7. File I/O

AlDaBi Praktikum

David Weese
© 2010/11

René Rahn / Jongkyu Kim
WS 2015/16

Inhalt

- Externspeicher
- I/O-Interfaces in C++

- Bemerkungen zum Code-Review und zu den P-Aufgaben

EXTERNSPEICHER

David Weese

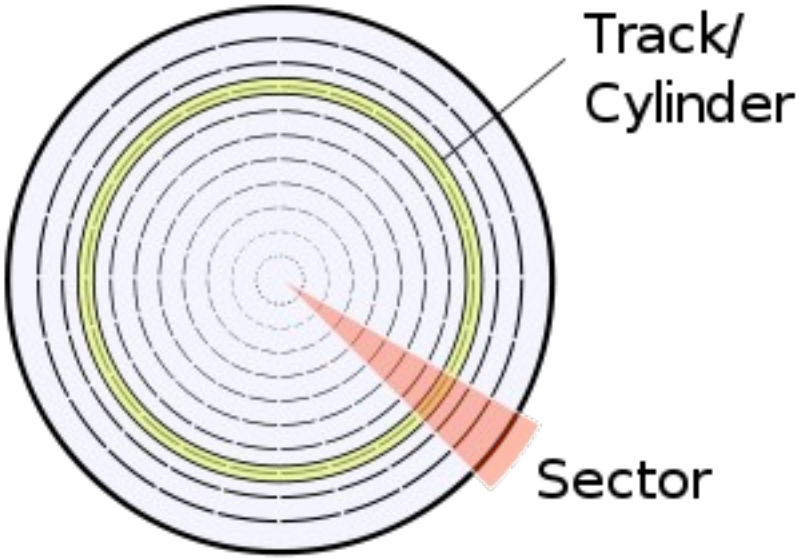
René Rahn / Jongkyu Kim
WS 2015/16

Aufbau einer Festplatte (HDD)

- Typische Bestandteile einer Festplatte:
  - Schnell rotierende **Spindel** (spindle), typ. 3.000-15.000 U/min
  - An der Spindel befestigte flache ferromagnetische **Scheiben** (platter)
  - Beweglicher **Arm** an dessen Ende die Köpfe montiert sind
  - Schreib-Lese-**Köpfe** (head) für jede Seite einer Scheibe

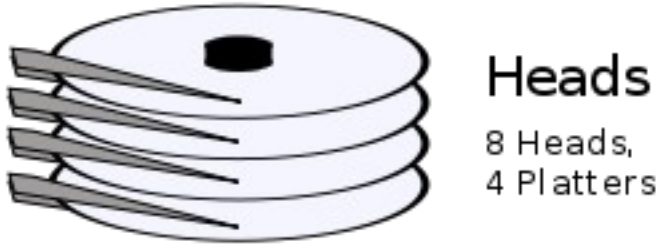
Datenanordnung

- Gespeicherte Daten liegen auf den Scheibenoberflächen und werden von den Köpfen gelesen/geschrieben (head)
- Scheibenoberfläche ist in konzentrische Spuren (track/cylinder) unterteilt
- Spur besteht aus Sektoren (sector)



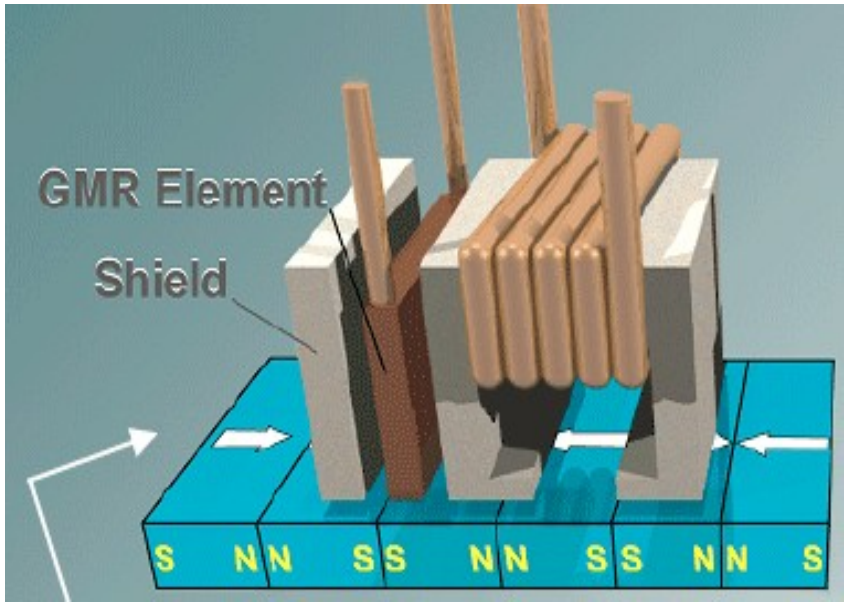
- Typische Sektorgröße:
  - 512 Byte

- Adresse eines Sektors:
  - CHS-Tripel
    (**C**ylinder, **H**ead, **S**ector)

