F Prozesse

F.1 Überblick

- UNIX-Prozesse
  - Prozeßbild, Speicherorganisation
  - Verwaltungsstrukturen, Identifikatoren
  - Prozeßzustände
  - Erzeugen von Prozessen
  - Prozeßumschaltungen
  - Ausführen von Programmen
  - Beenden von Prozessen

F.1 Überblick (2)

- MACH — Tasks und Threads
  - Motivation
  - Tasks
    - Konzept
    - Anwendungsschnittstelle
  - Threads
    - User-level-Threads / Kernel-Threads
    - P-Threads
    - Anwendungsschnittstelle

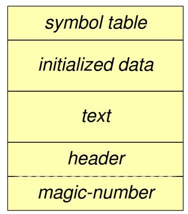
F.1 Überblick (3)

- Koordinierung
  - UNIX-Koordinierungsmechanismen
    - Semaphore
    - Kern-Koordinierung (sleep/wakeup)
  - Pthreads-Koordinierung
    - Mutexes
    - Condition Variables
- Scheduling
  - UNIX
    - SystemV
    - BSD
  - MACH

F.2 UNIX — Prozeßbild und Speicherorganisation

1 Speicherorganisation eines Programms

- Format einer ausführbaren Datei (***a.out***- oder ***executable***-**Format)**



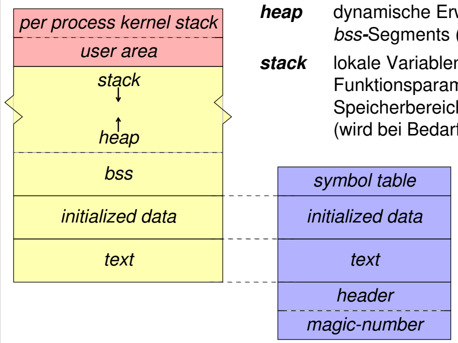
magic number	zur Identifizierung des Dateiformats (z. B. verschiedene *executable* Formate möglich)
header	Verwaltungsinformationen (z. B. Größen der einzelnen Segmente)
text	Programmkode
initialized data	initialisierte globale und *static* Variablen
symbol table	Zuordnung der im Programm verwendeten symbolischen Namen von Funktionen und globalen Variablen zu Adressen (z. B. für Debugger)

2 Speicherorganisation eines Prozesses

bss nicht initialisierte globale und *static* Variablen (wird vor der Vergabe an den Prozess mit 0 vorbelegt)

heap dynamische Erweiterungen des *bss*-Segments (***sbrk(2)***, ***malloc(3)***)

stack lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)



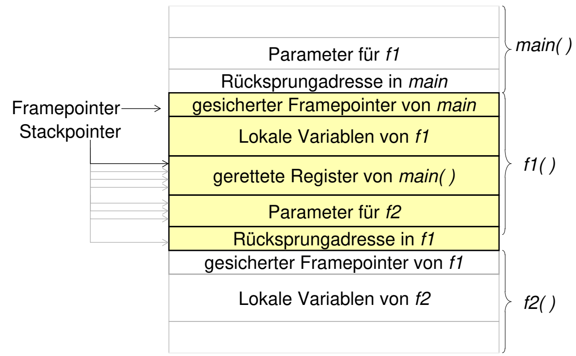
3 Stackaufbau eines Prozesses

- für jede Funktion wird ein ***Stack-Frame*** angelegt, in dem
  - lokale Variablen der Funktion
  - Aufrufparameter an weitere Funktionen
  - Registerbelegung der Funktion während des Aufrufs weiterer Funktionen

  gespeichert werden
- Stackorganisation ist abhängig von
  - Prozessor
  - Compiler und
  - Betriebssystem
- Beispiele aus einem UNIX auf Motorola 68k-Prozessor
  - typisch für CISC-Prozessoren
  - RISC-Prozessoren mit Registerfiles gehen anders vor!

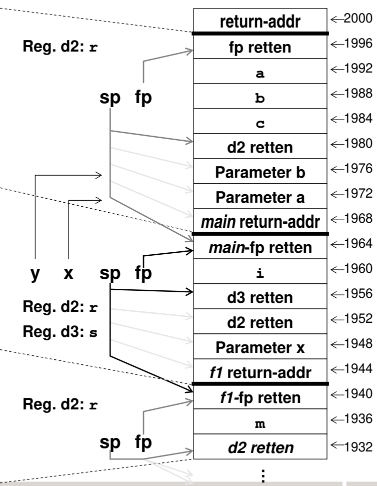
3 Stackaufbau eines Prozesses (2)

- Aufbau eines **Stack-Frames** (Funktionen *main( ), f1( ), f2( )*)



3 ■ Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;

        x++;

        i = f2(x);

        return(i);
}
f2(z) {
        int m;
        register r;

        r = z;
        m = 100;

        return(z+1);
}
```



3 Stackaufbau (4)

- Motorola 68000-Assembler — main( )

```
                        main() {
                                int a, b, c;
#NO_APP                         register r;
gcc2_compiled.:
.text                           a = 10;
    .even                       b = 20;
.globl _main                    r = 40;

                                f1(a, b);

                                return(a);
                        }
_main:
    link a6,#-12        # alten Framepointer retten und Stackbereich fuer lokale Var. reservieren (12 Byte)
    movel d2,sp@-       # Register d2 fuer Variable r retten - auf Stack legen und Stackpointer dekr.
    jbsr ___main        # (C++ - Schnittstelle - hier ihne Bedeutung)
    moveq #10,d1        # 10 in Hilfsregister d1 laden
    movel d1,a6@(-4)    # a (Framepointer - 4) = 10 (aus Register d1)
    moveq #20,d1        # 20 in Hilfsregister d1 laden
    movel d1,a6@(-8)    # b (Framepointer - 8) = 20 (aus Register d1)
    moveq #40,d2        # r (Register d2) = 40
    movel a6@(-8),sp@-# b (Framepointer - 8) auf Stack legen, Stackpointer um Wortlaenge (4 Byte) dekr.
    movel a6@(-4),sp@-# a (Framepointer - 4) auf Stack legen, Stackpointer um Wortlaenge (4 Byte) dekr.
    jbsr _f1            # Funkt. f1 aufrufen, dabei Ruecksprungaddr. auf Stack legen und Stackpointer dekr.

                        # ... hier Fortsetzung nach return aus f1
    addqw #8,sp         # Stackpointer um 8 erhoehen - damit entfernen der Aufrufpar. a und b (je 4 Byte)
    movel a6@(-4),d0    # a (Framepointer - 4) in Register d0 laden (Rueckgabeparameter)
    jra L1              # jump auf return-code

L1:                     # return-code fuer main
    movel a6@(-16),d2   # Register d2 restaurieren (war fuer Var. r auf Stack gerettet worden)
    unlk a6             # geretten alten Framepointer restaurieren, Stackpointer = Framepointer
    rts                 # return - Ruecksprungadresse von Stack nehmen - Stackpointer inkrementieren
```


3 Stackaufbau (5)

- Motorola 68000-Assembler — Funktion f1( )

```
                        f1(x, y)
                        {
                                int i;
                                register r, s;
.even                           x++;
.globl _f1                      i = f2(x);
                                return(i);
                        }
_f1:
    link a6,#-4         # Framepointer der Funktion main retten und
                          Stackbereich fuer lokale Var. reservieren (4 Byte)
    movel d3,sp@-       # Register d3 fuer Var. s retten - auf Stack legen und Stackptr. dekr.
    movel d2,sp@-       # Register d2 fuer Var. r retten - auf Stack legen und Stackptr. dekr.
    addql #1,a6@(8)     # 1 auf Parameter x addieren (x liegt auf Adresse Stackpointer + 8)
    movel a6@(8),sp@-   # x auf Stack legen, Stackpointer um Wortlaenge (4 Byte) dekrementieren
    jbsr _f2            # f2 aufrufen, dabei Ruecksprungaddr. auf Stack legen und Stackpointer dekr.

                        #  ... hier Fortsetzung nach return aus f2
    addqw #4,sp         # Stackpointer um 4 erhoehen - damit entfernen des Aufrufparameters x (4 Byte)
    movel d0,a6@(-4)    # Rueckgabewert von f2 (steht in Reg. d0) in Var. i (Framepointer - 4) speichern
    movel a6@(-4),d0    # Variable i in Register d0 laden (Rueckgabewert an main)
    jra L2              # jump auf return-code

L2:                         # return-code fuer f1
    movel a6@(-12),d2   # Register d2 restaurieren (war fuer Var. r auf Stack gerettet worden)
    movel a6@(-8),d3    # Register d3 restaurieren (war fuer Var. s auf Stack gerettet worden)
    unlk a6             # geretten Framepointer von main restaurieren, Stackpointer = Framepointer
    rts                 # return - Ruecksprungadresse von Stack nehmen - Stackpointer inkrementieren
```


3 Stackaufbau (6)

- Motorola 68000-Assembler — Funktion f2( )

```
                        f2(z)
                        {
                                int m;
                                register r;
.even                           r = z;
.globl _f2                      m = 100;
                                return(z+1);
                        }
_f2:
     link a6,#-4        # Framepointer der Funktion f1 retten und Stackbereich fuer
                        lokale Var. reservieren (4 Byte)
    movel d2,sp@-       # Register d2 retten - auf Stack legen und Stackptr. dekr.
    movel a6@(8),d1     # Parameter z (liegt auf Addr. Framepointer +8) an Var. r (Register d1) zuweisen
    moveq #100,d2       # 100 in Hilfsregister d2 schreiben
    movel d2,a6@(-4     # m (Framepointer - 4) = 100 (aus Register d2)
    movel a6@(8),d0     # Uebergabeparameter z in Register d0 (Rueckgabewert) kopieren
    addql #1,d0         # 1 auf Rueckgabewert addieren
    jra L3              # jump auf return-code

L3:                     # return-code fuer f1
    movel a6@(-8),d2    # Register d2 restaurieren
    unlk a6             # geretten Framepointer von f1 restaurieren, Stackpointer = Framepointer
    rts                 # return - Ruecksprungadresse von Stack nehmen - Stackpointer inkrementieren
```


F.3 UNIX — Verwaltungsstrukturen des Systemkerns

1 Verwaltungsdaten pro Prozeß

- **User area**
  - ◆ Daten zur Verwaltung des Programmablaufs durch den Systemkern, z. B.
    - – Segmenttabelle bzw. Verweis auf Seiten-Kachel-Tabelle
    - – Verweis auf Eintrag in die Prozeßtabelle
    - – Statistik-Daten (Laufzeiten, ...)
    - – aktuelle Directory *(current working directory)*
    - – aktuelle Root
    - – user file descriptor table
    - – Adressen der *signal-handler*-Funktionen
    - – Bereich zum Sichern des Prozeßzustands (Register) bei Prozeßwechsel
    - – Puffer für Systemaufruf-Parameter und -Rückgabewerte

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( )***)

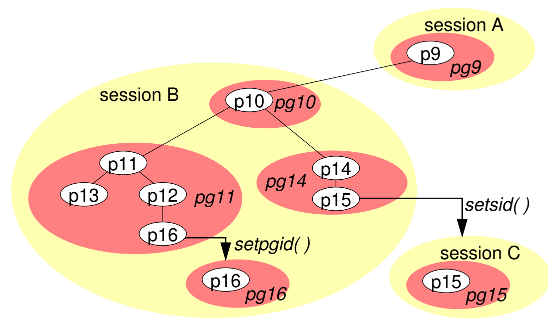
– die Prozesse dieser Prozeßgruppe laufen im **Vordergrund**

– 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)

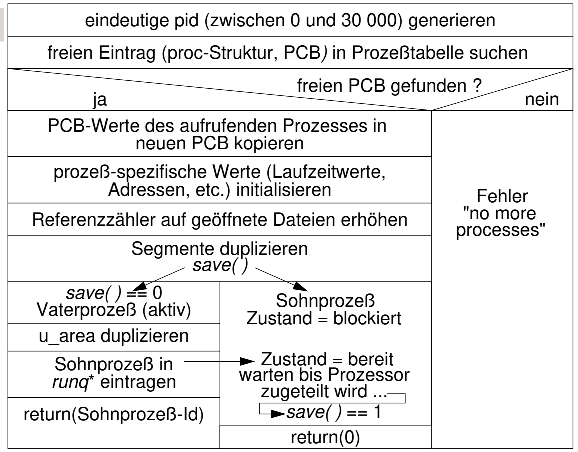
1 Überblick (2)

- 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

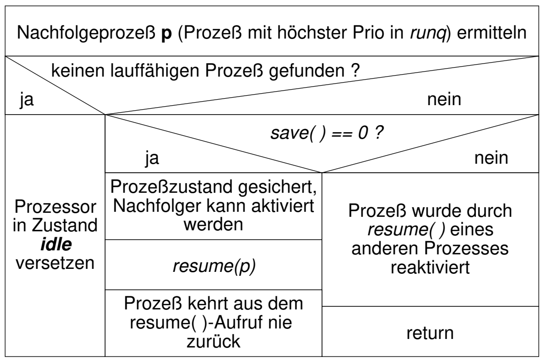
F.5 UNIX — Prozeßumschaltungen

1 Überblick

- Prozeßumschaltungen erfolgen nur an wenigen Stellen im Systemkern:
  - ➥ wenn sich ein Prozeß blockiert (***sleep( )***)
    - – Prozeß aktiviert höchstprioren, laufähigen 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, laufähigen 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( )

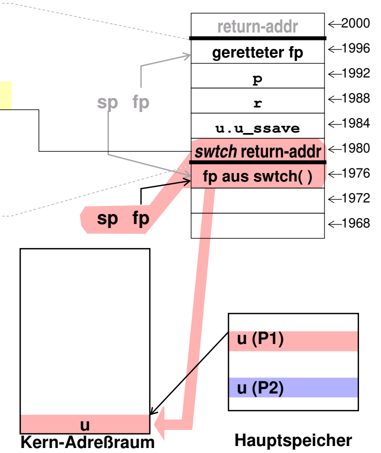
Register, Frame-Pointer, gesicherten Frame-Pointer von *swtch( )* und Rücksprungadresse nach *swtch( )* in ***u-area*** sichern
return(0)

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

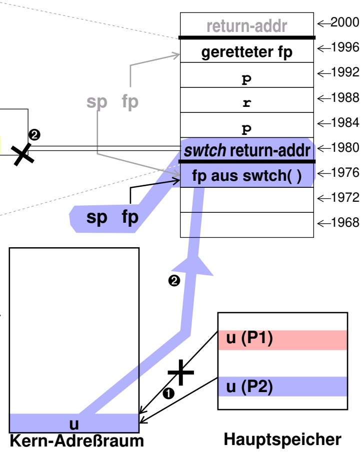
3 Ablauf (3)

