Design and Engineering of Computer Systems

Lecture 14:
File system and memory

Mythili Vutukuru
IIT Bombay
Storing program data

- User applications deal with data (variables, data structures, ...)
  - Global/static variables are in program executable
  - Memory allocated via malloc stored on heap
  - Function variables and arguments stored on stack
  - Can request one or more pages from OS via mmap to store data
- All of these mechanisms do not store data persistently
  - Data in main memory is lost when power is turned off
- How to store program data persistently?
  - Use files on secondary storage (hard disk and other such storage media)
- In this lecture: interaction of files with main memory
  - Next week: more details on filesystem, how file operations are done
File abstraction

- **File**: sequence of bytes, stored persistently on disk
- **Directory**: container for files and other sub-directories
- **Directory tree**: hierarchy of directories and sub-directories, containing files
  - Root file system: directory tree starting at root ("/")
  - Directory trees on disk can be mounted at various locations of root filesystem
- Containers have different root file system view on the same OS
- Steps to access a file
  - Open a file using system call, get a file descriptor
  - File descriptor is a handle to refer to file for read/write
  - Close file when done accessing it

```
fd = open("/home/foo/a.txt")
read(fd, ..)
write(fd, ..)
close(fd)
```
Reading and writing a file

- After opening a file, process can read/write to a file as a stream
  - **File descriptor** used as handle to refer to the open file stream
  - **Read system call** reads specified number of bytes into a user-defined buffer, returns number of bytes read
  - **Write system call** writes specified number of bytes from a user-defined buffer, returns number of bytes written

- Implemented by commands to disk controller via device driver in OS
  - Device driver gives command, data transfer via DMA, device raises interrupt when done

- Read and write system calls update the **offset** into a file
  - After reading N bytes, next read will return the next set of bytes
  - Can also update the offset from which to read/write using a **seek** system call

- Every open file descriptor will read/write file as independent stream, with independent offsets
  - Exception: parent and child processes share same offset after **fork**

```c
fd = open("/home/foo/a.txt")
char buf[64]
int n = read(fd, buf, 64)
buf[0] = ...
n = write(fd, buf, 64)
```
Memory mapping a file

- Alternate ways to read/write a file is via memory mapping
  - `mmap` system call takes the file descriptor, size of data to memory map, and other arguments, returns the starting virtual address of mmap region
  - File data is read into one or more physical memory frames, which are mapped at free addresses in the process virtual address space (new page table entries)
  - Can access memory mapped file data like any other memory region
  - With demand paging, physical frames can be assigned on-demand only when mmap region accessed

- File can be memory mapped in **private** or **shared** mode
  - Shared mode: changes to file are written to disk immediately, seen by others
  - Private mode: changes to file are written to disk when memory unmapped
Disk buffer cache

• What happens after memory mapped file is unmapped?
  • Why erase file data, if some other process requires the data in near future?

• File data that is read from hard disk is retained in memory for some time in the disk buffer cache = memory pages that cache recently read disk data
  • Both mmap files and files read via read/write syscalls are cached

• Any changes to disk data (via write or mmap) is made in the cached copy of disk buffer cache first, then written to disk later
  • Write-through cache: changes written to disk immediately (synchronous writes)
  • Write-back cache: changes written to disk after some delay (asynchronous writes)
  • Write-back cache has better performance, but can lose data in case of power failure

• Benefits of disk buffer cache
  • Improved performance due to fewer disk accesses
  • Merge changes when multiple processes modify same file data

• Most OS allocate unused physical memory to disk buffer cache
  • Some applications doing their own optimizations can bypass cache
Understand various layers of caches

- **CPU caches**: stores recent memory instructions/data accessed by CPU
- **TLB**: stores recent virtual to physical address mappings done by MMU
- **Disk buffer cache**: stores recently accessed filesystem data in memory
- **CPU cache and disk buffer cache store actual data, TLB stores mappings**
  - TLB hit avoids only page table walk, not actual memory access
- **All caches use locality of reference** to avoid extra work in future
  - Temporal locality of reference: data accessed in recent past will be likely used again
  - Spatial locality of reference: data around current access will be likely used again
- **All caches use some variant of LRU (least recently used) policy for evicting old entries when cache is full**
mmap vs. read/write syscalls

• mmap can be used for file-backed as well as anonymous pages
  • Physical frame mapped into address space can be empty frame or with file data
• Memory mapping a file is an easy way to read file data
  • Executable code, shared library code are memory mapped into virtual address space
• Memory mapping a file avoids extra data copies
  • Read/write system calls read data first into memory (disk buffer cache), then copy from disk buffer cache into user provided buffer
  • Memory mapping a file copies file data into free physical frames, which are directly accessed by user using virtual addresses
• Memory mapping allows reading disk data in large page-sized chunks
  • Useful when reading/writing large amounts of data from file
  • Not very efficient when reading files in small chunks
Data storage options in real applications

- **Local storage**: store data on local physical machine
  - CPU caches (SRAM) – fast, expensive, small memory close to CPU (~1-10 ns)
  - Main memory (DRAM) – random access memory for volatile storage (~100 ns)
  - Hard disk drive (HDD) – traditional magnetic disk, stores file data in blocks (~ms)
  - Solid State Drive (SSD) – faster option than HDD for files, common today
  - Non-volatile memory (NVM) – persistent memory, faster than hard disk

- **Remote storage**: storage accessible over the network in the cloud
  - Network Attached Storage (NAS) – store data in reliable file storage appliances
  - Databases – relational databases to store relational data durably
  - In-memory key-value stores – stores data in key-value format, distributed over several nodes
  - Distributed file systems – file storage built over a distributed system of nodes
  - Remote memory – DRAM-like memory accessible over the network

- Real computer systems use some combination of local and remote storage to achieve the functional and non-functional (performance, reliability) goals of applications
Summary

• In this lecture:
  • Storing program data in memory, files
  • Accessing file data using read/write vs. mmap syscalls
  • Disk buffer cache

• Look up the amount of main memory in your system, and find out how much of it is used as the disk buffer cache
  • The “free” command in Linux will show you the total memory available in your system, how much of it is used for the cache, and how much is free

• Programming exercise: practice accessing file data using read/write system calls as well as using mmap