

# Betriebssystemtechnik

*Adressräume: Trennung, Zugriff, Schutz*

## II. Systemaufruf

Wolfgang Schröder-Preikschat

3. Mai 2017



## Rekapitulation

- Mehrebenenmaschinen

- Teilinterpretierung

## Funktionale Hierarchie

- Analogie

- Abstraktion

## Implementierung

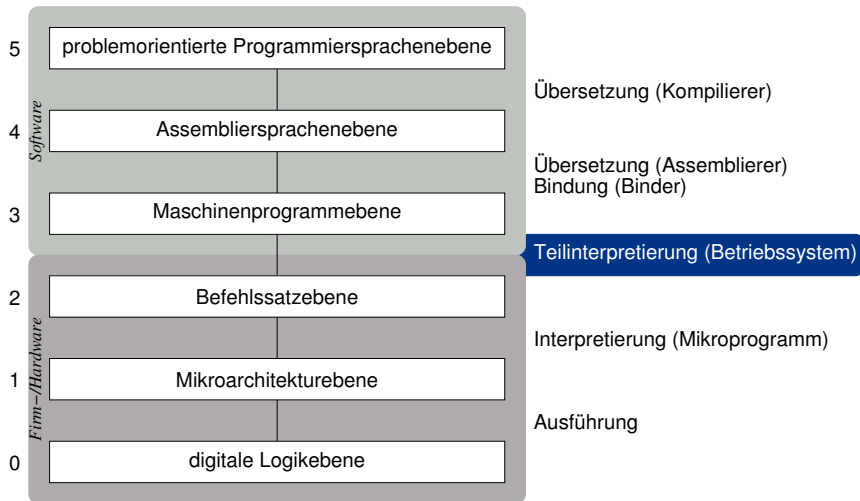
- Entvirtualisierung

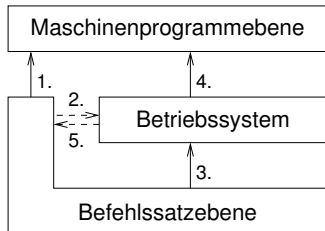
- Befehlsarten

- Ablaufkontext

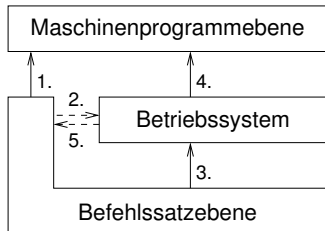
## Zusammenfassung



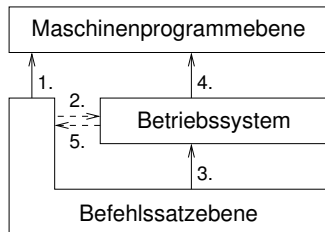




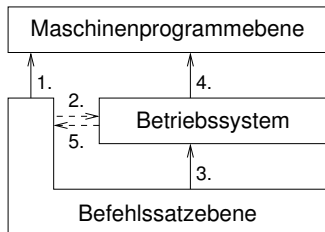
1. Die Befehlssatzebene interpretiert das Maschinenprogramm befehlswise



1. Die Befehlssatzebene interpretiert das Maschinenprogramm befehlsweise,
2. setzt dessen Ausführung aus,
  - Ausnahmesituation
  - **Programmunterbrechung**startet das Betriebssystem

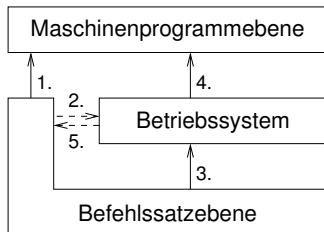


1. Die Befehlssatzebene interpretiert das Maschinenprogramm befehlsweise,
2. setzt dessen Ausführung aus,
  - Ausnahmesituation
  - **Programmunterbrechung**startet das Betriebssystem und
3. interpretiert die Programme des Betriebssystems befehlsweise.



1. Die Befehlssatzebene interpretiert das Maschinenprogramm befehlsweise,
2. setzt dessen Ausführung aus,
  - Ausnahmesituation
  - **Programmunterbrechung**startet das Betriebssystem und
3. interpretiert die Programme des Betriebssystems befehlsweise.

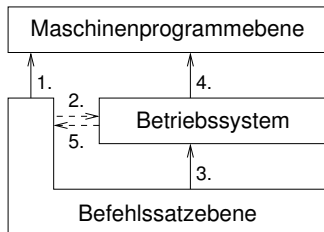
Folge von 3.: **Ausführung von Betriebssystemprogrammen**



1. Die Befehlsatzebene interpretiert das Maschinenprogramm befehlsweise,
2. setzt dessen Ausführung aus,
  - Ausnahmesituation
  - **Programmunterbrechung**startet das Betriebssystem und
3. interpretiert die Programme des Betriebssystems befehlsweise.

Folge von 3.: **Ausführung von Betriebssystemprogrammen**

4. Das *Betriebssystem interpretiert* das soeben oder zu einem früheren Zeitpunkt unterbrochene Maschinenprogramm<sup>1</sup> befehlsweise

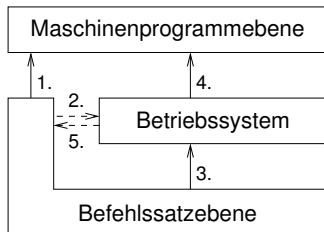


<sup>1</sup>Gegebenenfalls teilinterpretiert sich das Betriebssystem selbst partiell!

1. Die Befehlssatzebene interpretiert das Maschinenprogramm befehlsweise,
2. setzt dessen Ausführung aus,
  - Ausnahmesituation
  - **Programmunterbrechung**startet das Betriebssystem und
3. interpretiert die Programme des Betriebssystems befehlsweise.

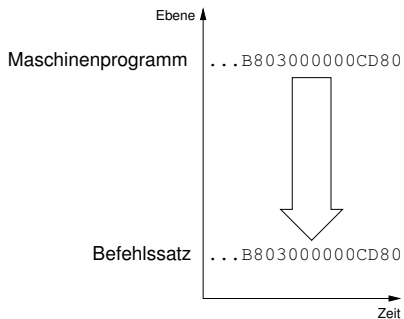
Folge von 3.: **Ausführung von Betriebssystemprogrammen**

4. Das *Betriebssystem interpretiert* das soeben oder zu einem früheren Zeitpunkt unterbrochene Maschinenprogramm<sup>1</sup> befehlsweise und
5. instruiert die Befehlssatzebene, die Ausführung des/eines zuvor unterbrochenen Maschinenprogramms<sup>1</sup> wieder aufzunehmen.



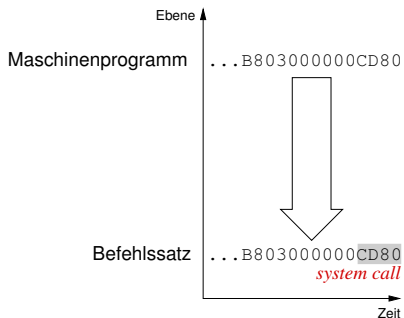
<sup>1</sup>Gegebenenfalls teilinterpretiert sich das Betriebssystem selbst partiell!





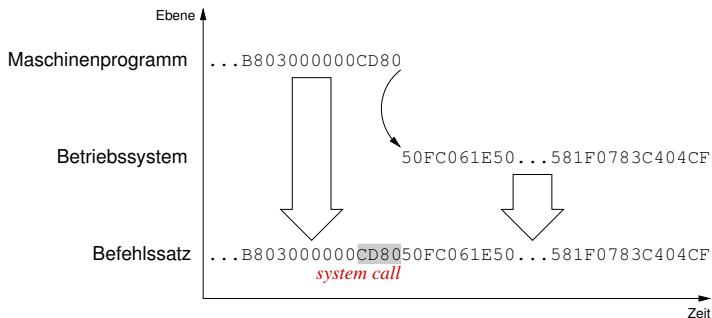
- Ausführung eines Maschinenprogramms





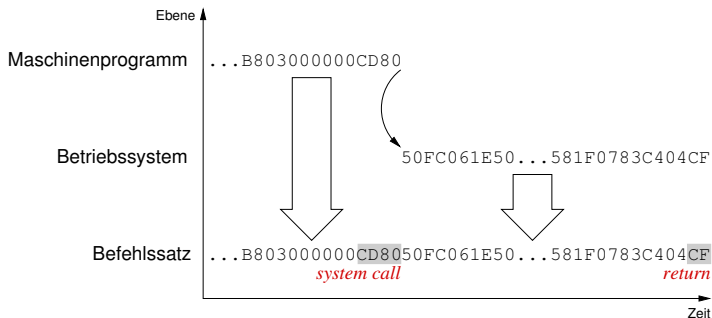
- Ausführung eines Maschinenprogramms
- Auslösung eines Systemaufrufs durch den Prozessor





