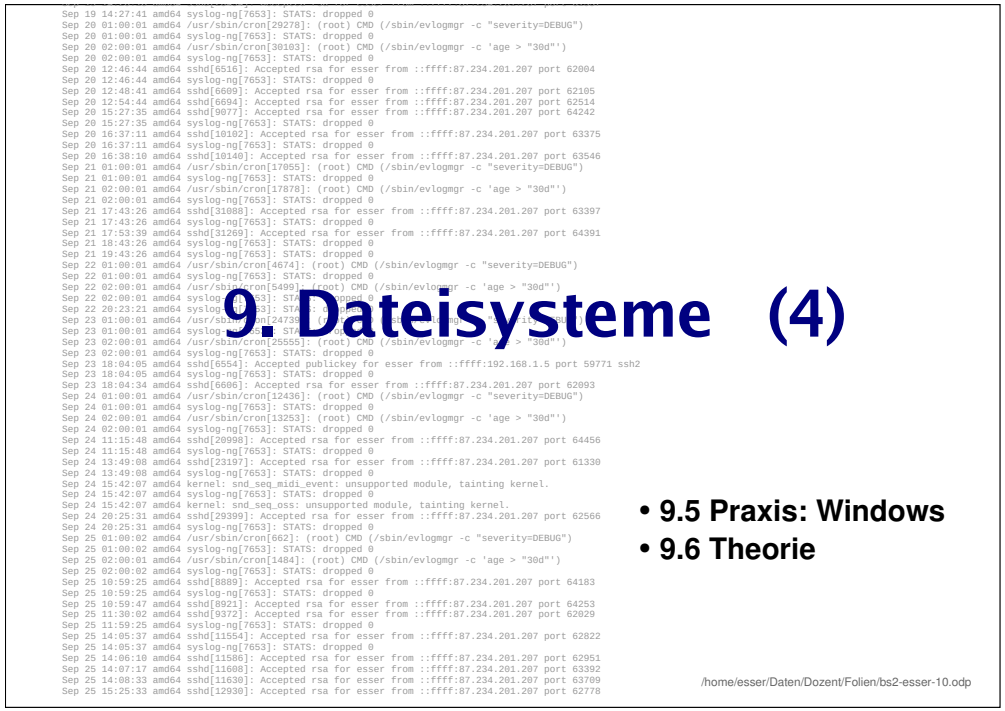


9. Dateisysteme (4)

- 9.5 Praxis: Windows
- 9.6 Theorie

/home/esser/Daten/Dozent/Folien/bs2-esser-10.odp

Windows-Dateizugriff (1)

- Auf elementarer Ebene stehen die gleichen Dateizugriffsfunktionen zur Verfügung wie unter Linux und haben die gleiche Syntax, also
  - fd=**open** (), **close** (fd)
  - **read** (fd,...), **write** (fd,...)
- Auch **fopen**, **fread** etc. gibt es unter Windows.
- Der Standardweg führt aber über Windows-spezifische Dateifunktionen

9.5 Praxis: Dateizugriff unter Windows

Windows-Dateizugriff (2)

- Windows-Standardfunktionen sind objekt-basiert:
  - handle = **CreateFile ()** erzeugt / öffnet Datei
  - **ReadFile** (handle, ...) liest aus Datei
  - **WriteFile** (handle, ...) schreibt in Datei
  - **CloseFile** (handle) schließt Datei
  - **SetFilePointer** (handle,...)
    Lese/-Schreibposition ändern
  - **WaitForSingleObject** (handle)
    wartet auf Beenden von I/O-Operation auf Datei

Windows-Dateizugriff (3)

- Datei öffnen / erzeugen: **CreateFile**

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile);
```

Beispiel 1: Datei zum Lesen öffnen

```
hFile = CreateFile("ONE.TXT",          // open ONE.TXT
        GENERIC_READ,                  // open for reading
        0,                             // do not share
        NULL,                          // no security
        OPEN_EXISTING,                 // existing file only
        FILE_ATTRIBUTE_NORMAL,         // normal file
        NULL);                         // no attr. template
```


Windows-Dateizugriff (4)

Beispiel 2: Datei im Append-Modus öffnen

