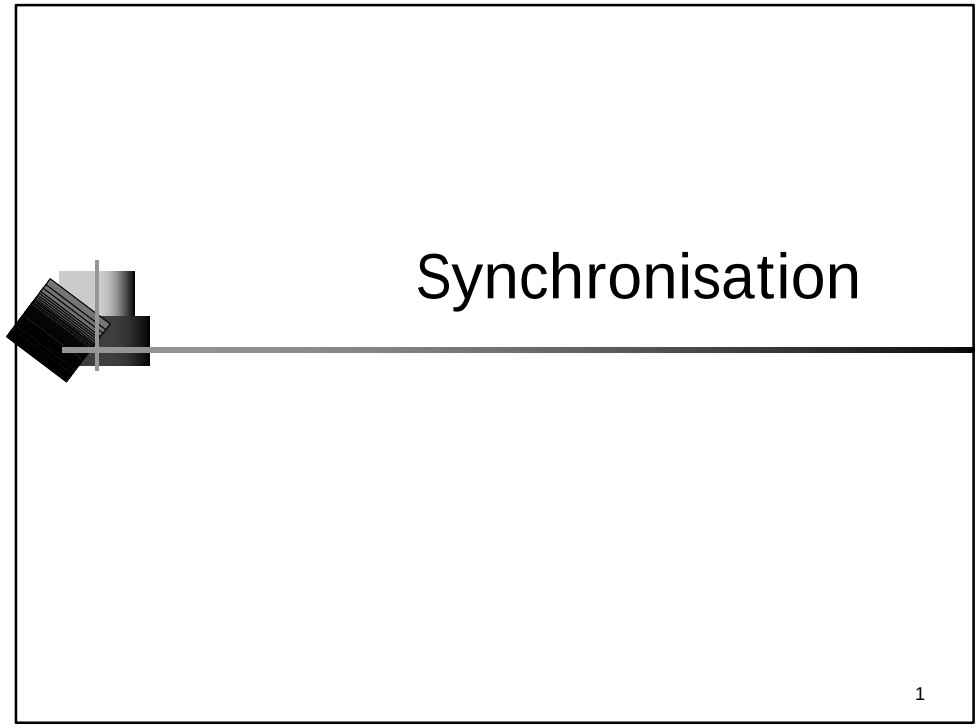
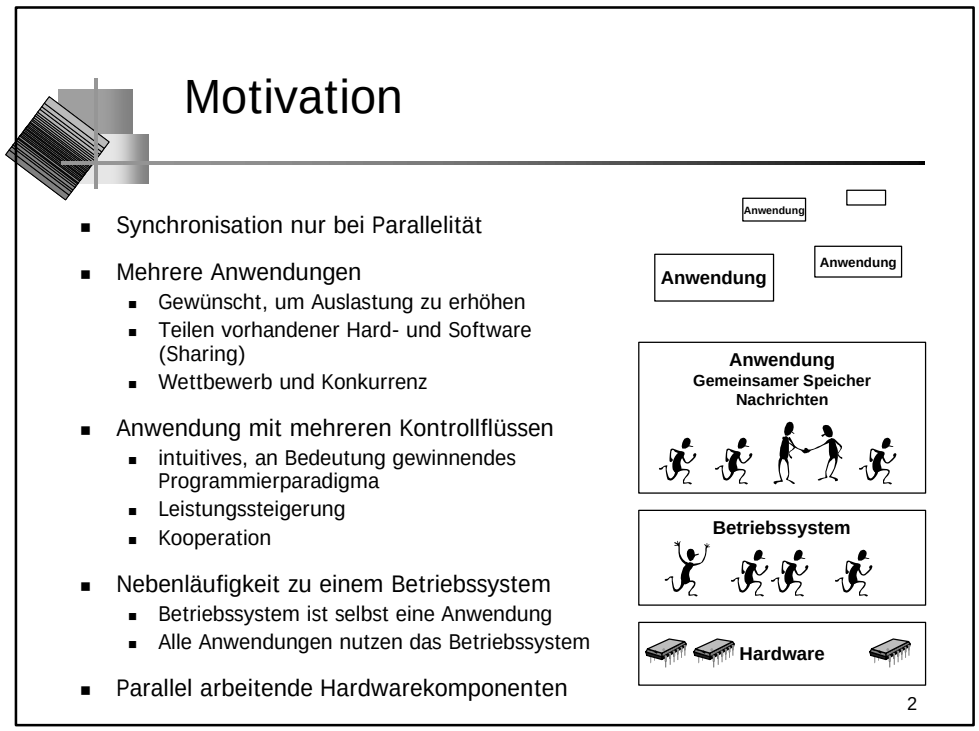
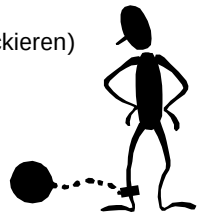

Synchronisation
1


Motivation
Synchronisation nur bei Parallelität
Mehrere Anwendungen
Gewünscht, um Auslastung zu erhöhen
Teilen vorhandener Hard- und Software (Sharing)
Wettbewerb und Konkurrenz
Anwendung mit mehreren Kontrollflüssen
intuitives, an Bedeutung gewinnendes Programmierparadigma
Leistungssteigerung
Kooperation
Nebenläufigkeit zu einem Betriebssystem
Betriebssystem ist selbst eine Anwendung
Alle Anwendungen nutzen das Betriebssystem
Parallel arbeitende Hardwarekomponenten
Anwendung
Anwendung
Anwendung
Anwendung
Gemeinsamer Speicher
Nachrichten
Betriebssystem
Hardware
2

Synchronisation =

- “Ausgleichen von Geschwindigkeitsunterschieden”
- (Notwendige) Einschränkung der Parallelarbeit
- “Langsamste” bestimmt die Geschwindigkeit aller
- Zwei Ansätze:
  - Langsamen Kontrollfluß beschleunigen
  - Schnelleren Kontrollfluß bremsen (blockieren)
- Ansatz 1 leider nur selten umsetzbar



3

Konkurrenz

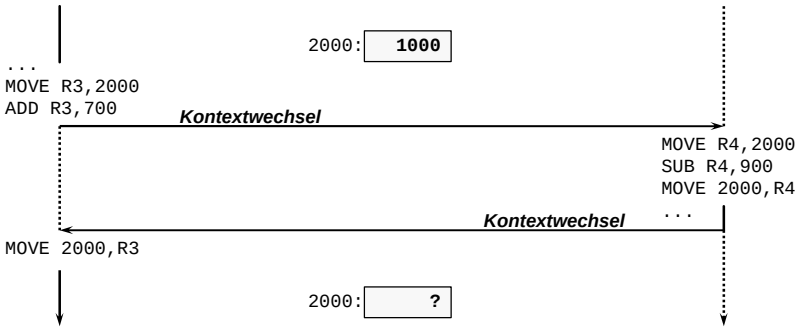
- Anwendungen konkurrieren um Rechner-Ressourcen
  - Anwendungsideal: Alleinige Nutzung des Gesamtrechners
- Ansätze
  - Explizit: Exklusive Nutzung
  - Implizit: Virtualisierung
- Probleme
  - Wechselseitiger Ausschluß
    - Exklusive Nutzung
  - Verklemmungen (Deadlocks)
  - Aushungerung (Starvation)

Kooperation

- Anwendungen kooperieren
- Teilen von Ressourcen
  - Interner und externer Speicher
  - Netzwerk
- Teilen von Aufgaben
  - Leistungssteigerung
  - Vermeidung von Engpässen
  - Fehlertoleranz
- Probleme
  - Wechselseitiger Ausschluß
    - Gemeinsam Lesen
    - Exklusiv Schreiben
  - Konsistenz

4

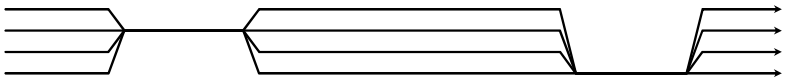
Quasi-Parallelität = Echte Parallelität



- Ohne Synchronisation kann ein Kontextwechsel bei preemptiven Scheduling jederzeit stattfinden
- Bei nicht-preemptiven Scheduling
  - Seitenfehler beim Ausführen einer Adresse
  - Aufruf von Bibliotheksfunktionen mit versteckten Blockadesituationen

