Distributed Systems

Chapter 3: Synchronization Mechanisms

Change history

Date	Changes	Name
x.x.2022	Completion of the slides	Khader Karkar
12.04.2024	Revision of the slides 1 to 28	Bora Derici
14.04.2024	Revision of the slides 29 to 81	Bora Derici

Synchronization problems with parallel access

Since parallel programs consist of several sequential execution threads, there is often a need to exchange data between programs or within programs or to report back results. As soon as several parallel accesses are made to a variable, a function or a device, synchronization problems or interference* can occur.

Synchronization methods are required whenever synchronization problems or interference occur. This chapter takes a closer look at the common synchronization methods.

*Interference is the unsynchronized influencing of program states of one process on another [Kredel 2002] p. 15.

3.1 Correctness of parallel programs

In order to be aware of the requirements for parallel programs, one has to consider when a parallel program is correct. A **sequential** program is correct if the program does not terminate in an undefined manner, i.e. it outputs a "correct" result. **Parallel** programs must also fulfill two additional properties: **safety** and **liveness** [Ben-Ari 1982] p. 20f

3.1.1 Safety*

Safety generally describes the property that no faulty states occur in a program and deals with the static areas of the program. A static area of a program is, for example, (the definition of critical areas**) these are fixed in advance. If more threads or processes enter the same critical area than are permitted in this area, then a security property is violated.

*Safety: There are no faulty states

** A critical area is a device, a program, or a code segment within a program that can only be executed by a certain number of threads or processes at the same time. If the number is exceeded, incorrect results or program crashes can occur. Very often access is restricted to exactly one process or thread.

Secure programs are partially correct.

The fulfillment of the safety property is also referred to as **partial correctness**.

A partial correctness statement can be made for a program, for a part of a program, or just for a single statement. For this purpose, pre- and post-conditions are formulated for the instruction by statements of mathematical logic [Winskel 1993].

Partial correctness statement:

{P}C{Q}

{Precondition} Command {Postcondition}

In the following, an example is presented for which a partial correctness statement can be formulated very easily: a swapping algorithm.

```
z = x;
x = y;
y = z;
```

A partial correctness statement would be formulated as follows:

```
{x = m ∧ y = n}
z = x;
x = y;
y = z;
{x = n ∧ y = m}
```

Before the swap algorithm, x and y have any values, symbolized by the values m and n. After the swap algorithm, the values m and n are swapped, i.e. the variable x now has the value n and the variable y has the value m.

Safe Objects

Transferring the security property to the object paradigm means that a parallel program is secure if *all parallel objects within a program are secure*. Such objects are called safe objects.

An object is safe if it is either immutable or properly synchronized and properly contained. [Lea 1997].

Immutability

An object is immutable if **it never changes its state**. For example, a Java String is immutable. If a concatenation* of two string objects is carried out, then a new string object is created from the two objects, which contains the character strings of the other two objects. The disadvantage of programs that only contain immutable objects is that such programs do not allow any user interaction and can become very computationally intensive. As soon as the modifiability of objects is allowed, the methods of objects may either be **stateless**, only **local copies** of transferred parameters may be created and changed within a method or synchronization must be carried out.

* concatenation: the process of joining two strings together.

Stateless Method

A stateless method is a method that does not change any attributes of its object, i.e. leaves the state of the object unchanged.

```
public class Example {
   ...
   public int add(int a, int b) { return a + b;}
}
```

Program 3.1: stateless method

The add() method does not change any attribute of the object. The two variables a and b are passed from an external object. Therefore, this method is stateless.

Local Copies

When objects are passed to a method, they are usually passed by reference since copying an object usually takes more time and memory. The Java programming language does not provide an implicit mechanism for copying an object when parameters are passed. So if you want to operate safely on an object, you first have to create a copy of the object and then apply all operations to the local copy. However, this copy should only be used locally and may not be made accessible to other parallel objects.

```
public class Example {
   ...
   public String[] sort(String[] array) {
      String[] copy = new String[array.length];
      /* sort elements and store these in a copy */
      return copy; }}
```

Program 3.2: Method that makes a local copy

The strings to be sorted are copied first. Therefore, the calling method does not make any change in the string array. However, you have to make sure that no other objects can access this method.

Synchronization

When objects allow changes in their state, synchronization is required to keep the objects in a consistent state. To maintain the consistent state, parallel access to objects in the transient state must be prevented. Once a code segment that changes the state of an object is executed, the same code segment must not be executed in parallel (violation of security property).

Fully synchronized object

In Java, synchronization is (usually) controlled via the keyword synchronized. An object is said to be fully synchronized when all methods of the object have been synchronized. When an object is fully synchronized, it can only execute one instruction at a time. A fully synchronized object can be described as in Figure 3.1 [Heinzl 2004].