```
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);
}
```

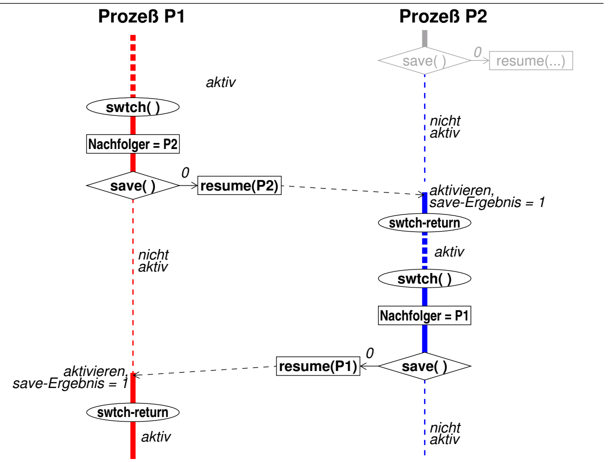


3 Ablauf (4)

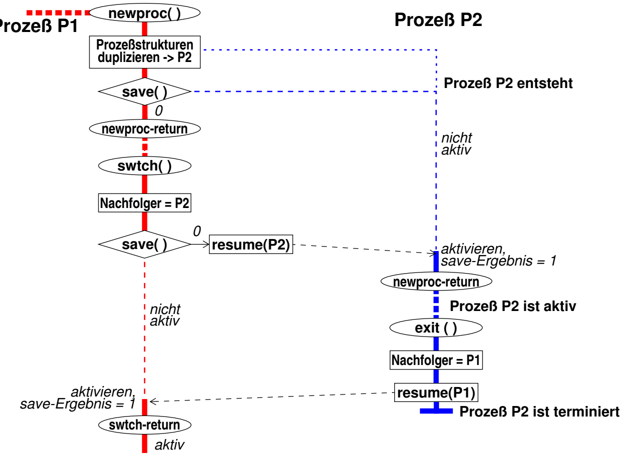
```
swtch() {
    struct proc *p;
    int r;
    p = getrq();

    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 *signalhandler*-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 *signalhandler*-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

F.8 MACH — Tasks und Threads

1 Motivation

UNIX-Prozeßkonzept ist für viele heutige Anwendungen unzureichend

- in Multiprozessorsystemen werden häufig parallele Abläufe in einem virtuellen Adreßraum benötigt
- zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adreßraums nützlich
- typische UNIX-Server-Implementierungen benutzen die *fork*-Operation, um einen Server für jeden Client zu erzeugen
  - ➥ Verbrauch unnötig vieler System-Ressourcen (Datei-Deskriptoren, Page-Table, Speicher, ...)

2 Vergleich von Prozeß- und Thread-Konzepten

- mehrere **UNIX-Prozesse** mit gemeinsamen Speicherbereichen

Bewertung:

+ echte Parallelität möglich
– viele Betriebsmittel zur Verwaltung eines Prozesses notwendig; Prozeßumschaltungen aufwendig → teuer
– innerhalb einer solchen Prozeßfamilie wäre häufig ein anwendungsorientiertes Scheduling notwendig: schwierig realisierbar

2 Vergleich von Prozeß- und Thread-Konzepten (2)

- ***User-Level-Threads*** (Koroutinen) — Realisierung von Threads auf Benutzerebene innerhalb eines Prozesses

Bewertung:

+ Erzeugung von Threads und Umschaltung extrem billig
+ Verwaltung und Scheduling anwendungsorientiert möglich
– Systemkern hat kein Wissen über diese Threads
  - ➥ Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)
  - ➥ in Multiprozessorsystemen keine parallelen Abläufe möglich
  - ➥ wird eine Koroutine wegen eines *page faults* oder in einem Systemaufruf blockiert, ist der gesamte Prozeß blockiert

2 Vergleich von Prozeß- und Thread-Konzepten (3)

- ***Kernel-Threads***: leichtgewichtige Prozesse *(lightweight processes)*

Bewertung:

+ eine Gruppe leichtgewichtiger Prozesse nutzt gemeinsam eine Menge von Betriebsmitteln
+ jeder leichtgewichtige Prozeß ist aber als eigener Aktivitätsträger dem Betriebssystemkern bekannt
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack

2 Vergleich von Prozeß- und Thread-Konzepten (3)

... Bewertung *Kernel-Threads (lightweight processes)*

+ Umschalten zwischen zwei leichtgewichtigen Prozessen einer Gruppe ist erheblich billiger als eine normale Prozeßumschaltung
  - ➥ es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf)
  - ➥ Adreßraum muß nicht gewechselt werden
  - ➥ alle Systemressourcen bleiben verfügbar

– Kosten für Erzeugung und Umschaltung zwar erheblich geringer als bei "schwergewichtigen" Prozessen, aber erheblich teurer als bei *user-level-Threads*

– Verwaltung und Scheduling meist durch Kern vorgegeben

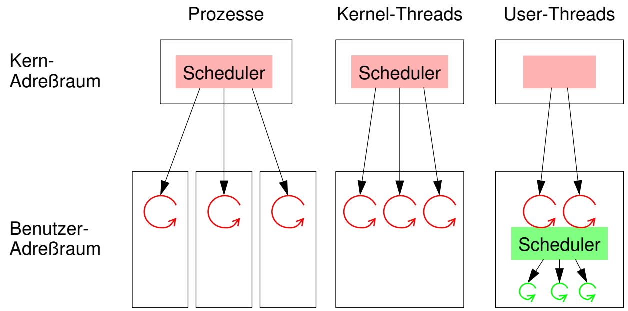
2 Vergleich von Prozeß- und Thread-Konzepten (4)

■ Vergleich

	Prozesse	Kernel-Threads	User-Threads
Kosten	– teuer	O mittel	+ billig
Betriebssystemeingliederung	+ gut	+ gut	– schlecht
Interaktion untereinander	– schwierig	+ einfach	+ einfach
Benutzerkonfigurierbarkeit	– nein	– nein	+ ja
Gerechtigkeit	– nein	+ ja	± teils

– Gerechtigkeit bedeutet:
wie kommt das System damit klar, wenn eine Anwendung eine große Anzahl von Aktivitätsträgern erzeugt, eine andere dagegen eine geringe — werden Zeitscheiben an Anwendungen oder an Aktivitätsträger vergeben?

2 Vergleich von Prozeß- und Thread-Konzepten (5)



3 Abstraktionen in MACH: Tasks und Threads

▲ ***Task:*** Betriebsumgebung (System-Resourcen) für Aktivitätsträger
  - ➤ virtueller Adreßraum
  - ➤ Zugriffsrechte (= Portrechte)
  - ➤ Betriebsmittel-Informationen
  - ➤ kein Programm-Ablauf und keine Register

▲ ***Thread:*** Aktivitätsträger und seine Ablaufumgebung
  - ➤ Registersatz
  - ➤ Stack
  - ➤ Programmzähler

◆ Innerhalb einer *Task* können beliebig viele *Threads* existieren

◆ In einer echten Multiprozessorumgebung können verschiedene *Threads* einer *Task* parallel auf mehreren Prozessoren ablaufen

◆ Traditioneller UNIX-Prozeß $\triangleq$ MACH-*Task* mit einem *Thread*

F.9 UNIX — Prozesse, LWPs & Threads

- Thread-Konzept zunehmend auch in UNIX-Systemen realisiert
  - ◆ Solaris
  - ◆ HP UX
  - ◆ Digital UNIX
  - ◆ Linux
  - ◆ …
- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
  - ➥ IEEE POSIX Standard P1003.4a
- Pthreads-Implementierungen aber sehr unterschiedlich!
  - ➤ reine User-level-Threads (Linux, HP-UX)
  - ➤ reine Kernel-Threads (MACH, KSR-UNIX, Digital UNIX)
  - ➤ parametrierbare Mischung (Solaris)
- Daneben z. T. auch andere Thread-Bibliotheken (z. B. Solaris-Threads)

1 pthread-Benutzerschnittstelle

- **Pthreads-Schnittstelle (Basisfunktionen)**:

pthread_create	Thread erzeugen & Startfunktion angeben
pthread_exit	Thread beendet sich selbst
pthread_cancel	Anderen Thread beenden
pthread_join	Auf Ende eines anderen Threads warten
pthread_self	Eigene Thread-Id abfragen
pthread_yield	Prozessor zugunsten eines anderen Threads aufgeben

1 pthread-Benutzerschnittstelle (2)

- Threaderzeugung

```
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void *),
               void *arg)
```

`thread`	Thread-Id
`attr`	modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **`NULL`** für Standardattribute.