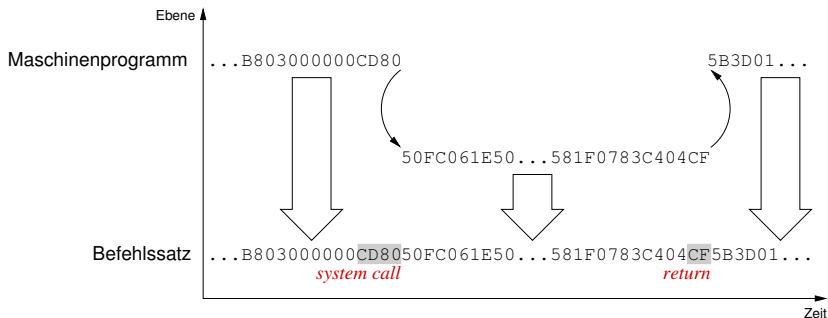
- Ausführung eines Maschinenprogramms
- Auslösung eines Systemaufrufs durch den Prozessor
- Verzweigung zum Betriebssystem und Behandlung des Systemaufrufs





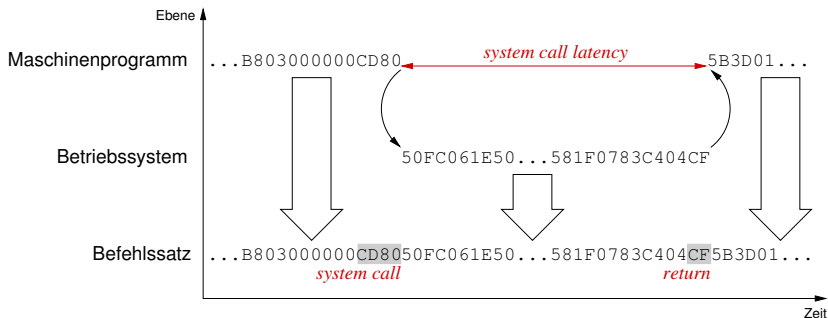
- Ausführung eines Maschinenprogramms
- Auslösung eines Systemaufrufs durch den Prozessor
- Verzweigung zum Betriebssystem und Behandlung des Systemaufrufs
- Beendigung des Systemaufrufs





- Ausführung eines Maschinenprogramms
- Auslösung eines Systemaufrufs durch den Prozessor
- Verzweigung zum Betriebssystem und Behandlung des Systemaufrufs
- Beendigung des Systemaufrufs
- Rückverzweigung zum Maschinenprogramm





- Ausführung eines Maschinenprogramms
- Auslösung eines Systemaufrufs durch den Prozessor
- Verzweigung zum Betriebssystem und Behandlung des Systemaufrufs
- Beendigung des Systemaufrufs
- Rückverzweigung zum Maschinenprogramm



Rekapitulation

Mehrebenenmaschinen

Teilinterpretierung

**Funktionale Hierarchie**

Analogie

Abstraktion

Implementierung

Entvirtualisierung

Befehlsarten

Ablaufkontext

Zusammenfassung

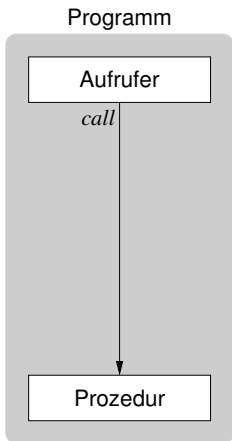


# Prozedur- vs. Systemaufruf

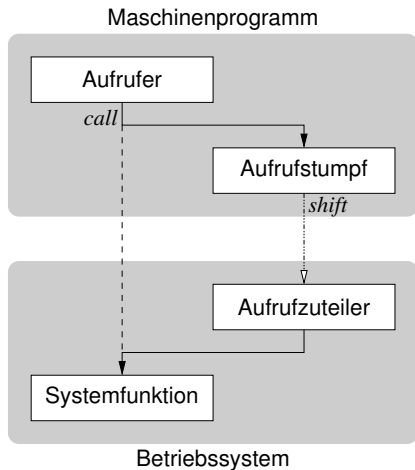
---



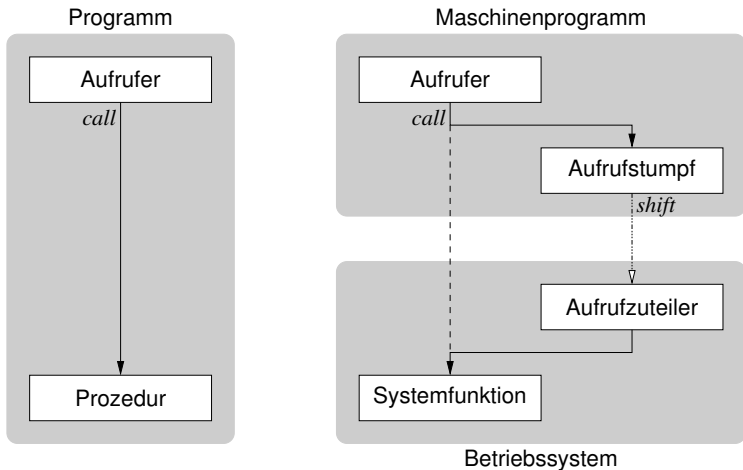
# Prozedur- vs. Systemaufruf



# Prozedur- vs. Systemaufruf



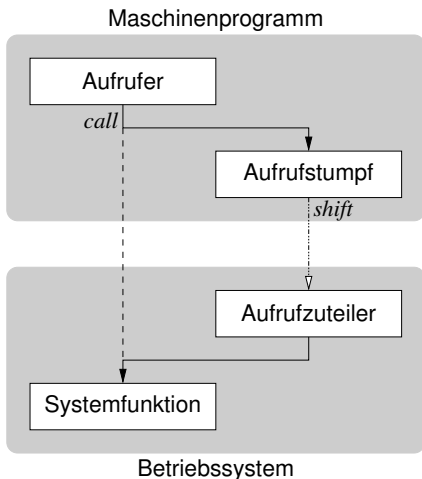
# Prozedur- vs. Systemaufruf



- Systemaufruf als adressraumübergreifender Prozeduraufruf
  - verlagert (*shift*) die weitere Prozedurausführung ins Betriebssystem



# Abstraktion von Betriebssystemabschottung

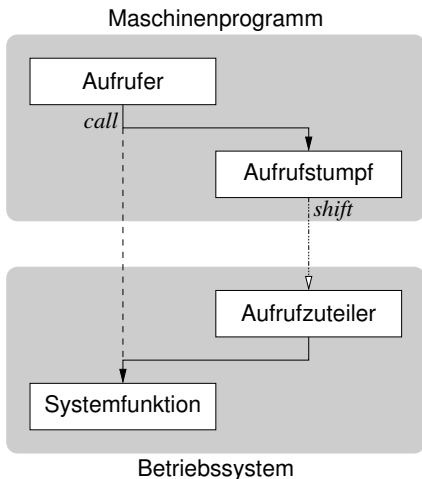


## Ortstransparenz

- durch den Aufrufstumpf
  - für den Aufrufer
- durch den Aufrufzuteiler
  - für die Systemfunktion



# Abstraktion von Betriebssystemabschottung



## Ortstransparenz

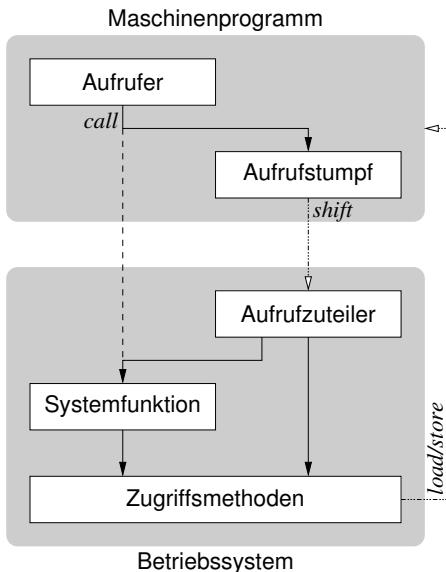
- durch den Aufrufstumpf
  - für den Aufrufer
- durch den Aufrufzuteiler
  - für die Systemfunktion

## Entkopplung

- des Maschinenprogramms
  - von Programmen des Betriebssystems
- ↪ ursprüngliches Anliegen



# Abstraktion von Maschinenprogrammabschottung



## Ortstransparenz

- durch den Aufrufstumpf
  - für den Aufrufer
- durch den Aufrufzuteiler
  - für die Systemfunktion

## Entkopplung

- des Maschinenprogramms
- von Programmen des Betriebssystems

## Zugriffstransparenz

- durch Zugriffsmethoden
  - für den Aufrufzuteiler
  - für die Systemfunktion



Standard ist die **synchrone Programmunterbrechung** (*trap*)

- Ausnahme (*exception*) von der „normalen“ Programmausführung
  - OS/360 ■ `svc`, für System/360 und danach
  - Unix V6 ■ `trap`, für PDP 11
  - Windows ■ `int $0x2e`
  - Linux ■ `int $0x80`, für x86
    - `swi`, für ARM
    - `t`, für SPARC



Standard ist die **synchrone Programmunterbrechung** (*trap*)

- Ausnahme (*exception*) von der „normalen“ Programmausführung

OS/360 ■ `svc`, für System/360 und danach

Unix V6 ■ `trap`, für PDP 11

Windows ■ `int $0x2e`

Linux ■ `int $0x80`, für x86

■ `swi`, für ARM

■ `t`, für SPARC

MacOS ■ `$0xa`, für m68k: *A-traps*, illegaler Operationskode<sup>2</sup>

■ `int $0x80`, für x86

---

<sup>2</sup>Motorola verwendete Befehle beginnend mit  $1111_2$  (reserviert für 68881, FPU-Koprozessor) und  $1010_2$  niemals in Prozessoren der 68000-Familie.