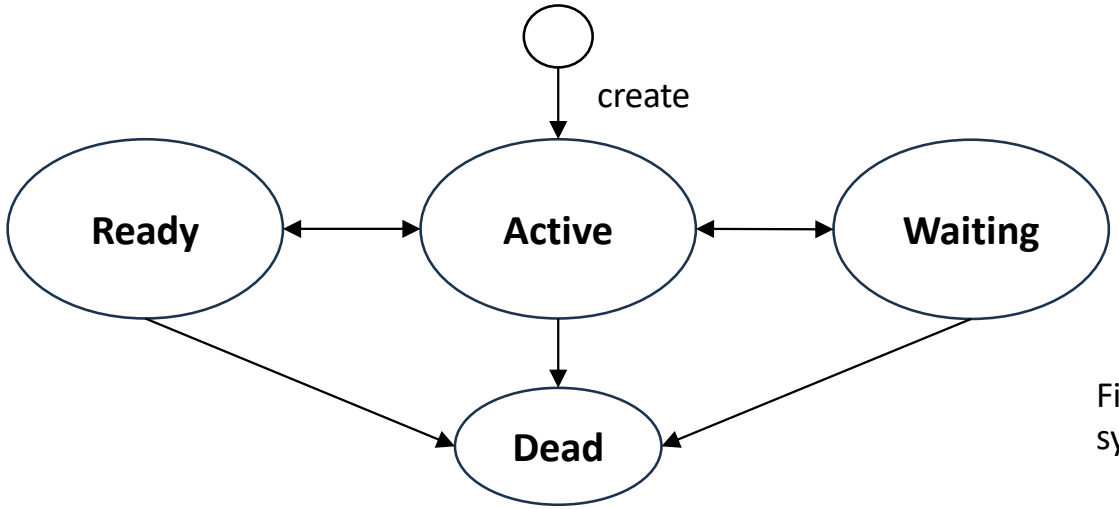
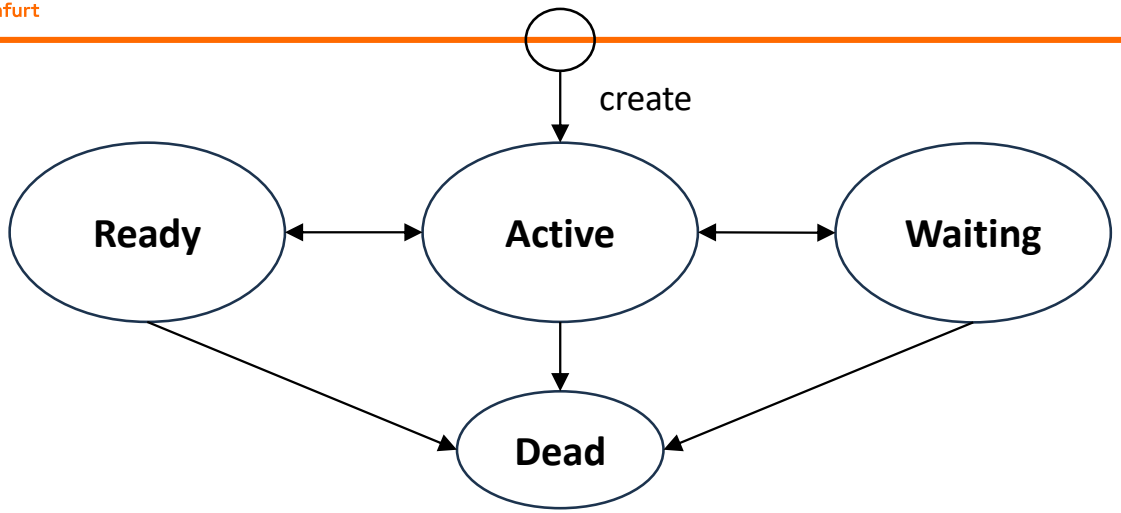


Figure 3.1: States of a fully synchronized object



Status	Description
Active (processing)	An object has seized the lock of the fully synchronized object. The fully synchronized object is processing the request.
Ready (idle)	No object has seized the lock of the fully synchronized object.
Waiting (for a reply)	The fully synchronized object has called a method of another object and is waiting for it to be processed.
Dead	No object holds a reference to the fully synchronized object.

```
public class Example{
   private int result = 0;
   private static int number = 0; // parallel Object
   public Example(){
   }//Constructor cannot be synchronized
   public synchronized  void add(int x){
      result = result + x; }
   public synchronized  void sub(int x){
      result = result - x; }
   public void increaseObjectNumber(){
      synchronized (getClass()){ // Get class lock
         number = number + 1;
      } }
   public void decreaseObjectNumber(){
      synchronized (getClass()){// Get class lock
         number = number - 1;
      } }
}
```

If available, access to static variables (class variables) must also be synchronized within a class. For this purpose one can make the synchronization through a class lock.

If an unsynchronized private method is only called from synchronized methods, then full synchronization is not violated.

Program 3.3: Fully synchronized object

Encapsulation

Instead of synchronization, which dynamically limits access to objects, access to objects can also be **structurally** limited by containment. Objects are logically encapsulated as shown in Figure 3.2.

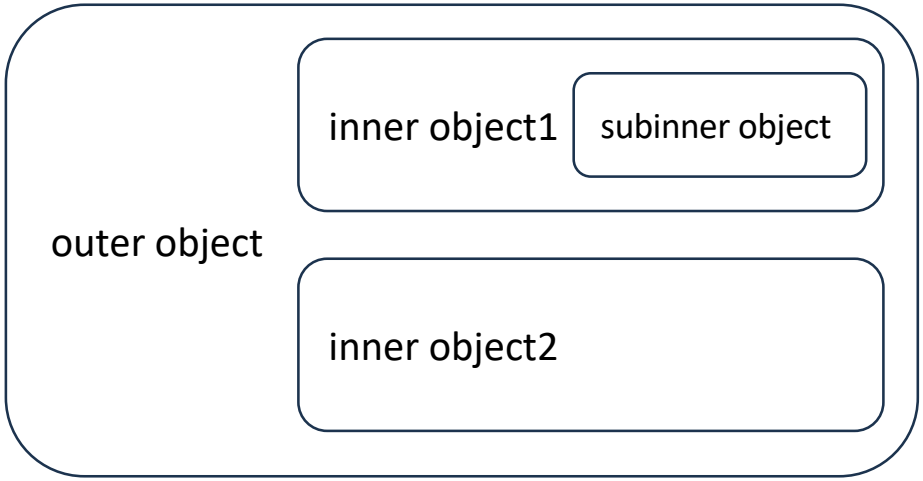
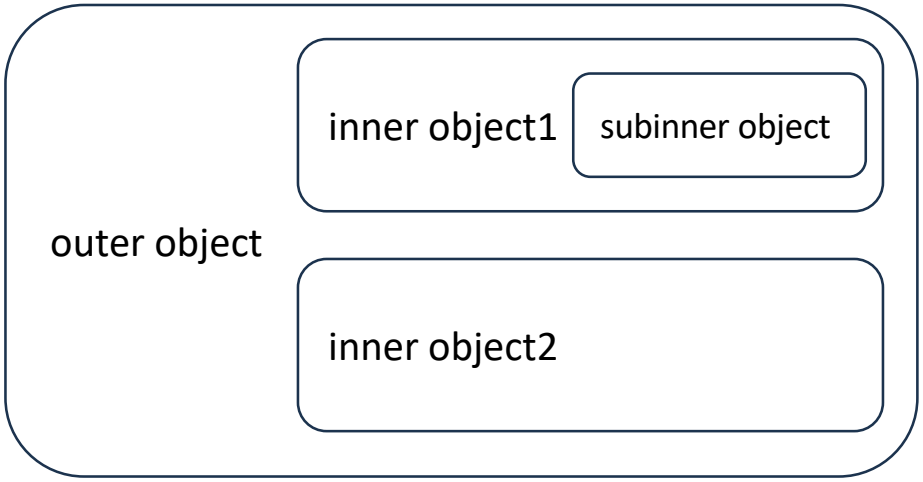


Figure 3.2: Encapsulation (containment) of objects [Lea 1997]



- **The following guidelines must be followed for a property to be safe:**
  - The outer object must create all inner objects within its own constructor.
  - The outer object must not pass any references of the inner object to other objects.
  - The outer object must be fully synchronized or embedded in another fully synchronized object.

