Hochschule für Technik und Wirtschaft HTW Chur
Studiengang Telekommunikation/Elektrotechnik

Seminararbeit

Prozessorbefehle zur Synchronisation

Christopher Walther

März,2010

Diese Arbeit wurde im Rahmen des Bachelor-Studienganges Telekommunikation/Elektrotechnik an der Hochschule für Technik und Wirtschaft HTW Chur erstellt.

Studierender

Christopher Walther
Schulstrasse 65
7302 Landquart

Dozent

Martin Studer, Dipl. Inf.-Ing. ETH HTW
Chur Ringstrasse/Pulvermühlestrasse 57
7004 Chur

Inhaltsverzeichnis

Inhaltsverzeichnis 3
Abbildungsverzeichnis 3
1 Fragestellung 4
2 Einführung in Synchronisation zwischen Prozessoren 5
2.1 Kritische Abschnitte 5
2.2 Performance-Einbussen durch Synchronisation 5
3 Was sind Atomare Operationen? 6
3.1 Erklärung 6
4 Synchronisation in Java mit Semaphoren 8
4.1 Einleitung 8
4.2 Begriff Semaphor 8
4.3 Beispiel einer Semaphor-Implementation in Java 9
5 Der TAS-Befehl (Test-And-Set beim M68000) 11
6 Anhang 14
6.1 Semaphoren in Delphi [10] 14
7 Quellenverzeichnis 17

Abbildungsverzeichnis

Figure 1: Test and Set beim M68000 11
Figure 2: Programmablauf des TAS-Commands 13

Tabellenverzeichnis

Tabelle 1: Bitbelegung des <ea>-Operanden des TAS-Befehls [6] 11
Tabelle 2: Condition Code Register des Motorola 68000 12

1 Fragestellung

Für die Synchronisation Prozessen und/oder Threads kennen Sie beispielsweise das Konstrukte der Monitore in Java. Einfachere Konstrukte sind Semaphoren, auch diese sind in Java vorhanden.

Stellen Sie sich folgende Fragen:

- Wie lassen sich Semaphoren auf Mikroprozessoren implementieren?
- Braucht es für die Implementation spezielle Befehle?
- Was versteht man unter dem Begriff der atomaren Operation?

2 Einführung in Synchronisation zwischen Prozessoren

Prozessoren und Threads haben grundsätzlich ihren eigenen Speicherbereich. Dort können die Threads über den Speicherzugriff frei verfügen und kommen sich nicht mit anderen Threads oder Prozessen in die Quere. Mehrere Threads können aber auch ohne Probleme gemeinsame Daten lesen. Problematisch wird es erst, wenn mehrere Prozesse auf einen gemeinsamen Speicher schreiben wollen. Dies kann eine Speicheradresse im Hauptspeicher, eine Text-Datei auf der Festplatte aber auch ein Netzwerkdrucker, auf den mehrere User zugreifen können, sein.

2.1 Kritische Abschnitte

In dieser Arbeit wird immer wieder der Begriff „Kritischer Abschnitt“ verwendet. Darunter verstehen wir beispielsweise einen Programmblock in der Software, welcher nicht unterbrochen werden darf, da sonst inkonsistente oder auch korrupte Daten entstehen könnten. Auch eine instabile Software mit Laufzeitfehlern kann die Folge sein.

Es gibt Prozessoren, welche „atomare Befehle“ zur Verfügung stellen. Diese Befehle können einfach formuliert eine Ressource prüfen und als „belegt“ markieren, damit kein anderer Zugriff erfolgen kann. [3]

☞ (Siehe dazu Kapitel **3. Was sind Atomare Operationen?)**

2.2 Performance-Einbussen durch Synchronisation

Beim Zugriff eines Prozesses oder Threads auf eine gemeinsame verwendete Ressource, muss zunächst überprüft werden ob die Ressource bereits verwendet wird. Anschliessend muss die Ressource für alle anderen Prozesse „verschlossen“ werden. Alle Threads die nun auf die gemeinsame Ressource zugreifen möchten, werden nun von der CPU in einer Warteschlange verwaltet. Der Zugriff auf die gemeinsame Ressource und das Verwalten der Warteschlange benötigt CPU Ressourcen. [1]

3 Was sind Atomare Operationen?

Unter den Begriff „Atomare Operationen“ versteht man einzelne Anweisungen, welche aus einzelnen Prozessorbefehlen bestehen, welche nicht durch externe Einflüsse unterbrochen werden können. [2]

3.1 Erklärung

Folgende Beispiele erklären um welche Befehle es sich um „Atomare Anweisungen“ es sich handelt.