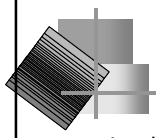
5

Wechselseitiger Ausschluß



- Fixierung an bestimmten Code-Passagen (kritischer Abschnitt)
  - Besondere Funktionen für Betreten und Verlassen des kritischen Abschnitts
  - Atomare Aktionen
    - Kann durch andere Kontrollflüsse nicht unterbrochen werden
    - Unteilbarkeit
- Eigenschaften
  - Maximal 1 Kontrollfluß darf sich im kritischen Abschnitt befinden
  - Kontrollfluß nur endliche Zeit im kritischen Abschnitt
  - Keine unendlichen Verzögerungen
  - Keine Verzögerung bei freiem kritischen Abschnitt
  - Keine Annahmen über relative Geschwindigkeiten und Prozeßanzahl

6

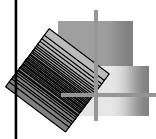


Explizite und implizite Synchronisation

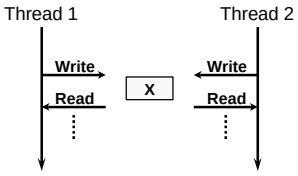
- Implizit
  - Anwendung wird indirekt blockiert
- Beispiele:
  - Seitenfehler
  - Ein- und Ausgabe
  - Schedulingentscheidungen (z.B. abgelaufene Zeitscheibe)
- Vorteile:
  - Transparenz
  - Unterstützung konkurrierender Anwendungen
  - weniger Synchronisationsfehler
- Nachteile:
  - u.U. unnötige Einschränkungen bei Parallelarbeit
  - Blockaderisiko nicht abschätzbar (Echtzeitanwendungen)

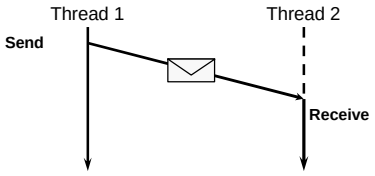
- Explizit
  - Anwendung verwendet Synchronisationsprimitive direkt
- Beispiele:
  - Wechselseitiger Ausschluß
  - Viele lesen, einer schreibt
  - ...
- Vorteile:
  - Potentielle Blockadesituation sichtbar
  - Blockaderisiko abschätzbar (Echtzeitanwendungen)
- Nachteile:
  - Anwendung muß synchronisieren
  - Potentiell Synchronisationsfehler

7



Kopplungsgrad





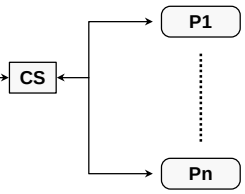
- Enge Kopplung
  - Gemeinsamer Speicher
  - Lese- und Schreiboperationen
- Mono- und Multiprozessoren
- Blockierende und Nicht-blockierende Ansätze
  - Blockieren = 2 Kontextwechsel + Δ
  - Busy Waiting bei kurzen Synchronisationsabschnitten

- Lose Kopplung
  - Rechnernetz
  - Austausch von Nachrichten
- Mono-, Multiprozessoren und Mehrrechnersysteme
- Überwiegend blockierende Ansätze

8

Beispiel

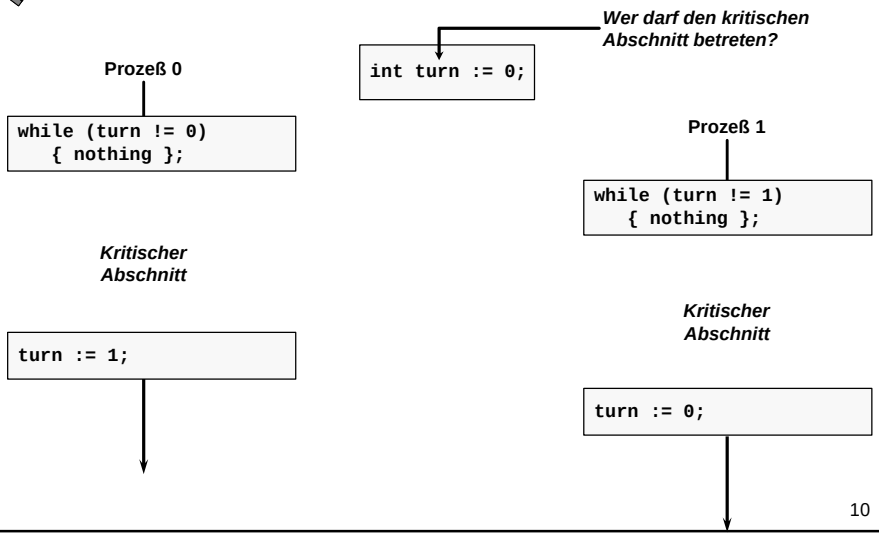
- Wechselseitiger Ausschluß ohne fremde Hilfe
  - keine Hardwareunterstützung
  - keine Betriebssystemunterstützung
  - keine Unterstützung durch Programmiersprachen
- Annahmen
  - Gemeinsamer Speicher zwischen Anwendungen
  - Einzelne Lese- und Schreiboperationen auf gleicher Adresse werden (willkürlich) serialisiert
- Programmstruktur
  - Prozeß k:



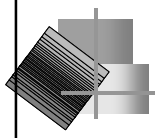

```
...
Begin_Critical_Section(CS);
   /* kritischer Abschnitt */
   ...
End_Critical_Section(CS);
...
```

9

Variante 1: Wer ist dran?



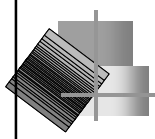
10



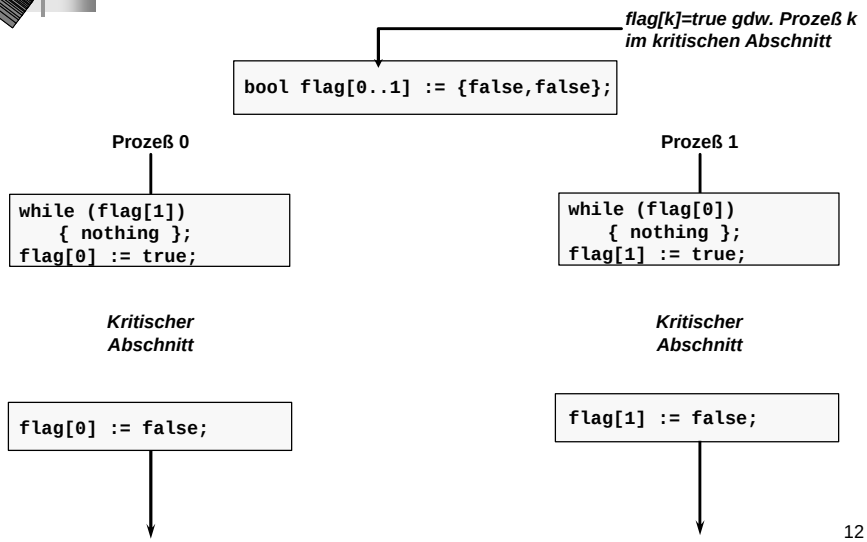
Bemerkungen

- Wechselseitiger Ausschluß garantiert?
- Busy Waiting
  - Nicht-blockierende Synchronisation
- Stärkere (implizite) Annahmen:
  - Voraussetzung: preemptives Scheduling
  - Nur duch Kontextwechsel kommt anderer Prozeß an die Reihe
- Alternierender Zugang zum kritischen Abschnitt
  - Nach Prozeß 0 muß zuerst wieder Prozeß 1 in den kritischen Abschnitt
  - Zugangsfrequenz und -reihenfolge unberücksichtigt
  - `turn` gibt an, wer als Nächstes darf
  - Verbesserung: Variante 2

11



Variante 2: Ist jemand drin?