3.1.2 Liveness

While the security property deals with the static areas of a program, the liveliness of a parallel program describes its dynamic properties [Ben-Ari 1982] p. 2lf. A program is alive if all desired states occur after finite time [Kredel 2002] p. 17. If a program is blocked forever, for example by a deadlock*, this is referred to as a liveness failure.

Liveliness: All desired states are reached after a finite amount of time

*Deadlock: explained in the next slide

Liveness failures are as follows:

[Lea 1997] p. 57f

Failure type	Description
Contention	Although a process/thread is in the ready/runnable state, it cannot access a processor because another process/thread has taken over the CPU. This Liveness failure is also referred to as starvation.
Dormancy	A process/thread in a non-executable state never returns to the runnable/ready state. In Java, for example, this can be done with a suspend() call without a corresponding resume().
Deadlock	Two or more threads are blocked forever, waiting for each other.
Premature Termination	A process/thread is terminated during its run time. In the programming language Java, this is possible using the stop() method.

Table 3.2: Possible viability failure

Deadlocks very often play a crucial role in liveness failures in the Java programming language.

Deadlocks

The first step in detecting a deadlock can be performed using a request-allocation graph or wait-for-graph. A request allocation graph describes all processes/threads that are waiting for resources.

The elements of the graph are defined as in Figure 3.3 [Spaniol 2002] p. 82:

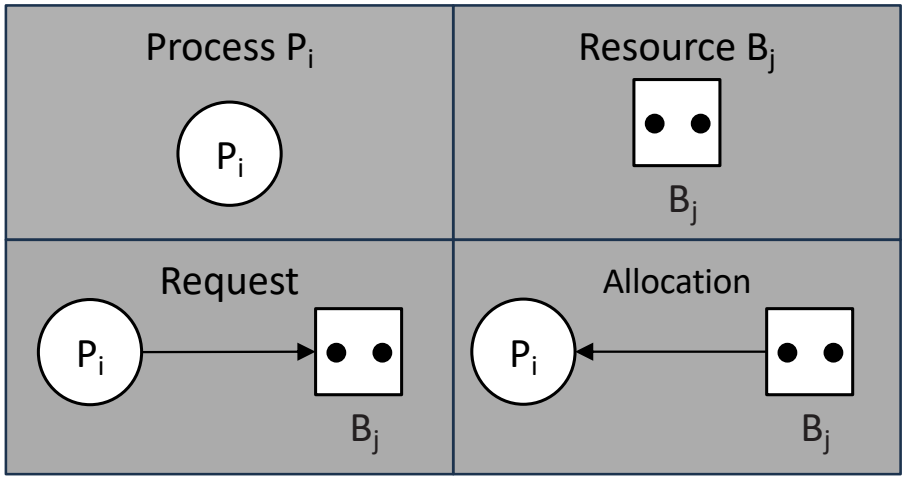


Figure 3.3: Elements of the request allocation graph

When a process references a resource in the graph, it requests that resource. If a resource references a process, then that resource is occupied by the process. The points of a resource indicate how many processes are allowed to occupy this resource.

All relationships between resources and processes can be entered in the request allocation graph. A process could both request and allocate multiple resources. Figure 3.4 shows an example graph:

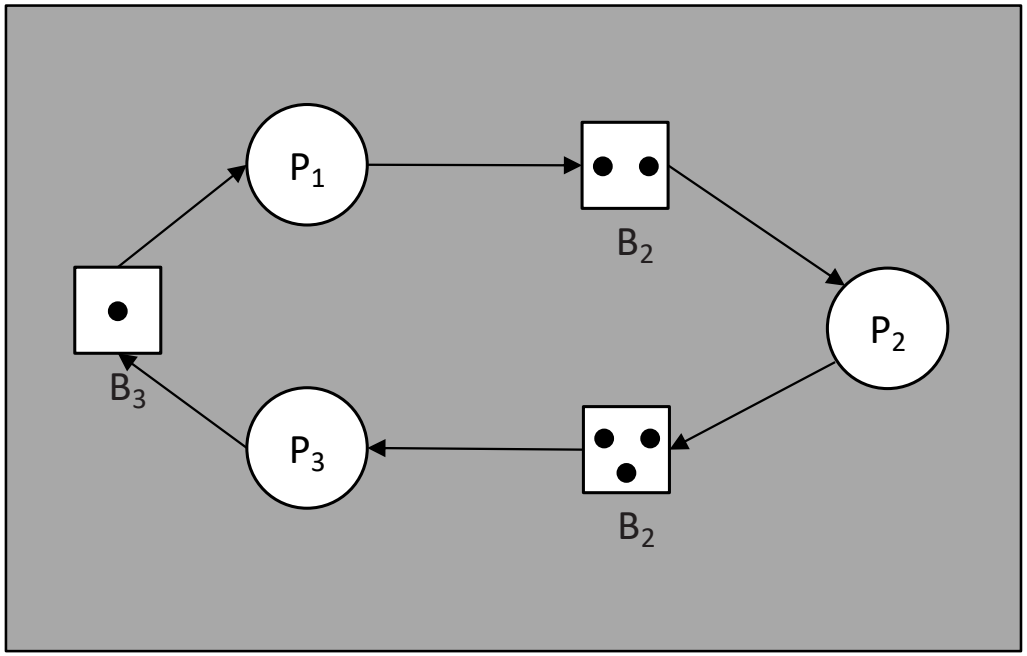


Figure 3.4: Example request allocation graph

The request-allocation graph can be simplified by removing all resources from the graph. A graph simplified in this way is called a wait-for graph. If the graph from Figure 3.4 is simplified in the way just described, the following wait-for graph results:

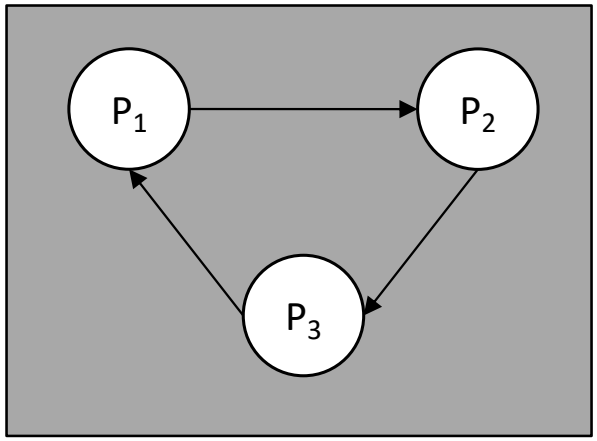


Figure 3.5: Wait-For-Graph

An arrow in this graph means that a process is requesting a resource that another process is holding. In Figure 3.5, P_1 requests a resource that is occupied by P_2. This graph can be used to determine whether a **cycle** is present.

