Threads und Synchronisation

Dr. Arthur Zimmermann

Dr. Arthur Zimmermann

E-Mail: *azimmermann@beuth-hochschule.de*

Web: *http://azdom.de/go.htm*

Lernziele:

- Überblick über die Taxonomie der Threads und Prozessen.
- Aufbau eines grundsätzlichen Verständnisses der Synchronisation und Kommunikation zwischen Prozessen in einem Betriebssystem.
- Kennenlernen der Funktionsweise, Eigenschaften und Besonderheiten der Synchronisation.
- Überblick über die Realisierung der Synchronisation in gängigen Betriebssystemen.

Methoden:

- Theoretische Konzepte und Vorgehensweisen.
- Dialogisches Lernen.

Struktur und Inhalt des Seminars:

- Definitionen
- Taxonomie der Threads
- Taxonomie der Synchronisationsmechanismen
- Synchronisation
  - Mutexe
  - Signale
  - Semaphore
  - Monitore
    - Atomare Klassen
    - Bedingte Synchronisation
  - Sprachkonstruktionen
- Zusammenfassung
- Literatur

Prozess ist ein grundlegender Begriff für weitere Betrachtungen. Um ein Programm zu starten, erzeugt das Betriebssystem einen Prozess.

Prozess ist eine Zusammensetzung der Ressourcen (Betriebsmittel) und deren Ausnutzung zwecks Ausführung eines Programms.

Folgende Ressourcen werden einem Prozess zugeordnet (Prozesskontext):

- Prozessor (Prozessorkern);
- Adressraum;
- Stapel (Stack);
- Codesegment;
- Datensegment;
- Dateien;
- Zugriffsberechtigungen.

Thread ist ein sequenzieller Ablauf innerhalb eines Prozesses. Mehrere Threads können in einem Prozess gleichzeitig ausgeführt werden.

Dabei gibt es einige Ressourcen

- die gemeinsam verwendet werden können (Codesegment, Adressraum);
- die nur getrennt verwendet werden können (Prozessor, Stack).

Meisten moderne Betriebssysteme lassen eine flexible Konfiguration der Betriebsmittel, sogar zur Laufzeit.

Sinn der Threads — Ausführung des Programms zu beschleunigen.

Threads werden selbstverständlich laut der Semantik der Aufgabe vom Programmierer bestimmt.

Abgrenzung: Multitasking und Multithreading.

Die in einem Programm geschriebenen Operationen (Addition, Multiplikation, Funktion- und Methoden-Aufruf, Objekt-Instanziierung, Freigabe des Speichers eines dynamischen Arrays) sind im allgemeinen nicht auf die gleichen Operationen der Prozessors abgebildet. Prozessor stellt teilweise andere Operationen zur Verfügung. Das bedeutet, dass eine Programm-Operation aus mehreren Prozessor-Operationen bestehen kann.

Ein Thread kann bei Ausführung einer Programm-Operation unterbrochen werden, und Prozessor wird einem anderen Thread übergeben. Wenn die beiden Threads in diesem Moment die selbe Variable benutzen, dann können inkonsistente Zustände in beiden Threads entstehen.

Atomar (unteilbar) sind Programm-Operationen, die einzelnen Prozessor-Operationen entsprechen. Es gibt nicht viele davon: Zuweisung, logischer Vergleich und einige anderen.

