**Fakultät Informatik/Mathematik**

Lehrstuhl für Betriebssysteme

HOCHSCHULE FÜR
TECHNIK UND WIRTSCHAFT
DRESDEN
UNIVERSITY OF APPLIED SCIENCES

Vorlesungsmitschrift

# BETRIEBSSYSTEME 2

Mitschrift von

Falk-Jonatan Strube

Vorlesung von

Prof. Dr.-Ing. Robert Baumgartl

27. März 2018

# INHALTSVERZEICHNIS

# VORBEMERKUNGEN

## PRÜFUNG

keine Unterlagen

# 1 SYNCHRONISATION

Vorlesung
22.03.2017

## 1.1 BEISPIEL BANKAUTOMAT

**bs2-00-synchronisation.pdf**
Folie 2

### MÖGLICHER (TYPISCHER) ABLAUF

**bs2-00-synchronisation.pdf**
Folie 3

## 1.2 RACE CONDITION

**bs2-00-synchronisation.pdf**
Folie 4

## 1.3 KRITISCHER ABSCHNITT

**bs2-00-synchronisation.pdf**
Folie 5

→ Lost-Update-Problem!

Zugriffsoperationen zur gemeinsam genutzten Variable bilden einen so genannten KRITISCHEN ABSCHNITT (critical section).

Zur Erinnerung: Scheduling (kooperativ → syscalls [⇒ Semaphor], . . . )

## 1.4 STEUERUNG DES KRITISCHEN ABSCHNITTS

**bs2-00-synchronisation.pdf**
Folie 6

### STEUERUNG DURCH KLAMMERNDE FUNKTIONEN

**bs2-00-synchronisation.pdf**
Folie 7

→ dezentrale Steuerung
Was müssen die Funktionen `enter_cs()` und `leave_cs()` tun?

**bs2-00-synchronisation.pdf**
Folie 8

## 1.5 REALISIERUNGSVARIANTEN

**bs2-00-synchronisation.pdf**
Folie 9

## 1.6 DEZENTRALE LÖSUNGSVERSUCHE

### 1.6.1 NAIVER VERSUCH 1: „PING-PONG“

**bs2-00-synchronisation.pdf**
Folie 11

⇒ funktioniert daher nicht vernünftig.

### 1.6.2 VERSUCH 2: VARIABLE ZEIGT FREIEN KRITISCHEN ABSCHNITT AN

**bs2-00-synchronisation.pdf**
Folie 12

#### BEWERTUNG

**bs2-00-synchronisation.pdf**
Folie 13

⇒ funktioniert gar nicht, da kritischer Abschnitt von beiden betreten werden kann!

### 1.6.3 VERSUCH 3: MEHRERE VARIABLEN ZEIGEN FREIEN KRITISCHEN ABSCHNITT AN

**bs2-00-synchronisation.pdf**
Folie 14

#### BEWERTUNG

**bs2-00-synchronisation.pdf**
Folie 15

⇒ funktioniert auch nicht, aus selben Grund wie Versuch 2!

### 1.6.4 VERSUCH 4: MEHRERE VARIABLEN ZEIGEN WUNSCH ZUM EINTRETEN IN KRITISCHEN ABSCHNITT AN

**bs2-00-synchronisation.pdf**
Folie 16

**BEWERTUNG**

**bs2-00-synchronisation.pdf**
Folie 17

⇒ funktioniert nicht vernünftig, da Deadlock-Risiko.

### 1.6.5 VERSUCH 5: MEHRERE VARIABLEN MIT EINTRITTSWUNSCH UND WEITERER PRÜFUNG

**bs2-00-synchronisation.pdf**
Folie 18

**BEWERTUNG**

**bs2-00-synchronisation.pdf**
Folie 19

⇒ durch zufällige Verzögerung ist diese Lösung schon relativ gut. Statistisch kann trotzdem (bei Mehrprozessorsystemen) noch ein Livelock eintreten.

### 1.6.6 ALGORITHMUS VON PETERSON

Vergleich auch: Algorithmus von Dekker und Dijkstra.

**bs2-00-synchronisation.pdf**
Folie 20

**BEWERTUNG**

**bs2-00-synchronisation.pdf**
Folie 21

Anmerkung: Bei Implementation kann es passieren, dass es nicht funktioniert → Der Algorithmus geht davon aus, dass Wertänderungen (bspw. von `turn`) sofort von dem anderen Prozess gesehen wird. Tatsächlich passiert das (bei typischen Multiprozessorsystemen) erst zeitverzögert. Dafür müssten Schreibbarrieren eingeführt werden.

### 1.6.7 FAZIT

**bs2-00-synchronisation.pdf**
Folie 22

„busy waiting“: Prozess verbraucht noch immer Ressourcen (durch `while`-Schleife), obwohl er nur warten soll.

## 1.7 ZENTRALE LÖSUNG: SEMAPHORE

#### MOTIVATION

**bs2-00-synchronisation.pdf**
Folie 23

### 1.7.1 ZUSTÄNDE EINES SEMAPHORS

**bs2-00-synchronisation.pdf**
Folie 24

### 1.7.2 OPERATIONEN EINES SEMAPHORS

**bs2-00-synchronisation.pdf**
Folie 25

**bs2-00-synchronisation.pdf**
Folie 26

#### IMPLEMENTIERUNG DER P()-FUNKTION

**bs2-00-synchronisation.pdf**
Folie 27

### 1.7.3 API FÜR SEMAPHORE

#### UNTER UNIX

**bs2-00-synchronisation.pdf**
Folie 28

#### WEITERE

**bs2-00-synchronisation.pdf**
Folie 29



### 1.7.4 ANWENDUNG: ZEITLICHE SYNCHRONISATION VON PROZESSEN

**bs2-00-synchronisation.pdf**
Folie 31

#### RENDEZVOUS

**bs2-00-synchronisation.pdf**
Folie 34

Das Vertauschen würde zu einem Deadlock führen.

### 1.7.5 WEITERE ASPEKTE

**bs2-00-synchronisation.pdf**
Folie 35

Bekannte anschauliche Probleme:

- Leser-Schreiber-Problem (gleichzeitiges Lesen, nur einer darf immer schreiben)
- Erzeuger-Verbraucher-Problem (beschränkter Puffer)
- Problem des schlafenden Friseurs (verschieden exklusive Ressourcen)-
- Problem der dinierenden Philosophen (Verteilung von begrenzten Ressourcen ohne Verhungern/Deadlock)

#### 1.7.5.1 PROBLEM DER DINIERENDEN PHILOSOPHEN

**bs2-00-synchronisation.pdf**
Folie 36

#### NAIVE LÖSUNG

```
/* Code für den n-ten Philosophen, naive Loesung */
void philosopher(int n){
  while(1) {
    /* denken */
    take_fork(n);              /* rechte Gabel nehmen */
    take_fork((n+1)%5);        /* linke Gabel nehmen */
    /* essen */
    put_fork((n+1)%5);         /* linke Gabel weglegen */
    put_fork(n);               /* rechte Gabel weglegen */
  }
}
```

Problem: jeder greift nach seiner rechten Gabel → keiner kann mehr nach der linken greifen (Deadlock)!

