Vorlesung Betriebssysteme I

Thema 9: Synchronisation

Robert Baumgartl

31. Januar 2022

C-Code-Fragment Bankautomat

```
/* gemeinsam genutzte Variable */
int guthaben = 3000;     /* Dublonen */

int abheben (int cash)
{
  int verbleiben;
  verbleiben = guthaben - cash;
  if (verbleiben >= 0) {
    guthaben = verbleiben;
    return cash;
  }
  else {
    return 0;
  }
}
```

Unterbrechung

Abbildung: Routine für gemeinsam genutztes Bankkonto

Möglicher (typischer) Ablauf

2 Prozesse greifen unabhängig voneinander auf gemeinsam genutzte Variable (`guthaben`) zu. Ein möglicher Ablauf wäre:

- Prozess A ruft `abheben(500);`
- A wird am Pfeil unterbrochen (z. B. weil seine Zeitscheibe abgelaufen ist), `verbleiben == 2500`
- nun Prozess B aktiviert, dieser ruft `abheben(2300);` (z. B. verschwenderische Ehefrau)
- B erhält 2300 Dublonen ausbezahlt (`guthaben` jetzt 700!)
- A fortgesetzt: `verbleiben == 2500` $\rightarrow$ 500 Dublonen ausbezahlt und `guthaben = 2500`
- Wundersame Geldvermehrung!?

Begriff der *Race Condition*

- Leider nicht, sondern nur eine sog. *Race Condition* oder „Wettlaufbedingung“!
- Resultatwert der gemeinsam genutzten Variable hängt vom Ablauf der Zugriffsoperationen ab:
  - erst A komplett, dann B komplett: `guthaben == 200`
  - erst B komplett, dann A komplett: `guthaben == 200`
  - erst A, Unterbrechung am Pfeil, B komplett, dann Rest A: `guthaben == 2500`
  - erst B, Unterbrechung am Pfeil, dann A komplett, dann Rest B: `guthaben == ?`
  - ausbezahlt werden stets 2300 Dublonen für B und 500 Dublonen für A

Denkaufgabe: Maximieren Sie den Ertrag für das Paar! Wie sieht es unter Hinzunahme weiterer für das Konto Verfügungsberechtigter (z. B. gieriger Kinder) aus?

Begriff des *kritischen Abschnittes*

- Die Aktualisierung von `guthaben` durch den zweiten Prozess geht verloren → “Lost Update Problem”
- sog. *zeitabhängiger* Fehler
- Zugriffsoperationen zur gemeinsam genutzten Variable bilden einen so genannten *kritischen Abschnitt* (critical section)
- Die Variable darf nicht gelesen und später basierend auf dem gelesenen Wert modifiziert werden → Zugriffe müssen *atomar* erfolgen!
- Grau hinterlegter Bereich im Codebeispiel stellt einen kritischen Abschnitt bezüglich `guthaben` dar
- kritischer Abschnitt *immer* in Verbindung mit einer (gemeinsam genutzten) Ressource

→ Kritische Abschnitte müssen (sorgfältig!) gemanagt werden.

Steuerung von kritischen Abschnitten

Folgende Forderungen muss eine korrekte Steuerung garantieren:

1. Durchsetzung des *wechselseitigen Ausschlusses* (mutual exclusion) : zu jedem Zeitpunkt darf sich maximal ein Prozess in einem kritischen Abschnitt befinden.
2. Jeder Eintrittswunsch sollte in endlicher Zeit befriedigt werden (kein *Verhungern*, kein *Deadlock*).
3. Prozesse außerhalb eines kritischen Abschnittes sollten Prozesse innerhalb nicht beeinflussen.
4. Die Lösung darf keine Annahmen über Abarbeitungsgeschwindigkeit, Anzahl Prozesse, externe Faktoren usw. treffen.