```
hAppend = CreateFile("TWO.TXT", // open TWO.TXT
  GENERIC_WRITE,                // open for writing
  0,                            // do not share
  NULL,                         // no security
  OPEN_ALWAYS,                  // open or create
  FILE_ATTRIBUTE_NORMAL,        // normal file
  NULL);                        // no attr. template
```

Daten von ONE.TXT an TWO.TXT anhängen; mit Locking

```
do {
  if (ReadFile(hFile, buff, 4096, &dwBytesRead, NULL)) {
    dwPos = SetFilePointer(hAppend, 0, NULL, FILE_END);
    LockFile(hAppend, dwPos, 0, dwPos + dwBytesRead, 0);
    WriteFile(hAppend, buff, dwBytesRead,&dwBytesWritten, NULL);
    UnlockFile(hAppend, dwPos, 0, dwPos + dwBytesRead, 0);
  }
} while (dwBytesRead == 4096);
// Close both files.
CloseHandle(hFile);
CloseHandle(hAppend);
```


Windows-Dateizugriff (5)

- Lesen: **ReadFile**

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped);
```

- Schreiben: **WriteFile**

```
BOOL WriteFile(
  HANDLE hFile,
  LPCVOID lpBuffer,
  DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped);
```


Windows-Dateizugriff (6)

- Beispiel: Datei kopieren und Daten konvertieren

```
#include <windows.h>
#include <stdio.h>
#define BUFSIZE 512

int main() {
  HANDLE hFile, hTempFile;
  DWORD dwRetVal, dwBytesRead, dwBytesWritten, dwBufSize=BUFSIZE;
  UINT uRetVal; BOOL fSuccess;
  char szTempName[BUFSIZE], buffer[BUFSIZE], lpPathBuffer[BUFSIZE];

  // Open the existing file.
  hFile = CreateFile("original.txt",        // file name
                     GENERIC_READ,          // open for reading
                     0,                     // do not share
                     NULL,                  // default security
                     OPEN_EXISTING,         // existing file only
                     FILE_ATTRIBUTE_NORMAL, // normal file
                     NULL);                 // no template
  if (hFile == INVALID_HANDLE_VALUE) {
      printf ("CreateFile failed with error %d.\n", GetLastError()); return (1);
  }
```

Windows-Dateizugriff (7)

```
// Get the temp path.
dwRetVal = GetTempPath(dwBufSize,     // length of the buffer
                       lpPathBuffer); // buffer for path
if (dwRetVal > dwBufSize || (dwRetVal == 0)) {
    printf ("GetTempPath failed with error %d.\n", GetLastError()); return (2);
}

// Create a temporary file.
uRetVal = GetTempFileName(lpPathBuffer, // directory for tmp files
                          "NEW",        // temp file name prefix
                          0,            // create unique name
                          szTempName);  // buffer for name
if (uRetVal == 0) {
    printf ("GetTempFileName failed with error %d.\n", GetLastError()); return (3);
}

// Create the new file to write the upper-case version to.
hTempFile = CreateFile((LPTSTR) szTempName, // file name
                       GENERIC_READ | GENERIC_WRITE, // open r-w
                       0,                    // do not share
                       NULL,                 // default security
                       CREATE_ALWAYS,        // overwrite existing
                       FILE_ATTRIBUTE_NORMAL,// normal file
                       NULL);                // no template
if (hTempFile == INVALID_HANDLE_VALUE) {
    printf ("CreateFile failed with error %d.\n", GetLastError()); return (4);
}
```


Windows-Dateizugriff (8)

```
// Read BUFSIZE blocks to the buffer. Change all characters in the buffer to
// upper case. Write the buffer to the temporary file.
do {
    if (ReadFile(hFile, buffer, BUFSIZE, &dwBytesRead, NULL)) {
        CharUpperBuff(buffer, dwBytesRead);
        fSuccess = WriteFile(hTempFile, buffer, dwBytesRead,
                             &dwBytesWritten, NULL);
        if (!fSuccess) {
            printf ("WriteFile failed with error %d.\n", GetLastError()); return (5);
        }
    } else {
        printf ("ReadFile failed with error %d.\n", GetLastError()); return (6);
    }
} while (dwBytesRead == BUFSIZE);
// Close the handles to the files.
fSuccess = CloseHandle (hFile);
if (!fSuccess) {
   printf ("CloseHandle failed with error %d.\n", GetLastError()); return (7);
}
fSuccess = CloseHandle (hTempFile);
if (!fSuccess) {
   printf ("CloseHandle failed with error %d.\n", GetLastError()); return (8);
}
// Move the temporary file to the new text file.
fSuccess = MoveFileEx(szTempName, "allcaps.txt", MOVEFILE_REPLACE_EXISTING);
if (!fSuccess) {
    printf ("MoveFileEx failed with error %d.\n", GetLastError()); return (9);
}
return (0);
}
```


