# Problem des schlafenden Frisörs

Dient zur Erklärung wie dead locks entstehen können. Tritt auf, wenn Prozesse abwechselnd ausgeführt werden und vom scheduler zur falschen Zeit unterbrochen werden:

Die Prozesse haben jeweils zwei Aufgaben:

1. Information sammeln
2. Reagieren auf die Information

Probleme entstehen, wenn der Prozess nach dem Sammeln der Information unterbrochen wird und erst später darauf reagieren kann und sich die Lage inzwischen verändert hat:

Frisör:

1. Information sammeln: Schaut ob eine Kunde da ist
2. Reaktion: Wenn kein Kunde da ist, schlafen gehen und warten bis er geweckt wird

Kunde:

1. Information sammeln: Schauen ob Frisör schläft
2. Reaktion: Wenn er der erste Kunde ist, den Frisör aufwecken sonst warten bis er dran kommt

Problematischer Ablauf, weil die Prozesse unterbrochen werden und der jeweils andere Prozess die CPU Ressourcen bekommt

Es ist gerade kein Kunde im System

1. Frisör schaut ob Kunde da ist: Keiner da – Prozess wird unterbrochen
2. Kunde schaut ob Frisör schläft: Frisör ist munter – Prozess wird unterbrochen
3. Frisör reagiert auf seine letzte Information (kein Kunde da) und geht schlafen – Prozess unterbrochen
4. Kunde reagiert auf seine letzte Information (Frisör ist munter) und wartet bis er dran kommt

Jetzt schläft der Frisör und der Kunde wartet bis er dran kommt –> dead lock  
Auch ein weiterer Kunde würde den Frisör nicht aufwecken weil er ja nicht der erste Kunde ist.  
Lösung atomare Prozesse: = Verbieten, dass Prozesse an solchen kritischen Stellen unterbrochen werden können. Das Holen von Information und die Reaktion darauf bilden einen kritischen Abschnitt, der nicht unterbrochen werden darf. Dies erreicht man durch Befehle, die den Ablauf des Programms entsprechend steuern und den Start und das Ende des kritischen Abschnitts signalisieren.

Oft verwendet man ein Kontrollelement (zB eine boolesche Variable) das gesetzt wird, um zu signalisieren, dass die Ressource gerade nicht frei ist. Die Befehle, die diese Kontrollelemente auf bestimmte Werte setzen, müssen natürlich selber ebenfalls atomare Befehle sein und gehören üblicherweise direkt zum Befehlssatz der CPUs.

Schlüsselwörter sind lock, mutex oder synchronize

Codeausschnitt vom Frisör nach Bearbeitung eines Kunden (erster thread):

lock();//Wartet bis kein lock mehr aktiv ist und setzt dann selber ein lock

if (kundeAnwesend) { //Info beschaffen

schneideHaare(); //Reaktion auf die beschaffte Information

}

else {

geheSchlafen();  
}

release(); //gibt den lock frei

Codeausschnitt vom Kunden beim Eintreten in den Frisörsalon (zweiter thread):

lock(); //Wartet bis kein lock mehr aktiv ist

if (frisoerSchlaeft) {

weckeFrisoer();

}

else {

warte();

}  
release();

Lock selber ist blockierend, dh der thread wartet bis ein anderer thread das release gemacht hat.   
Aktives/Passives Warten: Beim Warten auf Fortführung des Prozesses gibt es zwei Vorgangsweisen, aktives und passives Warten.

Beim **aktiven Warten** (engl busy waiting oder spinning oder polling) überprüft der wartende Prozess ständig, ob er weitermachen kann, dh welchen Wert die Kontrollvariable hat:

solange (Bedingung b nicht erfüllt) {tu nix}

Die Prozessorbelastung ist also höher. Üblicherweise stellt das OS Befehle zur Verfügung, damit wenigstens das Warten keine CPU Ressourcen verwendet (man nennt das dann slow busy waiting oder oder lazy polling)

solange (Bedingung b nicht erfüllt) {

warte für einige Zeit;

}  
Problem ist, das man nicht gleich mitbekommt, wenn die Bedingung erfüllt ist, oder man in kürzeren Zeitabschnitten nachschaut, dann hat man wieder den Ressourcenverbrauch

Beim **passiven Warten** wird die Fortführung von einem Kontrollprozess (des OS) übernommen, der den wartenden Prozess weiterführt, wenn die gewünschte Ressource freigegeben wird.

# Verfahren zum Schutz gemeinsamer Resourcen

* Mutex (gegenseitiger Ausschluss) : Gruppe von Verfahren zum Lösen der Problematik der kritischen Abschnitte.
* Semaphor/Monitor: Einem Abschnitt, Objekt, Funktion wird ein Kontrollelement zugeordnet. Dieses Kontrollelement speichert, ob die Komponente gerade in Verwendung ist oder nicht. Ist sie in Verwendung, muss das Programm warten, bis sie wieder frei ist.
* Lock: Ähnlich den Semaphoren, unterstützt aber Read- und Write-Locks. Somit können zwei Prozesse, die nur Daten lesen wollen, gleichzeitig lesen. Erst bei einem Write-Lock wird der Zugriff für andere gesperrt. Findet bei Datenbanken oder File IO oft Verwendung.

