Kapitel 6: Nebenläufigkeit, Synchronisation, Wechselseitiger Ausschluss

- Motivation und Einführung
- Begriffe und ein Beispielproblem
- Klassische Wettbewerbsprobleme
- Gestaltungsparameter für das Koordinieren von Threads
- Synchronisationsmechanismen
    - Signalisieren
    - Wechselseitiger Ausschluss
        - Kritische Abschnitte
        - HW unterstützte Lösungen
        - Lösungen mit Kernunterstützung
        - Lösungen auf Anwenderebene

6.1 Motivation und Einführung

Mitunter konkurrieren Prozesse und Tasks bzw. deren Threads um Betriebsmittel (Ressourcen), die sie jeweils nur *exklusiv* nutzen können. Beispielsweise ist es wenig sinnvoll, den gleichen Drucker gleichzeitig von mehreren Prozessen zur Ergebnisausgabe zu benutzen. (Hinweis: Die meisten heutigen Desktopsysteme umgehen dieses konkrete Wettbewerbsproblem dadurch, dass sie nur noch dem Spooler erlauben, Druckaufträge an den Druckertreiber zu schicken. Die eigentlichen Applikationen geben ihre Druckergebnisse in eine Zwischendatei aus.) (Man spricht in diesem Zusammenhang auch davon, dass der Spooler, Druckertreiber und der Drucker aus der Sicht der nutzenden Applikationen ein logisches Gerät darstellt. Die Wettbewerbsprobleme um den physischen Drucker sind somit vor dem Nutzer verborgen.)

Manchmal gibt es auch in zusammenarbeitenden Threads *Programmabschnitte*, die so *kritisch* sind, dass man sie besser *seriell* ausführt, da es ansonsten zu so genannten *Wettlaufsituationen* (*race conditions*) kommen kann, die im allgemeinen zu inkonsistenten Daten bzw. zu fehlerhaften Programmverhalten führen.

Beide geschilderte Situationen sind im Prinzip vergleichbar und können somit auch mit ähnlichen Mechanismen gelöst werden.

Synchronisationsprobleme treten in Systemen auf zwei wohl unterscheidbaren Ebenen auf, die man nicht verwechseln sollte, da deren Lösungen sich deutlich von einander unterscheiden.

- Anwenderebene (*application layer*)
- Kernebene (*kernel layer*)

Lösungen, die z.B. auf der Kernebene benutzt werden, sind i.d.R. für Applikationen nicht erlaubt bzw. können dort sogar zu einem Systemstillstand (*live lock*) oder zu einer Verklemmung (*dead lock*) führen.

Als einführendes Beispiel wollen wir ein Bankkonto betrachten, auf das neben der Besitzerin bzw. dem Besitzer auch deren Freund bzw. dessen Freundin Zugriff hat.

Wir können uns abstrakt in Pseudocode folgende Abbuchungsfunktion definieren und prüfen, in wie weit deren Implementierung zu Problemen führen kann.

```
int withdraw(account, amount){
   balance = get_balance(account);
   balance -= amount;
   put_balance(account, balance);
   return balance;
}
```

Stellen wir uns vor, dass sowohl Kontoinhaber(in) als auch der/die andere Zugriffsberechtigte gleichzeitig an zwei Bankschaltern/-automaten Geld abheben. Im günstigen Fall für die Kontoinhaber könnte Folgendes passieren:

```
balance = get balance(account);                    bankautomat 1
balance -= amount;                                 bankautomat 1
balance = get balance(account);                    bankautomat 2
balance -= amount;                                 bankautomat 2
put balance(account, balance);                     bankautomat 2
put balance(account, balance);                     bankautomat 1
```

Tatsächlich wird zweimal abgebucht, aber nur der abgebuchte Betrag am Bankautomat 1 wird tatsächlich vermerkt, in diesem Fall hätte die Bank das Nachsehen. Da dies leider in der Realität nie der Fall ist, scheinen die Banker da einen Riegel vorschieben zu können. Nun die Lösung ist relativ simpel, man verhindert gleichzeitiges Abheben, d.h. sobald eine der beiden Abbuchungen angefangen hat, muss dafür gesorgt werden, dass die zweite Abbuchung so lange verzögert wird, bis die erste Abbuchung vollständig durchgeführt worden ist. Dieses so genannte Transaktionskonzept muss jedoch auf den Banksystemen auch noch implementiert werden, genau darum wird es in den nächsten Abschnitten gehen.