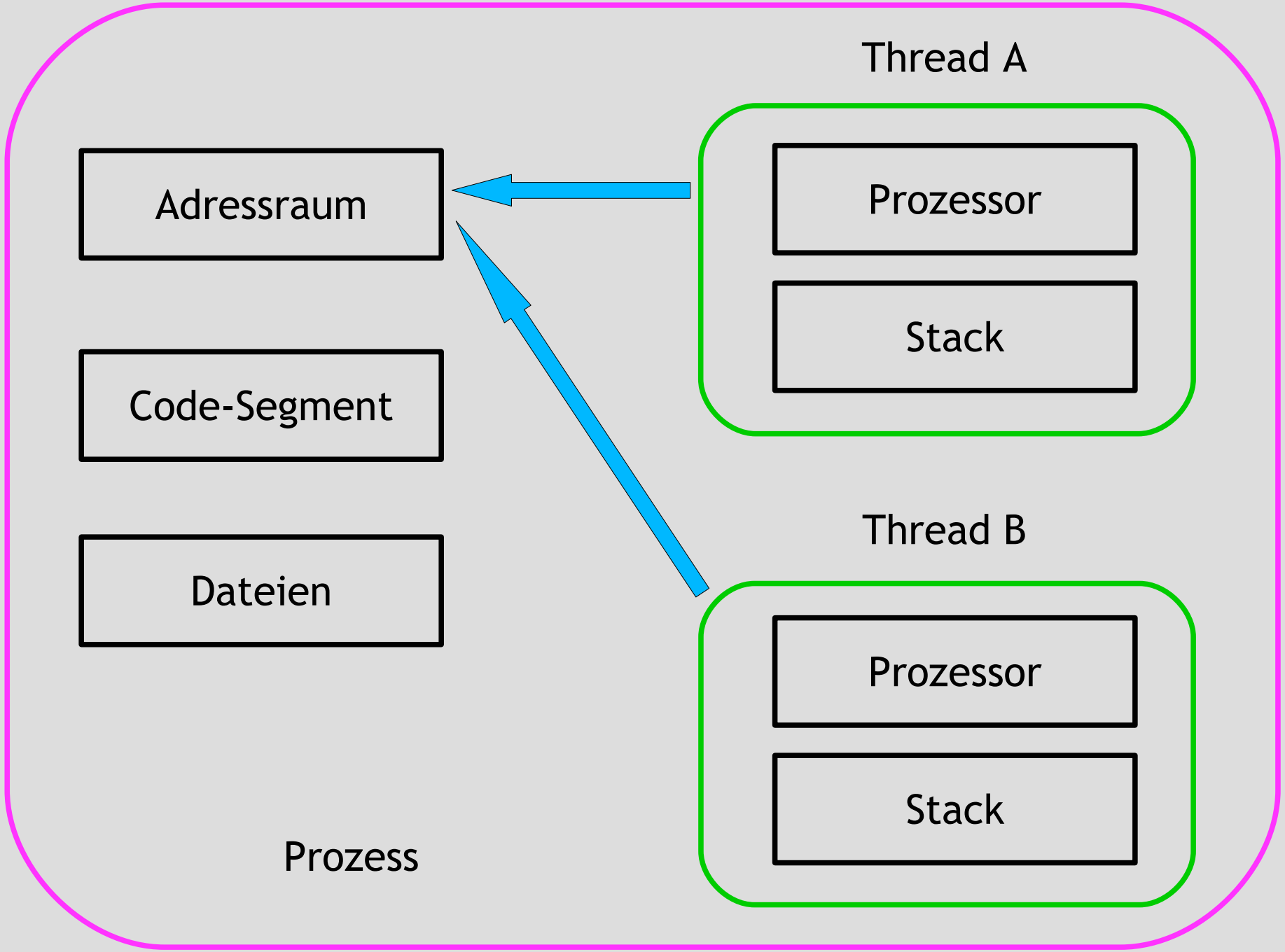
Welche Operationen sind atomar, ist von Hardware abhängig.

Da die Threads gleichzeitig laufen und Zugriff auf gemeinsame Ressourcen benötigen, können Probleme unterschiedlicher Arten auftreten, die in meisten Fällen einen undefinierten Zustand des ganzen Prozesses verursachen.

Unter Synchronisation versteht man Mechanismen, die mögliche Konflikte bei gemeinsamer Verwendung der Ressourcen vorhersehen und vermeiden.

Beispiele für mögliche Konflikte:

- das Erzeuger-Verbraucher-Problem;
- das Problem der speisenden Philosophen;
- Schreibzugriff auf einen Datensatz in der Datenbank.


Thread A
Adressraum
Prozessor
Stack
Code-Segment
Thread B
Dateien
Prozessor
Stack
Prozess

Es existieren zwei Arten von Threads:

- Kernel-Threads. Diese Threads werden vom Programmierer definiert und beim Programmstart direkt in die Systemaufrufe des Betriebssystems abgebildet. Das Betriebssystem verwaltet vollständig diese Threads. Das Betriebssystem weist den Threads unterschiedliche Prozessoren zu, und die Threads laufen in diesem Fall tatsächlich parallel ab. Präemptives Multitasking.
- User-Threads. Diese Threads werden ebenfalls vom Programmierer definiert, aber nicht vom Betriebssystem, sondern von den mitgelieferten Bibliotheken unterstützt werden. Somit hat das Betriebssystem in diesem Fall keine Ahnung, dass Threads existieren. Zeitablauf dieser Threads wird von dem Programm implementiert. Sie übergeben die Steuerung einander selbst. Sie laufen alle sequenziell auf einem Prozessor ab. Kooperatives Multitasking.

Wählt man Abstraktionsniveau als Kriterium, dann kann man die Synchronisationsmechanismen in folgende Kategorien zuordnen:

- Funktionen, Datenstrukturen und Primitive der prozedurale Programmiersprachen;
- Klassen und Objekte der objektorientierten Programmiersprachen;
- Steuerungskonstruktionen der parallelen Programmiersprachen.

Abhängig von der Aufgabenstellung, der eingesetzten Programmiersprache und Erfahrungen ergibt sich ein großer Spielraum für einen Programmierer bezüglich Anwendung und Gestaltung der Synchronisationsmechanismen.

Dieser Mechanismus kann verwendet werden, um einen gleichzeitigen Zugriff von mehreren Threads auf gemeinsame Ressourcen zu vermeiden, so dass nur ein Thread aktuell zugreifen darf (gegenseitiger Ausschluss).