Javacode:

```
10 int temp;

20 temp++;
```

Javacode kompiliert Pseudo-ASM-Code:

```
100 Shift accu

200 Inc + 1

300 Shift back
```

Der Befehl der Inkrementierung wird auf Assemblerstufe auf mehrere ASM-Befehle aufgeteilt. Ein anderer Prozess kann also die Anweisung **`20 temp++`** zwischen Befehl `100` und `200` unterbrechen. Falls der unterbrechende Befehl auf die gleichen Ressourcen zugreift, können die Daten verfälscht werden. [4]

Schlussfolgerung: Die Standardanweisung **`temp++`** ist **nicht** atomar, da sie an einer unbekannten Stelle von anderen Quellen unterbrochen werden kann.

Als Beispiel für einen „Atomaren Befehl“ wird das Test-And-Set-Verfahren verwendet.

Anwendung in Pseudo-ASM- code:

```
10  ;Uncritical section
20  ...
30  TAS      testbyte    ;     testet das testbyte ob bereits andere
40                       ;     Zugriffe auf den Speicherbereich
50                       ;     bereits erfolgen
60
70
80  ...
```

Der **TAS**-Befehl überprüft an der in **testbyte** angegebenen Adresse ob die Ressource bereits von einem anderen Prozess verwendet wird. Falls dies der Fall ist, kann beispielsweise gewartet werden, bis die Ressource freigegeben ist.

Falls aber die Ressource verfügbar ist, wird im **testbyte** signalisiert das nun die Ressource belegt ist.

Dieser „test and set“-Ablauf kann von keinem anderen Prozess unterbrochen werden. Somit ist dieser **TAS**-Befehl atomar.

☞ Weitere detailiertere Angaben zum Befehl und zur Anwendung, sind im Kapitel **5. Der TAS-Befehl (Test-And-Set beim M68000)** zu finden.

4 Synchronisation in Java mit Semaphoren

4.1 Einleitung

Neben den bereits bekannten Verfahren mit **`synchronised`**-Anweisungen gibt es auch in Java eine Variante Semaphoren zu implementieren. [5]

4.2 Begriff Semaphor

4.2.1 Binäre Semaphoren

Bei binären Semaphoren kann **`value`** nur den Wert 0 oder 1 (bereits besetzt oder Verfügbar) annehmen. Falls ein Thread auf einen bereits verwendeten Abschnitt zugreifen will, wird der Zugriff verwehrt. Der Thread kann nun…

- …auf die Freigabe des Semaphors warten
- …eine bestimmte Zeit auf die Freigabe des Semaphors warten (Timeout)
- …weiter im Programmablauf.

4.2.2 Zählende Semaphoren

Bei zählenden Semaphoren kann **`value`** einen wert zwischen 0 und n annehmen. Der Wert n entspricht den verfügbaren Ressourcen innerhalb des kritischen Abschnitts. Folgendes Beispiel soll diesen Fall verdeutlichen:

In einem Büro-LAN stehen 3 Netzwerkdrucker zur Verfügung welcher von mehreren Benutzern verwendet werden können. Es können 3 Benutzer gleichzeitig auf einen Drucker zugreifen. Der User löst also mit dem „Drucken“-Befehl eine Anfrage an die Semaphore aus und falls nur 1 oder 2 Drucker belegt sind, wird der kritische Abschnitt trotzdem ausgeführt und die Druckanfrage an Drucker 3 gesendet. Dieses Handling wird innerhalb des kritischen Abschnitts ausgeführt.

4.3 Beispiel einer Semaphor-Implementation in Java

Folgender Code zeigt wie Semaphoren implementiert werden können. Java bietet im Package **`java.util.concurrent`** die Klasse **`semaphore`** an. Diese Implementation ist atomar und sollte grundsätzlich verwendet werden. Dieses Beispiel ist eine einfachere Implementation die lediglich die Funktionsweise erklären soll.

```
10 public class Semaphore {
20  private int value;
30  public Semaphore(int initial) {
40    value = initial;
50  }

60  public synchronized void release() {
70    ++value;
80    notify();
90  }

100 public synchronized void acquire() throws InterruptedException {
110    while (value== 0) {
120      wait();
130   }
140    -value;
150  }
160}
```

Konstruktor `Semaphore (int intital)`

Der Konstruktor legt die Semaphore im Speicher an, und erzeugt mittels dem Parameter **`initial`** eine binäre oder zählende Semaphore. **`inital`** entspricht den verfügbaren Ressourcen im kritischen Abschnitt.

