Betriebssystemtechnik

Prozesssynchronisation: Sperrverfahren

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

19. Juni 2012

Gliederung

Synchronisationskonzepte: Maschinenprogrammebene [5]

Alleinstellungsmerkmal dieser Abstraktionsebene ist allgemein die durch ein **Betriebssystem** erreichte funktionale Anreicherung der CPU, hier:

(a) in Bezug auf die Einführung des Prozesskonzeptes und

(b) hinsichtlich der Art und Weise der Verarbeitung von Prozessen

Techniken zur Synchronisation gleichzeitiger Prozesse können demzufolge auf Konzepte zurückgreifen, die die Befehlssatzebene nicht bietet

zu (a) die Möglichkeit, Prozessinkarnationen kontrolliert schlafen legen und wieder aufwecken zu können

- Bedingungsvariable, Semaphor
- (*sleeping*) *lock* ✗

zu (b) die Möglichkeit, den Zeitpunkt der Einplanung oder Einlastung solcher Inkarnationen gezielt vorgeben zu können

- Verdrängungssperre ✗

Mehrseitige Synchronisation: kritischer Abschnitt (KA)

Schutz kritischer Abschnitte durch **Ausschluss gleichzeitiger Prozesse** ist mit verschiedenen Ansätzen möglich

(a) asynchrone Programmunterbrechungen unterbinden, deren jeweilige Behandlung sonst einen gleichzeitigen Prozess impliziert

(b) Verdrängung des laufenden Prozesses aussetzen, die anderenfalls die Einlastung eines gleichzeitigen Prozesses bewirken könnte ✗

(c) gleichzeitige Prozesse allgemein zulassen, sie allerdings dazu bringen, die Entsperrung des KA eigenständig abzuwarten ✗

Alleingang (engl. *solo*) eines Prozesses durch einen kritischen Abschnitt sicherzustellen basiert dabei auf ein- und demselben **Entwurfsmuster**:

```
CS_ENTER(solo);
⋮
CS_LEAVE(solo);
```

`CS_ENTER` (a) `cli`, (b) NPCS *enter*, (c) *P*, *acquire*

`CS_LEAVE` (a) `sti`, (b) NPCS *leave*, (c) *V*, *release*

`solo` spezifiziert die fallabhängige Sperrvariable

Gliederung

Verdrängungsfreie kritische Abschnitte

NPCS, Abk. für (engl.) *non-preemptive critical section*: vgl. Kap VI, S. 24–26

Ereignisse, die zur Verdrängung eines sich in einem kritischen Abschnitt befindlichen Prozesses führen könnten, werden unterbunden

enter
- mögliche Verdrängung des laufenden Prozesses zurückstellen

leave
- die ggf. zurückgestellte Verdrängungsanforderung weiterleiten

guide
- Verdrängungsanforderung durch kritischen Abschnitt führen

Schutzvorrichtung (engl. *guard*): „Aufgaben durchschleusen“

Bitschalter (engl. *flag*) zum Sperren/Zurückstellen von Verdrängungen

Warteschlange zurückgestellter Verdrängungsanforderungen

Aussetzen der Verdrängung des laufenden Prozesses ist durch (einfache) Maßnahmen an zwei Stellen der Prozessverwaltung möglich:

- der **Einplanung** oder **Einlastung** eines laufbereiten Prozesses

Zurückstellung und Weiterleitung von Aufgaben

Universelle Schleuse (engl. *universal positing system*, UPS)

```
void ups_avert(ups_t *this) {
    this->busy = true;   /* defer tasks */
}
```

```
void ups_admit(ups_t *this) {
    ups_treva(this);     /* let pass tasks */
    if (ups_stock(this)) /* any pending? */
        ups_clear(this); /* forward tasks */
}
```

```
void ups_relay(ups_t *this, order_t *task) {
    if (ups_state(this)) /* defer? */
        ups_defer(this, task);
    else                 /* no, run task */
        job_enact(&task->work);
}
```

```
typedef struct job job_t;

struct job {
    void (*call)(job_t*);
};
```

```
typedef struct order {
    chain_t next;
    job_t work;
} order_t;
```

```
typedef struct ups {
    bool busy;
    queue_t load;
} ups_t;
```

