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 —

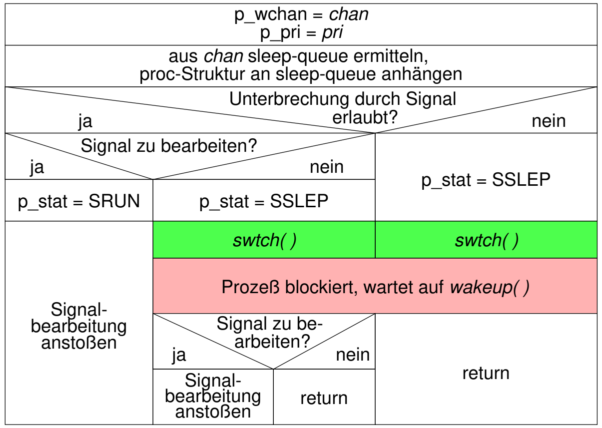
- es wird die Priorität angegeben, mit der der Prozeß wieder ausgeweckt werden soll
  - sehr hohe Priorität bei Vorgängen, die schnell fortgeführt werden müssen (z. B. weil noch *locks* gehalten werden)
- es wird ein Prozeßwechsel durchgeführt

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

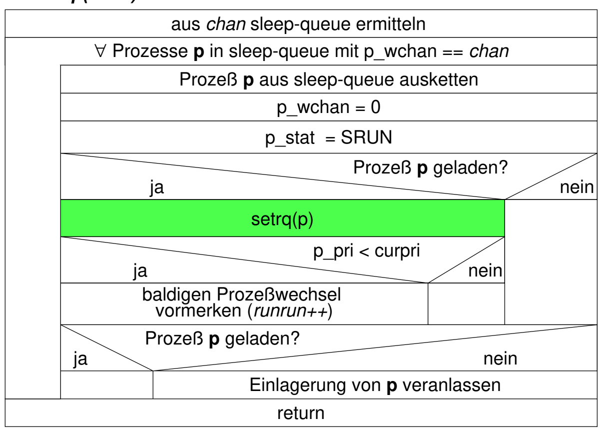
4 Koordinierung im UNIX-Kern (5)

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

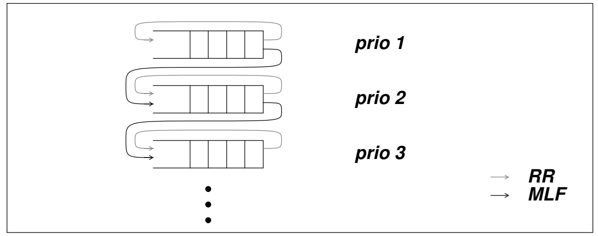
- jeder Prozeß prüft vor Verlassen des Systemkerns (am Ende eines Systemaufrufs) und vor sowie nach einem Blockieren (*sleep( )*), ob Signale für ihn vorliegen
- blockierte Signale werden nicht bearbeitet, bleiben aber in der Signal-Bitmaske `p_sig` vorgemerkt
- für alle anderen Signale wird der Reihe 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)

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

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

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

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

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

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 (2)

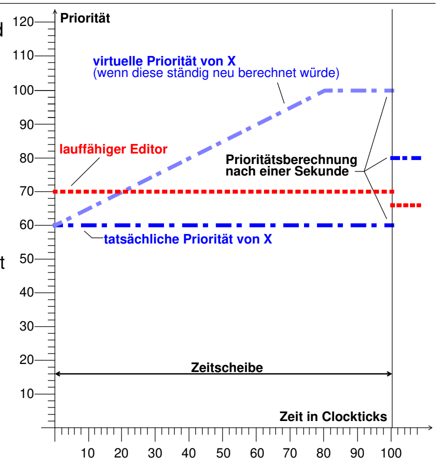
- Unterschiede zu SystemV
  - ◆ durch kürzere Zeitscheibe wird die CPU hochprioren Prozessen gleichmäßiger zugeteilt
  - ◆ Priorität wird regelmäßig und nicht abhängig von Systemaufrufen neu berechnet
  - ◆ *p_nice* hat stärkeren Einfluß auf *p_usrpri*
  - ◆ *p_nice* beeinflußt Veränderung von *p_cpu*
  - ◆ *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