A **deadlock** occurs when the four conditions from Table 3.3 are met:

Condition	Description
Circular-Wait	There is a cycle in the request-allocation graph or wait-for graph.
Exclusive-Use	No resource may be used by multiple processes (no resource sharing).
Hold-and-Wait	Processes can request new resources without returning those already occupied.
No-Preemption	Resources cannot be withdrawn from processes.

Table 3.3: Conditions for a deadlock

The request and allocation of resources not only applies to processes, but of course also to threads. For example, the Banker's algorithm can be used for deadlock avoidance with several similar assets [Spaniol 2002] p. 87.

Performance

The definition of the liveness property can be extended to the performance of a program (response times, etc.). If a program requires a certain level of performance in order to ensure meaningful execution, then the definition that a result is given in a finite time is not sufficient. Considering program liveness as the **minimum required performance** creates a dichotomy between program security and program liveliness.

Since synchronization mechanisms always take time, the question arises as to whether a program is still "alive enough" when fully synchronized. On the other hand, a question arises whether a deadlock that occurs very rarely within a program can be accepted if the program runs much faster.

Liveness is the minimum required performance of a program

3.2 Mutual exclusion

If synchronization is to be carried out, the problem of **mutual exclusion** often arises. If two or more processes or threads want to access a device or a program section, it must be ensured that the accesses are carried out one after the other, as otherwise it is possible to obtain undefined results. For example, if two processes are outputting to the standard output at the same time, then these outputs may be completely mixed up if no synchronization has been carried out.

Mutual exclusion is sufficient for critical areas in which only one process/thread is allowed to reside at a time. In object-oriented programming languages, it makes sense to implement the mutual exclusion via a lock class. In the following, various options are considered for how a mutual exclusion can be carried out.

3.2.1 Disabling Interrupts

When executing a program, a process/thread can be interrupted at any time by a context switch triggered by an expired time slice, events or I/O. These interrupts can be Deactivated for a processor. After the Deactivation, the running process will not respond to any interrupts. If the process enters an infinite loop, there is no way to allow another process to run. However, the easiest way to implement mutual exclusion is to turn off the interrupts [Heinzl 2004].

The simplest implementation would be to

1. disable the interrupts
2. run the critical section,
3. and re-enable the interrupts.

If this functionality is to be housed in a lock class, then this could be realized as in program 3.4 (after [Krishnamurthy 2004]).

```
class Lock{
   public void acquire(){
      /* Deactivating Interrupts */
   }
   public void release(){
      /* Activating Interrupts */
   }}
```

Program 3.4: Outline of a simple implementation of the mutual exclusion

The problem with this simple implementation is that a critical section can take a long time to run. Scheduling can no longer take place during the entire execution time because the execution of the current thread cannot be interrupted.

A better implementation of the lock class only occupies one variable with a specific value after the interrupts have been deactivated and then activates the interrupts again. Before acquiring the lock, other threads must first verify that the variable has the specified value. In this way, threads that want to enter the critical area block based on a variable value. The deactivation of the interrupts is very short.

No Busy Wait:

Between the deactivation and activation of the interrupts, queuing operations can still be executed which manage the waiting threads, so that the threads do not actively wait (busy wait).

The problem with the two solutions presented is their incompatibility with multiprocessor systems. The function test_and_set() (and possibly test_and_clear()) is used for multiprocessor systems.

```
class Lock{
   private int value = FREE;
   private /* Reference to queue */
   public Lock(){
      /* Initializing queue */ }
   public void acquire(){
      /* Deactivating Interrupts */
      if ( value == BUSY){
         /* Add thread to lock queue */
         /* Put thread to sleep */ }
      else { value = BUSY; }
      /* Enable interrupts; */ }
public void release(){
      /* Disable Interrupts */
      if (/* Queue not empty */){
      /* remove a thread from the lock queue*/
         /* Chain him to the  already queue */ }
      else { value = FREE; }
      /* Enable interrupts */ } }
```

3.2.2 test_and_set and test_and_clear

Test-and-Set and **Test-and-Clear** are atomic operations commonly used in synchronization mechanisms in concurrent programming. They are employed to handle critical sections, ensuring that only one thread or process accesses a shared resource at a time.

```
while (test_and_set(&lock) == 1);
critical section
lock = 0;
```

only one process can be in this section at a time

release lock when finished with the critical section

test_and_set:

test_and_set is an instruction used to write a value to a memory location and return its old value.

It assumes the memory location was initialized to 0. If the returned value is 0, the calling process obtains the lock. Otherwise, it enters a while-loop, spinning until the lock is available. This is called Spinlock. If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

How test_and_set works:

- It checks the value of a variable.
- It sets the variable to a new value (usually true or 1).
- It returns the original value of the variable
- If the function returns value 0, the calling thread may continue with the execution.

test_and_clear:

test_and_clear is a similar atomic operation used to clear (reset) a variable and return its old value. While test_and_set is more common in locking mechanisms.

- It checks the value of a variable.
- It clears the variable to a new value (usually false or 0).
- It returns the original value of the variable.

Use Cases

test_and_set:

- Mainly used for acquiring locks in concurrent programming.
- Ensures mutual exclusion.
- Commonly results in busy-waiting or spinning

test_and_clear:

- Used in scenarios where a thread needs to reset a flag or condition variable atomically after checking its previous state.
- Less common than Test-and-Set but valuable in specific contexts.

3.3 Semaphore [W]

A semaphore is an integer variable that can only take non-negative values. If a semaphore can only accept the values 0 and 1, then it is called a binary semaphore, otherwise it is called a general semaphore.

Two operations are defined for semaphores: P(s) and V(s). A semaphore is assigned to a queue using s (according to [Hansen 2002] Edsger W. Dijkstra: Cooperating Sequential Processes, p. 90ff). The queue is required to wait on blocked processes.

As described above, a semaphore can also be thought of as a data structure that contains a value and a reference to a queue.

