CSCE 313-200
Introduction to Computer Systems
Spring 2025

File System

Dmitri Loguinov

Texas A&M University

March 18, 2025

Chapter 11: Roadmap

11.1 I/O devices

11.2 I/O function

11.3 OS design issues

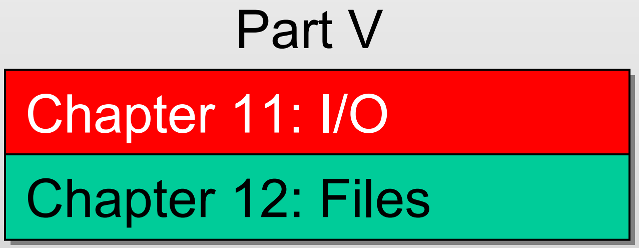
11.4 I/O buffering

11.5 Disk scheduling

11.6 RAID

11.7 Disk cache

11.8-11.10 Unix, Linux, Windows



Part V

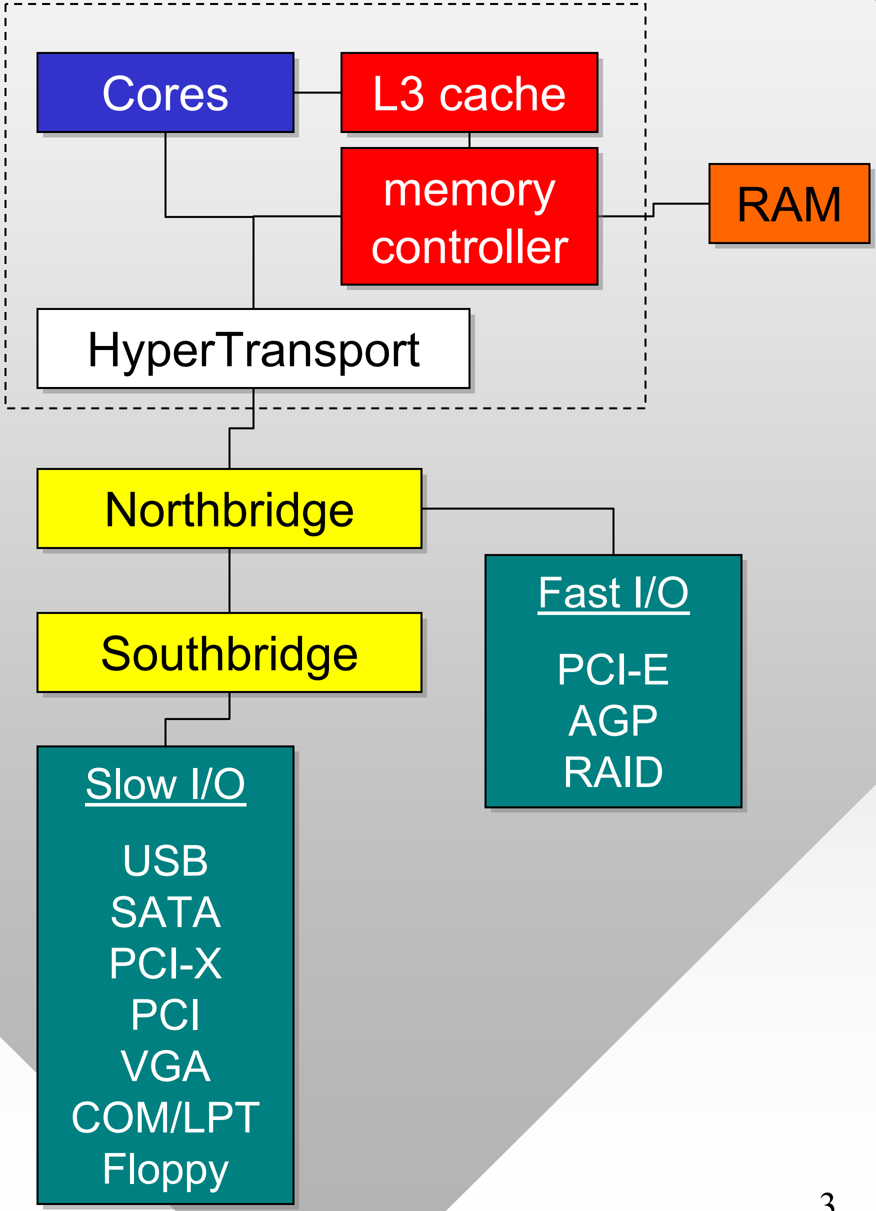
Chapter 11: I/O

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
- Transfer of data between devices and RAM thru DMA
  - Direct Memory Access allows device to talk to RAM without CPU

