1 Verwaltungsdaten pro Prozeß (2)

◆ der Speicherbereich der *user area* des gerade aktiven Prozesses

➤ wird an eine feste Adresse des virtuellen Adreßraums des Systemkerns abgebildet

➥ Zugriff auf die *user area* erfolgt in den Kern-Funktionen über die Struktur **u** (z. B. ***u.u_ofile***)

➥ bei einem Prozeßwechsel wird die *user area* des neuen Prozesses auf den Speicherbereich der Struktur **u** abgebildet

➥ bei jedem Prozeßwechsel muß die Abbildung virtuelle Adressen → physikalische Adressen geändert werden

➤ oder wird über einen festen Zeiger im Systemkern adressiert

➥ Zugriff auf die *user area* erfolgt in den Kern-Funktionen über den Zeiger **uunix** (z. B. ***uunix->u_ofile***)

➥ bei einem Prozeßwechsel wird der Zeiger **uunix** auf die neue *user area* gelegt

➥ jeder Zugriff auf Elemente der *u-area* erfolgt indirekt

1 Verwaltungsdaten pro Prozeß (3)

■ Kernel stack

◆ Stack für Funktionen des Systemkerns, die im Kontext des Prozesses ablaufen (zur Abwicklung von Systemaufrufen)

◆ bei Prozeßwechsel wird der Kernel Stack des neuen Prozesses entsprechend in den virt. Adreßraum des Kerns gemapped

2 Globale Prozeßverwaltungsdaten

■ Prozeßtabelle

◆ alle wichtigen Verwaltungsdaten über einen Prozeß, die immer verfügbar sein müssen, werden in

***proc*-Strukturen**

gehalten

◆ die *proc*-Strukturen bilden die **Prozeßtabelle**
(bei Booten statisch als Feld angelegt oder dynamisch als Liste verwaltet)

2 Globale Prozeßverwaltungsdaten (2)

■ … Prozeßtabelle

◆ eine *proc*-Struktur enthält u. a. folgende Daten:

➤ Prozeßzustand (laufend/bereit, blockiert, ausgelagert, ...)

➤ Verweise auf die Segmente des Prozesses (meist über *page table*)

➤ Verweis auf die *user area*

➤ Verweis auf die *credential*-Struktur mit den Zugriffsrechten (*User-Id*s, Gruppen-Rechte)

➤ Speicher für Daten zur Signalverarbeitung

➤ Scheduling-Parameter (z. B. Priorität)

➤ Information über Ereignis, auf das der Prozeß wartet

◆ das Kommando ***ps(1)***, gibt Auszüge der Prozeßtabelle aus

2 Globale Prozeßverwaltungsdaten (3)

■ Prozeßidentifikatoren und Prozeßgruppen

pid ***process id*** - eindeutiger Prozeßidentifikator
($0 \le pid \le 30000$) ***getpid(2)***

ppid ***parent process id*** = *pid* des "Vaterprozesses"
getppid(2)

pgrp ***process group id*** = *pid* des Prozeßgruppenführers
setpgrp(2), getpgrp(2), ***setpgid(2)*** (POSIX)
(siehe auch Abschnitt "Sessions und Prozeßgruppen")

real user id (ruid) und ***real group id (rgid)***
uid und *gid* des Benutzers, der den Prozeß erzeugt hat
(festgelegt durch *login* des Benutzers)
setuid( ), getuid( ), setgid( ), getgid( )

2 Globale Prozeßverwaltungsdaten (4)

- … Prozeßidentifikatoren und Prozeßgruppen

effective uid (euid), effective gid (egid), access groups
uid und *group ids,* die während des Programmablaufs zum Prüfen der Zugriffsrechte auf Systemressourcen verwendet werden

– *euid* und *egid* entsprechen normalerweise *ruid* und *rgid*

– *access groups* ist eine Menge zusätzlicher *group ids*, deren Rechte der Prozeß zusätzlich besitzt (BSD)

