2 Systemdienste

Beachte:	Systemaufrufe sind sprachunabhängig. Ihre Beschreibung bezieht sich daher auf das „Typsystem“ der Hardware, d.h. sie ist so gut wie typenlos.

Wiederholungsempfehlung: Rechnerorganisation

Systemdienste

- werden für die Benutzerprogramme an der *Systemschnittstelle sprachunabhängig* bereitgestellt,
- werden angefordert über **Systemaufrufe** *(system calls)*

Somit kann ein **Benutzerprozess** *(user process)* verfügen über

- einen virtuellen **Adressraum**,
- die **Maschinenbefehle** (außer den *privilegierten Befehlen*)
- die **Systemdienste** (darunter **I**nter**P**rozess**K**ommunikation)
- eventuell Dienste anderer Prozesse (über IPK)

Technische Realisierung von Systemaufrufen – 4 Alternativen:

(monolithischer Kern:)

❶ **Unterprogrammsprung** ins Betriebssystem

❷ **Maschinenbefehl „Systemaufruf“**

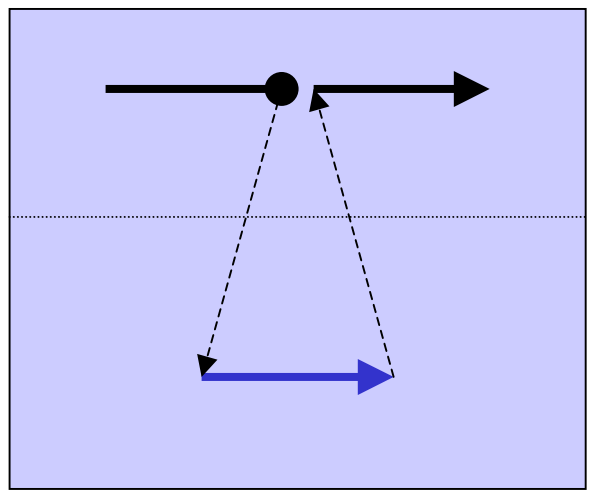
(Mikrokern:)

❸ **Aufruf** eines Systemmoduls/objekts

❹ **Auftragserteilung** an einen Systemprozess

❶ **Unterprogrammsprung** ins Betriebssystem:

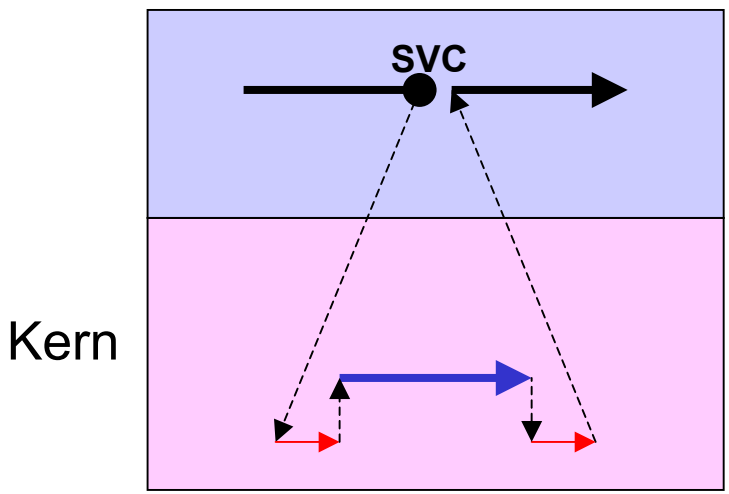
bei sehr einfachen Systemen ohne getrennte Adressräume.
Übersetzer/Binder/Lader setzt Sprungadressen ein.
Abschließend Rücksprung.