Steuerung durch klammernde Funktionen

Prinzip: zwei Funktionen `enter_cs` und `leave_cs` klammern den kritischen Abschnitt und steuern ihn durch ihre Funktionalität:

```
enter_cs(guthaben);
verbleiben = guthaben - cash;
if (verbleiben >= 0) {
  guthaben = verbleiben;
leave_cs(guthaben);
```

Abbildung: Zwei Funktionen klammern den kritischen Abschnitt

Jeder, der auf eine gemeinsam genutzte Variable zugreift, muss diese Funktionen vor bzw. nach dem Zugriff einsetzen.

Funktionen zur Steuerung kritischer Abschnitte

Was müssen `enter_cs()` und `leave_cs()` tun?

`enter_cs()`:

- Prüfen, ob kritischer Abschnitt gesperrt ist (ein anderer Prozess ist offensichtlich drin)
- wenn ja: rufenden Prozess blockieren
- wenn nein: kritischen Abschnitt als *gesperrt* markieren und zurückkehren

`leave_cs()`:

- kritischen Abschnitt als *frei* markieren
- ggf. einen Eintritt wünschenden Prozess benachrichtigen
- zurückkehren

Wichtig: Da es in einem Programm mehrere kritische Abschnitte geben kann, müssen diese unterschieden werden (z. B. über einen Parameter)

Realisierungsvarianten

1. busy-waiting vs. blockierende Verfahren
   - Prozesse warten aktiv in einer Schleife (Prozessorzeit wird verschwendet) oder
   - Prozesse gehen in Zustand *wartend*, bis kritischer Abschnitt wieder frei
2. reine SW-Lösung vs. hardwareunterstützte Methoden
   - Unterstützung: spezielle Maschineninstruktionen
3. zentrale vs. dezentrale Lösungen
   - das Betriebssystem stellt die Funktionen zur Verfügung oder
   - die Prozesse regeln den Zugriff zu den Ressourcen selbst

Versuch, dezentral zu koordinieren

Naiver Versuch: „Ping-Pong“

Prozess π_1		Prozess π_2
`init():`	`var = 1;`	
`enter_cs():`		
`while(var==2)`		`while(var==1)`
`;`		`;`
`/* k.A. */`		
`leave_cs():`		
`var=2;`		`var=1;`

- es kann stets nur einer den kritischen Abschnitt betreten
- ABER: alternierende Eintrittsreihenfolge erzwungen! (Was, wenn π_2 nie k.A. betreten will, π_1 aber andauernd?)

Versuch, dezentral zu koordinieren

Naiver Versuch: „Ping-Pong“

Prozess π_1		Prozess π_2
`init():`	`var = 1;`	
`enter_cs():`		
`while(var==2)`		`while(var==1)`
`;`		`;`
`/* k.A. */`		
`leave_cs():`		
`var=2;`		`var=1;`

- es kann stets nur einer den kritischen Abschnitt betreten
- ABER: alternierende Eintrittsreihenfolge erzwungen! (Was, wenn π_2 nie k.A. betreten will, π_1 aber andauernd?)

Versuch, dezentral zu koordinieren

Variante 2

Umwidmung von `var`.

- `var == 0`: kritischer Abschnitt frei
- `var == 1`: kritischer Abschnitt belegt

	Prozess π_1	Prozess π_2
`init():`		`var = 0;`
`enter_cs():`		
	`while(var == 1)`	`while(var == 1)`
	`;`	`;`
	`var = 1;`	`var = 1;`
`/* k.A. */`		
`leave_cs():`		
	`var = 0;`	`var = 0;`

Versuch, dezentral zu koordinieren

Variante 2 – Bewertung

- Vorteil: jeder Prozess führt identischen Code aus
- Problem: Unterbrechung zwischen `while` und Schreiben auf `var`
- $\rightarrow$ Möglichkeit, dass mehr als ein Prozess im k. A.

