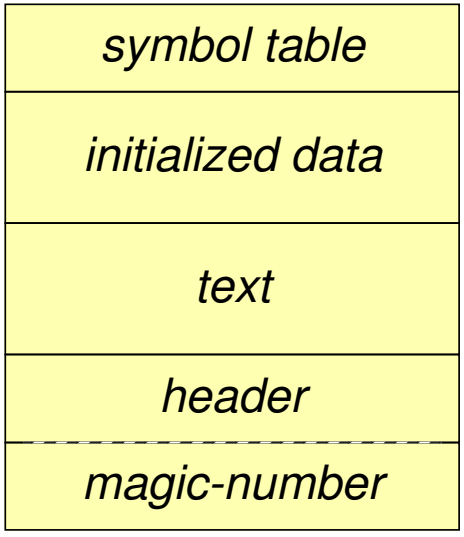
1 Speicherorganisation eines Programms

- 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)

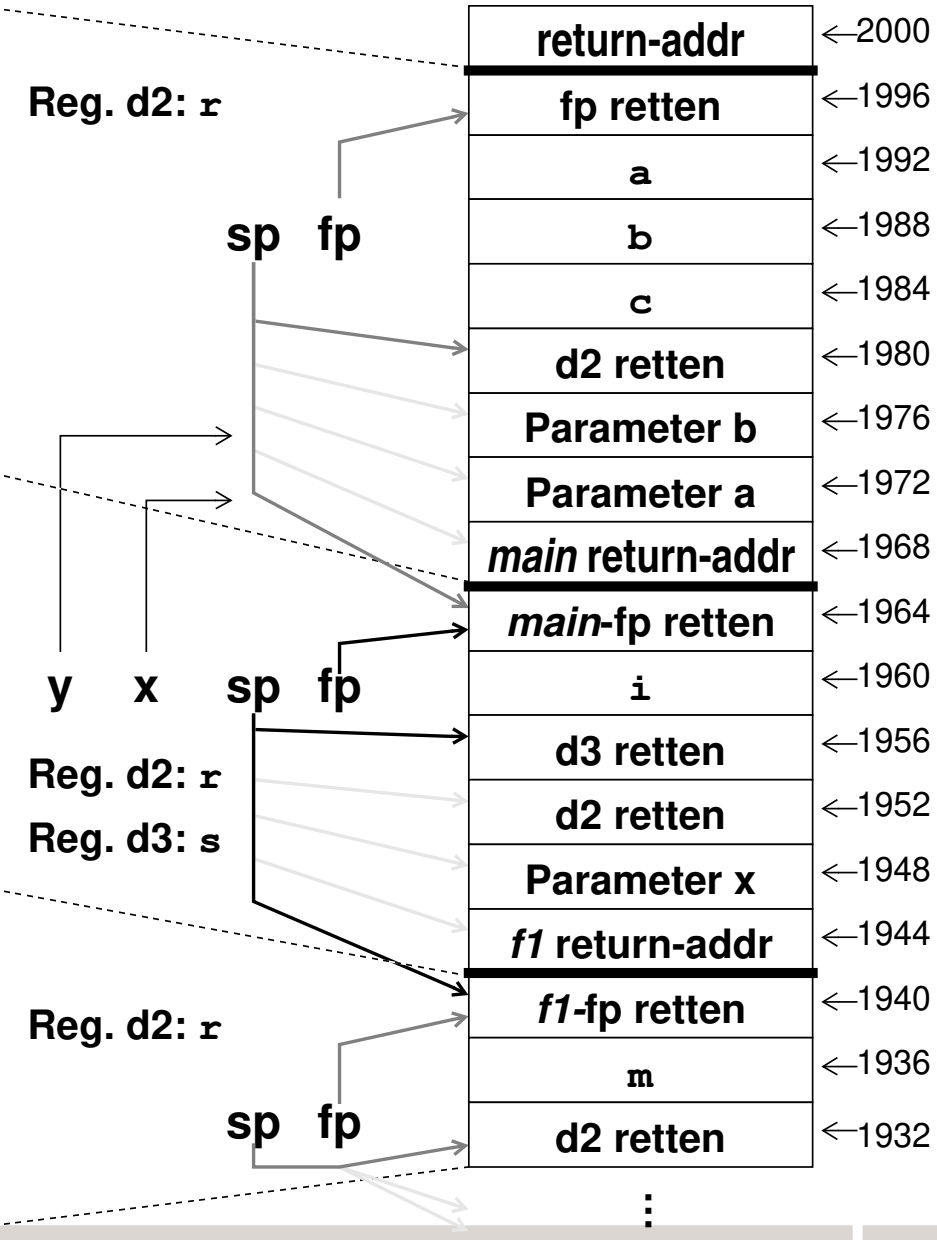
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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);
}
```



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Globale Prozeßverwaltungsdaten (3)

- Prozeßidentifikatoren und Prozeßgruppen

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

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

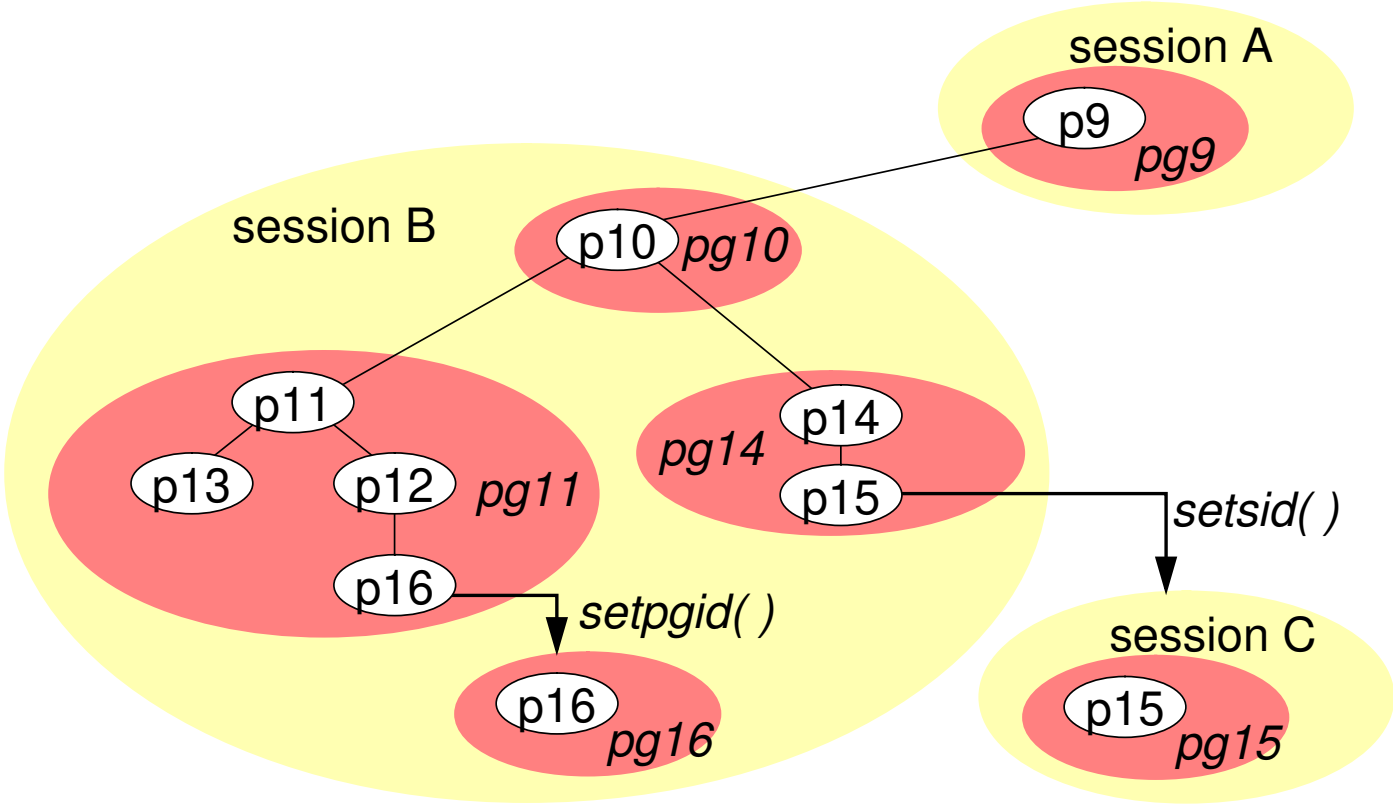
pgrp ***process group id*** = *pid* des Prozeßgruppenführers
setpgrp(2), getpgrp(2), ***setpgid(2)*** (POSIX)
(siehe auch Abschnitt "Sessions und Prozeßgruppen")

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( )

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Globale Prozeßverwaltungsdaten (7)

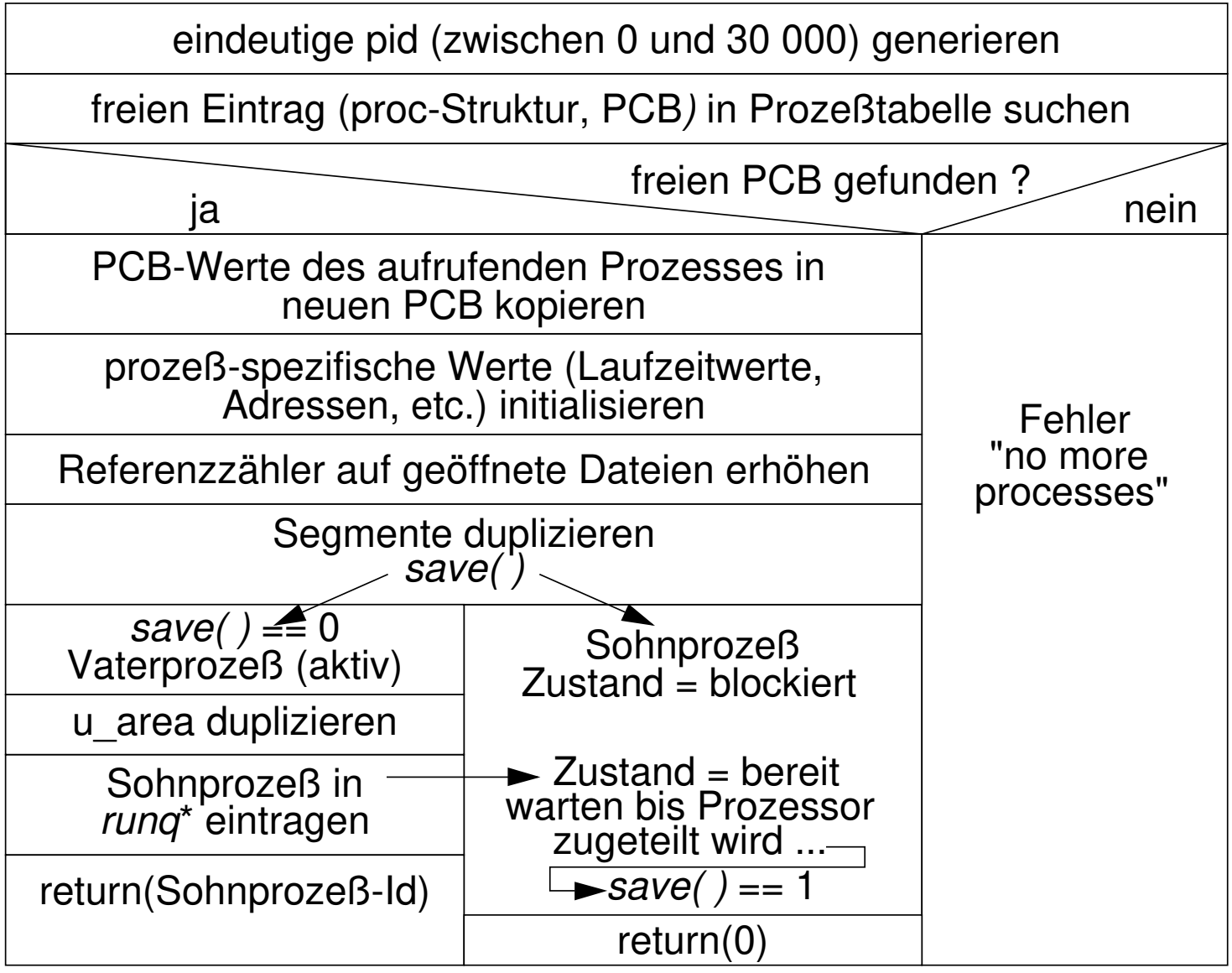
- … Sessions und Prozeßgruppen



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Ablauf

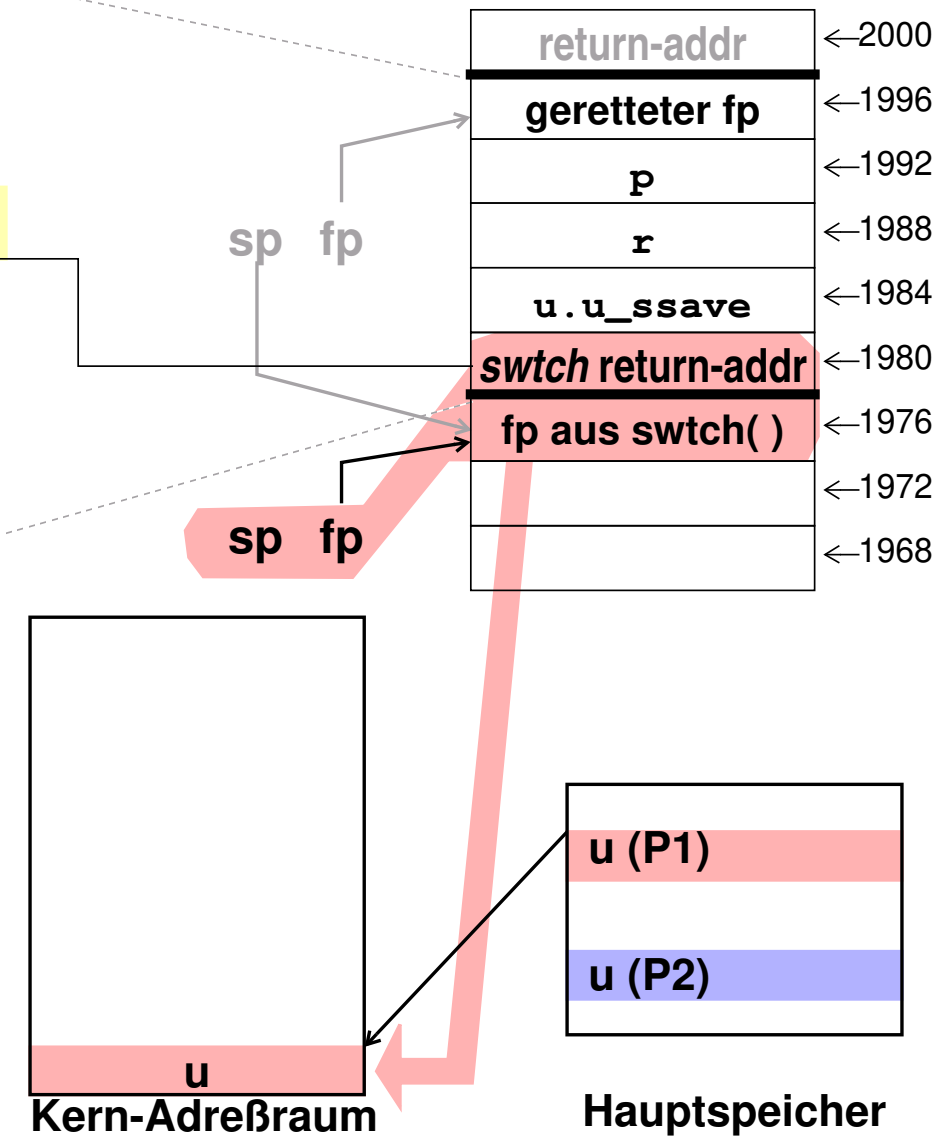
newproc( )



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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