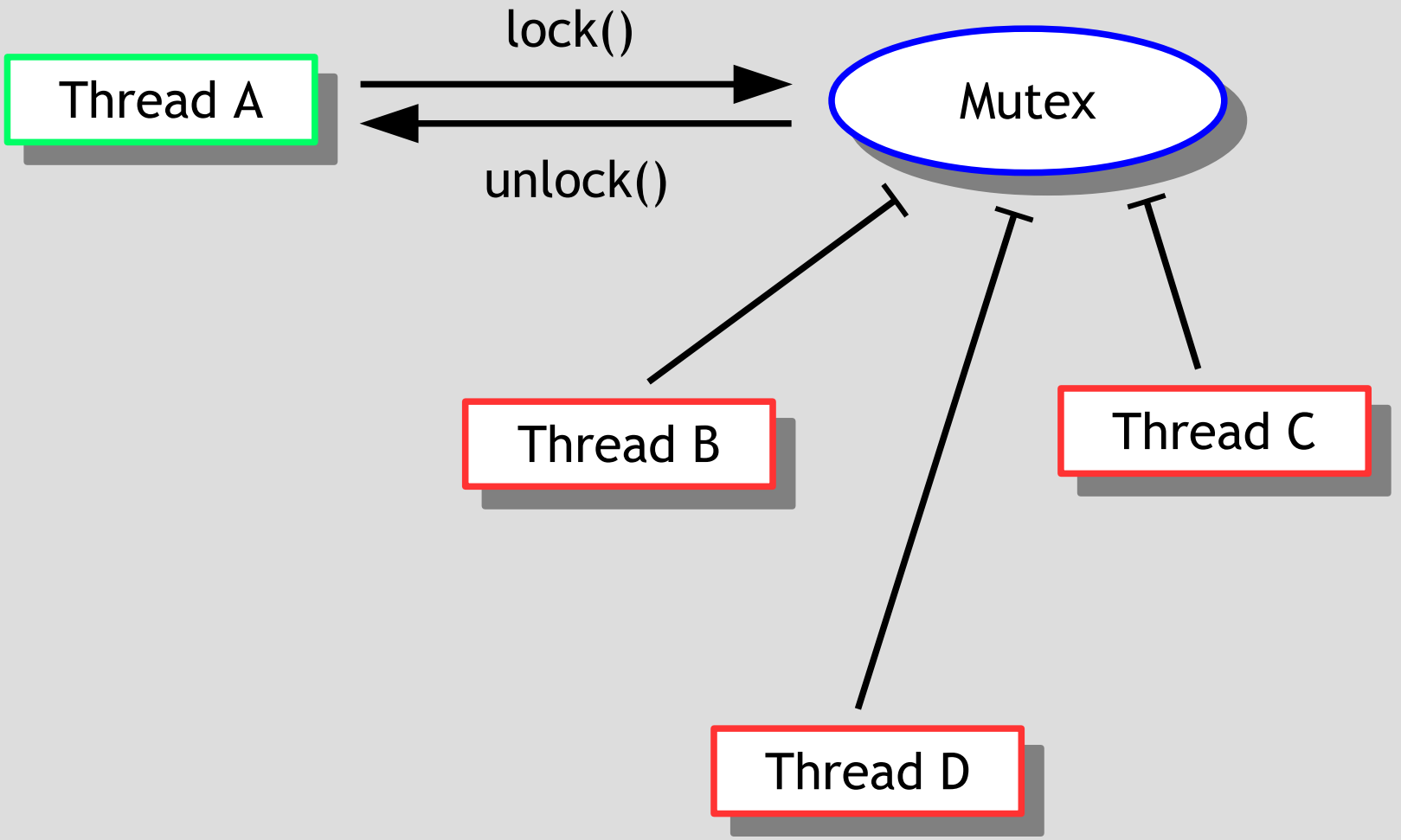
Unter dem Mutex (mutual exclusion) versteht man

- ein Verfahren;
- eine Datenstruktur, die für gegenseitigen Ausschluss verwendet wird.

Die Management-Funktionen für Mutexe werden von Programmiersprachen geliefert.

Idee:
Ein Thread blockiert einen Mutex und führt eigene Anweisungen durch. Solange Mutex bleibt blockiert, kann er von einem anderen Thread nicht blockiert werden.

Wichtige Eigenschaft dieses Mechanismus ist die Tatsache, dass ein Mutex von dem selben Objekt gesetzt und aufgehoben werden kann.


lock()
Thread A
Mutex
unlock()
Thread B
Thread C
Thread D

Beispiel

Zwei Threads schreiben unterschiedliche Daten in eine globale Variable und geben diese Variable auf dem Standard-Gerät aus.