– beim Laden eines Programms mit *user-s-Bit* wird die *euid* auf die *uid* des Dateibesitzers gesetzt,
analog für *egid*

– ***seteuid( ), geteuid( ), setegid( ), getegid( )***

2 Globale Prozeßverwaltungsdaten (5)

- Sessions und Prozeßgruppen
(POSIX 1003.1 legt Standard für Prozeßgruppen und Job-Control fest)

◆ Prozesse werden in **Sessions** und **Prozeßgruppen** gruppiert

◆ Eine **Session** entspricht i. a. einer Login-Sitzung oder einem Fester in einer Window-Umgebung

➤ eine neue *session* wird mit ***setsid(2)*** erzeugt, der aufrufende Prozeß wird dadurch ***session leader***

➤ der *session leader* wird gleichzeitig Führer einer eigenen Prozeßgruppe

➤ ein *session leader* kann ein (noch keiner *session* zugeordnetes) Terminal als ***controlling terminal*** anfordern,
er wird dadurch ***controlling process*** des Terminals

➤ *session*- und Prozeßgruppen-Mitgliedschaft und damit auch das *controlling terminal* werden an Sohnprozesse vererbt

➤ durch *setsid(2)* verliert ein Prozeß sein bisheriges *controlling terminal*

2 Globale Prozeßverwaltungsdaten (6)

- … Sessions und Prozeßgruppen

◆ Eine **Prozeßgruppe** ist eine Menge von Prozessen einer *session*

➤ mit ***setpgid(2)*** kann ein Prozeß die Zuordnung zu einer Prozeßgruppe für sich oder einen seiner Sohnprozesse verändern

– Prozeßgruppe, Aufrufer und betroffener Prozeß müssen zu der selben *session* gehören

➤ dem *controlling tty* einer *session* kann eine Prozeßgruppe der *session* zugeordnet werden (***tcsetpgrp( )***)

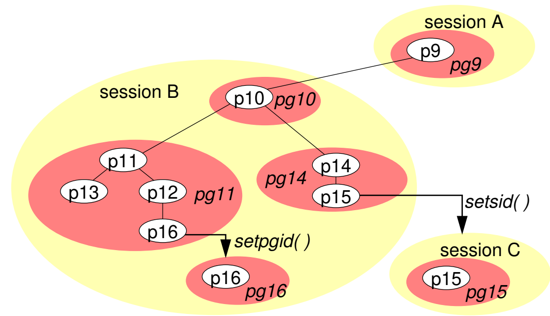
– die Prozesse dieser Prozeßgruppe laufen im **Vordergrund**

– die Prozesse aller anderen Prozeßgruppen der *session* laufen im **Hintergrund**

– Signale (*Interrupt, Stop, Hangup*) vom Terminal-Treiber werden allen Prozessen dieser Prozeßgruppe zugestellt