Methode `void aquire()`

Bevor eine Thread auf eine gemeinsame Ressource zugreifen kann, muss der Thread mittels der **`aquire()`**-Funktion anfordern. Wird nun die **`aquire`**-Methode aufgerufen, wird bei binären Semaphoren zunächst geprüft ob die Ressource verfügbar ist. Bei zählenden Semaphoren wird überprüft ob noch freie Ressourcen zur Verfügung stehen. Falls der Thread nicht auf die Ressource zugreifen darf, kann der Thread schlafen gelegt werden, damit dieser keine CPU-Leistung benötigt. Falls **`value`** einen Wert grösser 0 besitzt darf der Thread zugreifen und befinden sich nun im kritischen Abschnitt

Methode `void release()`

Wenn der Thread den kritischen Abschnitt abgeschlossen hat, muss nun die Ressource wieder freigegeben werden. Dies wird mittels der **`release`** erreicht. Beim Aufruf wird **`value`** um 1 erhöht und anderen Threads somit mitgeteilt das wieder auf den kritischen Bereich zugegriffen werden.

Verwendung von Semaphoren

In diesem Beispiel wird der Zugriff auf die Semaphore mittels **`try-finally`** abgesichert. Falls nun eine **`exception`** im kritischen Abschnitt abgeworfen wird, wird immer die **`release`**-Methode ausgeführt. Was schlussendlich bedeutet, dass die gemeinsam genutzte Ressource auch nach einem Misserfolg freigegeben wird.

```
200  //…
210  semaphor.acquire();             //Hier wird überprüft ob noch
220  try{                            //Drucker frei ist
230    DruckerHandler.SendDocument(myDocument.doc);
240  }       //Psudocode: Dokument wird an den Handler
250   //geschickt, welcher überprüft welcher
260   // Drucker verwendet werden soll.
270
280   catch(SomeException se){
290     //…
300   }
310   finally{
320     semaphor.release();
330
340   // Die Semaphore wird wieder freigegeben, bzw value wird wieder
350   // um 1 inkrementiert damit andere Prozesse darauf zugreifen
360   // können.
370   }
//…
```

5 Der TAS-Befehl (Test-And-Set beim M68000)

Integer Instructions

TAS — **Test and Set an Operand** (M68000 Family) — **TAS**

Operation: Destination Tested → Condition Codes; 1 → Bit 7 of Destination

Assembler Syntax: TAS < ea >

Attributes: Size = (Byte)

Description: Tests and sets the byte operand addressed by the effective address field. The instruction tests the current value of the operand and sets the N and Z condition bits appropriately. TAS also sets the high-order bit of the operand. The operation uses a locked or read-modify-write transfer sequence. This instruction supports use of a flag or semaphore to coordinate several processors.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X — Not affected.
N — Set if the most significant bit of the operand is currently set; cleared otherwise.
Z — Set if the operand was zero; cleared otherwise.
V — Always cleared.
C — Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Effective Address field—Specifies the location of th

Figure 1: Test and Set beim M68000

Instruction Fields:

Effective Address field—Specifies the location of the tested operand. Only data alterable addressing modes can be used as listed in the following tables:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	—	—
(An) +	011	reg. number:An			
– (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—

MC68020, MC68030, and MC68040 only

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
(bd,An,Xn)*	110	reg. number:An	(bd,PC,Xn)*	—	—
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	—	—
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	—	—

*Can be used with CPU32.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	EFFECTIVE ADDRESS					
										Mode			**Register**		

Tabelle 1: Bitbelegung des <ea>-Operanden des TAS-Befehls [6]

Die Testadresse für den `TAS`-Befehl ist in die Bits „Mode“ und „Register“ aufgeteilt. Der Befehl `TAS` überprüft nun das Most-Signifikant-Bit (MSB) des Bytes welches sich an **„EFFECTIVE ADDRESS”** befindet und schreibt anschliessend das Resultat in das `CCR`-Register.