Standard ist die **synchrone Programmunterbrechung** (*trap*)

- Ausnahme (*exception*) von der „normalen“ Programmausführung

OS/360 ■ `svc`, für System/360 und danach

Unix V6 ■ `trap`, für PDP 11

Windows ■ `int $0x2e`

Linux ■ `int $0x80`, für x86

■ `swi`, für ARM

■ `t`, für SPARC

MacOS ■ `$0xa`, für m68k: *A-traps*, illegaler Operationskode<sup>2</sup>

■ `int $0x80`, für x86

- im Vergleich zum normalen Prozeduraufruf, sehr kostspielig (S. 27)

---

<sup>2</sup>Motorola verwendete Befehle beginnend mit  $1111_2$  (reserviert für 68881, FPU-Koprozessor) und  $1010_2$  niemals in Prozessoren der 68000-Familie.

Standard ist die **synchrone Programmunterbrechung** (*trap*)

- Ausnahme (*exception*) von der „normalen“ Programmausführung

OS/360 ■ `svc`, für System/360 und danach

Unix V6 ■ `trap`, für PDP 11

Windows ■ `int $0x2e`

Linux ■ `int $0x80`, für x86

■ `swi`, für ARM

■ `t`, für SPARC

MacOS ■ `$0xa`, für m68k: *A-traps*, illegaler Operationskode<sup>2</sup>

■ `int $0x80`, für x86

- im Vergleich zum normalen Prozeduraufruf, sehr kostspielig (S. 27)

Avantgarde sind Ansätze, die im Grunde frei von Aufrufsemantik sind

- der Fokus liegt auf **Moduswechsel**: `sysenter/syscall` (x86-64)

---

<sup>2</sup>Motorola verwendete Befehle beginnend mit `11112` (reserviert für 68881, FPU-Koprozessor) und `10102` niemals in Prozessoren der 68000-Familie.

## Rekapitulation

Mehrebenenmaschinen

Teilinterpretierung

## Funktionale Hierarchie

Analogie

Abstraktion

## Implementierung

Entvirtualisierung

Befehlsarten

Ablaufkontext

## Zusammenfassung



- Systemaufruf als Konstrukt **problemorientierter Programmiersprache**

```
1 int done;  
2 char buf[1];  
3  
4 done = read(0, buf, sizeof(buf));
```



- Systemaufruf als Konstrukt **problemorientierter Programmiersprache**

```
1 int done;  
2 char buf[1];  
3  
4 done = read(0, buf, sizeof(buf));
```

- seine semantisch äquivalente Umsetzung in Assemblersprache

- gcc -O6 -m32 -fomit-frame-pointer -S

```
1 subl $12, %esp      ; allocate parameter block  
2 movl $1, 8(%esp)   ; input buffer: length (in bytes)  
3 movl $buf, 4(%esp) ; input buffer: address  
4 movl $0, (%esp)    ; file descriptor: standard input  
5 call read          ; execute system function  
6 movl %eax, done    ; save return code  
7 addl $12, %esp     ; release parameter block
```



- Systemaufruf als Konstrukt der **Maschinenprogrammzebene**:

```
1 read:
2   pushl %ebx           ; backup callee-save register
3   movl  16(%esp), %edx ; pass 3rd input parameter
4   movl  12(%esp), %ecx ; pass 2nd input parameter
5   movl  8(%esp), %ebx  ; pass 1st input parameter
6   scar  $3             ; perform system call and return
7   popl  %ebx           ; restore callee-save register
8   ret
```



- Systemaufruf als Konstrukt der **Maschinenprogrammzebene**:

```
1 read:
2   pushl %ebx           ; backup callee-save register
3   movl  16(%esp), %edx ; pass 3rd input parameter
4   movl  12(%esp), %ecx ; pass 2nd input parameter
5   movl  8(%esp), %ebx  ; pass 1st input parameter
6   scar  $3            ; perform system call and return
7   popl  %ebx          ; restore callee-save register
8   ret
```

- problemspezifische Varianten, je nach **Betriebssystembefehlsart**:

- Primitivbefehl (RISC-artig), im Beispiel hier (Linux-artig) und *ff*.
  - Anzahl der zu sichernden nichtflüchtigen (*callee-save*) Register
  - Hauptspeicher oder flüchtige (*caller-save*) Register als Sicherungspuffer
  - stapel- oder registerbasierte Parameterübergabe
  - rückkehrende oder rückkehrlose Interaktion mit dem Betriebssystem
- Komplexbefehl (CISC-artig), vgl. auch S. 21



- rückkehrender Systemaufruf mit zwei Eingabeparametern:

```
1 kill:
2     movl %ebx, %edx      ; backup into caller-save register
3     movl 8(%esp), %ecx  ; pass 2nd input parameter
4     movl 4(%esp), %ebx  ; pass 1st input parameter
5     scar $37           ; perform system call and return
6     movl %edx, %ebx    ; restore from caller-save register
7     ret
```



- rückkehrender Systemaufruf mit zwei Eingabeparametern:

```
1 kill:
2     movl %ebx, %edx    ; backup into caller-save register
3     movl 8(%esp), %ecx ; pass 2nd input parameter
4     movl 4(%esp), %ebx ; pass 1st input parameter
5     scar $37          ; perform system call and return
6     movl %edx, %ebx   ; restore from caller-save register
7     ret
```

- rückkehrloser Systemaufruf mit einem Eingabeparameter:

```
1 _exit:
2     movl 4(%esp), %ebx ; pass input parameter
3     sc    $252         ; perform system call, no return
```



- rückkehrender Systemaufruf mit zwei Eingabeparametern:

```
1 kill:
2   movl %ebx, %edx    ; backup into caller-save register
3   movl 8(%esp), %ecx ; pass 2nd input parameter
4   movl 4(%esp), %ebx ; pass 1st input parameter
5   sc   $37          ; perform system call and return
6   movl %edx, %ebx   ; restore from caller-save register
7   ret
```

- rückkehrloser Systemaufruf mit einem Eingabeparameter:

```
1 _exit:
2   movl 4(%esp), %ebx ; pass input parameter
3   sc   $252          ; perform system call, no return
```

- rückkehrender parameterloser Systemaufruf:

```
1 getpid:
2   sc   $20           ; perform system call and return
3   ret
```



## ■ Absetzen des Systemaufrufs

```
1 .macro sc scn
2     movl  \scn, %eax    ; pass system call number
3     int   $128         ; cause software interrupt
4 .endm
```



## ■ Absetzen des Systemaufrufs

```
1 .macro sc scn
2     movl  \scn, %eax    ; pass system call number
3     int   $128         ; cause software interrupt
4 .endm
```

## ■ Systemaufruf und Fehlerbehandlung nach Rückkehr

```
1 .macro scar scn
2     sc     \scn        ; perform system call and return
3     cmpl  $-4095, %eax ; check for system call error
4     jb    .s\@         ; normal operation, if applicable
5     neg   %eax         ; derive (positiv) error code
6     movl  %eax, errno  ; put aside for possibly reworking
7     movl  $-1, %eax    ; indicate failure of operation
8 .s\@:                ; come here if error free
9 .endm
```



### ■ Absetzen des Systemaufrufs

```
1 .macro sc scn
2     movl  \scn, %eax    ; pass system call number
3     int   $128         ; cause software interrupt
4 .endm
```

### ■ Systemaufruf und Fehlerbehandlung nach Rückkehr

```
1 .macro scar scn
2     sc     \scn        ; perform system call and return
3     cmpl  $-4095, %eax ; check for system call error
4     jb    .s\@         ; normal operation, if applicable
5     neg   %eax         ; derive (positiv) error code
6     movl  %eax, errno  ; put aside for possibly reworking
7     movl  $-1, %eax    ; indicate failure of operation
8 .s\@:          ; come here if error free
9 .endm
```

### ■ Platzhalter für den Fehlercode (im Datensegment, .data)

```
1 .long  errno
```



- Problem: Schutzdomänen überschreitende **Ausnahmeauslösung**
  - normale Funktionsergebnisse von ausnahmebedingten unterscheiden
  - eine für das gesamte Rechensystem **effiziente Umsetzung** durchsetzen



- Problem: Schutzdomänen überschreitende **Ausnahmeauslösung**
  - normale Funktionsergebnisse von ausnahmebedingten unterscheiden
  - eine für das gesamte Rechensystem **effiziente Umsetzung** durchsetzen
- Lösungen dazu hängen ab von Betriebssystem und Befehlssatzebene



- Problem: Schutzdomänen überschreitende **Ausnahmeauslösung**
  - normale Funktionsergebnisse von ausnahmebedingten unterscheiden
  - eine für das gesamte Rechensystem **effiziente Umsetzung** durchsetzen
- Lösungen dazu hängen ab von Betriebssystem und Befehlssatzebene
  - Wertebereich für Funktionsergebnisse beschneiden (z. B. Linux)
    - Wert im Rückgaberegister (%eax) zeigt den Ausnahme- oder Normalfall an
      - $v \in [-1, -4095] \Rightarrow v$  ist Fehlercode  $\geq 0xffffffff000$  (IA-32)
      - sonst  $\Rightarrow v$  ist Funktionsergebnis  $< 0xffffffff000$  (IA-32)
    - betriebssystemseitig einfach, sofern *alle* Funktionsergebnisse dazu passen