12

Bemerkungen

```
bool flag[0..1] := {false,false};
```

Prozeß 0

```
while (flag[1]) do ;
flag[0] := true;
```

im kritischen Abschnitt

Prozeß 1

```
while (flag[0]) do ;
flag[1] := true;
```

im kritischen Abschnitt

- Wechselseitiger Ausschluß nicht garantiert!
  - Funktion zum Betreten des kritischen Abschnitts atomar ausgeführen
  - Betreten des kritischen Abschnitts ist selbst ein kritischer Abschnitt
- Test wird mit wachsender Prozeßanzahl aufwendig
- Problem: Testen und kurz darauf setzen

13

Variante 3: Erst setzen, dann testen

flag[k]=true gdw. Prozeß k will in kritischen Abschnitt

```
bool flag[0..1] := {false,false};
```

Prozeß 0

```
flag[0] := true;
while (flag[1])
    { nothing };
```

Kritischer Abschnitt

```
flag[0] := false;
```

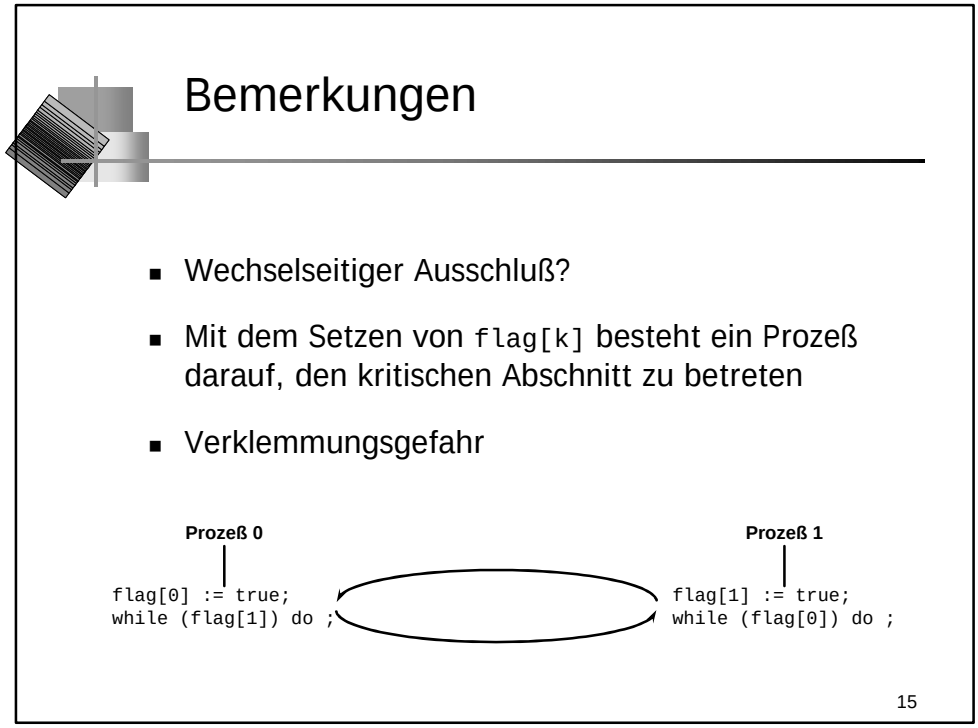
Prozeß 1

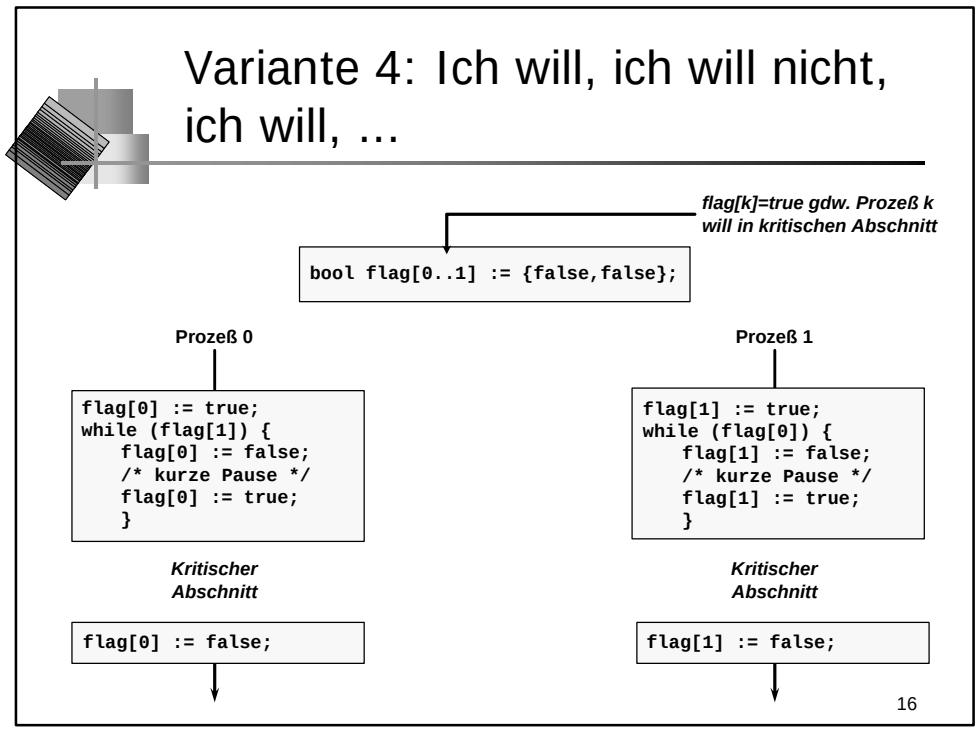
```
flag[1] := true;
while (flag[0])
    { nothing };
```

Kritischer Abschnitt

```
flag[1] := false;
```

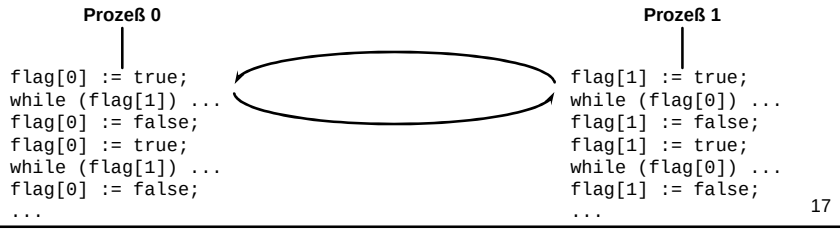
14


Bemerkungen
Wechselseitiger Ausschluß?
Mit dem Setzen von flag[k] besteht ein Prozeß darauf, den kritischen Abschnitt zu betreten
Verklemmungsgefahr
Prozeß 0
Prozeß 1
flag[0] := true;
while (flag[1]) do ;
flag[1] := true;
while (flag[0]) do ;
15


Variante 4: Ich will, ich will nicht, ich will, ...
flag[k]=true gdw. Prozeß k will in kritischen Abschnitt
bool flag[0..1] := {false,false};
Prozeß 0
Prozeß 1
flag[0] := true;
while (flag[1]) {
flag[0] := false;
/* kurze Pause */
flag[0] := true;
}
flag[1] := true;
while (flag[0]) {
flag[1] := false;
/* kurze Pause */
flag[1] := true;
}
Kritischer Abschnitt
Kritischer Abschnitt
flag[0] := false;
flag[1] := false;
16

Bemerkungen

- Wechselseitiger Ausschluß ist garantiert
- Bei gleich “getakteten” Prozessen können verklemmungs-ähnliche Zustände eintreten
  - Wahrscheinlichkeit gering, aber größer 0
  - Livelock
  - “mutual courtesy” (siehe W. Stallings)



17

Dekker´s Algorithmus

```
bool flag[0..1] := {false,false};
int turn := 0;
```

Prozeß 0

```
flag[0] := true;
while (flag[1]) {
    if (turn = 1) then {
        flag[0] := false;
        while (turn = 1)
            ;
        flag[0] := true;
        }
    }
```

