

Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf
Basiert auf ALP IV, SS 2003
Klaus-Peter Lühr
Freie Universität Berlin

Überblick

Inhalt

- Prozeßprioritäten
- Auswahlstrategien
- Alterungsmechanismen

4 Ablaufsteuerung

Ablaufsteuerung

- **Ablaufsteuerung** (*scheduling*) =
- Zuteilung von *Betriebsmitteln* (*resources*)
 - wiederverwendbaren oder
 - verbrauchbaren

an Prozesse, z.B.

- Prozessor (wiederverwendbar),
- Speicher (wiederverwendbar),
- Nachrichten (verbrauchbar),
- Ereignisse (verbrauchbar)

4.1 Prozeßprioritäten

- **Def.:**
Priorität eines Prozesses =
einem Prozess (*task, thread, ...*) zugeordnetes
Attribut, üblicherweise eine natürliche Zahl, das zur
Entscheidung über die Zuteilung von Betriebsmitteln
herangezogen wird.
- Priorität wird einem Prozess bei der Erzeugung
(explizit oder implizit) zugeteilt und kann eventuell
programmgesteuert verändert werden.

Beispiel Java

▪ Java:

- `Thread.MIN_PRIORITY` = 1
 - `Thread.NORM_PRIORITY` = 5
 - `Thread.MAX_PRIORITY` = 10
-
- Der *Urprozess (main thread)* hat die Priorität 5.
 - Ein Thread *vererbt* seine Priorität an seine Kinder.
 - *Abfragen* der Priorität mit `int getPriority()`
 - *Ändern* der Priorität mit `void setPriority(int p)`

Beispiel Java

- Die JVM *kann* bei der Zuteilung von Prozessorkapazität an die Threads die Prioritäten berücksichtigen, muß aber nicht.
- Typische, sinnvolle Prioritätenvergabe:
 - 2-3 Rechnen
 - 4-6 Ein/Ausgabe
 - 7-9 schnelle Ein/Ausgabe
(Interaktion, harte Realzeit-Anforderungen)
- Nicht vergessen: im *Mehrprozessbetrieb* erhält die JVM bereits vom Betriebssystem nur ein Teil der Prozessorkapazität!

Prioritätenvergabe vs. Synchronisation

- **Achtung 1:**
Prioritätenvergabe ist kein Ersatz für Synchronisation!
- **Achtung 2:**
Gefahr der **Prioritätsumkehr** (*priority inversion*),
z.B. bei Einprozessorsystem:
 - 3 Prozesse A, B, C mit abfallenden Prioritäten a,b,c
 - C betritt kritischen Abschnitt k, weckt B und wird von B verdrängt, d.h. belegt weiterhin k;
 - B weckt A und wird von A verdrängt,
 - A blockiert beim Eintrittsversuch in k;
 - solange B nicht blockiert, hat A keine Chance!
- **Lösungstechnik:** spezielle Sperroperationen, die für temporäre Prioritätserhöhung sorgen.

4.2 Auswahlstrategien

- **Problem:**
Wenn *mehrere* Prozesse auf Betriebsmittelzuteilung bzw. auf ein bestimmtes Ereignis warten (mit **when**, **wait**, **P**, **join**, ...)

und

beim Eintreten des Ereignisses *nicht alle* aufgeweckt werden können/sollen, **welche** ?

- → **Auswahlstrategie** (*scheduling policy*)

Fairness

- **Def.:**
Eine Auswahlstrategie heißt **fair**, wenn jedem wartenden Prozess garantiert ist, daß ihm nicht permanent und systematisch andere Prozesse vorgezogen werden.
- **Genauer:**
 - **streng fair:**
wenn ein Prozess blockiert, kann eine *obere Schranke* für die Anzahl der Prozesse angegeben werden, die ihm vorgezogen werden;
 - **schwach fair:**
sonst

Unfairness

- Bei unfairer Auswahlstrategie droht den Prozessen **unbestimmte Verzögerung**
(*indefinite delay, starvation*)
- **Achtung:**
In Java gibt es keinerlei Fairness-Garantien !

