CSCE 313-200
Introduction to Computer Systems
Spring 2024

File System
Dmitri Loguinov
Texas A&M University

March 20, 2024
Chapter 11: Roadmap

11.1 I/O devices
11.2 I/O function
11.3 OS design issues
11.4 I/O buffering
11.5 Disk scheduling
11.6 RAID
11.7 Disk cache
11.8-11.10 Unix, Linux, Windows

Part V

Chapter 11: I/O
Chapter 12: Files
I/O Devices

- I/O usually refers to physical devices
  - Such as disk, network card, printer, keyboard
- Almost all components in the system do I/O
  - Except RAM & CPU, and possibly certain chipsets built into the motherboard
- Transfer of data between devices and RAM thru DMA

Example: AMD Opteron

- Cores
- L3 cache
  - memory controller
  - HyperTransport
  - Northbridge
  - Southbridge
  - Fast I/O
    - PCI-E
    - AGP
    - RAID
  - Slow I/O
    - USB
    - SATA
    - PCI-X
    - PCI
    - VGA
    - COM/LPT
    - Floppy
  - Northbridge
  - RAM
I/O Devices

• How fast is I/O compared to RAM speed?
  − Usually slow, but it depends…

• How to measure speed?
  − Kbps, Mbps, Gbps refer to bits/sec
  − KB/s, MB/s, GB/s refer to bytes/sec

• Use a notation with $K = 1000$ bits/bytes

<table>
<thead>
<tr>
<th>I/O Devices</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard/mouse</td>
<td>~100 bytes/s</td>
</tr>
<tr>
<td>Modem</td>
<td>53 Kbps</td>
</tr>
<tr>
<td>Floppy</td>
<td>70 KB/s</td>
</tr>
<tr>
<td>CD-ROM 1x</td>
<td>150 KB/s</td>
</tr>
<tr>
<td>Ethernet</td>
<td>10 Mbps</td>
</tr>
<tr>
<td>USB 1.0</td>
<td>1.5 MB/s</td>
</tr>
<tr>
<td>DVD-ROM 32x</td>
<td>4.7 MB/s</td>
</tr>
<tr>
<td>Fast Ethernet</td>
<td>100 Mbps</td>
</tr>
<tr>
<td>USB 2.0</td>
<td>60 MB/s</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>1 Gbps</td>
</tr>
<tr>
<td>Hitachi 2TB drive</td>
<td>150 MB/s</td>
</tr>
<tr>
<td>SSD hard drive</td>
<td>500 MB/s</td>
</tr>
<tr>
<td>USB 3.0</td>
<td>600 MB/s</td>
</tr>
<tr>
<td>10G Ethernet</td>
<td>10 Gbps</td>
</tr>
<tr>
<td>DDR2-667 RAM</td>
<td>5.3 GB/s</td>
</tr>
<tr>
<td>100G Ethernet</td>
<td>100 Gbps</td>
</tr>
<tr>
<td>DDR4-3200 RAM</td>
<td>90 GB/s</td>
</tr>
<tr>
<td>L2 cache (8 core)</td>
<td>500 GB/s</td>
</tr>
<tr>
<td>L1 cache (8 core)</td>
<td>1.5 TB/s</td>
</tr>
<tr>
<td>DDR2-667 RAM</td>
<td>5.3 GB/s</td>
</tr>
<tr>
<td>100G Ethernet</td>
<td>100 Gbps</td>
</tr>
<tr>
<td>DDR4-3200 RAM</td>
<td>90 GB/s</td>
</tr>
<tr>
<td>L2 cache (8 core)</td>
<td>500 GB/s</td>
</tr>
<tr>
<td>L1 cache (8 core)</td>
<td>1.5 TB/s</td>
</tr>
</tbody>
</table>
I/O Devices

• OS also allows certain IPC to be modeled as communication with an abstract I/O device
  – Example: inter-process pipes, mailslots, network sockets
  – This explains why ReadFile is so universal

• Our main focus here is on file I/O, but similar principles apply to other types of devices
  – Just reading files is simple; however, achieving decent speed and parallelizing computation is more challenging

• Before solving this problem, we start with a general background on files and APIs
  – Homework #3 requires multi-CPU searching of Wikipedia for user-specified substrings
Background on Files