❷ **Maschinenbefehl „Systemaufruf“** *(supervisor call, SVC):*

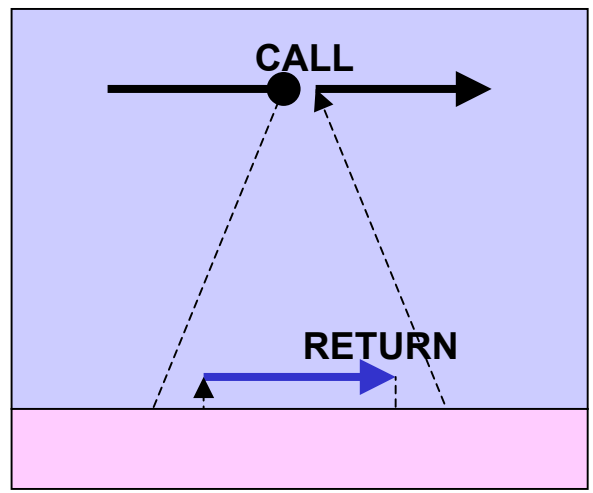
löst **Alarm** *(trap, exception)* aus. Unterbrechungsbehandlung erkennt Unterbrechungsursache „Systemaufruf“ und verzweigt über Sprungtabelle zur gewünschten Dienstroutine. Parametrisierung des Systemaufrufs direkt durch den Übersetzer oder durch zwischengeschaltete Bibliotheks-routine. Abschließend „Rücksprung“.

Beispiel: klassiches Unix



❸ **Aufruf** eines Systemmoduls/objekts:

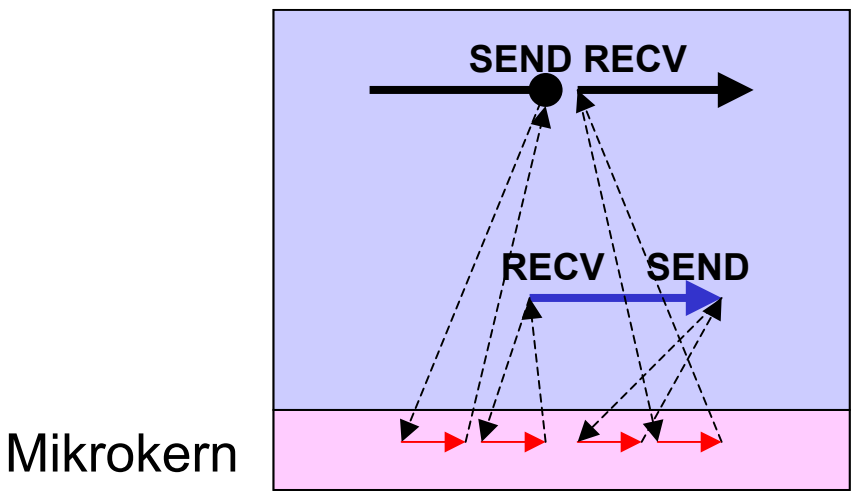
Als Reaktion auf den Alarm CALL vermittelt der Mikrokern Eintritt in den Adressraum des gewünschten Systemmoduls. Rückkehr in den aufrufenden Adressraum über RETURN.



❹ **Auftragserteilung** an Systemprozess

Als Reaktion auf den Alarm SEND besorgt der Mikrokern eine Auftragserteilung an den gewünschten Systemprozess. Dieser nimmt den Auftrag über RECV entgegen. (Entsprechend gegebenenfalls für Ergebnislieferung.)

Beispiele: Mach, Minix



Programmieren in hardwarenaher Sprache – z.B Assembler oder C – mit direkter Benutzung der Systemaufrufe
wird manchmal (unpräzise) *Systemprogrammierung* genannt

Beispiel Unix und C:

C-Bibliotheken bieten umfangreiches
application programming interface (API),
darunter die *Systembibliothek*
(Online-Handbuch: **`man -s2`** name)

- Systembibliothek *verbirgt die eigentlichen Systemaufrufe*, z.B. verbirgt sich hinter **`write(...)`**

```
        PUSH   EBX              ; EBX retten
        MOV    EBX,  8(ESP)     ; 1. Parameter
        MOV    ECX, 12(ESP)     ; 2. Parameter
        MOV    EDX, 16(ESP)     ; 3. Parameter
        MOV    EAX, 4           ; 4 steht für „write“
        INT    0x80             ; eigentlicher Systemaufruf
        JBE    DONE             ; kein Fehler
        NEG    EAX              ; Fehlercode
        MOV    errno, EAX
        MOV    EAX, -1
DONE:   POP    EBX              ; EBX wiederherstellen
        RET                     ; Rücksprung
```