Example: AMD Opteron



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

Keyboard/mouse	~100 bytes/s
Modem	53 Kbps
Floppy	70 KB/s
CD-ROM 1x	150 KB/s
Ethernet	10 Mbps
USB 1.0	1.5 MB/s
Fast Ethernet	100 Mbps
USB 2.0	60 MB/s
Gigabit Ethernet	1 Gbps
Hitachi 2TB drive	150 MB/s
SSD drive	550 MB/s
USB 3.0	600 MB/s
10G Ethernet	10 Gbps
DDR2-667 RAM	5.3 GB/s
100G Ethernet	100 Gbps
m.2 PCIe 5.0 drive	14 GB/s
DDR4-3200 RAM	90 GB/s
L2 cache (8 core)	500 GB/s
L1 cache (8 core)	1.5 TB/s

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:

  This is a text file.
  Second line.

  - Byte contents give by hex viewer (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 f
ile...Second lin
e.
```

- 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 is some library (wrapper) between the application and OS that applies certain 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

This is a text file.
Second line.

- 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

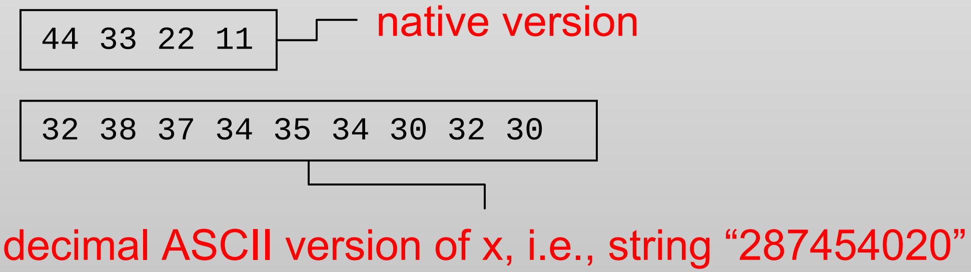
```
54 68 69 73 20 69 73 20 61 20 74 65 78 74 20 66
69 6C 65 2E 0A 53 65 63 6F 6E 64 20 6C 69 6E 65
2E
```

- 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)

Background on Files

- 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;
```



```
44 33 22 11
```

native version

```
32 38 37 34 35 34 30 32 30
```

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 native
  - Those that attempt to read individual variables are ASCII

Background on Files

This is a text file.
Second line.

```
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
```

- Suppose we read an integer natively from the beginning of this file

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

  - What is the value of x?
  - Equivalent versions →

```
char buf[] = "This";
int x = *(int*) buf;
```

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

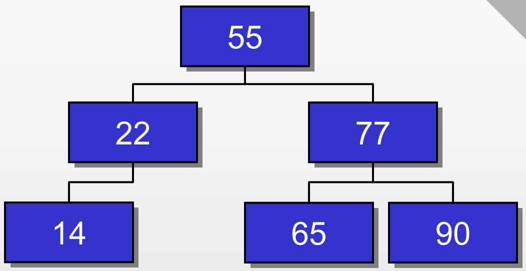
- How to write contents of some class natively to disk?
  - If it has no pointers, then it's trivial

```
class MyClass {
   double a;
   uint64 b;
};

MyClass mc;
mc.a = 3.1415;
mc.b = 0x55;
WriteFile (…, &mc, sizeof(MyClass), …);
```

$$\underbrace{\texttt{6F 12 83 C0 CA 21 09 40}}_{\text{mc.a}}\ \underbrace{\texttt{55 00 00 00 00 00 00 00}}_{\text{mc.b}}$$

Notepad shows: o↕ ƒÀÊ!@U

Background on Files



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

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

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

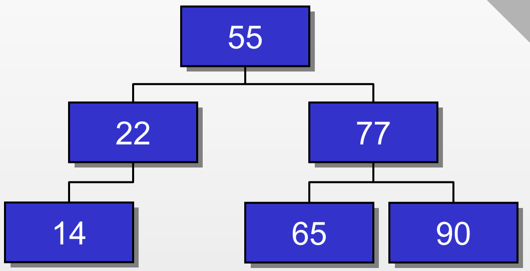
- Data structure must first be converted to an array
  - Hierarchical structure must be flattened

```
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()];
```

val = 55 left = 1 right = 2	val = 22 left = 3 right = 0	val = 77 left = 4 right = 5	val = 14 left = 0 right = 0	val = 65 left = 0 right = 0	val = 90 left = 0 right = 0
0	1	2	3	4	5