- Problem: Schutzdomänen überschreitende **Ausnahmeauslösung**
  - normale Funktionsergebnisse von ausnahmebedingten unterscheiden
  - eine für das gesamte Rechensystem **effiziente Umsetzung** durchsetzen
- Lösungen dazu hängen ab von Betriebssystem und Befehlssatzebene
  - Wertebereich für Funktionsergebnisse beschneiden (z. B. Linux)
    - Wert im Rückgaberegister (%eax) zeigt den Ausnahme- oder Normalfall an
$$v \in [-1, -4095] \Rightarrow v \text{ ist Fehlercode} \geq 0xffffffff000 \text{ (IA-32)}$$
$$\text{sonst} \Rightarrow v \text{ ist Funktionsergebnis} < 0xffffffff000 \text{ (IA-32)}$$
    - betriebssystemseitig einfach, sofern *alle* Funktionsergebnisse dazu passen
  - Übertragsmerker (*carry flag*) im Statusregister setzen<sup>3</sup>
    - Stapelrahmen (*stack frame*) des Systemaufrufs so manipulieren, dass bei Rückkehr der Merker den Ausnahme- (1) oder Normalfall (0) anzeigt
    - betriebssystemseitig mit größerem Mehraufwand (*overhead*) verbunden

<sup>3</sup>Jeder Merker zur Steuerung bedingter Sprünge eignet sich dafür.

- Problem: Schutzdomänen überschreitende **Ausnahmeauslösung**
    - normale Funktionsergebnisse von ausnahmebedingten unterscheiden
    - eine für das gesamte Rechensystem **effiziente Umsetzung** durchsetzen
  - Lösungen dazu hängen ab von Betriebssystem und Befehlssatzebene
    - Wertebereich für Funktionsergebnisse beschneiden (z. B. Linux)
      - Wert im Rückgaberegister (%eax) zeigt den Ausnahme- oder Normalfall an
- $$\begin{array}{llll} v \in [-1, -4095] & \Rightarrow & v \text{ ist Fehlercode} & \geq 0xffffffff000 \text{ (IA-32)} \\ & & \text{sonst} & \Rightarrow v \text{ ist Funktionsergebnis} < 0xffffffff000 \text{ (IA-32)} \end{array}$$
- betriebssystemseitig einfach, sofern *alle* Funktionsergebnisse dazu passen
  - Übertragsmerker (*carry flag*) im Statusregister setzen<sup>3</sup>
    - Stapelrahmen (*stack frame*) des Systemaufrufs so manipulieren, dass bei Rückkehr der Merker den Ausnahme- (1) oder Normalfall (0) anzeigt
    - betriebssystemseitig mit größerem Mehraufwand (*overhead*) verbunden
- als **Befehlssatzebenerweiterung** wäre der Merkeransatz konsequent

<sup>3</sup>Jeder Merker zur Steuerung bedingter Sprünge eignet sich dafür.

```
1  scd :
2      pushl %ebp
3      pushl %edi
4      pushl %esi
5      pushl %edx
6      pushl %ecx
7      pushl %ebx
8      cmpl  $NJTE,%eax
9      jae   scd_fault
10     call  *jump_table(,%eax,4)
11  scd_leave :
12     popl  %ebx
13     popl  %ecx
14     popl  %edx
15     popl  %esi
16     popl  %edi
17     popl  %ebp
18     iret
```



1	<code>scd :</code>	
2	<code>  pushl %ebp</code>	<i>system call dispatcher:</i>
3	<code>  pushl %edi</code>	2–7 i Sicherung
4	<code>  pushl %esi</code>	ii Parametertransfer
5	<code>  pushl %edx</code>	
6	<code>  pushl %ecx</code>	8–9 Überprüfung
7	<code>  pushl %ebx</code>	10 Ausführung
8	<code>  cmpl \$NJTE,%eax</code>	
9	<code>  jae  scd_fault</code>	12–17 Wiederherstellung
10	<code>  call *jump_table(,%eax,4)</code>	18 Wiederaufnahme
11	<code>scd_leave :</code>	
12	<code>  popl %ebx</code>	
13	<code>  popl %ecx</code>	
14	<code>  popl %edx</code>	
15	<code>  popl %esi</code>	
16	<code>  popl %edi</code>	
17	<code>  popl %ebp</code>	
18	<code>  iret</code>	



```
1  scd :
2    pushl %ebp
3    pushl %edi
4    pushl %esi
5    pushl %edx
6    pushl %ecx
7    pushl %ebx
8    cmpl  $NJTE,%eax
9    jae   scd_fault
10   call  *jump_table(,%eax,4)
11  scd_leave :
12   popl  %ebx
13   popl  %ecx
14   popl  %edx
15   popl  %esi
16   popl  %edi
17   popl  %ebp
18   iret
```

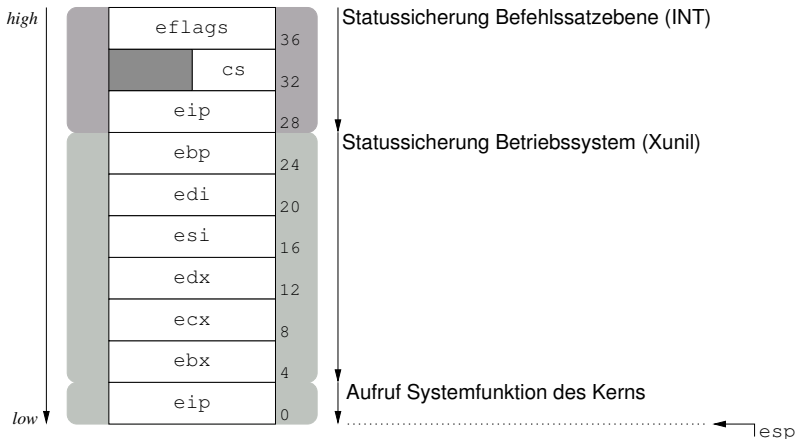
*system call dispatcher:*

- 2–7 i Sicherung
- ii Parametertransfer
- 8–9 Überprüfung
- 10 Ausführung
- 12–17 Wiederherstellung
- 18 Wiederaufnahme

### Fehlerbehandlung

```
1  scd_fault :
2    movl  $-ENOSYS,%eax
3    jmp   scd_leave
```





- Stapelaufbau<sup>4</sup> nach Aufruf der Systemfunktion über die Sprungtabelle
  - `call *jump_table(,%eax,4)`

<sup>4</sup>IA-32 *real-address mode*



```
1 extern int sys_ni_syscall(void);
2 extern int sys_exit(int);
3 extern int sys_fork(void);
4 extern int sys_read(int, void*, int);
5 extern int sys_write(int, void*, int);
6 ...
7
8 #define NJTE 326 /* number of jump table entries */
9
10 int (*jump_table[NJTE])() = { /* opcode */
11     sys_ni_syscall,          /* 0 */
12     sys_exit,                /* 1 */
13     sys_fork,                /* 2 */
14     sys_read,                /* 3 */
15     sys_write,              /* 4 */
16     ...
17 };
```



```
1  asmlinkage
2  ssize_t sys_read(unsigned fd, char *buf, size_t count) {
3      ssize_t ret;
4      struct file *file;
5
6      ret = -EBADF;
7      file = fget(fd);
8      if (file) {
9          ...
10     }
11     return ret;
12 }
```

#### asmlinkage

Instruiert gcc, die Funktionsparameter auf dem Stapel zu erwarten und nicht in Prozessorregistern.



```
1  asmlinkage
2  ssize_t sys_read(unsigned fd, char *buf, size_t count) {
3      ssize_t ret;
4      struct file *file;
5
6      ret = -EBADF;
7      file = fget(fd);
8      if (file) {
9          ...
10     }
11     return ret;
12 }
```

#### asmlinkage

Instruiert gcc, die Funktionsparameter auf dem Stapel zu erwarten und nicht in Prozessorregistern.

```
1  asmlinkage long sys_ni_syscall(void) {
2      return -ENOSYS;
3  }
```



## ■ Primitivbefehl

```
1  movl  op6, %ebp
2  movl  op5, %edi
3  movl  op4, %esi
4  movl  op3, %edx
5  movl  op2, %ecx
6  movl  op1, %ebx
7  movl  opc, %eax
8  int   $42
```



## ■ Primitivbefehl

```
1  movl  op6, %ebp
2  movl  op5, %edi
3  movl  op4, %esi
4  movl  op3, %edx
5  movl  op2, %ecx
6  movl  op1, %ebx
7  movl  opc, %eax
8  int   $42
```

## ■ Komplexbefehl

```
1          int   $42
2  .long  opc
3  .long  op1
4  .long  op2
5  further operands
6  .long  opn
```



## ■ Primitivbefehl

```
1  movl  op6, %ebp
2  movl  op5, %edi
3  movl  op4, %esi
4  movl  op3, %edx
5  movl  op2, %ecx
6  movl  op1, %ebx
7  movl  opc, %eax
8  int   $42
```

## ■ Komplexbefehl: uniforme (li.) oder individuelle (re.) Operanden

1	int \$42	1	int \$42
2	.long op	2	.byte op
3	.long op1	3	.align 4
4	.long op2	4	.long op1
5	further operands	5	.long op2
6	.long opn	6	further operands/alignments
		7	.long opn