Versuch, dezentral zu koordinieren

Variante 3

- offenbar reicht eine Variable nicht aus
- $\rightarrow$ eine Variable pro Prozess

	Prozess π_1	Prozess π_2
`init():`	`var1 = 0;`	`var2 = 0;`
`enter_cs():`		
	`while(var2 == 1)`	`while(var1 == 1)`
	`;`	`;`
	`var1 = 1;`	`var2 = 1;`
`/* k.A. */`		
`leave_cs():`		
	`var1 = 0;`	`var2 = 0;`

Versuch, dezentral zu koordinieren

Variante 3 – Bewertung

- Problem der Unterbrechung zwischen `while` und Schreiben auf `var`x persistiert
- Erhöhung der Komplexität ohne Qualitätsverbesserung

Versuch, dezentral zu koordinieren

Variante 4

- Idee: Vertauschung von `while`-Schleife und Variablenzugriff
- bei Unterbrechung nach `while` ist man bereits im k. A.
- leichte Semantikverschiebung von `var`x: zeigt nun Wunsch an, den kritischen Abschnitt zu betreten

	Prozess π_1	Prozess π_2
`init():`	`var1 = 0;`	`var2 = 0;`
`enter_cs():`		
	`var1 = 1;`	`var2 = 1;`
	`while(var2 == 1)`	`while(var1 == 1)`
	`;`	`;`
`/* k.A. */`		
`leave_cs():`		
	`var1 = 0;`	`var2 = 0;`

Versuch, dezentral zu koordinieren

Variante 4 – Bewertung

- Gefahr des Doppeleintritts in k. A. gebannt
- dafür Gefahr einer zyklischen Wartebedingung (*Deadlock*) bei Unterbrechung in der `while`-Schleife
- pathologischer Fall weitaus wahrscheinlicher als bei Variante 3

Versuch, dezentral zu koordinieren

Variante 5

- Vermeidung des Deadlocks, indem in Warteschleife geprüft wird, ob weiterer Prozess wartet
- wenn ja $\rightarrow$ erhält dieser Vortritt

	Prozess π_1		Prozess π_2
`init():`	`var1 = 0;`		`var2 = 0;`
`enter_cs():`			
`label1:`	`var1 = 1;`	`label2:`	`var2 = 1;`
	`if (var2==1) {`		`if (var1==1) {`
	`  var1 = 0;`		`  var2 = 0;`
	`  goto label1;`		`  goto label2;`
	`}`		`}`
`/* k.A. */`			
`leave_cs():`			
	`var1 = 0;`		`var2 = 0;`

Versuch, dezentral zu koordinieren

Variante 5 – Bewertung

- kein Deadlock mehr möglich
- einziges Problem: zwei Prozesse wünschen genau gleichzeitig Eintritt (nur auf SMP-Maschinen möglich, trotzdem selten) $\rightarrow$ unendliche (aktive) Warteschleife
- aka *Livelock*
- pragmatische Lösung: kurze zufällige Verzögerung beim Rücksprung

Reicht das?

Algorithmus von Peterson

- dritte Variable `turn` zur Auflösung der Gleichzeitigkeit nötig

	Prozess π_1	Prozess π_2
`init():`	`var1 = 0;`	`var2 = 0;`
	`turn = 1;`	
`enter_cs():`		
	`var1 = 1;`	`var2 = 1;`
	`turn = 1;`	`turn = 2;`
	`while((var2==1)`	`while((var1==1)`
	`&&(turn==1))`	`&&(turn==2))`
	`;`	`;`
`/* k.A. */`		
`leave_cs():`		
	`var1 = 0;`	`var2 = 0;`

(Gary Peterson: *Myths about the Mutual Exclusion Problem*. Information Processing Letters, Vol. 12(3), 1981)

Algorithmus von Peterson – Bewertung

- im Falle der Gleichzeitigkeit sorgt `turn` für deren Auflösung
- Prozess, der zuerst `turn` beschreibt, gewinnt
- nachteilig: Funktionalität prozessspezifisch
- Weitere Randbedingungen für Funktionieren auf modernen (SMP-)Maschinen und Compilern:
  - Compiler darf Funktionen nicht optimieren
  - Verhinderung des *Instruction Reordering*
  - Schreiben der Variablen muss auf Speicher durchgesetzt werden, bevor nächste Instruktion ausgeführt wird (z. B. durch *Barrier*), Ursache: schwache Speicherkonsistenz!