Kritischer Abschnitt

```
turn := 1;
flag[0] := false;
```

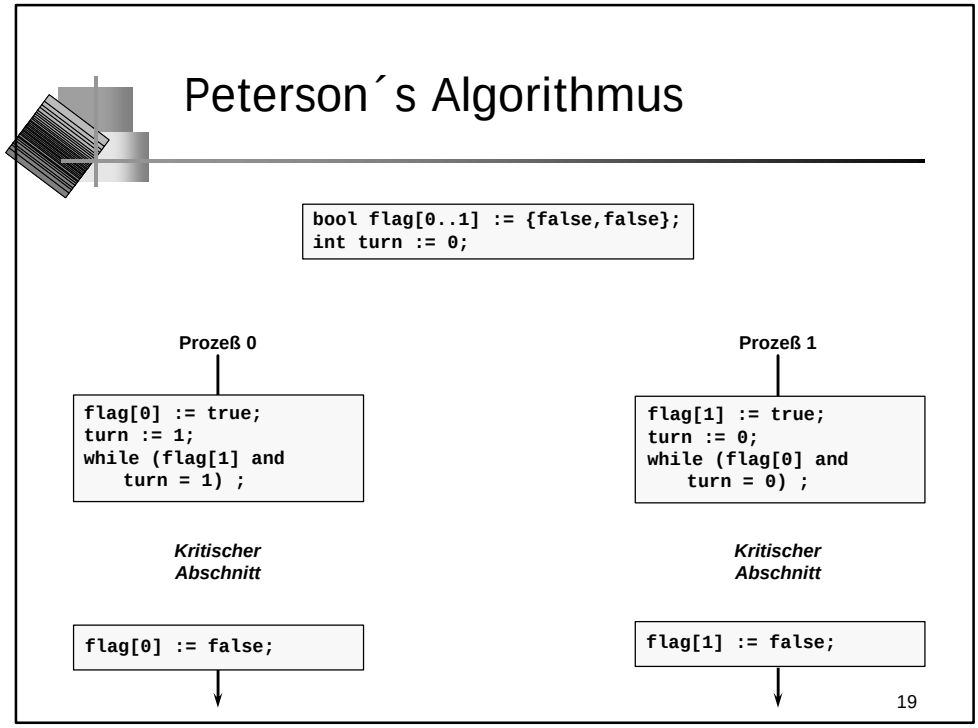
Prozeß 1

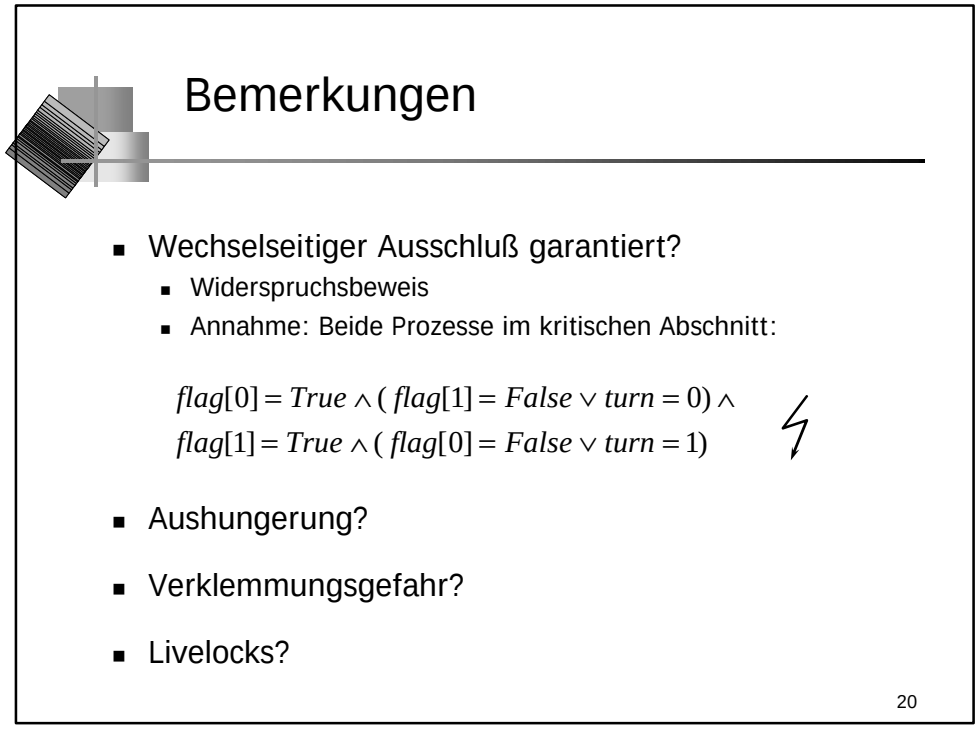
```
flag[1] := true;
while (flag[0]) {
    if (turn = 0) then {
        flag[1] := false;
        while (turn = 0)
            ;
        flag[1] := true;
        }
    }
```

Kritischer Abschnitt

```
turn := 0;
flag[1] := false;
```

18


Peterson´s Algorithmus
bool flag[0..1] := {false,false};
int turn := 0;
Prozeß 0
flag[0] := true;
turn := 1;
while (flag[1] and
turn = 1) ;
Kritischer
Abschnitt
flag[0] := false;
Prozeß 1
flag[1] := true;
turn := 0;
while (flag[0] and
turn = 0) ;
Kritischer
Abschnitt
flag[1] := false;
19


Bemerkungen
Wechselseitiger Ausschluß garantiert?
Widerspruchsbeweis
Annahme: Beide Prozesse im kritischen Abschnitt:
flag[0] = True ∧ (flag[1] = False ∨ turn = 0) ∧
flag[1] = True ∧ (flag[0] = False ∨ turn = 1)
Aushungerung?
Verklemmungsgefahr?
Livelocks?
20

Synchronisation ohne fremde Hilfe

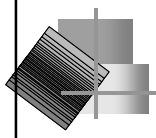
- Wird schnell kompliziert und unübersichtlich
  - Schwierig, alle Möglichkeiten abzudecken
  - Hohe Wahrscheinlichkeit für Synchronisationsfehler
    - Verklemmungen
    - Livelocks
    - Aushungerung
- Busy-Waiting-Varianten fragwürdig
- Unterstützung wünschenswert
  - Hardware-gestützte Synchronisationstechniken
  - Synchronisationsprimitive des Betriebssystems
  - Parallele und verteilte Programmiersprachen mit Synchronisationselementen

21

Hardware-gestützte Synchronisation

- Wünschenswert: Unteilbarkeit mehrerer Instruktionen
  - Vermeidung inkonsistenter Zwischenzustände bei Nebenläufigkeit
  - Welche “Wechselseitiger Ausschluß“-Variante wäre dann bereits sinnvoll einsetzbar?
- Minimale Anforderung
  - Unteilbarkeit einer Lese- und Schreiboperation
- Zwei Ansätze
  - Spezielle unteilbare Instruktionen
  - Sperren von Interrupts

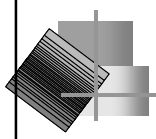
22



Ausschalten von Interrupts

- Spezielle Instruktionen
  - Enable Interrupts:
    - Eintreffende Interrupts werden angenommen
  - Disable Interrupts:
    - Eintreffende Interrupts lösen keine Unterbrechung aus
    - Zwischenspeicherung
    - kein Zählen (1-Bit Speicher)
    - Ausnahmen: nicht maskierbare Interrupts (NMI)
- Konsequenz:
  - Instruktionen zwischen `Disable` ... `Enable` auf einem Prozessor unteilbar
- Nachteile
  - Für Multiprozessorsysteme ungeeignet
  - Keine Gerätebehandlung während der Sperre
  - Lange Sperren kritisch bei Echtzeitanwendungen
  - Priveligierte Instruktionen

23



Beispiel

- Anwendung z.B. bei wechselseitigem Ausschluß (Variante 2)
- Erster Ansatz:
  - Prozeß 1:

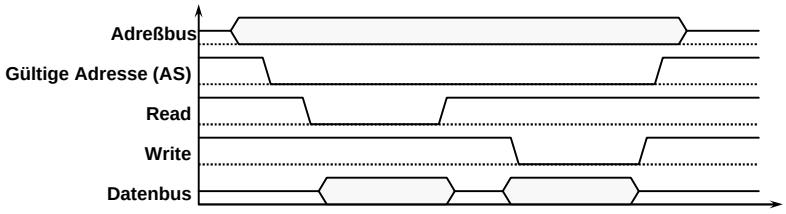
```
DI
while (flag[0]) ;
flag[1] := True;
EI
```

  - Warum funktioniert es so nicht?
- Lösung:
  - Prozeß 1:

```
BCS: DI
     H := Flag[0]
     if (H = False) goto CS
     EI
     goto BCS
CS:  Flag[1] := True;
     EI
```

24

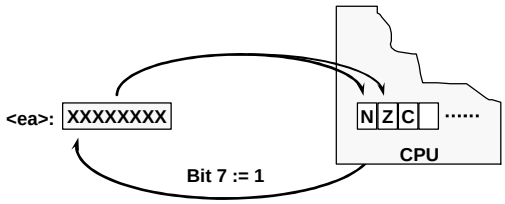
Spezielle Maschineninstruktionen



- Kombinierter Lese- und Schreibzyklus
- Wird von jedem modernen Prozessor unterstützt
  - SPARC: LDSTUB (Atomic Load-Store Unsigned Byte), SWAP
  - 680x0: TAS (Test and Set)
  - ...