```
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);
}
```



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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);
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

★ 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!

© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

... 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)

© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

... 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_mutex_unlock(&m1);
pthread_cond_signal(&c1);
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

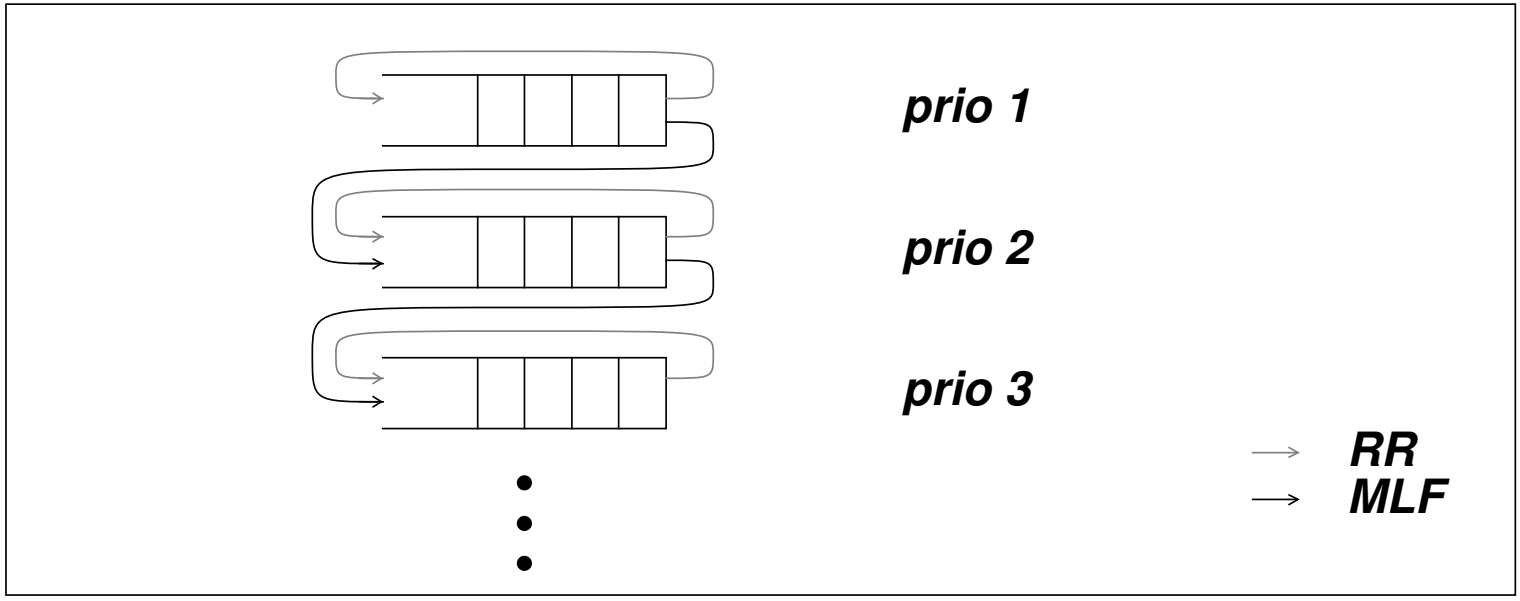
3 Koordinierung im UNIX-Kern (3) — sleep —

- 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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.10 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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 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

  $$p_usrpri = PUSER + \frac{p_cpu}{4} + 2 \cdot p_nice$$