3.3.1 Operation P( ) and V( )

The P( ) operation decreases the value of a semaphore by 1 if its value is greater than zero. If not, the calling process will be suspended.

The V( ) operation wakes up a suspended process, if there is one waiting. Otherwise, it increases the value of the semaphore by 1.

The two operations P() and V() are defined as primitive or atomic operations, i.e. if several P() and/or V() operations are to be executed at the same time, they are executed one after the other. To achieve this effect, one of the methods presented in Section 3.2 can be used. Therefore, P() or V( ) operations are always completed and cannot be interrupted.

The definitions are changed slightly for the implementation of the semaphore and the operations P() and V() [Freisleben1987 p.59ft]:

- A semaphore can also have negative values. The integer value tells you how many processes are currently waiting at the semaphore. If the semaphore is -n, it means that n processes are currently waiting for the semaphore.
- The P() operation decreases the value of the semaphore by 1. If the value of the semaphore is less than 0, the calling process is suspended.
- The V() operation increases the value of a semaphore by 1. If the value of the semaphore is less than or equal to 0, a suspended process is woken up.

A semaphore can also be implemented very well in an object-oriented language such as Java. Since the semaphore is defined via a structure to which a set of functions is applied, it makes sense to implement this concept via a class. Each semaphore object then has a P() and V() operation. In Section 3.5, the implementation of a semaphore in the Java programming language is considered with the help of its monitor concept.

3.3.2 Fairness

A semaphore is said to be fair if it ensures that no waiting process/thread can starve (starvation). Since no strategy is specified for queue management, it can happen that a process/thread is never executed. To illustrate why fairness can be very important, the following example considers a LIFO (last in, first out) process.

fair semaphore: no waiting process or thread can starve on the semaphore

Take a binary semaphore s with value 1 protecting a critical area from multiple simultaneous access and three processes P1, P2, P3.

- P1 performs the P() operation on s and enters the critical region.
- P2 and P3 also perform the P() operation on s and are blocked because the semaphore allows only one process into the critical area.
- P1 exits the critical area with a V() operation, P3 is awakened and next allowed into the critical area.
- P1 performs another P() operation and is blocked. P 3 leaves the critical area with a V() operation, P1 is woken up and allowed into the critical area.

	P1	P2	P3	
1	P(s)			s.value = s.value – 1
2	Critical Area			P1 enters critical area
3		P(s)		P2 is blocked
4			P(s)	P3 is blocked
5	V(s)			P3 is waken up
6			Critical Area	P3 enters critical area
7	P(s)			P1 is blocked
8			V(s)	P1 is waken up
9	Critical Area			P1 enters critical area

Program 3.9: Starving a process on an "unfair" semaphore

Theoretically, steps 4 to 9 can be repeated an infinite number of times without P2 ever entering the critical area. The process P2 is then referred to be starved.

To prevent a process from starving, a process queue management strategy must be implemented within the V() operation. An example of an easy-to-implement strategy is the FIFO (first-in, first-out) method. The process that is the first to be blocked at a semaphore is also the first to be woken up again. If a FIFO method is used in the above example, the following sequence results:

	P1	P2	P3	
1	P(s)			s.value = s.value -1
2	Critical Area			P1 enters the critical area
3		P(s)		P2 is blocked
4			P(s)	P3 is blocked
5	V(s)			P2 is Awaken
6		Critical Area		P2 enters Critical area
7	P(s)			P1 is Blocked
8		V(s)		P3 is Awaken
9			Critical Area	P3 enters critical area
10		P(s)		P2 is Blocked
11			V(s)	P1 is Awaken
12	Critical Area			P1 enters Critical area

Program 3.10: Process Order Sequence at a Fair Semaphore

For example:

if the critical work for each process requires the same amount of time and the scheduling is uniform, steps 4 through 12 could be repeated.

In any case, it is ensured that every process that wants to enter the critical area can enter the area in a finite time.

3.3.3 Semaphores and fairness in Java

acquire() and release()

In addition to various locks, the scope of the new J2SE 5 also includes an implementation of a semaphore class for the first time. The Semaphore class provides the acquire() and release() methods, with the acquire() method essentially corresponding to the P() operation and the release() method essentially corresponding to the V() operation. There is an additional possibility to use an integer as a parameter, which specifies by how much the value of the semaphore should be decreased or increased.

Barging

Furthermore, in the constructor of the semaphore it can be specified whether a fair or non-fair semaphore is to be created. A fair semaphore uses an FIFO (First-In-First-Out) technique for thread queue management. An unfair semaphore, however, differs significantly from the above definition. Such a semaphore enables so-called **barging**,

i.e. a thread executing the acquire( ) operation can be executed before a thread already waiting at the semaphore. So the release( ) method doesn't wake up a thread directly, it just increments the value of the semaphore. This allows a thread to starve.

Consider the following example: s is a binary semaphore and T1 and T2 are threads that want to enter the critical area managed by s. Both threads execute the following commands in an endless loop:

T_1	T_2
s.aquire()	s.aquire()
Critical Area	Critical Area
s.release()	s.release()

Program 3.11: Pseudocode for possible thread starvation

With a fair semaphore, there is no problem in the above example. An unfair semaphore can result in thread starvation. The pseudocode described in Program 3.11 can be converted into a simple Java program, as shown in Program 3.12.

```
1.  import java.util.concurrent.Semaphore;
2.  public class SemaphoreThread implements Runnable{
3.                  Thread t;
4.                  Semaphore s;
5.                  public SemaphoreThread(Semaphore s){
6.                                  this.s = s;
7.                                  t = new Thread(this);
8.                                  t.start();}
9.                  public void run(){
10.                                 for(::){
11.                                 try{
12.                                 System.out.print(t.getName());
13.                                 System.out.println(" trying to acquire permit");
14.                                 s.acquire();
15.                                 System.out.print(t.getName());
16.                                 System.out.print(" acquired permit");
17.                                 for(int i = 0; i <= 10000000; i++); // Critical operation
18.                                 s.release();
19.                 }catch (InterruptedException e){
20.                 e.printStackTrace();}}}
21.                 public static void main(System[] args){
22.                 // 1 permit, fairness: false
23.                 Semaphore s = new Semaphore(1,false);
24.                 SemaphoreThread st1 = new SemaphoreThread(s);
25.                 SemaphoreThread st2 = new SemaphoreThread(s);}
```

Program 3.12: Possible thread starvation

For example, this process produces the following result:

Thread-0: Trying to enter critical region.
Thread-0: Entered critical region.
Thread-1: Trying to enter critical region.
Thread-0: Left critical region.
Thread-0: Trying to enter critical region.
Thread-0: Entered critical region.
Thread-0: Left critical region.
Thread-0: Trying to enter critical region.
Thread-0: Entered critical region.
Thread-0: Left critical region.
…

Thread-0 is started, executes acquire(), enters the critical area and starts doing its work. Thread-1 starts, tries to enter the critical section and gets blocked. Thread-0 finishes its work and releases the semaphore. Since thread-1 is not woken up, thread-0 again gets permission to enter the critical area and performs critical work. Thread-0, in turn, finishes the critical work and releases the semaphore. From this point, a cycle can arise because thread-0 never has to give up execution involuntarily. If, on the other hand, the FIFO method is activated, the threads take turns doing critical work.