Thread wird erzeugt und ruft Funktion **`start_routine`** mit Parameter **`arg`** auf

- Thread beenden (bei return aus **`start_routine`** oder):

```
void pthread_exit(void *retval)
```

- Auf Thread warten und exit-Status abfragen:

```
pthread_join(pthread_t thread, void **retvalp)
```


1 pthread-Benutzerschnittstelle (3)

- Beispiel (Multiplikation Matrix mit Vektor)

```
double a[100][100], b[100], c[100];

main() {
    pthread_t tids[100];
...
    for (i = 0; i < 100; i++)
        pthread_create(tids + i, NULL, mult,
                       (void *)(c + i));
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
...
}

void *mult(void *cp) {
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```

2 Threads & UNIX-Semantik

★ **fork-Semantik**

■ Problem

◆ alle Threads duplizieren?

◆ nur den aufrufenden Thread duplizieren?

■ Lösung im Pthreads-Standard

◆ im neu erzeugten Prozeß existiert nur der Thread, der fork aufgerufen hat

■ Alternative bei Solaris-Threads

◆ ***Fork One Model***:
nur der aufrufende Thread wird in den Sohnprozeß übernommen

◆ ***Fork All Model:***
Sohnprozeß enthält alle Threads

2 Threads & UNIX-Semantik (2)

★ **Signal-Semantik**

■ Grundlegendes Signal-Behandlungskonzept unverändert:

➤ Signal ignorieren / Default-Reaktion / Signal abfangen

➥ Einstellung gilt immer für alle Threads eines Prozesses

■ Problem

◆ welchem Thread wird ein eintreffendes Signal zur Bearbeitung zugestellt

2 Threads & UNIX-Semantik (3)

★ **Signal-Semantik (2)**

▲ Signal-Zustellung an Threads

■ Lösung unterscheidet nach der Art der Signal-Entstehung

◆ Traps: synchron durch die Programmausführung erzeugt
(Segmentation fault, Illegal instruction, …)

➤ Signal wird an den verursachenden Thread zugestellt

◆ Explizite Signalerzeugung durch das Programm (Funktion *pthread_kill*)

➤ Signal wird an den angegebenen Ziel-Thread zugestellt

◆ Interrupts: asynchron von "außen" erzeugt
(Interrupt, Quit, Hangup, SIGIO, …)

➤ Signal wird dem gesamten Prozess zugestellt

■

2 Threads & UNIX-Semantik (3)

★ **Signal-Semantik (3)**

■ Signal-Masken

◆ Signale können in UNIX maskiert werden — die Zustellung zurückgehalten

■ Signal-Masken werden von Threads an neu erzeugte Threads vererbt

■ Threads können Thread-spezifische Signal-Masken setzen
(Funktion *pthread_sigmask*)

▲ Bearbeitung von Signalen an den gesamten Prozess

◆ Zustellung erfolgt an einen der Threads, die das Signal nicht blockiert haben

◆ Auswahl des Threads erfolgt zufällig

F.10 Koordinierungsmechanismen

1 UNIX — Semaphore

- Atomare Operationen auf einem Semaphorvektor
  Realisierung folgender theoretischer Konzepte:
  - ◆ Vektoradditionssystem (VAS)
    - ➥ mehrere Semaphorvariablen können in einer atomaren Operation individuell dekrementiert (P-Operation) bzw. inkrementiert (V-Operation) werden
  - ◆ up/down-System - ähnliche Funktionalität (Lösung des 2. Leser/Schreiber-Problems)
    - ➥ in einer atomaren Operation können mehrere Variablen auf 0 getestet werden und im Erfolgsfall werden Variablen des Semaphorvektors modifiziert
- Include-Dateien: `<sys/types.h>` `<sys/ipc.h>` `<sys/sem.h>`

1 UNIX — Semaphore (2)

- Systemschnittstelle:

semget(2)	Erzeugen eines neuen Semaphorverktors, bzw. Zugriff zu einem bestehenden Semaphor besorgen
semcntl(2)	Statuskontrolle, Initialisierung und Löschen
semop(2)	Semaphor-Operationen

- Semaphor-Operation
  - ➥ **Aufruf:**

```
int semid;
int nsops = N;
struct sembuf sbuf[N];
...
semop(semid, sbuf, nsops);
```


1 UNIX — Semaphore (3)

- Eine Semaphor-Operation ist aus mehreren Einzeloperationen zusammengesetzt, jede Einzeloperation ist durch eine Struktur ***sembuf*** beschrieben:
  - ➥ Struktur:

```
struct sembuf {
    short sem_num;          /* Sem. Nummer
*/
    short sem_op;           /* Operation
*/
```

sem_num	Nummer der Semaphorkomponente auf die sich die Operation bezieht
sem_op < 0	P-Operation (VAS)
sem_op > 0	V-Operation (VAS)
sem_op = 0	Test auf 0 (2. Leser/Schreiber-Problem)

- Eine Semaphor-Operation blockiert immer dann als ganzes, wenn eine der Einzeloperationen blockiert!

1 UNIX — Semaphore (4)

★ **UNIX-Semaphore als Vektoradditionssystem**

- ◆ vgl. F. Hofmann: Betriebssysteme: Grundkonzepte und Modellvorstellungen; 2. Aufl., Kap. 2.4, S. 73

  Vektoraddition: $e \in \mathbb{Z}^m$, Semaphorvektor $S \in \mathbb{Z}^m$,
  Φ: $'S + e \geq 0$ $\quad$ Θ: $S = {'S} + e$

- **Realisierung**:
  Eine Operation *e* auf dem Semaphorvektor wird durch ein Feld von sembuf-Strukturen beschrieben:

  $\forall\ e_i \in e \wedge e_i \neq 0$: $\exists$ sembuf (sembuf.sum_num = i $\wedge$ sembuf.sem_op = e_i)
  $\forall\ e_i \in e \wedge e_i = 0$: $\nexists$ sembuf (sembuf.sum_num = i)

1 UNIX — Semaphore (5)

... **UNIX-Semaphore als Vektoradditionssystem**

sem_op < 0 P-Teiloperation
sem_op wird vom Wert der Semaphorkomponente abgezogen, wenn diese dadurch nicht kleiner 0 wird.
Gesamtoperation blockiert oder liefert Fehler (je nach ***flag***) wenn nicht möglich

sem_op > 0 V-Teiloperation
sem_op wird zum Wert der Semaphorkomponente addiert

kein sembuf für Semaphorkomponenten, die unverändert bleiben sollen!

1 UNIX — Semaphore (6)

★ **2. Leser-Schreiber-Problem**

- vgl. F. Hofmann: Betriebssysteme: Grundkonzepte und Modellvorstellungen; 2. Aufl., Kap. 2.4, S. 76

■ UNIX-Sempahore realisieren kein echtes *up/down*-System, entscheidend für das 2. Leser-Schreiber-Problem ist aber:

- ➥ atomarer Test mehrerer Variablen, verbunden mit Modifikation anderer Variablen
- ➥ beliebig viele P-Operationen ermöglichen und so lange warten können, bis die entsprechenden V-Operationen komplett durchgeführt wurden
  - ➤ alle anderen Koordinierungssysteme können nur von einem endlichen Wert bis 0 belegen!

1 UNIX — Semaphore (7)

... **2. Leser-Schreiber-Problem**

■ Lösung mit UNIX-Semaphoren

- ➥ Semaphorvektor mit 3 Komponenten
  1. gegenseitiger Ausschluß zwischen Schreibern
  2. Zähler für Leser
  3. Anmeldung der Schreiber

  Vorbelegung: $\begin{pmatrix}1\\0\\0\end{pmatrix}$

- ➥ **Leser**

  semop $\begin{pmatrix}-\\1\\0\end{pmatrix}$; *lesen*; semop $\begin{pmatrix}-\\-1\\-\end{pmatrix}$;

- ➥ **Schreiber**

  semop $\begin{pmatrix}-\\-\\1\end{pmatrix}$; semop $\begin{pmatrix}-1\\0\\-\end{pmatrix}$; *schreiben*; semop $\begin{pmatrix}1\\-\\-1\end{pmatrix}$;

  – bedeutet, daß bei semop für die entsprechende Vektorkomponente keine sembuf-Struktur übergeben wird!

2 Pthreads-Koordinierung

■ UNIX-Semaphore für Koordinierung von leichtgewichtigten Prozessen zu teuer

- Implementierung durch den Systemkern
- komplexe Datenstrukturen

■ Bei Koordinierung von Threads reichen meist einfache ***mutex***-Semaphore

- gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

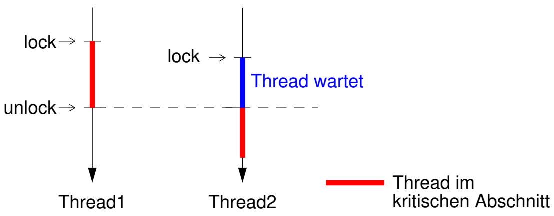
■ Komplexere Semaphore können alleine mit Mutexes nicht implementiert werden

- ➥ Problem:
  - Ein Mutex sperrt die Datenstruktur des komplexen Semaphors
  - Der Zustand der Datenstruktur erlaubt die Operation nicht
  - Blockieren an einem weiteren Mutex kann zu Verklemmungen führen
- ➥ Lösung: mutex in Verbindung mit sleep/wakeup-Mechanismus
- ➥ **Condition Variables**

2 Pthreads-Koordinierung (2)

★ **Mutexes**

- Koordinierung von kritischen Abschnitten



2 Pthreads-Koordinierung (3)

... **Mutexes (2)**

- Realisierung:
  - ◆ Implementierung weitgehend im Anwendungsmodus
  - ◆ Voraussetzung: atomarer Maschinenbefehl zum Testen und Belegen der Semaphorvariablen
    (z. B. Test-and-Set oder Compare-and-Swap)
  - ◆ Ist Semaphor frei kann, der Thread den kritischen Abschnitt ohne Verzögerung betreten
  - ◆ Ist Sempahor belegt kann alternativ
    - – der Thread den Prozessor aufgeben (thread_yield)
      (normale Pthreads-Vorgehensweise)
    - – der Thread aktiv auf die Freigabe warten (Spin)
      (Option z. B. bei der KSR Pthreads-Implementierung)
  - ◆ Bei der unlock-Operation werden blockierte Threads aufgeweckt (thread_resume)

2 Pthreads-Koordinierung (4)

... **Mutexes (3)**

- Schnittstelle
  - ◆ Mutex erzeugen

```
pthread_mutex_t m1
s = pthread_mutex_init(&m1, pthread_mutexattr_default);
```

  - ◆ Lock & unlock

```
s = pthread_mutex_lock(&m1);
... kritischer Abschnitt
s = pthread_mutex_unlock(&m1);
```

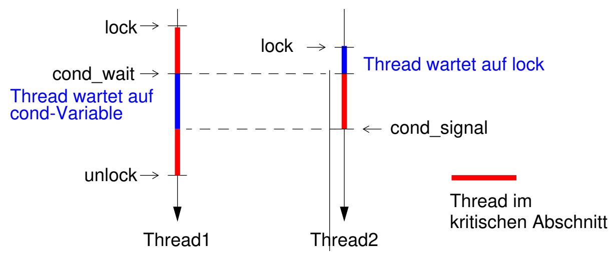
  - ◆ weitere Funktionen

pthread_mutex_trylock:	wie lock, blockiert aber nicht wenn krit. Abschnitt belegt ist
pthread_mutex_getowner:	liefert pthread-ID des aktuellen lock-Besitzers
pthread_mutex_destroy:	löscht Mutex

2 Pthreads-Koordinierung (5)

★ **Condition Variables**

- Mechanismus zum Blockieren (mit gleichzeiter Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads



2 Pthreads-Koordinierung (6)

... **Condition Variables (2)**

- Realisierung
  - ◆ Thread reiht sich in Warteschlange der Condition Variablen ein
  - ◆ Thread gibt Mutex frei
  - ◆ Thread gibt Prozessor auf
  - ◆ Ein Thread der die Condition Variable "frei" gibt weckt einen (oder alle) darauf wartenden Threads auf
  - ◆ Deblockierter Thread muß als erstes den kritischen Abschnitt neu betreten (lock)
  - ◆ Da möglicherweise mehrere Threads deblockiert wurden, muß die Bedingung nochmals überprüft werden
    (Analogie zu UNIX sleep/wakeup !)

2 Pthreads-Koordinierung (7)

... **Condition Variables (3)**

- Schnittstelle
  - ◆ Condition Variable erzeugen

```
    pthread_cond_t c1;
    s = pthread_cond_init(&c1, pthread_condattr_default);
```

- ◆ Betriebsmittel belegen

```
pthread_mutex_lock(&m1);
    while ( resource_busy )
      s = pthread_cond_wait
            (&c1, &m1);
    resource_busy = TRUE;
pthread_mutex_unlock(&m1);
... /* Betriebsmittel nutzen */
```

Betriebsmittel freigeben