  wird alle 40 ms für den gerade aktiven Prozeß ermittelt

  $$p_cpu = \frac{2 \cdot load}{2 \cdot load + 1} \cdot p_cpu + p_nice$$

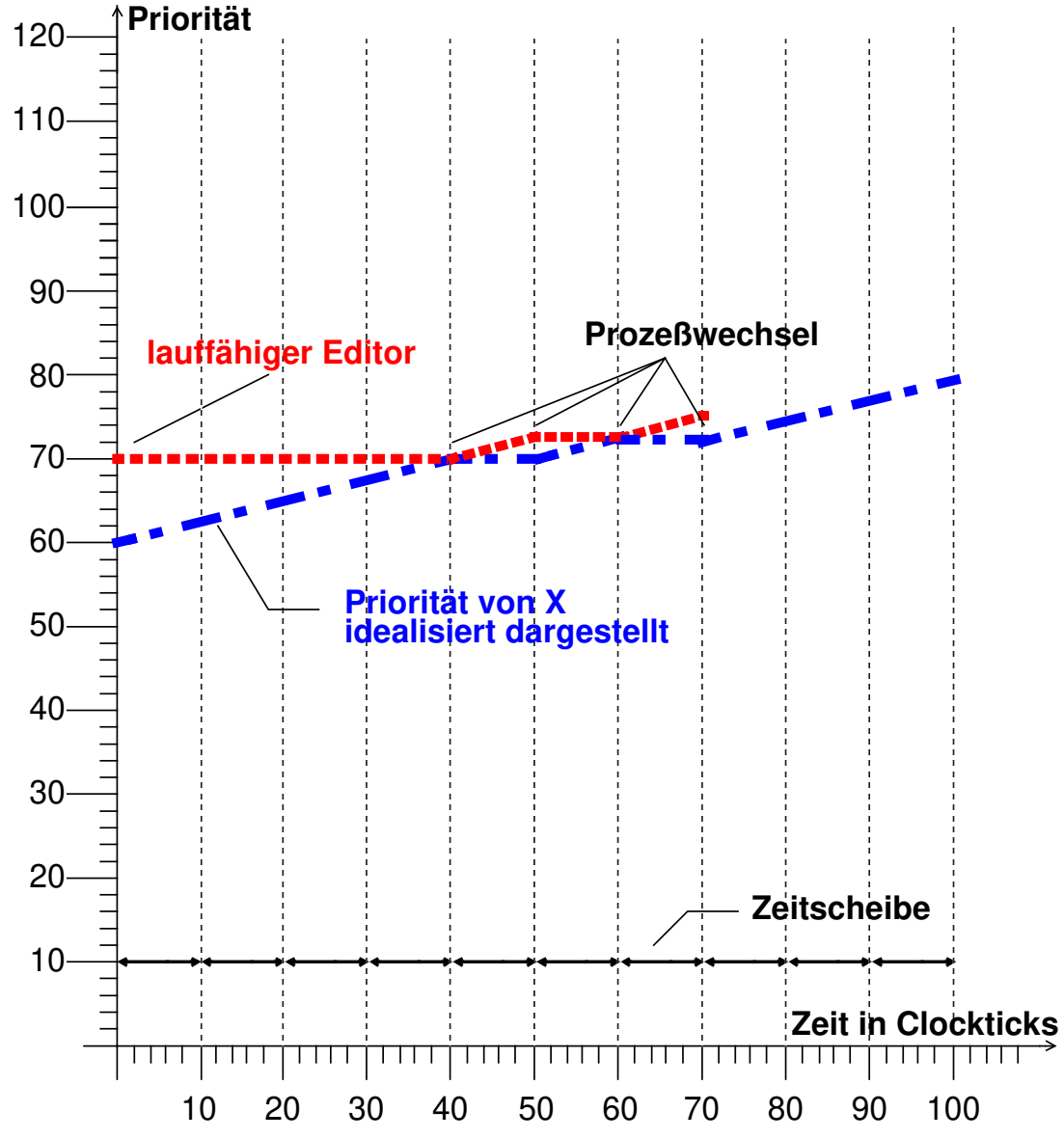
  einmal pro Sekunde ∀ Prozesse

  $p_cpu = p_cpu + 1$ alle 10 ms für den aktiven Prozeß

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Vergleich: SystemV ↔ BSD (2)

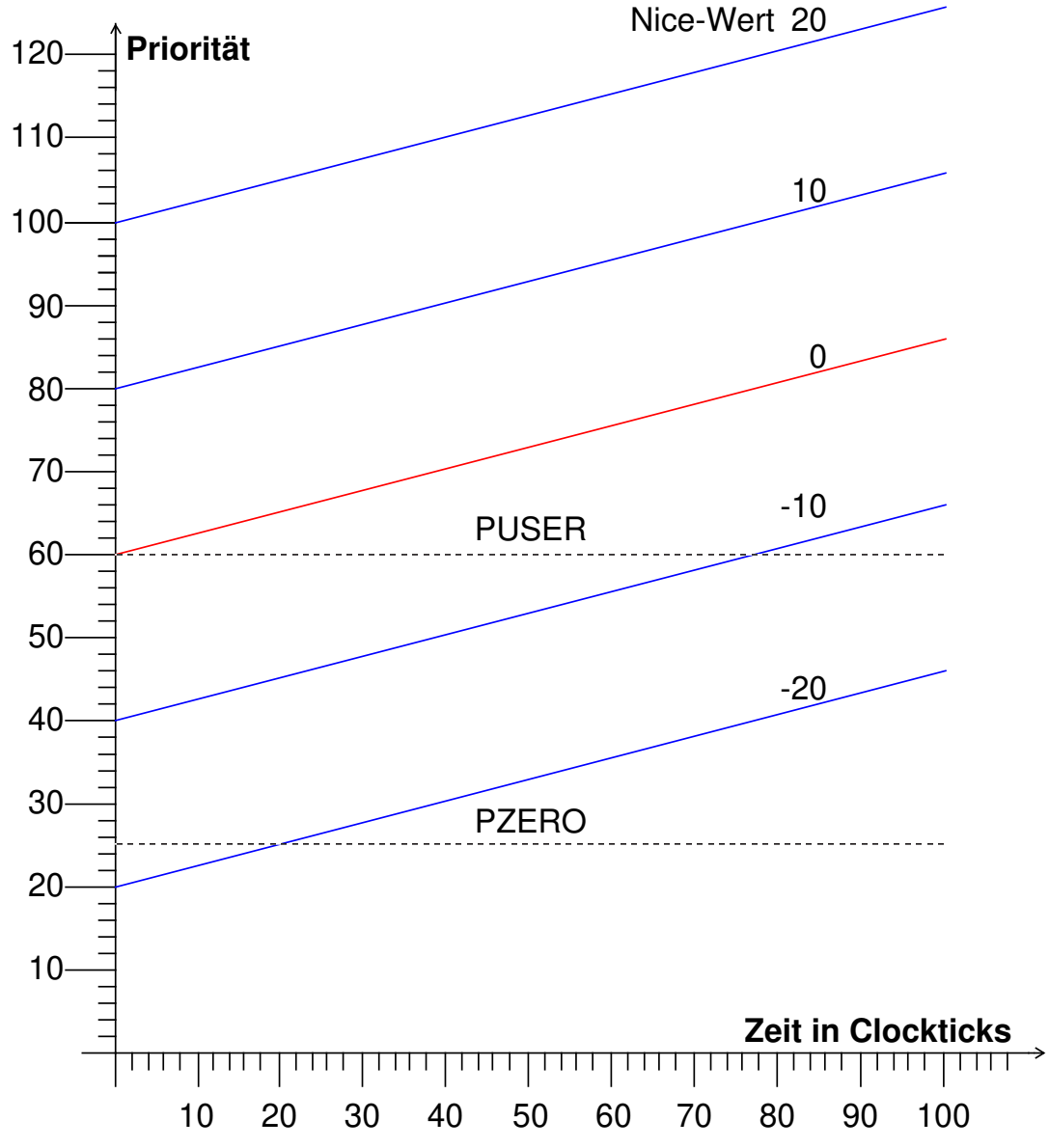
- Prioritätsverhalten und Prozeßwechsel in Berkeley-UNIX
  - Rechenprozeß X, interaktiver Prozeß (z. B. Editor),
  - gleiche Bedingungen wie Beispiel für SystemV



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Vergleich: SystemV ↔ BSD (6)

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



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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)

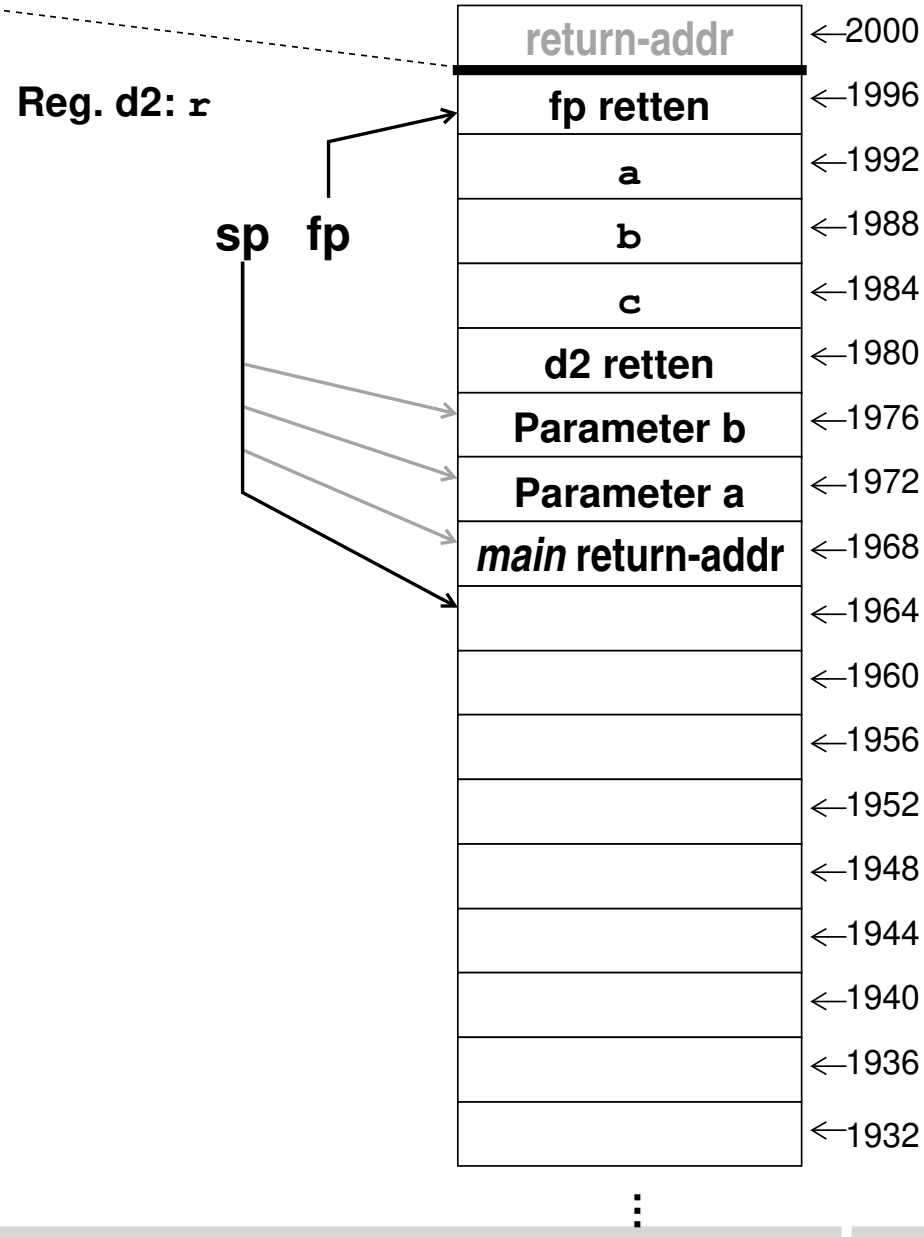
- drei Stufen:

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

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Stack mehrerer Funktionsaufrufe

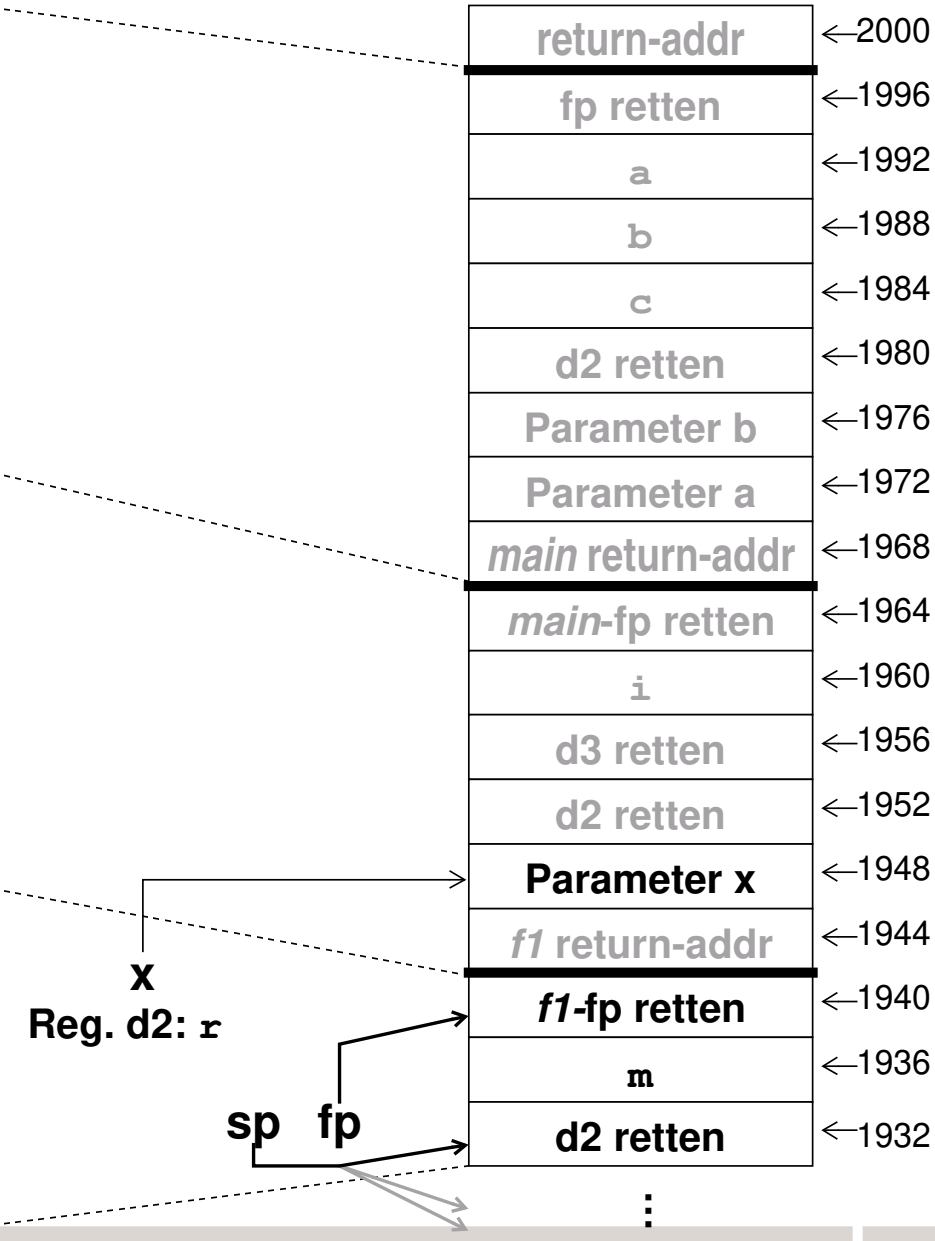
```
main() {
        int a, b, c;
        register r;