3.3.4 Mutual exclusion through semaphores

A binary semaphore is sufficient to implement mutual exclusion using semaphores. In the following example, assume that two worker threads are doing work in parallel. Both threads take turns doing normal and critical work. Critical work (e.g. printing, outputting to the screen, writing to a variable) can only be performed by one thread at a time. The other may have to wait.

```
import java.util.concurrent.Semaphore;
public class HelloFriend{
   public static void main(String[] args){
      // 1 permit, fairness: true
      Semaphore s = new Semaphore(1,true);
      new Worker(s);
      new Worker(s);
   }
}
import java.util.concurrent.Semaphore;
public class Worker implements Runnable{
   Thread t;
   Semaphore s;
   boolean running = true;

   public Worker(Semaphore s){
      this.s = s;
      t = new Thread(this);
      t.start();
    }
```

Program 3.13: Synchronized performance of critical work

Program 3.13: Synchronized performance of critical work

```
public void run(){
    while(running){
      try{
        // do normal work
        Thread.sleep((long) Math.random() * 3000 + 4000);
        // do critical work
        System.out.println(t.getName() + ": Trying to enter critical region");
        s.acquire(); // P(s)
        System.out.println(t.getName() + ": Entered critical region");
        System.out.println(t.getName() + ": Doing critical work");
        Thread.sleep((long) Math.random() * 3000 + 4000);
        System.out.println(t.getName() + ": Left critical region");
        s.release(); // V(s)
      }catch (InterruptedException e){
        e.printStackTrace();
      }}}}
```

If no synchronization is carried out, the security of the program will be violated (see Section 3.1.1 Security), since the critical areas are entered by several threads at the same time.

The P() operation **(acquire)** prevents a second thread from entering the critical area. When the first thread calls the P( ) operation, the value of the semaphore is reduced to 0. When the second thread performs the P( ) operation, it is blocked and queued. When the first worker leaves the critical area and executes the V() **(release)** operation, the second thread is removed from the queue and can then enter the critical area and do its work there.

3.4 Monitors

The monitor concept was introduced to enable synchronized access to **structured** data and their functions. A monitor can initially be thought of as *a class from the modern object-oriented languages*. A class consists of related data and functions (methods) that operate on that data. The class concept was implemented for the first time in the Simula 67 programming language. **according to [Hansen 2002] p. 272 C.A.R. Hoare: Monitors: An operating system structuring concept 1974).**

A monitor is a class that only allows sequential (not simultaneous) access to its functions by several processes/threads, .e. the monitor synchronizes the accesses of the processes/threads through an inherent mutual exclusion procedure, e.g. via locks [Hoare 1985] p. 228ff.

A monitor still protects direct access to its variables. The variables of the monitor can only be accessed via monitor functions. Each monitor has a body that is executed when the program or monitor is started [Ben-Ari 1982] p. 73ff. The body is used to initialize the monitor variable, similar to a constructor in object-oriented languages. As with the semaphores, a signaling mechanism is required to carry out the synchronization in the monitor.

3.4.1 Signaling Mechanisms

A new type of variable, the condition variable, is defined for monitors. The two functions wait(e) and signal(e) are defined for a condition variable e.

- wait ()

The wait(e) function schedules the process/thread waiting for condition e to be signaled. In addition, a process/thread is awakened at the monitor input and allowed into the monitor.

- signal ()

The signal(e) function wakes up a process/thread blocked by wait(e). If no process/thread is waiting in the queue of condition variable e, the signal(e) operation does nothing. Otherwise, the signal(e) operation wakes a process/thread from the queue. Since the problem arises after the execution of the signal(e) statement that there are two processes/threads in the monitor, a solution can be implemented in three different ways [Groß 2003] Chapter 3.4.3.

- The process/thread executing the signal() function is queued in a priority queue. As soon as the woken up process/thread executes the wait() function or leaves the monitor, a process/thread is woken up from the priority queue.
- The signal() function may only be used as the last function in the monitor. This ensures that the process/thread that calls signal() then leaves the monitor.
- The awakened process/thread is queued in a queue within the monitor or in the monitor's input queue. When the signaling process/thread leaves the monitor or calls wait(), the awakened process is activated. Instead of signal(), one also speaks of notify() in this context. This is also how the Java programming language implements the notify() statement

3.4.2 Nested Monitor Calls

A disadvantage of the monitor concept is the possibility of nested monitor calls. If a process calls a monitor function that in turn calls a function of another monitor, calling wait() will unlock the second monitor, but not the first. Processes blocked on the first monitor are not woken up. This can result in deadlocks.

Four methods are distinguished by which this problem can be solved:

1. Nested monitor calls are forbidden.
2. All locks are released again in call sequence when a lock is released.
3. Managers manage the equipment protected by monitors. A manager hereby distributes the permission to use a resource to a requesting process.
4. Monitors are used as a structuring tool for managing resources. Monitor functions can be executed simultaneously, only access to variables is protected by mutual exclusion.

Unfortunately, none of the four proposed solutions is entirely convincing. This problem persists when using monitors.

3.4.3 Monitors in Java

Since monitors are essentially classes whose functions (methods) can only be executed by one process/thread (in Java only by one thread) at the same time, their implementation in Java is relatively simple. The keyword synchronized can be used to protect a method from parallel access by threads, i.e. this keyword implements mutual exclusion. A monitor in Java is therefore a normal class whose variables are declared as private and whose methods are declared as public synchronized. The signaling mechanism is implemented in Java using the methods wait() and notify() or notifyAll(). The wait() and notify() or notifyAll() methods can be used for each synchronized block that is applied to the same object. A very simple example of the use of Java monitors (more precisely: for the signaling mechanism) is the safe suspension of threads.

