Betriebssysteme und Netzwerke Vorlesung 7

Artur Andrzejak

Umfragen: https://pingo.coactum.de/301541

Ein Multi-Threaded Programm

Zählen von Bitcoins

- Sie haben sehr viele Bitcoins und wollen diese schnell durchzählen
- CPU mit mehreren Cores => Multi-Threaded-Programm!
- Grobe Programmstruktur:

- Portfolio in k Teile P_1,…, P_k splitten
- Einen globalen Zähler counter = 0 setzen
- Thread i bekommt P_i und zählt counter hoch
- Ausgabe: Wert von counter

Zählen von Bitcoins – ein gutes Programm?

Thread 1

<u>while</u> bitcoin in P_1:
counter++

…

Thread k

<u>while</u> bitcoin in P_k:
counter++

- Wie wird eigentlich counter++ umgesetzt?
- Assembler-Befehle (schematisch):

1. Kopiere Inhalt des Speichers mit Adresse "counter" in CPU-Register EAX (oder ein anderes Register)
2. Erhöhe den Inhalt von EAX um 1
3. Kopiere Inhalt des Registers EAX in den Speicher mit Adresse "counter"

Assembler-Befehle für counter++

RAM

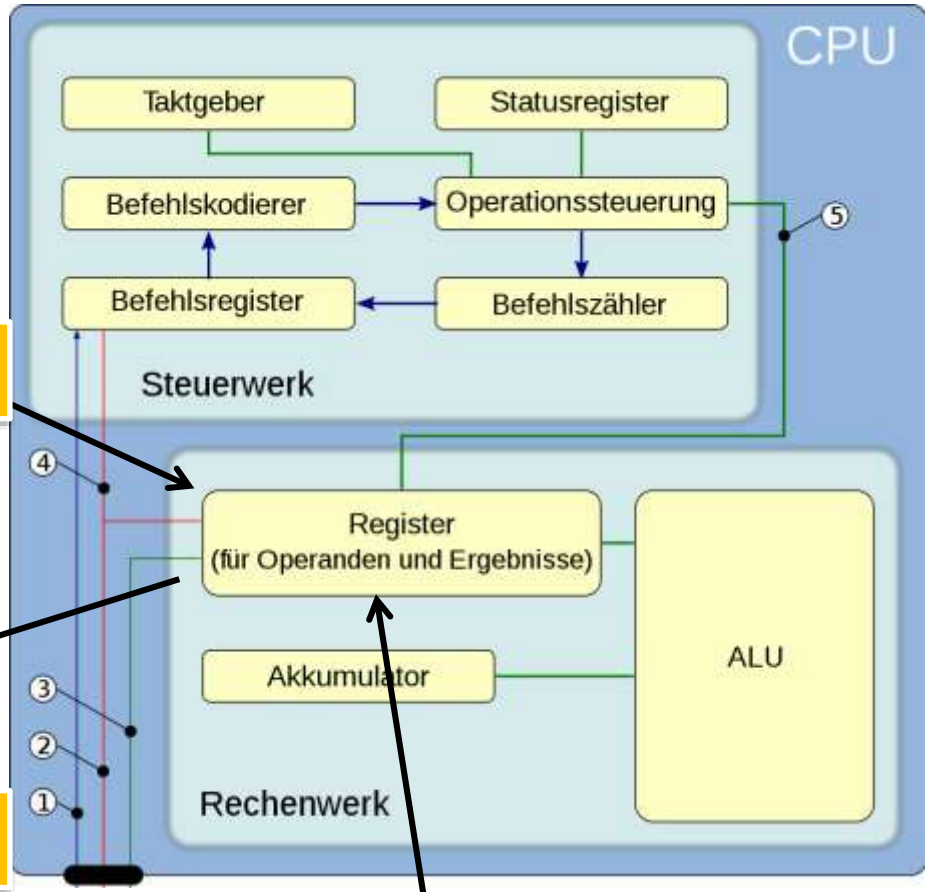
Adresse	Inhalt
…	…
"counter"	5
…	…

1. MOV EAX, M[counter]

Adresse	Inhalt
…	…
"counter"	6

3. MOV M[counter], EAX

CPU



2. ADD EAX, 1

Unterbrechungen

- Interrupt kann zwischen diesen Befehlen stattfinden
  - Bei CISC-Prozessoren, sogar während der Ausführung
- Interrupt kann zu Thread-Wechsel führen

Ausführung 1 (counter = 5)	
Thread A	**Thread B**
MOV EAX, M[counter]	
ADD EAX, 1	
MOV M[counter], EAX	
	MOV EAX, M[counter]
	ADD EAX, 1
	MOV M[counter], EAX

Wert nach der Ausführung?