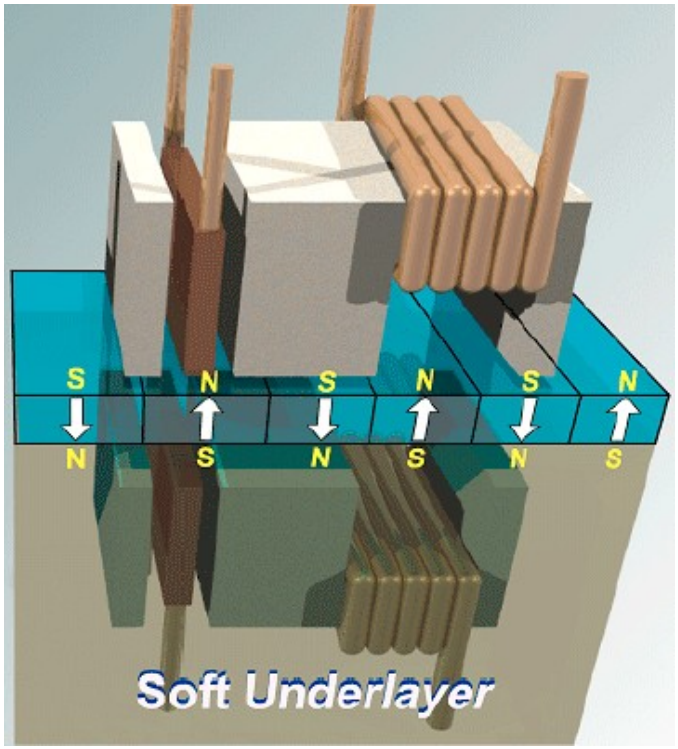


Schreib-Lese-Kopf

- Ausrichtung der magnetischen Domänen kodiert Daten (RLL, MFM)
- Magnetische Domänen werden entweder entlang der Kopfbewegung (longitudinal) oder senkrecht dazu (perpendicular) ausgerichtet



Longitudinal (vor 2006)

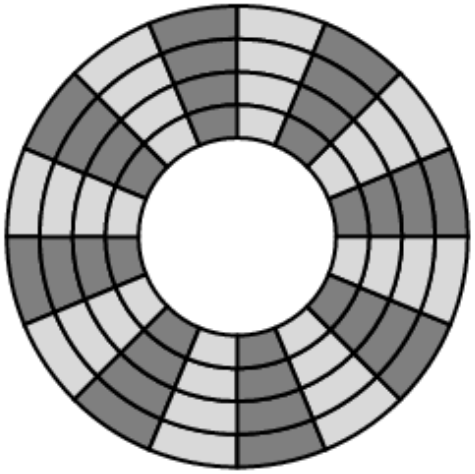


Perpendicular (aktuell)

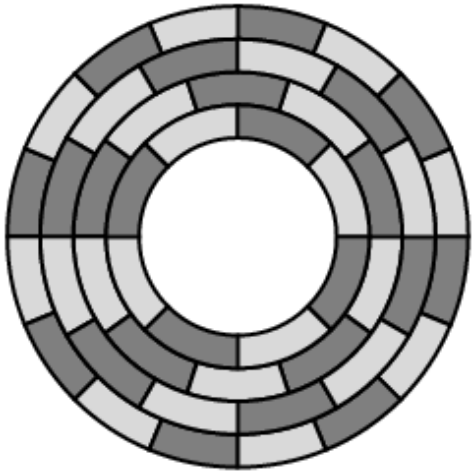
Datendichte

- Festplatten rotieren mit konstanter Geschwindigkeit
  - nach außen nimmt die Oberflächengeschwindigkeit der Köpfe zu
  - Haben alle Sektoren das gleiche Bogenmaß (CAV) nimmt die Datendichte nach außen ab
  - Aktuelle Festplatten wählen möglichst gleichgroße Sektoren (MZR)

Constant **A**ngular **V**elocity



Multiple **Z**one **R**ecording



Performanz

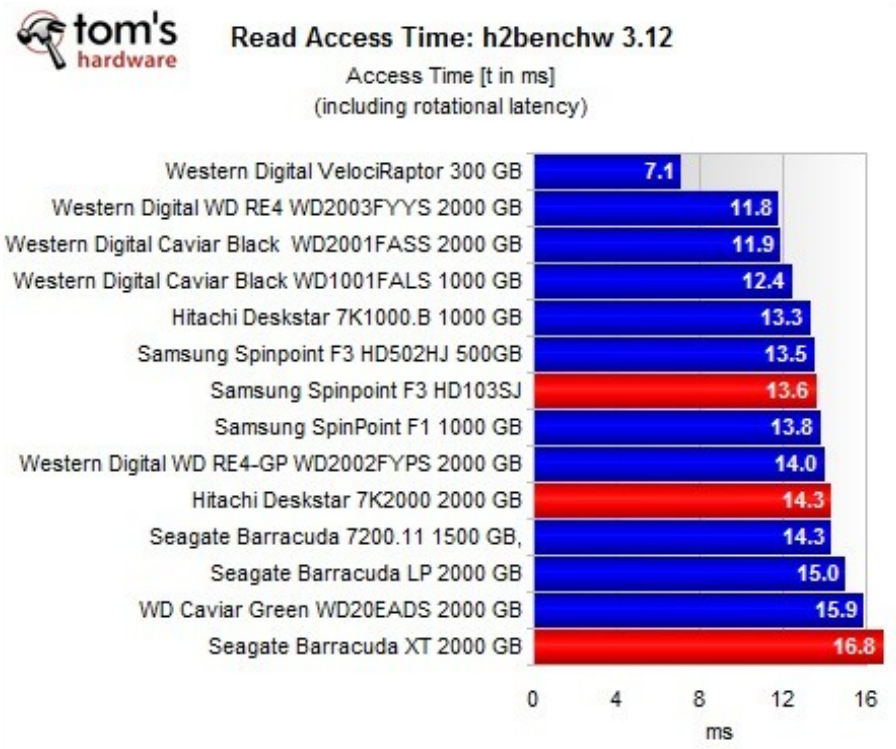
- **Zugriffszeit** ist die Summe aus:
  - **Seek Time**
    - Bewegung des Armes (Beschleunigung, Max. Geschwindigkeit, Abbremsen)
    - Langsam (Millisekunden)
  - **Rotational Delay**
    - Warten bis rotierender Sektor den Kopf erreicht hat
    - Durchschnitt: Zeit für eine halbe Plattenrotation
    - Langsam (Millisekunden)
  - **Controller Overhead**
    - Zeit die der Festplattencontroller zum Interpretieren und Koordinieren des Kommandos braucht
    - Schnell