```
pthread_mutex_lock(&m1);
    resource_busy = FALSE;
    pthread_cond_signal(&c1);
pthread_mutex_unlock(&m1);
```


2 Pthreads-Koordinierung (8)

... **Condition Variables (4)**

- Bei `pthread_cond_signal` wird mindestens einer der wartenden Threads augeweckt — es ist allerdings nicht definiert welcher
- Mit `pthread_cond_broadcast` werden alle wartenden Threads aufgeweckt
- Ein aufwachender Thread wird als erstes den Mutex neu belegen — ist dieser gerade gesperrt bleibt der Thread solange blockiert!
- Lost Wake-Up Problem
  - ◆ Aufruf von `pthread_cond_signal` nach Freigabe des Mutex-LOcks würde eine Thread-Umschaltung sparen
  - ◆ Aber: wenn Aufruf von `pthread_con_signal` erfolgt, während der andere Thread zwischen `resource_busy`-Abfrage und `pthread_cond_wait` ist, geht das Signal verloren

3 Thread-Cancellation

- Probleme
  - ◆ Thread hält Locks, Thread hat Ressourcen allokiert
  - ◆ Thread bearbeitet gerade komplexe Datenstrukturen und diese sind momentan in inkonsistentem Zustand
- Lösungen
  - ◆ Cancellation in bestimmten Situationen verhindern,
    - ➥ cancelability state: PTHREAD_CANCEL_DISABLE
  - ◆ Cancellation verzögern, wird er am nächsten *Cancellation Point* wirksam
    - ➥ cancelability type: PTHREAD_CANCEL_DEFERRED
  - ◆ *Cleanup Stack*: Liste von Funktionen, die beim Termineren aufgerufen werden
- Funktionen
  - ◆ *pthread_cancel(3T), pthread_setcancelstate(3T), pthread_setcanceltype(3T), pthread_textcancel(3T), pthread_cleanup_push(3T), pthread_cleanup_pop(3T)*

4 Koordinierung im UNIX-Kern (sleep/wakeup)

- der Ablauf eines Prozesses muß häufig mit dem anderer Prozesse oder mit äußeren Ereignissen *(external event)* koordiniert werden
  - ◆ Daten sind nicht immer unmittelbar verfügbar, wenn sie angefordert werden (z. B. Zeichen von Terminal, Block von Platte) *(ext. event)*
  - ◆ Semaphoroperationen blockieren
  - ◆ (quasi-)gleichzeitige Zugriffe auf globale Datenstrukturen des Systemkerns müssen geeignet koordiniert werden z. B. ***mutual exclusion***
- Betriebssystem-interner Koordinierungsmechanismus
  - ◆ Prozessor muß für andere, lauffähige Prozesse freigeben werden
  - ◆ sobald der Grund für die Blockierung nicht mehr besteht, muß der blockierte Prozeß in den Zustand **bereit** zurückgeführt werden
  - ◆ hat ein Prozeß, als er sich blockiert hat, bereits wichtige Betriebsmittel belegt (z. B. *inode* gesperrt, *inode* muß von Platte geladen werden), muß er möglichst schnell in den Zustand **laufend** zurückgelangen

4 Koordinierung im UNIX-Kern (2) — sleep —

- Prozeß blockiert sich an einem Ereignis: ***wchan (wait channel)***
- in der Praxis ist *wchan* die Adresse einer Datenstruktur, die mit dem Ereignis korrespondiert - z. B. *inode*-Struktur-Adresse
- die *proc*-Strukturen blockierter Prozesse werden verkettet, die Anker dieser Queues werden durch Hashing aus dem *wchan* ermittelt
- Parameter gibt an, ob Prozeß durch Signal unterbrechbar ist
  - ◆ Abbruch wird verboten, wenn Prozeß in einem Vorgang blockiert wird, der in jedem Fall zu Ende gebracht werden muß
    (z. B. weil noch Datenstrukturen gesperrt sind)
    - ➥ Prozeß wird nur durch *wakeup* auf *wchan*, nicht aber durch eintreffende Signale aufgeweckt
  - ◆ sonst wird der Prozeß entweder durch *wakeup* auf *wchan* oder Zustellung eines Signals deblockiert

4 Koordinierung im UNIX-Kern (3) — sleep —

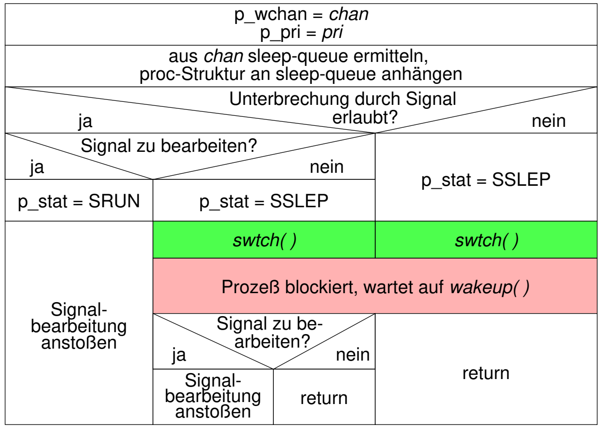
- es wird die Priorität angegeben, mit der der Prozeß wieder ausgeweckt werden soll
  - ◆ sehr hohe Priorität bei Vorgängen, die schnell fortgeführt werden müssen (z. B. weil noch *locks* gehalten werden)
- es wird ein Prozeßwechsel durchgeführt

4 Koordinierung im UNIX-Kern (4) — wakeup —

- für Ereignisse, auf die Prozesse warten können, wird ein ***wakeup( )*** ausgeführt
- es werden **alle** Prozesse in den Zustand **bereit** versetzt, die auf den angegebenen ***wchan*** gewartet haben
- jeder Prozeß muß überprüfen, ob der Blockierungsgrund wirklich nicht mehr besteht (z. B. mehrere Prozesse haben auf Freigabe eines *locks* gewartet und der erste davon hat das *lock* bereits wieder belegt)
- jeder Prozeß erhält die bei seinem ***sleep( )***-Aufruf angegebene Priorität
- die *wakeup*-Priorität wird bis zum Verlassen des Systemkerns (oder erneutem *sleep/wakeup*) beibehalten

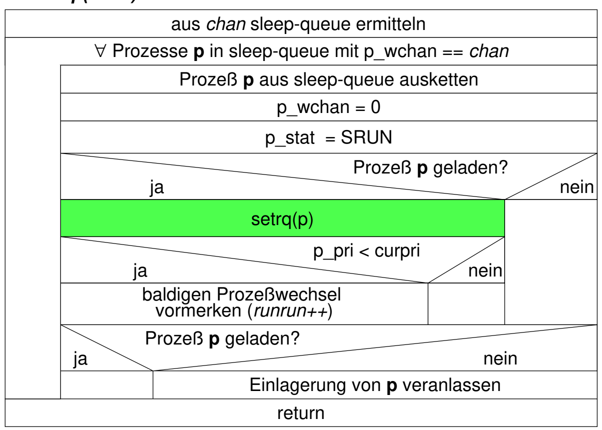
4 Koordinierung im UNIX-Kern (5)

sleep(chan, pri)



4 Koordinierung im UNIX-Kern (6)

wakeup(chan)



F.11 UNIX — Signale

- Signale können von einem Prozeß mit ***kill(2)*** oder ***sigsend(2)*** gezielt an andere Prozesse oder Prozeßgruppen zugestellt werden
- Kommunikationsablauf
  - ➥ der Signal-Empfänger hat für die Signale, die zur Kommunikation genutzt werden sollen, eine *signal-handler*-Funktion installiert
  - ➥ bei Eintreffen eines Signals wird die *signal-handler*-Funktion beim Empfänger automatisch angesprungen
  - ◆ mit einem Signal kann nur 1 Bit Information übertragen werden
- Systemaufrufe: ***sigaction(2)*** Signalbearbeitung steuern
  sigprocmask(2) Signalmaske installieren
- Prozeß schlafen legen:
  - ◆ ***pause(2)*** blockiert Prozeß bis Signal eintrifft
  - ◆ ***sigsuspend(2)*** installiert erst eine Signalmaske und blockiert Prozeß dann (zum Warten auf bestimmte Signale)

1 Ablauf der Signalverarbeitung im Systemkern

(1) Signal zustellen

- das Signal wird durch Setzen eines Bits in der Komponente `p_sig` der *proc*-Struktur des Empfängerprozesses zugestellt
- falls der Prozeß blockiert ist (→ *sleep*), wird er in den Zustand "bereit" überführt (*setrq( )*)
- die Signalbearbeitung wird dem Empfängerprozeß selbst überlassen
- nur die Stop-Signale bewirken einen direkten Eingriff in den Ablauf des Empfängerprozesses:
  - ➥ der Prozeß wird durch die Signalzustellung blockiert

1 Ablauf der Signalverarbeitung im Systemkern (2)

(2) Signal verarbeiten

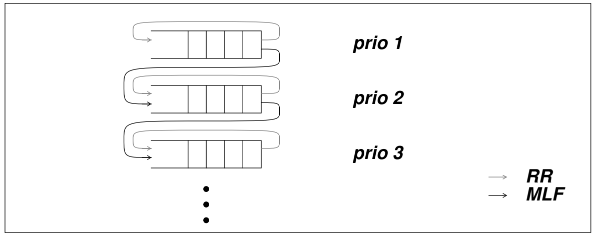
- jeder Prozeß prüft vor Verlassen des Systemkerns (am Ende eines Systemaufrufs) und vor sowie nach einem Blockieren (*sleep( )*), ob Signale für ihn vorliegen
- blockierte Signale werden nicht bearbeitet, bleiben aber in der Signal-Bitmaske `p_sig` vorgemerkt
- für alle anderen Signale wird der Reihe nach die Signalbearbeitung veranlaßt
  - ◆ *core-dump* erzeugen (wenn nötig) und *exit( )* aufrufen, oder
  - ◆ Signalbearbeitungsfunktion aufsetzen
  - ◆ während des Ablaufs einer Signalbearbeitungsfunktion können andere Signale eintreffen, die (sofern nicht blockiert) die Bearbeitung des ersten Signals sofort unterbrechen

2 SystemV ↔ BSD/POSIX-Semantik

- SystemV-Probleme
  - ◆ unsicher
    da die Signal-Reaktion vor Aufruf der Bearbeitungsfunktion zurückgesetzt wird, kann man zwei schnell aufeinanderfolgend eintreffende Signale nicht sicher bearbeiten
  - ◆ man kann Signale nur ignorieren, nicht aber verzögern
- Vorteil der BSD/POSIX-Semantik
  - ◆ entspricht vollständig der Bearbeitung von Hardware-Interrupts
    - ➤ während ein Signal bearbeitet wird, werden weitere blockiert
    - ➤ es kann maximal nur ein weiteres Signal gespeichert werden
    - ➤ zur Signalbearbeitung kann ein spezieller Stack verwendet werden (***sigaltstack(2)*** oder ***sigstack(2)***)

F.12 UNIX — Scheduling

- der Scheduling-Algorithmus legt fest, welcher der lauffähigen Prozesse den Prozessor wie lange maximal zugeteilt bekommt
- Strategie: ***multilevel feedback with round robin***



➥ entscheidende Parameter