2 Globale Prozeßverwaltungsdaten (7)

- … Sessions und Prozeßgruppen



3 Kommando ps


```
 F S   UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN TTY       TIME CMD
19 T     0     0     0  0   0 SY 10416f88      0          ?         0:00 sched
 8 S     0     1     0  0  41 20 604df608     85 604df800 ?         0:03 init
19 S     0     2     0  0   0 SY 604def48      0 10432c34 ?         0:00 pageout
19 S     0     3     0  0   0 SY 604de888      0 10435cdc ?        34:37 fsflush
 8 S     0   239     1  0  51 20 604dd448    253 604dd640 ?         0:11 sendmail
 8 S 10402  3705     1  1  51 20 608c3620   4808 6098490e console 28:16 maker5X.
 8 S     0   122     1  0  40 20 60693610    257 600a8b8e ?         0:15 rpcbind
 8 S     0   124     1  0  94 20 60692f50    249 600a8aee ?         0:00 keyserv
 8 S     0   135     1  0  41 20 60691b10    231 600a8bb6 ?         0:00 ypbind
 8 S     0   179     1  0  41 20 60691450    479 609848e6 ?         8:06 automoun
 8 S     0   238     1  0  40  8 606906d0    230 606908c8 ?         1:01 xntpd
 8 S     0   201     1  0  51 20 60759b18    206 6002deb8 ?         0:02 cron
 8 S 10402  1776     1  0  50 20 609ea028    396 60985ade console  0:01 xterm
 8 S     0   269     1  0  41 20 60758018    210 604fe060 ?         0:12 sshd
 8 S 10402  3579     1  0  58 20 608c1b20    159 608c1d18 console  0:00 csh
 8 S     0   313     1  0  40 20 608c1460    726 600a8206 ?         0:00 dtlogin
 8 S 10402  3638  3629  0  40 20 60aac8b8    180 60a680be pts/3     0:01 elm
 8 S 10402  3597  3579  0  77 20 608c21e0    121 608c2250 console  0:00 openwin
 8 O     0  8150  3633  0  41 20 609ec1e8    114          pts/2     0:00 ps.exe
 8 S 10402  3601  3597  0  51 20 60aaadb8    331 60aaae28 console  0:00 xinit
 8 S 10402  3693  3668  0  51 20 60a921f0   1963 609853fe console  0:00 netscape
 8 S 10402  1777  1776  0  40 20 609ea6e8    188 60ade4c6 pts/4     0:00 rlogin
 8 S 10402  3613  3603  0  40 20 60a90db0    381 600a813e console  0:45 tvtwm
 8 S 10402 27875     1  0  51 20 60758d98    399 60985ba6 console  0:18 xterm
 8 S 10402  3603  3601  0  49 20 60aab478    121 60aab4e8 console  0:00 .xinitrc
 8 S 10402  3669     1  0  50 20 608c2f60    708 6098585e console  0:04 cm
 8 Z 10402  3610  3603  0   0                                       0:00 <defunct>
 8 R 10402  3602  3601  2  43 20 60a92f70  23544          console 63:44 Xsun
 8 S 10402  3633  3622  0  41 20 609ecf68    276 609ed160 pts/2     0:02 tcsh-6.0
 8 S 10402  3623  3618  0  40 20 608c28a0    258 60a689be pts/0     0:01 tcsh-6.0
 8 S 10402  3668     1  2  51 20 60b5edc0   4165 600a809e console 125:56 netscape
 8 S 10402  3622     1  0  40 20 60a928b0    401 600a804e console  0:03 xterm
 8 S 10402  3680     1  0  40 20 60b5fb40    373 600a818e console  0:00 xclipboa
 8 S 10402 27876 27875  0  40 20 60b60200    286 60b905de pts/1     3:15 elm
 8 T 10402  3799  3633  0  52 20 60b5e700    188          pts/2     0:00 rlogin
 8 Z 10402  3614  3613  0   0                                       0:00 <defunct>
 8 S 10402  3019     1  0  50 20 60a91b30    397 6098463e console  0:01 xterm
```


F.4 UNIX - Erzeugen eines neuen Prozesses

1 Überblick

- jeder UNIX-Prozeß kann mit dem Systemaufruf ***fork(2)*** einen neuen Prozeß erzeugen
- *fork( )* erzeugt eine nahezu identische Kopie des aufrufenden Prozesses
- der *fork( )* aufrufende Prozeß wird **Vaterprozeß *(parent process)*** genannt
- der durch *fork( )* neu erzeugte Prozeß wird als **Sohnprozeß *(child process)*** bezeichnet
- der Sohnprozeß erbt alle Rechte und Einschränkungen vom Vaterprozeß (einschließlich aller Daten der *user area*, z. B. offene Dateien)

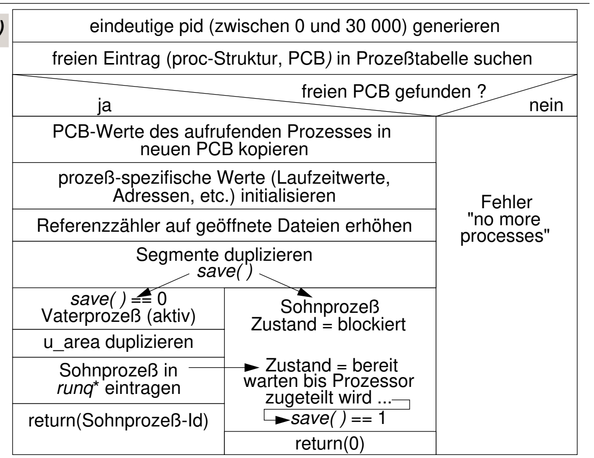
1 Überblick (2)

