Ankündigungen 20.11.2008

- Terminplanung:
  - Freitag 21.11.2008, **9:00**:
    Besprechung von Übungsblatt 8 (Implementierung von User Level Threads)
  - Freitag 21.11.2008, 10:15:
    Übung 9 (Programmierung mit Semaphoren)
  - Donnerstag, 27.11.2008, 15:30:
    Nebenläufiges Programmieren in Java
  - Freitag 28.11.2008, **9:00**: Überraschung
  - Freitag 28.11.2008, 10:15: Übungsblatt 10 (Nebenläufiges Programmieren in Java)
  - Donnerstag, 4.12.2008: Abschlussvorlesung
  - Freitag, 5.12.2008, 10:15: Fragestunde
- Prüfungstermine:
  - Freitag, 19.12.2008, 14:00 Uhr in A5 B144
  - Basiskurs schreibt 66 Minuten (3 Anmeldungen)
  - Rest schreibt 100 Minuten (31 Anmeldungen)

Implementierung

- Monitore sind Konstrukte einer Programmiersprache
  - Sie werden vom Compiler in einen Code auf niedrigerer Abstraktionsebene übersetzt
  - Wir betrachten beispielhaft Implementierungen mit Semaphoren (Vorschlag von Hoare)
- Jeder Monitor `M` wird in eigenes Modul/eigene Klasse übersetzt
  - Monitoreintritt und Monitoraustritt werden über ein Semaphor `M_Mutex` geregelt
  - Jeder Monitor hat sein eigenes Mutex-Semaphor
  - Beim Eintritt in einer Monitorprozedur: `P(M_Mutex)`
    - Man sagt hier: "Ein Thread bewirbt sich um den Monitor"
  - Beim Austritt: `V(M_Mutex)`
    - Man sagt hier: "Der Monitor wird freigegeben"
  - Initialisierung: `M_Mutex = 1`

Übersetzung

- Compiler transformiert Monitor in folgenden Code
  - (blau = veränderter Code)

```
MODULE M
  Datendeklaration // gemeinsame Daten
  Semaphore M_Mutex
  ENTRY Funktionsname1 (Parameter){
    P(M_Mutex);
    Prozedurkörper
    V(M_Mutex);
  }
  ...
  INIT {
    M_Mutex = 1;
    Initialisierungscode
  }
END
```

Condition-Variablen

- Zur einfachen Formulierung bedingter kritischer Abschnitte gehören zum Monitorkonzept sogenannte Condition-Variablen
  - Spezielle Variablen, die Bedingungen über den Monitordaten repräsentieren
- Auf den Condition-Variablen kann man sich blockieren
  - Aufruf: `Conditionvariable.WAIT();`
  - Bedeutung: Warte an dieser Stelle, bis die Bedingung der Conditionvariable gilt
- Man kann andere Threads aufwecken, die an einer Conditionvariablen warten
  - Aufruf: `Conditionvariable.SIGNAL();`
  - Bedeutung: abhängig von der Signal-Semantik (es gibt drei verschiedene)
- Man kann testen, wieviele Threads warten
  - Aufruf: `Conditionsvariable.STATUS();` liefert `int`

WAIT und SIGNAL

- Zu einer Condition gehört immer eine einfache FIFO-Warteschlange für Threads
  - Ähnlich wie bei Semaphoren: Kann für Echtzeitzwecke auch prioritätsbasiert sein
- Wirkung von `WAIT()`
  - Der ausführende Thread blockiert sich in die Warteschlange der Conditionvariable
  - Falls noch ein Thread den Zugang in den Monitor verlangt hat, deblockiere einen solchen Thread
  - Ansonsten wird der Monitor freigegeben
- Wirkung von `SIGNAL()` (Variante 1)
  - Deblockiere einen Thread, falls einer an der Warteschlange der Conditionvariable wartet
  - Stelle sicher, dass nach Beendigung der `SIGNAL`-Operation höchstens ein Thread im Monitor rechnet

Beispiel

- Bedingter kritischer Abschnitt:

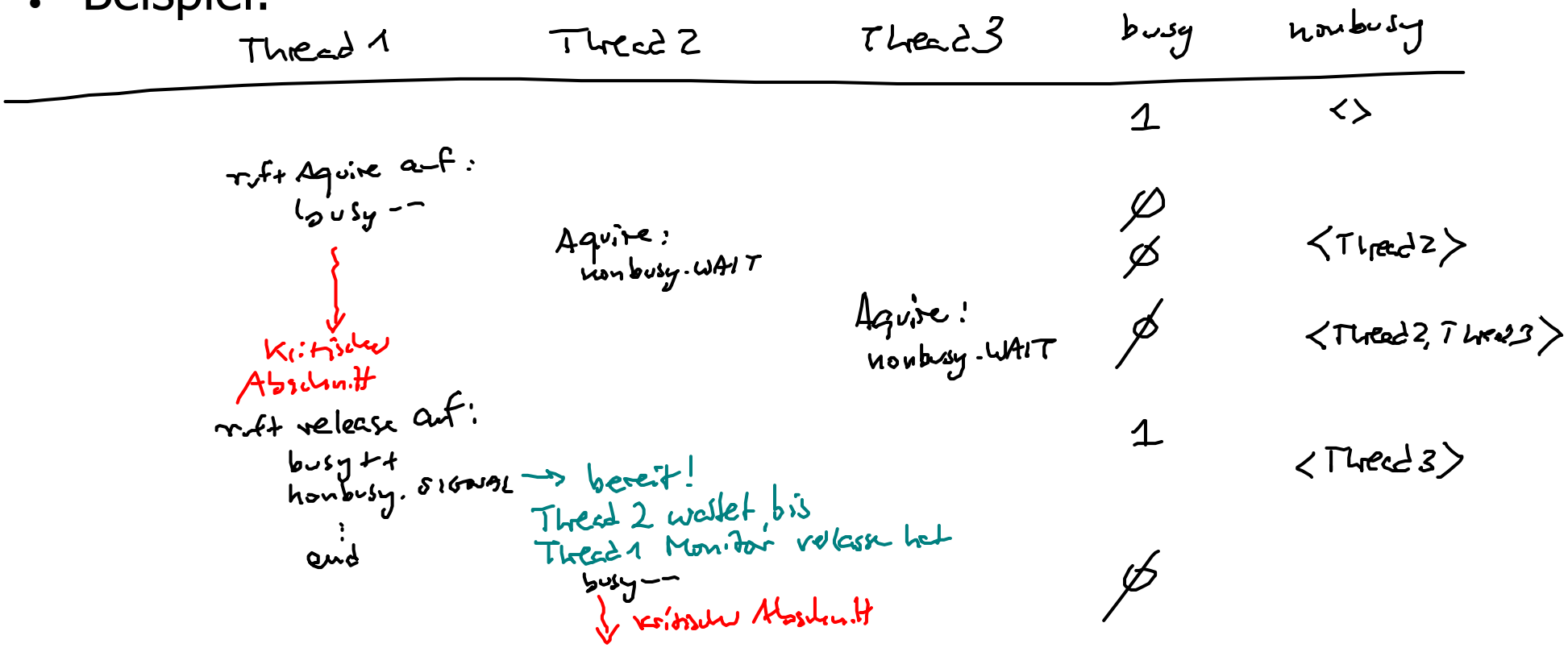
```
MONITOR SingleResource
  int busy; // >0 bedeutet frei, <1 bedeutet belegt
  Condition nonbusy;
  ENTRY Aquire {
    while (busy < 1) nonbusy.WAIT();
    busy--;
  }
  ENTRY Release {
    busy++;
    nonbusy.SIGNAL();
  }
  INIT {
    busy = 1;
  }
}
```

  - Kapseln der kritischen Abschnitte:

```
...
SingleResource.Aquire()
// kritischer Abschnitt
SingleResource.Release()
...
```

Beispielablauf

- Wir nehmen an, dass `SIGNAL` maximal einen Thread aus der Condition-Warteschlange deblockiert
  - Dieser Prozess muss sich allerdings neu um den Eintritt in den Monitor bewerben
- Beispiel:



Erläuterungen

- Beispielablauf:
  - Thread1: `Aquire`, geht in kritischen Abschnitt, jetzt ist `busy = 0`
  - Thread2: `Aquire`, `busy` = 0 < 1 also `nonbusy.WAIT`, Thread2 blockiert in `nonbusy`-Warteschlange
  - Thread3: `Aquire`, `busy` = 0 < 1 also `nonbusy.WAIT`, Thread3 blockiert in `nonbusy`-Warteschlange
  - Thread1: `Release`, `busy++`, `nonbusy.SIGNAL`, ein Thread (Thread2) wird aufgeweckt, jetzt ist `busy` = 1, Thread1 hat Monitor verlassen
  - Thread2: `busy--`, geht in kritischen Abschnitt, jetzt ist `busy` = 0
  - ... Thread3 bleibt blockiert, solange Thread2 kein Release macht
- Bemerkung: `Aquire` und `Release` implementieren die Semaphor-Operationen `P` und `V`
  - Damit gezeigt: Alles, was man mit Semaphoren kann, kann man auch mit Monitoren