- ➤ Länge der Zeitscheibe für *Round Robin*
- ➤ Algorithmus zur Berechnung der Priorität

F.12 UNIX — Scheduling

- grundsätzliche Unterschiede zwischen UNIX SystemV und BSD UNIX

Wertung: Berkeley-Strategie besser für interaktives Arbeiten + bessere Einflußnahme auf einzelne Prozesse möglich

Scheduling in UNIX System V

- **Zeitscheibe** für Prozesse, die Prozessor nicht vorher aufgeben: **1 s**
- Prioritätsberechnung

Priorität ***p_pri*** (Element der proc-Struktur) wird ermittelt aus:

p_cpu Maß für die CPU-Nutzung des Prozesses

p_nice vom Benutzer einstellbarer Wert zwischen -20 und +20 (eigentlich zwischen 0 und 40, wovon dann NZERO=20 abgezogen wird)

$$\boldsymbol{p\_pri} = PUSER + \frac{p\_cpu}{2} + p\_nice$$

Berechnung einmal pro Sekunde und nach jedem Systemaufruf bei Veränderungen an den ***runqueues*** wird der gerade aktive Prozeß in einen Pseudo-Systemaufruf gezwungen

$$\boldsymbol{p\_cpu} = \frac{p\_cpu}{2}$$ einmal pro Sekunde ∀ Prozesse

$\boldsymbol{p\_cpu} = p\_cpu + 1$ alle 10 ms für den aktiven Prozeß

Scheduling in UNIX System V (2)

- generelle Auswirkung des Algorithmus:
  - ◆ rechenintensive Prozesse neigen zu niedriger Priorität (großes *p_pri*), stark interaktive Prozesse (die häufig blockiert werden) zu höherer Priorität (kleines *p_pri*)
- Probleme bei hoher Systemlast
  - ◆ es existieren viele Prozesse, wobei jeder Prozeß nur relativ selten den Prozessor zugeteilt bekommt
    - → *p_cpu* der rechenintensiven Prozesse ist im Verhältnis zum wenig belasteten System niedrig
    - → geringer Prioritätsunterschied zwischen stark interaktiven und rechenintensiven Prozessen
    - → Antwortverhalten schlecht, da die rechenintensiven Prozesse ziemlich gleichberechtigt werden

Scheduling in BSD UNIX

- **Zeitscheibe** für Prozesse, die Prozessor nicht vorher aufgeben: **100 ms**
- Prioritätsberechnung

Priorität ***p_usrpri*** (Element der ***proc*-Struktur**) wird ermittelt aus:

p_cpu Maß für die CPU-Nutzung des Prozesses

p_nice vom Benutzer einstellbarer Wert zwischen -20 und +20

load mittlere Anzahl lauffähiger Prozesse in der letzten Minute

$$\boldsymbol{p\_usrpri} = PUSER + \frac{p\_cpu}{4} + 2 \cdot p\_nice$$

wird alle 40 ms für den gerade aktiven Prozeß ermittelt

$$\boldsymbol{p\_cpu} = \frac{2 \cdot load}{2 \cdot load + 1} \cdot p\_cpu + p\_nice$$

einmal pro Sekunde ∀ Prozesse

$\boldsymbol{p\_cpu} = p\_cpu + 1$ alle 10 ms für den aktiven Prozeß

Scheduling in BSD UNIX (2)

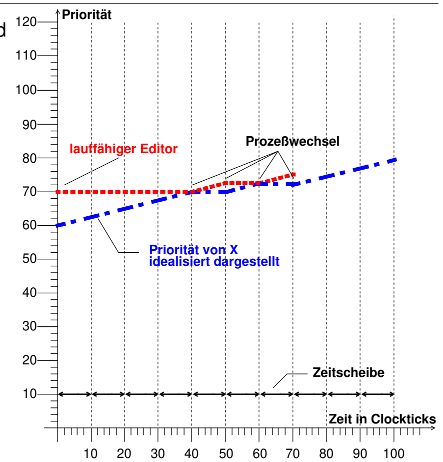
- Unterschiede zu SystemV
  - ◆ durch kürzere Zeitscheibe wird die CPU hochprioren Prozessen gleichmäßiger zugeteilt
  - ◆ Priorität wird regelmäßig und nicht abhängig von Systemaufrufen neu berechnet
  - ◆ *p_nice* hat stärkeren Einfluß auf *p_usrpri*
  - ◆ *p_nice* beeinflußt Veränderung von *p_cpu*
  - ◆ *load* beeinflußt Veränderung von *p_cpu*
    → bei hoher Last erholt sich die Priorität langsamer
- besseres interaktives Verhalten, weil
  - ◆ kürzere Zeitscheibe
  - ◆ rechenintensive Prozesse erhalten bei hoher Last und interaktiven Anforderungen kaum die aktuell beste Priorität

2 Scheduling in BSD UNIX (3)

- Prioritätsverhalten mit ***nice(2)*** leichter zu beeinflussen
  - ◆ *p_nice* wirkt sich erheblich stärker aus als in SystemV
- Verwaltungaufwand erheblich größer (z. B. häufigere Prozeßwechsel)
- Rechner, deren CPU durch den interaktiven Betrieb bereits voll ausgelastet ist, bearbeiten andere Aufgaben praktisch nicht mehr
  → Berkeley-Scheduling nur für leistungsfähige Anlagen geeignet

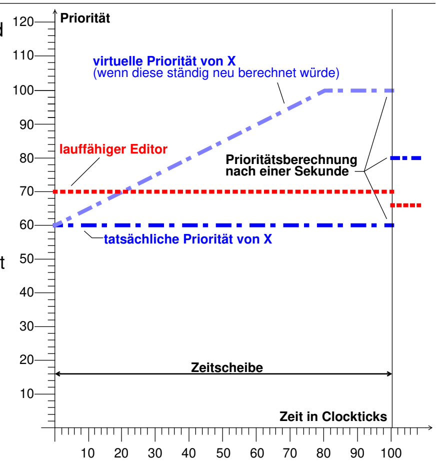
3 Vergleich: SystemV ↔ BSD (2)

- Prioritätsverhalten und Prozeßwechsel in Berkeley-UNIX
  - ◆ Rechenprozeß X, interaktiver Prozeß (z. B. Editor),
  - ◆ gleiche Bedingungen wie Beispiel für SystemV



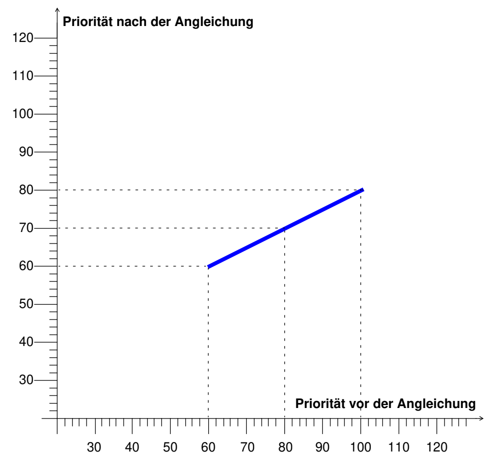
3 Vergleich: SystemV ↔ BSD

- Prioritätsverhalten und Prozeßwechsel in SystemV
  - ◆ Rechenprozeß X, interaktiver Prozeß (z. B. Editor),
  - ◆ anfangs *p_cpu* = 0
  - ◆ ohne Aufruf von Systemdiensten (dadurch wird Priorität nicht neu berechnet), keine Interrupts



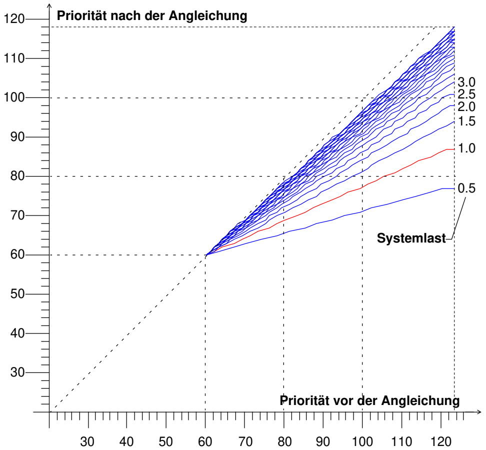
3 Vergleich: SystemV ↔ BSD (3)

- Verbesserung der Prioritäten durch Zerfallsfunktion für *p_cpu*
  - ◆ SystemV



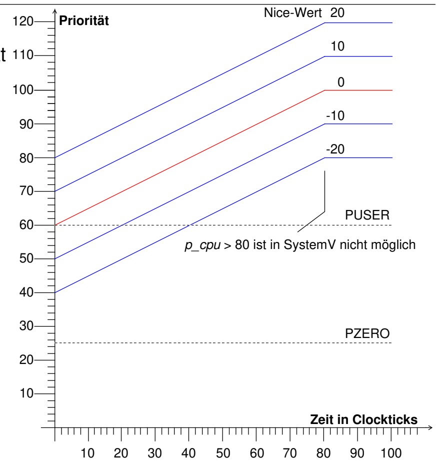
3 Vergleich: SystemV ↔ BSD (4)

- Verbesserung der Prioritäten durch Zerfallsfunktion für *p_cpu*
  - ◆ Berkeley-UNIX - abhängig von der aktuellen Systemlast



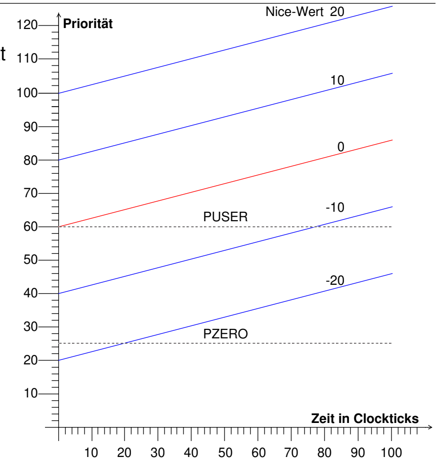
3 Vergleich: SystemV ↔ BSD (5)

- Einfluß des *nice*-Wertes auf die Priorität eines Prozesses
  - ◆ SystemV



3 Vergleich: SystemV ↔ BSD (6)

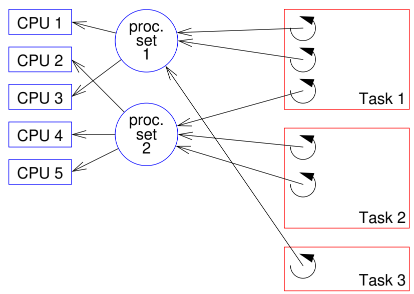
- Einfluß des *nice*-Wertes auf die Priorität eines Prozesses
  - ◆ Berkeley-UNIX



F.13 MACH — Scheduling

1 Architektur

- Scheduling auf Thread-Basis — unabhängig von Tasks
- Prozessoren sind zu ***processor sets*** zusammengefaßt
  - ◆ jeder Prozessor ist genau einem *processor set* zugeordnet
  - ◆ Threads können dynamisch verschiedenen *processor sets* zugeordnet werden
  - ◆ eigenes Scheduling für jedes *processor set*



2 Schedulingstrategie & Prioritäten

- Multi-level feedback Strategie mit 32 globalen Warteschlangen pro *processor set* für bereite Prozesse
- Pro Prozessor eine lokale Warteschlange für prozessor-gebundene Threads
  - ◆ hat immer Vorrang vor den globalen Warteschlangen
  - ◆ z. B. für Threads, die E/A-Funktionen für dem Prozessor zugeordnete Geräte abwickeln

2 Schedulingstrategie & Prioritäten (2)

- Dynamische Prioritätsberechnung ähnlich wie bei UNIX
  - ◆ Priorität aktiver Threads sinkt
  - ◆ Priorität passiver Threads wird abhängig von der Wartezeit verbessert

current priority	aktuelle Priorität, wird vom Kern auf der Basis der CPU-Nutzung des Threads, der aktuellen Systemlast und der *base priortity* berechnet
min. base priority	untere Grenze für *base priority*
base priority	vom Thread selbst festlegbare Basis für die Prioritätsberechnung

- Länge von Zeitscheiben abhängig von der Systemlast
  – kurze Zeitscheiben bei hoher Last (→ bessere Antwortzeiten)
  – lange Zeitscheiben bei geringer Last (→ weniger Overhead)

3 Scheduling-Unterstützung für nebenläufige Anwendungen

- MACH bietet Mechanismen um im Rahmen einer Anwendung das Scheduling von Threads der Anwendung zu beeinflussen

★ **Discouragement**

➥ Hinweis an das System, daß der aktuelle Thread nicht laufen soll (weil er z. B. aufgrund des Anwendungszustands ohnehin keinen Fortschritt machen kann)

• drei Stufen:

mild	Prozessor wenn möglich an anderen Thread abgeben
strong	Prozessor aufgeben und vorübergehend Priorität senken
absolute	Thread für bestimmten Zeitraum blockieren

3 Scheduling-Unterstützung für nebenläufige Anwendungen (2)

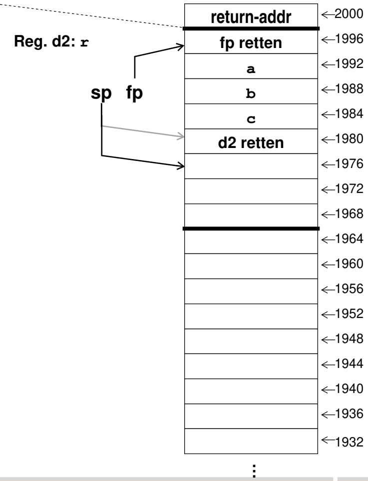
★ **Handoff-Scheduling**

➥ Ein Thread gibt den Prozessor auf und bestimmt dabei seinen Nachfolger

- ➤ Scheduling-Mechanismen des Kerns werden dabei umgangen
- ➤ Anwendung z. B. wenn ein Thread auf einen leeren Puffer trifft und den Thread, der Daten nachfüllen kann kennt
- ➤ Lösung des *priority inversion* Problems:
  ein hochpriorer Thread blockiert sich an einer Sperre, die von einem Thread mit niedriger Priorität gehalten wird

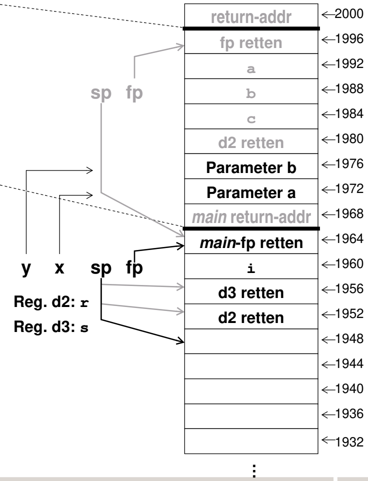
3 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;
        a = 10;
        b = 20;
        r = 40;
        f1(a, b);
        return(a);
}
```