Suspend Threads

If threads are very computationally intensive, it may make sense to defer their execution. In Java this is usually possible with the suspend( ) method. However, the suspend( ) method should only be used if the thread does not hold any locks, since the locks are not released during suspension. However, it is very difficult to find out whether the thread is holding locks when it is suspended, which is why deadlocks can very easily arise. Therefore, [Sun/Deprecation] proposes the implementation of a safely suspendable thread using monitors. Program 3.14 shows the implementation.

```
public class SuspensibleThread implements  Runnable{
   Thread t;
   boolean suspended = false;
   public SuspensibleThread(){
      t = new Thread(this);
      t.setName("SuspensibleThread");
      t.start();
   }
   public void run(){
      for(;;){
         try{
            synchronized(this){
               while(suspended) wait();
            }
         }catch (InterruptedException e){
            e.printStackTrace();
         }
         //Do work
        ....} }
```

The code continues on the next slide

Program 3.14: Safely suspendable thread

```
public void suspend(){
    suspended = true;
  }
  public synchronized void resume(){
    suspended = false;
    notify();
  }
}
```

Program 3.14: Safely suspendable thread

The downside of this method is that the thread to be suspended is not suspended immediately.

The thread must first get to the while(suspended) query in the run() method before it is blocked. The time it takes to reach this query depends on the particular run() method. With long run() methods, it would also be conceivable to use the while(suspended) query more than once. However, the query should only be used in places where no locks needed by another thread are held by the thread being suspended.

3.4.4 Nested monitor calls in Java

Nested monitor calls in Java have the following form:

```
1.  public class Nested {
2.  Object x = new String();
3.  Object y = new String();
4.  public void waitForSignal() {
5.  try {
6.  System.out.println(Thread.currentThread() + " waitForSignal: Try to aquire lock of x");
7.  synchronized(x) {
8.  System.out.println(Thread.currentThread() + " waitForSignal: Aquired lock of x");
9.  Thread.sleep((long) (Math.random()*5000));
10. System.out.println(Thread.currentThread() + "waitForSignal: Try to aquire lock of y");
```

```
11. //also method call. which contains synchronized(y). possible
12. synchronized(y){
13. System.out.println(Thread.currentThread() + ” waitForSignal: Aquired lock of y");
14. Thread.sleep((long) (Math.random()*5000));
15. System.out.printlnCThread.currentThreadO + " wait");
16. y.wait();
17. }}}catch(InterruptedException e){
18. e.printStackTrace();}}
19. public void signal() {
20. System.out.println(Thread.currentThread() + ” Signal: Try to aquire lock of x");
21. synchronized(x){
22. System.out.println(Thread.currentThread() + " Signal: Aquired lock of x");
23. try{
24. Thread.sleep((long) (Math.random()*5000));
25. System.out.println(Thread.currentThread() + " Signal: Try to aquire lock of y");
```

```
26. synchronized(y){
27. System.out.println(Thread.currentThread() + " Signal: Aquired lock of y";
28. Thread.sleep((long) (Math.random()*5000));
29. y.notifyAll(); } }
30. catch(InterruptedException e) {
31. e.printStackTrace(); } } }
32. public static void main(String[] args){
33. final Nested n = new Nested();
34. Thread t1 = new Thread(){
35. //this method will be executed parallel
36. public void run(){
37. for (;;) n.waitForSignal(); } };
38. t1.start();
39. Thread t2 = new Thread(){
40. //this method will be executed parallel
41. public void run(){
42. for(;;) n.signal(); } };
43. t2.start();}}
```

Program 3.15: Nested monitor call

In Program 3.15, the first thread in the waitForSignal() method requests the locks on objects x and y. Then it goes to sleep with the wait() method, thereby releasing the lock on object y. A second thread tries to get the lock on object x in the signal() method. However, the object x is still locked by the first thread. The first thread is waiting for the signal from y, which can no longer take place. As a result, a deadlock has arisen.

This situation becomes even more confusing if the lock on one of the objects is requested via a method. Therefore, methods should document the use of locks.

Avoid nested monitor calls

The only way to solve the nested monitor call problem is to avoid it. For example, both methods could be synchronized with the keyword instead of the synchronized blocks within the methods.

This would only request a lock protecting objects x and y from concurrent access. However, making the locks coarser usually reduces the performance of the program. Instead of nested monitor calls, the use of sempahores is a good idea, since they can be released again from any point in the program.

3.5 Power of Synchronization Means

Because the use of semaphores and monitors - depending on the application - is conceivable, the question arises whether the same can be achieved with both concepts, i.e. the power of the two constructs must be considered. To put it more precisely: The question arises whether the same synchronizations can be achieved with monitors as with semaphores. This is the case if a semaphore can be simulated via monitors.

3.5.1 Semaphore Implementation by Monitors

If semaphores can be implemented by monitors, one can conclude that the size M_M of monitors is at least as large as the size M_S of semaphores, i.e. $M_M >= M_S$ Program 3.16 shows that this is possible:

Program 3.16: Implementation of a semaphore by Java monitors ➔

➔ Hence $M_M >= M_S$.

```
public class Semaphore{
   private int value;
   public Semaphore(int value) {
      this. value = value;
   }
   public synchronized void P() {
      value--;
      if (value < 0){
         try{
            wait();
         }catch (InterruptedException e){
            e.printStackTrace();
         } } }
   public synchronized void V()
   {
      value++;
      if (value <= 0) notify();
   }}
```

3.5.2 Monitor Implementation by Semaphore

Conversely, one can conclude that class semaphores are at least as powerful as monitors if monitors can be implemented by semaphores, i.e. $M_S >= M_M$.

A possible implementation can be found in [Ben-Ari 1982] p. 86ff. Therefore $M_S >= M_M$.

From $M_S >= M_M$ and $M_M >= M_S$ => $M_M = M_S$. Since both constructs can be implemented by the other, semaphores and monitors are equal in terms of expressive power.

3.5.3 Importance Overview

At this point, all of the means of synchronization presented in the Synchronization chapter are considered in relation to their power.

