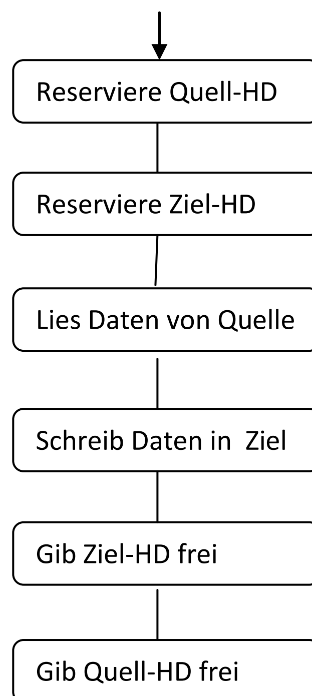
Probleme durch Multitasking

Leider bringt der Betrieb eines Rechners (oder einer SPS) in Multitasking nicht nur Vorteile, sondern auch Schwierigkeiten mit sich. So können fehlerfreie Programme sporadisch falsche Ergebnisse produzieren, Prozesse können sich sogar gegenseitig blockieren.

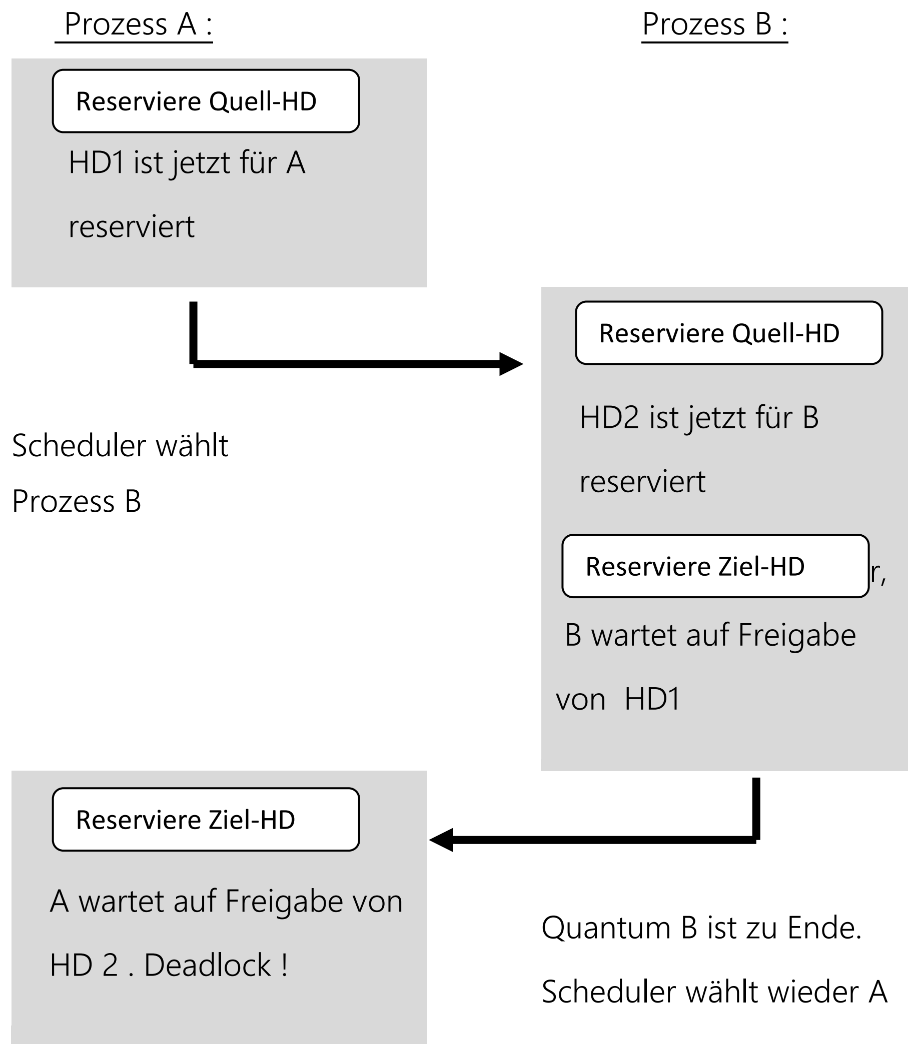
Deadlock

Prozesse A und B wollen Daten von einer Platte zur anderen kopieren. Prozess A will HD1->HD2, Prozess B andersrum. Die Medien werden vor dem Zugriff mit einem „lockbit“ reserviert.

Programmablauf beider Prozesse :



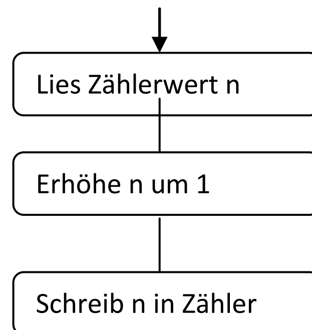
Im Multitasking kann jetzt folgendes passieren:



Die beiden Prozesse blockieren sich gegenseitig : deadlock !