(Intel IA-32 Assembler)

- *Fehler* bei Systemaufruf wird angezeigt durch den Ergebniswert **`-1`** ;

die *Fehlerart* findet man dann in der Variablen **`extern`** `int errno` ;

mnemonische Bezeichnungen der Fehlerarten sind in der Datei **`errno.h`** zusammengestellt, z.B. `EIO` *(I/O error)* für Ein/Ausgabe-Fehler.

(z.B. Solaris: **`/usr/include/sys/errno.h`**)

Beispiel:

```
extern int errno;

main() {
      int written = 0;
      written = write(1,"hello!\n",7);
      if(written == -1)
            exit(errno);
      else exit(0);
}
```

2.1 Prozessverwaltung

(am Beispiel *Unix*)

umfasst Systemaufrufe zum

Erzeugen, Beenden, Abfragen, ... von Prozessen

Prozess wird identifiziert über *Prozessnummer (process id, pid)*

(*Beachte:* Mehrprozessbetrieb erfordert *nicht notwendig*
solche Aufrufe – einfache Betriebssysteme
kommen auch *ohne* sie aus!)

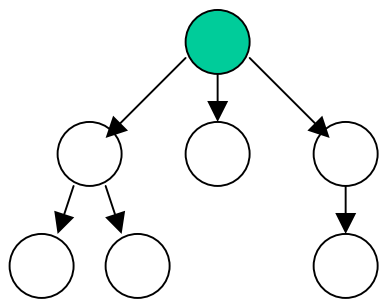
Prozessmodell:

- ☛ Anmelden am System führt zur Einrichtung eines Benutzerprozesses, der typischerweise (aber nicht notwendigerweise) den **Befehlsinterpretierer** *(command interpreter, shell)* ausführt

Prozessmodell:

- Anmelden am System führt zur Einrichtung eines Benutzerprozesses, der typischerweise (aber nicht notwendigerweise) den **Befehlsinterpretierer** *(command interpreter, shell)* ausführt.

- Jeder Prozess kann weitere Prozesse erzeugen; somit entsteht nach dem Anmelden ein Baum von Prozessen, genannt **Prozessgruppe** *(process group),* identifizierbar über eine *Prozeßgruppennummer*.



- Prozess ist *schwergewichtig*, d.h. ist virtueller Rechner mit
  - virtuellem Prozessor
  - virtuellem Adressraum,
  - virtueller Peripherie.

Im Gegensatz zum realen kennt der *virtuelle Prozessor*

ausschließlich die **nichtprivilegierten Befehle** *(non-privileged, user-mode instructions),*

ausschließlich die **allgemeinen Register** (d.h. nicht Programmstatus, Adressumsetzer *(MMU)* etc.),

nicht alle Unterbrechungen *(interrupts):* zwar die *Alarme (traps),* aber statt der *Eingriffe (interrupts)* andersartige **Software-Eingriffe** *(software interrupts).*

2.1.1 Erzeugen und Beenden von Prozessen

(*Unix*-ähnliche Systeme gemäß *POSIX*-Standard:)

`fork()` *erzeugt Kopie* des laufenden Prozesses;
Erzeuger *(parent process)* erhält als Ergebnis
die Prozeßnummer des erzeugten
Kindprozesses *(child process),*
und dieser erhält **`0`** als Ergebnis.

Fehler **`EAGAIN`**, wenn zu viele Prozesse.

(*online manual* [Abschnitt 2]: Befehl **`man fork`** liefert Details,
insbesondere zu weiteren Fehlern!)

`exit(status)`

beendet den laufenden Prozess mit
dem angegebenen *Endestatus (return status,* 1 Byte)
(normal: `0`; Fehlerfall: `!=0`)

Fehler: keine

`wait(&status)` [siehe auch **`waitpid(...)`**]

wartet auf das Beenden eines Kindprozesses, löscht
diesen Prozess und liefert seine Nummer als Ergebnis;
liefert in **`status`**: (Endestatus,Abbruchstatus [s.u.])