## ■ Primitivbefehl

```

1  movl  op6, %ebp
2  movl  op5, %edi
3  movl  op4, %esi
4  movl  op3, %edx
5  movl  op2, %ecx
6  movl  op1, %ebx
7  movl  opc, %eax
8  int   $42

```

### Beachte

- bei Primitivbefehlen erfolgt die Auswertung der Operanden dynamisch, zur Laufzeit
  - Prozessorregister müssen freigemacht werden
- bei Komplexbefehlen geschieht dies statisch, zur Assembler-/Bindezeit, und registerlos

## ■ Komplexbefehl: uniforme (li.) oder individuelle (re.) Operanden

```

1      int   $42
2  .long  opc
3  .long  op1
4  .long  op2
5  further operands
6  .long  opn

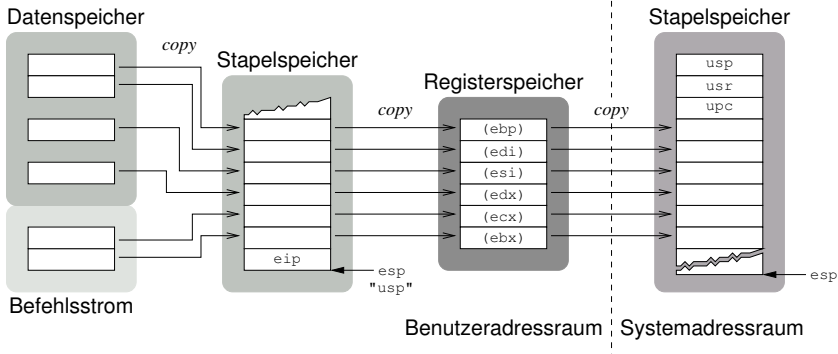
```

```

1      int   $42
2  .byte  opc
3  .align 4
4  .long  op1
5  .long  op2
6  further operands/alignments
7  .long  opn

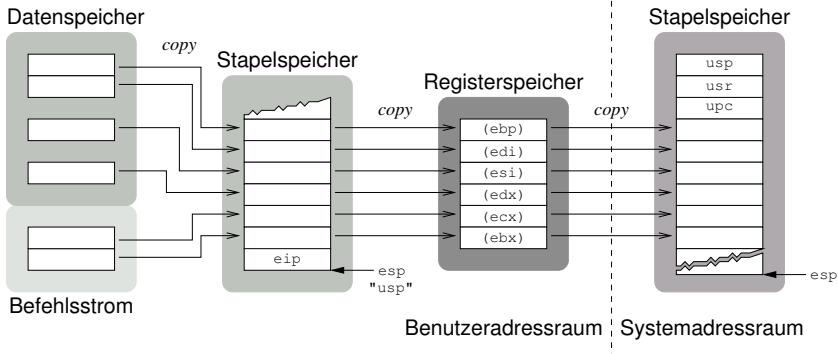
```





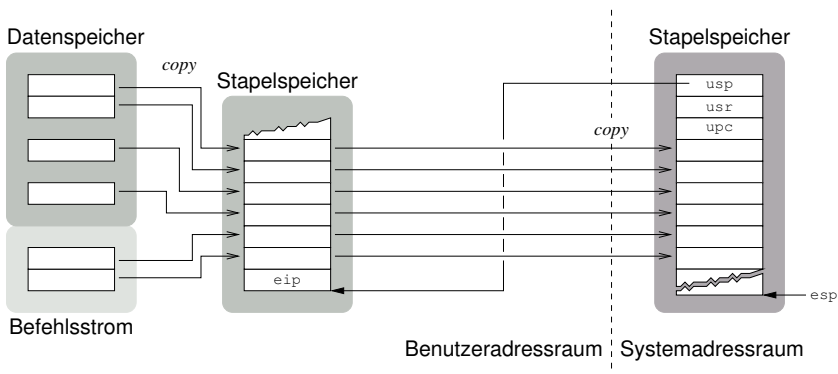
- **Werteübergabe** (*call by value*) für alle Parameter
  - Variable: Befehlsoperand ist Adresse im Datenspeicher inkl. Register
  - Direktwert: Bestandteil des Befehls im Befehlsstrom





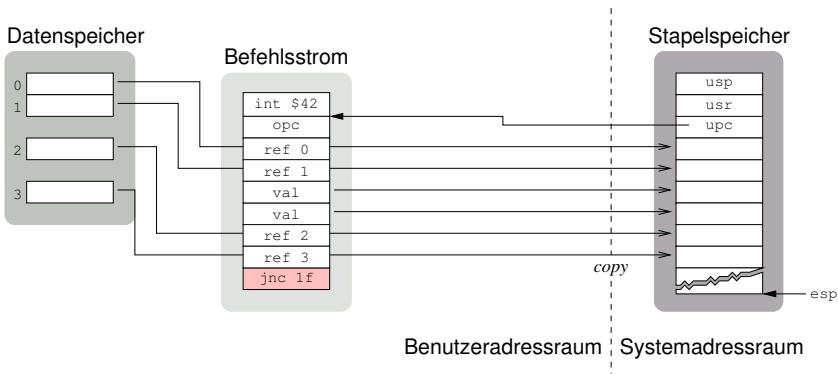
- **Werteübergabe** (*call by value*) für alle Parameter
  - Variable: Befehlsoperand ist Adresse im Datenspeicher inkl. Register
  - Direktwert: Bestandteil des Befehls im Befehlsstrom
- Systemaufrufe als Primitivbefehle sind (meist) **Unterprogramme**





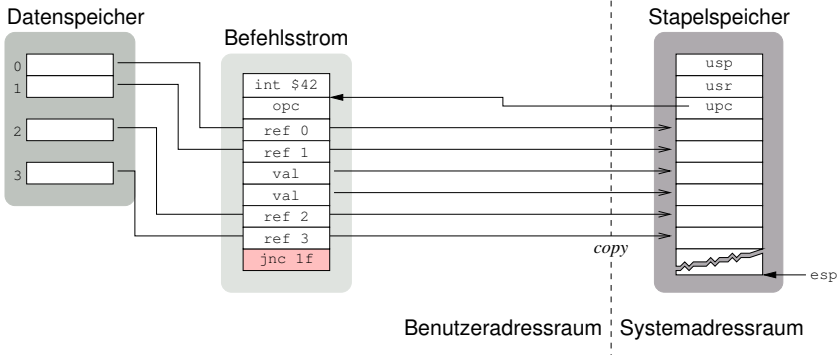
- Systemaufrufparameter werden nicht (mehr) in Registern transferiert
  - Systemaufrufe sind Unterprogramme, Parameter werden gestapelt
  - in Ergänzung zum Registeransatz, falls die Parameteranzahl zu groß ist
- das Betriebssystem lädt Parameter direkt vom Benutzerstapel





- das Betriebssystem lädt Parameter direkt vom Benutzeradressraum
  - Werteübergabe (*call by value*) für alle Direktwerte
  - Referenzübergabe (*call by reference*) sonst





- das Betriebssystem lädt Parameter direkt vom Benutzeradressraum
  - **Werteübergabe** (*call by value*) für alle Direktwerte
  - **Referenzübergabe** (*call by reference*) sonst
- Systemaufrufe als Komplexbefehle sind (meist) **Makroanweisungen**





- Primitivbefehl
  - +/- Werteübergabe von Operanden im Maschinenprogramm
  - +/- dynamische Operandenauswertung (Laufzeit)



## ■ Primitivbefehl

- +/- Werteübergabe von Operanden im Maschinenprogramm
- +/- dynamische Operandenauswertung (Laufzeit)
  - durch Prozessorregistersatz begrenzte Operandenanzahl
  - betriebssystemseitig bestenfalls teilweise Zustandssicherung
  - maschinenprogrammseitiger Mehraufwand zum Operandenabruf



## ■ Primitivbefehl

- +/- Werteübergabe von Operanden im Maschinenprogramm
- +/- dynamische Operandenauswertung (Laufzeit)
  - durch Prozessorregistersatz begrenzte Operandenanzahl
  - betriebssystemseitig bestenfalls teilweise Zustandssicherung
  - maschinenprogrammseitiger Mehraufwand zum Operandenabruf

## ■ Komplexbefehl

- + entspricht dem (statischen) Befehlsformat der Befehlssatzebene
- + kompakte Darstellung/Kodierung von Systemaufrufen
- + vollständige betriebssystemseitige Zustandssicherung



## ■ Primitivbefehl

- +/- Werteübergabe von Operanden im Maschinenprogramm
- +/- dynamische Operandenauswertung (Laufzeit)
  - durch Prozessorregistersatz begrenzte Operandenanzahl
  - betriebssystemseitig bestenfalls teilweise Zustandssicherung
  - maschinenprogrammseitiger Mehraufwand zum Operandenabruf

## ■ Komplexbefehl

- + entspricht dem (statischen) Befehlsformat der Befehlssatzebene
- + kompakte Darstellung/Kodierung von Systemaufrufen
- + vollständige betriebssystemseitige Zustandssicherung
- +/- statische Operandenauswertung (Assembler- oder Bindezeit)