25

Beispiel: TAS <ea> (Test and Set)

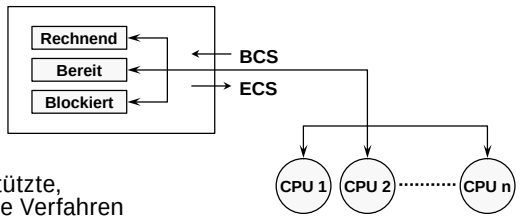


- Unteilbares Testen und Setzen eines Bits
- Einfacher wechselseitiger Ausschluß:

```
        X := 0
        ...
BCS:    TAS X
        JNZ BCS   /* Jump not zero */
        ...
        /* Kritischer Abschnitt */
        ...
ECS:    X := 0
```

26

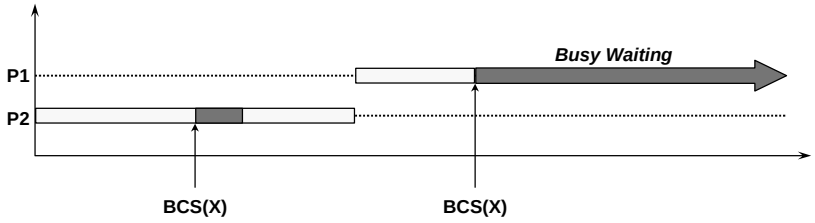
Spin Locks



- hardware-unterstützte, nicht-blockierende Verfahren
- Sinnvoll bei kurzen Blockaden
  - Synchronisation auf schnell eintretende Hardware-Ereignisse
  - Wechselseitiger Ausschluß bei der Manipulation der Prozeß-Zustandslisten in einem Multiprozessor
  - Synchronisation zwischen gleichzeitig ablaufenden Kontrollflüssen auf einem Multiprozessor
- Potentielle Probleme
  - Aushungerung
  - Livelocks
  - Leistung: Bei Multiprozessoren Caching nicht anwendbar (Cache Disable)

27

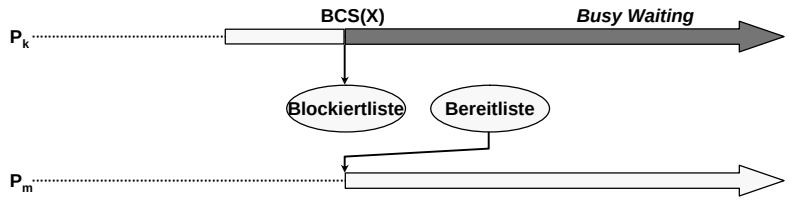
Spin Locks und Prioritäten



- Beispiel für Verklemmungssituation
  - P1 hohe Priorität
  - P2 niedrige Priorität
  - P2 betritt kritischen Abschnitt und wird danach von P1 unterbrochen
- Lösungsansatz?

28

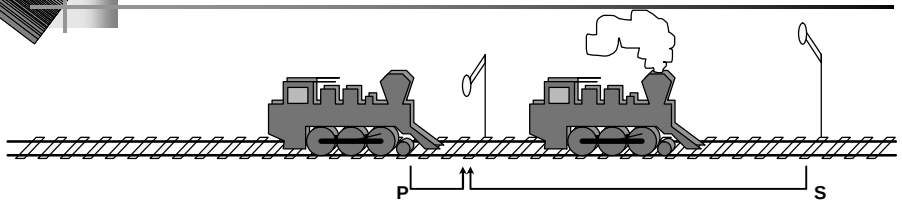
Betriebssystem-gestützte Synchronisation



- Sinnvolle Nutzung von Wartezeiten
  - Implizit: Anwendung ruft blockierende Systemfunktion auf
  - Explizit: Anwendungsspezifische Synchronisation
    - Blockierende Synchronisationsprimitive
- Kosten für blockierende Synchronisation
  - Kontext des blockierten Kontrollflusses retten
  - unbestimmte Anzahl anderer Kontrollflüsse
  - Kontextwechsel nach Eintritt der Synchronisationsbedingung

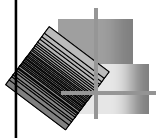
29

Allgemeine Semaphore



- Elementares Synchronisationsprimitiv
- Zwei Grundfunktionen (Semaphor S)
  - P(S)
    - Bei Freigabe: Aufruf beendet; Ausführung fortsetzen
    - Bei Belegt: Kontrollfluß am Semaphor blockieren
  - V(S)
    - Signalisieren: einen am Semaphor blockierten Kontrollfluß befreien
- Dijkstra (1968), “THE” Multiprogramming System
  - P = passeren (passieren)
  - V = vrygeven (freigeben)

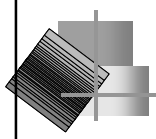
30



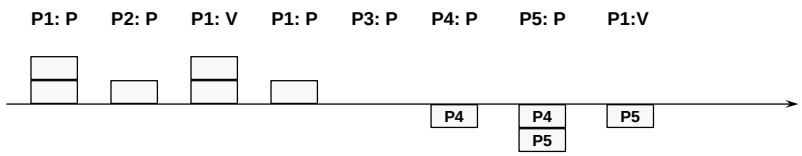
Semantik

- Semaphore sind einfache (zählende) Signale
  - P(S): Kontrollfluß wartet auf die Ankunft eines bestimmten Signals S
  - V(S): Kontrollfluß sendet ein bestimmtes Signal S
- Semaphor = Ganze Zahl
  - Initialisierung: beliebiger Wert größer oder gleich 0
  - P(S):
    - Wert dekrementieren
    - Bei Result < 0 Kontrollfluß blockieren
  - V(S):
    - Wert erhöhen
    - Bei Result < 1 einen am Semaphor blockierten Prozeß befreien

31



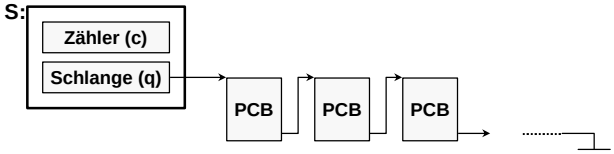
Semaphor-Invariante



- Beispiel
  - Initialisierung mit dem Wert 3
  - Prozesse P1 bis $\#P \leq \#V + init$
- Invariante
- Anzahl der beendeten (nicht-blockierenden) P-Aufrufe darf den beendeten V-Aufrufen höchstens um init vorauslaufen

32

Realisierung der P- und V-Operationen



- P-Operation
  - Ausführung atomar
  - Prozeß k führt P-Operation aus

```
s.c := s.c - 1;
if (s.c < 0) {
   tail(s.q) := k;
   block(k);
   }
```

- V-Operation
  - Ausführung atomar
  - Ausführender Prozeß blockiert in der Regel nicht

```
s.c := s.c + 1;
if (s.c <= 0) {
   ready(head(s.q))
   delhead(s.q)
   }
```

33

Binäre Semaphore

- Wert des Semaphors ist nur 0 oder 1 (Invariante)
- Abbildung auf allgemeine Semaphore gängig
  - Achtung V-Operation: Einfaches Inkrementieren verletzt Invariante
- P-Operation

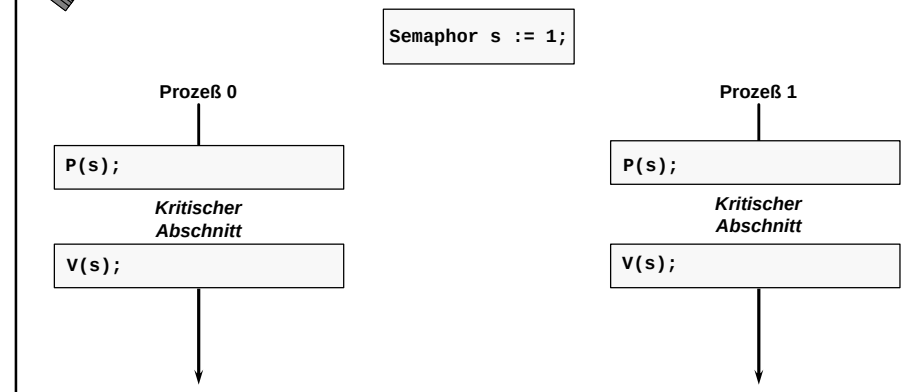
```
if (s.c = 0) {
   tail(s.q) := k;
   block(k);
   }
s.c := 0;
```

- V-Operation

```
if (s.q != 0) {
   ready(head(s.q));
   delhead(s.q)
   }
else
   s.c := 1;
```

34

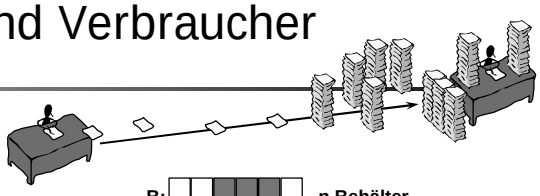
Wechselseitiger Ausschluß



- Korrektheit, Verklemmungen, Aushungerung?