### VORSICHTIGES GREIFEN NACH 2. GABEL

```
/* Code für den n-ten Philosophen, 'vorsichtiges' Greifen nach 2. Gabel */
void philosopher(int n){
  while(1) {
    /* denken */
  again:
    take_fork(n);              /* rechte Gabel nehmen */
    if (!available(fork((n+1)%5)) { /* falls linke Gabel nicht verfuegbar ... */
      put_fork(n);             /* ... rechte Gabel zuruecklegen ... */
      sleep(10);               /* ... eine Weile warten ... */
      goto again;              /* ... und von vorn probieren. */
    }
    take_fork((n+1)%5);        /* linke Gabel nehmen */
    /* essen */
    put_fork((n+1)%5);         /* linke Gabel weglegen */
    put_fork(n);               /* rechte Gabel weglegen */
  }
}
```

Problem: Kann zu einem Livelock führen (besser: sleep-Zeit zufällig auswählen).

### 1 SEMAPHOR

```
/* Code für den n-ten Philosophen, 1 Semaphor */
semaphore eat = 1;         /* Init: offen */
void philosopher(int n){
  while(1) {
    /* denken */
    P(eat);                    /* Erlaubnis zum Aufnehmen der Gabeln einholen */
    take_fork(n);              /* rechte Gabel nehmen */
    take_fork((n+1)%5);        /* linke Gabel nehmen */
    /* essen */
    put_fork((n+1)%5);         /* linke Gabel weglegen */
    put_fork(n);               /* rechte Gabel weglegen */
    V(eat);                    /* Erlaubnis zurückgeben */
  }
}
```

Problem: Maximale Anzahl an essenden Philosophen wird auf 1 reduziert!

### LÖSUNG NACH TANENBAUM

```
/* Loesung des Philosophenproblems nach Tanenbaum */
#define N 5
#define RIGHT(i) (((i)+1) % N)
#define LEFT(i) (((i)==N) ? 0: (i)+1)
```



```
5
6  enum phil_state {THINKING, HUNGRY, EATING};
7  enum phil_state state[N];      /* geeignet initialisiert */
8
9  semaphore mutex = 1;
10 semaphore s[N];               /* müssen geschlossen initialisiert
       werden */
11
12 void test(int i){
13   if ((state[i] == HUNGRY) &&
14       (state[LEFT(i)] != EATING) &&
15       (state[RIGHT(i)] != EATING)) {
16     state[i] = EATING;
17     V(s[i]);
18   }
19 }
20
21 void get_forks(int i){
22   P(mutex);
23   state[i] = HUNGRY;
24   test(i);
25   V(mutex);
26   P(s[i]);                  /* hungrig blockieren oder verlassen, d.h
         ., essen */
27 }
28
29 void put_forks(int i){
30   P(mutex);
31   state[i] = THINKING;
32   test(LEFT(i));            /* linken Nachbarn ggf. wecken */
33   test(RIGHT(i));           /* rechten Nachbarn ggf. wecken */
34 }
35
36 void philosopher(int n){
37   while(1) {
38     /* denken */
39     get_forks(n);
40     /* essen */
41     put_forks(n);
42   }
43 }
```

Achtung: Programme mit vielen Semaphoren sind unübersichtlich!

### EINFACHE PRAGMATISCHE LÖSUNG

```
1  /* alternative Loesung des Philosophenproblems */
2  #define N 5
3  #define RIGHT(i) (((i)+1) % N)
4  #define LEFT(i) (((i)==N) ? 0: (i)+1)
5
```



```
6  semaphore free = N-1;                    /* nicht-binaerer Semaphor
       */
7  semaphore fork[N] = { 1, 1, 1, 1, 1};    /* offen initialisiert */
8
9  void philosopher(int n){
10   while(1) {
11     /* denken */
12     P(free);
13     P(fork[RIGHT(n)]);
14     P(fork[LEFT(n)]);
15     /* essen */
16     V(fork[LEFT(n)]);
17     V(fork[RIGHT(n)]);
18     V(free);
19   }
20 }
```

## 1.8 FAZIT

**bs2-00-synchronisation.pdf**
Folie 38

## 1.9 ZUSAMMENFASSUNG SEMAPHORE

Vorlesung
05.04.2017

**bs2-02-sync.pdf**
Folie 2

**bs2-02-sync.pdf**
Folie 3

## 1.10 IMPLEMENTATION DER P()-OPERATION

**bs2-02-sync.pdf**
Folie 4

**bs2-02-sync.pdf**
Folie 5

## 1.11 SPINLOCKS (SCHLOSSVARIABLEN)

**bs2-02-sync.pdf**
Folie 6

Spinlock → busy waiting

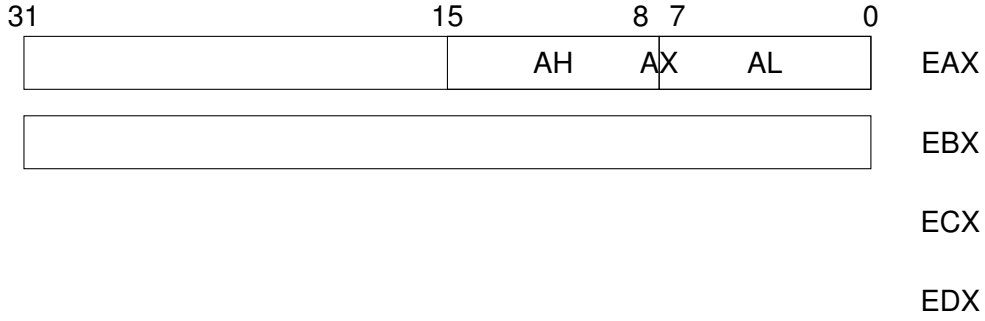


### 1.11.1 IMPLEMENTIERUNG ANSATZ 1

Möglicher Syntax bei x86 (Unterschiede):

- Intel-Syntax:
  `instruction target, source`
  `mov` (allgemein)
- AT&T Syntax
  `instruction source, target`
  `movb, movw, movl, movq` (je nach Länge des Operants: Byte, Word, ...)

