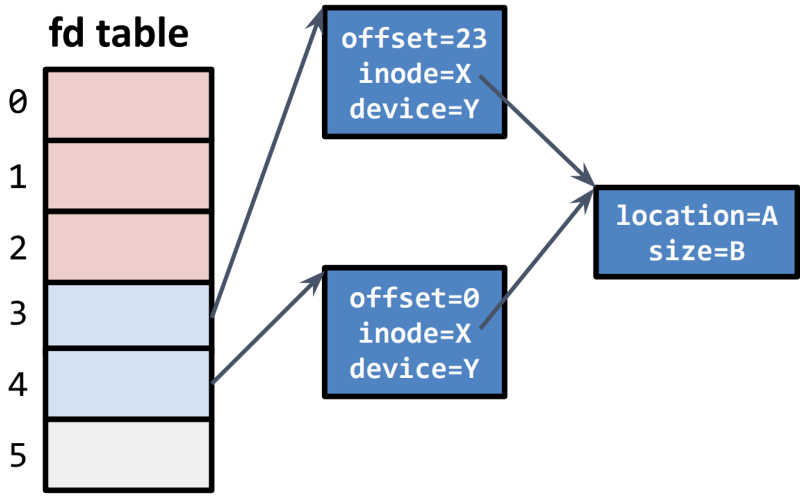
- inode has reference count
- called shortcut on windows or hardlink on unix/linux

Process view: file descriptor (process view)

The combination of file name and inode/device IDs are sufficient to implement persistent storage
Drawback: constant lookups from file names to inode are costly
Idea: do expensive tree traversal once, store final inode in a **per process table**

- Also keeps additional information such as file offset
- Per process table of **open files**, Each process has its own file descriptor (fd) table
- Use **linear numbers** (fd 0, 1, 2 ...), reuse when freed

0-2 are mapped to STDIN, STDOUT and STDERR



File system API

- Create a file: `int fd open(path, flags, permissions)`
  - returns a fd that grants the capability to perfom certain operations on the file
- Close the file: int close(fd)
- Reading/writing to a file:
  - Reading: ssize_t read(fd, void *buff, size_t count)
  - Writing: ssize_t write(int fd, void *buff, size_t count);
  - Returns the number of bytes read
- Manage file offset: off_t lseek(int fd, off_t offset, int whence);
  - Repositions file offset
  - Initially, set to 0, updated on read/write
    - to offset bytes from the beginning of the file whence = SEEK_SET
    - to offset bytes from current location whence = SEEK_CUR
    - to offset bytes after the end of the file whence = SEEK_END
- Unlink/delete a file: int unlink(const char *pathname);
  - Removes a file from a directory (when reference count is 0)
  - Decreases the reference count of the file

The path abstraction: Directory

A special file that stores mapping from file name to inode
Contains subdirectories:

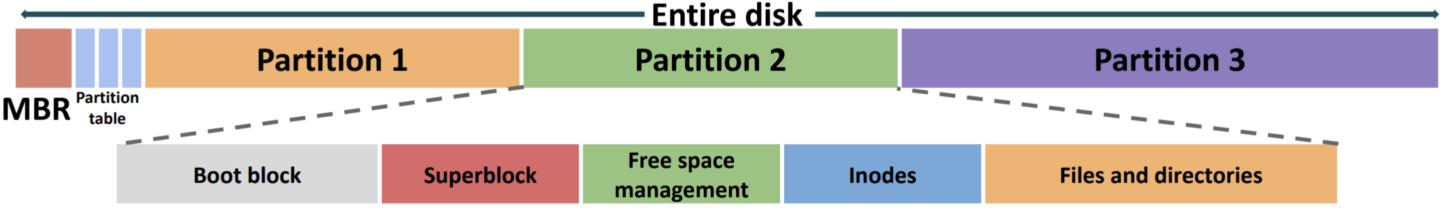
- List of directories, files
- '/' designates the root (usually inode:1)

File system implementation (REWATCH)

File system layout

File system is stored on disk

- Disk can be divided into one or more partitions



File allocation

Various ways to allocate data to files:

- **Contiguous allocation:** All bytes together, in order
- Linked

Multi-level indexing:
data is stores from most direct to most indirect. For Example: first 48 bytes of a 1TB file is still rapidly accessible