35

Erzeuger und Verbraucher




```
Aus := Ein := 0;
Semaphor Frei := n;
Semaphor Arbeit := 0;
Bsemaphor Mutex := 1;
```

- Erzeuger

```
Forever {
   Erzeuge Element;
   P(Frei);
   P(Mutex);
   B[Ein] := Element;
   Ein := (Ein+1) mod n;
   V(Mutex);
   V(Arbeit);
   }
```

- Verbraucher

```
Forever {
   P(Arbeit);
   P(Mutex);
   Element := B[Aus];
   Aus := (Aus+1) mod n;
   V(Mutex);
   V(Frei);
   Verbrauche Element;
   }
```

36

Reader-Writer-Problem

- Abgestufte Ausschlußkriterien
  - Lesender Zugriff
    - Keine Veränderung der Daten
    - Andere Reader können ebenfalls eintreten
  - Schreibender Zugriff
    - Inkonsistenzen; weder lesender noch schreibender Zugriff
- Varianten bgzl. der Priorisierung
  - Lesen bevorzugen
    - Wenn ein Reader bereits Zutritt hat, weitere Reader zulassen
    - Risiko: Aushungern der Writer
  - Schreiben bevorzugen
    - Sobald ein Writer Eintritt verlangt, diesen möglichst früh gestatten
    - Risiko: Aushungern der Schreiber

37

Reader bevorzugen

```
int anzahl_reader := 0;
Bsemaphor Mutex := 1;
Bsemaphor KA := 1;
```

- Reader

```
Forever {
   P(Mutex);
   anzahl_reader += 1;
   if (anzahl_reader = 1)
      P(KA);
   V(Mutex);

   Daten lesen

   P(Mutex);
   anzahl_reader -= 1;
   if (anzahl_reader = 0)
      V(KA);
   V(Mutex);
   }
```

- Writer

```
Forever {
   P(KA);

   Daten schreiben

   V(KA);
   }
```

38

Writer bevorzugen

- Vorrecht für Eintritt wird von Writer zu Writer weitergegeben
  - Analog zur vorherigen Reader-Lösung
  - “Passing the Baton”
- Writer

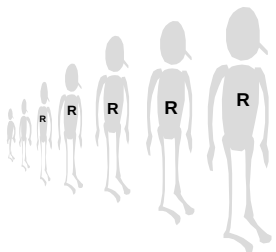
```
Forever {
   P(KA);

   Daten schreiben

   V(KA);
   }
```

```
int anzahl_writer := 0;
Bsemaphor Mutex2 := 1;
Bsemaphor R := 1;

Forever {
   P(Mutex2)
   anzahl_writer += 1;
   if (anzahl_writer = 1)
      P(R);
   V(Mutex2);
   P(KA);

   Daten schreiben

   V(KA);
   P(Mutex2)
   anzahl_writer -= 1;
   if (anzahl_writer = 0)
      V(R);
   V(Mutex2);
   }
```

39

Änderungen am Reader

- P(R) nicht blockierend gdw. kein Writer erhielt oder verlangt Eintritt
- Warum mutex3?
  - Übungsblatt 2




```
Bsemaphor Mutex3 := 1;

Forever {
   P(Mutex3);
   P(R);
   P(Mutex);
   anzahl_reader += 1;
   if (anzahl_reader = 1)
      P(KA);
   V(Mutex);
   V(R);
   V(Mutex3);

   Daten lesen

   P(Mutex);
   anzahl_reader -= 1;
   if (anzahl_reader = 0)
      V(KA);
   V(Mutex);
   }
```

40

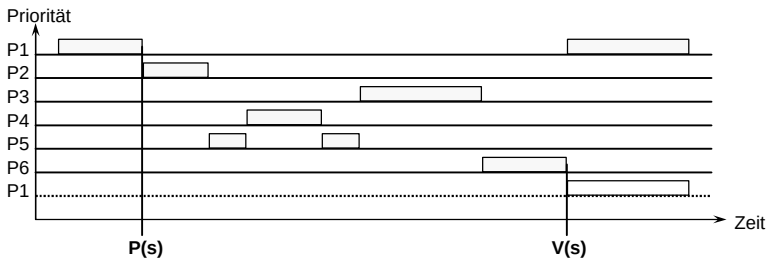
Geschachtelte (binäre) Semaphore

```
Bsemaphor Mutex := 1;

f1 (...) {        f2 (...) {                  fn (...) {
   P(Mutex);         P(Mutex);                   P(Mutex);
   ...               f1();          .....        ...
   V(Mutex);         V(Mutex);                   V(Mutex);
   }                 }                           }
```

- Schutz mehrerer kritischer Abschnitte (Prozeduren) durch ein binäres Semaphor
  - Wechselseitiger Ausschluß zwischen mehreren Bibliotheksfunktionen
    - Bibliotheken, die nicht gleichzeitig von mehreren Kontrollflüssen in einem Adreßraum genutzt werden können
- Realisierung
  - Kontrollfluß mit nicht-blockierender P-Operation vermerken
  - Weitere P-Aufrufe durch diesen Kontrollfluß nicht blockieren

41

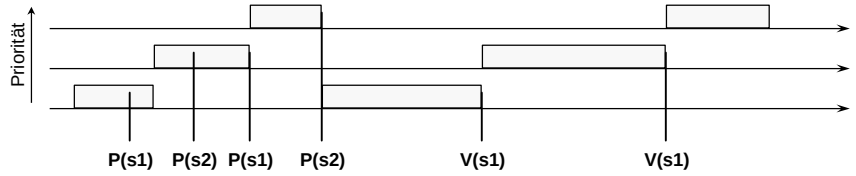
Semaphore und Prioritäten



- P1 und P6 synchronisieren sich über Semaphor s
- Prioritätsinversion
  - scheinbare Priorität von P1 ist niedriger als die von P6
- Kritisch bei Echtzeitanwendungen
  - Einhaltung der Fristen nicht mehr gewährleistet
  - Maximale Wartezeit für hochprioren Prozeß?

42

Maximale Wartezeit



- Beschränkung auf einen kritischen Abschnitt nicht garantiert
- Risiko mehrfacher Blockaden (Ketten)
- Maximal min(n,m) kritische Abschnitte bei Prozeß k, mit
  - n = Anzahl niederpriorer Prozesse mit Nutzung von Semaphoren, auf die Prozesse mit Priorität größer oder gleich Prio(k) ebenfalls zugreifen
  - m = Anzahl unterschiedlicher Semaphore, die von niederprioren Prozessen belegt werden können

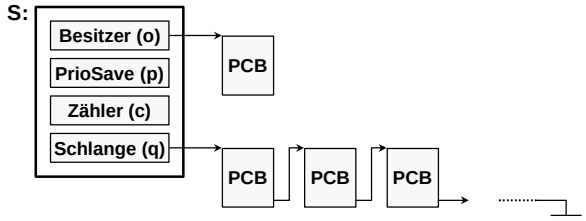
43

Prioritätsvererbung (Priority Inheritance)

- Ein niederpriorer Prozeß, der einen höherprioren Prozeß blockiert, erbt höhere Priorität bis zur Deblockade
- Solaris-Betriebssystem nutzt diese Technik