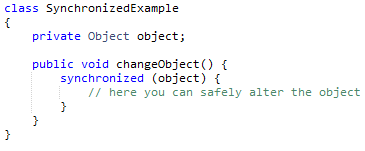
# Implementierung

## Locks:

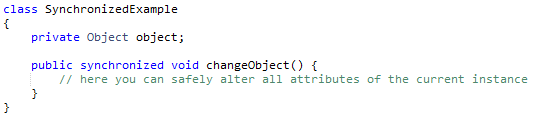
Gibt es in den meisten Programmiersprachen, das Konzept ist immer gleich, die Befehle heißen anders:

zB Linux   
pthread\_mutex\_lock(&mutex);  
pthread\_mutex\_unlock(&mutex);

Beispiel zur Implementierung von kritischen Abschnitten in Java. Dazu verwendet man das synchronized-Schlüsselwort. Es kann sowohl auf Abschnitte, als auch auf ganze Methoden angewendet werden.



Das Schlüsselwort setzt einen Lock auf das als Parameter angegebene Objekt (hier object). Will ein weiterer Thread auf dasselbe object einen Lock setzen, so muss er warten bis das Objekt frei ist.



Das Schlüsselwort in dieser Verwendung setzt einen Lock auf die gesamte betroffene Instanz. Ruft ein anderer Thread also eine andere Methode auf, die mit synchronized gekennzeichnet ist, so muss er warten bis der andere Thread den Lock wieder freigibt.

## Sempahoren

Semaphoren sind Siganlgeber, die anzeigen ob eine Resource frei ist oder nicht. (Sempahore ist ein Eisenbahnsignal, das bei eingleisigen Strecken gesetzt wurde, wenn ein Zug im Abschnitt war, damit kein entgegengesetzter Zug in den eingleisigen Abschnitt einfährt)

Es gibt binäre Semaphoren (diese entsprechen den normalen locks) und zählende Semaphoren, die eine gewissse Menge an Ressourcen verwalten können. Immer wenn eine Ressource verbraucht wurde, wird die semaphore heruntergezählt, erst bei 0 entsteht ein Wartezyklus für den Prozess, der noch eine Ressource benötigen würde.   
Wenn eine Ressource frei wird die Semaphore um 1 hinaufgezählt.

Wo braucht man so etwas: bei beschränkten Ressourcen zB wenn Druckjobs aus einer Druckerwarteschlange an einen Drucker gesendet werden sollen, der nur eine gewisse Datenmenge zwischenspeichern kann. Jeder anstehende Druckjob verringert den freien Platz, jeder abgearbeitete erhöht ihn wieder. Wenn kein Platz mehr da ist, muss der nächste Job warten.

### Semaphore unter Linux

Folgende Funktionen werden benötigt:

* semget: Erzeugt ein Semaphore oder holt ein bereits bestehendes. Rückgabewert ist die Id des Semaphores. Semaphoren werden über einen key erzeugt bzw verwendet. Wer den key kennt, kann die Semaphore benutzen, also auch verschiedene Tasks am gleichen Rechner (nicht nur threads).
* semop: Reduziert oder erhöht den Wert des Semaphores um den angegebenen Wert. Ist nicht mehr genug vorhanden, so wartet das Programm entweder oder semop() gibt einen Fehler zurück (Verhalten ist über die Parameter spezifizierbar).
* semctl: Wird für verschiedene Konfigurationen der Semaphoren verwendet. Z.B. zum Setzen des initialen Wertes des Semaphores.

### Semaphore unter Windows

Folgende Funktionen werden benötigt:

* CreateSemaphore: Zum Erstellen und Öffnen eines Semaphores.
* WaitForSingleObject: Zum Anfordern eines binären Semaphores. Die Funktion wartet solange, bis entweder ein freies Semaphore gefunden wurde oder das übergebene Timeout ausläuft. Rückgabewert gibt an, was passiert ist.
* WaitForMultipleObjects. Wartet bis etwas frei ist und zählt herunter und führt das Programm weiter aus, sobald etwas frei ist
* ReleaseSemaphore: Zum Freigeben einer Semaphore.

# Pipes

Pipes (First In First Out Buffer) liefern eine einfache Möglichkeit für den geregelten Datenaustausch zwischen Prozessen und sind standardmäßig blockierend (das kann man aber aufheben)

Beispiel: pipe von Prozess1 zu Prozess2, damit P1 Infos zu P2 senden kann

Prozess1 schreibt in die pipe und macht erst weiter wenn P2 gelesen hat.  
P2 wartet beim Lesebefehl bis etwas in der pipe steht und macht erst dann weiter wenn P! etwas in die pipe geschrieben hat.  
Das Warten geschieht auch umgekehrt dh Der schreibende Prozess wartet bis der lesende Prozess die Infos ausgelesen hat. Lesen leert die pipe.

Durch das gegenseitige Blockieren ist sichergestellt, dass die Daten beim Gegenüber ankommen bzw das sie gelesen wurden.

Pipes gibt es unnamed (für verwandte Prozesse oder threads) und named für getrennte Tasks,

Wer bei named pipes den Namen kennt, kann sie verwenden, wobei der erste Prozess die pipe erzeugt (create Befehl mit entsprechenden Rechten) und die restlichen die pipe öffnen (ust wie das Schreiben und Lesen von Files, aber mit entsprechendem Warten bis was drinnen steht

Befehle: mkfifo, pipe