- **Durchsatz:**
  - Hängt ab von Datendichte und Oberflächengeschwindigkeit
  - Steigt von inneren zu äußeren Spuren (bei Multiple Zone Recording)

Zugriffszeiten

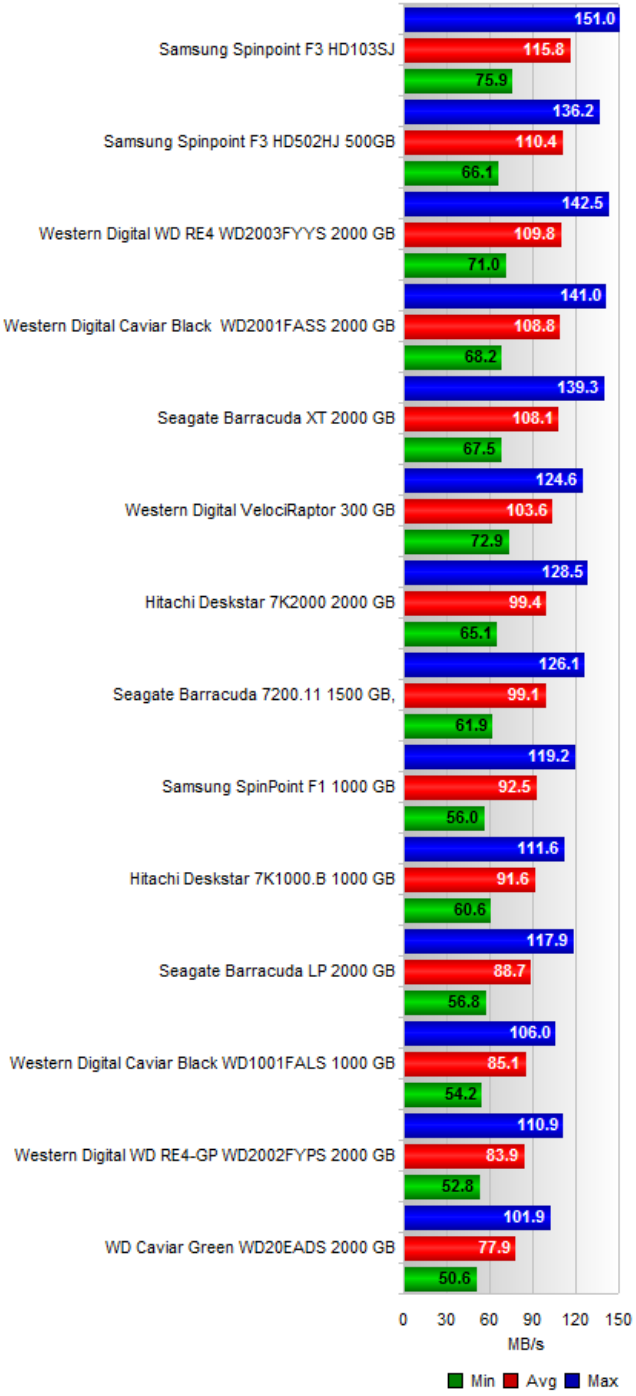
- Durchschnittliche Zugriffszeiten (2010):
  - Zwischen 7 und 17ms
  - Tendenziell haben Platten mit größerer Kapazität höhere Zugriffszeiten

- Zugriffszeiten im Vergleich:
  - RAM 10-20ns (Faktor 10^6)
  - L1-Cache 1ns (Faktor 10^7)



Durchsätze

- Durchschnittlicher Durchsatz (2010):
  - Zwischen 50 und **150 MB/s**
  - Variiert mit Rotationsgeschwindigkeit und Entfernung der Daten vom Mittelpunkt
- Zugriffszeiten im Vergleich:
  - RAM 1-5 GB/s (nur noch Faktor 30)
- Durchsatz bei byteweisen wahlfreien Zugriffen:
  - 1 Byte/Zugriffszeit = 1 Byte/7ms = **143 B/s**
- **Wahlfreie Zugriffe vermeiden!!!**
- **Benutze Blocktransfers**



Speichermodelle

- Schlussfolgerungen:
  1. Wahlfreie Zugriffe entsprechen den Cache-Misses im Hauptspeicher
     - Techniken um Cache-Misses zu reduzieren lassen sich übertragen auf Externspeicher
     - Wahlfreier Zugriff ist im Verhältnis aber **10.000 mal** langsamer als ein Cache-Miss
  2. Algorithmen die im Hauptspeicher effizient sind lassen sich i.d.R. nicht effizient 1-zu-1 auf Externspeicher übertragen
  3. Die asymptotische Laufzeit (im RAM-Modell) sagt wenig über die tatsächliche Laufzeit mit Externspeicher aus
     - Es bedarf eines Speichermodells, das die Eigenschaften von Festplatten widerspiegelt