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

        f1(a, b);

        return(a);
}
```



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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);
}
```

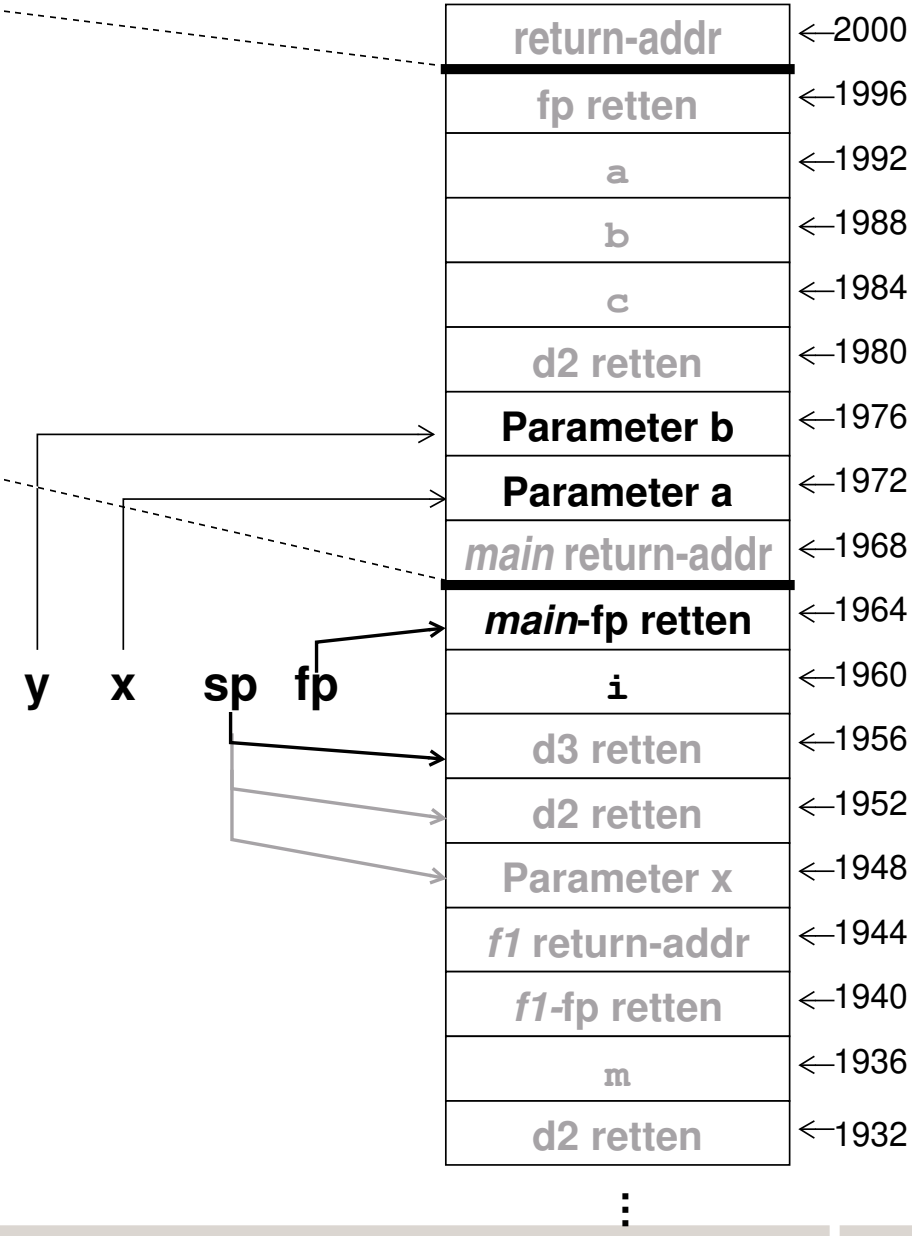


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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);
}
```

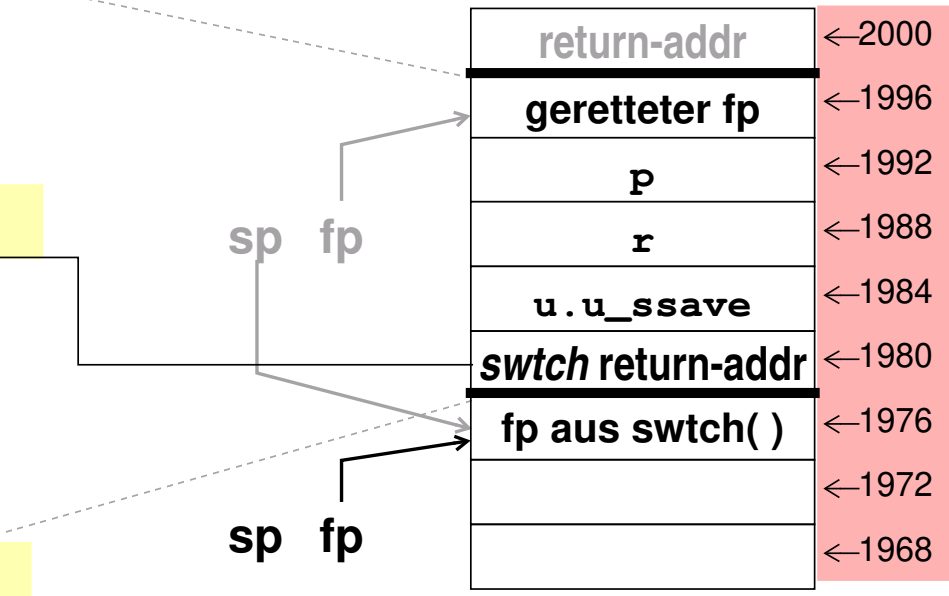


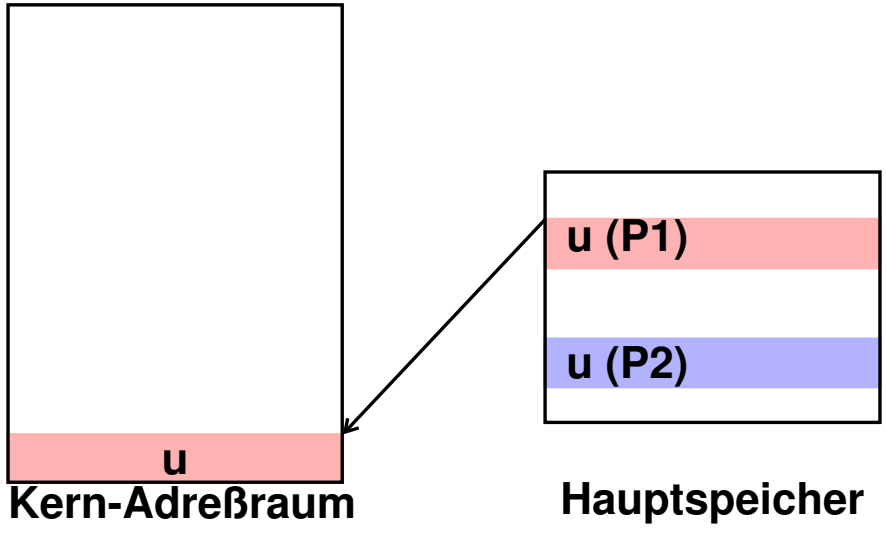
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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);
}
```