- **zurückgestellte Prozeduraufrufe** (engl. *deferred procedure calls*)

Universelle Schleuse: Implementierungsmerkmale

avert	• ELOP: atomares setzen des Bitschalters, Abschnittsmarkierung
treva	• dito: atomares zurücksetzen des Bitschalters ($\neg$*avert*)

admit	• bedingte Weiterleitung zurückgestellter Prozeduraufrufe
	• die Überprüfung erfolgt außerhalb des markierten Abschnitts
	  ◦ beugt der „*lost wakeup*“-Problematik vor
	  ◦ erlaubt aber das Überholen „geschleuster“ Prozeduraufrufe

clear	• leeren der Liste zurückgestellter Prozeduraufrufe • der Vorgang läuft außerhalb des markierten Abschnitts ab
relay	• bedingtes Weiterleiten/Zurückstellen eines Prozeduraufrufs
state	• ELOP: atomares lesen des Kopfzeigers der Aufrufliste
defer	• füllen der Liste zurückgestellter Prozeduraufrufe
enact	• den Prozeduraufruf in die Tat umsetzen

Beachte: Operationen der Aufrufliste

- lassen sich gut **nichtblockierend synchronisiert** implementieren

Prozesseinplanung (bedingt) zurückstellen

Seitenpfad heraus aus der Unterbrechungsbehandlung bewachen

Augenmerk ist auf jene Einplanungsfunktionen zu legen, die als Folge der Behandlung asynchroner Programmunterbrechungen aufzurufen sind

i bei **Ablauf der Zeitscheibe** des laufenden Fadens

ii bei **Beendigung des E/A-Stoßes** eines wartenden Fadens

Beispiel: Zeitscheibenablauf

```
extern void act_check();                    /* scheduler's clock handler */

order_t sad_order = {{0}, {act_check}};     /* order to reschedule CPU */

void __attribute__ ((interrupt)) clock() {
    ...
    CS_GUIDE(&npcs, &sad_order);            /* release scheduler call */
    ...
}
```

`&npcs` • identifiziert den zu schützenden kritischen Abschnitt

Prozesseinlastung zurückstellen

Übergang vom Prozessplaner zum Prozessabfertiger bewachen

Augenmerk ist auf die Einlastungsfunktion zu legen, die ggf. als Folge der Prozesseinplanung vom Planer aufgerufen wird

- und zwar zum **Umschalten des Prozessors** auf einen anderen Faden

Planer und Abfertiger lose koppeln (durch eine Art *lazy binding*)

- Umschaltfunktion im Planer vorsehen: *shift*
- diese mit der des Umschalters assoziieren: *board*
- eine Brückenfunktion verbindet beide Einheiten: *serve*
- Auftragsdeskriptor (*order*) aufsetzen und mittels Steuerfunktion durchschleusen: *guide*

```
extern void act_board(action_t *);

typedef struct board {
    order_t main;    /* deferral request */
    action_t *plot;  /* parameter placeholder */
} board_t;

void sad_serve(job_t *task) {
    act_board(((board_t*)task)->plot);
}

board_t sad_order = {{{0}, {sad_serve}}, 0};

void act_shift(action_t *next) {
    sad_order.plot = next; /* thread to be boarded */
    CS_GUIDE(&npcs, &sad_order.main);
}
```

Schutz eines kritischen Abschnitts vor Verdrängung

Abbildung auf die Zurückstellung von Prozeduraufrufen

```
ups_t npcs = {false, {0, &npcs.load.head}};

#define CS_ENTER(cs)        ups_avert(cs)
#define CS_LEAVE(cs)        ups_admit(cs)
#define CS_GUIDE(cs, op)    ups_relay(cs, op)
```

Geschützter KA

```
/*atomic*/ {
    CS_ENTER(&npcs);
    :
    CS_LEAVE(&npcs);
}
```