Gallery of Processor Cache Effects (Igor Ostrovsky, 19.01.2010)
http://igoro.com/archive/gallery-of-processor-cache-effects/

I/O-INTERFACES IN C++

David Weese

René Rahn / Jongkyu Kim
WS 2015/16

Verfügbare Interfaces

- Plattformunabhängig:
  - C Standard Library (stdio.h)
  - C++ Standard Library File Streams (fstreams)

- Plattformabhängig:
  - POSIX Files (fcntl.h, aio.h)
  - Windows CRT* (io.h)
  - Windows SDK (windows.h)
  - Memory Mapped Files (sys/mman.h oder windows.h)

* C Run-Time Libraries

POSIX

- POSIX (Portable Operating System Interface) [1]
  - Standard für Betriebssysteme
  - Inhalt:
    - Basis-Defintionen (bspw. \n entspricht newline)
    - Systemschnittstellen (C-Header, bspw. stdio.h)
      - Threads, Network, **Files**, Interprocess Communication, …, **Memory Mapped Files**
    - Vorgaben für Shells und Programme

- Weitestgehend POSIX-kompatibel sind:
  - Linux
  - xBSD
  - Mac OS X
  - Solaris

[1] **POSIX.1:2008**, http://pubs.opengroup.org/onlinepubs/9699919799

Öffnen einer Datei

- Funktion zum Öffnen benötigt:
  - **Dateiname**
    - Kann Pfad enthalten, Slashes beachten (Windows: \\ und Linux, Mac: /)
  - **Flags**
    - Soll Datei erzeugt, erweitert, nur gelesen, nur geschrieben werden?
    - Beispiel (POSIX): O_CREAT, O_APPEND, O_RDONLY, O_WRONLY, O_RDWR, …
    - Beispiel (C Lib): “w+“, “a+“, “r“, “w“, “r+“
  - **Rechte**
    - Wenn Datei erzeugt wird, welche Zugriffsrechte soll sie haben (Bsp.: 755)?

- Rückgabe:
  - **Dateihandle**
    - Wird von allen folgenden Dateioperationen benötigt
    - Hat bestimmten Wert bei Fehler (bspw. -1)
    - Fehlercode kann abgefragt werden (bspw.: errno)

I/O Zugriff

- Blockweise
  - Liest/Schreibt einen zusammenhängenden Teil der Datei direkt aus dem/in den Speicher
  - Synchron/Asynchron
  - Datei kann auch direkt in den Speicher eingeblendet werden (Memory Mapping)
  - Schnell und damit für externen Speicher geeignet

- Zeichenweise
  - Liest/Schreibt einzelne Zeichen/Zeilen der Datei (nur synchron)
  - Teilweise mit automatischer Konversion der Zeilenendungen
  - Langsam und gedacht zum Parsen/Ausgeben von Dateiformaten

Synchroner Dateizugriff

- Synchroner Dateizugriff
  - Lese- und Schreiboperationen sind **blockierend**
  - Aufrufer-Thread pausiert (suspend) bis zum Abschluss der Operation

- Dateien
  - Zu einer geöffneten Datei gehört ein **Dateizeiger** = Position innerhalb der Datei
  - Operationen werden ab dem Dateizeiger ausgeführt
  - Dateizeiger kann verschoben (seek) oder abgefragt werden (tell)

- Streams
  - Streams haben **keinen Dateizeiger** (seek und tell schlagen fehl)
  - Beispiele: cin, cout, cerr

Synchroner Dateizugriff

C Standard Library

```c
#include <stdio.h>

FILE* fp = fopen(path, type);
fseek  (fp, offset, origin);
fread  (buf, num, len, fp);
fwrite (buf, num, len, fp);
fclose (fp);
```

C++ Standard Library

```c
#include <fstream>

std::fstream fs(path, flags);
fs.seekg (fp, offset, origin);
fs.read  (buf, size);
fs.write (buf, size);
```

POSIX

```c
#include <unistd.h>

int fd = open(path, flags);
lseek (fd, offset, origin);
read  (fd, buf, size);
write (fd, buf, size);
close (fd);
```

Windows CRT

```c
#include <io.h>

int fd = _open(path, flags);
_lseek (fd, offset, origin);
_read  (fd, buf, size);
_write (fd, buf, size);
_close (fd);
```

Asynchroner Dateizugriff

- Asynchroner Dateizugriff
  - Lese- und Schreiboperationen sind **nicht-blockierend**
  - Parameter der asynchronen Operation wird in `struct` gespeichert
  - Asynchrones `read`/`write` benötigt Parameter-Struktur und kehrt sofort zurück
  - Es gibt Funktionen zum …
    - Abfragen des Zustands (läuft noch/fertig/fehlerhaft) einer Operation
    - Blockierenden Warten auf Abschluss einer Operation

- Operationen können in beliebiger Reihenfolge/parallel abgearbeitet werden
- Dateizeiger wird ignoriert
  - Startposition steht in Parameter-Struktur

