CSCE 313-200
Introduction to Computer Systems
Spring 2024

File System II

Dmitri Loguinov
Texas A&M University

March 22, 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

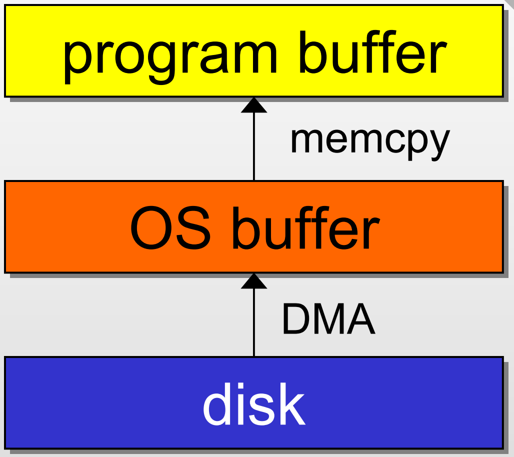
11.7 Disk cache

11.8-11.10 Unix, Linux, Windows

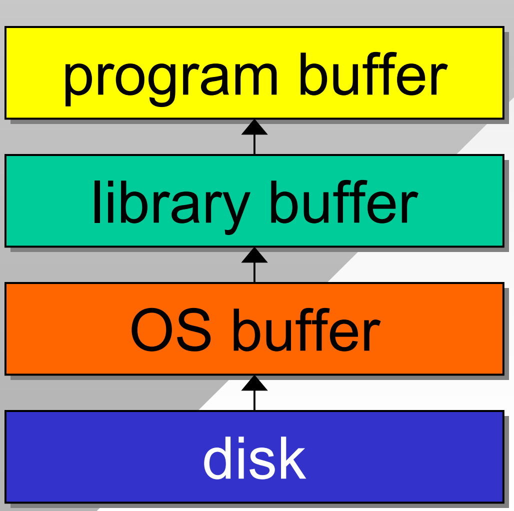
Part V

Chapter 11: I/O
Chapter 12: Files

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);
```

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

```
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()

```
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);
```

- STL streams are similar

```
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 PCI-e 4.0 m.2 drives
  - Up to 7 GB/s; multiple in RAID configuration up to 30 GB/s