Windows-Dateizugriff (9)

- An bestimmte Stelle springen

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

hFile: [in] A handle to the file that has a file pointer to be moved.

lDistanceToMove: [in] The low order 32-bits of a signed value that specifies the number of bytes to move the file pointer.

lpDistanceToMoveHigh: [in, out, optional] A pointer to the high order 32-bits of the signed 64-bit distance to move. If you do not need the high order 32-bits, this pointer must be set to NULL.

dwMoveMethod: [in] The starting point for the file pointer move:

FILE_BEGIN	The starting point is 0 (zero) or the beginning of the file.
FILE_CURRENT	The starting point is the current value of the file pointer.
FILE_END	The starting point is the current end-of-file position.

Dateien suchen (1)

- hFind = **FindFirstFile** ()

```
HANDLE FindFirstFile(
  LPCTSTR lpFileName,
  LPWIN32_FIND_DATA lpFindFileData
);
```

lpFileName: [in] A pointer to a null-terminated string that specifies a valid directory or path, and file name that can contain wildcard characters, for example, an asterisk (*) or a question mark (?).

lpFindFileData: [out] A pointer to the WIN32_FIND_DATA structure that receives information about a found file or subdirectory.

- **FindNextFile** (hFind)

```
BOOL FindNextFile(
  HANDLE hFindFile,
  LPWIN32_FIND_DATA lpFindFileData
);
```

- **FindClose** (hFind)

hFindFile: [in] Search handle returned by a previous call to the FindFirstFile or FindFirstFileEx function.

lpFindFileData: [out] Pointer to the WIN32_FIND_DATA structure that receives information about the found file or subdirectory. The structure can be used in subsequent calls to FindNextFile to indicate from which file to continue the search.

http://msdn2.microsoft.com/en-us/library/aa364418.aspx, http://msdn2.microsoft.com/en-us/library/aa364428.aspx

Dateien suchen (2)

Quelle: http://msdn2.microsoft.com/en-us/library/aa365200.aspx

```
#define _WIN32_WINNT 0x0501
#include <windows.h>
#include <stdio.h>
#include <strsafe.h>
#include <malloc.h>
#define BUFSIZE MAX_PATH

int main(int argc, char *argv[]) {
   WIN32_FIND_DATA FindFileData;
   HANDLE hFind = INVALID_HANDLE_VALUE;
   DWORD dwError; LPSTR DirSpec;
   size_t length_of_arg;

   DirSpec = (LPSTR) malloc (BUFSIZE);

   // Check for command-line parameter; otherwise, print usage.
   if(argc != 2) { printf("Usage: Test <dir>\n"); return 2; }

   // Check that the input is not larger than allowed.
   StringCbLength(argv[1], BUFSIZE, &length_of_arg);
   if (length_of_arg > (BUFSIZE - 2)) {
      printf("Input directory is too large.\n");
      return 3;
   }

   printf ("Target directory is %s.\n", argv[1]);
```