Programmabschnitt atomar ausgelegt mittels **Verdrängungssperre**

- die Konstruktion `/*atomic*/ {...}` ist nichts weiter als „syntaktischer Zucker“
- sie macht kritische Abschnitte besser deutlich

Beachte: Unabhängige gleichzeitige Prozesse

- werden unnötig zurückgehalten, obwohl sie den KA nicht durchlaufen

Gliederung

Schlossvariable (engl. *lock variable*)

Datentyp, der zwei grundlegende Operationen definiert:

acquire (auch: *lock*) $\models$ Eintrittsprotokoll

- verzögert einen Prozess, bis das zugehörige Schloss offen ist
  - bei geöffnetem Schloss fährt der Prozess unverzögert fort
- verschließt das Schloss („von innen“), wenn es offen ist

release (auch: *unlock*) $\models$ Austrittsprotokoll

- öffnet ein Schloss, ohne den öffnenden Prozess zu verzögern

Implementierungen dieses Konzepts werden auch als **Schlossalgorithmen** (engl. *lock algorithms*) oder **Sperralgorithmen** bezeichnet

- je nachdem, welche Sicht eingenommen wird:
  - (a) ersteres, wenn der Fokus auf der *Datenstruktur* liegt
  - (b) letzteres, wenn die *Wirkung* des Verfahrens im Vordergrund steht
- die betrachteten Schlossalgorithmen wirken sperrend auf Prozesse

Schlossalgorithmus: Prinzip — mit Problem(en)

```
typedef struct lock {
    volatile bool busy;   /* status of critical section */
    ...                   /* optional stuff needed for other variants */
} lock_t;
```

Laufgefahr

```
void lv_acquire(lock_t *lock) {
    while (lock->busy);
    lock->busy = true;
}
```

```
void lv_release(lock_t *lock) {
    lock->busy = false;
}
```

- die Phase vom Verlassen der Kopfschleife bis zum Setzen der Schlossvariablen ist kritisch
- gleichzeitige Prozesse können das Schloss geöffnet vorfinden, dann jeweils schließen und gemeinsam den kritischen Abschnitt belegen

Abprüfen und setzen (engl. *test and set*, TAS) der Schloßvariablen

- muss als **Elementaroperation** und wirklich atomar ausgelegt sein

Schlossvariable atomar abprüfen und setzen

Kritischer Abschnitt

```
int tas(lock_t *lock) {
    bool busy;

    CS_ENTER(&lock->gate);
    busy = lock->busy;
    lock->busy = true;
    CS_LEAVE(&lock->gate);

    return busy;
}
```

`lock_t` mit **Sperroption** (`gate`):

- Verdrängungssperre *oder*
- Unterbrechungssperre

- untauglich für Multiprozessoren

Elementaroperation: x86

```
int tas(volatile bool *lock) {
    bool busy;

    busy = true;
    asm volatile("lock xchgb %0,%1"
        : "=q" (busy), "=m" (*lock)
        : "0" (busy));

    return busy;
}
```

`lock`
- „*read-modify-write*“
- atomarer Buszyklus

`xchgb`
- Operandenaustausch

- tauglich für Multiprozessoren

Umlaufsperre (engl. *spin lock*)

„Drehschloss“

```
void lv_acquire(lock_t *lock) {
    while (TAS(lock));
}
```

`TAS` kommt in zwei Varianten (S. 15):

a) $\mapsto$ `tas(lock)`
b) $\mapsto$ `tas(&lock->busy)`

Gefahr von Leistungsabfall

- pausenloses Schleifen allein nur mit `TAS`:
  a) erhöht das Risiko, Anforderungen von Programmunterbrechungen oder Prozessverdrängungen zu verpassen
     - die Unterbrechungs- bzw. Verdrängungssperre ist fast nur noch gesetzt
  b) hindert andere Prozessoren am Buszugang bzw. sorgt für eine überaus hohe Last im Kohärenzprotokoll des Zwischenspreichers
     - der Prozessor führt fast nur noch „read-modify-write“-Zyklen durch
- das Problem verschärft sich massiv, wenn (viele) gleichzeitige Prozesse den **Wettstreit** (engl. *contention*) um das „Drehschloss“ aufnehmen