Background on Files



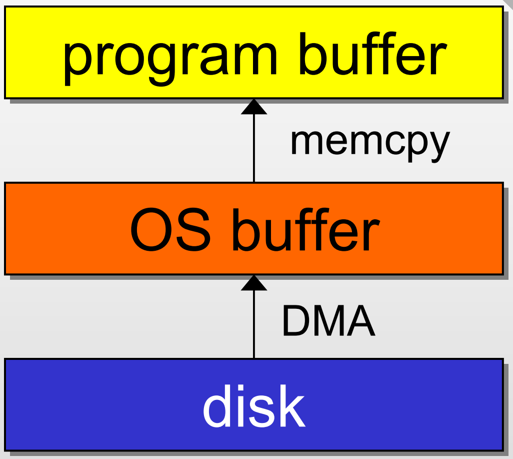
- In fact, trees stored as arrays in RAM are often much faster than pointer-based trees
  - Main drawback: difficult to deal with fragmentation
- Further compaction: 2 bits to store # of children
  - Suppose 00 = none, 01 = left, 10 = right, 11 = both

`val = 55`
`bits = 3`
	`val = 22`
	`bits = 1`
		`val = 77`
		`bits = 3`
			`val = 14`
			`bits = 0`
				`val = 65`
				`bits = 0`
					`val = 90`
					`bits = 0`

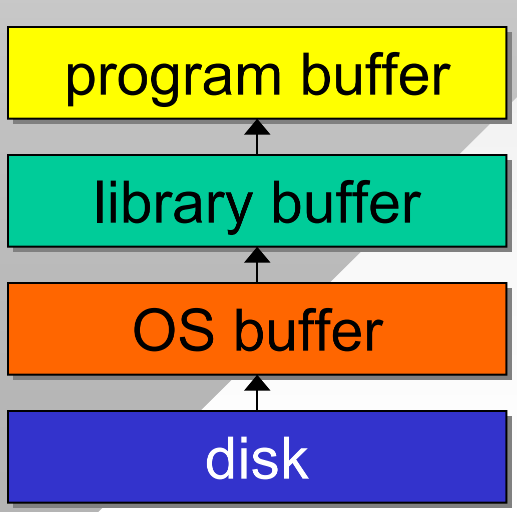
0	1	2	3	4	5

- 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)



APIs

```
HANDLE WINAPI CreateFile(
  __in      LPCTSTR lpFileName,
  __in      DWORD dwDesiredAccess,
  __in      DWORD dwShareMode,
  NULL,   // security
  __in      DWORD dwCreationDisposition,
  __in      DWORD dwFlagsAndAttributes,
  NULL    // template
);
```

- 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)

APIs

- Some functions take two DWORDs instead of one uint64
  - How to convert?

```
char buf [BUF_SIZE];
DWORD bytes;

// read a whole chunk
if (ReadFile (hFile, buf, BUF_SIZE,
          &bytes, NULL) == 0) {
    if (GetLastError () != ERROR_HANDLE_EOF) {
          // handle error
          exit (-1);
    }
    reachedEof = true;
}
else if (bytes < BUF_SIZE)
    reachedEof = true;

printf ("Obtained %d bytes, EOF = %d\n",
                    bytes, reachedEof);
```

```
DWORD low = GetFileSize(HANDLE hFile,
                   LPDWORD high);
```

```
// combining DWORDs into uint64
DWORD high, low = GetFileSize (h, &high);
uint64 size = ((uint64)high << 32) + low;

// splitting a uint64 into DWORDs
high = size >> 32;
low = size & ((DWORD) -1);
```

```
DWORD WINAPI SetFilePointer(
  __in          HANDLE hFile,
  __in          LONG lDistanceToMove,
  __inout_opt   PLONG lpDistanceToMoveHigh,
  __in          DWORD dwMoveMethod );
```

- Overlapped I/O allows multiple outstanding requests

```
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);
```

Note: each pending request must have its own struct ol

APIs

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

- The FILE stream is the classical C-style library
  - Portable to Unix and most other OSes

```
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 %llu\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

```
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\n”,
          buf, strlen(buf));
}
fclose (f);
```

```
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?

```
// 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

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

- Modern PCIe 5.0 m.2 drives in RAID
  - Up to 60 GB/s