6.2 Konkurrenzprobleme auf Anwenderebene

In der Literatur wird auf Anwenderebene eine Reihe von Problemen mit konkurrierenden Prozessen oder Threads behandelt, von denen die folgenden am häufigsten zitiert werden:

- Produzenten-/Konsumentenproblem
- Leser-/Schreiberproblem
- Philosophenproblem

6.2.1 Produzenten-/Konsumentenproblem

Ausgangspunkt ist ein Puffer (eine Art Zwischenlager), der in der Praxis nur eine begrenzte Anzahl von Produkten zwischenlagern kann. Den allgemeinen Fall kann man sich am besten dadurch veranschaulichen, dass man sich ein Warenlager vorstellt, zu dem in unregelmäßigen Abständen Lieferwagen angerollt kommen, um ihre Produkte abzuliefern. Auf der anderen Seite kommen in ebenfalls unregelmäßigen Abständen Kunden an, die Produkte mit nach Hause nehmen wollen. Zur Vereinfachung können wir annehmen, dass es nur eine Sorte von Produkten gibt, die angeliefert bzw. abgeholt werden soll.

Auf den ersten Blick gibt es offensichtlich zwei Situationen, die man bei einem Lösungsversuch des Problems berücksichtigen sollte:

1. Man kann in eine volles Zwischenlager kein Produkt anliefern.
2. Man kann aus einem leeren Zwischenlager offensichtlich kein Produkt entnehmen.

Daneben gibt es aber noch ein weiteres Problem, das des gleichzeitigen Zugriffs auf einen Lagerplatz, zwei oder mehr Kunden können nicht das gleiche Produkt nehmen bzw. zwei oder mehr Produzenten können ihr Produkt nicht auf den gleichen Lagerplatz abstellen.

6.2.2 Leser-/Schreiberproblem

Bei diesem Problem geht es im Wesentlichen darum, dass man den gleichzeitigen Zugriff auf ein gemeinsames Dokument vernünftig regelt. Solange nur Leser auf ein Dokument zugreifen, kann man diesen unbegrenzt Zutritt erlauben, d.h. es ist unerheblich ob gleichzeitig 100000 oder nur zwei Leser gleichzeitig lesend auf das Dokument zugreifen. Sobald jedoch gleichzeitig nur noch ein Schreiber hinzukommt, kann es Konsistenzprobleme geben, da dann die Leser evtl. teilweise alte und neue Daten lesen könnten.

Prinzipielle Lösungen des Leser-/Schreiberproblems werden dadurch unterschieden, welcher Seite man jeweils Vorrang einräumt.

6.2.2.1 Leservorrang

Bei dieser Lösung geht man davon aus, dass die Leser wichtiger sind, als jeder schreibende Vorgang. Die Lösung sieht demnach -grob skizziert- wie folgt aus:

- Solange zumindest noch ein Leser aktiv ist, müssen die Schreiber blockiert werden. Der letzte Leser aktiviert dann wiederum einen Schreiber (evtl. den am längsten wartenden), der dann *exklusiv* seine Schreibarbeit auf dem Dokument verrichten darf, d.h. insbesondere dass kein weiterer Leser oder Schreiber zugelassen wird. Sollte anschließend noch kein weiterer Leser einen Lesewunsch geäußert haben, dann wird mit dem nächsten wartenden Schreiber weitergemacht (sofern vorhanden), ansonsten werden sofort wieder alle wartenden Leser aufgeweckt.

6.2.2.2 Schreibervorrang

Bei dieser Lösung wird davon ausgegangen, dass Schreibvorgänge wichtiger als Lesevorgänge sind. Die Lösung sieht demnach -grob skizziert- wie folgt aus:

- Sobald ein Schreiber seinen Schreibwunsch geäußert hat und das Dokument aktuell gerade von n>=1 Lesern gelesen wird, werden keine weiteren Leser oder Schreiber mehr zugelassen, bis der am längsten wartende Schreiber nach Beendigung des letzten aktuellen Lesevorgangs seine exklusive schreibende Tätigkeit aufnehmen kann. Nach seinem erfolgreichen Schreiben wird zunächst geprüft, ob weitere Schreiber darauf warten, das Dokument zu modifizieren. Wenn dies der Fall ist, werden die Schreiber den wartenden Lesern vorgezogen. Ansonsten werden wiederum alle wartenden Leser auf einmal auf das Dokument zum parallelen Lesen zugelassen.