Sensitive Umlaufsperre: *spin on read*

Stilllegung (engl. *quiescence*)

```
void lv_acquire(lock_t *lock) {
    do {
        while (lock->busy);
    } while (TAS(lock));
}
```

- nur lesender Zugriff beim Warten
- Zwischenspeicherzeile gemeinsam benutzbar (MESI, [4])
- einmal „*read-modify-write*“ (`TAS`)
- beruhigend für den Busverkehr

Beachte: Nicht jeder Prozessor hat einen *Datencache* — sowie:

kurze kritische Abschnitte

- Gefahr, zur einfachen TAS-Umlaufsperre (S. 16) zu degenerieren
- wenn die Laufzeit des Ein-/Austrittprotokolls kürzer ist als die für den Umlauf im Zwischenspeicher benötigte Anlaufzeit der CPU

Invalidierungen des Zwischenspeichers

- ein scheiterndes TAS löscht alle zwischengespeicherten `busy`-Kopien
- Zugriffsfehler unterbrechen dann den Umlauf im Zwischenspeicher

Zurückdrehen (engl. *backoff*) des Wettstreits

Festlegung einer im Konfliktfall zu durchlaufenden Wartezeit — analog zu Verfahren, die **Mehrfachzugriff** auf Kommunikationskanäle regeln:

i statisch zugewiesene Verzögerungen zum Senden (ALOHA [1])

ii dynamisch zugewiesene Verzögerungen bei Sendekonflikten

⇒ Ethernet [3], *binary exponential backoff*

Stauauflösung oder -verkürzung wird durch unterschiedliche Wartezeiten für die Kommunikationsteilnehmer bzw. Prozessoren erreicht

- hier: „stilllegen“ des Prozessor(kern)s vor dem nächsten TAS-Versuch
- idealerweise nur für die untereinander sich im Wettstreit befindlichen Prozesse, Prozessorkerne oder Prozessoren

Beachte: **Interferenz mit der Prozesseinplanung**

- Vergabe von Wartezeiten entspricht einer **Einplanungsentscheidung**
- Durchsetzung der Entscheidung des Prozessplaners ist gefährdet

⇒ Prioritätsverletzung oder -umkehr

Sensitive Umlaufsperre: *spin with backoff*

Eintrittsprotokoll

```
void lv_acquire(lock_t *lock) {
    while (TAS(lock))
        cpu_pause(lv_backoff[cpu_index()]);
}
```

```
unsigned int lv_backoff[NCORE];
```

CPU „stilllegen“

```
void cpu_pause(int spin) {
    while (spin--);
}
```

- statische Zuweisung

- Wettstreit (engl. *contention*) durch Rücktritt aus dem Wege gehend
- prozessor(kern)eigene Wartezeit („*backoff*“) nach gescheitertem `TAS`

Beachte: Wartezeit

- das Verfahren ist effektiv bei vergleichsweise vielen Prozessor(kern)en
  - bei kurzen kritischen Abschnitten steigt das Risiko von Leerläufen
  - Problemverschärfung bei Häufung (engl. *burst*) gleichzeitiger Zugriffe
- Faustregel: die mittlere Verzögerung sollte die Hälfte der Anzahl der umlaufenden Prozessoren entsprechen

Sensitive Umlaufsperre: *spin with exponential backoff*

Eintrittsprotokoll

```
void lv_acquire(lock_t *lock) {
    unsigned long hold = 1;

    while (TAS(lock)) {
        cpu_pause(hold);
        hold *= 2;
    }
}
```

- Ansatz wie zuvor (S. 19), jedoch:
  - dynamische Wartezeitzuweisung
  - exponentielles Wachstum
- die Wartezeit richtet sich nach der Ankunft gleichzeitiger Prozesse
  - zuerst scheitern $\rightsquigarrow$ kürzer warten
  - kürzer warten $\rightsquigarrow$ früher TAS
- ggf. Scheitern bei Neuzugängen