Unterbrechungen /2

- Könnte etwas schiefgehen?
- Eine andere Ausführung?

Ausführung 2 (counter = 5)	
Thread A	**Thread B**
MOV EAX, M[counter]	
	MOV EAX, M[counter]
	ADD EAX, 1
	MOV M[counter], EAX
ADD EAX, 1	
MOV M[counter], EAX	

Wert nach der Ausführung?

Problem: Man kann nicht kontrollieren, wann die Unterbrechungen stattfinden => **Nicht-Determinismus**

Prozesssynchronisation

Aspekte der Prozesssynchronisation

1. Wie können Daten von einem Prozess zum anderen übertragen werden? (=> IPC)
2. Beachtung von Abhängigkeiten: Falls Prozess A Daten an Prozess B liefert, muss B warten, bis A fertig ist, bevor B etwas tut (später)
3. Prozesse / Threads sollen sich nicht „stören“, d.h. die Ausführung eines Prozesses soll nur dann von anderen abhängen, wenn das gewünscht ist

Race Condition - Wettlaufsituation

- Typische Situation: Zwei oder mehr Prozesse oder Threads lesen/schreiben auf die gleiche Variable
  - Z.B. Inkrementieren einer globalen Variable

Prozess A	Prozess B
1. Lese Wert w = M[ptr] aus Speicher in den CPU-Register EAX 2. Erhöhe w um 1	
	Lese w = M[ptr] in den Register EAX, erhöhe EAX um 1, schreibe EAX nach M[ptr]
3. Schreibe Inhalt von EAX nach M[ptr]	

Wechsel

Wechsel

=> Führt zum nicht-deterministischen Verhalten

Race Condition /2

- Wie sollte man eine **Race Condition** definieren?
- Definition: ein Softwareproblem / Situation, bei dem das Ergebnis einer Programmausführung (i.A. in einer nicht beabsichtigter Weise) von der Reihenfolge oder dem Timing von Ereignissen abhängt
- Video Race Conditions von Andreas Wilkens, [06b]
  - https://www.youtube.com/watch?v=dlOg4Dz-bgM
  - Ab 0:00 bis ca. 4:00 (min:sec)

Beispiel Producer – Consumer Problem

- Producer schreibt in einen Puffer, Consumer liest heraus (und löscht den Eintrag)

Producer	**Consumer**

// Start der Sender-Subroutine
// Eingabe ist in **nextProduced**
 	
while (**count** == BUFFER_SIZE)
  ; // warte, Puffer voll
 	
buffer [in] = **nextProduced**;
in = (in + 1) % BUFFER_SIZE;
count++;
	// Start der Empfänger-Subroutine
	
	while (**count** == 0)
	  ; // warte, nichts im Puffer
	
	nextConsumed = buffer [out];
	out = (out + 1) % BUFFER_SIZE;
	count--;
	
	// Ausgabe ist in **nextConsumed**

Race Condition bei P/C?

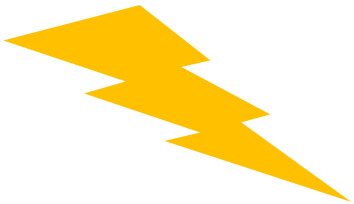
Implementierung von ++ und von --:

- count++
  - R1 = count
  - R1 = R1 + 1
  - count = R1
- count--
  - R2 = count
  - R2 = R2 - 1
  - count = R2
- Was kann passieren?