Im allgemeinen kann man selbstverständlich nicht voraussagen, in welcher Reihenfolge die Ausgabe-Daten erscheinen. Man will aber sicherstellen, das ein Thread diese globale Variable setzt und ausgibt, und dass der andere Thread nicht dazwischen die Variable ändert.

```
char c;

void *out_plus (void)
{
  c = '+';
  sleep (1);
  printf ("%c", c);
}

void *out_minus (void)
{
  c = '-';
  sleep (1);
  printf ("%c", c);
}
```

```
int main ()
{
  pthread_t t1, t2;

  pthread_create (&t1, NULL, out_plus, NULL);
  pthread_create (&t2, NULL, out_minus, NULL);

  pthread_join (t1, NULL);
  pthread_join (t2, NULL);

  return 0;
}
```

```c
char c;
pthread_mutex_t m;

void *out_plus (void)
{
  pthread_mutex_lock (&m);
  c = '+';
  sleep (1);
  printf ("%c", c);
  pthread_mutex_unlock (&m);
}

void *out_minus (void)
{
  pthread_mutex_lock (&m);
  c = '-';
  sleep (1);
  printf ("%c", c);
  pthread_mutex_unlock (&m);
}
```

```c
int main ()
{
  pthread_t t1, t2;

  pthread_mutex_init (&m, NULL);

  pthread_create (&t1, NULL, out_plus, NULL);
  pthread_create (&t2, NULL, out_minus, NULL);

  pthread_join (t1, NULL);
  pthread_join (t2, NULL);

  return 0;
}
```

```c
/* ... declarations ... */
pthread_mutex_t m;

void *funcA (void)
{
  /* ...  code  ... */
  pthread_mutex_lock (&m);
  /* ...  code  ... */
  pthread_mutex_unlock (&m);
  /* ...  code  ... */
}

void *funcB (void)
{
  /* ...  code  ... */
  pthread_mutex_lock (&m);
  /* ...  code  ... */
  pthread_mutex_unlock (&m);
  /* ...  code  ... */
}
```

```c
int main ()
{
  pthread_t t1, t2;

  pthread_mutex_init (&m, NULL);

  pthread_create (&t1, NULL, funcA, NULL);
  pthread_create (&t2, NULL, funcB, NULL);

  pthread_join (t1, NULL);
  pthread_join (t2, NULL);

  return 0;
}
```

Versucht ein Thread (A) einen schon von einem anderen Thread (B) gesetzten Mutex zu setzen, dann klappt dieser Vorgang nicht. Folgende Situationen sind für den Thread (A) denkbar:

- passiv auf Mutex-Freigabe warten, d.h. die Steuerung geht an das Betriebssystem, solange Mutex gesetzt ist;
- aktiv auf Mutex-Freigabe warten, d.h. der Thread (A) kann sich vorläufig irgendwelchen anderen Aufgaben widmen (sog. Spinlock);
- Versuche, den Mutex zu setzen, unterlassen (vielleicht nach Ablauf bestimmter Zeit).

Außerdem bietet der Mutex-Mechanismus auch andere Möglichkeiten an:

- die Mutex-Attribute zu setzen und abzufragen;
- Bedingungsvariablen einzusetzen.

Signale stellen ein altes und mächtiges Mittel zur Kommunikation zwischen den Prozessen und Threads dar. Sie reichen auch heute für viel Anwendungsbereiche aus.

Threads können Signale verschicken und empfangen.

Funktionsweise von Signalen ist eigentlich eine Interrupt-Verarbeitung. Bekommt ein Thread ein Signal, dann passiert folgendes:

- wurde entsprechende Verarbeitungsfunktion für dieses Signal vorprogrammiert, dann wird sie aufgerufen;
- wurde keine Verarbeitungsfunktion vorprogrammiert, dann führt Kernel eine Standard-Aktion aus.

Einige Signale können keiner Verarbeitungsfunktion entsprechen. Für diese Signale führt das Betriebssystem immer eine Standard-Aktion aus. Empfängt ein Thread so ein Signal, dann gilt es oft für den ganzen Prozess.

Eigentlich können viele Signale ignoriert werden. Welche Signale verarbeitet und welche blockiert werden, wird in einer dem Thread zugeordneten Signalmaske festgeschrieben.

Der Mechanismus der Signale stellt viele Funktionen zur Verfügung, u.a.:

- *pthread_sigmask ()* – hiermit kann die Signalmaske abgefragt oder gesetzt werden;
- *pthread_kill()* – ein Thread sendet dem anderen ein Signal;
- *sigwait()* – die Ausführung des Threads, wo dieser Aufruf stattfindet, hält an, bis ein bestimmtes Signal ankommt;
- *pthread_self()* – ID eigenes Threads wird ermittelt.

Dieser Mechanismus erlaubt den gleichzeitigen Zugriff von einer vorher festgelegten Anzahl der Threads auf gemeinsame Ressource.

Unter einem Semaphor versteht man ebenfalls

- ein Verfahren;
- eine Datenstruktur, die für diesen Mechanismus eingesetzt wird.

Die Management-Funktionen für Semaphore werden ebenfalls von Programmiersprachen zur Verfügung gestellt.

Als Datenstruktur besteht ein Semaphor im wesentlich aus

- einem Zähler;
- einer Warteschlange für Threads.

Funktionsweise des Semaphores:

- der Zähler wird mit einer maximal zulässigen Anzahl der Thread initialisiert;
- ein Thread dekrementiert den Zähler vor dem Zugriff auf die Ressource und inkrementiert danach;
- ist der Zähler gleich Null, wartet der Thread in der Warteschlange darauf, dass ein anderer Thread die Ressource freigibt.

Wichtig ist, dass die Ressourcen im Gegenteil zu Mutexen in einer beliebigen Reihenfolge freigegeben werden können.

Mit den Semaphoren kann man Mutex-Mechanismus simulieren, wenn man die maximale Anzahl der Threads mit 1 initialisiert.

Beispiel

Mehrere Digitalkameras überwachen eine Wartehalle eines Bahnhofs. Die Bilder dieser Kameras werden regelmäßig gespeichert. Gleichzeitig können mehrere Suchvorgänge von den Arbeitsplätzen der Operatoren gestartet werden. Die Suchvorgänge brauchen einen lesenden Zugriff auf Bilder, deswegen können (und sollen) mehrere Suchvorgänge gleichzeitig Bilder öffnen.

