

BERUFSAKADEMIE MANNHEIM  
Information Technology International

Lecture Notes

Programming I

Part 6 Threads, Concurrency, and Synchronization

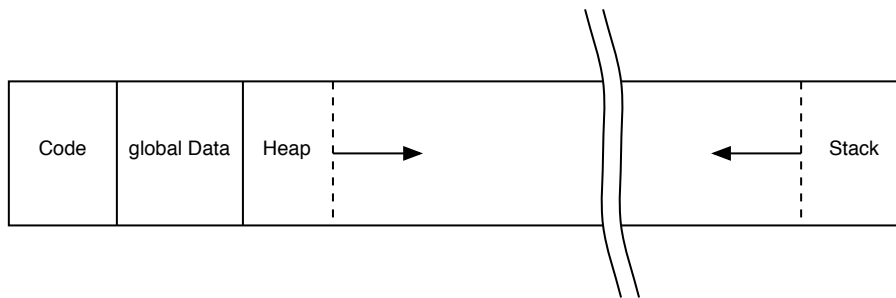
Dipl.-Betriebswirt (BA) VOLKAN YAVUZ  
{xmachina GmbH

November 2001–February 2002  
Classes: TIM01AGR && TIT01EGR  
Term: 1

Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	What are Threads . . . . .	2
1.2	Terminology . . . . .	2
1.3	Scheduling and Thread Priority . . . . .	3
<b>2</b>	<b>Using Threads in Java</b>	<b>5</b>
2.1	Subclassing Thread and Overriding run . . . . .	5
2.2	Implementing the Runnable Interface . . . . .	6
2.3	Additional Thread-Classes in Java 1.3 . . . . .	7
2.4	States of a Thread . . . . .	7
2.5	Daemon Threads . . . . .	8
2.6	Synchronization, Threads, and Locks . . . . .	8
<b>3</b>	<b>Concurrency and Race Conditions</b>	<b>10</b>
3.1	Introduction . . . . .	10
3.2	The Lost Update Problem . . . . .	11
3.3	The Uncommitted Dependency Problem . . . . .	12
3.4	The Inconsistent Analysis Problem . . . . .	12
3.5	The Producer-Consumer Problem . . . . .	13
3.5.1	First Implementation . . . . .	13
3.5.2	Second Implementation . . . . .	14
3.5.3	Third Implementation . . . . .	15
3.5.4	Analysis . . . . .	16
3.6	Detecting Deadlocks . . . . .	16
3.7	The Dining Philosophers Problem . . . . .	16
3.7.1	First Implementation . . . . .	17
3.7.2	Second Implementation . . . . .	18
3.7.3	Third Implementation . . . . .	19

**Figure 1:** Typical Memory Layout



**4 Using Threads with Swing** **20**

**References** **22**

**1 Introduction**

**1.1 What are Threads**

The sequential execution of programming statements are already familiar. The CPU (Central Processing Unit) executes one statement after the other, starting at the beginning, performing a number of execution steps and stopping at the end. During the complete runtime of the program, there is always exactly one single point of execution.

A modern multi-tasking OS (Operating System) allows many programs to run in parallel, without affecting each other. These programs are called *processes*. Depending on the robustness of the operating system, these processes do not share any data with each other, and are able to communicate only through explicitly provided mechanisms. Each process has its own code segment, global data, the heap, the runtime stack, etc. as shown in figure 1.

*Threads* are also able to run independently of each other, but unlike processes, threads share the code segment, global data and the heap. Each thread has its own runtime stack and program counter, but shares all resources with all the other threads. Therefore, threads are sometimes also called *lightweight* processes, in contrast to *heavyweight* processes. This is shown in figure 2 on the next page.

**1.2 Terminology**

**Process**

- possesses an environment of its own
- has its own memory (including code, global data, heap and stack)
- IPC (Inter Process Communication) is not trivial

