

Deadlocks

```
down(semaphore S) {
    // Dijkstra P(S): probieren
    if (S.count == 0) {
        append(P, S.queue);
        block(P);
    } else
        S.count--;
    // decrement counter, continue
}

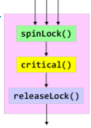
up(semaphore S) {
    // Dijkstra V(S): verhoken
    if (test(S.queue) != empty) {
        P = remove(S.queue);
        append(P, ready.queue);
    } else
        S.count++;
    // increment counter, continue
}
```

Hardwareunterstützung: Spin Lock

- **Implementation x86**
 - globale Variable
 - lock mit 0 initialisiert → offen
 - Lock schliessen

```
spinLock: MOV     AX, 1
try_lock: XCHG   AX, lock;
          CMP    AX, 0
          JNE    try_lock
          RET
```
 - Lock freigeben: unkritisch (atomar)

```
releaseLock: MOV    lock, 0
              RET
```



Hardwareunterstützung: x86

- **Maschinen-Instruktionen für TestAndSet**
 - "lesen und setzen" = "austauschen"
 - x86: XCHG reg, mem
 - ARM: SWAP (! LOAD/STORE-Architektur)
- **Implementation x86**
 - globale Variable: `gesperzt`, mit 0 initialisiert (=offen)

```
TestAndSet: MOV     AX, 1
              XCHG   AX, gesperzt; ← atomar
              XOR    AX, 1
              RET
```

XOR: invertiert Bit 0 von AX
 - Abfrage

```
while ( !TestAndSet() ) {};
```

Ressourcen-Klassen

- Preemptable (ohne Nebenwirkungen entziehbar) -> CPU, Hauptspeicher
- Nonpreemptable (Nebenwirkungen) -> Drucker, CD-Brenner

Grundsätzliche Probleme

- Starvation (Verhungern)
 - Prozess erhält keinen Zutritt zu Ressource
 - Ursache, z.B. unfaire Zuweisungspolicy: FILO (Stack)
 - Abhilfe: nur faire Policies verwenden, z.B. FIFO
- Deadlock (Verklemmung)
 - Prozesse warten gegenseitig auf Freigabe von Ressourcen
 - Die Prozesse und eventuell das gesamte System bleiben hängen

Voraussetzungen

- Mutual Exclusion
 - mindestens eine Ressource ist exklusiv reserviert
- Hold and wait
 - mindestens ein Task hat eine Ressource exklusiv reserviert und wartet auf weitere Ressourcen
- No preemption
 - reservierte Ressourcen können dem Task nicht entzogen werden (freiwillige Rückgabe nur, wenn Aufgabe gelöst)
- Circular wait
 - geschlossene "Kettennot be delayed infinitely" von Tasks existiert, in der jeder Prozess mindestens eine Ressource reserviert hat, die auch von einem Nachfolger in der Kette benötigt wird

1-3 sind Vorbedingungen, mit 4 ist es ein Deadlock.

die Circular Wait Bedingung kann nicht gelöst werden, wenn Bedingungen 1, 2 und 3 gegeben sind

Umgang mit Deadlocks

- Prevention: eines dieser verhindern: Mut. excl., hold & wait, no preemption, circular wait
 - ineffiziente "Serialisierung"
- Avoidance: neuer Zustand sicher
- Detection: zulassen, beim Auftreten Massnahmen treffen
 - OS muss Auftreten bemerken
 - DL auflösen und lauffähigen Zustand wiederherstellen
 - alle 40min oder bei tiefer CPU Auslastung überprüfen

Deadlock lösen

- alle beteiligten Prozesse stoppen
- checkpoint restore
- der reihe nach stoppen (bis gelöst)
- Ressourcen entziehen (bis gelöst)

zStrategie: wenigsten CPU / wenigsten Output / wenigsten Ressourcen / kleinste Prio / längste gesch. Rechenzeit
=> Am besten werden die Ressourcen in Klassen eingeteilt und je nach Klasse eine Strategie festgelegt.

Race Condition

Def: Gemeinsame Daten lesen und schreiben -> das Resultat hängt von der Ausführungsreihenfolge ab.
Forderungen:

- only one in the critical section
- for a finite time only
- makes no assumptions on speed/cores
- not be delayed infinitely
- requests entry and granted without delay
- halt in non critical section doesnt interfere with other processes

Software: Algorithmen: busy-wait / spinlock

Hardware: Maschineninstruktionen: atom. Inst: TestAndSet, CompareAndSwap, (Intterups off)

OS: Mutex, Semaphore

Kombiniert mit Programmiersprache: Monitore (Java)

Problem: Priority Inversion -> Solution: Priority Inheritance

-> high prio task wants lock on resource already locked by lower prio task

-> low prio task inherits higher prio so he can finish

Monitor: Klasse mit synchronized Methoden

Mutex: Zugriff auf kritische Ressource

Synchronisation: Reihenfolge der Verarbeitung (Barriere)

Paging

Buddy-Algo

solange Blockgrösse halbieren bis Block minimaler Grösse zur verfügung steht

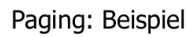
Paging

Aktuelle Betriebssysteme nur Paging, Segmentation wird auf logischer Ebene realisiert (Zugriffsrechte auf Pages Code, Stack, Data).

- logische Adressen -> Pages
- physikalische Adressen -> Frames
- pro Prozess eine Pagetable (lookup)

Pagesize = Framesize (1, 4, 8 KB)

- **Wie logisch Adresse in physikalische Adresse übersetzen ?**



-
- Figure 1 illustrates the structure of a document with four pages (A, B, C, D) and their corresponding page IDs. The pages are color-coded: Page A (blue), Page B (yellow), Page C (pink), and Page D (green). The page IDs are listed in a table on the right, showing the sequence of pages in the document. The table has two columns: 'Page ID' and 'Page Name'. The sequence is: 1 (Page A), 2 (Page B), 3 (Page C), 4 (Page D), 5 (Page A), 6 (Page B), 7 (Page C), 8 (Page D), 9 (Page A), 10 (Page B), 11 (Page C), 12 (Page D), 13 (Page A), 14 (Page B), 15 (Page C), 16 (Page D).

- Program, Data, Stack
- keine interne Fragmentierung, dafür externe
- pro Prozess eine Segment-Tabelle (OS) (lookup + Add)
 - + Speicherschutz +Programme unabh. verändern

- **Wie wird eine logische Adresse in eine physikalische Adresse übersetzt ?**