Fazit: Dezentrale Lösungen

- Es ist möglich, softwarebasiert kritische Abschnitte zu schützen.
- eine der einfachsten korrekten Lösungen ist der Algorithmus von Peterson
- Nachteil: *busy waiting* verschwendet Prozessorzeit
- Wechselseitiger Ausschluss führt zu schlechter Parallelisierbarkeit

Zentrale Lösung: Semaphore

Motivation

Hauptnachteil aller *busy waiting*-basierten Lösungen: Prozesse warten aktiv, d. h. CPU-Zeit wird verschwendet $\rightarrow$ Jeder Prozess muss Code für Steuerung selbst implementieren.

Verbesserung: auf Eintritt wartende Prozesse sollten in den Zustand „wartend“ geschickt werden und erst wieder geweckt werden, wenn Eintritt erlaubt

Weitere Verbesserung: der Code sollte nur einmal implementiert werden und allen Prozessen als „Dienstleistung“ zur Verfügung stehen.

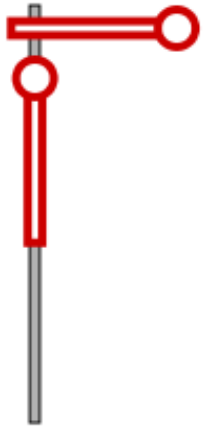
(Eine) Lösung: $\rightarrow$ abstrakter Datentyp *Semaphor* = Integer-Variable + Prozesswarteschlange

Zustände eines Semaphors

Semaphor kann zwei Zustände haben

offen – kritischer Abschnitt ist frei

geschlossen – kritischer Abschnitt ist gesperrt, da belegt



Analogie: Bahnsignal

Zwei (wesentliche) Operationen eines Semaphors

P() (, wait(), down(), lock())

- Semaphor prüft, ob kritischer Abschnitt frei
- wenn frei, Semaphor als *belegt* markieren und Rückkehr aus P()
- wenn belegt, dann den rufenden Prozess wartend (bezüglich des Semaphors) setzen (und einen anderen aktivieren)

V() (, signal(), up(), unlock())

- Semaphor prüft, ob Prozesse bezüglich des Semaphors warten
- wenn ja, einen solchen Prozess bereit setzen, Rückkehr aus V()
- wenn nein, Semaphor als *frei* markieren, Rückkehr aus V()

Weitere Operationen eines Semaphors

Init()

- Semaphor wird angelegt und entweder geöffnet oder geschlossen initialisiert

Try()

- ähnlich P()
- blockiert jedoch nicht, wenn Semaphor belegt, sondern kehrt mit Fehlercode zurück

Implementierung der P()-Funktion

```
void P()
{
  static int sem;

  lock();
  if (sem>0) {
    sem = sem - 1;
    unlock();
  }
  else {
    /* Prozess -> wartend (Warteschlange) */
    unlock();
  }
}
```

P()-Operation ist selbst ein kritischer Abschnitt!

Mehrere API für Semaphore unter Unix

1. Teil der sog. „System V“-IPC:
   - vergleichsweise alte API
   - Funktionen `semop()`, `semget()`, `semctl()`
   - relativ komplizierte Semantik
2. POSIX-Semaphore:
   - `man 7 sem_overview`
   - relativ neu
   - `sem_wait()`, `sem_post()`, `sem_open()`, `sem_close()`
3. Mutexe der Pthreads-Bibliothek
   - Konzept des Mutex (**Mut**ual **Ex**clusion device)
   - u. a. `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`