**Thread**

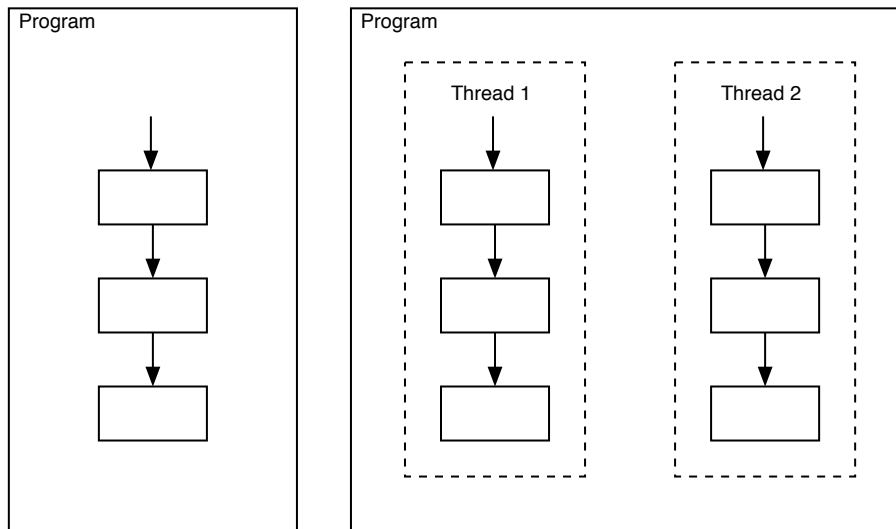
- a distinct flow of execution *within* a process
- *lightweight* process
- requires only very few information:
  - a set of CPU registers
  - an own stack
  - some additional data for administration

**Multithreading:**

several threads execute in parallel within one process.

- Threads are able to run completely independent of each other.

**Figure 2:** Threads



- Access to shared objects is possible.
- Synchronization is necessary.

**Differences**

between processes and threads:

- All threads within a process share the same memory.
- Memory of several processes are separated strictly.

**Typical applications:**

- Implementation of server functionality in C/S (Client/Server) architectures.
- Loading of large media files in the background.
- Performing animation in parallel to user interaction.
- Playing of sound in the background.
- Loading and saving of data in the background.

**1.3 Scheduling and Thread Priority**

Threads allow the programmer to write functions that execute in parallel. But depending on the hardware setup of a given machine, a machine with just one CPU will not be able to execute more than one thread at any given time. It is the responsibility of the *scheduler* to create the *illusion* that *n* threads run in parallel, and to choose which thread to execute actually, while ensuring that all threads get their fair share of CPU-time. This is performed by letting a thread run for a given amount of time, then terminate it and selecting another thread to run. This is depicted in figure 3 on the following page.

Given a set of known facts (like the state of threads, their number, and their running time until now), it has to be decided which thread to run next. Determining which thread to run is called *scheduling*, and many scheduling strategies are known. Some criterions a scheduling policy has to satisfy are [see Tanenbaum, 1995, pg. 78–88]:

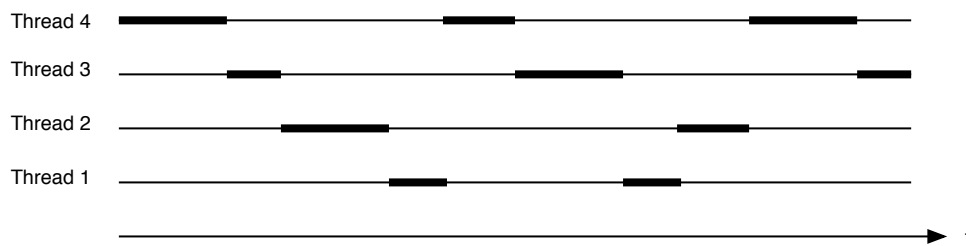
**fairness**

Each thread should be able to get its share of CPU-time.

**efficiency**

The CPU has to be operated at full capacity.

**Figure 3:** Thread Scheduling



<b>response time, latency</b>	Response time for threads that perform user interaction has to be acceptable.
<b>waiting time</b>	The total time for completion of a thread should be minimal.
<b>throughput</b>	The number of threads that are running in a given interval of time should be maximized.

Typically used scheduling policies are

**Run-to Completion** Belongig to the class of *non-preemptive* scheduling policies allows a thread to complete bevor another thread is chosen for execution. In effect, this policy corresponds to batch system, where each job (thread) is performed until completion before the next job is started.

**Round-Robin Scheduling** The oldest and easiest scheduling algorithm executes each thread in turn. Whenever a given amount of time has elapsed, the currently running thread is suspended, and the next one is selected.

**Priority Scheduling** All threads are assigned a priority, and the thread with the highest priority is executed. If there is more than one thread with equall priority, one of them might be chose in a round-robin fashion. Alternatively, the priority of an already executed thread might be decreased to prevent it from taking too much CPU-time.