- Ausführung, anfangs count = 5

producer: R1 = count {R1 = 5}
producer: R1 = R1 + 1 {R1 = 6}
consumer: R2 = count {R2 = 5}
consumer: R2 = R2 - 1 {R2 = 4}
producer: count = R1 {count = 6 }
consumer: count = R2 {count = 4}

Lost-Update Problem



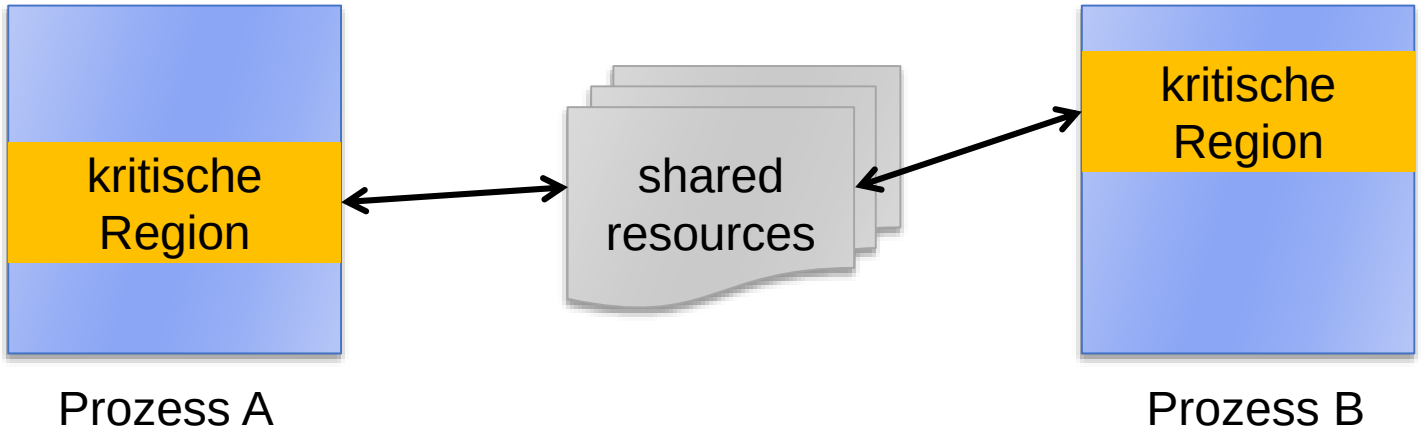
Wiederkehrendes Schema:
Unterbrechung zwischen dem Lesen und dem Schreiben