6.2.2.3 Wechselnder Vorrang

Bei dieser Lösung kann man entweder periodisch oder in Abhängigkeit von der Warteschlangenlänge der Leser bzw. der Schreiber mal den Lesern, mal den Schreibern Vorrang einräumen.

Problem: Überlegen Sie sich Anwenderfälle für jede der drei Vorrangsvarianten.

6.2.3 Philosophenproblem

Beim Philosophenproblem geht es darum, dass sich 5 Philosophen erfolgreich um jeweils zwei Gabeln bemühen müssen, damit sie aus der Spaghettischüssel in der Mitte des runden Tisches essen können. Insgesamt sind jedoch nur 5 Gabeln da, jeweils eine zwischen zwei Philosophen. Abwechselnd denken die Philosophen, abwechselnd müssen sie aber auch mal essen und zwar möglichst in einer Reihenfolge, dass keiner von ihnen verhungert.

Jeweils zwei benachbarte Philosophen konkurrieren dabei um die zwischen ihnen liegende Gabel, ohne die sie nicht essen können. Das auf den ersten Blick so simple Zugriffsprotokoll, jeder Philosoph nehme zuerst immer die linke Gabel und dann die rechte Gabel ist leider eine totale Scheinlösung, da es hierbei eben auch passieren kann, dass jeder Philosoph die linke Gabel gleichzeitig nimmt und dann vergeblich darauf wartet, dass er auch noch die rechte Gabel dazubekommt, die aber von seinem Nachbarn zur Rechten nicht wieder hergegeben wird. In dieser speziellen Version verhungern alle Philosophen zusammen, in dieser Situation ist das sprichwörtliche Verhungern zugleich auch eine Verklemmung (*deadlock*).

Ein anderes simples Zugriffsprotokoll erlaubt nur dem Philosophen mit dem "Essring" (*token*), dass er sich aktiv um Gabeln bemühen darf. Nachdem er gegessen hat, gibt er den Essring an seinen rechten Nachbarn weiter. Hierdurch wird zwar garantiert, dass jeder Philosoph mal zum Essen kommt, aber ein Philosoph, der schnell denkt und schnell Hunger bekommt, mag den Hungertod schon vor den Augen haben, wenn links von ihm ein oder mehrere eher lang denkende und lang essende Philosophenkollegen Platz genommen haben. Außerdem nützt dieses Protokoll nicht aus, dass zwei Philosophen gleichzeitig essen könnten, da sie ja nur 4 von 5 vorhandenen Gabeln benötigen.

6.3 Synchronisation

Es treten bei zusammenarbeitenden Threads gelegentlich Situationen auf, in denen man einen bestimmten Programmabschnitt in Thread T_i erst bearbeiten sollte, wenn ein anderer Programmabschnitt in Thread T_j bereits vollständig bearbeitet ist. Diese Problematik wird im Wesentlichen durch einfache Signalisierung, Synchronisation oder Barrieresynchronisation gelöst. Eine andere Form der Synchronisation benötigt man, wenn bestimmte Programmabschnitte (*critical sections*) so kritisch sind, dass sie exklusiv ausgeführt werden müssen, d.h. im Klartext es darf keine gleichzeitige Ausführung von *in Zusammenhang stehenden kritischen Abschnitten* geben. (Man beachte jedoch, dass kritische Programmabschnitte sehr wohl gleichzeitig ausgeführt werden dürfen, solange sie nicht im Zusammenhang stehen, also nicht auf gemeinsamen Ressourcen oder Daten operieren.)

6.3.1 Signalisierung

Zur Problembeschreibung kann man sich eine Anwendung mit zwei Threads vorstellen, bei denen es notwendig ist, dass Programmabschnitt a_1 von Thread T_1 vollständig bearbeitet sein muss, ehe mit dem Programmabschnitt b_2 in T_2 begonnen werden darf.

Eine mögliche Lösung könnte wie folgt aussehen:

```
1:1 signal s; /* type 1:1 signal object */
{Thread 1}                      {Thread 2}
.                               .
.                               .
.                               .
{ section a1                    { section b1
   ... }                           ... }
signal(s)          ------->     wait(s)
{ section a2                    { section b2
   ... }                           ... }
.                               .
.                               .
.                               .
```

Wie und wo implementiert man geschickterweise solch ein 1:1_Signalobjekt? Die Notation 1:1 bedeutet hierbei, dass nur ein signalisierender Thread und nur ein auf das Eintreffen des Signals wartender Thread im Spiel sind.