Asynchroner Dateizugriff in C

POSIX [1]

```
#include <aio.h>
#include <fcntl.h>
#include <errno.h>

int fd = open(”ex.txt”, O_RDONLY, 0);
aiocb cb;
memset(&cb, 0, sizeof(aiocb));
cb.aio_fildes = fd;
cb.aio_offset = 0;
cb.aio_buf    = buf;
cb.aio_nbytes = size;
aio_read(&cb);
...

if (aio_error(&cb) == EINPROGRESS)
   printf(”Still reading...\n”);
```

Windows SDK [2]

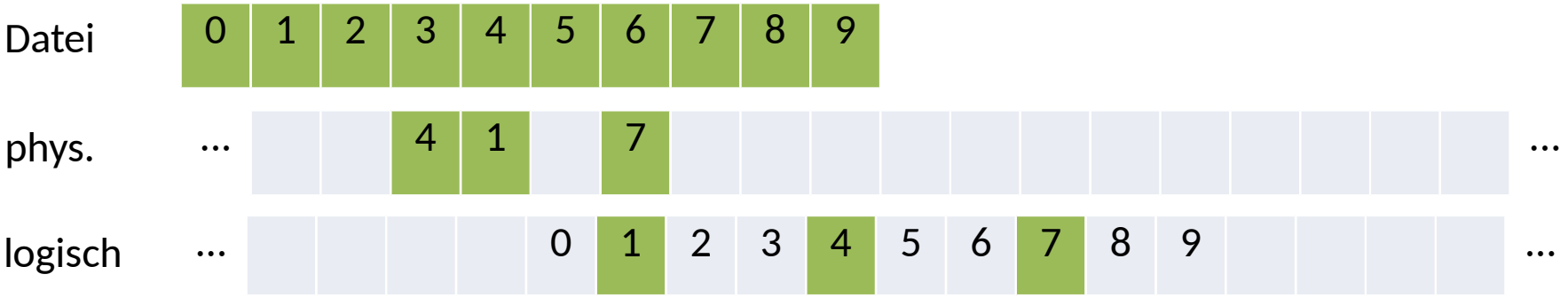
```
#include <windows.h>

HANDLE fh = CreateFile("ex.txt",
  GENERIC_READ, …, FILE_FLAG_OVERLAPPED,
   NULL);
OVERLAPPED cb;
cb.Offset     = 0;
cb.OffsetHigh = 0;
cb.hEvent     = NULL;
ReadFile(fh, buf, size, NULL, &cb);
...
LPWORD xmit_size;
GetOverlappedResult(fh, &cb, &xmit_size,
   FALSE);

if(GetLastError(&cb) == ERROR_IO_PENDING)
   printf(”Still reading...\n”);
```


[1] **Understanding the Linux Kernel 3rd Edition,** Kap. 16.4., http://www.wowebook.be/book/understanding-the-linux-kernel-3rd-edition/
[2] **MSDN – Synchronous and Asynchronous I/O,** http://msdn.microsoft.com/en-us/library/aa365683(v=VS.85).aspx

Memory Mapped Files

- Was sind Memory Mapped Files?
  - Dateien, die in den (logischen) Adressraum abgebildet werden
  - Erst bei Zugriff auf den Speicherbereich wird der entsprechende Block in den physikalischen Speicher geladen
  - Lange nicht benutzte, veränderte Blöcke werden zurückgeschrieben (analog zu Caches)

- Die Dateien können größer sein als physikalische Speicher
  - Nicht aber größer als logischer Adressraum (4GB auf 32bit Systemen)



Memory Mapped Files (II)

- Memory Mapping erstellen und beenden:
  - POSIX:

```
void *addr = mmap(NULL, fileSize, ..., fileDes, 0);

munmap(addr, fileDes);
```

  - Windows:

```
HANDLE handle = CreateFileMapping(fileHandle, ...);
void *addr = MapViewOfFile(handle, ...);

UnmapViewOfFile(addr);
CloseHandle(handle);
```

- `addr` zeigt auf das erste Byte der Datei
  - kann in `char*` Zeiger konvertiert und wie ein Feld benutzt werden

Fallstricke beim Blockweisen Zugriff

- Größe des Pufferspeicher entsprechend wählen
- Puffer muss zusammenhängend sein (bspw.: Feld, std::vector)
  - nicht zusammenhängend sind std::list, std::map, ...
- Für asynchrones I/O muss auf manchen Plattformen Pufferadresse durch Sektorgröße teilbar sein
- Klassen können größer sein als die Summe ihrer Elemente (Padding, P-VL6)
  - Notwendig damit bspw. ints an durch 4 teilbaren Adressen beginnen
  - Verschenkter Platz im Externspeicher
  - Daher: Elemente umordnen, Klassen packen

GCC

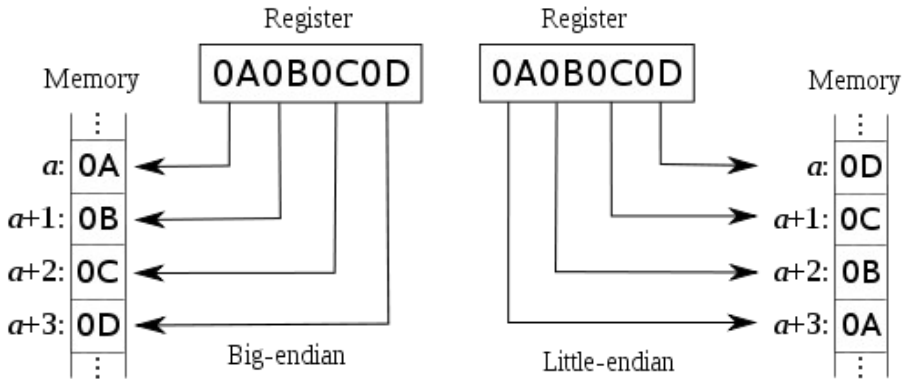
```
struct test_t {
  int  a;
  char b;
  int  c;
} __attribute__((__packed__));
```