Register: A, B, C, D




- ESI: Source Index
- EDI: Destination Index
- ESP: Stack Pointer (unteres Ende)
- EBP: Base Pointer (oberes Ende)

**bs2-02-sync.pdf**
Folie 7

Ablauf:

- globale Funktionen bestimmen
- definiere locked=0 und unlocked=1
- data-Sektion: setze lock=unlocked(1)
- text(Instruktions)-Sektion:
  - kopiere lock in EAX
  - vergleiche, ob EAX gleich mit locked(0)
  - wenn ja, dann gehe erneut zu enter_cs
  - sonst kopiere locked(0) in lock

**bs2-02-sync.pdf**
Folie 12



### 1.11.2 ATOMARE LESE-SCHREIB-INSTRUKTIONEN DER IA32 (INTEL 32BIT ARCHITEKTUR)

**bs2-02-sync.pdf**
Folie 13

### 1.11.3 IMPLEMENTIERUNG ANSATZ 2

**bs2-02-sync.pdf**
Folie 14

Atomarer Exchange-Befehl wird genutzt, um sicher zu stellen, dass nur 1 Prozess in kritischen Abschnitt kann:

| | | |
|---|---|---|
| EAX: | 0 | var |
| Lock: | var | 0 |

Also zu:

| | | |
|---|---|---|
| EAX: | 0 | 0 |
| Lock: | 0 | 0 |

und offen:

| | | |
|---|---|---|
| EAX: | 0 | 1 |
| Lock: | 1 | 0 |

**bs2-02-sync.pdf**
Folie 15

### 1.11.4 WARUM NICHT EINFACH INTERRUPTS VERBIETEN?

**bs2-02-sync.pdf**
Folie 16

## 1.12 API FÜR SEMAPHORE

#### UNTER UNIX

**bs2-02-sync.pdf**
Folie 17

#### WEITERE

**bs2-02-sync.pdf**
Folie 18

### 1.12.1 SEMAPHOR-API NACH SYSTEM V

**bs2-02-sync.pdf**
Folie 19

- `ipcs`: Anzeige der Semaphore (und Shared Memory Segments und Message Queues)
- `ipcrm`: löscht IPC-Ressource (bspw. Semaphor)



#### ERZEUGER-VERBRAUCHER-PROBLEM

Veranschaulichender Quellcode: `erzver-sysv.c`

#### WEITER ASPEKTE

**bs2-02-sync.pdf**
Folie 22

## 1.13 IDENTIFIKATION DER SEMAPHORE

**bs2-02-sync.pdf**
Folie 20

Funktion `ftok` kann einen eindeutigen Key für den Semaphor generieren.

## 1.14 SEMAPHORAUSGLEICHSWERT

**bs2-02-sync.pdf**
Folie 21

Semaphorausgleichswert ist dafür da, dass beim Beenden eines Prozesses die Manipulation des Semaphores zurück gesetzt wird (wenn Prozess nach dem schließen des Semaphors beendet ohne ihn zu öffnen, schafft der Semaphorausgleichswert dafür, dass der kritische Abschnitt noch von anderen Programmen erreicht werden kann).
Der Semaphorausgleichswert zählt im Prinzip die Anzahl von `P()`s und `V()`s.

## 1.15 DAS LESER-SCHREIBER-PROBLEM

Vorlesung
12.04.2017/1

**bs2-02-sync.pdf**
Folie 23

### 1.15.1 TRIVIALLÖSUNG

**bs2-02-sync.pdf**
Folie 24

### 1.15.2 STANDARDLÖSUNG

**bs2-02-sync.pdf**
Folie 26

Im Pseudocode:

```
// Schreiben:
P(wri)
/* schreibt */
V(wri)

// Lesen:
P(mutex)  // verhindert, dass mehre gleichzeitig Abfrage des P(wri)
    machen.
rc++  // zählt Leseprozesse
if (rc==1)
  P(wri)
V(mutex)
/* liest */
P(mutex)  // wie P(mutex) oben
rc--
if (rc==0)
  V(wri)
V(mutex)
```

- ✓ exklusives Schreiben
- ✓ parallel Lesen möglich
- ✓ kein Lesen, wenn geschrieben wird
- × Lösung bevorzugt (ggf.) Leseprozesse

### 1.15.3 ANALOGE BEVORZUGT SCHREIBER

**bs2-02-sync.pdf**
Folie 27

### 1.15.4 UMSETZUNG MIT PEARL UND BOLT-VARIABLEN

Bolt-Variablen in RTOS-UH und der EZ-Programmiersprache PEARL

- Boltvariable
- Leser-Schreibe-SEMAPHORE

```
,,P‘‘(n,Lesen)
,,V‘‘(n,Lesen)
,,P‘‘(n,Schreiben)
,,V‘‘(n,Schreiben)
```

**bs2-02-sync.pdf**
Folie 28

#### 1.15.4.1 ZUSTANDSDIAGRAMM EINER BOLTVARIABLE

**bs2-02-sync.pdf**
Folie 29

#### 1.15.4.2 LESER-SCHREIBER-PROBLEM IN PEARL

**bs2-02-sync.pdf**
Folie 30

### 1.15.5 DATENSTRUKTUREN IM LINUX-KERN

**bs2-02-sync.pdf**
Folie 31

**bs2-02-sync.pdf**
Folie 32

### 1.15.6 LESER-SCHREIBER-SPINLOCKS

**bs2-02-sync.pdf**
Folie 33

Erläuterung:

```
1:  rwlp++
    jump 3(forward), if sign bit not set
    rwlp--  (weil Schreiber anwesend)
2:  wenn rwlp noch nicht 0, dann
    jump 2(backward). Sonst:
    jumb 1(backward)  (vor vorn probieren)
3:
```

Vorlesung
12.04.2017/2

**bs2-02-sync.pdf**
Folie 34

Erläuterung:

```
1:  btsl (bit test and set: aktueller Wert des referenzierten Bits
    geht ins Carry Flag(C), danach Bit auf 1 setzen) => Bit 31 von
    rwlp ins Carry-Flag C
    wenn C==1, Sprung nach 2 (wenn also schon Schreiber da war)
    testl: Operanden mit logischem UND verknüpfen (und prüfen, ob
       null ist)
    je 3(forward) wenn testl 0 ergibt, also die Anzahl Leser 0 ist.
        Sonst:
    btrl (bit test and reset: wie btsl, nur dass Bit auf 0 gesetzt
       wird) => Bit 31 von rwlp auf 0 setzen
```

```
2:  wenn rwlp noch nicht 0, dann
    jump 2(backward). Sonst:
    jump 1(backward)  (von vorn probieren)
3:
```

### 1.15.7 ZWISCHENFAZIT

**bs2-02-sync.pdf**
Folie 35

## 1.16 MONITORE

### 1.16.1 IDEE

**bs2-02-sync.pdf**
Folie 36

### 1.16.2 BEDINGUNGSVARIABLEN

**bs2-02-sync.pdf**
Folie 37

Bedingungsvariable = Prozesswarteschlange
delay $\rightarrow$ Prozess wird in (betreffende) Warteschlange eingeordnet
resume $\rightarrow$ (anderer) Prozess wird aus betreffender Warteschlange entfernt und fortgesetzt

**bs2-02-sync.pdf**
Folie 38

#### 1.16.2.1 SEMANTIK VON RESUME()

**bs2-02-sync.pdf**
Folie 39

**bs2-02-sync.pdf**
Folie 40

#### 1.16.2.2 BEDINGUNGSVARIABLEN IN PTHREADS

**bs2-02-sync.pdf**
Folie 41

### 1.16.3 STRUKTUR

**bs2-02-sync.pdf**
Folie 43

**bs2-02-sync.pdf**
Folie 42

### 1.16.4 ANWENDUNGEN

**bs2-02-sync.pdf**
Folie 44

zu 1.: Von `monitor mutex;` bis `end; (* monitor mutex *)` sind die proceduren Monitor-Prozeduren. Daher kann bspw. die Prozedur `enter` nur von einem Prozess auf einmal ausgeführt werden.