Drei prinzipielle Lösungsklassen für dieses Synchronisationsproblem können unterschieden werden:

1. Reine Softwarelösungen auf Anwenderebene
2. Synchronisationsobjekt an der Kernschnittstelle
3. Hardwareunterstützung durch spezielle atomare Befehle

Implementierungen auf Anwenderebene haben entweder den Nachteil, dass sie sich auf aktives Warten (*busy waiting*) abstützen oder *exklusive atomare* Kernoperationen wie `sleep()` und `wakeup()` benötigen.

Man kann ein Signalisierungsobjekt so entwerfen, dass Signale verloren gehen dürfen bzw. dass kein Signalverlust auftritt. Sofern man im letzteren Fall gestattet, dass auf Anwenderebene `signal()` asynchron verwendet werden darf, muss man sich darüber im klaren sein, dass hierdurch eine Attacke vom Typ: "Denial of Service" ermöglicht wird.

Sofern man dazu übergeht, dass man sowohl die `signal()` als auch die `wait()` Funktion synchron auslegt, spricht man eigentlich erst von Synchronisation im engeren Sinne. Manche Systeme bieten deshalb an der Kernschnittstelle ein Funktion `synchronize()` an, die im Übrigen auf beliebig viele Threads

angewendet werden kann, was insbesondere die nebenläufige Programmierung vieler numerischer Programme erleichtert.

Man kann die Signalisierungsobjekte auch in anderer Hinsicht verallgemeinern, in dem man sowohl auf Signalisiererseite als auch auf der Empfangseite mehrere Threads zulässt, wobei man unterschiedliche Semantiken einführen kann.

Betrachten wir zunächst den Fall eines m:1 Signalobjekts. Eine Semantikvariante setzt voraus, dass der Empfängerthread erst weitermachen darf, wenn jeder der m Signalisierer ein Signal am m:1 Signalobjekt abgelegt hat. Die andere Semantikvariante setzt lediglich voraus, dass der Empfängerthread solange warten muss, bis irgendein Signalisierer ein Signal abgelegt hat.

Analog dazu kann man auch auf der Empfängerseite zwei unterscheidbare Semantiken einrichten.

Signale kann man entweder durch ein Bit oder durch eine Ganzzahl implementieren. Im zweiten Fall ist es besonders einfach, mehrere Signal zwischen zu puffern, in dem pro Signal einfach die Ganzzahl inkrementiert (ein hinreichend großer Wertebereich vorausgesetzt) und pro durchgeführter Warteoperation diese Ganzzahl wieder dekrementiert wird.

6.3.2 Signalisierung mittels Semaphor

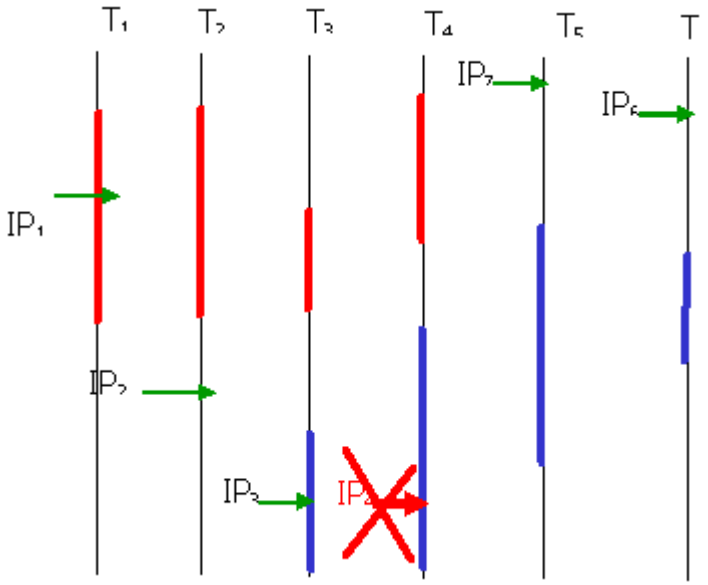
Dijkstra führte das Semaphorkonzept ein, das auch heute noch intensiv benutzt wird und das wohl die erste durchdachte Synchronisationslösung insgesamt war. Ein Semaphorobjekt bietet als Schnittstellenfunktionen die beiden Operationen `p()` und `v()` an, die sich wechselseitig ausschließen und atomar sind. Wenn man ein Semaphor zum Signalisieren verwenden will, dann wird es mit 0 initialisiert, was soviel bedeuten soll, dass anfangs an dem Semaphor kein Signal ansteht.