- wesentliche Unterschiede zwischen Vater- und Sohnprozeß:

fork-Ergebnis:	Vaterprozeß: *pid* des Sohnes, Sohnprozeß: 0
PID:	Sohnprozeß erhält nächste freie *pid*
Eintrag in Prozeßtabelle:	Sohnprozeß erhält eigene proc-Structur, Inhalt wird weitgehend vom Vaterprozeß kopiert

2 Ablauf

newproc( )



*) *runq*: Warteschlange von bereiten Prozesse in UNIX

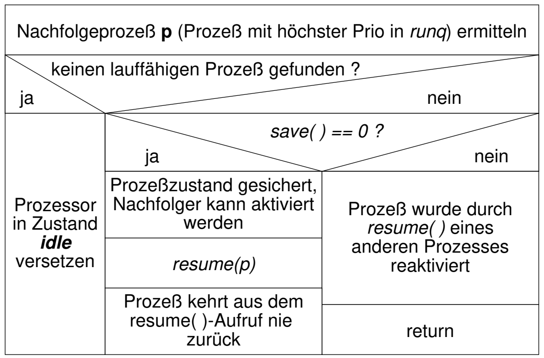
F.5 UNIX — Prozeßumschaltungen

1 Überblick

- Prozeßumschaltungen erfolgen nur an wenigen Stellen im Systemkern:
  - ➥ wenn sich ein Prozeß blockiert (***sleep( )***)
    - – Prozeß aktiviert höchstprioren, lauffähigen Prozeß
    - – wenn das Ereignis, auf das gewartet wird, eintritt, wird der Prozeß durch ***wakeup( )*** in die Warteschlange der bereiten Prozesse (***runq***) eingetragen und kann reaktiviert werden
  - ➥ vor Rückkehr aus einem Systemaufruf, wenn ein höherpriorer Prozeß bereitsteht
    - – Prozeß trägt sich in die ***runq*** ein
    - – Prozeß aktiviert höchstprioren, lauffähigen Prozeß
- die Prozeßumschaltung erfolgt in den Funktionen ***swtch( )***, ***save( )*** und ***resume( )*** des Systemkerns (stark abhängig von Hardware und UNIX-Version!)

2 Ablauf (Beispiel aus XELOS)

swtch( )



3 Ablauf (2)

save( )

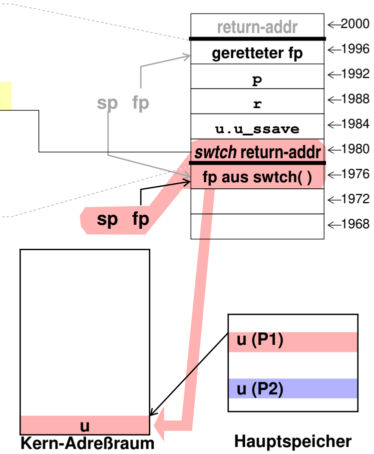
Register, Frame-Pointer, gesicherten Frame-Pointer von *swtch( )* und Rücksprungadresse nach *swtch( )* in ***u-area*** sichern
return(0)

resume( )