Implementierung WAIT/SIGNAL

- Bei einem `SIGNAL` wird ein Thread "im Monitor" aufgeweckt
  - Man muss dafür sorgen, dass nicht zwei Threads gleichzeitig im Monitor rechnen
- Idee:
  - Der aufweckende Thread darf noch weiterrechnen bis er den Monitor verlässt
  - Der aufgeweckte Thread wird vorrangig (vor "neu eintretenden") Threads behandelt
- Umsetzung:
  - `SIGNAL` führt zur Neueinreihung in eine bevorzugte Warteschlange `urgent` für wichtige Threads
  - Wenn ein Thread den Monitor verlässt, wird zunächst in der `urgent`-Warteschlange nachgeschaut
  - Nur wenn `urgent` leer, wird an der Eingangswarteschlange des Monitors nachgeschaut

Übersetzung

```
MODULE M
  Datendeklaration // gemeinsame Daten
  Semaphore M_Mutex, M_Urgent;
  int M_UrgentCount;

  ENTRY Funktionsname1 (Parameter){
    P(M_Mutex);
    Prozedurkörper
    if (M_UrgentCount > 0) V(M_Urgent) else V(M_Mutex);
  }
  ...
  INIT {
    M_Mutex = 1; M_Urgent = 0; M_UrgentCount = 0;
    Initialisierungscode
  }
END
```

Übersetzung (Forts.)

- Jede Condition-Variable `cond` wird bei der Übersetzung durch ein Semaphor `condsem` und einen Zähler `condsemcount` ersetzt:

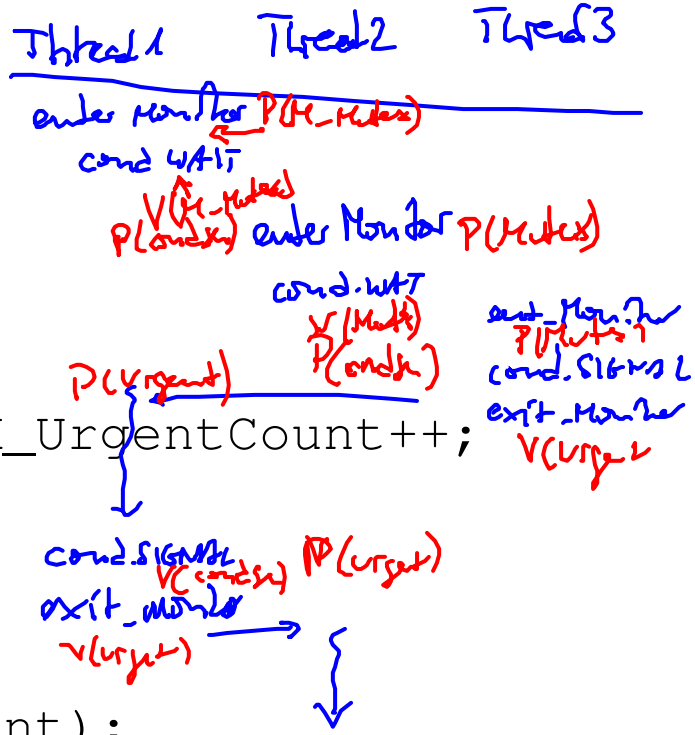
```
Semaphore condsem = 0;
int condsemcount = 0;
```

- `cond.SIGNAL` wird übersetzt in:

```
if (condsemcount > 0) {
  V(condsem); condsemcount--; M_UrgentCount++;
}
```

- `cond.WAIT` wird zu:

```
condsemcount++;
if (M_UrgentCount > 0) V(M_Urgent);
else V(M_Mutex);
P(condsem);
P(M_Urgent);
M_UrgentCount--;
```



Mächtigkeit von Monitoren

- Lösungen mit Monitoren werden meist einfacher
- Beispiel: Verwaltung gleichartiger Betriebsmittel

```
MONITOR DiscPool (int Anzahl)
  enum DiscStatus[N] {frei, belegt};
  int busy; // Zahl belegter Laufwerke
  Condition nonbusy;
  ENTRY int GetDisc() {
    int i = 1;
    while (busy == N) nonbusy.WAIT;
    while (DiscStatus[i] == belegt) i++;
    DiscStatus[i] = belegt;
    busy++;
    return(i);
  }
  ENTRY PutDisc(int ActDisc) {
    DiscStatus[ActDisc] = frei;
    busy--;
    nonbusy.SIGNAL;
  }
  INIT { N = Anzahl; DiscStatus = frei; busy = 0; }
}
```