Ruft nun ein signalisierender Thread irgendwann eine v-Operation auf, dann wird mit deren Hilfe die Semaphorvariable inkrementiert. Ein Thread auf der Empfangsseite des Semaphors wird irgendwann einmal eine p-Operation aufrufen, mit deren Hilfe ein zuvor geschicktes Signal verbraucht wird, d.h. das Semaphor wird dekrementiert.

Sollte der Thread auf der Empfangsseite zuerst die p-Operationen aufrufen, bevor eine v-Operation aufgerufen worden ist, dann wird der aufrufende Prozess am Semaphor blockiert, d.h. das Semaphorobjekt enthält eine objekt-spezifische Warteschlange. Kommt also der Signalisierer mit seiner v-Operation später, dann wird nicht nur inkrementiert, sondern es muss auch überprüft werden, ob auf dieses Signal schon ein Thread in der objekt-spezifischen Warteschlange am Semaphor wartet. Sofern dies der Fall ist, wird dieser Thread daraufhin wieder deblockiert, so dass er dann bei Gelegenheit vom Scheduler wieder auf eine(die) CPU zugeordnet werden kann. Wird in einem System das Semaphorkonzept angeboten, dann ist es leicht möglich, dass bei eng zusammenarbeitenden Applikationen oder auch Systemaufgaben gleichzeitig hunderte von Semaphorobjekten im Spiel sind, jedes dieser Semaphorobjekte hat somit seine eigenen Warteschlange, in der sich wartende Threads mittels der p() Operation einreihen können.

6.3.3 Wechselseitiger Ausschluss an kritischen Abschnitten

Insbesondere beim Zugriff auf gemeinsame Daten müssen Threads daran gehindert werden, gleichzeitig zu modifizieren, da ansonsten die Konsistenz der Daten in Frage gestellt ist. Programmabschnitte, in denen auf globale Daten zugegriffen wird, nennt man auch "kritische Abschnitte". Betrachten wir zunächst den Fall einer multi-threaded Anwendung aus sechs Kernel-Level Threads, in denen in den farbig hervorgehobenen Programmabschnitten der folgenden Skizze auf globale Daten zugegriffen wird.



Die rot eingefärbten Programmabschnitte der Threads T_1 bis T_4 beziehen sich möglicherweise auf den Zugriff eines Arrays, während sich die blauen Abschnitte auf den Zugriff auf ein Bitfeld beziehen mögen. Von den angedeuteten Befehlszeigern ist somit ausschließlich die Kombination IP_3 und IP_4 nicht zugelassen, da sich ansonsten zwei Threads in konkurrierenden kritischen Programmabschnitten befinden würden. Man beachte, dass ein kritischer Abschnitt den gleichen Code oder verschiedenen Code enthalten kann.

Damit Lösungen hinsichtlich Effektivität und Effizienz besser miteinander verglichen werden können, sei folgendes Protokoll im Zusammenhang mit kritischen Abschnitten vorausgesetzt:

- Enter_Critical_Section()
- Critical Section
- Exit_Citical_Section()
- Non Critical or Remainder Section

Wie wir dies auch schon bei der Signalisierung gesehen haben, haben sich auch im Umfeld von kritischen Abschnitten eine Reihe von Verfahren auf verschiedenen Implementierungsebenen herauskristallisiert, die hinsichtlich Effektivität und Effizienz deutliche Unterschiede aufweisen. In jedem Fall müssen sie aber, um als gültige Systemarchitekturlösungen akzeptiert zu werden, folgende vier Bedingungen erfüllen:

1. *Exklusivität* (*exclusiveness*), d.h. es darf jeweils nur ein Threads im zusammenhängenden kritischen Abschnitt sein
2. *Portabilität* (*portability*), d.h. es dürfen keine Annahmen über Anzahl der CPUs bzw. des Schedulingverfahrens gemacht werden
3. *Fortschritt* (*progress*), d.h. kein Thread außerhalb des kritischen Abschnitts darf auf einen eintrittswilligen Thread Einfluss haben
4. *Begrenzte Wartezeit* (*Bounded waiting*), d.h. kein Thread darf unbegrenzt lange in der Funktion Enter_Critical_Section warten müssen.

Diese vier Anforderungen an jede Lösung für den wechselseitigen Ausschluss von kritischen Abschnitten sind notwendig. Daneben gibt es nun noch weitere wünschenswerte Eigenschaften, die wir von einer eleganten Lösung erwarten.