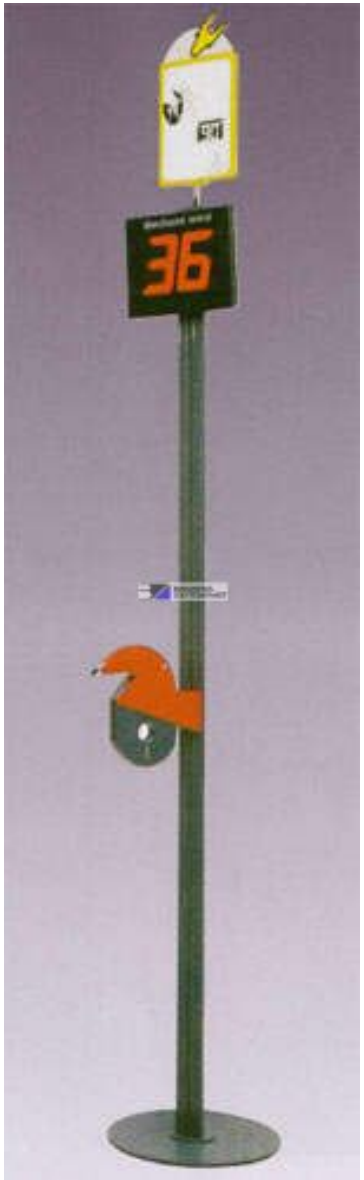
Beachte: Wartezeit

- sollte begrenzt sein, um nicht zu lange Leerlaufzeiten zu riskieren
- bei Erreichen des Grenzwerts ist sie dann für $N > 1$ Prozesse gleich
  $\Rightarrow$ Nachteile wie bei der einfachen Umlaufsperre (vgl. S. 16)

Beachte: Verhungerungsgefahr (mit beiden Varianten)

- Neuzugänge können Wiederholer scheitern lassen, überholen diese

Eindämmung von Wettstreit per Wartemarke (engl. *ticket*)



Vorbild für den Ansatz liefern Personenaufrufanlagen mit Wartemarkenspendern

- Kunden ziehen eine Wartemarke mit fortlaufender Nummer oder bekommen diese zugewiesen
- für die Nummernverwaltung sorgt eine Aufrufanlage, sie teilt Bearbeitern die nächste Nummer zu
- die Aufrufanlage zeigt an, welche Nummer als nächste an der Reihe ist und bearbeitet werden wird

Übertragen auf die Umlaufsperre

Kunde	• KA anfordernder Prozess(or(kern))
Bearbeiter	• KA belegender Prozess(or(kern))
Aufrufanlage	• atomarer Zähler

Sensitive Umlaufsperre: *spin with proportional backoff*

Schloss mit Wartemarken

```
typedef struct lock {
    unsigned int next;    /* number being served next */
    unsigned int this;    /* number being currently served */
} lock_t;
```

Eintrittsprotokoll

```
void lv_acquire(lock_t *lock) {
    unsigned int self = FAI((int *)&lock->next);

    do cpu_pause(self - lock->this);
    while (self != lock->this);
}
```

Austrittsprotokoll

```
void lv_release(lock_t *lock) {
    lock->this += 1;
}
```

FAI (x86)

```
INLINE int fai(int *ref) {
    int aux = 1;

    asm volatile (
        "lock\n\t"
        "xaddl %0,%1"
        : "=g" (aux), "=g" (*ref)
        :  "0" (aux),  "1" (*ref));

    return aux;
}
```

Umlaufsperre mit anteiligen Wartezeiten

Verwendung von Wartemarken reduziert die Anzahl atomarer Operationen im Eintrittsprotokoll einer Umlaufsperre auf eins

- die Vergabe der Wartemarken durch **FAI** geschieht strikt nach FCFS
- der Prozess(or(kern)) wartet daraufhin, bis seine Nummer dran ist
- kein anderer Prozess(or(kern)) wartet auf dieselbe Nummer

Wartezeiten lassen sich einfach proportional zur Anzahl der bereits auf Eintritt in den kritischen Abschnitt wartenden Prozesse bestimmen:

- Differenz von eigener Wartenummer und der des den kritischen Abschnitt gerade belegenden Prozesses: $Ticket_{self} - Ticket_{current}$

Beachte: Kohärente Zwischenspeicher

- das Verfahren ist eine Variante von „*spin on read*“ (vgl. S. 17)

$\Rightarrow$ Zähler (`next`, `this`) verschiedenen Zwischenspeicherzeilen zuteilen

Umlaufsperre mit Wartemarken $\rightsquigarrow$ FCFS

Verhungerung von Prozessen im Eintrittsprotokoll **ist ausgeschlossen**, wenn sich alle Prozesse an das Austrittsprotokoll halten

- zudem sollten die Zählergrößen der maximalen Anzahl gleichzeitiger Anforderungen Rechnung tragen

Gefahr von **Interferenz** mit Verfahren zur Prozesseinplanung, die den Prozessor(kern) eben nicht nach FCFS zuteilen (vgl. auch S. 18)

- ist vor allem besonders kritisch für **echtzeitabhängige Prozesse**
  - Prioritätsverletzung oder -umkehr kann die Folge sein
  - Prozesse laufen dadurch Gefahr, ihre Termine zu verpassen
- für andere Prozessarten ist dies weniger bis überhaupt nicht kritisch

Beachte: Dauer/Länge von kritischen Abschnitten [2, S. 27]

- ist wegen **Verdrängung** schwer einschätzbar bis unvorhersagbar:
  - i von Fäden, die sich in kritischen Abschnitten befinden
  - ii von Zwischenspeicherzeilen, die zur Arbeitsmenge dieser Fäden zählen
- überhaupt bringen **Unterbrechungen** extrem hohe Komplexität ein

Bestimmung der Laufzeitlänge kritischer Abschnitte

Atomarer Zähler

```
#include "luxe/lock.h"

typedef struct {
    volatile int value;
    lock_t lock;
} aint_t;

void aint_update(aint_t *ref, int val) {
    lv_acquire(&ref->lock);
    ref->value += val;
    lv_release(&ref->lock);
}
```

logisch
- eine Zeile C
- drei Zeilen ASM

real
- neun Zeilen ASM
- und mehr...

gcc -O6 -S

```
aint_update:
  subl  $28, %esp          # allocate local stack space
  movl  %ebx, 16(%esp)     # save non-volatile register
  movl  32(%esp), %ebx     # grab pointer to atomic counter
  movl  %esi, 20(%esp)     # save non-volatile register
  movl  %edi, 24(%esp)     # dito
  movl  36(%esp), %edi     # grab actual increment parameter
  leal  4(%ebx), %esi      # make reference to lock variable
  movl  %esi, (%esp)       # pass it as actual parameter
  call  lv_acquire         # enter critical section
  movl  (%ebx), %eax       # load counter
  leal  (%edi,%eax), %eax  # compute new value
  movl  %eax, (%ebx)       # store counter
  movl  16(%esp), %ebx     # restore non-volatile register
  movl  %esi, 32(%esp)     # pass lock reference
  movl  24(%esp), %edi     # restore non-volatile register
  movl  20(%esp), %esi     # dito
  addl  $28, %esp          # free local stack space
  jmp  lv_release          # leave critical section
```

Abstraktion und Fähigkeiten von Kompilierern lässt kritische Abschnitte größer werden, als sie wirklich sind

- automatisierte statische Programmanalyse ist bestenfalls hinreichend
- die Komplexität wird erst sichtbar auf Assembliersprachenebene

Sensitive Umlaufsperre: *spin with proportional backoff* (Forts.)