Systembyte								Userbyte, bzw CCR							
15	14	13	12	11	10	9	8	7	6	5	4	**3**	**2**	1	0
T		S		I_2	I_1	I_0					X	**N**	**Z**	V	C
*		*		*	*	*					-	*	*	0	0

Tabelle 2: Condition Code Register des Motorola 68000

Im Userbyte werden nach Operationen wie Addition, Multiplikation aber auch nach dem **TAS**-Befehl die entsprechenden Bits gesetzt. Für das **TAS**-Kommando sind die **Bits 2** und **3** relevant:

N: *„Negativ-Flag: Wird gesetzt, wenn das Ergebnis einer Operation negativ ist, d.h. wenn im Ergebnisregister das höchstwertigste Bit, MSB = 1 ist.“ [7]*

(Kritischer Bereich belegt)

Z: *„Zero-Flag: Ergebnisregister wird gleich Null“ [7]*

(Kritischer Bereich frei)

Zusammenfassend lässt sich nun sagen das der **TAS**-Befehl das Byte an der Stelle **„EFFECTIVE ADDRESS”** überprüft und schreibt das Ergebnis in das **CCR**-Register. Dieses **CCR**-Register kann nun überprüft werden und abhängig vom Resultat an eine Stelle gesprungen werden. Der untenstehende Pseudo-ASM-Code soll dies verdeutlichen: [7][8] [9]

Anwendung in Pseudo-ASM-Code:

```
10   ;Uncritical section
20   check:
30   TAS      testbyte    ;     testet das testbyte ob bereits andere
40                        ;     Zugriffe auf den Speicherbereich
50                        ;     bereits erfolgen, Resultat wird im CCR
60                        ;     abgespeichert
70
80   cmp      ccr, #0     ;     CCR wird überprüft und an die
90                        ;     entsprechende Stelle gesprungen.
100
110 jnz       check       ;     Falls das MSB = 1 ist, dann wird
120                       ;     nochmals das testbytegeprüft
130 returnpath            ;     Falls MSB = 0 wird der Zugriff in den
140                       ;     kritischen Bereich erlaubt.
150  ;Uncritical section
```