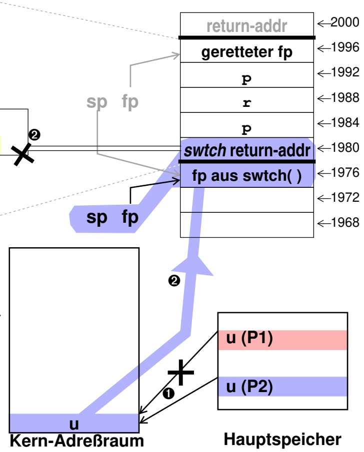
u-area des Nachfolgeprozesses auf **u** mappen Kernel-Stack des Nachfolgeprozesses mappen (= eigentliche Prozeßumschaltung)
Register, Frame-Pointer, gesicherten Frame-Pointer von *swtch( )* und Rücksprungadresse nach *swtch( )* aus ***u-area*** restaurieren
return(1) = return aus früherem *save( )*-Aufruf

3 Ablauf (3)

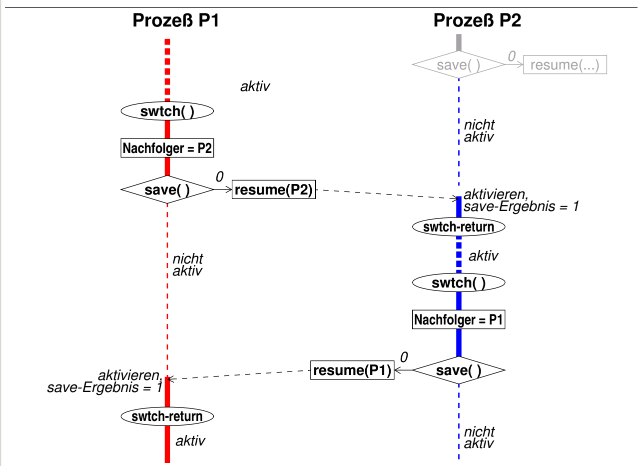
```
swtch() {
    struct proc *p;
    int r;
    p = getrq();

    r = save(u.u_ssave);

    if (r == 0) {
        resume(p);
    } else {
        return(1);
    }
}
save(savearea *s) {
    /* Register,
       sp, fp,
       fp aus swtch und
       swtch return-addr
       -> s sichern */

    return(0);
}
```



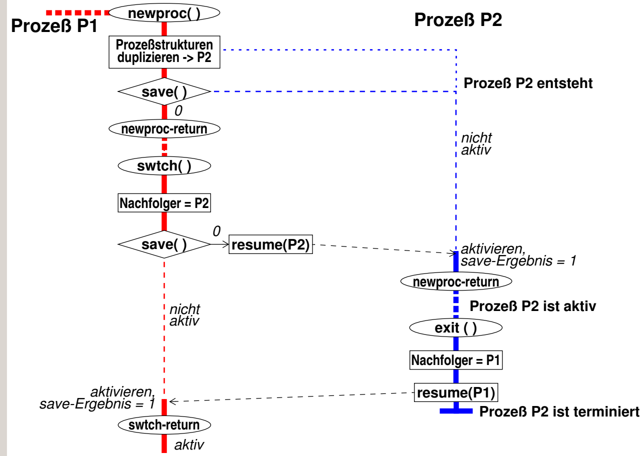
```
swtch() {
     struct proc *p;
     int r;
     p = getrq();

     r = save(u.u_ssave);

     if (r == 0) {
         resume(p);
     } else {
         return(1);
     }
}
resume(struct proc *p) {
     /* u-area und
❶        kernel-stack
         von p mappen */

     /* gesicherte Register,
❷        fp und return-addr
         restaurieren */

     return(1);
}
```



4 Wechsel zwischen zwei Prozessen



5 Erzeugen eines neuen Prozesses



F.6 UNIX — Ausführen eines Programms

- ein UNIX-Prozeß kann die Ausführung eines Programms durch ein anderes Programm ersetzen — ***exec(2), execve(2)***
  - ◆ durch Laden eines neuen Programms werden die zuvor von dem Prozeß bearbeiteten Programm-Datenstrukturen zerstört
    ➥ **im Erfolgsfall gibt es kein return aus *exec( )***
  - ◆ durch Laden eines neuen Programms entsteht **kein neuer Prozeß**
- *execve( )* werden die Argumente ***(argv)*** und ein Environment ***(envp)*** für das neue Programm mitgegeben
- Details:
  - ◆ programmspezifische Daten des Prozesses
    (Segmente, *signal-handler*-Funktionen) werden initialisiert
  - ◆ ***effective uid*** / ***effective gid*** des Prozesses werden auf Eigentümer/Gruppe der Programm-Datei geändert, wenn bei dieser ein ***s-bit*** gesetzt ist