Dabei gibt es folgende Bedingungen:

- aus Performance-Gründen können nicht mehr als 3 Such-Threads auf die Daten zugreifen (Anfangswert für Zähler ist 3);
- ein Schreib-Thread muss Vorrang vor Such-Thread haben.

Weiter werden nur die Suchvorgänge betrachtet. In jedem Suchvorgang (Thread) wird ein Bild von dem Speicherort geöffnet und mit einem vorgegebenen (gemeinsamen) Muster-Bild verglichen.

```
class T extends Thread
{
  private Semaphore sema;

  public T(Semaphore s)
  {
    this.sema = s;
  }

  public void run()
  {
    sema.P();  /* Zähler -- */

    /* Bild öffnen */

    sema.V();  /* Zähler ++ */

    /* Bild mit dem Muster
       vergleichen */
  }

}
```

```
void main (void)
{
  Semaphore s = new Semaphore(3);

  T t1, t2, t3, t4;

  t1 = new T(s);
  t2 = new T(s);
  t3 = new T(s);
  t4 = new T(s);

  t1.run();
  t2.run();
  t3.run();
  t4.run();

}
```

Dieser Mechanismus erlaubt sehr feine Handhabung der Semaphore:

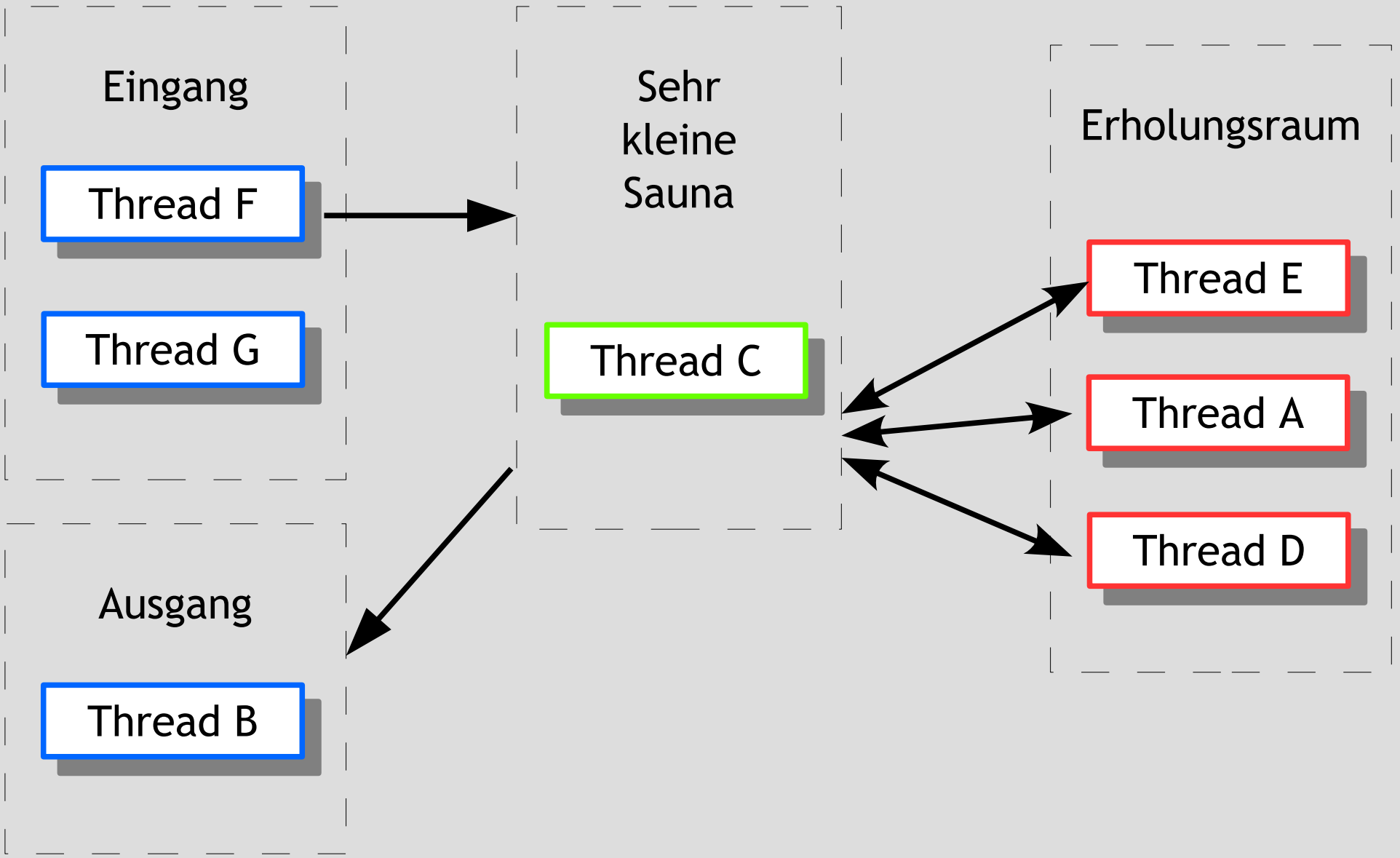
- Anweisungen für Manipulation der Semaphore
  (Erstellen, Löschen, Inkrementieren, Dekrementieren);
- Status-Abfrage der Anweisungen;
- Abfrage der Attribute;
- Einstellungen der Wartezeiten und anderen Optionen.