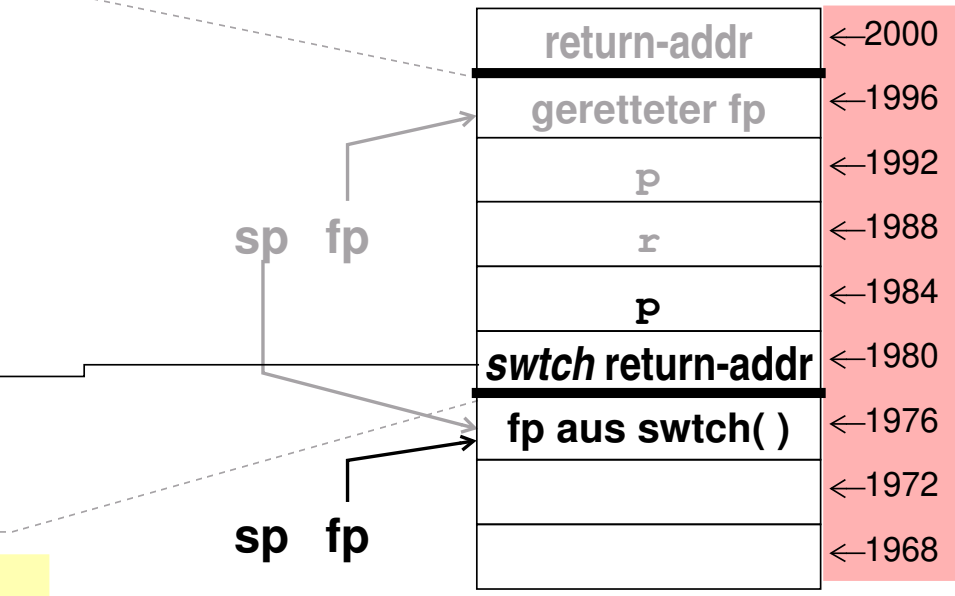


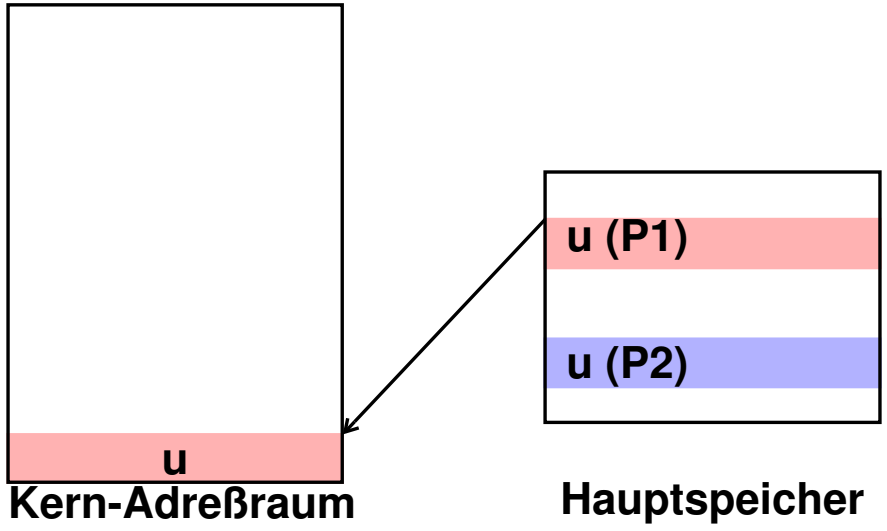
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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);
}
```



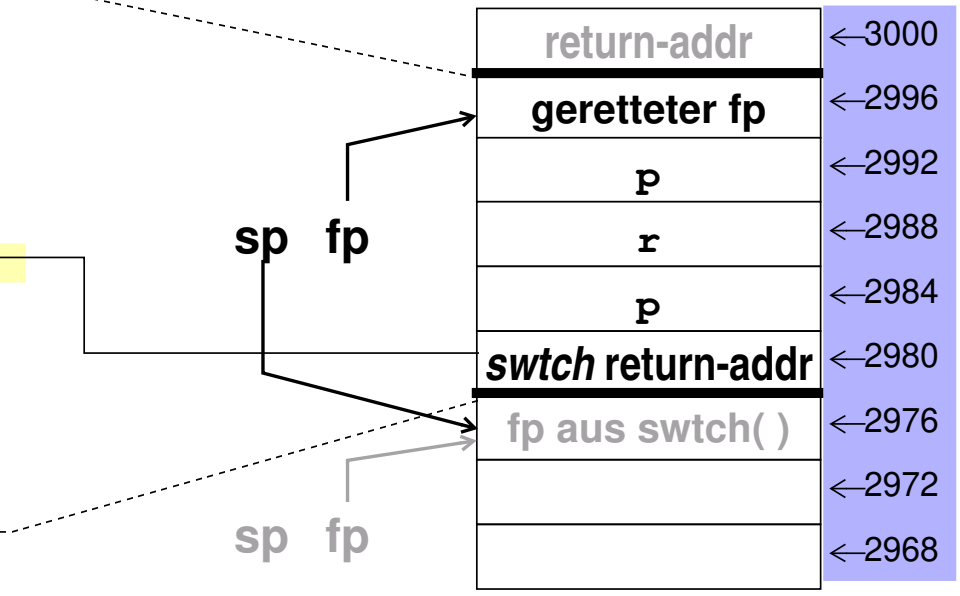


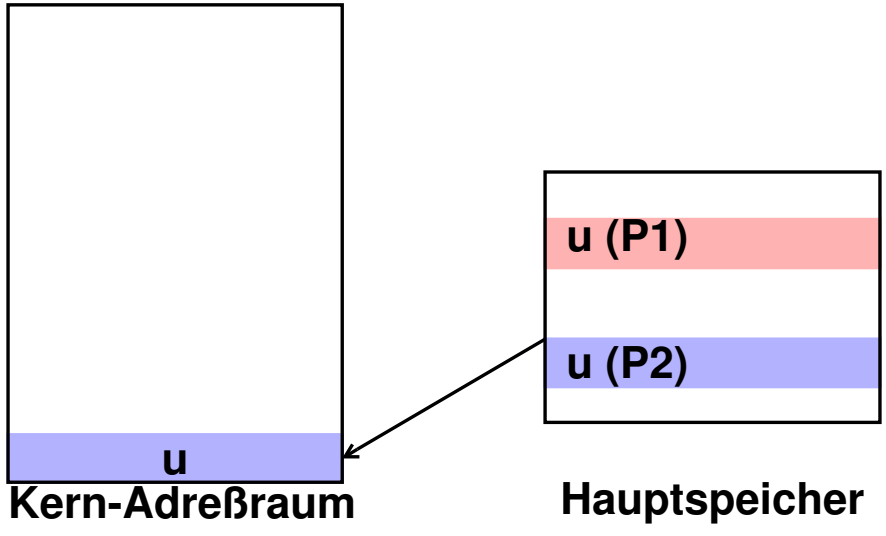
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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);
}
```





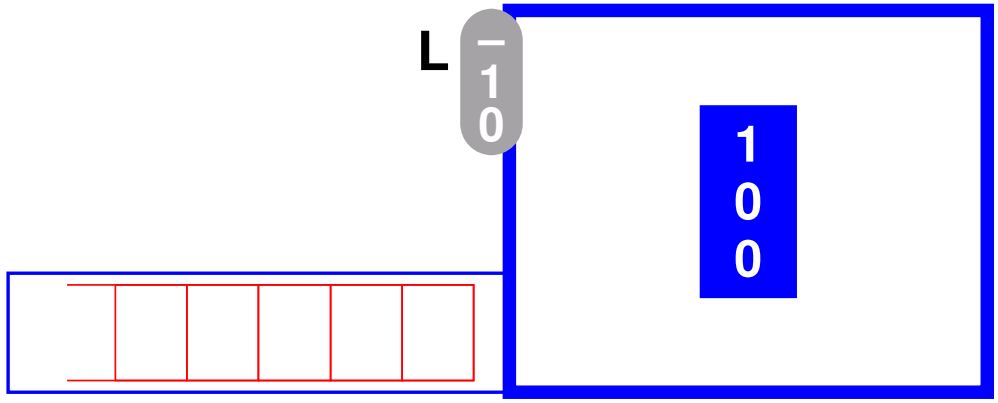
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

UNIX — Semaphore (8)