**bs2-02-sync.pdf**
Folie 45

### 1.16.5 BEWERTUNG

**bs2-02-sync.pdf**
Folie 46

## 1.17 PRÜFUNGSFRAGEN

- Semaphore Erklären (P/V)
- P/V programmieren können
- Abläufe erkennen (wer kommt zuerst dran)
- Monitor als Theorie (nicht programmiertechnisch)

# 2 INTER PROCESS COMMUNICATION (IPC)

Vorlesung
26.04.2017

Mechanismen:

- Message Passing
- Shared Memory Segment
- Datei
- Signale
- Pipe (unbenannt)
  - unidirektional
  - nur zwischen verwandten Prozessen möglich
- Pipe (benannt)
- Socket

## 2.1 MESSAGE PASSING

**bs2-03-ipc.pdf**
Folie 2

### 2.1.1 SYNCHRONISATION MIT NACHRICHTEN

**bs2-03-ipc.pdf**
Folie 3

### 2.1.2 ADRESSIERUNG

**bs2-03-ipc.pdf**
Folie 4

### 2.1.3 PRAXISBEISPIELE

#### 2.1.3.1 MESSAGE PASSING INTERFACE (MPI)

**bs2-03-ipc.pdf**
Folie 5

#### BEISPIEL 2 KNOTEN, FORTRAN

**bs2-03-ipc.pdf**
Folie 6

**bs2-03-ipc.pdf**
Folie 7

#### 2.1.3.2 MIKROKERN L4

**bs2-03-ipc.pdf**
Folie 8

#### 2.1.3.3 PASCAL-FC

**bs2-03-ipc.pdf**
Folie 9

**bs2-03-ipc.pdf**
Folie 10

#### 2.1.3.4 NACHRICHTENWARTESCHLANGEN IN DER SYSTEM-V-IPC

System-V-IPC:

- Semaphore
- Message Queues
- Shared-Memory-Signale

**bs2-03-ipc.pdf**
Folie 11

Details: `msgserver.c` und `msgclient.c`.

## 2.2 BENANNTE PIPES

- „sieht aus“ wie eine Datei, d.h. hat einen Eintrag in Dateisystem
- syscall: `mkfifo()` – legt FIFO (benutzt Pipe) in Dateisystem an
- zugehöriges Unix-Kommando: `mkfifo`
- Kommandos: `open()`, `read()`, `write()`, `close()` (kein `lseek`!)
- Löschen mittels `rmlink()` bzw `rm`-Kommando

## 2.3 SHARED MEMORY



**bs2-03-ipc.pdf**
Folie 12

### 2.3.1 SYSTEMRUFE IN DER SYSTEM-V-API

**bs2-03-ipc.pdf**
Folie 13

**bs2-03-ipc.pdf**
Folie 14

### 2.3.2 POSIX-API

**bs2-03-ipc.pdf**
Folie 15



# 3 HAUPTSPEICHEVERWALTUNG

## 3.1 GRUNDLAGEN

### 3.1.1 MOTIVATION

**bs2-04-mem.pdf**
Folie 2

### 3.1.2 PROBLEME

**bs2-04-mem.pdf**
Folie 3

Prüfungsfrage: Welche Arten von Fragmentierung gibt es?

- intern: Segmentgröße „zu groß“ → kleinere Segmente belegen trotzdem ganzes Segment
- extern: Stichwort „defragmentierung“ → Segmente sind verteilt mit Lücken dazwischen

## 3.2 BITMAP

**bs2-04-mem.pdf**
Folie 4

### BLOCKUNGSFAKTOR

**bs2-04-mem.pdf**
Folie 5

## 3.3 FREISPEICHERLISTE

**bs2-04-mem.pdf**
Folie 6

Nachteil: Verteilungsaufwand (Pflege der Liste, Folgen der Pointer in der Liste usw., Finden von freien Listenelementen)!

### 3.3.1 IMPLEMENTIERUNG: BLÖCKE MIT INTEGRIERTEN HEADERN

**bs2-04-mem.pdf**
Folie 7

→ Liste wird im Speicher integriert.
Nachteil: noch ineffizienterer Zugriff (generiert viel Cache-misses/pagefaults).
Vorteil: ist fehlertoleranter (Liste kann nicht überschrieben werden, da sie im Hauptspeicher integriert ist).

### 3.3.2 SUCHSTRATEGIEN

**bs2-04-mem.pdf**
Folie 8

First/Next Fit: beste Strategien, dabei First Fit das bessere (Daumenregel: da schneller, die Fragmentierung ist zweitrangig).

**bs2-04-mem.pdf**
Folie 9

Best Fit: es hat sich gezeigt, dass es sich nicht lohnt.
Wort Fit: nur „akademischer Natur“

### 3.3.3 TECHNIKEN ZUR EFFIZIENZSTEIGERUNG

**bs2-04-mem.pdf**
Folie 10

**bs2-04-mem.pdf**
Folie 11

### 3.3.4 GETRENNTE FREISPEICHERLISTE (SEGREGATED FITS)

**bs2-04-mem.pdf**
Folie 12

**bs2-04-mem.pdf**
Folie 13

Vereinigung benachbarter freier Segmente kostet wieder Geschwindigkeit!



### 3.3.5 BUDDY-VERFAHREN

**bs2-04-mem.pdf**
Folie 14

Teilung passiert rekursiv:
Wenn nur ein 1MiB Block frei ist und 78KiB gefordert sind, wird des 1MiB zerstückelt:

- 2*512KiB (davon eins wieder zerstückelt)
- 2*256KiB (davon eins wieder zerstückelt)
- 2*128KiB (davon wird eins ausgeliefert).

Resultiert in freien Blöcken:
1*512KiB, 1*256KiB, 1*128KiB

#### BEURTEILUNG

**bs2-04-mem.pdf**
Folie 15

## 3.4 VIRTUELLER SPEICHER

Vorlesung
10.05.2017

### 3.4.1 MOTIVATION

**bs2-04-mem.pdf**
Folie 16

### 3.4.2 SEITEN VS KACHELN

**bs2-04-mem.pdf**
Folie 17

**bs2-04-mem.pdf**
Folie 18

Kacheln/Seitenrahmen: Segmente im physischen Speicher
(virtuelle/logische)Seiten: Segmente im virtuellen Speicher
MMU+OS: Memory Management Unit (im Prozessor) + Betriebssystem (Prozessor muss es unterstützen und Betriebssystem muss es nutzen)

### 3.4.3 GESTREUTE SPEICHERUNG

Umsetzung virtueller in physische Adresse

**bs2-04-mem.pdf**
Folie 19

Seitentabelle existiert für jeden Prozess.



### 3.4.4 SEITENTABELLENEINTRAG/PAGE TABLE ENTRY (PTE)

**bs2-04-mem.pdf**
Folie 20

#### 3.4.4.1 GRÖSSE DER SEITENTABELLE

**bs2-04-mem.pdf**
Folie 21

**bs2-04-mem.pdf**
Folie 22

Zugriffe: Immer eins mehr als die Hierarchiestufe.