3 Stack mehrerer Funktionsaufrufe

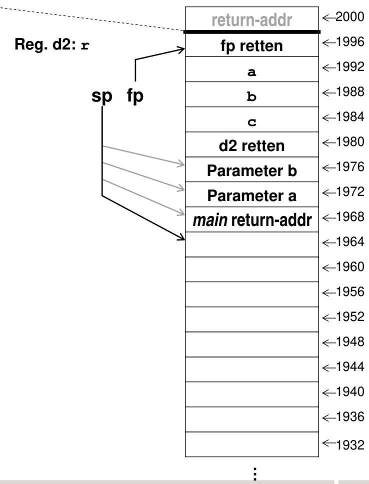
```
main() {
        int a, b, c;
        register r;
        a = 10;
        b = 20;
        r = 40;
        f1(a, b);
        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;
        x++;
        i = f2(x);
        return(i);
}
```



AKBP I
Ausgewählte Kapitel der praktischen Betriebsprogrammierung
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999
F-Prozesse.doc 1999-01-20 09.38
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Stack mehrerer Funktionsaufrufe

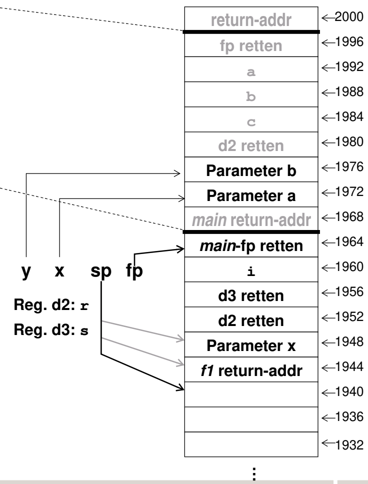
```
main() {
        int a, b, c;
        register r;
        a = 10;
        b = 20;
        r = 40;
        f1(a, b);
        return(a);
}
```



AKBP I
Ausgewählte Kapitel der praktischen Betriebsprogrammierung
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999
F-Prozesse.doc 1999-01-20 09.38
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;
        a = 10;
        b = 20;
        r = 40;
        f1(a, b);
        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;
        x++;
        i = f2(x);
        return(i);
}
```



AKBP I
Ausgewählte Kapitel der praktischen Betriebsprogrammierung
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999
F-Prozesse.doc 1999-01-20 09.38
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;

        x++;

        i = f2(x);

        return(i);
}
f2(z) {
        int m;
        register r;

        r = z;
        m = 100;

        return(z+1);
}
```

Inhalt	Adresse
return-addr	2000
fp retten	1996
a	1992
b	1988
c	1984
d2 retten	1980
Parameter b	1976
Parameter a	1972
main return-addr	1968
main-fp retten	1964
i	1960
d3 retten	1956
d2 retten	1952
Parameter x	1948
f1 return-addr	1944
f1-fp retten	1940
m	1936
d2 retten	1932
⋮	

sp fp

z

Reg. d2: r

sp fp

3 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;

        x++;

        i = f2(x);

        return(i);
}
f2(z) {
        int m;
        register r;

        r = z;
        m = 100;

        return(z+1);
}
```

Inhalt	Adresse
return-addr	2000
fp retten	1996
a	1992
b	1988
c	1984
d2 retten	1980
Parameter b	1976
Parameter a	1972
main return-addr	1968
main-fp retten	1964
i	1960
d3 retten	1956
d2 retten	1952
Parameter x	1948
f1 return-addr	1944
f1-fp retten	1940
m	1936
d2 retten	1932
⋮	

sp fp

sp fp

3 Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;

        x++;

        i = f2(x);

        return(i);
}
f2(z) {
        int m;
        register r;

        r = z;
        m = 100;

        return(z+1);
}
```

Inhalt	Adresse
return-addr	2000
fp retten	1996
a	1992
b	1988
c	1984
d2 retten	1980
Parameter b	1976
Parameter a	1972
main return-addr	1968
main-fp retten	1964
i	1960
d3 retten	1956
d2 retten	1952
Parameter x	1948
f1 return-addr	1944
f1-fp retten	1940
m	1936
d2 retten	1932
⋮	

z

Reg. d2: r

sp fp

3 Stack mehrerer Funktionsaufrufe

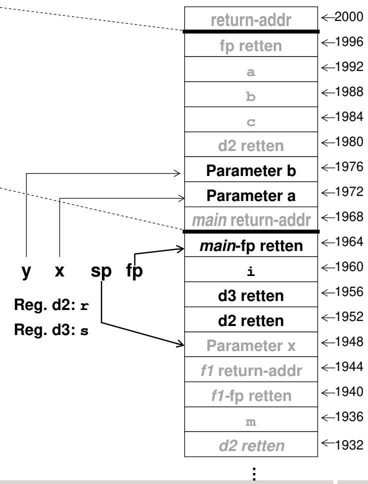
```
main() {
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;

        x++;

        i = f2(x);

        return(i);
}
```

Inhalt	Adresse
return-addr	2000
fp retten	1996
a	1992
b	1988
c	1984
d2 retten	1980
Parameter b	1976
Parameter a	1972
main return-addr	1968
main-fp retten	1964
i	1960
d3 retten	1956
d2 retten	1952
Parameter x	1948
f1 return-addr	1944
f1-fp retten	1940
m	1936
d2 retten	1932
⋮	

sp fp

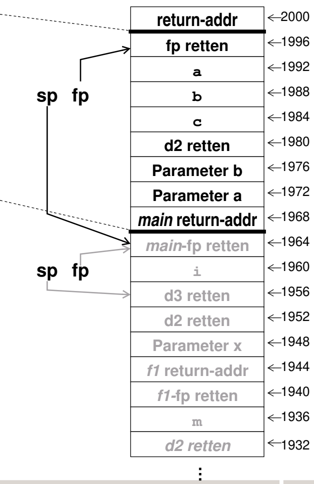
3 ■ Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;

        x++;

        i = f2(x);

        return(i);
}
```



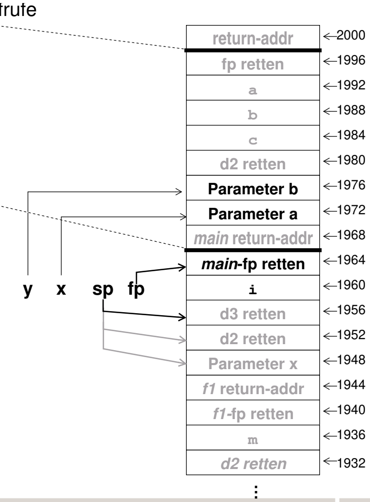
3 ■ Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;

        x++;

        i = f2(x);

        return(i);
}
```



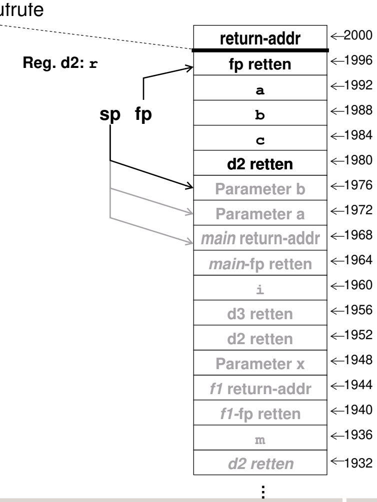
3 ■ Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
f1(x,y) {
        int i;
        register r;
        register s;

        x++;

        i = f2(x);

        return(i);
}
```



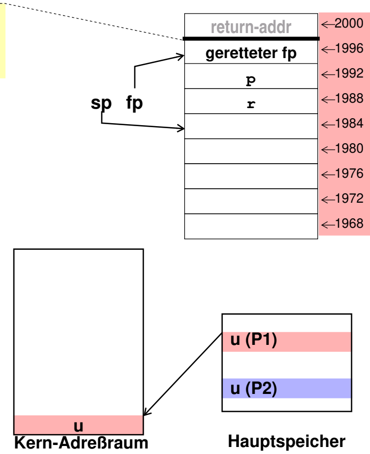
3 ■ Stack mehrerer Funktionsaufrufe

```
main() {
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
```



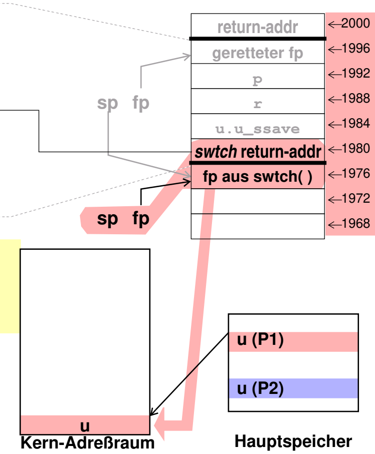
3 Ablauf (3)

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

    r = save(u.u_ssave);

    if (r == 0) {
        resume(p);
    } else {
        return(1);
    }
}
```



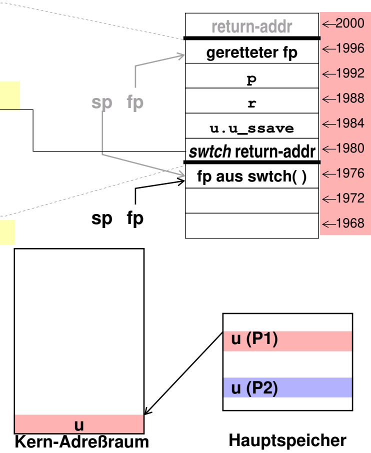
3 Ablauf (3)

```
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);
}
```



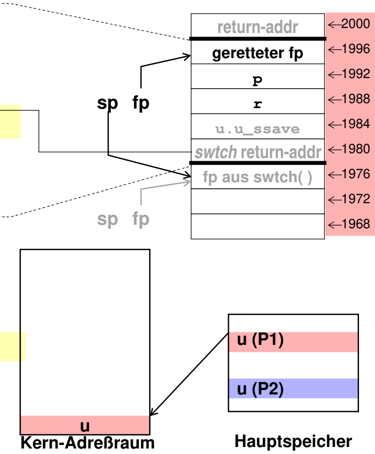
3 Ablauf (3)

```
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);
}
```



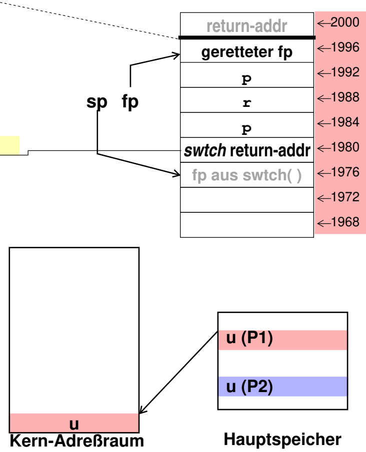
3 Ablauf (3)

```
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);
}
```



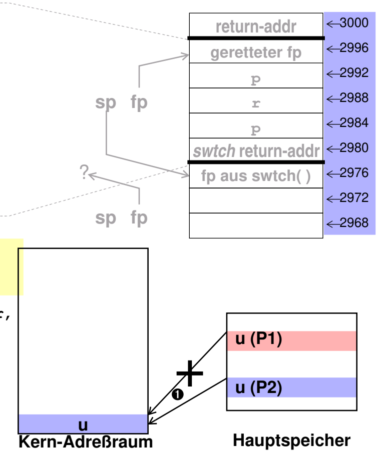
3 ■ Ablauf (3)

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

     r = save(u.u_ssave);

     if (r == 0) {
         resume(p);
     } else {
         return(1);
     }
}
```



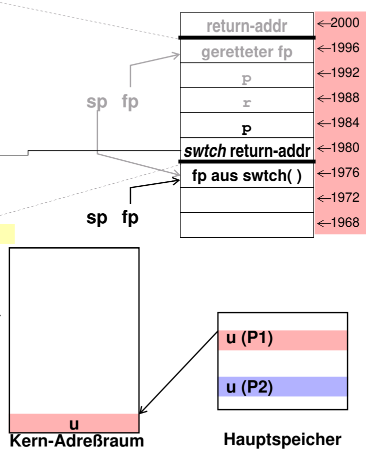
3 ■ Ablauf (4)

```
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) {
❶    /* u-area und
        kernel-stack
        von p mappen */

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

     return(1);
}
```



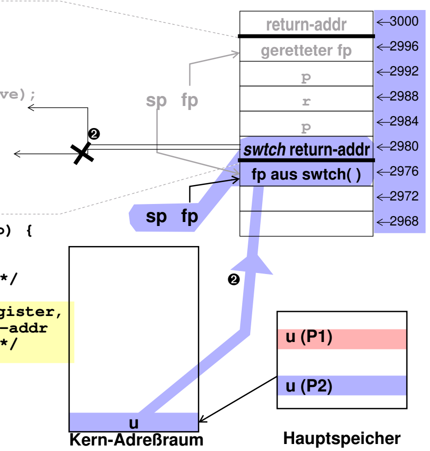
3 ■ Ablauf (4)

```
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
        kernel-stack
        von p mappen */

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

     return(1);
}
```



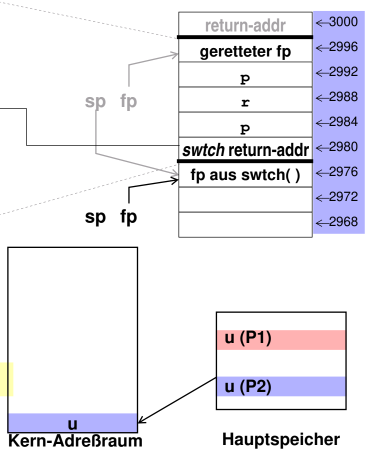
3 ■ Ablauf (4)