Weitere Semaphor-API

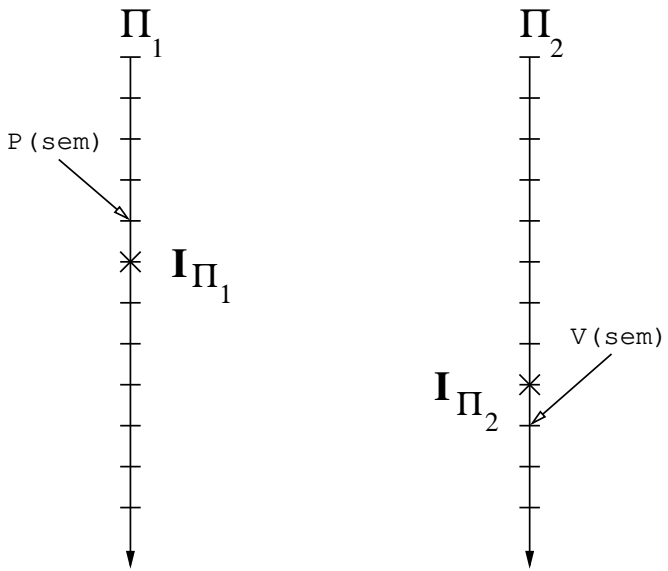
Win32:

- CreateSemaphore(), DeleteSemaphore(), WaitForSingleObject(), WaitForMultipleObjects(), ReleaseSemaphore()

Linux-Kern:

- Semaphore $\rightarrow$ Mutex
- kompliziert (vgl. `kernel/mutex.c`)
- nur für Kernel-Mode (z. B. Gerätetreiber)
- `mutex_lock()`, `mutex_unlock()`, ...

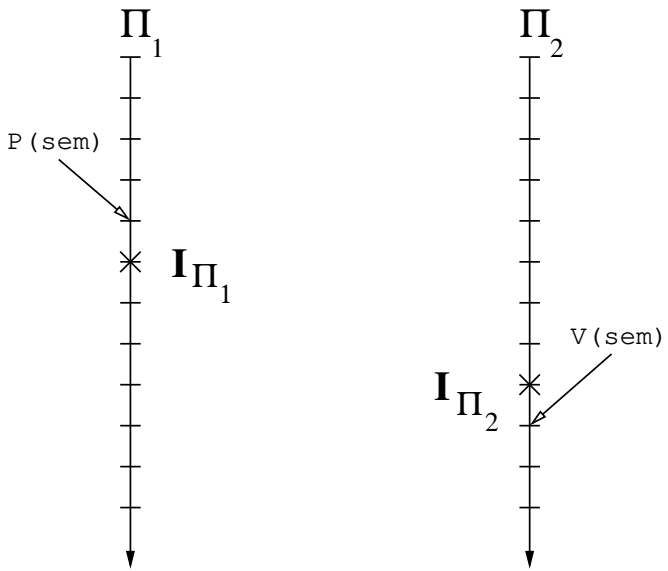
Anwendung: zeitliche Synchronisation von Prozessen



unabhängig von der konkreten Aktivierungsreihenfolge soll

- zuerst I_{Π_2} und
- danach I_{Π_1} passiert werden.
- 1 Semaphore, geschlossene Init
- V() vor P() (!)