#### 3.4.4.2 BEISPIEL: ZWEISTUFIGE SEITENTABELLE I386

**bs2-04-mem.pdf**
Folie 23

### 3.4.5 DEMAND PAGING

**bs2-04-mem.pdf**
Folie 24

### 3.4.6 SEITENFEHLER/PAGEFAULT

**bs2-04-mem.pdf**
Folie 25

### 3.4.7 EINLAGERUNGSTRATEGIEN

**bs2-04-mem.pdf**
Folie 26

### 3.4.8 SEITENAUSTAUSCHVERFAHREN

**bs2-04-mem.pdf**
Folie 27

#### 3.4.8.1 OPTIMALES VERFAHREN, LRU

**bs2-04-mem.pdf**
Folie 28



#### 3.4.8.2 NOT RECENTLY USED (NRU)

**bs2-04-mem.pdf**
Folie 29

**bs2-04-mem.pdf**
Folie 30

#### 3.4.8.3 FIFO, SECOND CHANCE

**bs2-04-mem.pdf**
Folie 31

Second Chance Lazy Evaluation: Im „Uhrzeigersinn“ durch die (Ring-)Liste gehen und alle R-Bits auf 0 setzen wenn 1 war oder Eintrag löschen, falls R-Bit 0.

#### 3.4.8.4 NFU, AGING

**bs2-04-mem.pdf**
Folie 32

**bs2-04-mem.pdf**
Folie 33

(Hier würde also 3. rausfliegen)

### 3.4.9 ARBEITSMENGE (WORKING SET)

(nur geringfügig relevant für Klausur)

**bs2-04-mem.pdf**
Folie 34

**bs2-04-mem.pdf**
Folie 35

**bs2-04-mem.pdf**
Folie 36

#### 3.4.9.1 ABHÄNGIGKEIT DER GRÖSSE DER ARBEITSMENGE VON DELTA

**bs2-04-mem.pdf**
Folie 37

**bs2-04-mem.pdf**
Folie 38



### 3.4.10 IDEE FÜR ERSETZUNGSSTRATEGIE

**bs2-04-mem.pdf**
Folie 39

### 3.4.11 BELADYS ANOMALIE

**bs2-04-mem.pdf**
Folie 40

**bs2-04-mem.pdf**
Folie 41

Vorlesung 17.05.2017

Bei bestimmten Prozess gibt es bestimmte Anzahl von Seiten. Dabei gibt es eine bestimmte Anzahl von Pagefaults. Die Annahme wäre, dass die Pagefaults bei mehr Seiten sinken würde (anders herum müsste sie weiter steigen). Beladys Annomalie besagt, dass das unter bestimmten Systemkonfigurationen auch anders sein kann (dass also Pagefaults mit mehr verfügbaren Seiten steigt).

### 3.4.12 WEITERE ASPEKTE ZUR SEITENERSETZUNG

**bs2-04-mem.pdf**
Folie 42

- lokale vs globale Ersetzung: global erleichtert Entscheidung, weil Auswahl der Seiten größer ist. Lokal hat weniger Entscheidungen und verdrängt damit vielleicht eigentlich noch benötigte Seiten. Im globalen können Seiten also besser verteilt werden. Im Ernstfall können aber Prozesse verdrängt werden ($\rightarrow$ Thrashing) – das kann im lokalen nicht passieren. Im lokalen können die benötigten Seiten ggf. schneller gefunden werden.
- variable vs konstante Größe: variable zwar flexibler, aber höherer Verwaltungsaufwand.

## 3.5 SCHNITTSTELLE ZU UNIX

**bs2-04-mem.pdf**
Folie 43

### 3.5.1 SPEICHERABBILD

**bs2-04-mem.pdf**
Folie 44

### 3.5.2 SYSTEMBRUF BRK()

**bs2-04-mem.pdf**
Folie 45

### 3.5.3 STACKANFORDERNUG ALLOCA()

**bs2-04-mem.pdf**
Folie 46

Keine Rückgabefunktion: Stack ist in Rahmen eingeteilt. Wenn die Funktion betreten wird wird ein Rahmen mit der Größe für die lokalen Variablen erstellt. Wird innerhalb dieser Funktion eine weitere Funktion (mit lokalen Variablen) aufgerufen wird, wird ein weiterer Stack-Rahmen erstellt. Sobald eine Funktion beendet ist (bspw. durch return), wird dieser Stack-Rahmen wieder gelöscht.
alloca verschiebt einfach den Stackpointer $\rightarrow$ keine Rückgabe nötig. Wenn Funktion beendet wird, wird Stackframe inkl. dem mit alloca vergrößerten Bereich gelöscht (daher kein free nötig). (Praktisch wenig relevant)

### 3.5.4 AUSSCHALTUNG DES PAGINGS (PINNING)

**bs2-04-mem.pdf**
Folie 47

Ausschaltung bei sicherheitskritischen Prozessen: wird beispielsweise in Passwort in die Pagefile geschrieben, wäre er persistent unverschlüsselt gespeichert!

### 3.5.5 MEMORY-MAPPED FILES

**bs2-04-mem.pdf**
Folie 48

(„speichereingeblendete Dateien“)

#### OPEN/READ/WRITE/CLOSE

- Quell- und Zieldatei eröffnen: 2 mal `open()`
- Speicherbereich „besorgen“: `malloc()`
- Quelldatei einlesen: x mal `read()`
- Zieldatei schreiben: x mal `write()`
- Speicher zurückgeben: `free()`
- 2 mal `close()` (in der Regel werden Dateien erst jetzt tatsächlich in die Datei geschrieben)

#### MMAP

- Quell- und Zieldatei eröffnen: 2 mal `open()`
- Quelldatei abbilden: `mmap()`
- Zieldatei auf Länge Quelldatei „aufblasen“: `lseek()`, 1 mal `write()`
- Zieldatei abbilden: `mmap()`



- Kopiervorgang: `memcpy()`
- 2 mal `close()`

→ 1 mal `memcpy` deutlich schneller als x mal `read/write`!

## 3.6 ZUSAMMENFASSUNG

**bs2-04-mem.pdf**
Folie 49



# 4 DEADLOCKS

Vorlesung
24.05.2017

## 4.1 GRUNDLAGEN

### 4.1.1 MOTIVATION

#### BEISPIELE FÜR DEADLOCKGEFÄHRDETE ABLÄUFE

**bs2-05-deadlocks.pdf**
Folie 2

(wenn beide Prozesse ihren ersten Semaphor sperren, bleiben sie beide beim sperren des zweiten stehen und schlafen unendlich

### 4.1.2 DEADLOCK VS LIVELOCK

**bs2-05-deadlocks.pdf**
Folie 3

## 4.2 EINTRITTSBEDINGUNGEN

**bs2-05-deadlocks.pdf**
Folie 4

($\rightarrow$ potentielle Prüfungsfrage)