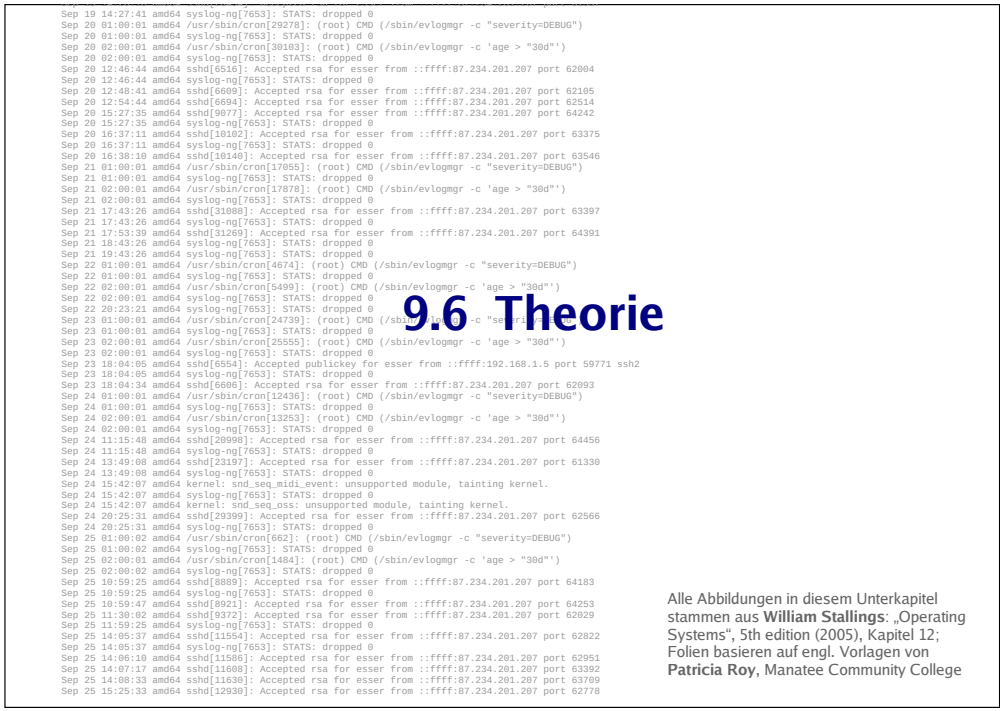

Dateien suchen (3)

```
   // Prepare string for use with FindFile functions.  First, copy the
   // string to a buffer, then append '\*' to the directory name.
   StringCbCopyN (DirSpec, BUFSIZE, argv[1], length_of_arg+1);
   StringCbCatN (DirSpec, BUFSIZE, "\\*", 3);

   // Find the first file in the directory.
   hFind = FindFirstFile(DirSpec, &FindFileData);

   if (hFind == INVALID_HANDLE_VALUE) {
      printf ("Invalid file handle. Error is %u.\n", GetLastError()); return (-1);
   } else {
      printf ("First file name is %s.\n", FindFileData.cFileName);

      // List all the other files in the directory.
      while (FindNextFile(hFind, &FindFileData) != 0) {
         printf ("Next file name is %s.\n", FindFileData.cFileName);
      }
      dwError = GetLastError();
      FindClose(hFind);
      if (dwError != ERROR_NO_MORE_FILES) {
         printf ("FindNextFile error. Error is %u.\n", dwError); return (-1);
      }
   }

   free(DirSpec);
   return (0);
}
```


Windows: stdin, stdout, stderr

- Wie unter Unix gibt es drei Standard-Datei-Deskriptoren
  - Standard Input (0, stdin)
  - Standard Output (1, stdout)
  - Standard Error Output (2, stderr)
- Zugriff auf diese ist genau wie unter Linux möglich:

```
#include <io.h>     /* Linux: <unistd.h> */
void main() {
  if (write(1, "Here is some data\n", 18) != 18)
    write(2, "A write error has occurred on file descriptor 1\n",46);
}
```

Alle Abbildungen in diesem Unterkapitel stammen aus **William Stallings**: „Operating Systems“, 5th edition (2005), Kapitel 12; Folien basieren auf engl. Vorlagen von **Patricia Roy**, Manatee Community College

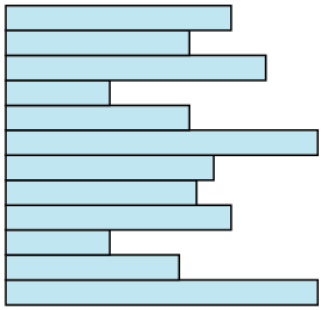
Datei-Organisation (1)

Kriterien für Dateiorganisation

- **Niedrige Zugriffszeit**
  - wichtig bei direktem Zugriff auf einen Datensatz
  - nicht nötig bei Verarbeitung der kompletten Datei
- **Einfache Updates**
  - Dateien auf CD werden z.B. nicht aktualisiert
- **Effiziente Ablage**
  - Minimale Redundanz in den Daten
  - Redundanz kann Zugriff beschleunigen (z. B. Index)
- **Einfache Wartung**
- **Zuverlässigkeit**

Datei-Organisation (2)