Visual Studio

```
#pragma pack(push,1)
struct test_t {
  int  a;
  char b;
  int  c;
};
#pragma pack(pop)
```

Fallstricke beim Blockweisen Zugriff (II)

- Blockweise geschriebene Datenstrukturen haben das Format, das sie bei der Ausgabe im Speicher hatten
  - Ohne weiteres nicht plattformunabhängig (Sparc Solaris ↯ x86 Linux)
  - Größe der Datentypen ist plattformabhängig
    - long ist entweder 32 oder 64 bit breit je nach Plattform
    - Abhilfe schaffen die typedefs in stdint.h: uint8_t, int32_t, uint64_t, …
  - Paddinganforderungen sind plattformabhängig
    - Umordnen, Klassen packen
  - Endianess ist plattformabhängig
    - Festlegen auf eine Endianess (bspw. Network Byte Order = Big Endian) und auf der anderen Plattform vor Schreib- und nach Lesezugriff um drehen (htonl, ntohs, …)



Zeichenweiser I/O Zugriff

- C++ File Streams:
  - ifstream (nur lesen)
  - ofstream (nur schreiben)
  - fstream (beides)
- Sind abgeleitet von istream, ostream oder iostream
  - File Streams haben also mind. die Funktionalität von `cin` und `cout`
- File Stream öffnen:

```
std::fstream fs;
fs.open(path, flags);

fs.close();
```

- Oder öffnen im Konstruktor und schließen im Destruktor:

```
std::fstream fs(path, flags);
```

Einzelne Zeichen lesen/schreiben

- Ein Zeichen lesen:

```
char c;
c = fs.get();      // char lesen und weitergehen
c = fs.peek();     // char lesen aber nicht weitergehen
fs.unget();        // ein Zeichen zurückgehen
```

- Ein Zeichen schreiben:

```
fs.put(c);         // char schreiben und weitergehen
```

- Stream-Status abfragen:

```
bool b;
b = fs.is_open(); // Ist die Datei geöffnet?
b = fs.eof();     // Hat letzte Operation Streamende überschritten?
b = fs.good();    // Weder EOF noch Formatfehler?
```

Formatierte Ein-/Ausgabe

- ostream hat operator << für alle eingebauten Typen überladen:

```
int i = 42;
double d = 3.14;
fs << i;
fs << "  ";
fs << d;                        // Datei: “42  3.14“
```

- operator << gibt Referenz auf Stream zurück Schachtelung möglich:

```
((fs << i) << "  ") << d;
fs << i << "  " << d;          // wird von links nach rechts ausgeführt
```

- operator >> (istream) überspringt Leerzeichen und liest Wert:

```
fs >> i;              // überspringe Leerzeichen, lies Ganzzahl
fs >> d;              // überspringe Leerzeichen, lies Gleitkommazahl

std::string s;
fs >> i >> s;         // lies Ganzzahl, lies String bis Leerzeichen
```

Zeilenende

- Lies String bis Zeilenende oder bis zu frei wählbarem Zeichen:

```
getline (fs, str);                  // lies ganze Zeile
getline (fs, str, ',');             // lies bis Komma
```

- Zeilenende schreiben:

```
fs << "Eine Zeile\n";               // eine Zeile schreiben
fs << "Eine Zeile" << std::endl;    // eine Zeile schreiben + flush
```

  - flush leert synchron Schreibpuffer in Datei (langsam)
- Zeilenende wird verschieden kodiert:
  - Windows benutzt CR+LF (`\r\n`)
  - Mac OS X benutzt LF (`\n`)
- IOStreams konvertieren Zeilenendungen implizit in \n
  - Konvertierung kann mit ios::binary abgeschaltet werden (schneller)

Stream Manipulatoren

- C++ bietet verschiedene Stream Manipulatoren mit versch. Aufgaben:
- Stream Manipulatoren (definiert in <iomanip>) beeinflussen:
  - Setzen der Breite und des Füllzeichens von Feldern
    - cout << setw(6) << setfill(0) << 3.2; // 0003.2
  - Setzen der angezeigten Präzision von Gleitkommazahlen
    - cout << setprecision(3) << 2.71828183;// 2.72
  - Setzen und Löschen von Formatierungsflags
  - Flushen von gepufferten Streams
    - cout << flush;
    - cout << endl; // entspricht cout << '\n' << flush;

Formatierungsflags

- ios::showpoint
  - Dezimalpunkt immer anzeigen
- ios::showpos
  - “+” vor positiven Zahlen anzeigen
- ios::basefield
  - Wählt die Zahlenbasis
    - ios::dec Dezimalzahlen
    - ios::oct Oktalzahlen
    - ios::hex Hexadezimalzahlen
- ios::floatfield
  - Art der Gleitkommadarstellung
    - ios::fixed feste Anzahl Stellen
    - ios::scientific wissenschaftliche Notation
- ios::adjustfield
  - Horizontale Ausrichtung
    - ios::left linksbündig
    - ios::right rechtsbündig
    - ios::internal Vorzeichen linksbündig, Wert rechtsbündig