## ■ Primitivbefehl

- +/- Werteübergabe von Operanden im Maschinenprogramm
- +/- dynamische Operandenauswertung (Laufzeit)
  - durch Prozessorregistersatz begrenzte Operandenanzahl
  - betriebssystemseitig bestenfalls teilweise Zustandssicherung
  - maschinenprogrammseitiger Mehraufwand zum Operandenabruf

## ■ Komplexbefehl

- + entspricht dem (statischen) Befehlsformat der Befehlssatzebene
- + kompakte Darstellung/Kodierung von Systemaufrufen
- + vollständige betriebssystemseitige Zustandssicherung
- +/- statische Operandenauswertung (Assembler- oder Bindezeit)
  - Referenzübergabe von Operanden im Maschinenprogramm
  - betriebssystemseitiger Mehraufwand zum Operandenabruf



## ■ Primitivbefehl

- +/- Werteübergabe von Operanden im Maschinenprogramm
- +/- dynamische Operandenauswertung (Laufzeit)
  - durch Prozessorregistersatz begrenzte Operandenanzahl
  - betriebssystemseitig bestenfalls teilweise Zustandssicherung
  - maschinenprogrammseitiger Mehraufwand zum Operandenabruf

## ■ Komplexbefehl

- + entspricht dem (statischen) Befehlsformat der Befehlssatzebene
- + kompakte Darstellung/Kodierung von Systemaufrufen
- + vollständige betriebssystemseitige Zustandssicherung
- +/- statische Operandenauswertung (Assembler- oder Bindezeit)
  - Referenzübergabe von Operanden im Maschinenprogramm
  - betriebssystemseitiger Mehraufwand zum Operandenabruf

- wie gravierend die Negativpunkte sind, hängt vom Anwendungsfall ab



- reale Sicht: ursprünglicher Zweck von Systemaufrufen (um 1955)
  - transiente Maschinenprogramme und residente Systemsoftware trennen



- reale Sicht: ursprünglicher Zweck von Systemaufrufen (um 1955)
  - transiente Maschinenprogramme und residente Systemsoftware trennen
- logische Sicht: Systemaufrufe aktivieren einen privilegierten Kontext
  - Abschottung des Betriebssystemadressraums
    - Wechsel hin zum eigenen Adressraum des Betriebssystems
    - Erweiterung um den Adressraum des aufrufenden Maschinenprogramms
  - Erlaubnis zur (eingeschränkten) Durchführung bevorzogter Funktionen
    - Speicher-/Geräteverwaltung, Ein-/Ausgabe, . . . , Betriebssystemdienste
    - allgemein: direkte Ausführung von Programmen der Befehlssatzebene
  - Zusicherung eigener Softwarebetriebsmittel zur Programmausführung
    - Stapelspeicher:  $1 : 1 \rightsquigarrow$  prozessbasierter,  $N : 1 \rightsquigarrow$  ereignisbasierter Kern
    - Prozessorregistersatz: Sicherung/Wiederherstellung oder Spiegelung



- reale Sicht: ursprünglicher Zweck von Systemaufrufen (um 1955)
  - transiente Maschinenprogramme und residente Systemsoftware trennen
- logische Sicht: Systemaufrufe aktivieren einen privilegierten Kontext
  - Abschottung des Betriebssystemadressraums
    - Wechsel hin zum eigenen Adressraum des Betriebssystems
    - Erweiterung um den Adressraum des aufrufenden Maschinenprogramms
  - Erlaubnis zur (eingeschränkten) Durchführung bevorzogter Funktionen
    - Speicher-/Geräteverwaltung, Ein-/Ausgabe, . . . , Betriebssystemdienste
    - allgemein: direkte Ausführung von Programmen der Befehlssatzebene
  - Zusicherung eigener Softwarebetriebsmittel zur Programmausführung
    - Stapelspeicher:  $1 : 1 \rightsquigarrow$  prozessbasierter,  $N : 1 \rightsquigarrow$  ereignisbasierter Kern
    - Prozessorregistersatz: Sicherung/Wiederherstellung oder Spiegelung



- reale Sicht: ursprünglicher Zweck von Systemaufrufen (um 1955)
  - transiente Maschinenprogramme und residente Systemsoftware trennen
- logische Sicht: Systemaufrufe aktivieren einen privilegierten Kontext
  - Abschottung des Betriebssystemadressraums
    - Wechsel hin zum eigenen Adressraum des Betriebssystems
    - Erweiterung um den Adressraum des aufrufenden Maschinenprogramms
  - Erlaubnis zur (eingeschränkten) Durchführung bevorzogter Funktionen
    - Speicher-/Geräteverwaltung, Ein-/Ausgabe, . . . , Betriebssystemdienste
    - allgemein: direkte Ausführung von Programmen der Befehlssatzebene
  - Zusicherung eigener Softwarebetriebsmittel zur Programmausführung
    - Stapelspeicher:  $1 : 1 \rightsquigarrow$  prozessbasierter,  $N : 1 \rightsquigarrow$  ereignisbasierter Kern
    - Prozessorregistersatz: Sicherung/Wiederherstellung oder Spiegelung



- reale Sicht: ursprünglicher Zweck von Systemaufrufen (um 1955)
  - transiente Maschinenprogramme und residente Systemsoftware trennen
- logische Sicht: Systemaufrufe aktivieren einen privilegierten Kontext
  - Abschottung des Betriebssystemadressraums
    - Wechsel hin zum eigenen Adressraum des Betriebssystems
    - Erweiterung um den Adressraum des aufrufenden Maschinenprogramms
  - Erlaubnis zur (eingeschränkten) Durchführung bevorzogter Funktionen
    - Speicher-/Geräteverwaltung, Ein-/Ausgabe, . . . , Betriebssystemdienste
    - allgemein: direkte Ausführung von Programmen der Befehlssatzebene
  - Zusicherung eigener Softwarebetriebsmittel zur Programmausführung
    - Stapelspeicher:  $1 : 1 \rightsquigarrow$  prozessbasierter,  $N : 1 \rightsquigarrow$  ereignisbasierter Kern
    - Prozessorregistersatz: Sicherung/Wiederherstellung oder Spiegelung



Systemaufrufe als eine synchrone Programmunterbrechung (*trap*) zu realisieren, ist ein mögliches Mittel zum Zweck und kein Muss

- effektiv müssen mit dem Mittel zwei Eigenschaften durchsetzbar sein:
  - i **privilegierter Arbeitsmodus** für den Betriebssystemkern
  - ii **Integrität** – Verhinderung einer Infiltration<sup>5</sup> ersterer Eigenschaft

---

<sup>5</sup>Im Sinne von „verdeckte Spionage und Sabotage in anderen Strukturen“.

# Abschottung und bevorrechtigte Ausführung

Systemaufrufe als eine synchrone Programmunterbrechung (*trap*) zu realisieren, ist ein mögliches Mittel zum Zweck und kein Muss

- effektiv müssen mit dem Mittel zwei Eigenschaften durchsetzbar sein:
  - i **privilegierter Arbeitsmodus** für den Betriebssystemkern
  - ii **Integrität** – Verhinderung einer Infiltration<sup>5</sup> ersterer Eigenschaft
- ein *Trap* ist hinreichendes Mittel, aber auch vergleichsweise teuer
  - Zustandssicherung, Speicher- bzw. Tabellensuchen (*table look-up*)

<sup>5</sup>Im Sinne von „verdeckte Spionage und Sabotage in anderen Strukturen“.



# Abschottung und bevorrechtigte Ausführung

Systemaufrufe als eine synchrone Programmunterbrechung (*trap*) zu realisieren, ist ein mögliches Mittel zum Zweck und kein Muss

- effektiv müssen mit dem Mittel zwei Eigenschaften durchsetzbar sein:
  - i **privilegierter Arbeitsmodus** für den Betriebssystemkern
  - ii **Integrität** – Verhinderung einer Infiltration<sup>5</sup> ersterer Eigenschaft
- ein *Trap* ist hinreichendes Mittel, aber auch vergleichsweise teuer
  - Zustandssicherung, Speicher- bzw. Tabellensuchen (*table look-up*)

Systemaufrufbeschleunigung durch **Spezialbefehle** (Intel, Pentium II)

- privilegierten Programmtext nahezu „in Reihe“ (*inline*) anordnen

```
1     movl $1f, %edx ; user mode continuation address
2     movl %esp, %ecx ; user mode stack pointer
3     sysenter      ; enlist in privileged mode
4 1:
```

- vgl. auch VDSO (*virtual dynamic shared object*) in Linux

<sup>5</sup>Im Sinne von „verdeckte Spionage und Sabotage in anderen Strukturen“.