### 4.2.1 DYNAMIK DES DEADLOCKEINTRITTS

#### BEISPIEL: 3 PROZESSE KONKURRIEREN UM 3 RESSOURCEN R, S, T

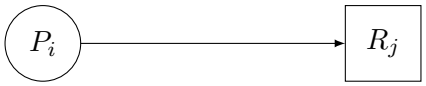
**bs2-05-deadlocks.pdf**
Folie 5

#### LEHREN AUS DIESEM BEISPIEL:

**bs2-05-deadlocks.pdf**
Folie 6

### 4.2.2 RESSOURCENZUTEILUNGSGRAPH

**bs2-05-deadlocks.pdf**
Folie 7




$P_i$ fordert $R_j$


$P_l$ besitzt $R_k$

**BEISPIEL**

**bs2-05-deadlocks.pdf**
Folie 8

## 4.3 STRATEGIEN ZUR DEADLOCK-BEHANDLUNG

**bs2-05-deadlocks.pdf**
Folie 9

### 4.3.1 IGNORIEREN VON DEADLOCKS – VOGEL-STRAUSS-ALGORITHMUS

**bs2-05-deadlocks.pdf**
Folie 10

MTBF: Mean Time Before Failure

### 4.3.2 ERKENNEN UND BEHEBEN VON DEADLOCKS

#### 4.3.2.1 ERKENNUNG

**bs2-05-deadlocks.pdf**
Folie 11

#### 4.3.2.2 ALGORITHMUS ZUM ERKENNEN EINES ZYKLUS IM RZG

**bs2-05-deadlocks.pdf**
Folie 12

#### 4.3.2.3 BESCHREIBUNG MITTELS BELEGUNGS- UND ANFORDERUNGSMATRIX

**bs2-05-deadlocks.pdf**
Folie 13

$\rightarrow$ Effizienteres Verfahren zur Erkennung als über den RZG.

**BEISPIEL**

**bs2-05-deadlocks.pdf**
Folie 14

#### 4.3.2.4 ALGORITHMUS ZUR ERKENNUNG VON DEADLOCKS

**bs2-05-deadlocks.pdf**
Folie 15

**BEISPIEL**

**bs2-05-deadlocks.pdf**
Folie 16

$\underline{E}$: Spaltensumme von $\mathcal{R}$.
Bei $\underline{A}$ wird von $\underline{E}$ subtrahiert: Die Spaltensummen von $\mathcal{C}$!

Prozesse die vollständig ablaufen können, geben ihre Ressourcen wieder zurück. Prüfe, ob alle Prozesse ablaufen können, nachdem fertige Prozesse ihre Ressourcen jeweils freigegeben haben!

### 4.3.3 ZEITPUNKT DER ERKENNUNG / BEHEBUNG EINES DEADLOCKS

**bs2-05-deadlocks.pdf**
Folie 17

Entzug einer Ressource Beispiel: Speicherblock auf Festplatte zwischenspeichern, Prozess anhalten, Speicher anderem Prozess geben und wenn der fertig ist, Speicherblock von Festplatte wieder zurück schreiben.
Rollback mit Checkpointing: Wie ein Auto-/Quicksave im FPS

### 4.3.4 DYNAMISCHES VERMEIDEN

Vorlesung
31.05.2017

**bs2-05-deadlocks.pdf**
Folie 18

**BEISPIEL**

**bs2-05-deadlocks.pdf**
Folie 19

N fordert max. 5, hat schon 2, braucht also noch 3 – die sind noch frei. Daher kann N durchlaufen und mit den freigegebenen Ressourcen kann auch M durchlaufen $\rightarrow$ ist also sicher.

**bs2-05-deadlocks.pdf**
Folie 20

Das ist nicht mehr sicher, da kein Prozess beendet werden kann (beide brauchen noch 3, können aber maximal 2 bekommen).

#### 4.3.4.1 REALISIERUNGSVERFAHREN

**bs2-05-deadlocks.pdf**
Folie 21

### 4.3.5 STATISCHES VERHINDERN

**bs2-05-deadlocks.pdf**
Folie 22

**bs2-05-deadlocks.pdf**
Folie 23

## 4.4 AUSBLICK

**bs2-05-deadlocks.pdf**
Folie 24



# 5 DATEISYSTEME / MASSENSPEICHER

Vorlesung
24.05.2017

## 5.1 IMPLEMENTIERUNGEN VON DATEISYSTEMEN

### 5.1.1 KONTINUIERLICHE ALLOKATION

**bs2-06-filesystems.pdf**
Folie 3

### 5.1.2 VERKETTETE LISTE

**bs2-06-filesystems.pdf**
Folie 4

Beispiel ist nicht so schnell, da gesprungen werden muss (0015→FFFF→0014).

#### 5.1.2.1 NACHTEILE

**bs2-06-filesystems.pdf**
Folie 5

Wichtiger konzeptioneller Nachteil: Dadurch, dass in jedem Nutzdatenblock eine Verwaltungsinformation gespeichert ist, ist der Netto-Nutzdatenblock keine 2er-Potenz mehr!

#### 5.1.2.2 LISTE MIT ZUORDNUNGSTABELLE

**bs2-06-filesystems.pdf**
Folie 6

Größe bspw. FAT16: $2^{16} \cdot 32\,\mathrm{KiB}$ ($32\,\mathrm{KiB}$: Clustergröße)

### 5.1.3 INDIZIERTE SPEICHERUNG

**bs2-06-filesystems.pdf**
Folie 7

#### 5.1.3.1 SPEICHERUNG MIT VARIABLEN INDEXBLOCKS

**bs2-06-filesystems.pdf**
Folie 8

#### 5.1.3.2 INDIREKT-INDIZIERTE SPEICHERUNG

**bs2-06-filesystems.pdf**
Folie 9

**bs2-06-filesystems.pdf**
Folie 10

#### 5.1.3.3 BEISPIEL UNIX DATEISYSTEM

**bs2-06-filesystems.pdf**
Folie 13

#### DATEIADRESSIERUNG MITTELS INODES

**bs2-06-filesystems.pdf**
Folie 14

(graue Datenblöcke sind gleich groß wie die weiß gezeichneten Datenblöcke!)

**bs2-06-filesystems.pdf**
Folie 15

**Klausurfrage** Zwischen welchen Parametern wird im Unix-Dateisystem ein Kompromiss gefunden:

- kurze Zugriffszeit auf kleine Dateien
- große Dateien realisierbar

### 5.1.4 VERWEISE (LINKS)

- HARD LINKS
  Link auf Adresse wird erstellt. Wenn „Original“ gelöscht wird, bleibt Datei durch Link erhalten (Löschen ist viel mehr `unlink()`, wo geprüft wird, ob noch ein Verweis auf Adresse besteht ($\rightarrow$ Link-Counter, Info über `stat Datei`). Nur wenn kein Verweis mehr besteht, werden Daten gelöscht, sonst nur der Verweis).
  Alle Attribute (Änderungsdatum usw.) sind gleich.
  Im Prinzip wird nur ein neuer Name für die Datei erstellt.
  Es kann nicht mehr herausgefunden werden, welches das „Original“ ist: Beide Dateien/-Verweise sind gleichberechtigt.
  Funktioniert nur auf gleicher Partition.

- SOFT/SYMBOLIC LINKS
  Link auf Datei wird erstellt. Dieser Link ist eine Textdatei, die die Pfadangabe enthält.
  Funktioniert über Dateisystem-Grenzen hinweg.