Anpassung der Auswahlstrategie

- *Konsequenz:*
im Eigenbau
 1. *entweder* wiederverwendbare Varianten der Synchronisationskonstrukte bereitstellen → 4.2.1.1
 2. *oder* ad-hoc-Strategie programmieren → 4.2.1.2

4.2.1.1 Prioritätsgesteuerte Semaphore

```
interface Semaphore { // scheduling unspecified  
void P();  
void V();  
}
```

```
class Sema implements Semaphore { // FCFS  
...  
}
```

```
class PrioSema implements Semaphore {  
public PrioSema(int init) { count = init; }  
private int count;  
private final Sema mutex = new Sema(1);  
private final PrioSet prios = new PrioSet();
```

Prioritätsgesteuerte Semaphore

```
class PrioSet { // set of pairs (int, Sema)
.....
public void add(int prio, Sema sema) {...}
    // adds pair (prio,sema)

public Sema rem() {...}
    // delivers semaphore with highest priority
    // and removes entry
}
```

Prioritätsgesteuerte Semaphore

```
public void P() { // this is the modified P
    mutex.P();
    count--;
    if(count>=0) mutex.V();
    else {
        Sema ready = new Sema(0);
        prios.add(Thread.currentThread().getPriority(), ready);
        mutex.V();
        ready.P();
    }
}
```

Prioritätsgesteuerte Semaphore

```
public void V() { // this is the modified V
    mutex.P();
    count++;
    if(count<=0) {
        Sema ready = prios.rem();
        ready.V(); }
    mutex.V();
}
```

▪ *Beachte:*

1. Die Semaphore **ready** sind private Semaphore, d.h. bei ihnen blockiert jeweils höchstens ein Thread.
 2. Die kritischen Abschnitte (**mutex**) sind kurz genug, dass sie keinen Engpass darstellen.
- → Daher ist es *irrelevant*, welche Auswahlstrategie von **Sema** praktiziert wird.

Wahl einer Sperroperation:

- langfristiges Sperren realer Betriebsmittel:
Auswahlstrategie wählen, ggfls. selbst implementieren
- kurzfristiges Sperren kritischer Abschnitte:
Auswahlstrategie irrelevant
 - **sofern kein Engpass** bei der Benutzung der Ressource bzw. des kritischen Abschnitts vorliegt !
 - **sonst: Entwurf revidieren** –
lange Prozesswarteschlangen sind in jedem Fall unsinnig –
wie auch immer die Auswahl praktiziert wird.

4.2.1.2 Anforderungsbezogene Auswahl

- bei jeder Art von Anforderung *mehrerer Ressourcen* z.B.
 - verallgemeinerte P/V-Operationen (3.3.1)
 - `request(n)` , `release(n)`
 - und ähnliche
- Häufig ist **Effizienz** wichtiger als Fairness, z.B.
 - gute Ressourcen-Ausnutzung
 - kurze Reaktionszeiten

Anforderungsbezogene Auswahl

- Beispiele für effizienzorientierte Auswahlstrategien bei `request(n)`/`release(n)`:
- *SRF (smallest request first)*:
die *kleinsten* Anforderungen sollen vorrangig bedient werden
- *LRF (largest request first)*:
die *größten* Anforderungen sollen vorrangig bedient werden
- (jedenfalls nicht die, welche am längsten warten; weitere Alternativen sind möglich.)

Anforderungsbezogene Auswahl

- LRF mit Verwendung von **PrioSet** (4.2.1.1) mit Operationen
- **public int top(int limit)**
 - liefert den höchsten prio-Wert im **PrioSet** , der **limit** nicht übersteigt (0, falls kein solcher vorhanden)
- **public Sema rem(int limit)** (modifiziert)
 - liefert das zugehörige Semaphor und löscht den Eintrag

Anforderungsbezogene Auswahl

```
class Resources {// LRF – largest request first
public Resources(int init) { avail = init; }
private int avail;
private final PrioSet claims = new PrioSet();

public void request(int claim) {
    Sema ready;
    synchronized(this) {
        if(claim<=avail) {
            avail -= claim;
            return; }
        else {
            ready = new Sema(0);
            claims.add(claim,ready);
        }
    }
    ready.P();
}
```

Anforderungsbezogene Auswahl

```
public synchronized void release(int free) {  
    avail += free;  
    do {  
        int claim = claims.top(avail);  
        if(claim == 0) return;  
        else {  
            avail -= claim;  
            claims.rem(avail).V();  
        }  
    } while(true);  
}
```