Wechselseitiger Ausschluss /1

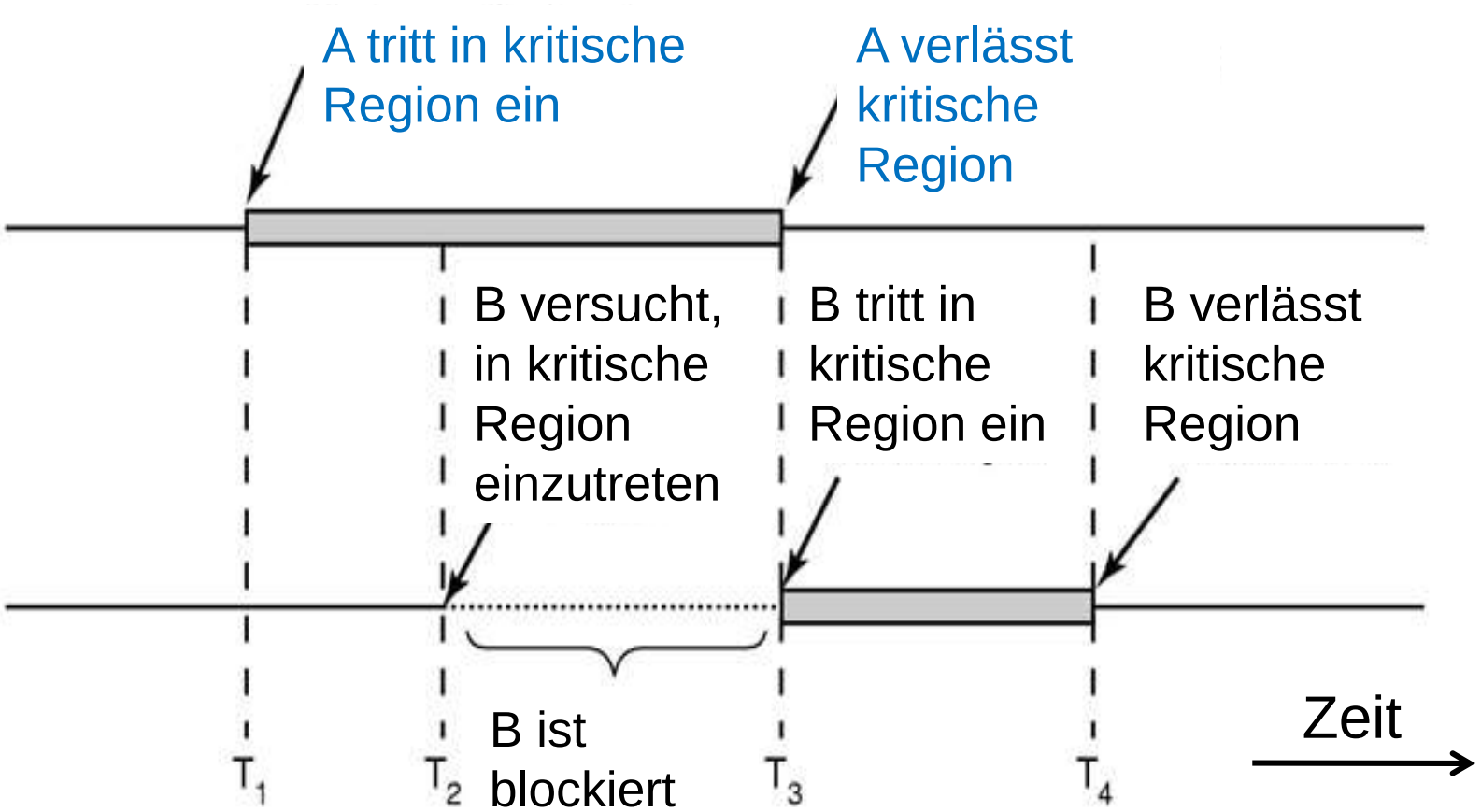
- Generelle Lösung: Man muss sicherstellen, dass nur einer der Prozesse auf einmal auf (gewisse) gemeinsam genutzte Ressourcen / Daten (**shared resources**) zugreifen kann
- Dieses Lösungsansatz wird **wechselseitiger Ausschluss** (**mutual exclusion**) genannt
- Im Folgendem auch abgekürzt **WA**
- Im Englischen auch oft abkürzend „**Mutex**“

Wechselseitiger Ausschluss /2

- Andere Sichtweise: Man muss die Prozesse davon abhalten, zugleich Programmteile auszuführen, die die gemeinsam genutzten Ressourcen manipulieren
  - Man nennt diese Programmteile **kritische Regionen** (**critical regions**) oder **kritische Abschnitte** (**critical sections**)



Prozess-Zeit-Diagramm



Wechselseitiger Ausschluss - Bedingungen

- Eine korrekte Implementierung des WA erfüllt folgende Bedingungen:

1. Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Regionen sein (mutual exclusion)
2. Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der CPUs gemacht werden
3. Kein Prozess, der außerhalb seiner kritischen Region läuft, darf andere Prozesse blockieren
4. Kein Prozess sollte ewig warten müssen, in seine kritische Region einzutreten

Wechselseitiger Ausschluss: Lösungen mit **aktivem Warten**

Abstrakter Ablauf eines Programms

- Wir können schematisch annehmen, dass ein Prozess/Thread abwechselnd ausführt:
  - In seinen nicht-kritischen Regionen
  - In der kritischen Region
- In Wirklichkeit ist das etwas komplizierter
  - z.B. mehrere kritische Regionen

```
while (TRUE) {
    Führe Code in nicht-kritischer Region aus
    <Erlange Zutritt zur kritischen Region>
    Führe Code in der kritischen Region aus
    <Lasse andere Prozesse/Threads in die k. R. eintreten>
}
```

Lösungen

- Ausschalten von Interrupts
  - Hilft nicht bei Multi-Core / Multi-Prozessor Systemen
- Strikter Wechsel:

Prozess A	**Prozess B**

while(TRUE) {
  while (turn != **0**)
  ; // Schleife
  critical_region();
  turn = **1**;
  noncritical_region();
}
	while(TRUE) {
	  while (turn != **1**)
	  ; // Schleife
	  critical_region();
	  turn = **0**;
	  noncritical_region();
	}

Gibt es hier evtl. ein Problem?

