

# Betriebssysteme

## Tutorium 6

Philipp Kirchhofer

`philipp.kirchhofer@student.kit.edu`

`http://www.stud.uni-karlsruhe.de/~uxbtt/`

**Lehrstuhl Systemarchitektur  
Universität Karlsruhe (TH)**

2. Dezember 2009

# Was machen wir heute?

## 1 Organisatorisches

## 2 Tutorien Übungsblatt

- Synchronization Basics
- Protecting Shared Data Structures
- Concurrent Modifications of SHM
- Towards High-Perf. Spinlocks
- Implementing General Semaphores

## Programmieraufgaben Vorstellung

**Montag, den 7. Dezember, 14:00 Uhr bis 15:30 Uhr**

**Poolraum G im Rechenzentrum (Gebäude 20.21)**

In circa 10 Minuten Lösung der Math & Restaurant Aufgabe sowie das Design-Dokument vorstellen.

**Wichtig:** Beide Gruppenteilnehmer müssen vorstellen!

## Frage 6.1.a

Erkläre die Begriffe „Kritischer Abschnitt“ (Critical Section), „Entry Section“, „Exit Section“ und „Remainder Section“.

## Antwort

### **Kritischer Abschnitt**

Ein kritischer Abschnitt ist ein Teil eines Programms, der auf gemeinsame Daten zugreift oder diese verändert. Es dürfen gleichzeitig keine weiteren Teile des Programms, die ebenfalls auf die gleichen gemeinsamen Daten zugreifen, ausgeführt werden (Exklusivität).

### **Entry Section**

Exklusiver Zugang wird angefordert

### **Exit Section**

Freigabe des exklusiven Zugangs

### **Remainder Section**

Restlicher Programmcode

## Frage 6.1.b

Was sind die Bedingungen für eine korrekte Synchronisation?

## Antwort

- Exklusivität (Exclusiveness)  
Nur eine Aktivität (Thread/Prozess) darf kritischen Abschnitt zur selben Zeit ausführen
- Fortschritt (Progress)  
Der Programmcode in der Remainder Section darf den Eintritt in den kritischen Abschnitt nicht verhindern
- Begrenztes Warten (Bounded Waiting)  
Jede auf den Eintritt in den kritischen Abschnitt wartende Aktivität kommt irgendwann dran (kein Verhungern)
- [Portabilität (Portability)]  
Lösung darf keine Annahmen über Prozessorgeschwindigkeit, Schedulingverfahren, usw. machen

## Frage 6.1.c

Bei Systemen mit einem Prozessor können Datenstrukturen des Betriebssystemkerns durch Ausschalten (Maskieren) der Interrupts gegen gleichzeitigen Zugriff geschützt werden. Weshalb funktioniert dieses Vorgehen nicht bei Systemen mit mehreren Prozessoren?

## Antwort

Das Maskieren von Interrupts beeinflusst nur einen Prozessor. Für ein Maskieren von Interrupts auf allen Prozessoren muss das Betriebssystem eine Nachricht an alle Prozessoren versenden.

⇒ Hoher Overhead

Dieses Vorgehen verhindert allerdings immer noch nicht den mehrfachen gleichzeitigen Eintritt in den Kern: Ein Programm kann z.B. weiterhin mit einer „syscall“ Instruktion in den Kern gelangen.

## Frage 6.1.d

Wie kann gegenseitiger Ausschluss bei Multiprozessorsystemen gewährleistet werden?

## Antwort

Bei Multiprozessorsystemen müssen Spinlocks eingesetzt werden.

### **Big Lock**

Ein Spinlock für den ganzen Kern

- ⊕ Einfach zu implementieren
- ⊖ Skalierbarkeit erheblich beschränkt (kein Parallelismus im Kern)

### **Locking auf Datenstrukturebene**

Spinlocks für jede Datenstruktur

- ⊕ Skalierbarkeit wenig eingeschränkt (Parallelismus im Kern möglich)
- ⊖ Fehlerträchtige Umsetzung

$n > 1$  Threads greifen auf eine doppelt verkettete Liste und einen binären Baum zu. Jeder Knoten enthält Daten über einen Kunden wie z.B. Vor- und Nachname, Telefonnummer usw. Die Knoten in den Datenstrukturen werden so hinzugefügt, dass sie nach den Nachnamen sortiert sind.

## Frage 6.2.a

Welche Race Conditions können auftreten und wie können diese verhindert werden?

## Antwort

Jede Datenstruktur enthält mehrere Pointer. Bei jedem Hinzufügen oder Löschen von Daten müssen mehrere Pointer geändert werden.

⇒ Nicht atomar

⇒ Race Conditions bei gleichzeitigem Ändern von mehreren Knoten

Race Conditions können durch den Einsatz von Locking verhindert werden.

## Frage 6.2.b

Muss ein Knoten mit Kundendaten gültige Daten enthalten bevor er in die Datenstruktur eingefügt wird?

## Antwort

### **Kein Lock für Lesezugriff auf Knoten**

Kundendaten müssen gültig sein, da sofort nach dem Einfügen in die Datenstruktur Leseprozesse auf den Knoten zugreifen können.

### **Lock für Lesezugriff auf Knoten**

Kundendaten müssen nicht gültig sein solange der Erstellprozess den Leselock für den neu eingefügten Knoten hält.

## Programm

```
const n = 50;
var sum : integer;

procedure total;
var count : integer;
begin
  for count := 1 to n do sum := sum + 1
end;

begin (* main program *)
  sum := 0;
  parbegin
    (* parallel execution block *)
    total;
  parend;
  write(sum)
end.
```

## Frage 6.3.a

Die Anzahl der startenden Threads sei 2. Welche minimalen und maximalen Werte kann die Variable „sum“ nach Ablauf des Programms annehmen?