F.6 UNIX — Ausführen eines Programms (2)

- … Details:
  - ◆ alle anderen Prozeßparameter (in *proc*-Struktur und *user area*) bleiben unverändert (*pid, ppid, pgrp, session, controlling tty, current working directory, current root directory, ...*)
  - ◆ **offene Dateideskriptoren bleiben erhalten**, es sei denn, sie wurden als *close-on-exec* (siehe ***fcntl(2)***) markiert
- Programmier-Beispiel:

```
if ( (pid = fork()) < 0 ) {
    perror("fork");
    exit(1)
} else if (pid == 0) {          /* child process */
    execl("/bin/cp", "cp",    "/tmp/a", "/tmp/b", (char
*)0);
    perror("exec");
    exit(1);
} else {                        /* parent process */
              ...
}
```


F.7 UNIX — Terminieren von Prozessen

1 Überblick

- Prozesse terminieren, wenn
  - ◆ sie den Systemdienst ***exit(2)*** aufrufen
  - ◆ ein Signal an den Prozeß zugestellt wurde, für das keine Signalbearbeitungsfunktion vorgesehen ist
- die Startumgebung für C-Programme ruft nach einem *return* aus der Funktion *main* automatisch *exit(0)* auf
- dem *exit*-Aufruf kann ein Status-Wert (1 Byte) mitgegeben werden, der durch den Vaterprozeß abgefragt werden kann (→ *wait(2)*)
- ein Vaterprozeß kann auf das terminieren von Sohnprozessen warten un deren exit-Status abfragen — ***wait(2)***

2 Ablauf von exit im Kern

- ◆ System V:
  wenn der Prozeß ein Prozeßgruppenführer mit zugeordnetem Terminal war (typischerweise *login-shells*), wird ein SIGHUP-Signal an alle Prozesse dieser Prozeßgruppe gesendet
- ◆ POSIX:
  wenn der Prozeß ein *controlling process* (*session leader* mit zugeordnetem *controlling tty*) war, wird allen Prozessen in der Vordergrund-Prozeßgruppe ein SIGHUP-Signal gesendet, das *controlling tty* wird von der *session* abgekoppelt
- ◆ alle offenen Dateideskriptoren werden geschlossen
- ◆ alle Segmente (Text, Daten, Stack) werden freigegeben; *shared memory* wird abgekoppelt, aber nicht gelöscht
- ◆ *user area* (enthält Kernel-Stack auf dem exit operiert!) und *proc*-Struktur bleiben noch erhalten
- ◆ alle noch existierenden Sohnprozesse erhalten den *init*-Prozeß (*pid* = 1) als neuen Vaterprozeß
- ◆ Statistikdaten und exit-Status werden in der *proc*-Struktur notiert

2 Ablauf von exit im Kern (2)

- ◆ neuer Prozeßzustand: **ZOMBIE**
- ◆ Vaterprozeß wird informiert
  (*wakeup( ),* falls in *wait( )* blockiert, und Signal SIGCLD/SIGCHLD)
- ◆ Nachfolgeprozeß bestimmen und Prozeßwechsel
- ZOMBIE-Prozesse bleiben so lange bestehen, bis der Vaterprozeß mit dem Systemaufruf ***wait(2)*** den exit-Status abfragt
  - ◆ mit *wait( )* werden Statistikdaten an den Vaterprozeß übertragen und können danach mit ***times(2)*** abgefragt werden
    - – ***times(2)*** liefert die Summe der Laufzeiten aller Sohnprozesse, auf die mit ***wait(2)*** gewartet wurde
  - ◆ alternativ können die Statistikdaten des Sohnprozesses mit ***wait3(2)*** oder ***wait4(2)*** direkt abgefragt werden
  - ◆ der Eintrag in Prozeßtabelle wird freigegeben
  - ◆ die *user area* des ZOMBIE-Prozesses ist jetzt passiv (Kernel-Stack wird nicht mehr genutzt) und kann freigegeben werden