```
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
        kernel-stack
        von p mappen */

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

     return(1);
}
```



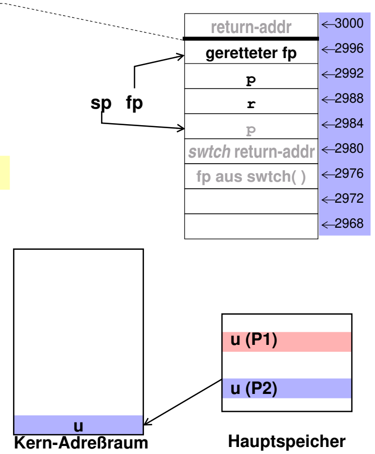
3 Ablauf (4)

```
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
       kernel-stack
       von p mappen */

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

    return(1);
}
```



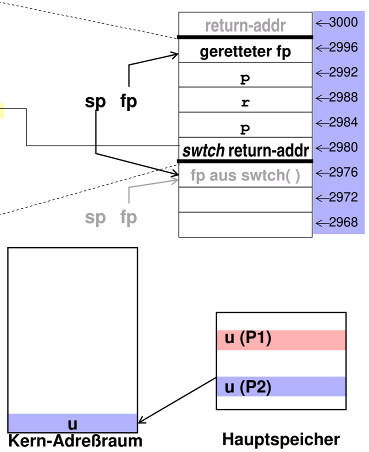
3 Ablauf (4)

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

    r = save(u.u_ssave);

    if (r == 0) {
        resume(p);
    } else {
        return(1);
    }
}
```



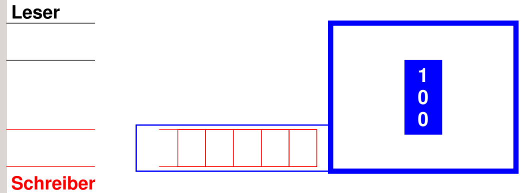
3 Ablauf (4)

```
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
       kernel-stack
       von p mappen */

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

    return(1);
}
```

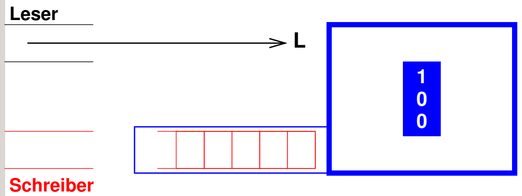


1 UNIX — Semaphore (8)

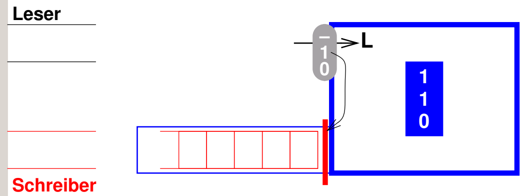
... **2. Leser-Schreiber-Problem**



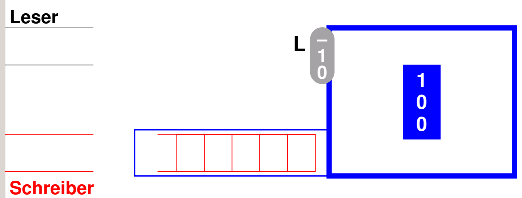
... 2. Leser-Schreiber-Problem



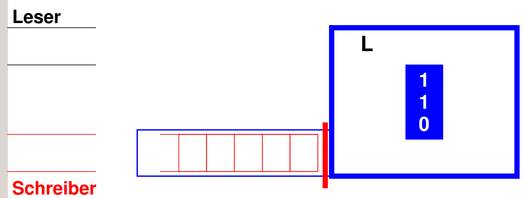
... 2. Leser-Schreiber-Problem



... 2. Leser-Schreiber-Problem

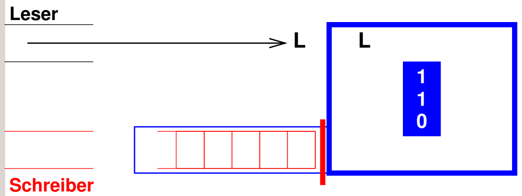


... 2. Leser-Schreiber-Problem



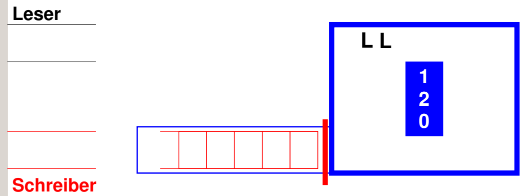
1 UNIX — Semaphore (8)

... **2. Leser-Schreiber-Problem**



1 UNIX — Semaphore (8)

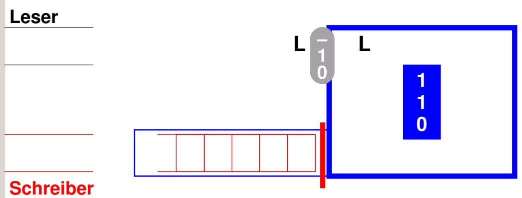
... **2. Leser-Schreiber-Problem**



AKBP I
Ausgewählte Kapitel der praktischen Betriebsprogrammierung
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999
F-Prozesse.doc 1999-01-20 09.38
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 UNIX — Semaphore (8)

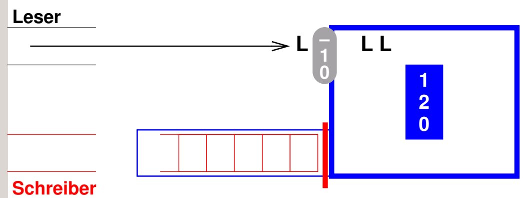
... **2. Leser-Schreiber-Problem**



AKBP I
Ausgewählte Kapitel der praktischen Betriebsprogrammierung
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999
F-Prozesse.doc 1999-01-20 09.38
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 UNIX — Semaphore (8)

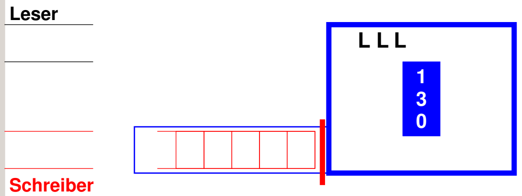
... **2. Leser-Schreiber-Problem**



AKBP I
Ausgewählte Kapitel der praktischen Betriebsprogrammierung
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999
F-Prozesse.doc 1999-01-20 09.38
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

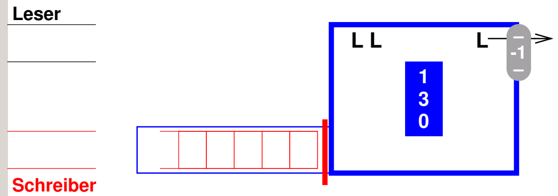
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem



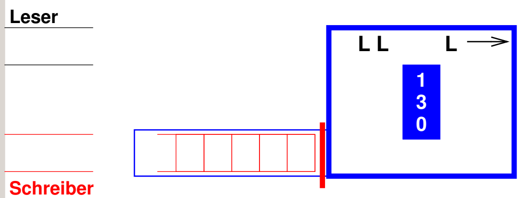
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem



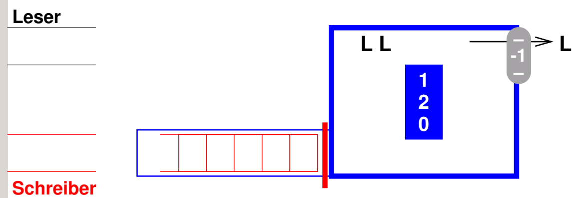
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

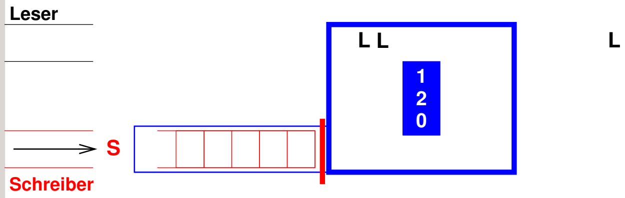


1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

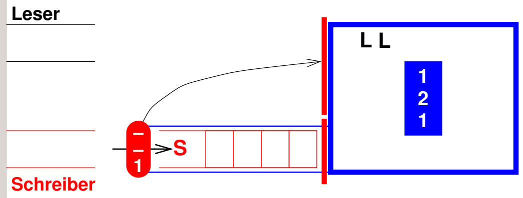


... 2. Leser-Schreiber-Problem



1 UNIX — Semaphore (8)

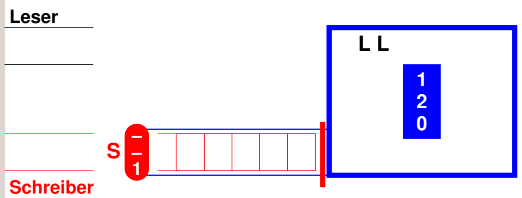
... 2. Leser-Schreiber-Problem



AKBP I Ausgewählte Kapitel der praktischen Betriebsprogrammierung
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999 F-Prozesse.doc 1999-01-20 09.38
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 UNIX — Semaphore (8)

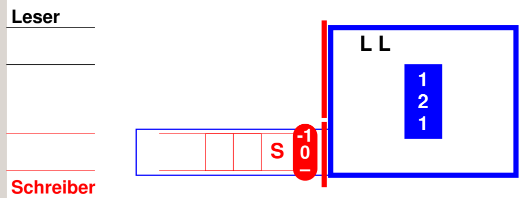
... 2. Leser-Schreiber-Problem



AKBP I Ausgewählte Kapitel der praktischen Betriebsprogrammierung
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999 F-Prozesse.doc 1999-01-20 09.38
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 UNIX — Semaphore (8)

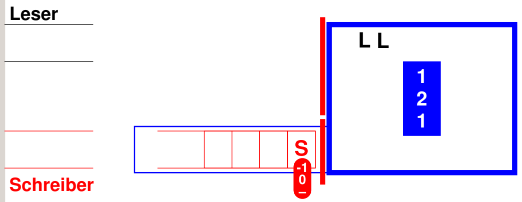
... 2. Leser-Schreiber-Problem



AKBP I Ausgewählte Kapitel der praktischen Betriebsprogrammierung
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999 F-Prozesse.doc 1999-01-20 09.38
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

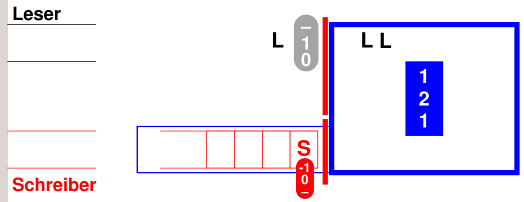
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem



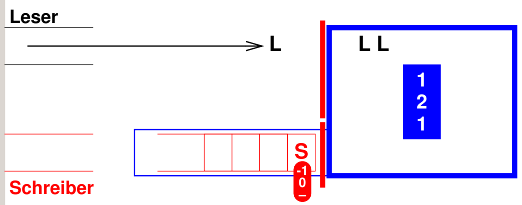
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem



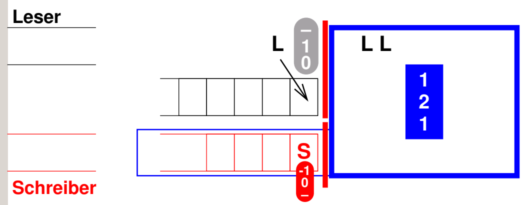
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