• Just like RAM, a file is a sequence of bytes
• Supports 3 main operations: read, write, and seek
• File pointer specifies the current position within the file
  – Read/write operations proceed from that location forward
• Example: test.txt written in notepad:
  – Byte contents give by hex viewer (e.g., HxD)

<table>
<thead>
<tr>
<th>54 68 69 73 20 69 73 20 61 20 74 65 78 74 20 66 69 6C 65 2E 0D 0A 53 65 63 6F 6E 64 20 6C 69 6E 65 2E</th>
<th>This is a text file. Second line.</th>
</tr>
</thead>
<tbody>
<tr>
<td>54 68 69 73 20 69 73 20 61 20 74 65 78 74 20 66 69 6C 65 2E 0D 0A 53 65 63 6F 6E 64 20 6C 69 6E 65 2E</td>
<td>This is a text file...Second line.</td>
</tr>
</tbody>
</table>

• What is the ASCII table?
  – Why is there 0xD and 0xA in the file?
Two modes of file I/O: text and binary
- Must be requested when you open the file

Binary means disk contents are an exact copy of the RAM buffer that is written and vice versa

Text means there is some library (wrapper) between the application and OS that applies certain “magic” translation before your program sees the data
- For fopen/fprintf, this involves \r\n → \n, terminating the read at Ctrl-Z markers (ASCII code 26), and certain multi-byte to wide char mapping based on the locale

Note: text files can be always read in binary mode, while the opposite is not true
Background on Files

- **Example**: binary mode reads the file as is:
  ```plaintext
  54 68 69 73 20 69 73 20 61 20 74 65 78 74 20 66 
  69 6C 65 2E 0D 0A 53 65 63 6F 6E 64 20 6C 69 6E 
  65 2E
  ```
  - while **text mode** removes \r
    ```plaintext
    54 68 69 73 20 69 73 20 61 20 74 65 78 74 20 66 
    69 6C 65 2E 0D 0A 53 65 63 6F 6E 64 20 6C 69 6E 
    65 2E
    ```

- If the file is tweaked before it reaches your program, lots of confusing things may happen
  - E.g., file size 100,050 bytes, but your buffer gets only 99,800

- Since text-mode processing does usually unwanted things to the file and is much slower than binary mode, it is not recommended (see later for benchmarks)
Number representation can be ASCII or native
- ASCII is human-readable form (e.g., printf ("%d", x))
- Native is identical to how numbers are stored in RAM

Example:
```
int x = 0x11223344;
```

ASCII output depends on how the numbers are written (e.g., decimal, hex) and the separator between them
- Conversion to/from ASCII is usually slow
- Format inefficient in terms of storage

APIs that read raw buffers are usually native
- Those that attempt to read individual variables are ASCII
• Suppose we read an integer natively from the beginning of this file

```c
int x;
ReadFile (&x, sizeof(int));
```

- What is the value of x?
- Equivalent versions

• How to write contents of some class natively to disk?
  - If it has no pointers, then it’s trivial

```c
class MyClass {
    double a;
    uint64 b;
};
MyClass mc;
mc.a = 3.1415;
mc.b = 0x55;
WriteFile (... , &mc, sizeof(MyClass), ...);
```
How to store pointers, e.g., a linked list or binary tree?

- Data structure must first be converted to an array
  - Hierarchical structure must be flattened

```cpp
class LinkedListElem {
    int val;
    LinkedListElem *next;
};

class TreeElem {
    int val;
    TreeElem *left, *right;
};

int valArray = new int [LinkedList.size()];
// traverse the list, copy into valArray
WriteFile (... , valArray, sizeof(int) * LinkedList.size(), ...);

class TreeElem2 {
    int val;
    int left, right; // offsets
    TreeElem2 *arr = new TreeElem2 [tree.size()];
    TreeElem2 *arr = new TreeElem2 [tree.size()];
    val = 55
    val = 22
    val = 77
    val = 14
    val = 65
    val = 90
left = 1
left = 3
left = 4
left = 0
left = 0
left = 0
right = 2
right = 0
right = 5
right = 0
right = 0
right = 0
```
In fact, trees stored as arrays in RAM are often much faster than pointer-based trees.
- Main drawback: difficult to deal with fragmentation.

Further compaction: 2 bits to store # of children.
- Suppose 00 = none, 01 = left, 10 = right, 11 = both.

Conversion from random-access (RAM) structures to sequential arrays is called serialization.
- Similar to serial transmission over COM ports or networks.