Anwendung: zeitliche Synchronisation von Prozessen

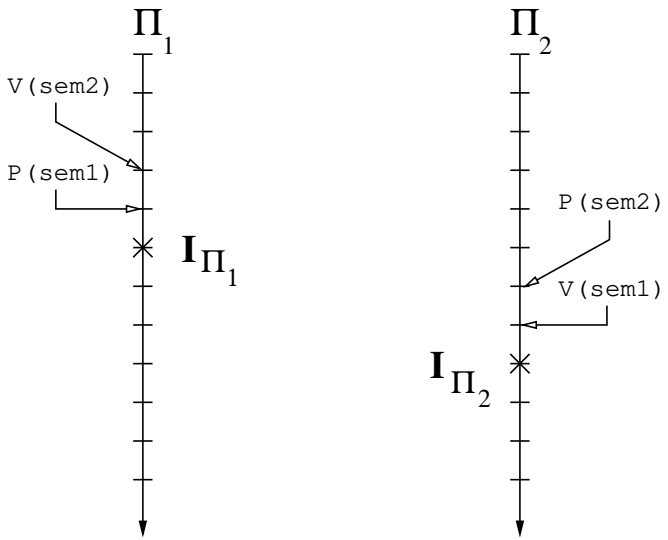


unabhängig von der konkreten Aktivierungsreihenfolge soll

- zuerst I_{Π_2} und
- danach I_{Π_1} passiert werden.
- 1 Semaphore, geschlossene Init
- V() vor P() (!)

Anwendung: zeitliche Synchronisation von Prozessen

Rendezvous

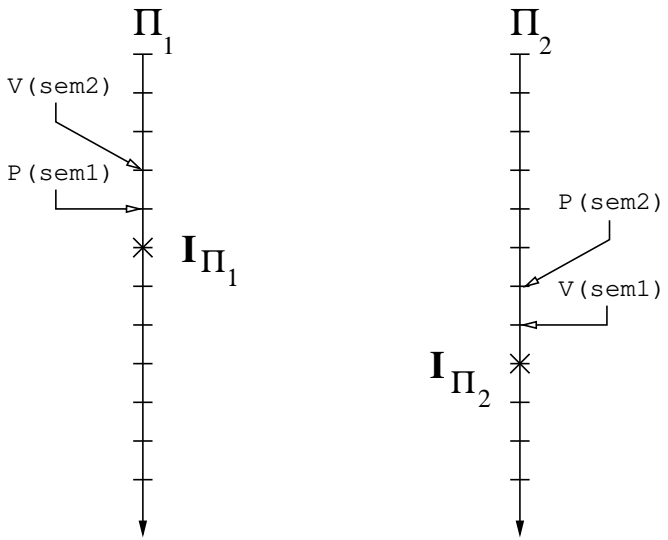


beide Prozesse sollen bei I_{Π_1} bzw. I_{Π_2} aufeinander warten (unabhängig von der konkreten Aktivierungsreihenfolge)

- 2 Semaphore, geschlossene Init
- Was passiert, wenn P() und V() in Π_1 vertauscht werden?

Anwendung: zeitliche Synchronisation von Prozessen

Rendezvous

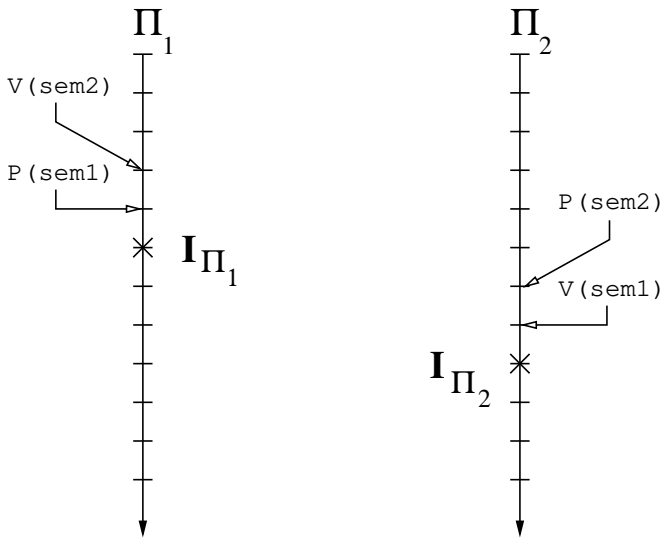


beide Prozesse sollen bei I_{Π_1} bzw. I_{Π_2} aufeinander warten (unabhängig von der konkreten Aktivierungsreihenfolge)

- 2 Semaphore, geschlossene Init
- Was passiert, wenn P() und V() in Π_1 vertauscht werden?