… 2. Leser-Schreiber-Problem

Leser

Schreiber

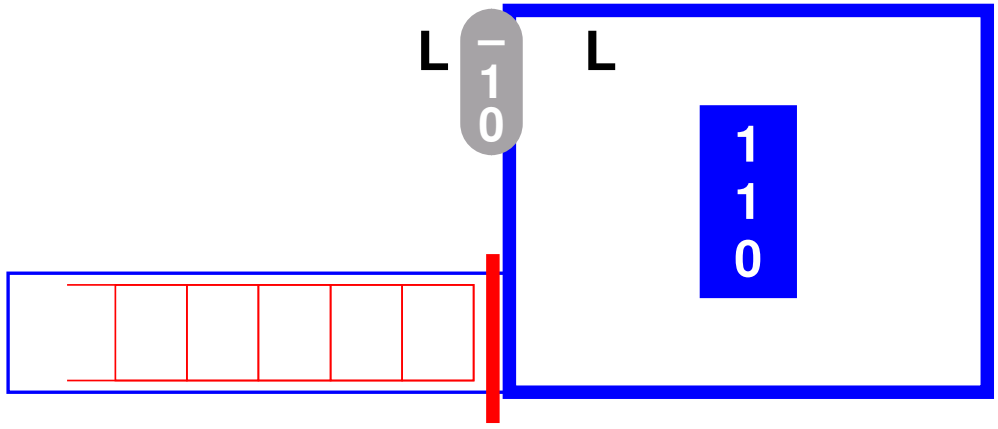


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

Leser

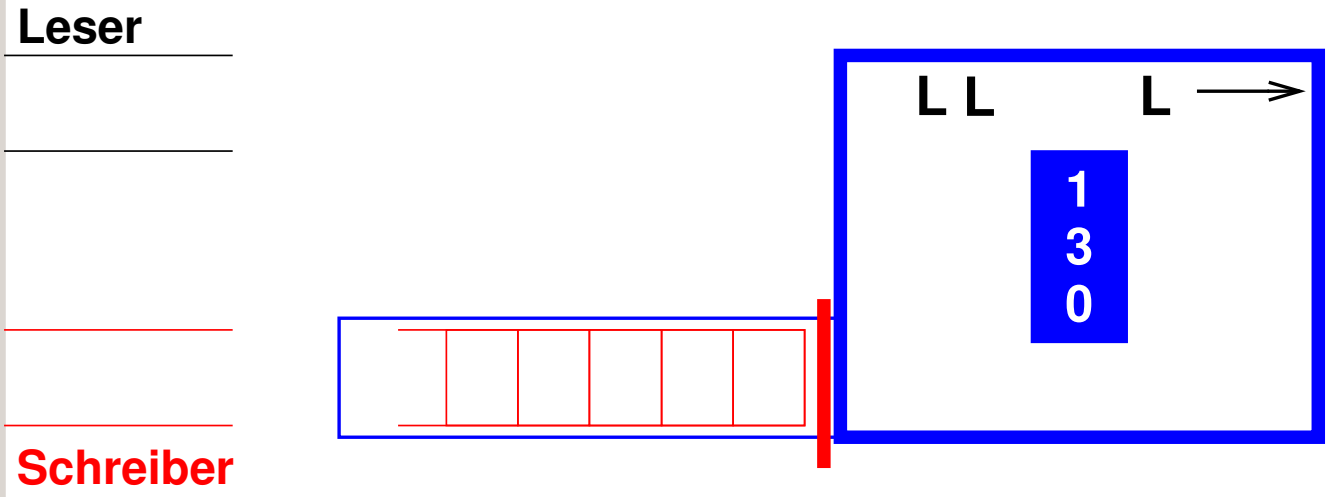


Schreiber

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

UNIX — Semaphore (8)

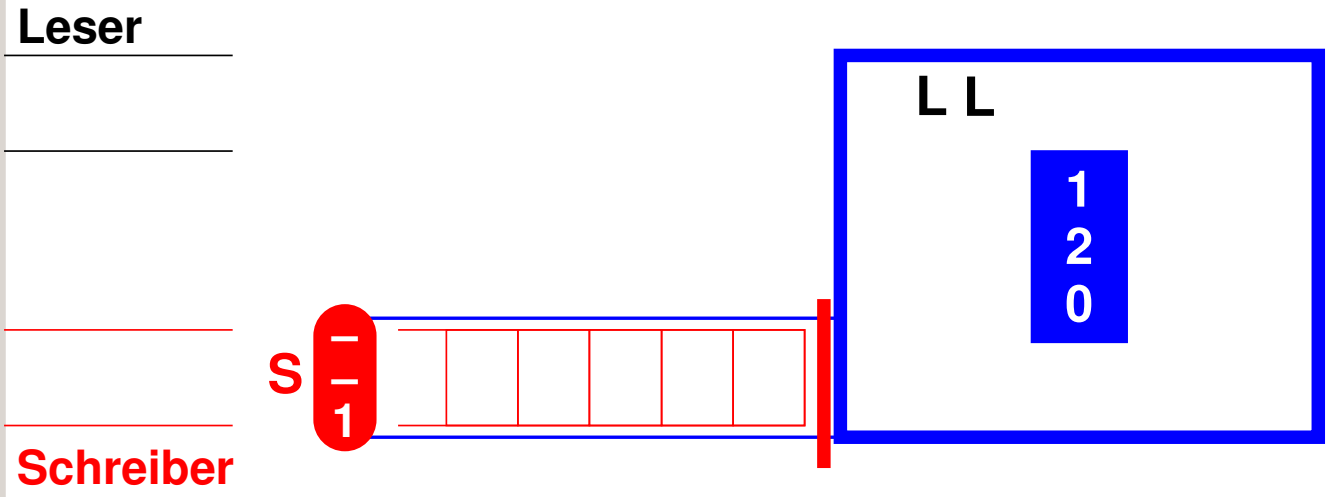
… 2. Leser-Schreiber-Problem



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 UNIX — Semaphore (8)

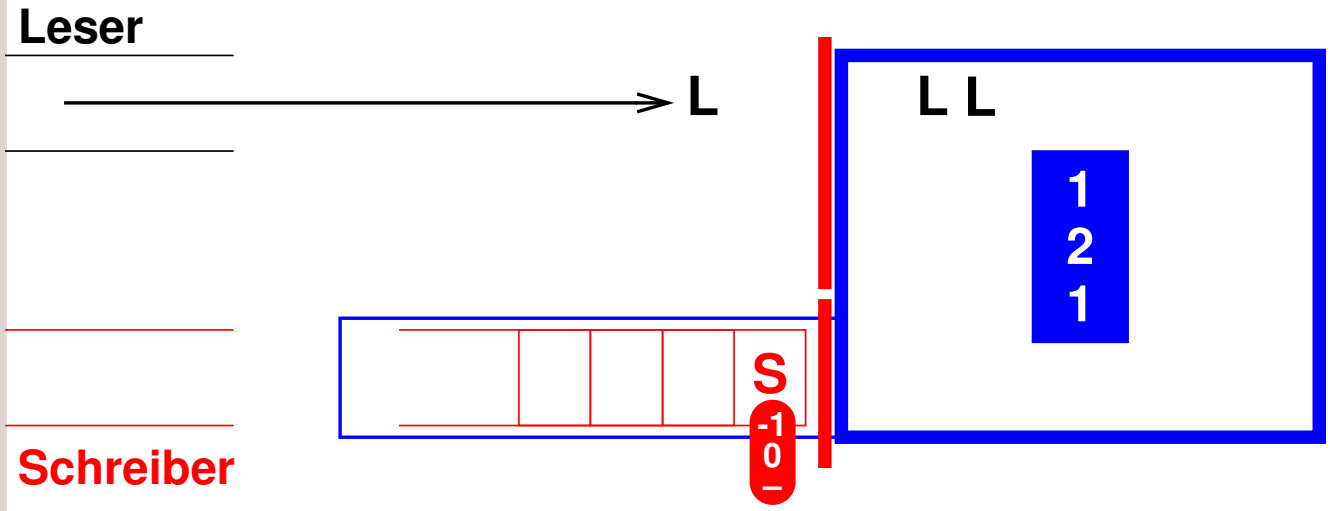
… 2. Leser-Schreiber-Problem



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

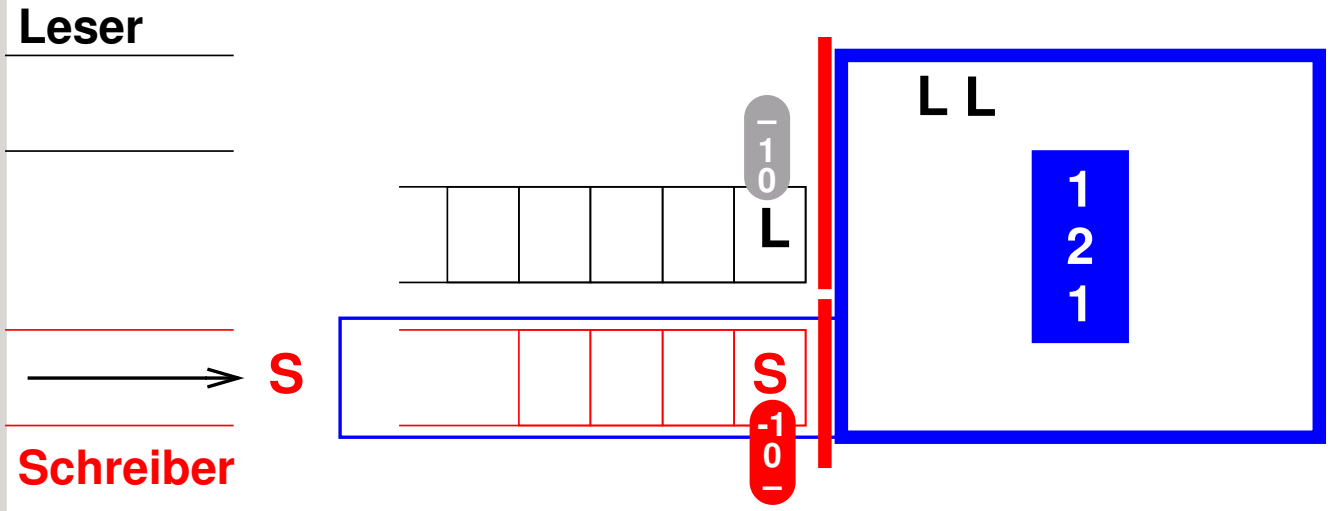
UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem



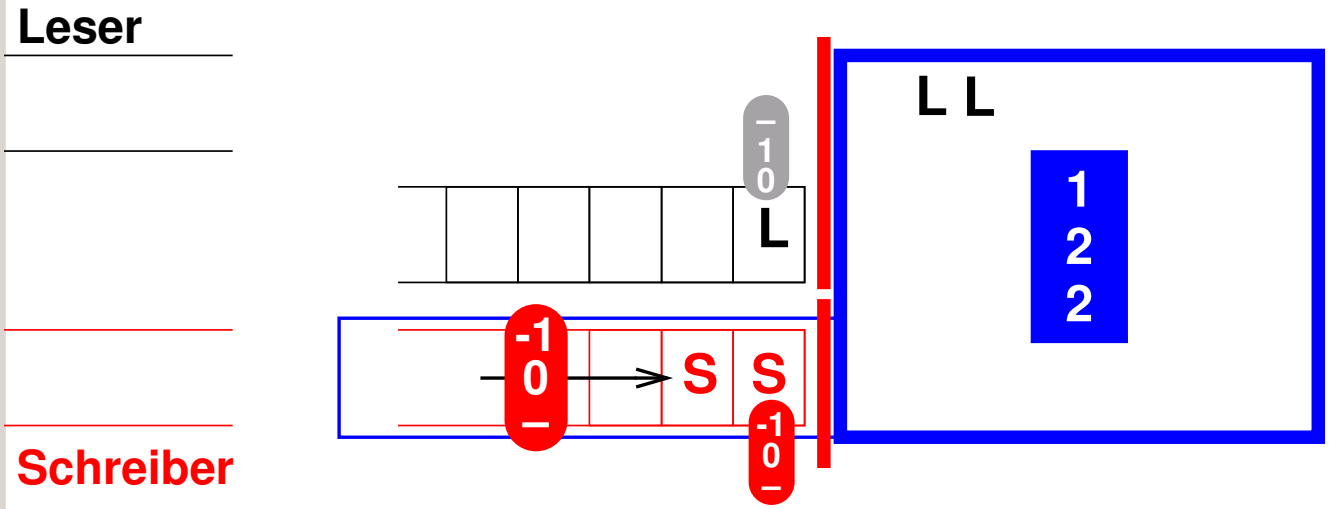
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

... 2. Leser-Schreiber-Problem



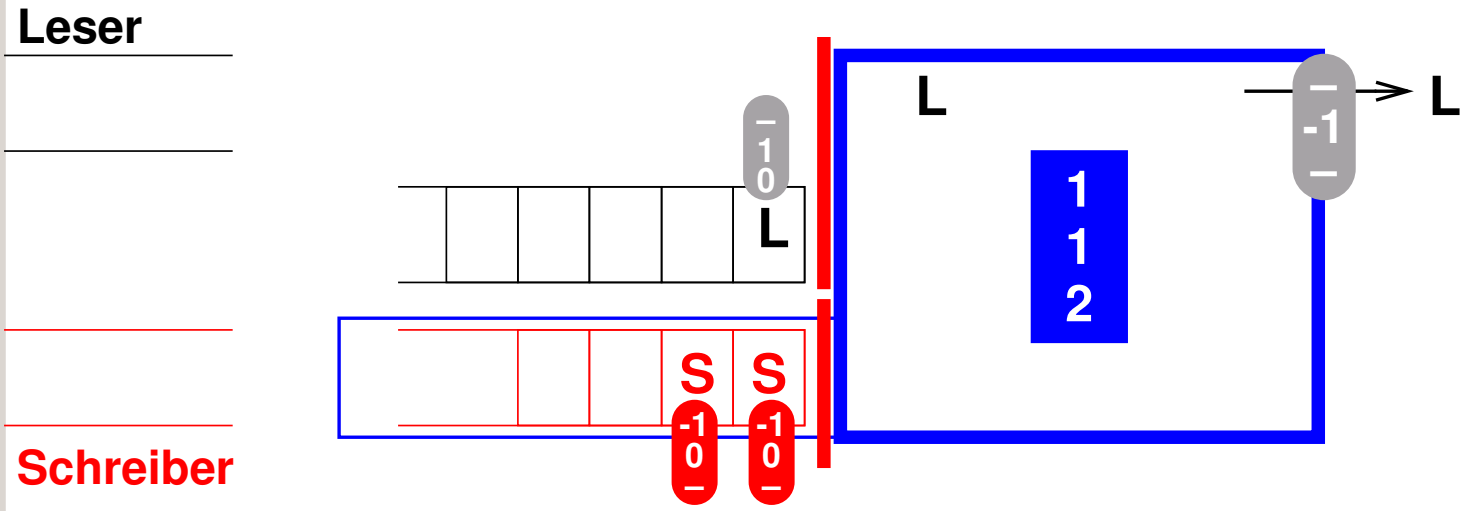
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

... 2. Leser-Schreiber-Problem



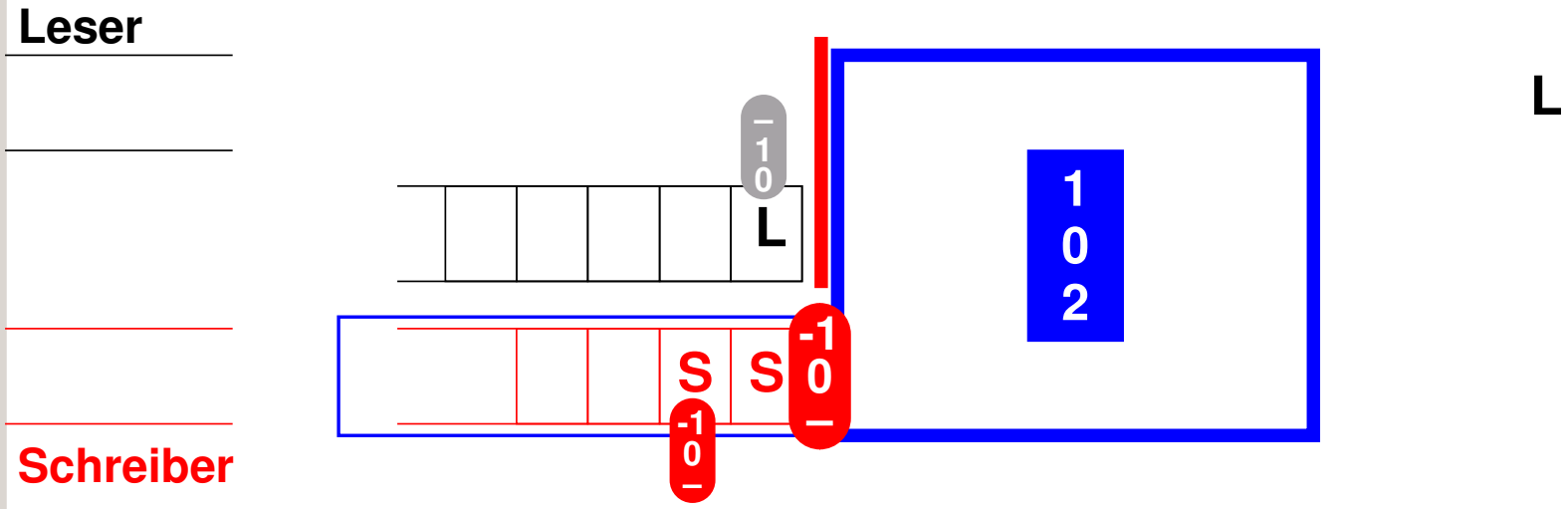
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

... 2. Leser-Schreiber-Problem



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

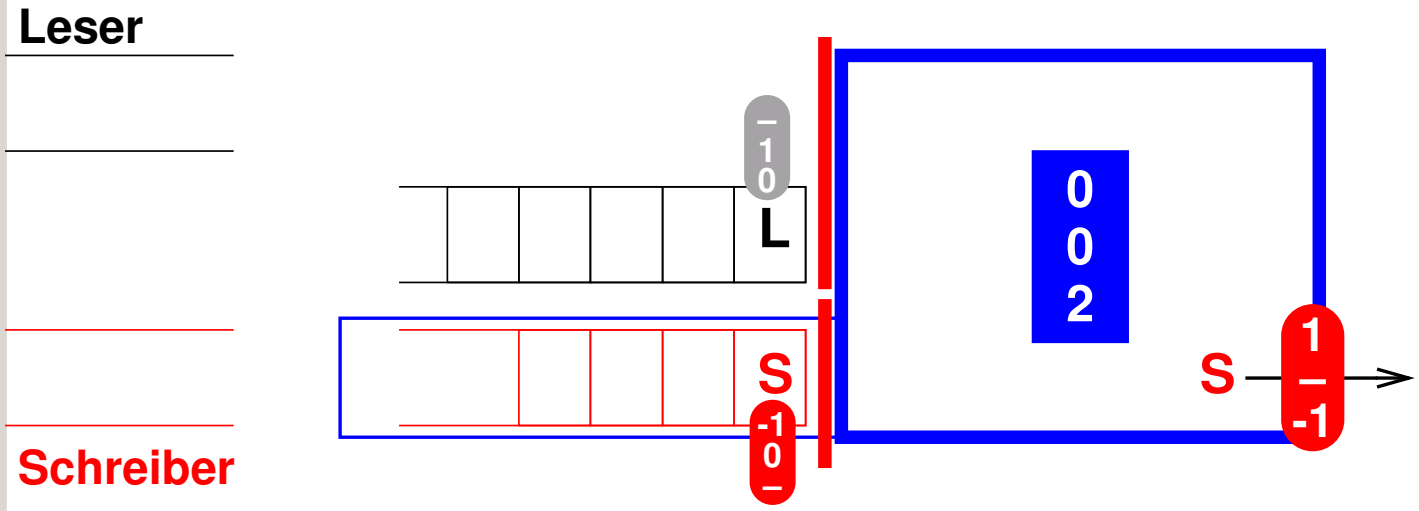
... 2. Leser-Schreiber-Problem



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

UNIX — Semaphore (8)