44

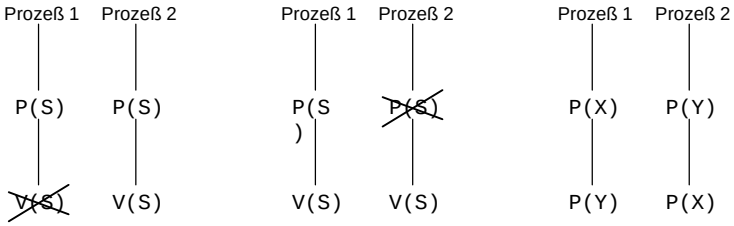
Realisierung von Prioritätsvererbung



- Beschränkung auf binäre Semaphore
- P-Operation
  - Keine Blockade: Ursprüngliche Priorität und PCB werden gespeichert
  - Blockade: Eventuell höhere Priorität auf Besitzer übertragen
- V-Operation
  - Ursprüngliche Priorität des Prozesses wiederherstellen
  - Blockierten Prozeß mit maximaler Priorität aktivieren

45

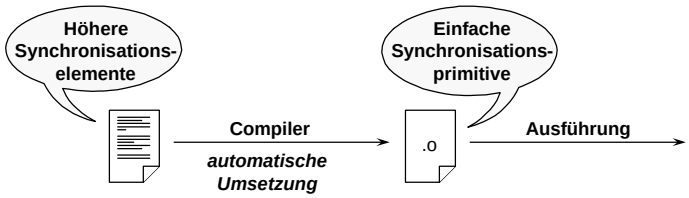
Häufige Fehler bei Semaphoren



- Synchronisation kann kompliziert und umständlich werden
- Weglassen (Vergessen) von P- und V-Operationen
  - z.B. kein wechselseitiger Ausschluß mehr
  - Verklemmungen
  - Inkonsistenzen durch falsche Zählerstände bei allgemeinen Semaphoren
- Gegenläufige Schachtelung
  - Verklemmungen

46

Sprach-gestützte Synchronisation



- Parallele und verteilte Programmiersprachen
  - Implizit: z.B. parallelisierende Fortran 90 Compiler (Vektorrechner)
  - Explizit: ADA, Modula-2, Modula-3, Concurrent Pascal, LINDA, CSSA, ...
- Elegante Synchronisationsmittel = Sprachelemente
  - z.T. aufwendige Umsetzung durch Compiler
  - Fehler bei der Anwendung von Synchronisationsprimitiven (Semaphore, ...) unwahrscheinlich
  - Synchronisationszwang

47

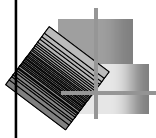
Bedingte kritische Abschnitte

- Hoare, 1972
- Bedingung kann gemeinsame und lokale Variablen referenzieren
- Compiler sperrt kritische Abschnitte korrekt

```
Shared sv {
   ...
   };

Region sv When b do
   Anweisungen
end
```

```
Bsemaphor bka := 1;

P(bka);
while (not b) {
   V(bka);
   P(bka);
   }
Anweisungen;
V(bka);
```

48



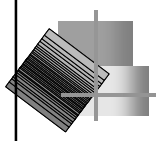
Vermeidung von Busy Waiting

- Explizite Blockade, bis ein anderer Prozeß die Daten eventuell verändert hat
- Nach dem Datenzugriff haben alle blockierten Prozesse die Möglichkeit, die Bedingung erneut zu überprüfen

```
Bsemaphor bka := 1;

P(bka);
while (not b) {
   V(bka);
   P(bka);
   }
Anweisungen;
V(bka);
```

```
Bsemaphor bka := 1;
Bsemaphor bsem := 0;
int wait_count := 0;

P(bka);
while (not b) {
   wait_count += 1;
   V(bka);
   P(bsem);
   P(bka);
   }

Anweisungen;

while (wait_count > 0) {
   wait_count -= 1;
   V(bsem);
   }
V(bka);
```

49



Monitore

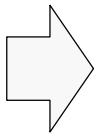
- T. Hoare, 1974
- Zusammenführung der gemeinsam verwendeten Daten und der darauf angewandten Zugriffsfunktionen
  - Abstrakter Datentyp
  - Klassenbegriff (objektorientierte Programmierung)
- Bei der Ausführung schließen sich alle Zugriffsfunktionen wechselseitig aus
  - Maximal ein Kontrollfluß befindet sich zu einem Zeitpunkt im Monitor

```
Monitor M {
   gemeinsame Daten;

   Entry f1 ( Parameter )
      : Resultat {
      Anweisungen;
      }

   ...

   Entry fn ( Parameter )
      : Resultat {
      Anweisungen;
      }
   }
```

50

Umsetzung durch Compiler

```
Monitor M {
   gemeinsame Daten;

   Entry f1 ( Parameter )
      : Resultat {
      Anweisungen;
      }

   ...

   Entry fn ( Parameter )
      : Resultat {
      Anweisungen;
      }
   }
```



```
Monitor M {
   gemeinsame Daten;
   Bsemaphor MonSem := 1;

   Entry f1 ( Parameter )
      : Resultat {
      P(MonSem);
      Anweisungen;
      V(MonSem);
      }

   ...

   Entry fn ( Parameter )
      : Resultat {
      P(MonSem);
      Anweisungen;
      V(MonSem);
      }
   }
```

- Korrekte Semaphorverwendung durch Compiler
  - Wechselseitiger Ausschluß garantiert

51

Erzeuger und Verbraucher I

- Puffer und Pufferverwaltung im Monitor

```
Monitor Puffer {
   Element b[n];
   int Aus := Ein := 0;

Erzeuger ──→ Entry Erzeuge ( Element e ) {
      b[Ein] := e;
      Ein := (Ein+1) mod n;
      }

   Entry Verbrauche () : Element { ←── Verbraucher
      Element e;
      e = b[Aus];
      Aus := (Aus+1) mod n;
      return e;
      }
   }
```

52

Erzeuger und Verbraucher II

Erzeuger:

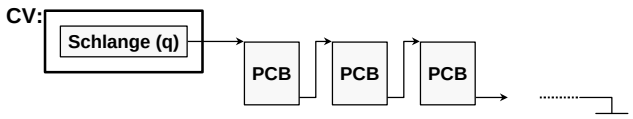
```
Monitor Puffer p;
Element e;
Boolean b;

Forever {
   Erzeuge e;
   repeat
      b := p.Erzeuge(e);
   until b;
   }
```

- Synchronisation bei vollem und leerem Puffer
  - Keine Blockade im Monitor möglich
  - Verwendung z.B. von Semaphoren fatal
- Busy-Waiting-Ansatz

```
Entry Erzeuge (Element e) : Boolean {
   if (Anzahl = n)
      return False;
   else {
      b[Ein] := e;
      Ein := (Ein+1) mod n;
      Anzahl += 1;
      return True;
      }
   }
```

53

Condition-Variablen



- Synchronisation von Kontrollflüssen innerhalb eines Monitors
  - `Wait ( Condition-Variable )`
    - Aufrufender Kontrollfluß wird immer blockiert
    - Monitor wird freigegeben
  - `Signal ( Condition-Variable )`
    - Ein an der Condition-Variablen blockierter Kontrollfluß k wird befreit
    - k bewirbt sich erneut um den Monitor
- Unterschiedliche Signal-Varianten

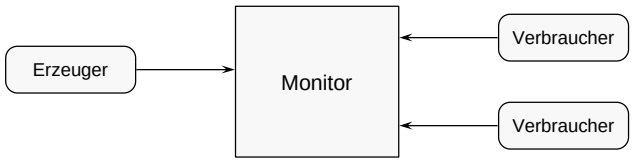
54

Erzeuger und Verbraucher III

```
Monitor Puffer {
   Element b[n];
   Condition Voll;
   Condition Leer;
   int Aus := Ein := Anzahl := 0;

Entry Erzeuge ( Element e ) {
   while (Anzahl = n)
      Wait(Leer);
   b[Ein] := e;
   Ein := (Ein+1) mod n;
   Anzahl += 1;
   Signal(Voll);
   }

Entry Verbrauche () : Element {
   Element e;
   while (Anzahl = 0)
      Wait(Voll);
   e = b[Aus];
   Aus := (Aus+1) mod n;
   Anzahl -= 1;
   Signal(Leer);
   return e;
   }

}
```

55

Warum die While-Schleife



- Aufrufen der Signal-Funktion bedeutet meist Eintritt einer bestimmten Bedingung
  - Puffer enthält mindestens ein Element
  - Puffer enthält mindestens einen freien Behälter
- Signal befreit einen Prozeß, der auf den Eintritt dieser Bedingung wartet