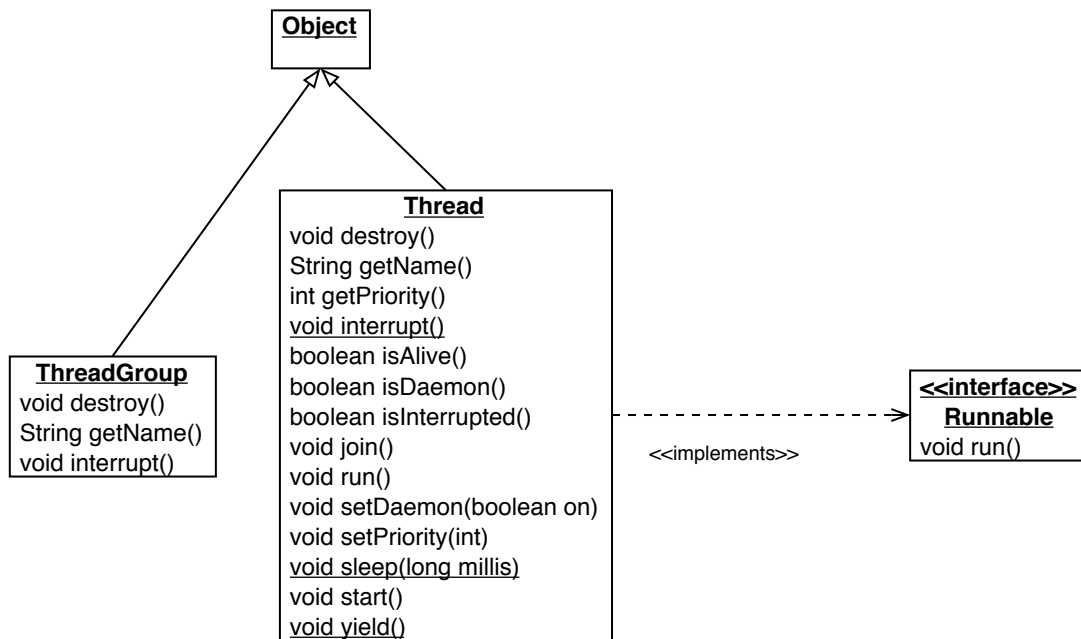
**Shortest Job First** This scheduling policy is not suited for interactive systems, but instead used in batch processing installations. Jobs are performed in order of increasing running length, so the shorter a job is, the shorter the running time until completion. This policy also assumes that the normal running time of a job is known prior to submission.

The Java runtime implements a very simple, deterministic scheduling algorithm known as *fixed priority scheduling* . This algorithm schedules threads based on their *priority* relative to other runnable threads.

When a Java thread is created, it inherits its priority from the thread that created it. Modification can be done through the ‘`setPriority`’ method. Valid values are integers between ‘`MIN_PRIORITY`’ and ‘`MAX_PRIORITY`’. At any given time, the thread with the highest priority is chosen for execution. If more than one thread have the same priority, one of them is chosen in a *round-robin* fashiong. Only when a thread stops, yiels, or becomes not runnable for some reason will a lower priority thread start executing. The chosen thread will run until one of the following conditions is true:

- A higher priority thread becomes available. This makes the scheduler *preemptive*, as a newly runnable high-priority threads *preempts* already running, lower-priority threads.
- It yields, or its run method exits.
- On systems that support *time-slicing*, its time allotment has expired.

Figure 4: Thread Classes



**Note:** Given the scheduling policy, some assumptions can be made about the behaviour of many threads than run with certain priorities. But priority of threads and the used scheduling policy should *under no circumstances* be relied upon to ensure algorithmic correctness.

**Definition 1 (Schedule)** The sequence of statements executed by a set of thready running in parallel taken together is known as a *schedule*. □

## 2 Using Threads in Java

A threads work is performed in the ‘run’ method. To implement a thread, two possibilities exist:

- Subclassing ‘Thread’ and overriding the ‘run’ method
- Implementing the ‘Runnable’ interface

As a rule of thumb, whenever the class must subclass some other class (e.g. ‘Applet’ or ‘JFrame’, implementing ‘Runnable’ is the only alternative, as Java does not support multiple inheritance. The only reason for subclassing ‘Thread’ directly lies in the easier usage.