### 5.1.5 JOURNALING

#### 5.1.5.1 NACHTEIL TRADITIONELLER DATEISYSTEME

**bs2-06-filesystems.pdf**
Folie 16

#### 5.1.5.2 IDEE DES JOURNALS

**bs2-06-filesystems.pdf**
Folie 17

→ nach Absturtz muss nicht (mit `fsck`) das gesamte Dateisystem überprüft werden, sondern nur das Journal erneut erstellt werden.

#### 5.1.5.3 BETRIEBSMODI

**bs2-06-filesystems.pdf**
Folie 18

#### 5.1.5.4 ZUSAMMENFASSUNG

**bs2-06-filesystems.pdf**
Folie 19

#### 5.1.5.5 ERWEITERTE ATTRIBUTE

**bs2-06-filesystems.pdf**
Folie 20

#### . . . IN LINUX-DATEISYSTEMEN

**bs2-06-filesystems.pdf**
Folie 21

#### BENÖTIGTE SYSTEMRUFE

**bs2-06-filesystems.pdf**
Folie 22

## 5.2 I/O-SCHEDULING

#### PRINZIP DER FESTPLATTE

**bs2-06-filesystems.pdf**
Folie 31



Zwei Komponenten, die Zugriffszeiten beeinflussen:

- Seek: Zeit, die der Lesekopf braucht, um auf die richtige Spur zu kommen.
- Rot: Zeit, die die Platte zur Rotation braucht. Rotationslatenz liegt zwischen $0$ (Sektor zufällig unter Lesekopf), und $\frac{1}{f}$ (Sektor gerade am Lesekopf vorbei).

Weiterhin muss noch die Zeit eingerechnet werden, die das Lesen des Sektors benötigt.

**OPTIMIERUNG VON MASSENSPEICHERZUGRIFFEN**

**bs2-06-filesystems.pdf**
Folie 23

### 5.2.1 FCFS, SSTF

#### 5.2.1.1 FIRST COME FIRST SERVE

**bs2-06-filesystems.pdf**
Folie 24

#### 5.2.1.2 SHORTEST SEEK/SERVICE TIME FIRST

**bs2-06-filesystems.pdf**
Folie 25

### 5.2.2 SCAN (ELEVATOR) UND VARIANTEN

**bs2-06-filesystems.pdf**
Folie 26

**bs2-06-filesystems.pdf**
Folie 27

#### 5.2.2.1 CIRCULAR SCAN (C-SCAN) UND FSCAN

**bs2-06-filesystems.pdf**
Folie 28

**bs2-06-filesystems.pdf**
Folie 29

### 5.2.3 SHORTEST ACCESS TIME FIRST (SATF)

**bs2-06-filesystems.pdf**
Folie 30

**ÜBERLAPPUNG VON ROTATION UND SEEK**

**bs2-06-filesystems.pdf**
Folie 31

**BERECHNUNG DER ACCESS TIME**

**bs2-06-filesystems.pdf**
Folie 32

#### 5.2.3.1 TECHNIKEN ZUM VERHINDERN DES AUSHUNGERNS

**BATCH-ALGORITHMEN**

**bs2-06-filesystems.pdf**
Folie 33

**ERZWUNGENES AUSFÜHREN DES ÄLTESTEN AUFTRAGS**

**bs2-06-filesystems.pdf**
Folie 34

### 5.2.4 VERFAHREN IM LINUX-KERNEL

#### 5.2.4.1 GRUNDLAGEN

**bs2-06-filesystems.pdf**
Folie 35

#### 5.2.4.2 „WRITES-STARVING-READS“

**bs2-06-filesystems.pdf**
Folie 36

**BEISPIEL**

**bs2-06-filesystems.pdf**
Folie 37

#### 5.2.4.3 LINUS ELEVATOR (KERNEL 2.4)

**bs2-06-filesystems.pdf**
Folie 38

2.+3. Schritt wäre dann wieder FIFO, was wieder langsam wird!



#### 5.2.4.4 DEADLINE I/O SCHEDULER

**bs2-06-filesystems.pdf**
Folie 39

#### 5.2.4.5 ANTICIPATORY SCHEDULER

**bs2-06-filesystems.pdf**
Folie 40

Ist mittlerweile nicht mehr im Kernel.

#### 5.2.4.6 CFQ UND NOOP

**bs2-06-filesystems.pdf**
Folie 41

#### 5.2.4.7 PRAXIS

**bs2-06-filesystems.pdf**
Folie 42

## 5.3 ZUSAMMENFASSUNG

**bs2-06-filesystems.pdf**
Folie 43



# 6 SICHERHEIT

Vorlesung
28.06.2017

## 6.1 GRUNDBEGRIFF

### 6.1.1 ZIELE DER SYSTEMSICHERHEIT

**bs2-07-security.pdf**
Folie 3

### 6.1.2 BEDROHUNGEN

**bs2-07-security.pdf**
Folie 4

## 6.2 BÖSARTIGE SOFTWARE

### 6.2.1 ÜBERBLICK

**bs2-07-security.pdf**
Folie 5

- lokaler Angriff (insider Angriff): Nutzer hat bereits Zugang zu System und versucht bspw. `root` zu werden.
- entfernter Angriff: Zugriff von außen (kein Account auf angegriffenen System vorhanden).
- on-line-Angriff: Hacker ist verbunden ist mit angegriffenen System und „hackt“ (selten) [auch lokal möglich].
- off-line-Angriff: bspw. durchforsten von Passwort-Dateien (die erklaut wurden).

### 6.2.2 LOGISCHE BOMBEN

**bs2-07-security.pdf**
Folie 6

#### AUSSCHNITT MCAFEE AKTIVIERUNGSKALENDER

**bs2-07-security.pdf**
Folie 7

### 6.2.3 HINTERTÜREN (BACKDOORS)

**bs2-07-security.pdf**
Folie 8

#### BEISPIEL 1

**bs2-07-security.pdf**
Folie 9

#### BEISPIEL 2

**bs2-07-security.pdf**
Folie 10

Bei `current->uid = 0` wird nicht verglichen, sondern `uid` gesetzt! Wenn also die angegeben Option-Flags gesetzte werden, wird der ausführende Nutzer `root`.

**bs2-07-security.pdf**
Folie 11

### 6.2.4 TROJANISCHES PFERD

**bs2-07-security.pdf**
Folie 12

#### BEISPIEL UNIX

**bs2-07-security.pdf**
Folie 17

`u+s`: Setzen des Sticky-Bits (Programm läuft unter rechten des Eigentümers der Datei, nicht unter den rechten des Ausführenden)

### 6.2.5 (COMPUTER-)VIREN

**bs2-07-security.pdf**
Folie 18

#### EINFACHES BEISPIEL

**bs2-07-security.pdf**
Folie 23



#### ERWEITERTES BEISPIEL

**bs2-07-security.pdf**
Folie 24

→ keine Mehrfachinfektion mehr

### 6.2.6 WÜRMER

**bs2-07-security.pdf**
Folie 25

#### KOMPONENTEN EINES WURMS

**bs2-07-security.pdf**
Folie 26

**bs2-07-security.pdf**
Folie 27

Vokabeln:

- Vulnerability: Schwachstelle
- Exploit: Ausnutzen einer Schwachstelle

### 6.2.7 ROOTKITS

**bs2-07-security.pdf**
Folie 28

Hinweis: Rootkit ist nicht dafür da, root zu werden, sondern das Schadprogramm zu verbergen.