- Kontextwechsel der CPU ohne Kontextsicherung und Tabellensuche
  - `sysenter`
    - setzt CS, EIP und SS, ESP auf systemsspezifische Werte
    - schaltet Segmentierung ab (CS und SS:  $[0..2^{32} - 1]$ )
    - sperrt asynchrone Programmunterbrechungen (IRQ)
    - aktiviert Schutzring 0
  - `sysexit`
    - setzt CS und SS auf prozessspezifische Werte
    - setzt EIP/ESP auf die in EDX/ECX stehenden Werte
    - aktiviert Schutzring 3 – nur von Ring 0 aus ausführbar



- Kontextwechsel der CPU ohne Kontextsicherung und Tabellensuche
  - `sysenter`
    - setzt CS, EIP und SS, ESP auf systemspezifische Werte
    - schaltet Segmentierung ab (CS und SS:  $[0..2^{32} - 1]$ )
    - sperrt asynchrone Programmunterbrechungen (IRQ)
    - aktiviert Schutzring 0
  - `sysexit`
    - setzt CS und SS auf prozessspezifische Werte
    - setzt EIP/ESP auf die in EDX/ECX stehenden Werte
    - aktiviert Schutzring 3 – nur von Ring 0 aus ausführbar
- das Betriebssystem belegt **modellspezifische Register** der CPU vor
  - MSR (*model-specific register*) 174h, 175h, 176h: CS, ESP und EIP, resp.
    - bei `sysenter`:  $SS = MSR[174h] + 8$
    - bei `sysexit`:  $CS = MSR[174h] + 16$ ,  $SS = MSR[174h] + 24$
  - mit  $MSR[174h]$  als eine Art „Basisindexregister“ in die Segmenttabelle



- Kontextwechsel der CPU ohne Kontextsicherung und Tabellensuche
  - `sysenter`
    - setzt CS, EIP und SS, ESP auf systemspezifische Werte
    - schaltet Segmentierung ab (CS und SS:  $[0..2^{32} - 1]$ )
    - sperrt asynchrone Programmunterbrechungen (IRQ)
    - aktiviert Schutzring 0
  - `sysexit`
    - setzt CS und SS auf prozessspezifische Werte
    - setzt EIP/ESP auf die in EDX/ECX stehenden Werte
    - aktiviert Schutzring 3 – nur von Ring 0 aus ausführbar
- das Betriebssystem belegt **modellspezifische Register** der CPU vor
  - MSR (*model-specific register*) 174h, 175h, 176h: CS, ESP und EIP, resp.
    - bei `sysenter`:  $SS = MSR[174h] + 8$
    - bei `sysexit`:  $CS = MSR[174h] + 16, SS = MSR[174h] + 24$
  - mit  $MSR[174h]$  als eine Art „Basisindexregister“ in die Segmenttabelle
- Kontextsicherung liegt komplett in Hand des Benutzerprozesses. . .



- Kontextwechsel der CPU ohne Kontextsicherung und Tabellensuche
  - `sysenter`
    - setzt CS, EIP und SS, ESP auf systemspezifische Werte
    - schaltet Segmentierung ab (CS und SS:  $[0..2^{32} - 1]$ )
    - sperrt asynchrone Programmunterbrechungen (IRQ)
    - aktiviert Schutzring 0
  - `sysexit`
    - setzt CS und SS auf prozessspezifische Werte
    - setzt EIP/ESP auf die in EDX/ECX stehenden Werte
    - aktiviert Schutzring 3 – nur von Ring 0 aus ausführbar
- das Betriebssystem belegt **modellspezifische Register** der CPU vor
  - MSR (*model-specific register*) 174h, 175h, 176h: CS, ESP und EIP, resp.
    - bei `sysenter`:  $SS = MSR[174h] + 8$
    - bei `sysexit`:  $CS = MSR[174h] + 16$ ,  $SS = MSR[174h] + 24$
  - mit  $MSR[174h]$  als eine Art „Basisindexregister“ in die Segmenttabelle
- Kontextsicherung liegt komplett in Hand des Benutzerprozesses. . .
- alternativ: `syscall/sysret` (ursprünglich AMD; aber auch Intel 64)



- Prozessorregistersatz
  - im Regelfall durch Sicherung und Wiederherstellung von Registerinhalten



- Prozessorregistersatz
  - im Regelfall durch Sicherung und Wiederherstellung von Registerinhalten
    - etwa der Stapelzeiger bei IA-32 [2]: Tupel (SS, ESP) sichern<sup>6</sup>
    - Statusregister und Befehlszeiger (*program counter*) sichern
    - alle, nur flüchtige oder wirklich verwendete Arbeitsregister sichern [4]
  - ↪ dazu den Stapelspeicher des Betriebssystemkerns nutzen  $\leadsto$  Stapelwechsel

<sup>6</sup>Ausnahme *real-address mode*.



- Prozessorregistersatz
  - im Regelfall durch Sicherung und Wiederherstellung von Registerinhalten
    - etwa der Stapelzeiger bei IA-32 [2]: Tupel (SS, ESP) sichern<sup>6</sup>
    - Statusregister und Befehlszeiger (*program counter*) sichern
    - alle, nur flüchtige oder wirklich verwendete Arbeitsregister sichern [4]
    - ↪ dazu den Stapelspeicher des Betriebssystemkerns nutzen  $\leadsto$  Stapelwechsel
  - verschiedentlich auch (zusätzlich) durch Spiegelung einzelner Register
    - etwa der Stapelzeiger beim MC68020: A7  $\Leftrightarrow$  SP und USP [3]

<sup>6</sup>Ausnahme *real-address mode*.



- Prozessorregistersatz
  - im Regelfall durch Sicherung und Wiederherstellung von Registerinhalten
    - etwa der Stapelzeiger bei IA-32 [2]: Tupel (SS, ESP) sichern<sup>6</sup>
    - Statusregister und Befehlszeiger (*program counter*) sichern
    - alle, nur flüchtige oder wirklich verwendete Arbeitsregister sichern [4]
    - ↪ dazu den Stapelspeicher des Betriebssystemkerns nutzen  $\leadsto$  Stapelwechsel
  - verschiedentlich auch (zusätzlich) durch Spiegelung einzelner Register
    - etwa der Stapelzeiger beim MC68020: A7  $\Leftrightarrow$  SP und USP [3]
- Stapelspeicher
  - dem Systemaufruf einen Stapel für den Betriebssystemkern zuteilen
    - ↪ logische Konsequenz, wenn der Betriebssystemadressraum abgeschottet ist

<sup>6</sup>Ausnahme *real-address mode*.



- Prozessorregistersatz
  - im Regelfall durch Sicherung und Wiederherstellung von Registerinhalten
    - etwa der Stapelzeiger bei IA-32 [2]: Tupel (SS, ESP) sichern<sup>6</sup>
    - Statusregister und Befehlszeiger (*program counter*) sichern
    - alle, nur flüchtige oder wirklich verwendete Arbeitsregister sichern [4]
    - ↪ dazu den Stapelspeicher des Betriebssystemkerns nutzen  $\leadsto$  Stapelwechsel
  - verschiedentlich auch (zusätzlich) durch Spiegelung einzelner Register
    - etwa der Stapelzeiger beim MC68020: A7  $\Leftrightarrow$  SP und USP [3]
- Stapelspeicher
  - dem Systemaufruf einen Stapel für den Betriebssystemkern zuteilen
    - ↪ logische Konsequenz, wenn der Betriebssystemadressraum abgeschottet ist
    - einen Stapel im Betriebssystem für alle Kernfäden im Maschinenprogramm
      - ↪ typisch für ereignisbasierte Kerne ( $N : 1$ )
    - einen Stapel im Betriebssystem pro Kernfaden im Maschinenprogramm
      - ↪ typisch für prozessbasierte Kerne ( $1 : 1$ )

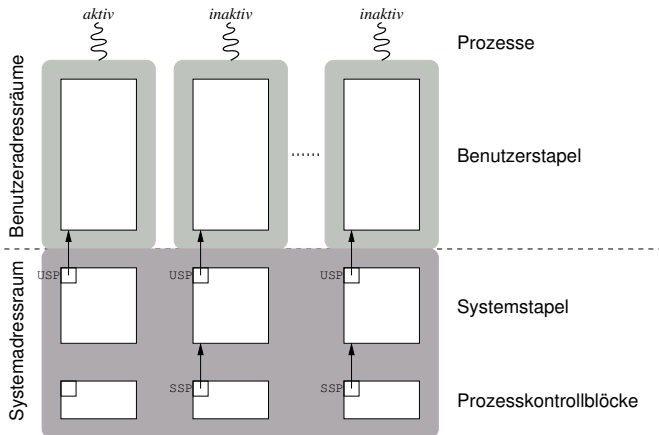
<sup>6</sup>Ausnahme *real-address mode*.



- Prozessorregistersatz
  - im Regelfall durch Sicherung und Wiederherstellung von Registerinhalten
    - etwa der Stapelzeiger bei IA-32 [2]: Tupel (SS, ESP) sichern<sup>6</sup>
    - Statusregister und Befehlszeiger (*program counter*) sichern
    - alle, nur flüchtige oder wirklich verwendete Arbeitsregister sichern [4]
    - ↪ dazu den Stapelspeicher des Betriebssystemkerns nutzen  $\leadsto$  Stapelwechsel
  - verschiedentlich auch (zusätzlich) durch Spiegelung einzelner Register
    - etwa der Stapelzeiger beim MC68020: A7  $\Leftrightarrow$  SP und USP [3]
- Stapelspeicher
  - dem Systemaufruf einen Stapel für den Betriebssystemkern zuteilen
    - ↪ logische Konsequenz, wenn der Betriebssystemadressraum abgeschottet ist
    - einen Stapel im Betriebssystem für alle Kernfäden im Maschinenprogramm
      - ↪ typisch für ereignisbasierte Kerne ( $N : 1$ )
    - einen Stapel im Betriebssystem pro Kernfaden im Maschinenprogramm
      - ↪ typisch für prozessbasierte Kerne ( $1 : 1$ )
  - ähnlich wird (oft) bei asynchronen Programmunterbrechungen verfahren

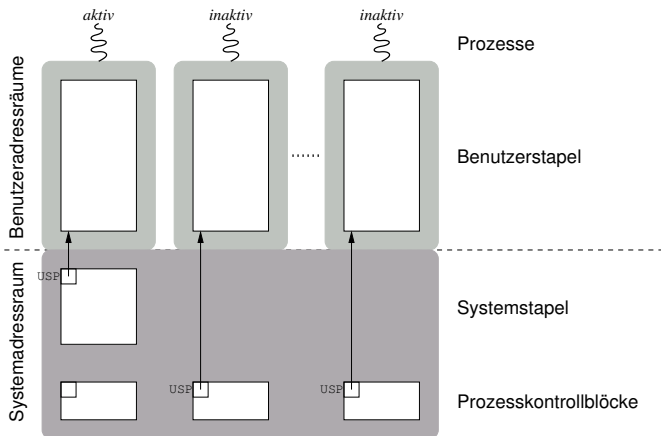
<sup>6</sup>Ausnahme *real-address mode*.





