File System

Dmitri Loguinov
Texas A&M University

March 26, 2019
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
I/O Devices

- I/O usually refers to physical devices
  - Such as disk, network card, printer, keyboard
- Almost every device in the system is I/O
  - Except RAM, CPU, L3 cache, and certain chipsets built into the motherboard
- Transfer of data between devices and RAM thru DMA

Example: AMD Opteron

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

- Northbridge
- Southbridge
- Memory controller
- RAM
- CPU
- cache
- HyperTransport
### 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

- **Kbit is always 1000 bits**

- **KByte is traditionally 1024 bytes**

| I/O Devices | ~100 bytes/s | 53 Kbps | 70 KB/s | 150 KB/s | 10 Mbps | 1.5 MB/s | 100 Mbps | 44 MB/s | 60 MB/s | 1 Gbps | 150 MB/s | 500 MB/s | 600 MB/s | 10 Gbps | 5.3 GB/s | 100 Gbps | 90 GB/s | 500 GB/s | 1.5 TB/s |
|--------------|--------------|---------|---------|----------|---------|---------|----------|---------|---------|--------|----------|----------|----------|---------|----------|---------|----------|---------|----------|---------|
| Keyboard/mouse Modem |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Floppy |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| CD-ROM 1x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Ethernet |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| USB 1.0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Fast Ethernet |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DVD-ROM 32x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| USB 2.0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Gigabit Ethernet |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Hitachi 2TB drive |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SSD hard drive |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| USB 3.0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 10G Ethernet |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DDR2-667 RAM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 100G Ethernet |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| DDR4-3200 RAM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| L2 cache (8 core) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| L1 cache (8 core) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
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 as viewed with hex editor (e.g., HxD)

| 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 | This is a text file. Second line. |

• What is the ASCII table?
  – Why is there 0xD and 0xA in the file?
Background on Files

• 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 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:

```
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

```
54 68 69 73 20 69 73 20 61 20 74 65 78 74 20 66
69 6C 65 2E 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 similar to what you see on the screen
- Native is identical to how numbers are stored in RAM

**Example:**

```
int x = 0x11223344;
```

- Native version:
  ```
  44 33 22 11
  ```
- Decimal ASCII version of x, i.e., string “287454020”

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
  - 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;
SomeNativeWriteFunc (&mc,
    sizeof(MyClass) );
```

```
char buf[] = “This”;  // 6F 12 83 C0 CA 21 09 40 55 00 00 00 00 00 00 00
int x = *(int*) buf;
```

```
int x = 0x73696854;
```

```
Notepad shows: o↑ fÀÊ!@U
```
Background on Files

- How to store pointers, e.g., a linked list or binary tree?

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

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

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

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

```cpp
class TreeElem2 {
    int val;
    int left, right; // offsets
};
TreeElem2 *arr = new TreeElem2 [tree.size()];
```
• 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
Background on Files

- Asking the kernel for chunk of data
  - How large should the chunk be?
- Clearly not too small, otherwise many kernel-mode transitions, which are costly
- Some wrapper libraries (FILE and STL streams) have yet another buffer to avoid kernel-mode switching
  - Also needed if they perform text-mode pre-processing
- **OS buffering** can be disabled
  - Disk driver directly DMAs data into your program’s buffer
  - Caveat: buffer size must be a multiple of sector size (512 bytes)
• CreateFile is the most flexible and high-performance method of doing I/O
  - Treats the memory as a sequence of bytes
  - Operates in binary mode and gives you the native representation of RAM data structures
• Read MSDN about access (read, write, both), sharing, and disposition (e.g., open existing, create new)
• The flag field sets the attributes (e.g., hidden, encrypted, read-only, archived, system)
  - Also can be used to disable OS buffering (FILE_FLAG_NO_BUFFERING) or enable overlapped operation (FILE_FLAG_OVERLAPPED)
Some functions take two DWORDs instead of one uint64

- How to convert?

```c
DWORD low = GetFileSize(hFile, &high);
uint64 size = ((uint64)high << 32) + low;

DWORD high, low = GetFileSize(hFile, &high);
uint64 size = ((uint64)high << 32) + low;
```

Overlapped I/O allows multiple outstanding requests

```c
OVERLAPPED ol;
memset (&ol, 0, sizeof (OVERLAPPED));
ol.hEvent = CreateEvent (NULL, false, false, NULL);
ReadFile (hFile, buf, len, NULL, &ol);
// if error == ERROR_IO_PENDING, continue
WaitForSingleObject (ol.hEvent, INFINITE);
GetOverlappedResult (hFile, &ol, &bytesRead, false);
```
The FILE stream is the classical C-style library
- Portable to Unix and most other OSes