Den Ablauf des Pseudo-ASM-Code soll folgende Grafik noch weiter verdeutlichen

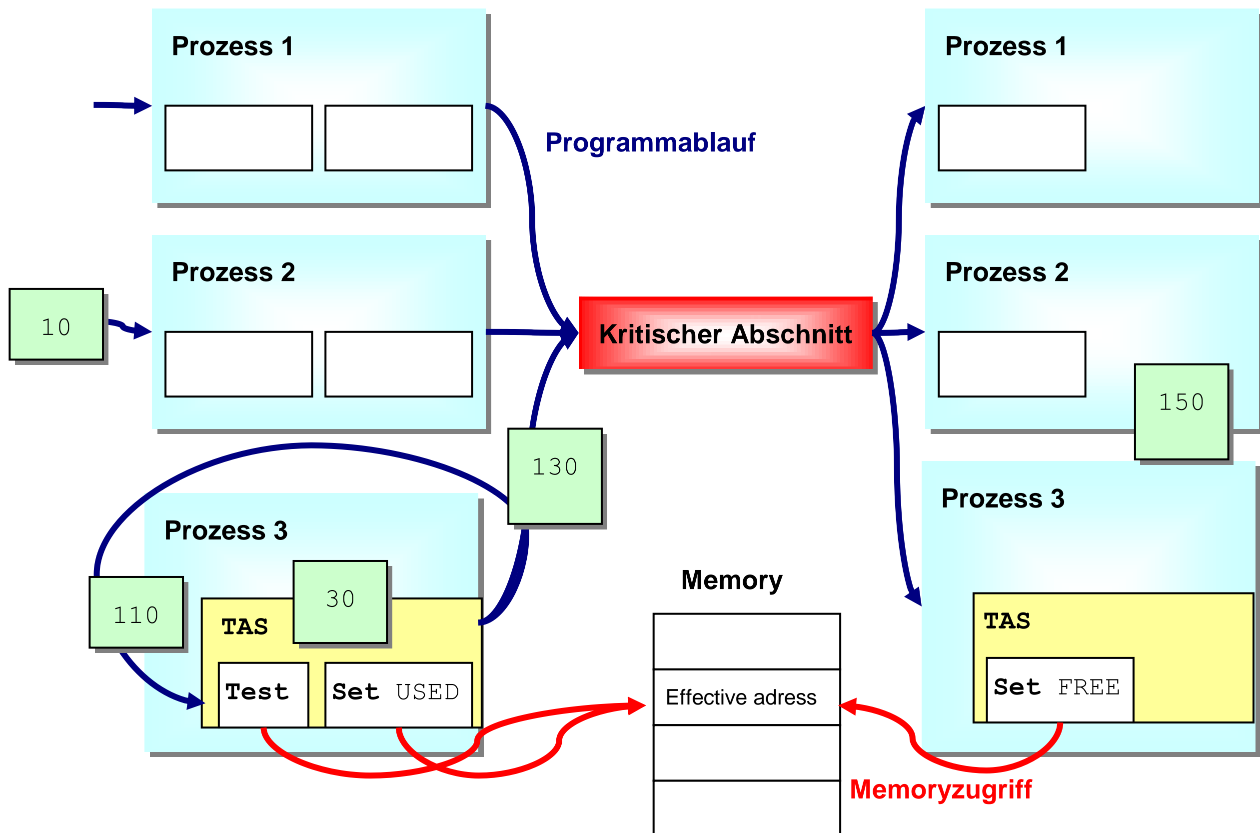

Figure 2: Programmablauf des TAS-Commands

In diesem Beispiel wollen 3 Prozesse auf den kritischen Abschnitt zugreifen (An Stelle `10`). Alle Prozesse testen mittels **`TAS`**-Befehl (Zeile `30`) ob der kritische Abschnitt bereits belegt ist. Falls der Abschnitt bereits belegt ist, wird nochmals getestet (Befehl auf Zeile `110`)

Ist der kritische Bereich verfügbar ist, wird in den kritischen Abschnitt verzweigt (Zeile `130`). Nach Ablauf des kritischen Bereiches wird mittels **`TAS`** automatisch der Bereich wieder freigegeben und wieder im normalen Programmablauf weiter verzweigt. (Zeile `150`)

Während dem ganzen lese und Testzugriff ist das **`/AS`** auf 0 gesetzt. Das **`/AS`**-Bit signalisiert, das der Hauptspeicher für andere Prozesse gesperrt ist

6 Anhang

6.1 Semaphoren in Delphi [10]

Dieses Beispiel soll dazu dienen um einen Einblick zu verschaffen wie Semaphoren in anderen Programmiersprachen implementiert werden können. Die grundsätzlichen Funktionsweisen von Semaphoren wurden markiert.

```
{
    This file is part of the Web Service Toolkit
    Copyright (c) 2006 by Inoussa OUEDRAOGO

    This file is provide under modified LGPL licence
    ( the files COPYING.modifiedLGPL and COPYING.LGPL).


    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
}
{$INCLUDE wst_global.inc}

unit semaphore;

interface

uses
  Classes, SysUtils, syncobjs{$IFNDEF FPC},Windows{$ENDIF};

{$INCLUDE wst.inc}
{$INCLUDE wst_delphi.inc}

type

  ESemaphoreException = class(Exception);

  { TSemaphoreObject }
```

Semaphoren-Klasse

```
  TSemaphoreObject = class
  private
    FHandle : {$IFNDEF FPC}THandle{$ELSE}PRTLEvent{$ENDIF};
    FLimit: Integer;
    {$IFDEF FPC}
    FCurrentState : Integer;
    FCriticalSection : TCriticalSection;
    {$ENDIF}
  public
    constructor Create(const ALimit : Integer);
    destructor Destroy(); override;
    function WaitFor(ATimeout : Cardinal) : TWaitResult;
    procedure Release();
    property Limit : Integer read FLimit;
  end;

implementation

{ TSemaphoreObject }
```