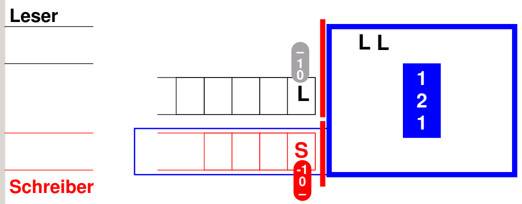


1 UNIX — Semaphore (8)

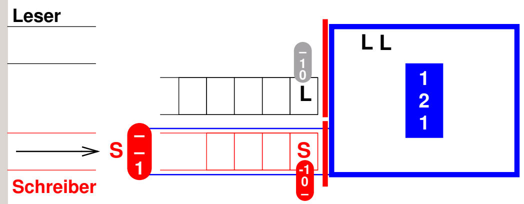
... 2. Leser-Schreiber-Problem



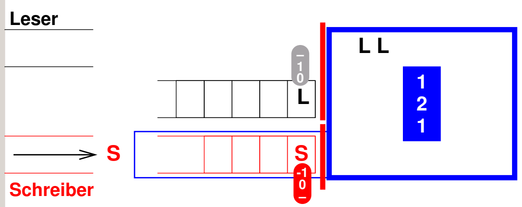
... **2. Leser-Schreiber-Problem**



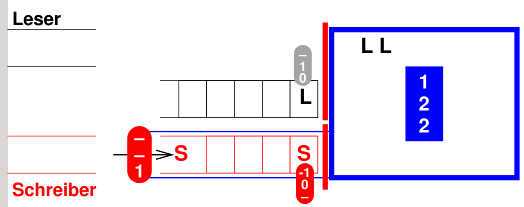
... **2. Leser-Schreiber-Problem**



... **2. Leser-Schreiber-Problem**

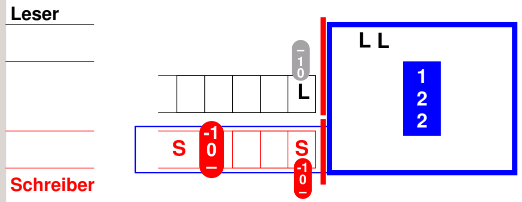


... **2. Leser-Schreiber-Problem**



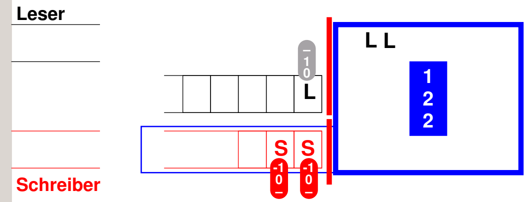
1 UNIX — Semaphore (8)

... **2. Leser-Schreiber-Problem**



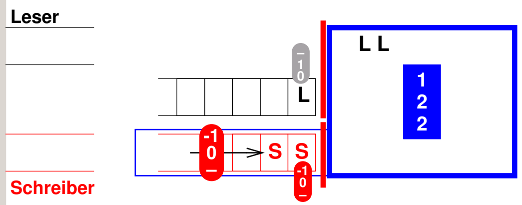
1 UNIX — Semaphore (8)

... **2. Leser-Schreiber-Problem**



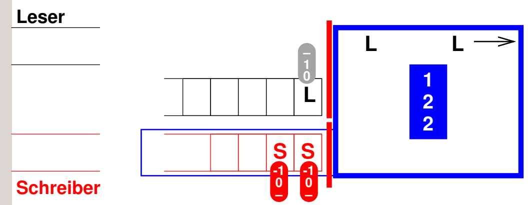
1 UNIX — Semaphore (8)

... **2. Leser-Schreiber-Problem**



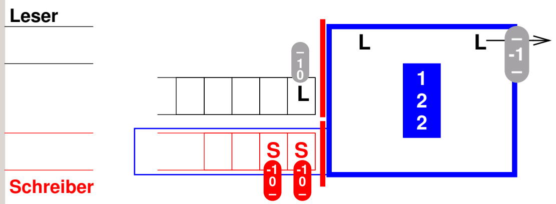
1 UNIX — Semaphore (8)

... **2. Leser-Schreiber-Problem**



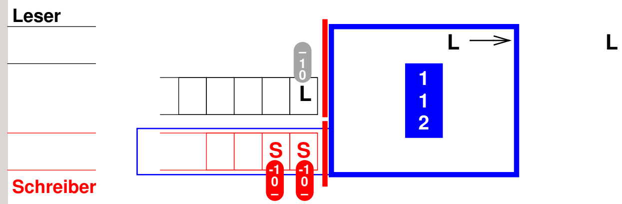
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem



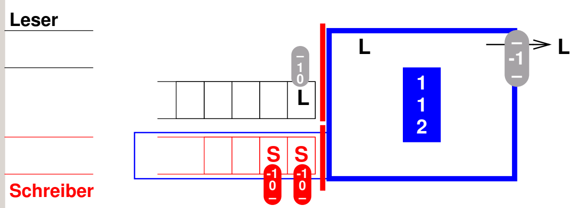
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem



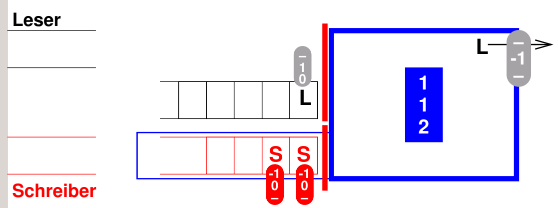
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem



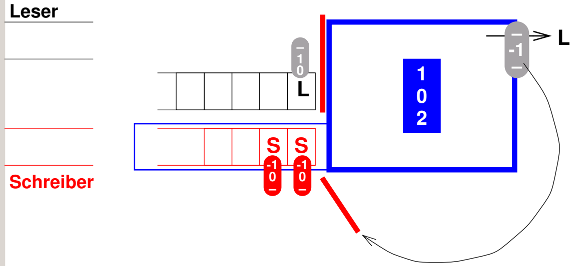
1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem



UNIX — Semaphore (8)

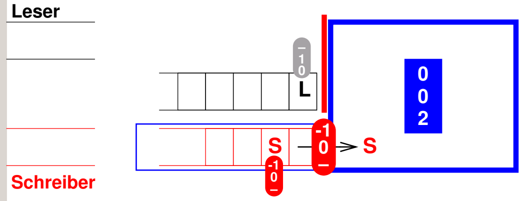
... 2. Leser-Schreiber-Problem



1

UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

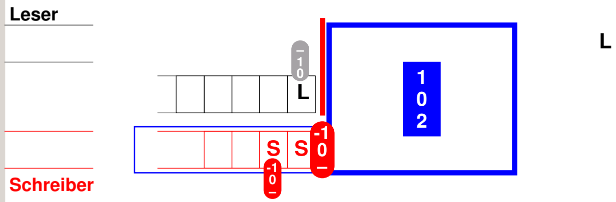


AKBP I
Ausgewählte Kapitel der praktischen Betriebsprogrammierung
F-Prozesse.doc 1999-01-20 09.38
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1

UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

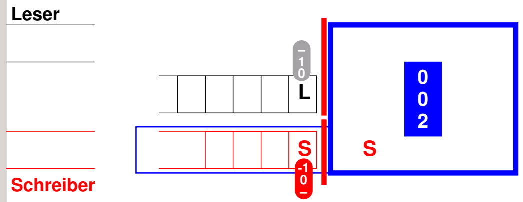


AKBP I
Ausgewählte Kapitel der praktischen Betriebsprogrammierung
F-Prozesse.doc 1999-01-20 09.38
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1

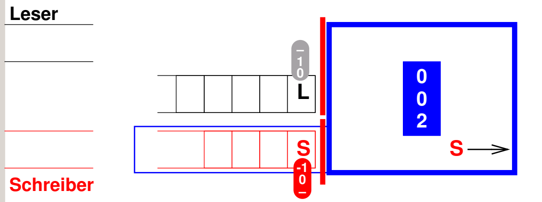
UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

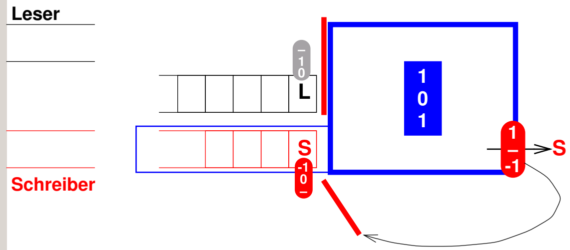


AKBP I
Ausgewählte Kapitel der praktischen Betriebsprogrammierung
F-Prozesse.doc 1999-01-20 09.38
© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1999
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

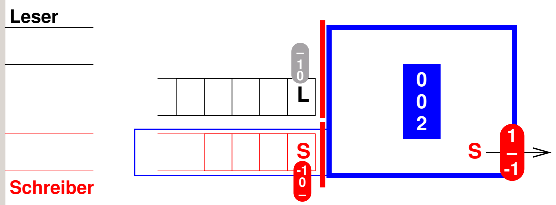
... **2. Leser-Schreiber-Problem**



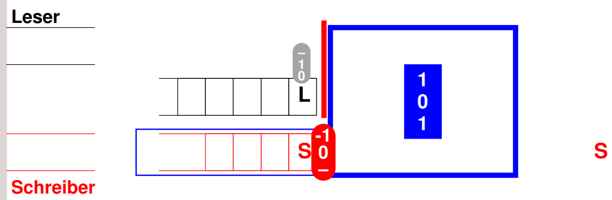
... **2. Leser-Schreiber-Problem**



... **2. Leser-Schreiber-Problem**



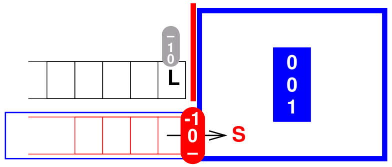
... **2. Leser-Schreiber-Problem**



1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

Leser

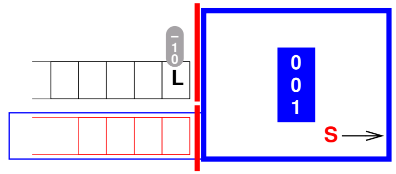


Schreiber

1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

Leser

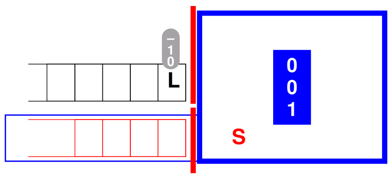


Schreiber

1 UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

Leser

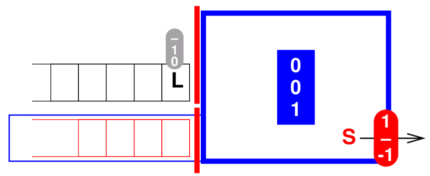


Schreiber

1 UNIX — Semaphore (8)

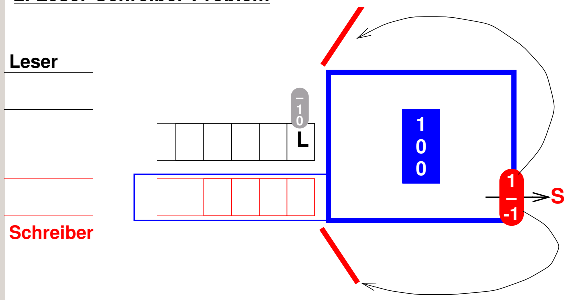
... 2. Leser-Schreiber-Problem

Leser

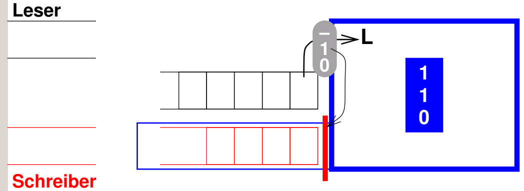


Schreiber

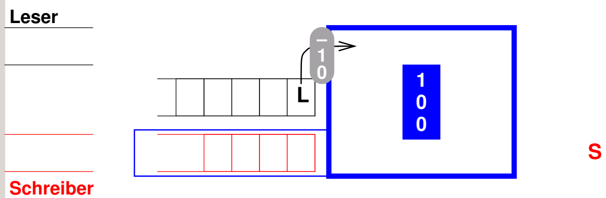
... 2. Leser-Schreiber-Problem



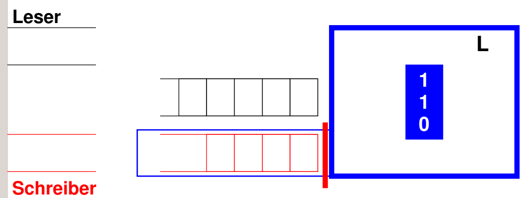
... 2. Leser-Schreiber-Problem



... 2. Leser-Schreiber-Problem

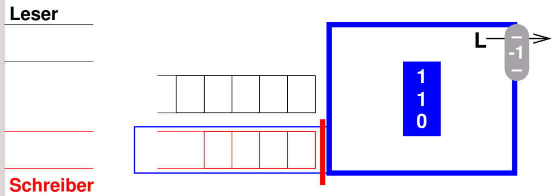


... 2. Leser-Schreiber-Problem



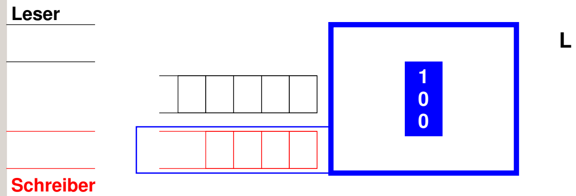
1 UNIX — Semaphore (8)

... **2. Leser-Schreiber-Problem**



1 UNIX — Semaphore (8)

... **2. Leser-Schreiber-Problem**



1 UNIX — Semaphore (8)

... **2. Leser-Schreiber-Problem**