Die Semaphore werden üblicherweise für Beschränkung des gemeinsamen Zugriffs seitens der Threads eingesetzt und nur indirekt für die Synchronisation der Datenstrukturen.

Dieser Mechanismus ermöglicht einen wechselseitigen Ausschluss der Threads auf einer höheren Abstraktionsebene. Daher kann man sich bei Programmierung mehr mit Funktionalität des Codes und weniger mit Synchronisation beschäftigen.

Unter einem Monitor versteht man einen gekapselten kritischen Bereich des Programmcodes. Dabei wird eine Funktion (Methode) oder ein Code-Abschnitt mit dem Schlüsselwort *synchronized* gekennzeichnet.

Wird ein Objekt von einer Klasse mit *synchronized*-Methoden oder *synchronized*-Abschnitten instanziiert, dann werden solche Methoden oder Abschnitte unter wechselseitigem Ausschluss ausgeführt. Deswegen wird eine Warteschlange jedem Monitor zugeordnet, in der sich die wartenden Threads aufhalten.


Eingang
Thread F
Thread G
Sehr
kleine
Sauna
Thread C
Erholungsraum
Thread E
Thread A
Thread D
Ausgang
Thread B

```
class MammaMia
{
  private Teigware t;

  synchronized public void Spaghetti(...)  /* synchronized-Methode */
  {
      ...
  }

  public void Pizza(...)
  {
    ...
    synchronized(this)  /* synchronized-Abschnitt */
    {
      cutMozarella();
      cutSalami();
    }
    ...
  }

}
```

Viele Prozessoren stellen atomare Anweisungen zur Verfügung, wie z.B. *compare-and-swap*, *test-and-set*, *fetch-and-add (i++)*. Aber die Compiler gewährleisten nicht unbedingt, dass der Quell-Code in solche Anweisungen übersetzt wird.

Mit dem Monitor-Mechanismus lassen sich Klassen konstruieren, die atomare Operationen auf dem höchsten Level erlauben.

Es ist selbstverständlich nicht garantiert, dass solche atomare Klassen die Effizienz der Prozessor-Anweisungen haben.

```
/* Mit dem Mechanismus der Monitore lassen sich Klassen konstruieren,
   die atomare Operationen auf dem höchsten Level erlauben */


public class AtomicInteger
{
  private int i;

  public synchronized boolean SafedSet (int oldValue, int newValue)
  {
    if ( i == oldValue )
    {
      i = newValue;
      return true;
    }
    return false;
  }
}
```

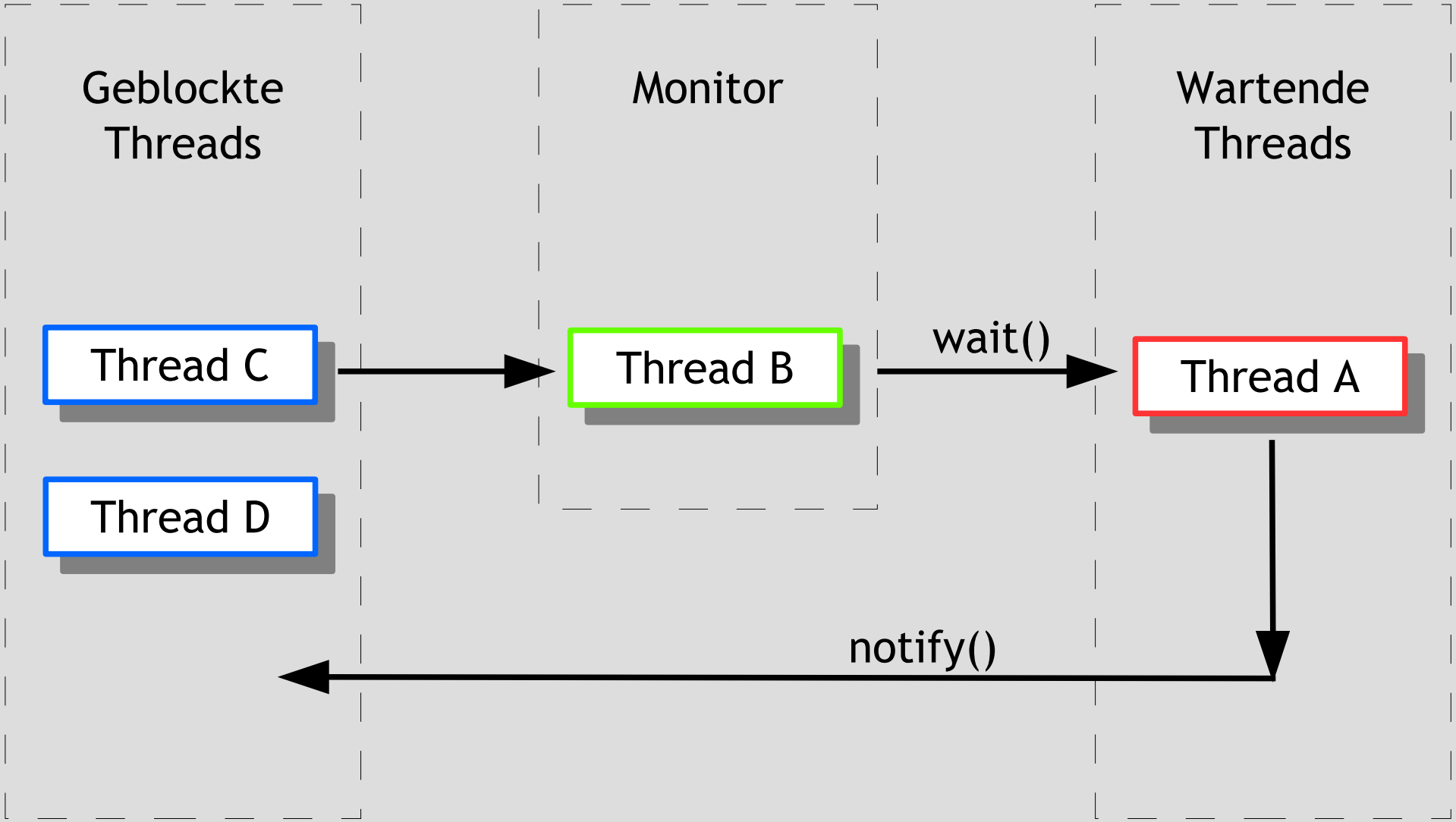
Synchronisation mit Monitoren ist nicht immer ausreichend, insbesondere wenn die Threads unter einander abhängig sind. In dem Fall müssen diese Threads explizit synchronisiert werden.