- Beispiel

```
V1.Verbrauche()
   Wait(Voll)
E.Erzeuge(e)
   Signal(Voll)
V2.Verbrauche()
V1.Verbrauche()
   ??? Wait(Voll)
```

56

Eine zweite Signal-Variante

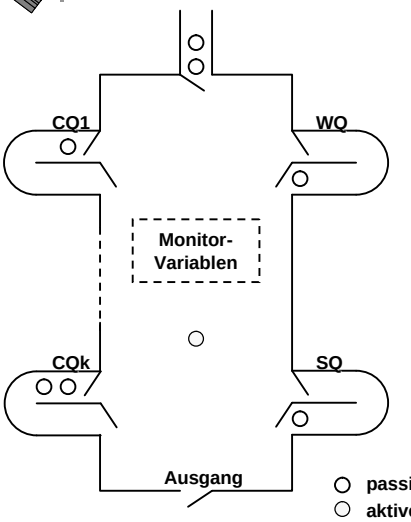
- Semantik
  - Signalisierende Prozeß wird blockiert
  - Signalisierter Prozeß (an Condition-Variable blockierter Prozeß) wird befreit
- Unter Umständen reicht einfache Bedingung
  - Bedingung muß tatsächlich zum Signal-Zeitpunkt erfüllt sein
  - Befreiter Prozeß muß Monitor als nächstes betreten
  - Signalisierender Prozeß betritt Monitor nochmal, nur um ihn zu verlassen

```
Entry Erzeuge ( Element e ) {
   if (Anzahl = n) then
      Wait(Leer);
   b[Ein] := e;
   Ein := (Ein+1) mod n;
   Anzahl += 1;
   Signal(Voll);
   }
```

```
Entry Verbrauche () : Element {
   Element e;
   if (Anzahl = 0) then
      Wait(Voll);
   e = b[Aus];
   Aus := (Aus+1) mod n;
   Anzahl -= 1;
   Signal(Leer);
   return e;
   }
```

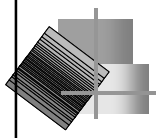
57

Aufbau eines Monitors



- Eintrittsliste (EQ)
  - Prozesse, die den Monitor betreten möchten
- Condition-Listen (CQ)
  - Prozesse, die an einer Condition-Variablen blockiert sind
- Signal-Liste (SQ)
  - Blockierte Prozesse, die Signal aufgerufen haben
- Warteliste (WQ)
  - Durch Signal befreite Prozesse

58

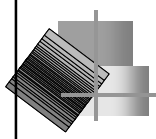


Monitor-Klassifikation

- Prioritäten der Listen EQ, SQ und WQ
- P.A. Buhr, M. Fortier, M.H. Coffin
  Monitor Classification
  ACM Computing Survey, Vol. 27, No. 1, pp. 63-107, 1995

EQp = WQp < SQp	Wait and Notify
EQp = SQp < WQp	Signal and Wait
EQp < WQp < SQp	Signal and Continue
EQp < SQp < WQp	Signal and Urgent Wait

59



Realisierungsbeispiel: Signal and Continue

- Prioritäten: EQp < WQq < SQq
- Wait- und Signal-Funktionen
  (Aufruf von Prozeß k)
  - Semaphor c.s für jede Condition-Variable c
  - Zähler c.c zählt an der Variablen c blockierte Prozesse
  - Analog Warteliste
  - Keine Signalliste (Continue)
- Entries

```
Entry f_k (...) {
   P(MonSem);
   ...
   if (w.c > 0)
      V(w.s);
   else
      V(MonSem);
   }
```

```
Wait ( Condition c ) {
   c.c += 1;
   if (w.c > 0)
      V(w.s);
   else
      V(MonSem);
   P(c.s);
   c.c -= 1;
   w.c += 1;
   P(w.s);
   w.c -= 1;
   }

Signal ( Condition c ) {
   if (c.c > 0)
      V(c.s);
   }
```

60

Monitorrealisierung eines BKA

- Condition-Variable bcv zum Aufsammeln der wartenden Prozesse
- Argumente des Region-Entries?

```
Shared sv {
   ...
   };

Region sv When b do
   Anweisungen
end
```

```
Monitor BKA {
   Gemeinsame Daten sv;
   Condition bcv;
   int wait_count := 0;

   Entry Region (...) {
      while (not b) {
         wait_count += 1;
         Wait(bcv);
         wait_count -= 1;
         }

      Anweisungen;

      while (wait_count > 0) {
         wait_count -= 1;
         Signal(bcv);
         }
      }
   }
```

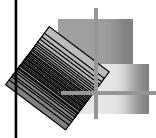
61

Eine dritte Signal-Variante

- Alle an einer Condition-Variablen blockierten Prozesse werden befreit und bewerben sich erneut um den Monitor
- Broadcast Signal and Continue (Howard, 1976)
- Problem
  - Häufig unnötige Prozeß-aktivierungen

```
Monitor BKA {
   Gemeinsame Daten sv;
   Condition bcv;

   Entry Region (...) {
      while (not b)
         Wait(bcv);

      Anweisungen;

      Signal(bcv);
      }
   }
```

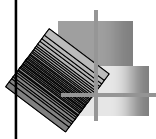
62



Alle 3 Signalvarianten im Überblick

- Signal and Continue
  - Eigenschaften
    - Signalisierender Prozeß bleibt im Besitz des Monitors
    - Maximal ein blockierter Prozeß wird befreit
  - Gängige Monitorvariante
- Signal and (Urgent) Wait
  - Eigenschaften
    - Signalisierender Prozeß gibt Monitorbesitz ab
    - Maximal ein blockierter Prozeß wird befreit und betritt sofort den Monitor
  - Ursprüngliche Signal-Semantik von Hoare
  - Signal meist am Ende der Entry-Funktion
- Broadcast Signal and Continue
  - Eigenschaften
    - Signalisierender Prozeß bleibt im Besitz des Monitors
    - Alle blockierten Prozesse werden befreit
  - Häufig unnötige Prozeßaktivierungen

63

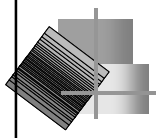


Reader-Writer: Monitorrealisierung

```
Monitor RW {
   int read_count := 0;
   Boolean Write := False;
   Condition OkToRead;
   Condition OkToWrite;

   Entry StartRead () {
      if (Write)
         Wait(OkToRead);
      read_count += 1;
      Signal(OkToRead);
      }

   Entry EndRead () {
      read_count -= 1;
      if (read_count = 0)
         Signal(OkToWrite);
      }
```

```
   Entry StartWrite () {
      if ((read_count>0) or Write)
         Wait(OkToWrite);
      Write := True;
      }

   Entry EndWrite () {
      Write := False;
      if (Empty(OkToRead))
         Signal(OkToWrite);
      else
         Signal(OkToRead);
      }
}
```

64



Bemerkungen

- Verwaltete Daten außerhalb des Monitors
- Reader

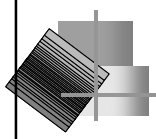
```
...
StartRead();
   Daten Lesen;
EndRead();
```

- Writer

```
...
StartWrite();
   Daten ändern;
EndWrite();
```

- Vorteile Synchronisation = Sprachelement gehen verloren
  - Vergessene oder Falsche Monitoraufrufe

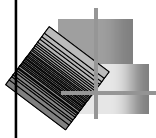
65



Path-Expressions

- Aufweichen des Monitorkonzepts
- Wechselseitiger Ausschluß manchmal zu restriktiv
  - siehe Reader-Writer-Problem
    - Eigentlichen Daten befinden sich außerhalb des Monitors
- Aussagen über erlaubte und verbotene Nebenläufigkeit
- Beispiele
  - `path 1:(Erzeuge;Verbrauche)`
    - Erzeuge gefolgt von Verbrauche (Strichpunkt)
    - Wechselseitiger Ausschluß
  - `path n:(1:(Erzeuge); 1:(Verbrauche))`
    - Erzeuge schließt sich wechselseitig aus
    - Verbrauche schließt sich wechselseitig aus
    - Erzeuge gefolgt von Verbrauche
    - Höchstens n beendete Erzeuge-Aufrufe mehr als Verbrauche

66



Weiterführende Literatur

- G. R. Andrews
  Concurrent Programming - Principles and Practice
  Benjamin/Cummings, 1991

67