Semaphoren-Konstruktor

```
constructor TSemaphoreObject.Create(const ALimit: Integer);
```

```
begin
  Assert(ALimit>0);
  FLimit := ALimit;
{$IFNDEF FPC}
  FHandle := CreateSemaphore(nil,ALimit,ALimit,'');
{$ELSE}
  FHandle := RTLEventCreate();
  FCriticalSection := TCriticalSection.Create();
  FCurrentState := FLimit;
  RTLeventSetEvent(FHandle);
{$ENDIF}
end;

destructor TSemaphoreObject.Destroy();
begin
{$IFNDEF FPC}
  CloseHandle(FHandle);
{$ELSE}
  RTLeventdestroy(FHandle);
  FreeAndNil(FCriticalSection);
{$ENDIF}
  inherited Destroy();
end;

function TSemaphoreObject.WaitFor(ATimeout: Cardinal): TWaitResult;
{$IFNDEF FPC}
var
  intRes : DWORD;
begin
  intRes := WaitForSingleObject(FHandle,ATimeout);
  case intRes of
    WAIT_OBJECT_0  : Result := wrSignaled;
    WAIT_TIMEOUT   : Result := wrTimeout;
    WAIT_ABANDONED : Result := wrAbandoned;
    else
                     Result := wrTimeout;
  end;
end;
{$ELSE}
var
  ok : Boolean;
begin
  Result := wrTimeout;
  ok := False;
  FCriticalSection.Acquire();
  try
    if ( FCurrentState > 0 ) then begin
      Dec(FCurrentState);
      ok := True;
      if ( FCurrentState = 0 ) then
        RTLeventResetEvent(FHandle);
    end;
  finally
    FCriticalSection.Release();
  end;
  if not ok then begin
    RTLeventWaitFor(FHandle,ATimeout);
    FCriticalSection.Acquire();
    try
      if ( FCurrentState > 0 ) then begin
        Dec(FCurrentState);
```

Delphi bietet bereits eine Funktion, welche das Resultat der Prüfung interpretiert.

Beispielsweise kann eine definierte Zeit gewartet werden bis die Ressource frei ist bzw. in ein Timeout abläuft.

Anforderung der Ressource

Freigabe der Ressource

```
        ok := True;
      end;
    finally
      FCriticalSection.Release();
    end;
  end;
  if ok then
    Result := wrSignaled;
end;
{$ENDIF}

procedure TSemaphoreObject.Release();
begin
{$IFNDEF FPC}
  ReleaseSemaphore(FHandle,1,nil);
{$ELSE}
  FCriticalSection.Acquire();
  try
    if ( FCurrentState < Limit ) then begin
      Inc(FCurrentState);
    end else begin
      raise ESemaphoreException.Create('Invalid semaphore
operation.');
    end;
  finally
    FCriticalSection.Release();
  end;
  RTLeventSetEvent(FHandle);
{$ENDIF}
end;

end.
```

7 Quellenverzeichnis

[1] Die Kosten der Synchronisation

http://www.angelikalanger.com/Articles/EffectiveJava/39.JMM-CostOfSynchronization/39.JMM-CostOfSynchronization.html

[2] Atomare Operation

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

[3] Galileo ebooks - Threads und nebenläufige Programmierung

http://www.sws.bfh.ch/~amrhein/Swing/javainsel7/javainsel_10_001.htm

[4] Skript zum Seminarvortrag über Semaphoren

http://www.rz.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaabudud

[5] Giu's Journal - Synchronisation in Java: Semaphore

http://giu.me/13-02-2008-synchronisation-in-java-teil-2-semaphore.html

[6] MOTOROLA M68000 FAMILY Programmer's Reference Manual
http://www-clips.imag.fr/projet-systeme/68040/68kprm.pdf

[7] Hochschule Ravensburg-Weingarten - Mikroprozessoren Aufbau des MC 68000

http://www.hs-weingarten.de/~georgi/mcpr/public_html/mikrop_4.pdf

[8] Wolfram Schiffmann - Grundlagen Der Computertechnik, 2005

http://books.google.ch/books?id=SxLRn9RQGrEC&pg=PA147&lpg=PA147&dq=TAS+anwendung+M68000&source=bl&ots=FvqIjLKlCB&sig=afeznVXggS9lGV18SMkWd0idXzI&hl=de&ei=mDX5S_DCM5KhOLHO4JUM&sa=X&oi=book_result&ct=result&resnum=3&ved=0CC4Q6AEwAjgK#v=onepage&q&f=false

[9] Wikipedia - Test-and-set

http://en.wikipedia.org/wiki/Test-and-set

[10] Koders.com – Semaphores in Delphi

http://www.koders.com/delphi/fid3F54458A9B441759D9F8103058434C1CF71A3D49.aspx?s=proxy