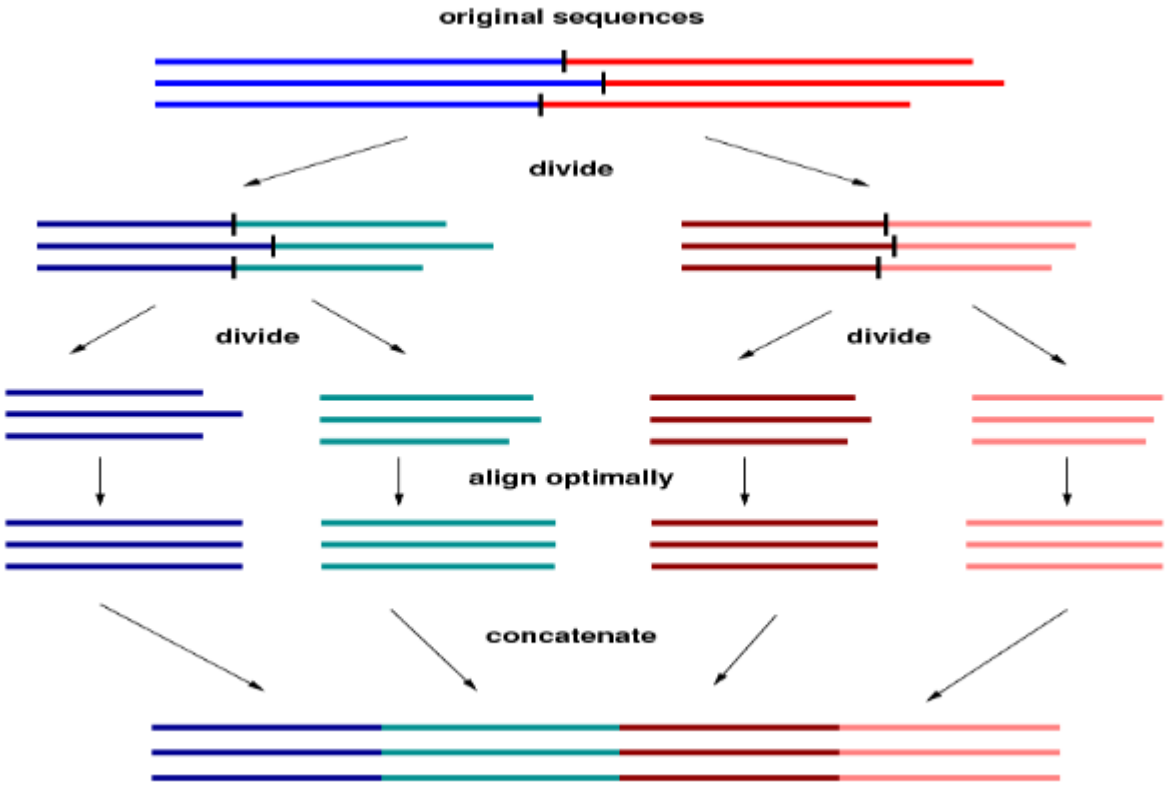
BEMERKUNGEN ZUR P-AUFGABE

David Weese
© 2010/11

René Rahn / Jongkyu Kim
WS 2015/16

Hinweise zu Aufgabe 6

- Divide-and-Conquer Alignment
  - Komplexität des vollständigen Dynamic-Programming zu groß
  - Aufteilen in Blöcke, diese optimal lösen und dann wieder zusammenfügen
  - Problem: Optimale Aufteilung finden



C-optimale Schnittstellen

- Diese Schnittstellen versuchen die zusätzlichen Kosten in den paarweisen Alignments zu minimieren
  - Damit sind es keine optimalen Schnittstellen im Sinne des MSA

- Paarweise *additional cost* Matrizen speichern die Zusatzkosten die ein Schnitt auf das paarweise Alignment der beiden Sequenzen hat
  - Beispiel: CTATAC gegen GTATC.
    - Optimales Alignment hat Distanz 2:
    - Schnitt an den Positionen (4,3) teilt das Alignment in 2 Blöcke, und das „bestmögliche“ Alignment hat eine Distanz von 3
    - Die *additional cost* für den Schnitt (4,3) ist somit 1

```
CTATAC
GTAT-C
```

```
CTAT | AC
GTA- | TC
```

Additional Cost Matrix

		C	T	A	T	A	C
	0	2	4	6	8	10	12
G	2	1	3	5	7	9	11
T	4	3	1	3	5	7	9
A	6	5	3	1	3	5	7
T	8	7	5	3	1	3	5
C	10	8	7	5	3	2	3

Figure 2a

	C	T	A	T	A	C	
G	3	4	3	4	6	8	10
T	4	2	3	2	4	6	8
A	6	4	2	2	2	4	6
T	8	6	4	2	1	2	4
C	10	8	6	4	2	0	2
	12	10	8	6	4	2	0

Figure 2b

		C	T	A	T	A	C
	0	3	4	7	11	15	19
G	3	0	3	4	8	12	16
T	7	4	0	2	4	8	12
A	11	8	4	0	1	4	8
T	15	12	8	4	0	0	4
C	19	15	12	8	4	1	0

Figure 2c

Figures 2a and 2b show the standard dynamic programming distance matrices of the sequences **s** = GTATC and **t** = CTATAC (using unit cost and penalty +2 for each single insertion/deletion), computed from the upper left to the lower right (Fig. 2a) and from the lower right to the upper left (Fig. 2b). Figure 2c displays the *additional–cost* matrix, containing the values $C_{\mathbf{s},\mathbf{t}}[c,d]$ for each pair of slicing positions (c,d), e.g.