Der Konzept der Monitore wurde um einige Methoden erweitert, die eine Abstimmung zwischen den Threads ermöglichen:

- Methode *wait()*. Ruft ein Thread diese Methode auf, dann wird er in Wartezustand versetzt und wartet auf ein Weck-Signal.
- Methode *notify()*. Ruft ein Thread diese Methode auf, so schickt er ein Weck-Signal an einen wartenden Thread.
- Methode *notifyAll()*. Ruft ein Thread diese Methode auf, so schickt er ein Weck-Signal an alle wartenden Threads.

Die Zusammenarbeit dieser Methoden kann man sich wie folgt vorstellen.

Abhängig von der Programmiersprache können die oben erwähnten Methoden entsprechend *Wait()*, *Pulse()*, *PulseAll()* heißen.


Geblockte Threads
Monitor
Wartende Threads
Thread C
Thread D
Thread B
wait()
Thread A
notify()

```
/* Semaphore mittels bedingter Synchronisation emulieren */

public class Semaphore
{
  private int Zaehler;

  public Semaphore (int obereGrenze)
  {
    Zaehler = obereGrenze;
  }

  synchronized public void P()
  {
    while (Zaehler == 0) wait();
    Zaehler --;
  }

  synchronized public void V()
  {
    Zaehler ++;
    notifyAll();
  }

}
```

Die einfachste Sprachkonstruktion, die eine wichtige Rolle in der Kommunikation der Threads spielen kann, ist die Primitive *volatile*.

Die Primitive *volatile* bedeutet, dass der Inhalt der Variable außerhalb eines Programms (eines Threads) verändert werden kann.

Ist eine Variable mit dieser Primitive deklariert/definiert, dann wird sie bei jeder Verwendung direkt aus dem Arbeitsspeicher aufgerufen. Compiler schaltet jegliche Optimierung für so eine Variable aus.

Dadurch wird es gewährleistet, dass ein veränderter Wert der Variable aus dem Cache nicht aufgerufen werden kann.