1. Leistungsfähigkeit (*performance*), die Relation zwischen dem zu treibenden Aufwand beim Eintritt in den kritischen Abschnitt und bei dessen Verlassen sollte dem im Abschnitt zu tätigenden Aktivitäten angemessen sein.
2. Fairness heißt, dass jeder vor dem kritischen Abschnitt wartende Thread eine faire Chance hat, den kritischen Abschnitt auch zu betreten.

3. Einfachheit (*simplicity*), die verwendeten Protokolle im Umgang mit kritischen Abschnitten sollten leicht anzuwenden und einzuhalten sein.

6.3.3.1 Erster Ansatz: Lockvariable im Hauptspeicher

Ein Lock sei ein Objekt, das die beiden Schnittstellenfunktionen `acquire()` und `release()` anbietet, und das mit dem Wert o (= Lock ist frei) initialisiert wird. Zwischen dem `acquire` und dem `release` Aufruf, kann der Thread seine Aktivität im kritischen Abschnitt voll entfalten.

Wie wird aber beispielsweise garantiert, dass diese Funktionen selbst exklusiv sind, also beispielsweise nicht gleichzeitig auf zwei Prozessoren aufgerufen werden können? Wir wollen dieses Problem auf spätere Abschnitte verschieben und zunächst die Lösung unseres Kontoproblems durch Verwendung eines simplen Lock-Objekts analysieren:

```
int withdraw(account, amount){
acquire(lock);
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
return balance;
}
```

Hierzu wollen wir folgende Implementierung der beiden Funktionen acquire und release annehmen:

```
struct lock{
int held=0;
}
void acquire(lock){
while (lock->held);
lock->held=1;
}
void release(lock){
lock->held=0;
}
```

Die obige Implementierung enthält das Problem, dass in beiden Operationen auf die Variable `lock.held` zugegriffen wird, also kann es wiederum zu Wettlaufsituationen kommen. Außerdem ist diese Lösung ineffizient, weil sie wiederum aktives Warten innerhalb von `acquire` verwendet.

6.3.3.2 Exklusive Operationen

Im Zusammenhang mit dem einfachen Lock aus 6.3.3.1 müssen wir eine Möglichkeit ebnen, wie wir zu exklusiven und atomaren Operationen vom Typ `acquire` und `release` kommen können. Hierzu müssen wir auf spezielle Fähigkeiten der Hardware aufbauen, und dabei wird es einen weiteren größeren Unterschied zwischen Einprozessor- und Mehrprozessorlösungen geben.

6.3.3.2.1 Exklusivität auf Einprozessorsystemen

Auf einem Einprozessor rechnet maximal ein KLT zur gleichen Zeit, entweder im Benutzer- oder im Kernmodus. Will man also einen kritischen Abschnitt exklusiv gestalten, so könnte man hierzu schlichtweg asynchrone Unterbrechungen ausschalten, also mittels eines Befehls der Art: `disable_interrupt`. Am Ende des kritischen Abschnitts könnte man mittels `enable_interrupt` die Exklusivität dann wieder aufheben.

Wirksamkeits- und Machbarkeitsanalyse:

Zumindest `disable_interrupt` ist auf den meisten Architekturen ein *privilegierter Befehl*, d.h. eine Anwendbarkeit auf Anwenderebene ist ausgeschlossen. Statt Exklusivität einzuleiten erhielten wir eine vorbelegte Ausnahmebehandlung, die auch nicht durch eine eigene vom Anwender programmierte Ausnahmebehandlung ersetzt werden kann.

Es wäre auch nicht sehr ratsam, diese Möglichkeit Anwendern zur Verfügung zu stellen, da hierdurch u.U. unbeabsichtigte Seiteneffekte auftreten können, u.a. wird dann ein Zeitscheibenmechanismus nicht mehr wirksam, ferner kann die verzögerte Abarbeitung von Unterbrechungen zu Datenverlusten bei einer Netzwerkkarte führen.

Somit kann `disable_interrupt` allenfalls auf Kernebene angewendet werden, und genau da liegt auch dessen Verwendungszweck. Jeder Systemaufruf kann nach dem hardwareabhängigen Moduswechsel in den Kern mittels `disable_interrupt` zu einer exklusiv und atomar auszuführenden Kernoperation verwandelt werden.