Race condition

Zwei Prozesse in einer Industrieanlage erhöhen den Wert eines gemeinsamen Zählers (z.b. 2 Stanzen arbeiten in den gleichen Behälter). Der Zählvorgang ist in beiden so programmiert :



In dieser Anordnung passiert nun etwas äußerst unangenehmes:

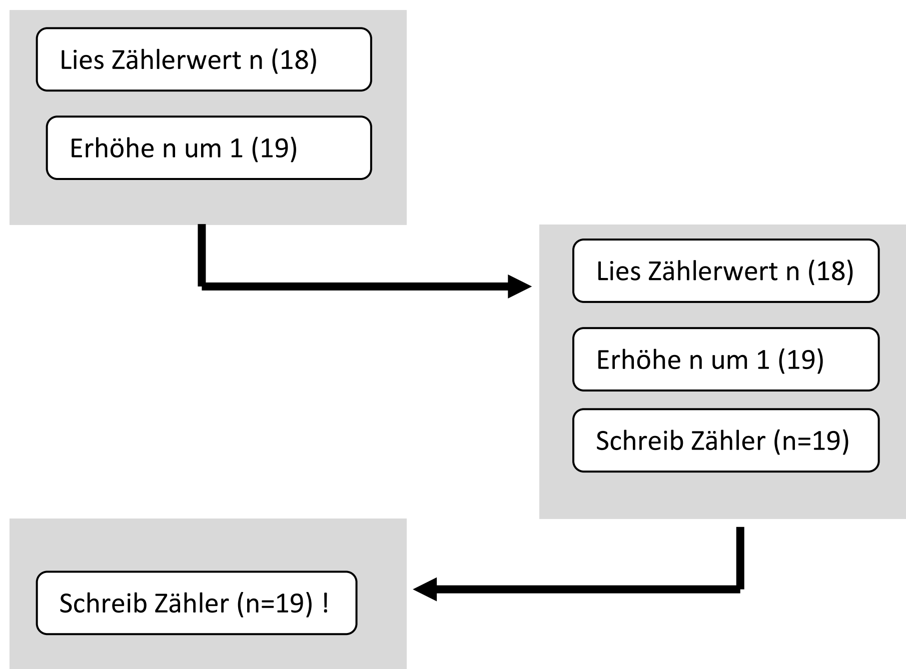
Die Prozesse bleiben nicht etwa stehen, sonden produzieren sporadisch, nicht reproduzierbar, Rechenfehler, obwohl sie offenbar fehlerfrei programmiert sind !

Der Zählerwert sei 18. Prozess A will 1 addieren, Prozess B auch :

Prozess A

Prozess

B



Also : 18 + 1 + 1 = 19   Oh !

Mehr dazu : Das Problem der 5 Philosophen (Djikstra, rund 1970)

http://de.wikipedia.org/wiki/Philosophenproblem

Ein Riesenproblem !
Das ganze schöne Multitasking wäre unbrauchbar, wenn man hier keine Lösung findet.

Erster Ansatz :
Man muß verhindern, daß eine Task mitten im Zugriff auf ein „Betriebsmittel" (also einen Speicher, eine Hardwarekomponente oder so..) unterbrochen wird. Man könnte z.b. einen Systembefehl bereitstellen, der den Scheduler anweist, die Task nicht zu unterbrechen.

Denken Sie nach, was halten Sie davon ... ?

Nun, das hatten wir schon. Damit wären wir bei kooperativem Multitasking. Wenn die Task „nicht unterbrechen" befiehlt, und dann in eine Endlosschleife geht (z.b. durch einen Programmfehler), steht das System !

Also anders !

Semaphore

Im Eisenbahnwesen kann es vorkommen, daß 2 Züge von 2 Seiten Seiten den selben eingleisigen Steckenabschnitt befahren wollen. Dazu hat man das sogenannte Semaphor erfunden. (https://de.wikipedia.org/wiki/Formsignal)

Dieses Signal wird gesetzt, sobald ein Zug den „kritischen Abschnitt" befährt, der aus der Weiche zum Tunnelabschnitt, dem Tunnel selber, sowie der Weiche nach dem Tunnelabschnitt besteht. Erst wenn er komplett durch ist, gibt das Semaphor den Streckenabschnitt wieder frei.

Angeblich wurde dafür ursprünglich ein Besen benutzt, der in einem Rohr steckte, das so an der Weiche montiert war, daß der Lokführer ihn rausziehen und an der anderen Weiche in ein gleiches Rohr wieder einstecken konnte. Ohne Besen keine Weiterfahrt !