## Antwort

$$2 \leq \text{sum} \leq 100$$

## Frage 6.3.b

Die Anzahl der startenden Threads sei  $t$ . Welche minimalen und maximalen Werte kann die Variable „sum“ nach Ablauf des Programms annehmen?

## Antwort

$$2 \leq \text{sum} \leq 50 * t$$

## Frage 6.3.c

Es werden nun  $t$  Userlevel-Threads gestartet. Wie sieht die Belegung der „sum“ Variable nach Abschluss des Programms aus?

## Antwort

Alle Threads werden seriell abgearbeitet, d.h. der erste Thread wird gestartet und läuft bis zum Ende durch. Anschließend wird der nächste Thread gestartet usw.

$$\text{sum} = 50 * t$$

## Geändertes Programm

```
procedure total;  
var count : integer;  
begin  
  for count := 1 to n do begin  
    sum := sum + 1;  
    yield  
  end  
end;
```

## Frage 6.3.d

Welche Ausgabe gibt das Programm bei den Many-to-One oder One-to-One Threadmodellen aus?

## Antwort

### Many-to-One

$$\text{sum} = 50 * t$$

### One-to-One

$$2 \leq \text{sum} \leq 50 * t$$

SMP System mit 4 Prozessoren ( $P_1, \dots, P_4$ ) mit gemeinsam genutzten Hauptspeicher und getrennten L1- und L2-Caches für jeden Prozessor. Drei kooperierende Threads  $T_1, \dots, T_3$ , deren Arbeitsdaten in den jeweiligen L2-Cache passen, laufen parallel auf den Prozessoren  $P_1, \dots, P_3$ . Die Threads synchronisieren sich mit Hilfe von Spinlocks. Unabhängig von diesen drei Threads führt Prozessor  $P_4$  ein Schachprogramm aus, dessen Arbeitsdaten nicht in den L2-Cache passen.

## Spinlock

```
do
  reg := myThreadId;
  (* atomically exchange shared variable 'spinlock' and reg *)
  swap(&spinlock, reg);
until (reg = 0);
(* critical section *)
spinlock := 0;
```

## Frage 6.4.a

Weshalb läuft das Schachprogramm erheblich langsamer wenn die Threads  $T_1$  bis  $T_3$  gleichzeitig ausgeführt werden?

## Antwort

Nur ein Thread kann gleichzeitig in einem kritischen Abschnitt sein und den Spinlock dazu halten. Die anderen beiden Threads konkurrieren währenddessen um den Zugang zum kritischen Abschnitt. Da beide konkurrierenden Threads lesend und schreibend auf die Spinlock Variable zugreifen werden die (geänderten) Cachezeilen zwischen den Caches der beteiligten Prozessoren hin- und hergeschoben. Dabei entsteht viel Verkehr auf dem Bus, der die Prozessoren gegenseitig und mit dem Hauptspeicher verbindet. Durch diesen Verkehr werden speicherintensive Programme wie das Schachprogramm verlangsamt.

## Frage 6.4.b

Wie kann man das Programm verändern um die Geschwindigkeit des Schachprogramms nicht zu stark zu verlangsamen?

## Antwort

Minimierung der Schreibzugriffe

**Wichtig: Auf atomare Swapoperation kann nicht verzichtet werden**

## Optimierter Spinlock

```
do
  reg := myThreadId;
  (* spinlock can be read from own cache until unlock *)
  while (spinlock != 0) do (* wait *) od
  (* now we got spinlock = 0, try to acquire the lock *)
  swap(&spinlock, reg);
until (reg = 0); (* otherwise another thread was faster... *)
(* critical section *)
spinlock := 0;
```

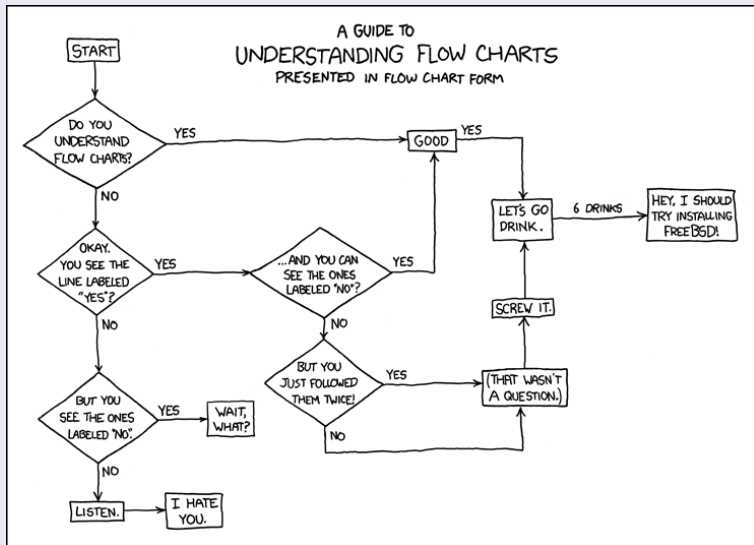
## Frage 6.5

Ist diese Lösung korrekt?

```
procedure c_wait(var s:semaphore)
begin
    b_wait(mutex);
    s := s - 1;
    if (s < 0) then begin
        b_signal(mutex);
        b_wait(delay)
    end else
        b_signal(mutex)
end;

procedure c_signal(var s:semaphore)
begin
    b_wait(mutex);
    s := s + 1;
    if (s <= 0) then b_signal(delay);
    b_signal(mutex)
end;
```

Fragen & Kommentare?



xkcd: Flow Charts