```c
FILE *fopen (const char *filename, const char *mode);
size_t fread (void *buffer, size_t size, size_t count, FILE *stream);

char buf [BUF_SIZE];
// open for reading in binary mode
FILE *f = fopen ("test.txt", "rb");
if (f == NULL) {
    printf ("Error %d opening file\n", errno);
    exit (-1);
}
// read up to one full buffer
// native representation
int bytesRead = fread (buf, 1, BUF_SIZE, f);
fclose (f);

FILE *f = fopen ("test.txt", "rb");
// seek to the end
_fseeki64 (f, 0, SEEK_END);
// get current position
uint64 fileSize = _ftelli64(f);
// return to beginning
_fseeki64 (f, 0, SEEK_SET);
printf ("file size %I64u\n", fileSize);

int a = 5;
double b = 10;
// open for writing in binary mode
FILE *f = fopen ("test.txt", "wb");
// ASCII representation
fprintf (f, "a = %d, b = %f\n", a, b);
fclose (f);

int a;
double b;
// ASCII decoding of numbers
int ret = fscanf (f, "%d %f", &a, &b);
if (ret == 0 || ret == EOF)
    printf ("Hit error or EOF\n");
else
    printf ("Obtained %d, %f\n", a, b);
// %s gets one word and NULL terminates it
// note: potential buffer overflow
fscanf (f, "%s", buf);
// recommended to specify buf length
fscanf (f, "%32s", buf);
```
APIs

- If an entire line is needed, a faster alternative to fscanf is fgets()
- STL streams are similar

```c
char buf [BUF_SIZE];
FILE *f = fopen ("test.txt", "rb");
while (!feof (f)) {
    // read one line at a time
    if (fgets (buf, BUF_SIZE, f) == NULL)
        break; // EOF or error
    printf ("Line '%s' has %d bytes
", buf, strlen(buf));
}
fclose (f);
```

```c
ifstream ifs;
// binary mode open
ifs.open (filename, ios::binary);
while (ifs) { // not EOF?
    // native read
    ifs.read (buf, BUF_SIZE);
    printf ("Read %d bytes\n", ifs.gcount());
    printf ("Position in file %d\n", ifs.tellg());
}
// now try ASCII read
int x;
ifs >> x; // attempts to read an int
string s;
ifs >> s; // reads the next word
// read one line up to some delimiter
getline (ifs, s, '\n');
```

Q: using Windows APIs, how to print contents of a text file?

```c
// assume file is small and fits in RAM
// allocate the buffer
char *buf = new char [fileSize + 1];
ReadFile (... , buf , fileSize , &bytes , ... );
// TODO: error checks
buf[bytes] = NULL;
printf ("%s\n", buf);
```
## Performance

- Dual RAID controllers, each with 12 disks in RAID-5
  - Speed given in MB/s, CPU utilization = fraction of 16 cores

<table>
<thead>
<tr>
<th></th>
<th>Text mode</th>
<th></th>
<th>Binary mode</th>
<th></th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Debug</td>
<td>Release</td>
<td>Debug</td>
<td>Release</td>
<td></td>
</tr>
<tr>
<td>ifs &gt;&gt; s</td>
<td>1.8</td>
<td>12</td>
<td>1.8</td>
<td>13</td>
<td>10%</td>
</tr>
<tr>
<td>fscanf (f, “%s”, buf)</td>
<td>6</td>
<td>19</td>
<td>7.5</td>
<td>19</td>
<td>9%</td>
</tr>
<tr>
<td>fgets (buf, BUF_SIZE, f)</td>
<td>26</td>
<td>50</td>
<td>39</td>
<td>79</td>
<td>7%</td>
</tr>
<tr>
<td>ifs.read w/32MB buffer</td>
<td>90</td>
<td></td>
<td>360</td>
<td></td>
<td>10%</td>
</tr>
<tr>
<td>fread w/32MB buffer</td>
<td>90</td>
<td>144</td>
<td>503</td>
<td>650</td>
<td>9%</td>
</tr>
<tr>
<td>ReadFile w/32MB buffer</td>
<td></td>
<td></td>
<td>982</td>
<td></td>
<td>11%</td>
</tr>
<tr>
<td>ReadFile + no OS buffering</td>
<td></td>
<td></td>
<td>1923</td>
<td></td>
<td>10%</td>
</tr>
<tr>
<td>ReadFile + no buf + overlapped</td>
<td></td>
<td></td>
<td>2500</td>
<td></td>
<td>11%</td>
</tr>
</tbody>
</table>

- Tom’s Hardware Guide
  - 3.4 GB/s with depth-32 overlapped I/O and 16 SSD drives