Hinweis zur Laufzeitlänge des kritischen Abschnitts übergeben und berücksichtigen

Überprüfung der Abbruchbedingung der Umlaufsperre sollte zeitnah zum Verlassen des kritischen Abschnitts durch den Vorgängerprozess erfolgen

- anteilige Wartezeit mit der Laufzeitlänge des KA verrechnen

Eintrittsprotokoll

```
void lv_acquire(lock_t *lock, int time) {
    unsigned int self = FAI((int *)&lock->next);

    do cpu_pause(time * (self - lock->this));
    while (self != lock->this);
}
```

Beachte (nochmals): Laufzeitlänge des kritischen Abschnitts

- den Wert in die Wartezeitberechnung einbringen, ist leicht
- ihn für unterbrechungsfreie kritische Abschnitte zu bestimmen, geht
- sind die Abschnitte unterbrechbar, ist das Problem ggf. unlösbar

Aktives Warten (engl. *busy waiting*)

Unzulänglichkeit der Schlossalgorithmen: der aktiv wartende Prozess. . .

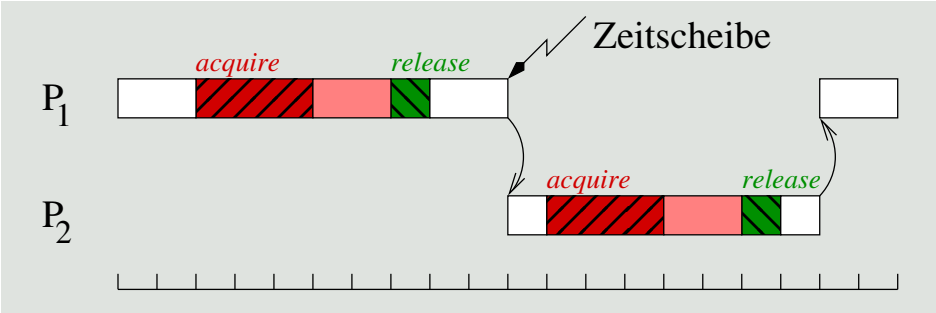
- kann keine Änderung der Bedingung herbeiführen, auf die er wartet
- behindert andere Prozesse, die sinnvolle Arbeit leisten könnten
- schadet damit letztlich auch sich selbst

Je länger der Prozess den Prozessor für sich behält, umso länger muss er darauf warten, dass andere Prozesse die Bedingung erfüllen, auf die er selbst wartet.

- in den meisten Fällen sind Effizienzeinbußen in Kauf zu nehmen
- auch, wenn jeder Prozess seinen eigenen realen Prozessor(kern) hat
  - Wettstreit um Zwischenspeicherzeilen ist das dominierende Problem
  - es sei denn, der Prozessor(kern):
    (a) hat keinen Zwischenspeicher oder
    (b) läuft mit ausgeschaltetem Zwischenspeicher

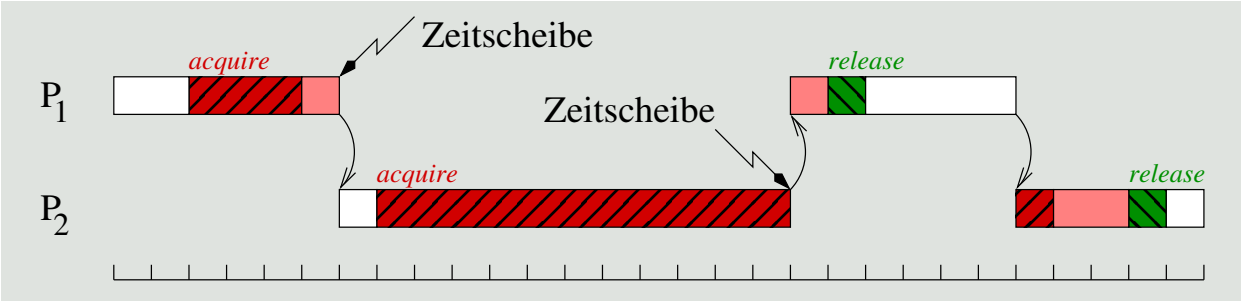
Aktives Warten $\models$ Warten ohne Prozessorabgabe