- Stapel (Pile)
  - Daten ablegen in der Reihenfolge des Entstehens
  - Zweck: große Datenmengen anhäufen und sichern
  - Records haben unterschiedliche Felder
  - Keine Struktur
  - Record-Zugriff durch vollständige Durchsuchung



Variable-length records
Variable set of fields
Chronological order

(a) Pile File

Datei-Organisation (3)

- Beispiel für **Stapel**: Aufbau typischer Konfigurationsdateien (hier: Wine-Konfiguration)

```
[Drive A]
"Type"="floppy"
"Path"="/media/floppy"
"Label"="/media/floppy"
"Device"="/dev/fd0"

[Drive C]
"Path"="${HOME}/.wine/fake_windows"
"Type"="hd"
"Label"="fake_windows"
"Filesystem"="win95"
```

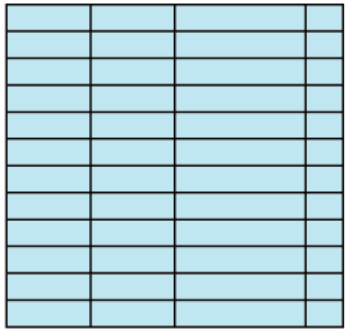
```
[Drive D]
"Path"="/media/cdrecorder"
"Label"="cdrecorder"
"Type"="cdrom"
"FS"="win95"
;"Device"="/dev/cdrecorder"

[Drive E]
"Path"="/media/cdrom"
"Label"="cdrom"
"Type"="cdrom"
"FS"="win95"
;"Device"="/dev/cdrom"
```


Datei-Organisation (4)

- Sequentielle Datei (1)
  - festes Format für Records
  - Alle Records haben die gleiche Länge
  - Alle Felder sind gleich (Reihenfolge und Länge)
  - Feldnamen und -längen sind Eigenschaften der Datei
  - Ein Feld ist „Schlüsselfeld“
    - identifiziert den Datensatz eindeutig
    - Records in Schlüsselreihenfolge sortiert

Datei-Organisation (5)

- Sequentielle Datei (2)
  - Neue Records in einem Logfile oder in einer Transaktionsdatei (zwischen)speichern
  - Gelegentlich Update ausführen, bei dem die Logdaten in die Masterdatei eingepflegt werden



Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

(b) Sequential File

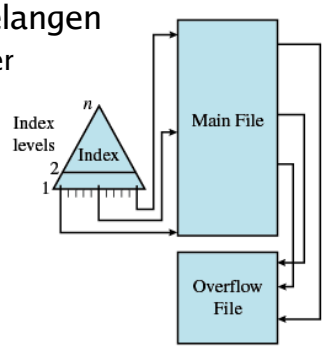
Datei-Organisation (6)

- Beispiel für **sequentielle Datei**:

```
struct data {
  int id;            // Länge  4
  char name[30];     // Länge 30
  char vorname[30];  // Länge 30
};

int blocksize = sizeof(struct data);
                     // 30+30+4=64

struct data buffer;

main() {
  printf ("Blockgroesse: %d \n", blocksize);
}
```


Datei-Organisation (7)

- Index-sequentielle Datei
  - Index bietet Möglichkeit, schnell in „die Nähe“ des gesuchten Datensatzes zu gelangen
    - Enthält Schlüsselfeld und Zeiger in die Hauptdatei
    - Index nach größtem Schlüsselwert durchsuchen, der <= dem gesuchten Schlüsselwert ist
    - Suche in der Hauptdatei fortsetzen (an der Stelle, auf die der Index-Eintrag zeigt)



(c) Indexed Sequential File

Datei-Organisation (8)

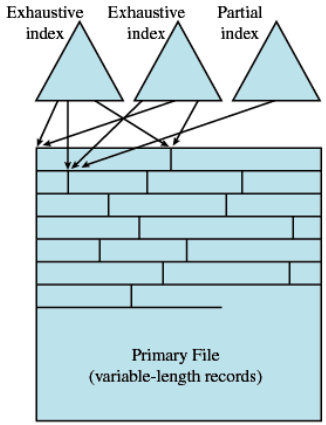
- Vergleich zwischen sequentiellen und index-sequentiellen Dateien
  - Beispiel: Datei enthält eine Million Records
  - Im Schnitt 500.000 Zugriffe nötig, um einen Record in einer sequentiellen Datei zu finden
  - Hat ein Index 1000 Einträge, braucht man
    - im Schnitt 500 Zugriffe, um den passenden Index-Eintrag zu finden, und
    - danach weitere 500 Zugriffe in der Hauptdatei
  - Damit durchschnittlich 1000 Zugriffe (Faktor 1/500)