Fehler **`ECHILD`**, wenn keine Kinder mehr

Beispiel: Erledigung einer Aufgabe durch Aufteilung in zwei unabhängige Aktivitäten:

```
if(fork() != 0) {
        stat = parentComputation();
        wait(&status);
        if(stat || status) stat = 1;
        exit(stat); }
else { stat = childComputation();
        exit(stat); }
```

2.1.2 Abfragen von Prozesseigenschaften

`getpid()`

liefert Nummer des laufenden Prozesses

`getpgrp()`

liefert Nummer der Prozessgruppe
des laufenden Prozesses

`times(&buffer)`

liefert diverse Angaben über den Rechenzeitverbrauch
des laufenden Prozesses und seiner Kinder

2.1.3 Unterbrechen/Abbrechen von Prozessen

Unterbrechungen beim virtuellen Prozessor – 3 Varianten:

❶ Es gibt keine.

Hardware-*Eingriffe* bleiben verborgen;

Alarm
- bleibt verborgen (z.B. Seitenfehler) oder
- bewirkt Systemaufruf oder
- führt zum Prozessabbruch(z.B. Arithmetikfehler)

❷ *Manche Alarme* können auftreten und sollten vom Prozess geeignet behandelt werden.

- Hardwaremäßig *vorgegebene* Alarmadressen im Adressraum des Prozesses *oder*
- *Systemaufruf* für Festlegung der Alarmadressen.

❸ (So bei Unix:) *Zusätzlich* zu ❷ gibt es **Software-Unterbrechungen** *(software interrupts):*

- Betriebssystem definiert verschiedene Arten
- Systemaufruf für Eingriff in anderen Prozess
- Systemaufruf für Festlegung der Adressen der Behandlungsroutinen aller Unterbrechungen

Die dem Prozess bekannten Unterbrechungen
(Alarme und Software-Unterbrechungen) heißen bei Unix

Signale *(signals)*

(durchnumeriert, mnemonisch in `signal.h`)

Z.B. Alarme:

(Hardware:)	`SIGSEGV`	*segmentation violation* ungültige Adresse
(Hardware:)	`SIGILL`	*illegal instruction* ungültige Instruktion
(!Software:)	`SIGPIPE`	*pipe error* Pipe hat keinem Empfänger

Z.B. Eingriffe

von anderen Prozessen:

`SIGTERM` *terminate*
Aufforderung zum Beenden

`SIGKILL` *kill*
Abbruch

sonstige Eingriffe:

`SIGINT` *interrupt*
Taste **`^C`** , wirkt auf gesamte Prozessgruppe

`SIGALRM` *alarm*
der Wecker ist abgelaufen
(nachdem er zuvor mit **`alarm`**`(sec)` – s.u. – gestellt worden war)

Systemaufrufe für Software-Unterbrechungen:

`kill``(pid,sig)`

schickt Signal `sig` an den Prozess `pid`

(bzw. im Fall `pid==0` an alle Prozesse

der Prozessgruppe des laufenden Prozesses)

Fehler: `ESRCH` – `pid` ist ungültig

`EINVAL` – `sig` ist ungültig

`alarm``(seconds)`

schickt Signal `SIGALRM` an den laufenden Prozess selbst – nach einer Verzögerung von `seconds` Sekunden

`pause``()`

wartet auf ein Signal – und liefert nach Abschluss der Signalbehandlung einen

„*Fehler*“ `EINTR`

`signal``(sig,&handler)` [siehe auch **`sigaction`**`(...)`]

vereinbart, daß Signale `sig` durch die Prozedur `handler` behandelt werden (und liefert als Ergebnis die zuvor vereinbarte Behandlungsroutine).

Fehler: `EINVAL` – `sig` ist ungültig

Signale unterdrücken und wieder zulassen über *Signalmaske*,

siehe z.B. Solaris **`man signal`**