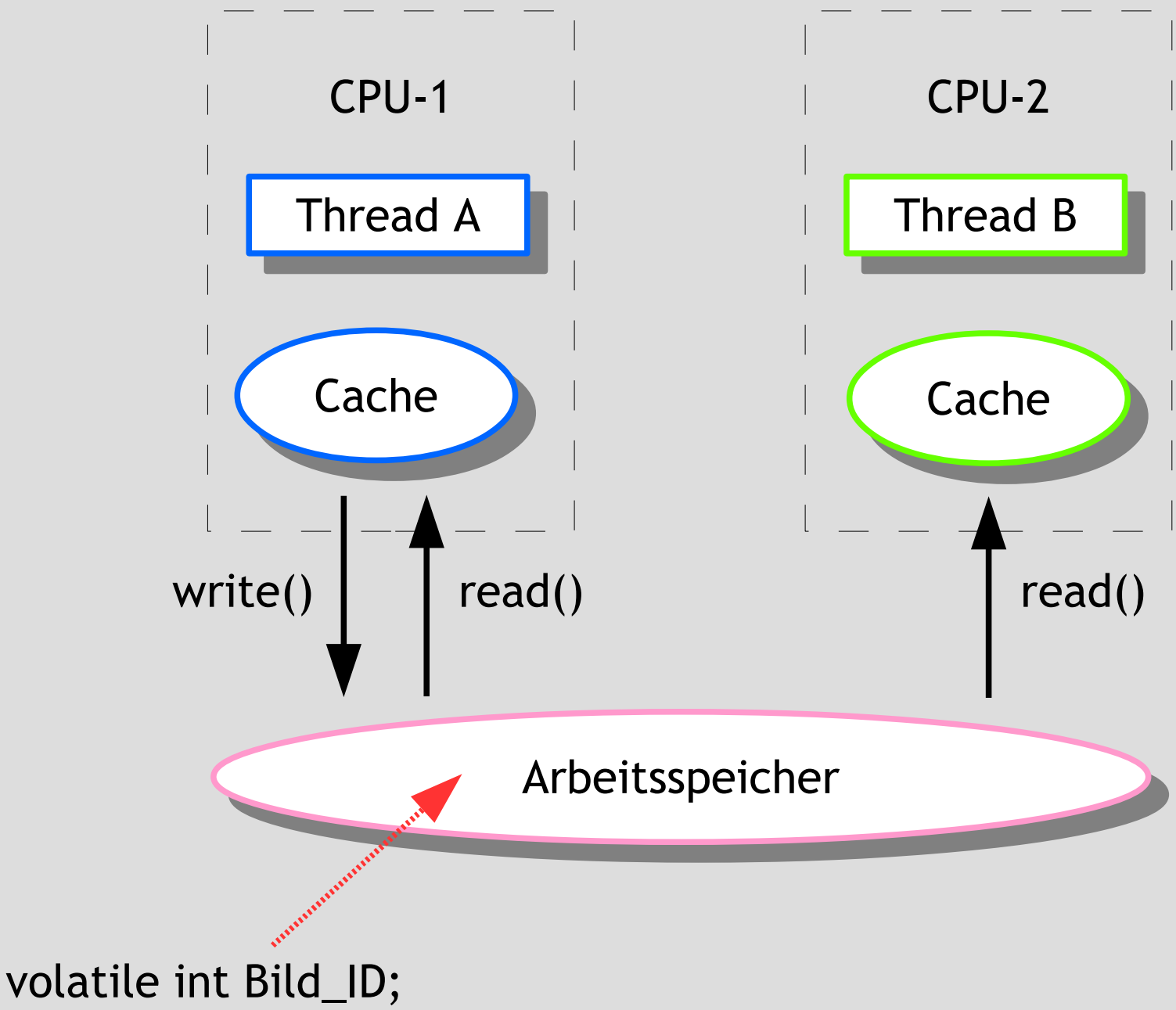
Zu einer Synchronisation der Threads bringt es noch nicht viel, aber es wird sichergestellt, dass die Threads immer einen aktuellen Wert der Variable benutzen.

```
/* ID eines Bildes ermitteln und entsprechendes Icon vergrößern */


volatile int Bild_ID;

void Thread_A (void)          /* ID eines Bildes ermitteln */
{
  ...
  Bild_ID = SearchPicID(); /* Ohne volatile                        */
  ...                      /* nicht unbedingt im Arbeitsspeicher */
}

void Thread_B (void)          /* Bild vergrößern */
{
  ...
  Zoom (Bild_ID);          /* Ohne volatile          */
  ...                      /* wird vom Cache gelesen */
}
```


CPU-1
CPU-2
Thread A
Thread B
Cache
Cache
write()
read()
read()
Arbeitsspeicher
volatile int Bild_ID;

In allen modernen Programmiersprachen sind Klassen, Objekte, Strukturen und Mechanismen vorhanden, um Threads zu erzeugen und deren Ablauf zu synchronisieren.

Es gibt Klassen von Mutexen, Semaphoren, Monitoren mit passenden Methoden und Eigenschaften.

Einige Programmiersprachen besitzen sogar Steuerungskonstruktionen zur Parallelisierung der Abläufe — das sind parallele Programmiersprachen, so wie Occam, X10, *Lisp, Ada, Scratch, Erlang.

```
/* Beispiele der Steuerungskonstruktionen in Programmiersprachen */


/* Sprache VECTOR */
  vector A(16), B(16), C(16);
  [3] -> A;
  [4] -> B;
  A + B -> C;


/* Sprache Occam */
PAR
  SEQ
    Read_1 ? A
    Read_2 ? B
    C := A * B
    Write_1 ! C
  SEQ
    Read_3 ? D
    Read_4 ? E
    F := D * E
    Write_1 ! F
```

The following was presented in this lecture:

- Important definitions and terms were considered first. They form a common basis for understanding the synchronization of the parallel running processes and threads.
- Then the classification of threads and synchronization mechanisms have been shown.
- At last, the individual mechanisms for synchronization of threads was discussed, whose realization and examples in programming languages.

1. A. Tanenbaum. Moderne Betriebssysteme.
   Pearson Studium, 2009.

2. A. Tanenbaum, M. van Steen. Verteilte Systeme, Grundlagen und Paradigmen.
   Pearson Studium, 2007.

3. P. Mandl. Grundkurs Betriebssysteme. Architekturen, Betriebsmittel-verwaltung, Synchronisation, Prozesskommunikation, Virtualisierung.
   Springer Vieweg, 2014.

4. J. Wolf. Linux-UNIX-Programmierung.
   Rheinwerk Computing, 2006.

5. R. Oechsle. Parallele ud verteilte Anwendungen in Java.
   Carl Haser, 2014.