Wichtig ist, daß sie den Kern der Angelegenheit verstehen :

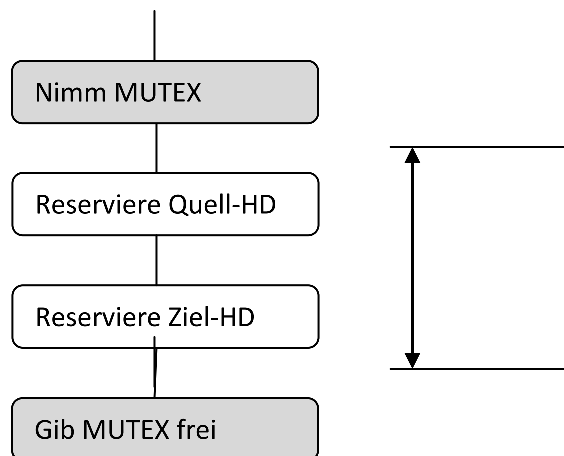
Die Prozesse dürfen nicht auf das Betriebssystem einwirken, sonst leidet die Stabilität.

Probleme wie deadlocks oder races (und davon gibt es noch eine Reihe anderer) müssen immer (!!) durch Kommunikation der Prozesse untereinander gelöst werden !

In der Informatik benutzt man dazu spezielle Variablen, die man Sempaphor, oder, wenn sie nur 2 Zustände haben können (0 oder 1), Mutex (von : mutual exclusion), nennt. Ein Semaphor mit z.b. 5 Zuständen kann mehrere (bis zu 5) Zugriffe auf ein Objekt zulassen, ein Mutex immer nur einen.

Wichtig ist hierbei, in einem Prozess, der mit anderen konkurrieren kann (siehe oben), den kritischen Abschnitt zu isolieren, und dann mit dem Mutex abzusichern.

Im Deadlock-Beispiel oben würde das so aussehen :



Wenn Prozess A den gemeinsamen MUTEX nimmt, kann Prozess B seine erste Platte nicht mehr reservieren -> kein Deadlock mehr möglich !

In der Praxis : Multithreading

Im Folgenden soll nun näher an der Praxis erklärt werden, dazu benutzen wir aber nicht Multitasking als Grundlage, sondern das einfacher zu handhabende Multithreading.

Threading ist die weitere Unterteilung von Tasks in noch kleinere Abschnitte. Diese können dann „innerhalb" einer Task quasi-parallel (bei Mehrprozessorsystemen u.U. auch wirklich parallel) ausgeführt werden. Alle Threads einer Task arbeiten im gleichen Speicherbereich (dem der Task), ansonsten verhalten sich im Wesentlichen wie Tasks.

Man benutzt das zum Beispiel, wenn man Teilkomponenten (Module) einer Anlage parallel unabhängig laufen lassen möchte, und das mit einem einzigen Programm gesteuert werden soll.

Beispiel in Visual Basic .net :

Anlegen eines Threads :

```
Dim thread1 As New System.Threading.Thread(AddressOf linear)
```

Programmcode :

```
Private Sub linear()
        ..
        .. irgendwelcher Code
        ..
End Sub
```

Starten des Threads

```
thread1.Start()
```

Zur Betrachtung von Semaphoren benutzen wir im Weiteren Threads, weil damit deren Funktion einfacher zu zeigen ist.
Für den Einsatz von Threads und die Nutzung dann nötiger Funktionen wie Semaphoeren und Mutexen stehen in allen Multitasking-Betriebssystemen Systemfunktionen bereit, die der Programmierer nutzen kann.

Beispiel :
An der Laboranlage „digitale Fabrik" wird das Tarnsportband von mehreren Modulen beauftragt und gesteuert. Die Steuerung der Odule geschieht im Multithreading, was für unsere Betrachtungen hier mit Multitasking indentisch ist. Dabei kann es zu race conditions kommen.

Definition eines Mutex in Microsoft Visual Basic .net :

```
  Dim band_mutex As New System.Threading.Mutex
```

Benutzung im Programm des Leitrechners :

```
'critical section band beginnt
'*******************************************************
   band_mutex.WaitOne()
   .
   .
   .
   .
   .
   band_mutex.ReleaseMutex()

'*******************************************************
'ende der critical section
```

Zu Beginn der critical section wird der Mutex gesetzt:

```
band_mutex.WaitOne()
```

Dieser Aufruf läuft als „atomare Operation“ im Betriebssystem ab, diese kann vom Scheduler nicht unterbrochen werden (also nicht selber wieder Probleme wie deadlocks erzeugen). Wenn Sie ans Scheduling zurückdenken, wird hier das Prinzip des „preremtive scheduling“ eigentlich mißachtet. Es geht aber nicht anders ...

Falls der Mutex schon gesetzt ist (von einem anderen Thread), geht der Thread hier „blocked“, wartet also. Falls nicht, setzt die Operation den Mutex.

Andere Threads, die den selben Mutex benutzen, können dann hier nicht weiter.

Am Ende der critical section gibt das Programm den Mutex mit wieder frei.

```
band_mutex.ReleaseMutex()
```