Leser-Schreiber-Problem

```
MONITOR ReadWrite {
  int Readcount, Writeflag;
  Condition OkToRead, OkToWrite;
  ENTRY StartRead() {
    Readcount++;
    while (Writeflag == 1) OkToRead.WAIT;
    OkToRead.SIGNAL;
  }
  ENTRY EndRead() {
    Readcount--;
    if (Readcount == 0) OkToWrite.SIGNAL;
  }
  ENTRY StartWrite() {
    while ((Readcount  > 0) or (Writeflag == 1)) OkToWrite.WAIT;
    Writeflag = 1;
  }
  ENTRY EndWrite() {
    Writeflag = 0;
    if (Readcount > 0) OkToRead.SIGNAL else OkToWrite.SIGNAL;
  }
  INIT { Readcount = 0; Writeflag = 0; }
}
```

Bemerkungen

- Lösung stammt von Hoare
- Es gibt vier Monitorprozeduren:
  - `StartRead`, `EndRead` klammern die Leseabschnitte der Leser
  - `StartWrite`, `EndWrite` klammern die Schreibabschnitte der Schreiber
- `Readcount` zählt die Anzahl der bereiten oder lesenden Leser
- `Writeflag` vermerkt einen potentiellen Schreiber
- Zwei Conditions:
  - `OkToRead` : Warteschlange für Leser
  - `OkToWrite` : Warteschlange für Schreiber

Zweites Leser-Schreiber-Problem

```
MONITOR ReadWrite {
  int Readcount, Writecount, Writeflag;
  Condition OkToRead, OkToWrite;
  ENTRY StartRead() {
    while (Writecount > 0) OkToRead.WAIT;
    Readcount++;
    OkToRead.SIGNAL;
  }
  ENTRY EndRead() {
    Readcount--;
    if (Readcount == 0) OkToWrite.SIGNAL;
  }
  ENTRY StartWrite() {
    Writecount++;
    while ((Readcount  > 0) or (Writeflag == 1)) OkToWrite.WAIT;
    Writeflag = 1;
  }
  ENTRY EndWrite() {
    Writecount--;
    Writeflag = 0;
    if (Writecount > 0) OkToWrite.SIGNAL else OkToRead.SIGNAL;
  }
  INIT { Readcount = 0; Writecount = 0; Writeflag = 0; }
}
```

SIGNAL-Varianten

- Variante 1 (bisher betrachtet):
  - Einer der wartenden Threads wird aufgeweckt
  - Falls mehrere Threads deblockiert werden sollen: Verketten der SIGNAL-Anweisungen
    - Der zuletzt Aufgeweckte weckt den nächsten auf usw.
- Variante 2:
  - Alle wartenden Threads werden aufgeweckt
  - Besonders vorteilhaft, wenn in der Regel durch ein SIGNAL mehrere Threads deblockiert werden
- Variante 3 (Hoare):
  - Wie Variante 1, nur geht der Besitz des Monitors auf den signalisierten Thread über
  - Vorteil: signalisierter Thread findet seine Wartebedingung vor
  - Signalisierender Thead muss sich erneut um den Besitz des Monitors bewerben

Implementierung Variante 2

- Zur Erinnerung: Variante 1

```
if (condsemcount > 0) {
  V(condsem); condsemcount--; M_UrgentCount++;
}
```

- Variante 2:

```
while (condsemcount > 0) {
  V(condsem); condsemcount--; M_UrgentCount++;
}
```

- Implementierung von `WAIT` ändert sich nicht

Implementierung Variante 3

- `cond.WAIT` wird zu:

```
condsemcount++;
if (M_UrgentCount > 0) V(M_Urgent);
else V(M_Mutex);  // Thread verläßt den Monitor
P(condsem); // Thread betritt erneut den Monitor
condsemcount--;
```

- `cond.SIGNAL` wird zu:

```
M_UrgentCount++;
if (condsemcount > 0) {
  V(condsem); P(M_Urgent);
}
M_UrgentCount--;
```

- (Einziger) Vorteil von Variante 3:
  - Statt `while (Bedingung) cond.WAIT`
    kann man nun schreiben `if (Bedingung) cond.WAIT`
- Viele Thread-Wechsel, kann aber noch optimieren