Zusammenfassung:

Zur Etablierung von *atomaren exklusiven Kernoperationen*, die i.d.R. kurz sind, kann ein Kernprogrammierer auch folgende i.d.R. privilegierten Befehle benutzen:

`disable_interrupt` für Enter_Critical_Section und
`enable_interrupt` für Exit_Critical_Section

Man beachte hierbei, dass die Wirksamkeit am größten und damit die Seiteneffekte am geringsten sind, wenn jede Kernoperation kurz ist. Leider sind nicht einmal alle Mikrokernoperationen besonders kurz, um wie viel mehr werden Probleme bei den Kernoperationen monolithischer Kerne (à la Unix, Linux, oder WinXP) auftreten.

Was tun sprach Zeus?

Nun diese Problematik führte zu den unterbrechbaren oder verdrängbaren Kernoperationen der so genannten verdrängbaren Kerne (*preemptible kernels*). Hierauf wollen wir am Ende der Veranstaltung nochmals eingehen.

6.3.3.2.2 Exklusivität auf Mehrprozessorsystemen

Leider ist diese relativ simple Lösung mittels `disable_interrupt` bzw. `enable_interrupt` auf Mehrprozessorsystemen überhaupt nicht effektiv. Dies liegt daran, dass eine Unterbrechungssperre nur lokal auf dem jeweiligen Prozessor wirksam ist, davon aber die anderen Prozessoren überhaupt nicht tangiert werden. Zum einen können auf den anderen Prozessoren nach wie vor zu beliebigen Zeitpunkten, also auch parallel zu einer "exklusiv zu machenden Kernoperation" Unterbrechungen auftreten, außerdem dürfen auf den anderen Prozessoren auch parallel dazu die gleiche Kernoperation ausgeführt werden.

Auf Mehrprozessorsystemen benötigen wir Hardwareunterstützung, die sowohl den Busverkehr, den Zugriff auf den Hauptspeicher zwangsserialisiert, dies wird durch die so genannten "atomaren Befehle" erreicht, zu denen eine Reihe von unterschiedlichen Ausprägungen realisiert worden sind, von denen der TAS (TestAndSet) sehr bekannt ist.

```
boolean test_and_set( boolean *flag){
   boolean old = *flag;
   *flag = true;
   return old;
}
```

Dessen Implementierung ist in der obigen Programmskizze verdeutlicht, man beachte, dass hierbei der Parameter `flag` sowohl Ein- als auch Ausgabeparameter ist. Initialisiert werden könnte diese globale

"Kernvaraibale" mit `false`, d.h. momentan ist noch kein KLT im kritischen Abschnitt. Der TAS Befehl sperrt jeglichen Zugriff auf den Bus und auf den Speicherbereich, in dem `flag` lokalisiert ist. Solange `true` nicht in allen Caches des Prozessors (und im Hauptspeicher steht, je nach Cachewrite policy), solange kann kein anderer Prozessor zeitgleich den TAS Befehl auf `flag` ausführen.

Man könnte somit im Kern wie folgt einen Spinlock anbieten:

```
struct lock_SMP{
int held = false;
}
void acquire_SMP(lock){
while (test_and_set(&lock->held));
}
void release_SMP(lock){
lock->held=false;
}
```

Allerdings könnten sich in SMPs mit einigen Cachespeichern bei der obigen Realisierung des Spinlocks weitere unangenehme Seiteneffekte ergeben, so dass man zu einer effizienteren Implementierung eines skalierbaren SMP-fähigen Spinlocks kommen muss (siehe Übungen).

6.3.3.3 Semaphor für den wechselseitigen Ausschluss

Ein Semaphor kann ebenfalls zum wechselseitigen Ausschluss benutzt werden, wobei er anders als bei der Signalisierung initialisiert werden muss. Will man ein Zählsemaphor für den wechselseitigen Ausschluss benutzen, dann muss die Semaphorvariable mit 1 initialisiert werden. Bei den Zählsemaphoren kann man zwischen schwachen und starken Semaphoren wählen, bei den letzteren wird im Falle einer v-Operation stets der Thread deblockiert, der darauf am längsten gewartet hat.

Im Folgenden ist eine Skizze abgebildet, wie man auf einem Einprozessorsystem ein Semaphorobjekt als Kernobjekt einrichten könnte:

```
p(sema S)
  begin
  DisableInterupt
    s.count++
    if (s.count < 0){
      insert T(s.QWT)
      BLOCK(S)
    }
    else
  EnableInterrupt
  end
```

```
v(sema S)
  begin
  DisableInterupt
    s.count--
    if (s.count <= 0){
      remove T(s.QWT)
      UNBLOCK(S)
    }

  EnableInterrupt
  end
```

Verständnisfrage:

Wann wird denn dann `enable_interrupt` wieder aufgerufen, wenn innerhalb von p() in der Funktion BLOCK() auf einen anderen KLT umgeschaltet werden muss?