Anwendung: zeitliche Synchronisation von Prozessen

Rendezvous



beide Prozesse sollen bei I_{Π_1} bzw. I_{Π_2} aufeinander warten (unabhängig von der konkreten Aktivierungsreihenfolge)

- 2 Semaphore, geschlossene Init
- Was passiert, wenn P() und V() in Π_1 vertauscht werden?

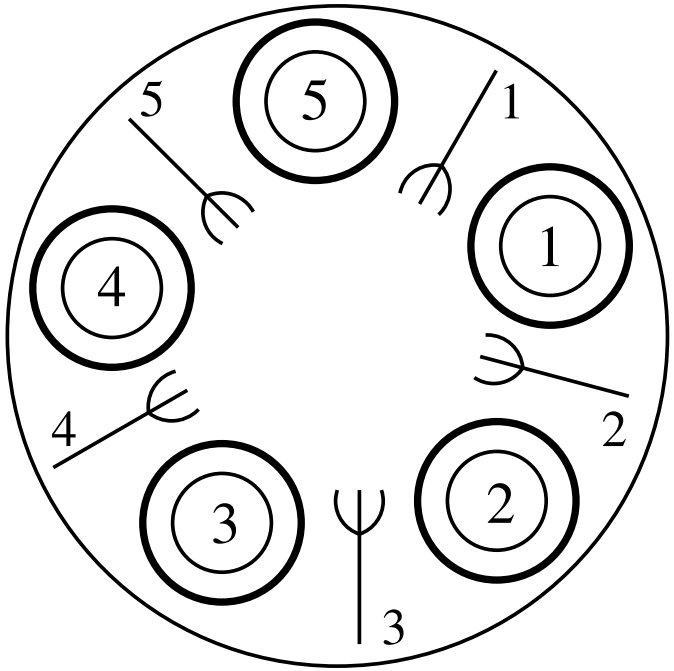
Weitere Aspekte

- neben *binären* gibt es auch *nichtbinäre* Semaphore $\rightarrow$ solange Zähler>0, kehrt P() zurück
- Nutzung zum Schutze relativ langer (bzw.) unbekannt langer kritischer Abschnitte
- Anwendung auch für komplexere Synchronisationsprobleme ($\uparrow$ BS2)

Das Problem der dinierenden Philosophen

Sie haben es gleich geschafft!

- 5 Proz–ähh–Philosophen
- 5 Ress–ähh–Gabeln
- Philosophen-Leben:
  1. Denken
  2. Essen
  3. Schlafen
  4. Goto 1.



Zum Essen benötigt ein Philosoph rechte *und* linke Gabel.

Lösungen bzw. -versuche

Lösung gesucht, so dass kein Philosoph verhungert.
Diskussion:

- philo1.c
- philo2.c
- philo3.c
- philo4.c
- philo5.c

Problem modelliert die Konkurrenz von Prozessen um beschränkte Ressourcen.

Was haben wir gelernt?

- Race Condition, kritischer Abschnitt
- Wechselseitiger Ausschluss
- klammernde Funktionen zur Steuerung des kritischen Abschnittes
- Versuche, kritische Abschnitte dezentral zu managen
- Algorithmus von Peterson
- Semaphore

Was haben wir nicht gelernt?

- Virtueller Speicher
- Deadlocks (Erkennen, Beheben, Vermeiden)
- kaum Prozesskommunikationsprobleme (Philosophen & Co.)
- Sicherheit
- Netzkommunikation
- Win32-API
- die meisten Implementierungsaspekte (Speicherverwaltung, Dateisysteme, Spinlocks, . . .)
- Aspekte der Verteilung

Ausblick

Falls Wiedersehen Freude macht . . .

- Betriebssysteme II (Informatik)
- Informationssicherheit
- Echtzeitsysteme (wo)

Weitere Kontaktmöglichkeiten:

- Projektseminar
- Bachelor-Arbeit?
- Master-Arbeit?

The End.

(for now)