Classes involved in supporting the usage of threads are shown in figure 4.

### 2.1 Subclassing Thread and Overriding run

Listing 1: Threads Example Subclassing

```

1 class SimpleThread extends Thread
2 {
3     // The superclass constructor is called with this threads name
4     public SimpleThread(String name) {
5         super(name);
6     }
    
```

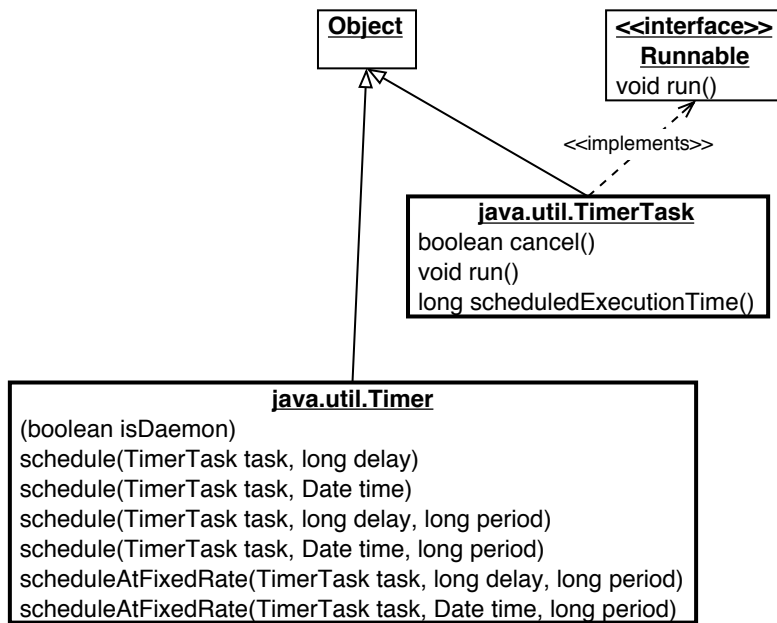
```
7
8 // The run method performs all the work. After ten iterations of
9 // the for-loop below, the thread is terminated.
10 public void run() {
11     for (int i = 0; i < 10; i++) {
12         System.out.println(i + " " + getName());
13         try {
14             sleep((long)(Math.random() * 1000));
15         } catch (InterruptedException e) {}
16     }
17     System.out.println("DONE! " + getName());
18 }
19 }
20
21 public class ManyThreadsDemo
22 {
23     public static void main(String[] argv) {
24         // Three new threads are created and each one is started
25         new SimpleThread("Jamaica").start();
26         new SimpleThread("Mexico").start();
27         new SimpleThread("New Zealand").start();
28     }
29 }
```

## 2.2 Implementing the Runnable Interface

Listing 2: Threads Example Subclassing

```
1 class Runner implements Runnable
2 {
3     private String name;
4     private Thread thread;
5
6     public Runner(String name) {
7         this.name = name;
8         this.thread = null;
9     }
10
11 // Create a thread associated with this runnable and start it.
12 public void start() {
13     if (this.thread == null) {
14         this.thread = new Thread(this, this.name);
15         this.thread.start();
16     }
17 }
18
19 // A Runnable's run method performs all the work. After ten
20 // iterations of the for-loop below, the thread is terminated.
21 public void run() {
22     for (int i = 0; i < 10; i++) {
23         System.out.println(i + " " + this.name);
24         try {
25             this.thread.sleep((long)(Math.random() * 1000));
26         } catch (InterruptedException e) {}
27     }
28     System.out.println("DONE! " + this.name);
29 }
30 }
```

Figure 5: Thread Classes available in 1.3



```

31
32 public class ManyThreadsDemo
33 {
34     public static void main(String[] argv) {
35         // Three new threads are created, each one is passed a
36         // Runnable, and then started.
37         new Runner("Jamaica").start();
38         new Runner("Mexico").start();
39         new Runner("New Zealand").start();
40     }
41 }
    
```

### 2.3 Additional Thread-Classes in Java 1.3

See figure 5.