Datei-Organisation (9)

- Index-sequentielle Datei
  - Neue Records in eine Überlaufdatei schreiben
  - Record in Hauptdatei, der „vor“ den neuen Eintrag gehört, wird aktualisiert (Zeiger auf den neuen Record)
  - Überlaufdatei und Hauptdatei werden regelmäßig zusammengefasst
  - Mehrstufiger Index erhöht Performance noch weiter (vgl. mehrstufige Seitentabellen)

Datei-Organisation (10)

- Index-Datei
  - Mehrere Indizes für verschiedene Felder
  - Vollständiger Index: für jeden Record in der Hauptdatei gibt es einen Index-Eintrag
  - Partieller Index: nur Einträge für Records, in denen das Feld existiert



(d) Indexed File

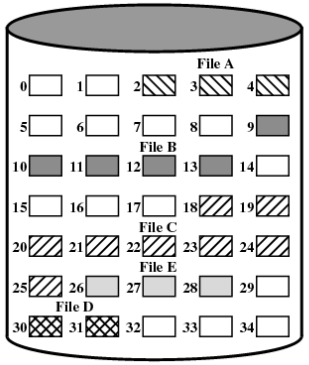
Sekundärspeicher-Management (1)

- Platz für Dateien zuordnen (allozieren)
- Platz verwalten, der für Allokationen zur Verfügung steht
- Pre-Allocation
  - muss maximale Dateigröße schon bei Allokation kennen
  - Es ist schwer, verlässlich die Maximalgröße einer Datei vorherzusagen
  - Tendenz dazu, die Dateigröße zu überschätzen, um nicht in Platznot zu geraten

Sekundärspeicher-Management (2)

- Zusammenhängende Allokation
  - Beim Erzeugen einer Datei wird eine bestimmte Menge Blöcke alloziert
  - Ein einziger Eintrag in der Dateizuordnungstabelle
    - Anfangsblock und Länge der Datei
- Es kommt zu externer Fragmentierung
  - Dateisystem defragmentieren

Sekundärspeicher-Management (3)

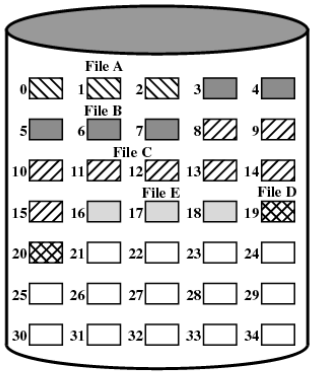


File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Figure 12.7 Contiguous File Allocation

Sekundärspeicher-Management (4)



File Allocation Table

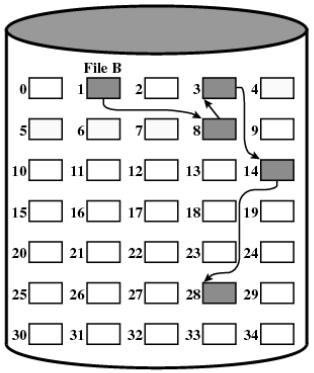
File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Figure 12.8 Contiguous File Allocation (After Compaction)

Sekundärspeicher-Management (5)

- Schlangen-Allokation
  - Allokation auf Basis individueller Blöcke
  - Jeder Block enthält einen Zeiger zum nächsten Block in der Schlange
  - Ein einziger Eintrag in der Zuordnungstabelle
    - Anfangsblock und Dateilänge
- Keine externe Fragmentierung
- optimal für sequentielle Dateien
- Lokalitätsprinzip wird nicht berücksichtigt
- Weit verteilte Blöcke zusammenfügen: **Konsolidierung**

Sekundärspeicher-Management (6)



File Allocation Table

File Name	Start Block	Length
...	...	...
File B	1	5
...	...	...