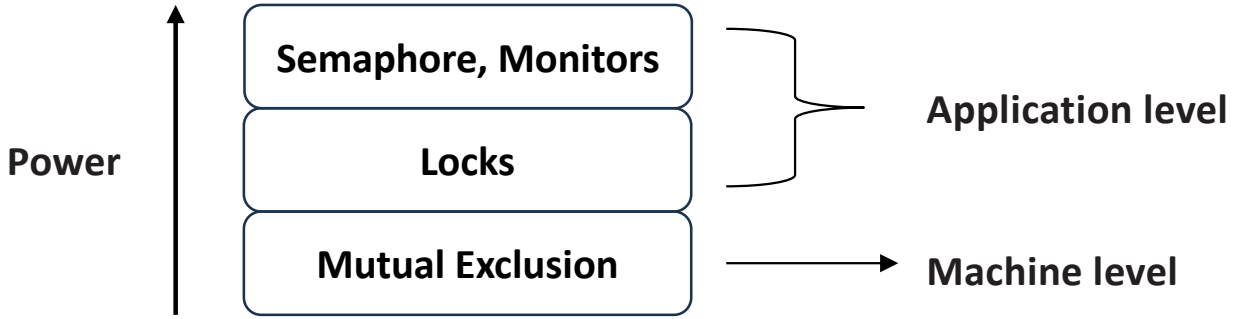


Figure 3.6: Overview of the power of means of synchronization

Although semaphores and monitors are powerful constructs, their functionality is built on mutual exclusion. Both semaphores and monitors use locks, which in turn are based directly on mutual exclusion.

In comparison

In a direct comparison, semaphores can be used more flexibly than monitors, since the signaling does not have to take place within a monitor, but is completely independent. The use of monitors, on the other hand, is usually a little clearer in more complex programs. If it is necessary to nest monitor calls, monitors should not be used.

3.6 Locks in Java

The last part of this chapter looks at locks in the Java programming language. The locks introduced in JDK 1.5 by the java.util.concurrent package can be coupled with condition variables and queues so that by combining these classes both the general monitor concept (as presented in section 3.4) and semaphores can be implemented.

3.6.1 The Interface Lock

In Java, a hole is initially represented by an interface. This interface provides the following methods:

- The `void lock()` method takes the lock in if no other thread holds it. If the lock is held by another thread, the thread trying to own the lock is put to sleep.
- `void lockInterruptibly()` works the same as lock() with `lockIntenuptibly()` except that the waiting thread can be interrupted.
- With the `boolean tryLock()` method, the calling `tryLock()` thread tries to take ownership of the lock. If the thread succeeds, the method returns true, otherwise false.

- With the boolean tryLock(long time. TimeUnit unit) method, the calling thread tries to take possession of the lock within the specified time. If the lock is available within the specified time, the method returns with the value true. Once the time is up, the method returns with a value of false. The enum TimeUnit specifies the time unit used. Four units of time are distinguished by using four constants, as shown in Table 3.4.

Time Unit	Constant
Second (s)	TimeUnit.SECONDS
Millisecond (ms)	TimeUnit.MILLISECONDS
Microsecond (µs)	TimeUnit.MICROSECONDS
Nanosecond (ns)	TimeUnit.NANOSECONDS

Table 3.4: Time units

- A thread uses the void unlock() method to release the lock it is holding.
- Condition newCondition() returns a condition variable (a Condition object) bound to the lock.

The two tryLock() methods are very well suited to avoiding deadlocks, expecially if locks already held by the thread are released again after an unsuccessful tryLock( ) call. A timeout is also a good way to avoid deadlocks. If after a period of time the lock has not been acquired, the thread can respond by releasing its held locks or by attempting to acquire the lock again. A lock implementation that will be discussed in more detail later is the ReentrantLock class.

3.6.2 Condition Variables

As seen, it is possible to have a lock create condition variables associated with that lock. A condition variable is represented by the interface Condition. The default implementation used is the AbstractQueuedSynchronizer.ConditionObject class.

For condition variables, like with semaphores and monitors, there exists a signaling mechanism through the methods await(), awaitUninterruptibly(), signal() and signalAll(). await() and awaitUninterruptibly() are counterparts to wait(), while signal() and signalAll() are counterparts to notify() and notifyAll().

Two parameters can be specified for the await() method in order to wait a certain period of time, as with the tryLock() method. If the method returns true, the lock was successfully acquired. Otherwise the value false is returned.

3.6.3 Example: The ReentrantLock class

There is one implementation of the Lock interface that is particularly worth looking at, the ReentrantLock class. The ReentrantLock provides two constructors, the default constructor and the constructor ReentrantLock(boolean fair). If the ReentrantLock object is created by ReentrantLock(false) or by the default constructor, fairness to other threads is ignored. Therefore, a thread can starve if other threads in the queue trying to get a lock ahead of it. If the constructor ReentrantLock(true) is used, the thread from the queue that has been waiting the longest gets the lock next. In addition to the methods of the Lock interface, this class also provides a number of other useful methods:

- int getHoldCount() returns the number of times the lock was held by the calling thread.
- Thread getOwner() indicates which thread is currently holding the lock.
- boolean getQueuedThread(Thread thread) returns true if thread is in the queue, false otherwise.
- boolean getQueuedThreads() returns true if any thread is waiting for the lock.
- int getQueueLength () returns the number of threads waiting for the lock. However, the number may change while the method is being executed, so this value can only be considered an estimate.
- boolean isLocked() returns true if any thread is locked, false otherwise.
- boolean isFair() returns true if the value true was passed to the ReentrantLock object in the constructor and is therefore fair.
- boolean isHeldByCurrentThread() returns true if the current thread is holding the lock, false otherwise. This method is used for debuggers.

Program 3.17 shows a simple example of using locks.

```
import java.util.concurrent.locks.*;
public class Locks implements Runnable {
   Thread t;
   public Locks() {
      t = new Thread(this);
      t.start() ;
   }
   public static void main(String[] args) {
      Locks l1 = new Locks();
      Locks l2 = new Locks();
   }
      public void run(){
         ReentrantLock r = new ReentrantLock();
         try{
            for(int i = 0; i < 100; i++){
            System.out.println(t.getName() +" try to acquire lock");
            r.lock();
            System.out.println(t.getName() + " Acquired lock. Working.");
            Thread.sleep((long)(Math.random() * 4000));
            System.out.println(t.getName() + " Work done. Releasing lock.");
            r.unlock();}
         } catch(InterruptedException e) {
            e.printStackTrace();}}}
```

Due to the possibility of coupling queues, condition variables and locks, semaphores and monitors can be modeled.

After the essential basics for the design of complex software have been clarified up to this point, the following chapters will deal with the development of distributed applications.