- Prozessverdrängung/-blockierung im Kern ist (fast) überall möglich





- Prozessverdrängung/-blockierung im Kern ist bedingt möglich [1]



## Rekapitulation

- Mehrebenenmaschinen

- Teilinterpretierung

## Funktionale Hierarchie

- Analogie

- Abstraktion

## Implementierung

- Entvirtualisierung

- Befehlsarten

- Ablaufkontext

## Zusammenfassung



- Rekapitulation
  - Maschinenprogramme werden durch Betriebssysteme teilinterpretiert
  - Teilinterpretierung wird (insb. auch) durch Systemaufrufe ausgelöst
- funktionale Hierarchie
  - Systemaufrufstümpfe trennen Maschinenprogramm von Betriebssystem
  - im Betriebssystem aktiviert ein Systemaufrufzuteiler die Systemfunktionen
  - der Systemaufruf ist ein adressraumübergreifender Prozeduraufruf
- Implementierung
  - ein Systemaufruf ist als Primitiv- oder Komplexbefehl realisiert
  - Primitivbefehle nutzen (ausschließlich) Register zur Parameterübergabe
  - Komplexbefehle erlauben einen unverfälschten Zustandsabzug
  - Fehler werden durch spezielle Rückgabewerte oder Merker signalisiert
  - einem Systemaufruf ist ein Betriebssystemstapel 1 : 1 oder  $N : 1$  zugeteilt



## ■ Rekapitulation

- Maschinenprogramme werden durch Betriebssysteme teilinterpretiert
- Teilinterpretierung wird (insb. auch) durch Systemaufrufe ausgelöst

## ■ funktionale Hierarchie

- Systemaufrufstümpfe trennen Maschinenprogramm von Betriebssystem
- im Betriebssystem aktiviert ein Systemaufrufzuteiler die Systemfunktionen
- der Systemaufruf ist ein adressraumübergreifender Prozeduraufruf

## ■ Implementierung

- ein Systemaufruf ist als Primitiv- oder Komplexbefehl realisiert
- Primitivbefehle nutzen (ausschließlich) Register zur Parameterübergabe
- Komplexbefehle erlauben einen unverfälschten Zustandsabzug
- Fehler werden durch spezielle Rückgabewerte oder Merker signalisiert
- einem Systemaufruf ist ein Betriebssystemstapel 1 : 1 oder  $N : 1$  zugeteilt



- Rekapitulation
  - Maschinenprogramme werden durch Betriebssysteme teilinterpretiert
  - Teilinterpretierung wird (insb. auch) durch Systemaufrufe ausgelöst
- funktionale Hierarchie
  - Systemaufrufstümpfe trennen Maschinenprogramm von Betriebssystem
  - im Betriebssystem aktiviert ein Systemaufrufzuteiler die Systemfunktionen
  - der Systemaufruf ist ein adressraumübergreifender Prozeduraufruf
- Implementierung
  - ein Systemaufruf ist als Primitiv- oder Komplexbefehl realisiert
  - Primitivbefehle nutzen (ausschließlich) Register zur Parameterübergabe
  - Komplexbefehle erlauben einen unverfälschten Zustandsabzug
  - Fehler werden durch spezielle Rückgabewerte oder Merker signalisiert
  - einem Systemaufruf ist ein Betriebssystemstapel 1 : 1 oder  $N : 1$  zugeteilt



- Rekapitulation
  - Maschinenprogramme werden durch Betriebssysteme teilinterpretiert
  - Teilinterpretierung wird (insb. auch) durch Systemaufrufe ausgelöst
- funktionale Hierarchie
  - Systemaufrufstümpfe trennen Maschinenprogramm von Betriebssystem
  - im Betriebssystem aktiviert ein Systemaufrufzuteiler die Systemfunktionen
  - der Systemaufruf ist ein adressraumübergreifender Prozeduraufruf
- Implementierung
  - ein Systemaufruf ist als Primitiv- oder Komplexbefehl realisiert
  - Primitivbefehle nutzen (ausschließlich) Register zur Parameterübergabe
  - Komplexbefehle erlauben einen unverfälschten Zustandsabzug
  - Fehler werden durch spezielle Rückgabewerte oder Merker signalisiert
  - einem Systemaufruf ist ein Betriebssystemstapel 1 : 1 oder  $N : 1$  zugeteilt

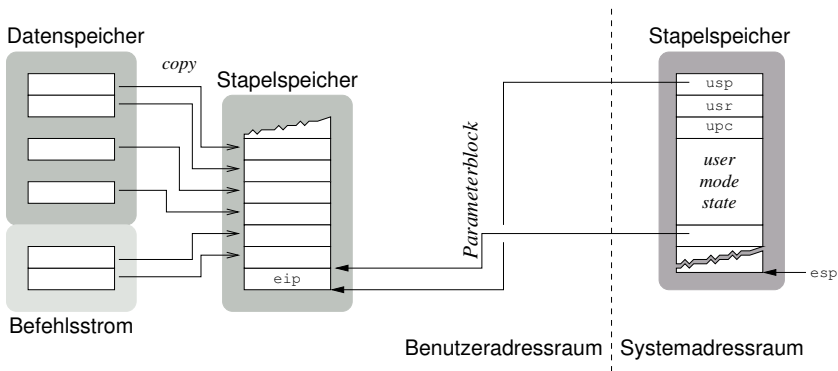


- [1] DRAVES, R. ; BERSHAD, B. N. ; RASHID, R. F. ; DEAN, R. W.:  
Using Continuations to Implement Thread Management and Communication in Operating Systems.  
In: *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP 1991)*, ACM Press, 1991. – ISBN 0-89791-447-3, S. 122-136
- [2] INTEL CORPORATION (Hrsg.):  
*Intel 64 and IA-32 Architectures: Software Developer's Manual*.  
Order Number: 325462-045US.  
Santa Clara, California, USA: Intel Corporation, Jan. 2013
- [3] MOTOROLA SEMICONDUCTOR PRODUCTS INC. (Hrsg.):  
*MC68020-MC68EC02009E Microprocessors User's Manual*.  
First Edition.  
Phoenix, Arizona, USA: Motorola Semiconductor Products Inc., 1992
- [4] SCHRÖDER-PREIKSCHAT, W. ; KLEINÖDER, J. :  
*Systemprogrammierung*.  
[http://www4.informatik.uni-erlangen.de/Lehre/WS08/V\\_SP](http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_SP), 2008 ff.



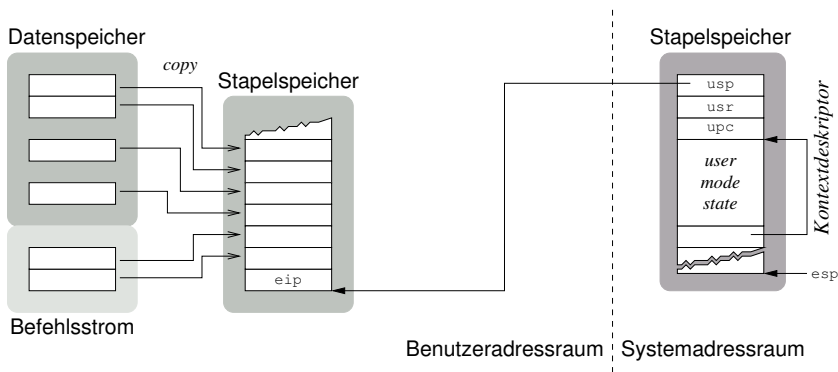
- [5] TANENBAUM, A. S.:  
**Multilevel Machines.**  
In: *Structured Computer Organization.*  
Prentice-Hall, Inc., 1979. –  
ISBN 0-130-95990-1, Kapitel 7, S. 344-386





- die Systemfunktion lädt Parameter direkt vom Benutzerstapel
  - indirekte Adressierung durch einen Zeiger auf den Parameterblock
  - Verzicht auf Ortstransparenz in der Systemfunktion
- der Prozessorstatus ist komplett betriebssystemseitig gesichert





- Systemaufrufparameter indirekt über einen Kontextdeskriptor laden
  - den Parameterblock vom Benutzerstapelzeiger ableiten
  - unterstützt insb. die merkerbasierte Signalisierung von Fehlercodes
- Offenlegung des durch die CPU gesicherten Prozessorzustands