„Spin locking considered harmful“



	T_s	T_q	T_q/T_s
P_1	12	20	1.67
P_2	8	8	1.0

Bedienzeit T_s, Durchlaufzeit T_q
normalisierte $T_q = T_q/T_s$



	T_s	T_q	T_q/T_s
P_1	12	24	2.0
P_2	17	23	1.35

Bedienzeit T_s, Durchlaufzeit T_q
normalisierte $T_q = T_q/T_s$

Alternative: Prozessorabgabe in der Warteschleife

laufend $\mapsto$ bereit	in Laufbereitschaft bleiben
laufend $\mapsto$ blockiert	schlafend die Schlossfreigabe erwarten

Passives Warten (engl. *lazy waiting*)

Prozessorabgabe, solange der kritische Abschnitt durch einen anderen Prozess(or(kern)) als belegt gilt, kann wie folgt geschehen:

(a) laufbereit bleibend: Effektivität hängt ab von der Umplanungsstrategie

RR
- der Prozess kommt ans Ende der Bereitliste ☺
- seine Wartezeit verlängert sich dadurch ggf. sehr stark ☹

sonst
- seine (stat./dyn.) Priorität bestimmt die Listenposition
- ggf. landet er ganz vorne, so dass er sofort weiterläuft ☹

(b) sich auf das Ereignis der Sperrfreigabe blockieren: *sleeping lock*
- also das Prinzip der **Bedingungssynchronisation** zum Einsatz bringen
  - $acquire(lock) \rightsquigarrow sleep(lock)$, $release(lock) \rightsquigarrow rouse(lock)$
- ggf. pro Schlossvariable eine Warteschlange schlafender Prozesse

Beachte: Dauer des Rücktritts vom Wettstreit

- den *Backoff* allein nur durch die Laufzeit eines kritischen Abschnitts zu bestimmen, ist schon schwer genug
- jetzt müssten noch die Verzögerungen durch Prozesswechsel, Last und Einplanungsstrategie mit eingerechnet werden

Gliederung

Resümee

- Synchronisation in der Maschinenprogrammebene kann auf Konzepte von Betriebssystemen zurückgreifen
  - die den Zeitpunkt von Einplanung oder Einlastung gezielt beeinflussen
  - die Prozesse kontrolliert schlafen legen und wieder aufwecken
- durch eine **Verdrängungssperre** wird die Einplanung bzw. Einlastung von Prozessen erst verzögert wirksam
  - kritische Abschnitte werden verdrängungsfrei durchlaufen, aber
  - unabhängige gleichzeitige Prozesse werden unnötig zurückgehalten
- die **Umlaufsperre** ist *der* Klassiker zur Synchronisation gleichzeitiger Prozesse in parallelen Systemen mit gemeinsamem Speicher
  - sie skalierbar auszulegen, ist schwer und abhängig vom Anwendungsfall
  - sie eignen sich ggf. für kurze unterbrechungsfreie kritische Abschnitte
    - dann jedoch können sie auch gut durch andere Verfahren ersetzt werden
  - ihre Implementierung ist durch und durch prozessorabhängig
    - vor allem von Art, Aufbau und Funktionsweise des Zwischenspeichers
- wenn überhaupt, **Prozessoren** damit synchronisieren, nicht Prozesse

Literaturverzeichnis

[1] Abramson, N. :
The ALOHA System: Another Alternative for Computer Communication.
In: *Proceedings of the Fall Joint Computer Conference (AFIPS '70).*
New York, NY, USA : ACM, 1970, S. 281–285

[2] Mellor-Crummey, J. M. ; Scott, M. L.:
Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.
In: *ACM Transactions on Computer Systems* 9 (1991), Febr., Nr. 1, S. 21–65

[3] Metcalfe, R. M. ; Boogs, D. R.:
Ethernet: Distributed Packet Switching for Local Computer Networks.
In: *Communications of the ACM* 19 (1976), Jul., Nr. 5, S. 395–404

[4] Papamarcos, M. S. ; Patel, J. H.:
A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories.
In: *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA '84), June 5–7, 1984, Ann Arbor, Michigan, USA*, ACM Press, 1984, S. 348–354

[5] Tanenbaum, A. S.:
Multilevel Machines.
In: *Structured Computer Organization.*
Prentice-Hall, Inc., 1979. –
ISBN 0–130–95990–1, Kapitel 7, S. 344–386