Obwohl das Semaphorkonzept in seiner zweifachen Semantik recht mächtig erscheint, so hat es doch auch gewisse Mängel, die zum einen darin begründet liegen, dass das Programmieren mit Semaphoren einen höchst disziplinierten Programmierstil erfordert, da es keinerlei Restriktionen zum Einhalten der Semaphorsemantik gibt, insbesondere fehlende p oder v Operationen in den diversen Programmzweigen einer komplexeren Anwendung können zu unvorhersehbaren Fehlern führen. Darüber hinaus erfordert eine orthogonale Implementierung von Semaphorobjekten erheblich mehr Aufwand, als in der obigen Implementierung beim Einprozessorsystem dargestellt.

Im folgenden Programmbeispiel wird eine solche orthogonale Implementierung skizziert:

```
module binary semaphore
  export P2,V2 import BLOCK, UNBLOCK
  type binsem = record
    owner: thread = nil                    {semaphore owner}
    QWT: list of Threads = empty           {queue for waiting threads}
    end
  P2(BS: binsem):
    if (BS.owner == nil)
      BS.owner = myself                    {close CS for current use}
    else {
      insert (QWT, myself)
      do BLOCK(myself) until owner = myself od }
    fi .
  V2(BS: binsem):
    if (BS.QWT == empty) then
      BS.owner = nil                       {open CS for future use}
    else {
      BS.owner = take first(QWT)
      UNBLOCK(BS.owner)}
    fi .
end binary semaphore
```

Durch die Verwendung des Objektattributs `owner` kann deutlich unterschieden werden, wer der Initiator eines Deblockiervorgangs (mittels `UNBLOCK`) ist, so dass somit `BLOCK()` und `UNBLOCK()` zu reinen Hilfsfunktionen für das Scheduling degenerieren können.

Folgerung: Zwei neue Regeln für den Entwurf einer Systemarchitektur

Regel 2: Entwerfe unabhängige Dinge unabhängig

Regel 3: Vermeide Seiteneffekte

6.3.3.4 Monitor für den wechselseitigen Ausschluss

Monitore sind Objekte auf Anwenderebene, deren Schnittstellenfunktionen exklusiv und atomar sind. Die Sprache Java bietet hierzu die so genannten "synchronisierten Objekte" an.

Monitore sind aus der Sicht von Anwenderprogrammierern das erheblich übersichtlichere und fehlerfreiere Konzept im Vergleich zum Semaphorkonzept und erfordern i.d.R. erheblich weniger Detailkenntnisse.

Beispiel für das begrenzte Pufferproblem:

```
monitor module bounded buffer
  export fetch, deposit
  import CondSignal,CondWait
  buffer object = record
    array buffer[1..n] of datatype
    head: integer = 1
    tail: integer = 1
    count: integer = 0
    not full: cond = true
    not empty: cond = false
  end
```

```
  procedure deposit(b:buffer object, d:datatype)
    begin
      while (b.count == n) do CondWait(b.not full) {only block thread}
      b.buffer[b.tail] = d
      b.tail = b.tail mod n +1
      b.count = b.count + 1
      CondSignal(b.not empty)
    end

  procedure fetch(b:buffer object, result:datatype)
    begin
      while (b.count == 0) do CondWait(b.not empty) {only block thread}
      result = b.buffer[b.head]
      b.head = b.head mod n +1
      b.count = b.count - 1
      CondSignal(b.not full)
    end
end monitor module
```

Dessen Anwendung könnte wie folgt aussehen:

```
ProducerI:
repeat
  produce v;
  deposit(v);
forever

ConsumerI:
repeat
  fetch(v);
  consume v;
forever
```

6.3.3.5 Reine Softwarelösungen auf Anwenderebene

Peterson's Lösung stellte historisch besehen erst die zweite effektive Lösung dar, allerdings lässt deren Effizienz erheblich zu wünschen übrig, da u.U. viel Zeit wiederum im aktiven Warten verbracht wird. Peterson's Lösung wie auch die erste Lösung von Dekker haben beide den Nachteil, dass sie nur für 2 konkurrierende Threads geeignet sind. Eine Verallgemeinerung der Lösung für n konkurrierende Threads stellt der so genannte Bäckereialgorithmus dar.