▪ *Beachte:*

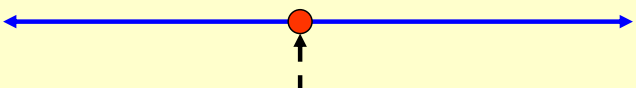
- Das Blockieren in **request** muß *außerhalb* des kritischen Abschnitts erfolgen
- In **release** werden i.a. gezielt *mehrere* Threads aufgeweckt
- Durch Modifikation der Klasse **PrioSet** können *verschiedene* Strategien realisiert werden

4.2.2 Fairness durch Alterungsmechanismen

▪ Gegensätzliche Ziele bei Auswahlstrategien:

- | | | |
|---------------------|---------------|------------------|
| ▪ <i>Ziel:</i> | Effizienz | Gleichbehandlung |
| ▪ <i>Fairness:</i> | unfair | streng fair |
| ▪ <i>Beispiel:</i> | SRF | FIFO |
| ▪ <i>Kriterium:</i> | Anforderungen | Wartezeit |

▪ **Spektrum** von Kriterien:

- Anforderungen ←  → Wartezeit

anforderungsorientiert + **Alterungsmechanismus**
(*ageing*)

Alterungsmechanismen

▪ Beispiel:

Alterungsmechanismus für SRF:

- Auswahl nach *Rang*ordnung 1,2,3,...
(hoher Rang = kleine Rangzahl)
- *Rang* = Anforderung + *Malus*
- *Malus* = Zeitpunkt des **request**, gemessen in Anzahl der begonnenen **requests**
- Einem **release** wird genau dann stattgegeben, wenn die Anforderung erfüllt werden kann **und** den höchsten Rang hat. (Bei Gleichrangigkeit zweier erfüllbarer Anforderungen...)

Beispiel-Szenario

Thread	request	release	counter	added	grantee	available
			0			5
p	1		1		p	4
q	4		2		q	0
r	4		3	(7, 4, R)		0
s	1		4	(5, 1, S)		0
t	1		5	(6, 1, T)		0
p		1			s	0
q		3			t	2
u	2		6	(8, 2, U) !		2
q		1				3
s		1			r	0

Alterungsmechanismen

- **Buchführung** über Anforderungen und Ränge mittels

```
class Ranking { // set of triples (int rank, int claim, EVENT e)
.....
public void add(int rank, int claim, EVENT e){...}
    // adds triple (rank,claim,e)

public EVENT rem() {...}
    // delivers event with highest rank
    // and removes entry (if any, else null)

public int firstRank() {...}
    // delivers highest rank (if any, else MAX)

public int firstClaim() {...}
    // delivers corresponding claim (or MAX)
}
```

Alterungsmechanismen

```
MONITOR Resources { // SRF, with ageing
                    // Monitor and Events!
public Resources(int n) {available = n;}
protected int available;
protected int counter;
protected final Ranking ranking = new Ranking();

public void release(int n) {
    available += n;
    tryWakeup();
}
protected void tryWakeup() {
    if(ranking.firstClaim() <= available){
        available -= ranking.firstClaim();
        ranking.rem().SIGNAL();
    }
} // any version of SIGNAL does the job
```

Alterungsmechanismen

```
public void request(int claim) {  
    counter++;  
    int rank = claim + counter;  
    if(rank <= ranking.firstRank() && claim <= available) {  
        available -= claim;  
        return; }  
    else {  
        EVENT ready = new EVENT();  
        ranking.add(rank,claim,ready);  
        ready.WAIT();  
        tryWakeup();  
    }  
}  
  
} // end monitor
```

Alterungsmechanismen

- **Flexible** Positionierung im **Spektrum** der Strategien:
 - anstelle von
 $\text{rank} = \text{claim} + \text{counter}$
 - Verwendung von
 $\text{rank} = \text{claim} + \mathbf{k} * \text{counter}$
- $k \in [0, m]$ ($m =$ maximal mögliche Anforderung)
ermöglicht *Tuning*
 - $k = 0$: SRF
 - $k = m$: FIFO
 - $\text{rank}_i < \text{rank}_{i+1}$ für zwei aufeinanderfolgende **requests**

Zusammenfassung

Zusammenfassung

- Prozeßprioritäten
- Auswahlstrategien
 - Fairness
- Alterungsmechanismen