#### ARTEN VON ROOTKITS

**bs2-07-security.pdf**
Folie 29

#### GEGENMASSNAHMEN

**bs2-07-security.pdf**
Folie 30

## 6.3 AUTHENTIFIZIERUNGSMECHANISMEN

Vorlesung
05.07.2017

**bs2-07-security.pdf**
Folie 31

Athentifizierung: Identifikation von einem Nutzer durch einen Host-Rechner.

### 6.3.1 ANGRIFFE AUF DEN AUTHORISIERUNGSVORGANG

**bs2-07-security.pdf**
Folie 32

#### 6.3.1.1 ERRATEN DES PASSWORTS

**bs2-07-security.pdf**
Folie 33

**BEISPIEL /VAR/LOG/AUTH.LOG**

**bs2-07-security.pdf**
Folie 34

#### 6.3.1.2 WÖRTERBUCHANGRIFF

**bs2-07-security.pdf**
Folie 35

#### 6.3.1.3 ERSCHWEREN DES WÖRTERBUCHANGRIFFS MITTELS SALZ

**bs2-07-security.pdf**
Folie 36

**BEISPIEL: BIBLIOTHEKSFUNKTION CRYPT()**

**bs2-07-security.pdf**
Folie 37

**WEITERE GEGENMASSNAHMEN**

**bs2-07-security.pdf**
Folie 38

### 6.3.2 CHALLENGE-RESPONSE ZUR AUTHENTIFIZIERUNG

**bs2-07-security.pdf**
Folie 39

### 6.3.3 BEISPIEL: AUTHENTIFIZIERUNG IN WINDOWS

**bs2-07-security.pdf**
Folie 40

### 6.3.4 SICHERHEIT VON NTLM

**bs2-07-security.pdf**
Folie 41

### 6.3.5 AUTHENTIFIZIERUNG MIT PHYSISCHEN OBJEKTEN

**bs2-07-security.pdf**
Folie 42

## 6.4 ANGRIFFSTECHNIKEN

- Heap Overflow (nicht regulär auszunutzen)
- Integer Overflow (nicht regulär auszunutzen)
- Stack Overflow: spezielle Form eines Buffer Overflows
- Return-into-Libc-Exploit: Libc (return-)Aufruf umleiten
- Return-Oriented-Programming (ROP)
- Format-String-Attacke: Versuche `printf` oder vergleichbares auszunutzen

### 6.4.1 BUFFER OVERFLOW

**bs2-07-security.pdf**
Folie 44

Sehr einfaches verwundbares Programm:
`strcopy` interessiert sich nicht für Buffergröße $\rightarrow$ gesamtes `argv[1]` wird in Speicher geschrieben. Dadurch kann die Rücksprungadresse überschrieben werden. Damit kann Schadcode ausgeführt werden (das wird von modernen Betriebssystemen allerdings verhindert):

#### 6.4.1.1 PRINZIP

**bs2-07-security.pdf**
Folie 45



#### 6.4.1.2 AUSSCHNITT DES STACKS

**bs2-07-security.pdf**
Folie 46

SFP: Saved Frame Pointer
<Ret>: Rückkehradresse

**bs2-07-security.pdf**
Folie 49

**bs2-07-security.pdf**
Folie 50

Return kehrt zu eigenem Code im Puffer zurück.

#### 6.4.1.3 EINFACHE GEGENMASSNAHMEN

**bs2-07-security.pdf**
Folie 51

#### 6.4.1.4 STACKGUARD

**bs2-07-security.pdf**
Folie 52

#### WAHL DES CANARY WORDS

**bs2-07-security.pdf**
Folie 53

Canary word verhindert, dass es vom Angreifer eingegeben und kopiert werden kann – wenn doch, dann würde beim Terminator Canary die Eingabe abbrechen.

#### GRENZEN DES KONZEPTS

**bs2-07-security.pdf**
Folie 54

#### 6.4.1.5 STACKSHIELD

**bs2-07-security.pdf**
Folie 55

#### 6.4.1.6 AUSFÜHRUNGSVERBOTE BESCHREIBBARER SEITEN ($W \oplus X$)

**bs2-07-security.pdf**
Folie 56

**bs2-07-security.pdf**
Folie 58

**bs2-07-security.pdf**
Folie 59

- Text: +X+R -W
- Data: +R+W -X
- Stack: +R+W -X

**ADRESSUMSETZUNG PAE (NX)**

**bs2-07-security.pdf**
Folie 57

#### 6.4.1.7 ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR)

**bs2-07-security.pdf**
Folie 60

Weiterhin negativ: tiefer Eingriff ins System; einige Programme müssen ohne ASLR laufen, weil sie sonst nicht mehr funktionieren (bleiben angreifbar)

### 6.4.2 RETURN-INTO-LIBC

**bs2-07-security.pdf**
Folie 61

**STACKLAYOUT**

**bs2-07-security.pdf**
Folie 62

**VERKETTUNG ZWEIER LIBC-AUFRUFE**

**bs2-07-security.pdf**
Folie 66

**bs2-07-security.pdf**
Folie 67



**ANMERKUNGEN**

**bs2-07-security.pdf**
Folie 63

#### 6.4.2.1 BESTIMMUNG DER EINSPRUNGSADRESSE (STATISCH)

**bs2-07-security.pdf**
Folie 64

#### 6.4.2.2 DYNAMISCHE BESTIMMUNG DER EINSPRUNGADRESSE

**bs2-07-security.pdf**
Folie 65

#### 6.4.2.3 WEITERE TECHNIKEN

**bs2-07-security.pdf**
Folie 68

### 6.4.3 FORMAT STRING EXPLOIT

#### 6.4.3.1 FUNKTIONALITÄT VON PRINTF()

**bs2-07-security.pdf**
Folie 70

**BEISPIEL**

**bs2-07-security.pdf**
Folie 71

#### 6.4.3.2 EXPLIZITE ADRESSIERUNG VON ARGUMENTEN

**bs2-07-security.pdf**
Folie 72

**STACKLAYOUT**

**bs2-07-security.pdf**
Folie 73

#### 6.4.3.3 BEISPIEL FÜR VERWUNDBARE FUNKTION

**bs2-07-security.pdf**
Folie 74

**bs2-07-security.pdf**
Folie 75

### 6.4.4 HEAP-OVERFLOW

**bs2-07-security.pdf**
Folie 91

Beispiel:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
  char *buf;
  char* num;

  buf = malloc(20);
  num = malloc(sizeof(int));

  *num = 42;
  printf("num = %d\n", *num);

  gets(buf);
  printf("num = %d\n", *num);
}
```

Wenn Eingabe zu Lang ist: Num wird überschrieben.

### 6.4.5 INTEGER-OVERFLOW

**bs2-07-security.pdf**
Folie 92

#### BEISPIEL

**bs2-07-security.pdf**
Folie 93

## 6.5 ANGRIFFSCODE (SHELL)