... 2. Leser-Schreiber-Problem

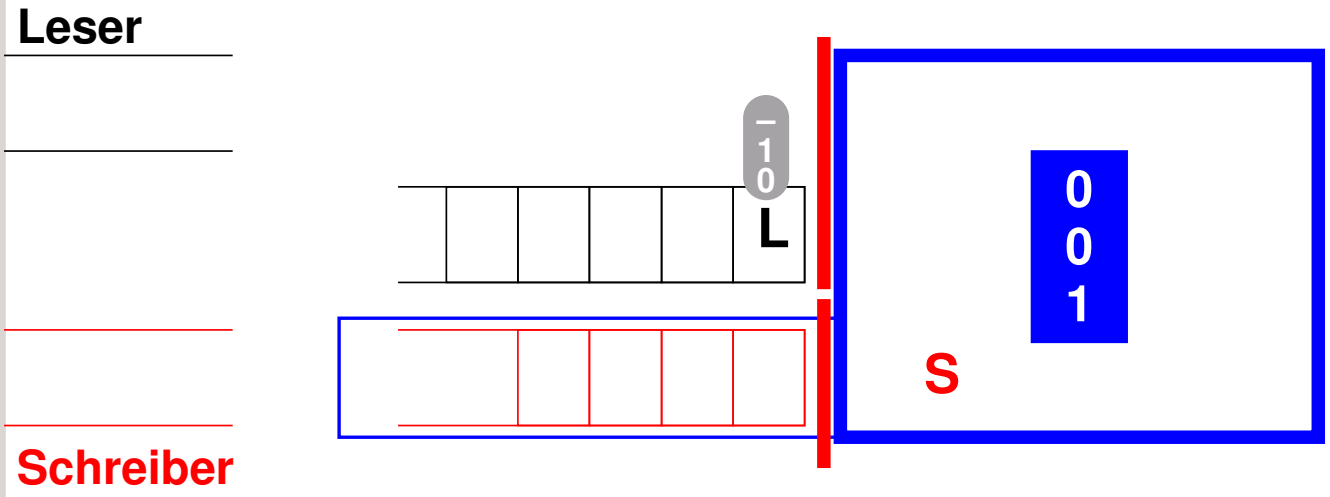


© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

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

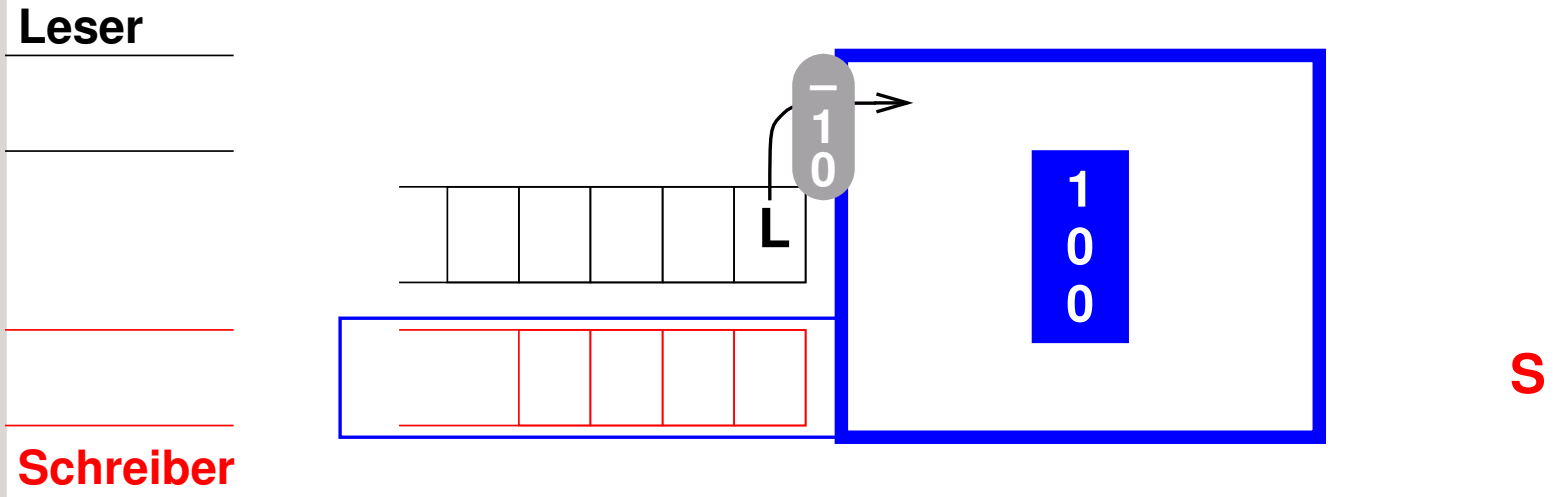


© Jürgen Kleinöder, Universität Erlangen-Nürnberg, IMMD IV, 1998

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

UNIX — Semaphore (8)

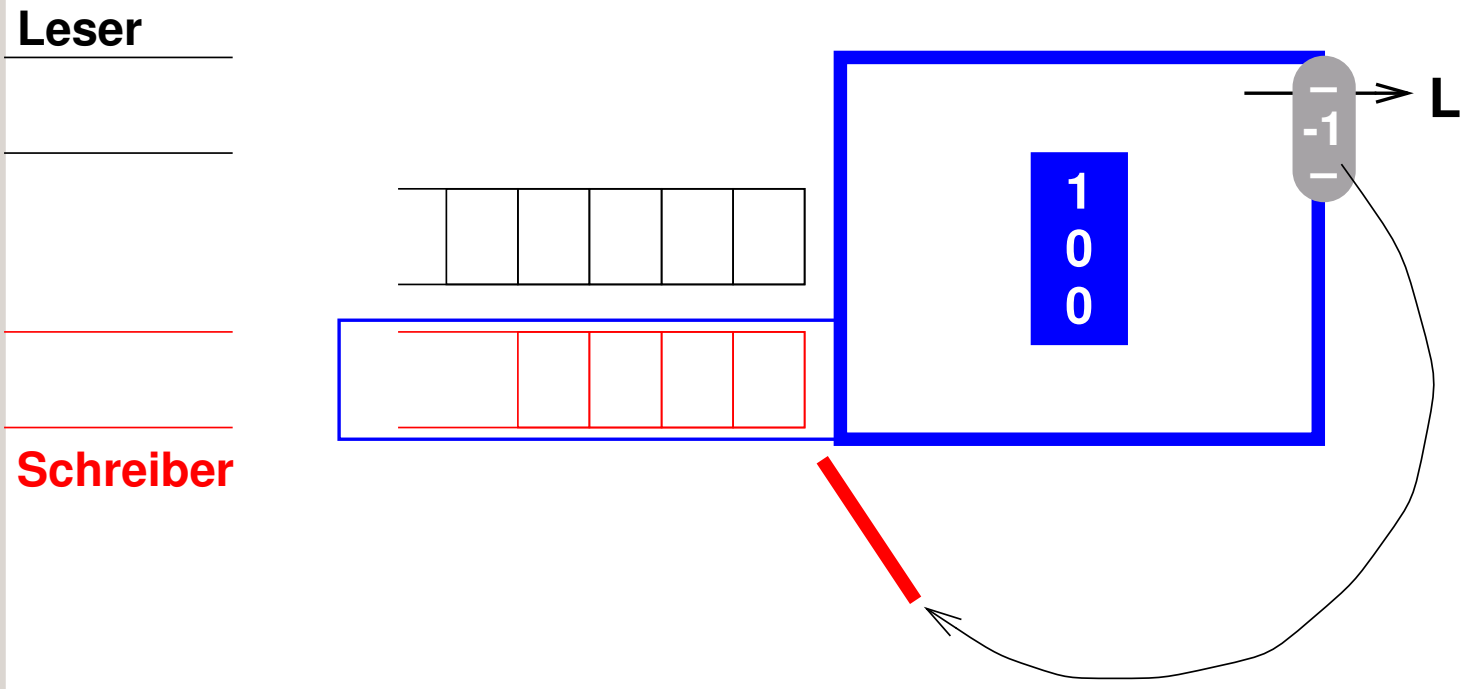
... 2. Leser-Schreiber-Problem



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 UNIX — Semaphore (8)

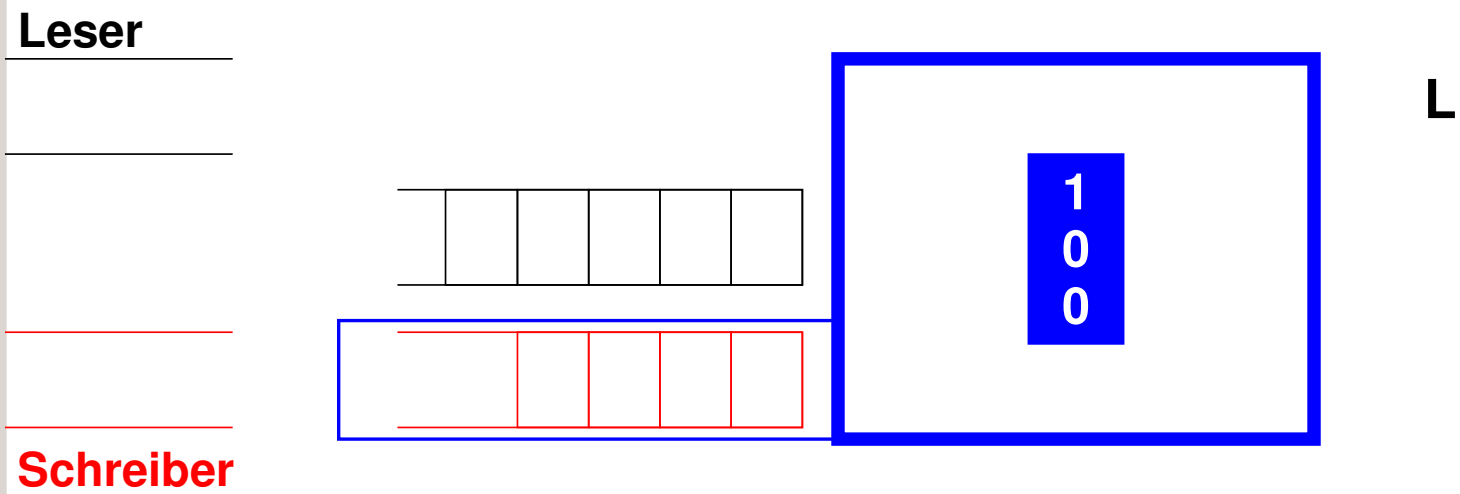
... 2. Leser-Schreiber-Problem



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

UNIX — Semaphore (8)

… 2. Leser-Schreiber-Problem



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.