Figure 12.9 Chained Allocation

Sekundärspeicher-Management (7)

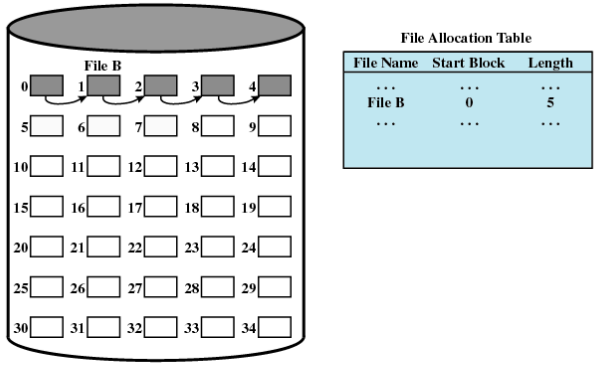


Figure 12.10 Chained Allocation (After Consolidation)

Sekundärspeicher-Management (8)

- Index-Allokation
  - Dateizuordnungstabelle enthält für jede Datei einen separaten 1-Level-Index
  - Index hat für jede „Portion“, die der Datei zugeordnet wurde, einen Eintrag
    (Portionen sind die kleinsten Einheiten, die alloziert werden, von einzelnen Blöcken bis zur kompletten Datei)
  - In Dateizuordnungstabelle steht die Nummer des Blocks, der den Index enthält

Sekundärspeicher-Management (9)

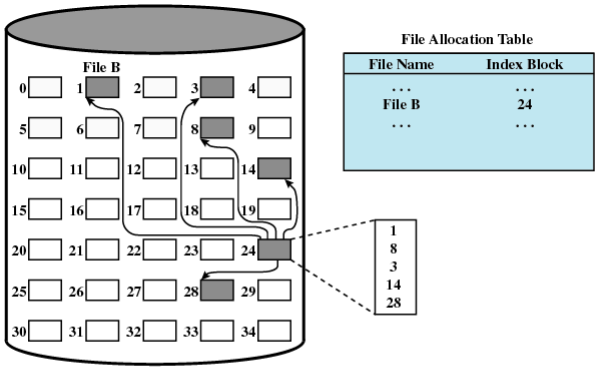


Figure 12.11 Indexed Allocation with Block Portions

Sekundärspeicher-Management (10)

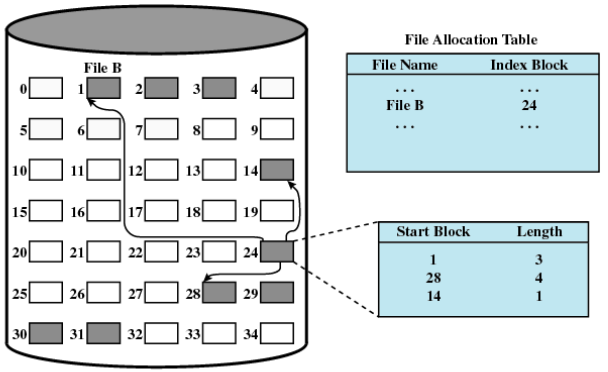


Figure 12.12 Indexed Allocation with Variable-Length Portions

Sekundärspeicher-Management (11)

Verwaltung des freien Plattenplatzes

- Platz, der zur Zeit nicht verwendet wird, verwalten
- Neben Dateizuordnungstabelle noch **Plattenzuordnungstabelle** mit Informationen über Freiräume
- **Bit-Table:** Bit-Vektor, in dem jedes Bit einen Block repräsentiert;
  0 = freier Block 1 = belegter Block
  (vgl. Bit-Vektoren für Verwaltung von freiem Hauptspeicher)
  -> Bit-Table passt evtl. komplett ins RAM (effizient)

Sekundärspeicher-Management (12)

- **Free Block List:** Blöcke werden durch nummeriert; eine Liste enthält die Nummern aller freien Blöcke
  - Liste zu groß, um sie komplett im Speicher zu halten
  - Abhilfe: Liste als FIFO-Warteschlange betrachten
    - jeweils ein paar 1000 Einträge vom Anfang und Ende der Schlange im Speicher halten
    - Block allozieren: Eintrag vom Anfang der Schlange nehmen
    - Block deallozieren: Eintrag ans Ende der Schlange hängen
    - ab und zu Anfangs- und Endstücke mit Platte synchronisieren

Vorschau

nach den Weihnachtsferien:

Standard-Dateisysteme
- Linux: ext3
- Windows: NTFS