Umfrage (https://pingo.coactum.de/301541)

Prozess A	Prozess B

while (TRUE) {
  while (turn != **0**); // Schleife
  critical_region();
  turn = **1**;
  noncritical_region();
}
	while (TRUE) {
	  while (turn != **1**); // Schleife
	  critical_region();
	  turn = **0**;
	  noncritical_region();
	}

Beim „Strikter Wechsel“ gibt es folgende Probleme:

- A. Die Prozesse nutzten die CPU, auch wenn sie warten
- B. Es ist unmöglich, diese Lösung auf mehr als 2 Prozesse zu generalisieren
- C. Kein Prozess darf 2x hintereinander die kritische Region betreten
- D. Falls der Aufruf noncritical_region(); lange dauert, wird der jeweils andere Prozess zu lange warten müssen

Lösung von Peterson

- Gary L. Peterson fand 1981 eine einfache Lösung
- Annahmen
  - Zwei Prozesse
  - CPU-Befehle LOAD (lese vom Speicher) und STORE (schreibe in den Speicher) sind nicht unterbrechbar
- Gemeinsame Variablen
  - boolean **interested[2]**: interested[**i**] == true sagt, dass der Prozess/Thread **i** in die kritische Region eintreten möchte
  - int **turnWait**: gibt an, welcher Prozess/Thread als nächster (ggf.) warten muss

Lösung von Peterson - Code

```
#define FALSE  0
#define TRUE   1
#define N      2
int turnWait;
int interested[N];
```

Was passiert, wenn beide Prozesse fast gleichzeitig in die kritische Region kommen?

- Beide speichern ihre Nummer in **turnWait**
- Z.B. Prozess 1 speichert zuletzt:
  - Prozess 0 wartet nicht
  - Prozess 1 wartet in der while-Schleife

```
void enter_region (int process) {
    int other = 1 – process;
    interested [process] = TRUE;
    turnWait = process;
    while (turnWait == process && interested [other] == TRUE)
        ;
}
```

Wenn „ich“ zuletzt (später) **turnWait** auf „mich“ gesetzt habe, muss ich warten!

```
void leave_region (int process) {
    interested [process] = FALSE;
}
```

Sperrvariablen - Locks

- Idee: Wir nutzen einen „Token“, dessen Besitz anzeigt, dass ein Thread in die kritische Region eintreten darf
- Man nennt solche Tokens **Sperren** bzw. **Locks**
  - Def.: Variablen, die anzeigen, dass ein Prozess in der kritischen Region ist, und kein anderer eintreten darf

```
while (TRUE) {
    erlange die Sperre – enter_region
    führe Code in der kritischen Region aus
    setze die Sperre frei – leave_region
    restlicher Code
}
```

Achtung: die While-Schleife ist eine abstrakte Darstellung, und soll nur ausdrücken, dass die Abarbeitung von kritischen und nicht-kritischen Regionen sich abwechselt (so sieht aber ein Programm ggf. nicht aus)

Probleme der Implementierung

- Der Aufruf **enter_region** ist blockierend - keine Rückkehr, bis die Sperre erlangt ist
- Der Aufruf **leave_region** ist nicht blockierend

- Die Bedingung, dass die Sequenz „LOAD und danach STORE“ atomar ausgeführt wird, ist auf moderner HW nicht garantiert
- Macht die Implementierung kompliziert

Zusammenfassung

- Ein Multi-Threaded-Programm (Bitcoins zählen)
- Synchronisation der Prozesse / Threads
  - Race Conditions und Wechselseitiger Ausschluss
  - Kritische Regionen
- Wechselseitiger Ausschluss
  - Bedingungen
  - Naive Lösung; Lösung von Peterson; Sperrvariablen - Locks
- Quellen
  - Synchronisation:
    - Silberschatz et al., Kapitel 6,
    - Tanenbaum et al., Kapitel 2 und 7 (Linux – case study)

Zusätzliche Folien