3 Systemaufruf *wait( )*

- ein Prozeß kann warten, bis ein Sohnprozeß terminiert oder gestoppt wird und dabei den Status des Sohnprozesses abfragen
  ➥ übliche Arbeitsweise einer Shell bei Vordergrundprozessen
- ***wait(2)*** blockiert den aufrufenden Prozeß so lange, bis ein Sohnprozeß im Zustand ZOMBIE existiert oder ein Sohnprozeß gestoppt wird
  - ◆ *pid* dieses Sohnprozesses wird als Ergebnis geliefert
  - ◆ als Parameter kann ein Zeiger auf einen *int*-Wert mitgegeben werden, in dem der Status (16 Bit) des Sohnprozesses abgelegt wird

Sohnprozeß ist	Status & `0xff00`	Status & `0x00ff`
gestoppt	Signalnummer	`0xff`
terminiert durch *exit( )*	exit-code	`0x00`
terminiert durch Signal	`0x00`	Signalnummer
terminiert durch Signal + *coredump*	`0x00`	Signalnummer + `0x80`

3 Systemaufruf *wait ( )* (2)

- existiert kein Sohnprozeß, liefert *wait( )* eine Fehlermeldung
- in BSD und POSIX gibt es zusätzlich die Möglichkeit, das Blockieren beim *wait*-Aufruf zu verhindern, wenn keine terminierten oder gestoppten Sohnprozesse existieren
  - ◆ ***wait3(2)***, ***waitpid(2)*** oder ***wait4(2)*** mit Option *WNOHANG*
- in POSIX kann man mit *wait4( )* oder *waitpid( )* die Menge der Prozesse, auf die gewartet wird einschränken

4 typische *wait( )*-Anwendung

- ein Prozeß führt eine Liste aller existierenden Sohnprozesse und interessiert sich für deren exit-Status, darf selbst aber nicht durch *wait( )* blockiert werden
  → Lösung: Signal SIGCLD (SystemV) bzw. SIGCHLD (Berkeley) auswerten
  - ◆ großer Unterschied zwischen UNIX SystemV und Berkeley-UNIX!

▲ Verfahren bei Berkeley-UNIX
- ◆ Prozeß setzt s*ignalhandler*-Funktion für Signal SIGCHLD
- ◆ in der *signalhandler*-Funktion wird so lange *wait3( )* mit Option *WNOHANG* aufgerufen, bis *wait3( )* meldet, daß keine weiteren Söhne in gestopptem oder terminiertem Zustand existieren

4 typische *wait( )*-Anwendung

▲ Verfahren bei SystemV
- ◆ Prozeß setzt s*ignalhandler*-Funktion für Signal SIGCLD
- ◆ in der *signalhandler*-Funktion wird **genau ein** *wait( )*-Aufruf ausgeführt, durch den der Prozeß nicht blockiert werden kann
- ◆ da die *signalhandler*-Funktion bei Eintreffen eines Signals zurückgesetzt wird, muß sie am Ende der Signalbearbeitung neu aufgesetzt werden
- ◆ beim Setzen einer *Handler*-Funktion für SIGCLD stellt der Systemkern ein neues SIGCLD zu, wenn noch Sohnprozesse im ZOMBIE-Zustand existieren

- weitere Besonderheiten von SIGCLD in SystemV
  - ◆ SIGCLD führt (im Gegensatz zu anderen Signalen) per default nicht zum Abbruch eines Prozesses - es wird einfach ignoriert
  - ◆ wird für SIGCLD explizit vereinbart, daß es ignoriert werden soll, werden Sohnprozesse automatisch - ohne *wait( )*-Aufruf - vom Kern aus dem ZOMBIE-Zustand befreit