$$C_{\mathbf{s},\mathbf{t}}[2,2] = w_{opt}[\text{CT,GT}] + w_{opt}[\text{ATAC,ATC}] - w_{opt}[\text{CTATAC,GTATC}] = 1 + 2 - 3 = 0$$

$$C_{\mathbf{s},\mathbf{t}}[4,3] = w_{opt}[\text{CTAT,GTA}] + w_{opt}[\text{AC,TC}] - w_{opt}[\text{CTATAC,GTATC}] = 3 + 1 - 3 = 1$$

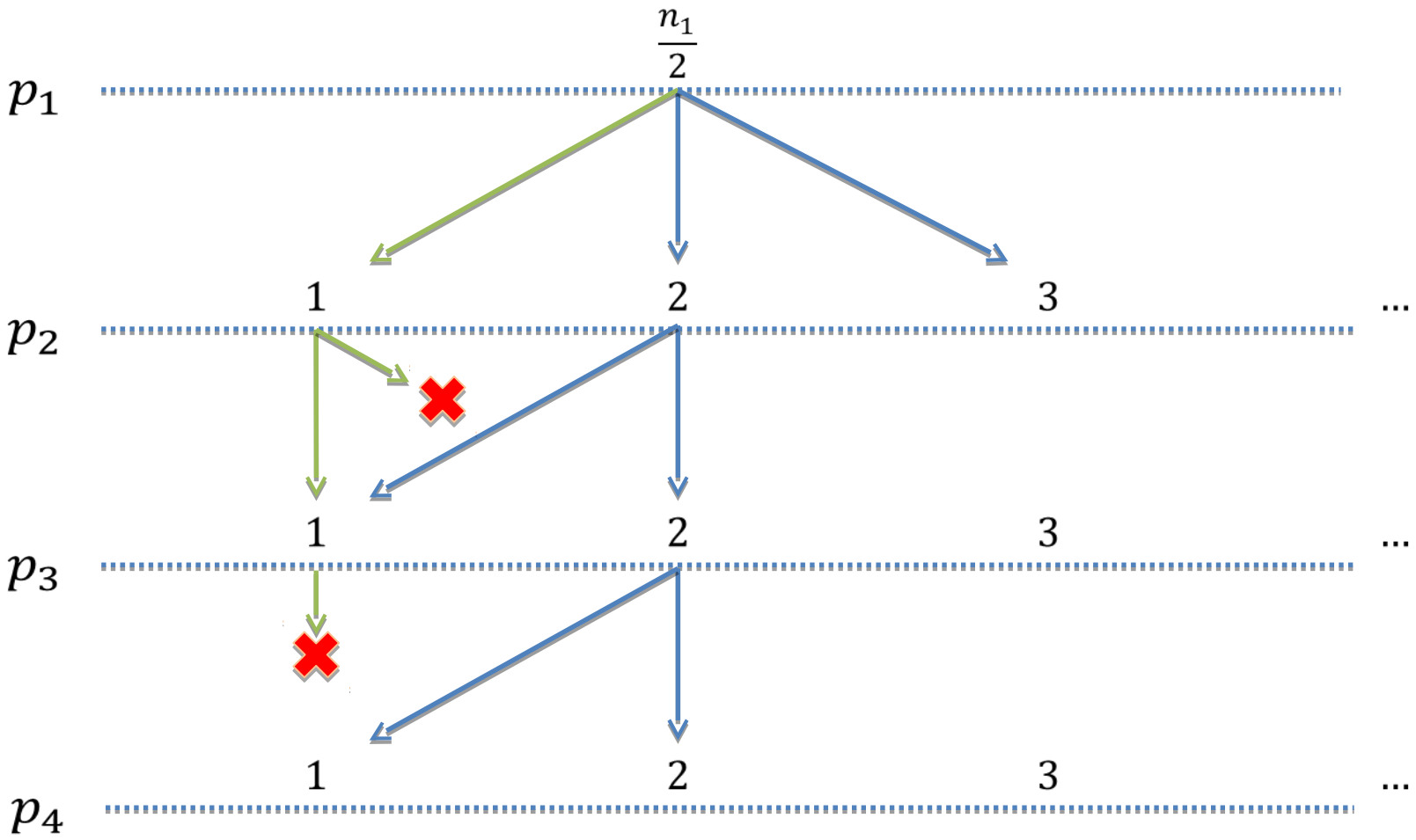
Finden der C-opt. Schittstellen

- Enumerativ:
  - Alle Kombinationen:

    $s_1 \in \{1, \ldots, n_1\}, s_2, \ldots, s_k \in \{1, \ldots, n_k\}$

    durchgehen und minimale Kosten berechnen.
  - Aufwand zu groß: $\Pi_{i=1}^{k} |s_i|$
- Besser: Branch-and-Bound
  - Rekursiv Suchraum abarbeiten:
    - Festsetzen der Schnittpositionen in der Reihenfolge $s_1, s_2, \ldots$ und berechnen der bisherigen additional Costs
    - Die Kosten können nur steigen!
    - Abbrechen wenn : $C_{curr} \geq C_{best}$

Branch and Bound (II)



Je näher die Kostengrenze am Optimum ist, umso mehr Teilbäume können übersprungen werden