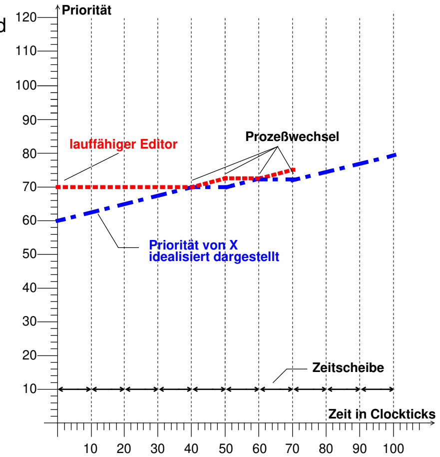
3 Vergleich: SystemV ↔ BSD

- 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



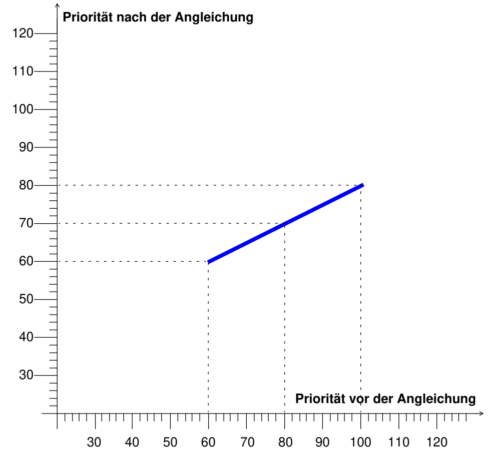
3 Vergleich: SystemV ↔ BSD (2)

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



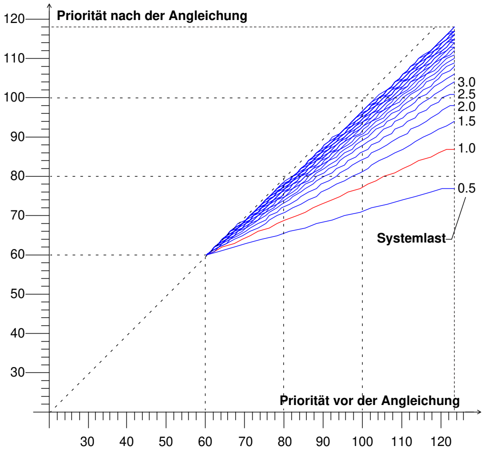
3 Vergleich: SystemV ↔ BSD (3)

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



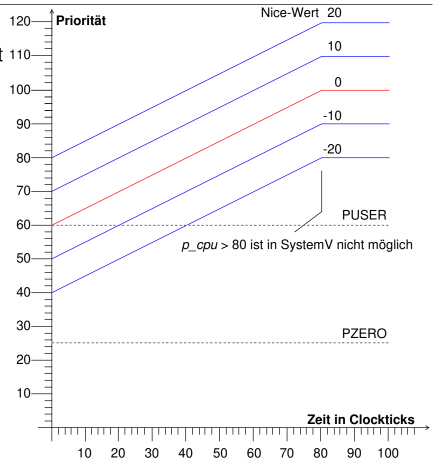
3 Vergleich: SystemV ↔ BSD (4)

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



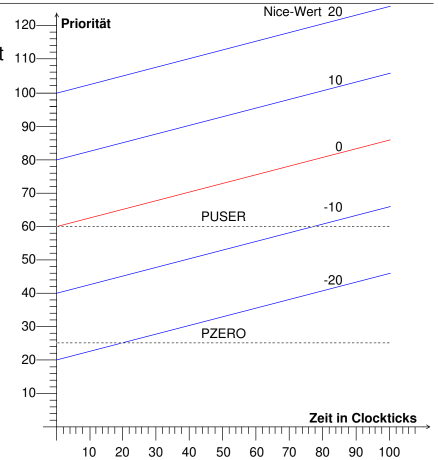
3 Vergleich: SystemV ↔ BSD (5)

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



3 Vergleich: SystemV ↔ BSD (6)

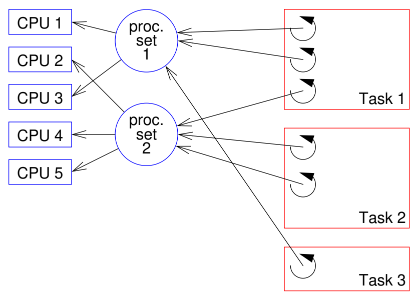
- Einfluß des *nice*-Wertes auf die Priorität eines Prozesses
  - ◆ Berkeley-UNIX



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