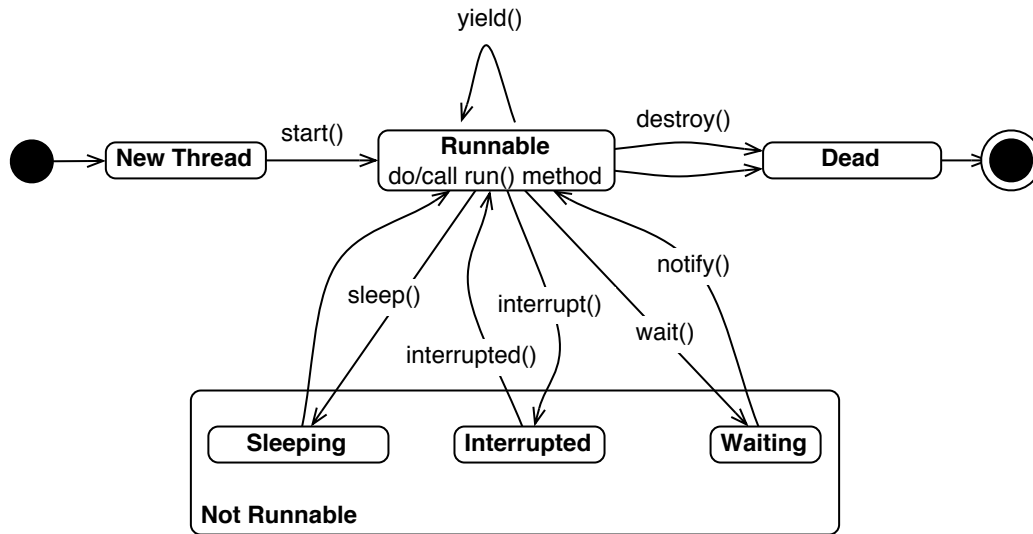
### 2.4 States of a Thread

The lifecycle of a thread is shown in figure 6 on the next page. Upon creation, a thread is in the **New Thread** state. The only method that can validly be called upon a thread in that state is the **start** method. The thread is then in the **Runnable** state, i. e. it is eligible for the scheduler, and it may execute at any time whenever the scheduler decides so. When the thread terminates normally, i. e. when its **run** method terminates, the thread changes to the **Dead** state, where it will remain until it is destroyed. A thread might also be destroyed explicitly by calling **destroy** on it.

A thread might also leave the **Runnable** state by calling one of the methods below.

- **yield** causes the currently executing thread to temporarily pause and allow other threads to execute.
- **sleep** causes the currently executing thread to sleep for a specified number of milliseconds.
- **interrupt** causes the thread to be interrupted. It changes to **Runnable** state if the **interrupted** method is called.

Figure 6: Thread States



- ‘wait’ causes this thread to wait until another thread invokes the ‘notify’ method or the ‘notifyAll’ method for this object. ‘wait’ is usually used to *synchronize* concurrent access to shared resources.

## 2.5 Daemon Threads

The JVM (Java Virtual Machine) will stop execution as soon as all threads (including the ‘main’ thread) have finished. *Daemon threads* are used to perform auxiliary operations for other threads in the background. The AWT (Abstract Window Toolkit)-threads and the garbage collector are implemented as daemon threads. Unlike for “normal” threads, the JVM will not wait for daemon threads to finish execution. A thread can be marked as daemon by calling ‘setDaemon(true)’ *prior* to calling the ‘start()’ method.

## 2.6 Synchronization, Threads, and Locks

The Java programming language provides mechanisms for *synchronizing* the concurrent activities of threads by means of *monitors*.

The ‘synchronized’ statement or modifier is associated both with a block of statements and a lock. The following actions take place whenever a ‘synchronized’ block is about to be executed:

1. First, the object to be *locked* is determined and a lock is associated with that object.
2. Then, the block of statements is executed.
3. Finally, after completion of the block of statements (either normally or abruptly), the lock created in the first step is released.

Shown below is the syntax of using the ‘synchronized’ keyword:

```

1 <ClassModifiers>opt class <ClassIdentifier> <Super>opt <Interfaces>opt
2 {
3     ....
4     <MethodModifiers>opt synchronizedopt <MethodDeclaration>
5     {
6         ....
    
```

```
7     synchronized ( <Expression> ) {
8         <Block>
9     }
10    ....
11 }
12    ....
13 }
```

‘synchronized’ can either be used as a method modifier or as a statement. When used as a method modifier, all statements of the method are synchronized, at the lock is associated either with the instance (for instance methods) or with the class (for static methods). If the ‘synchronized’ statement is used, the lock is associated with the expression which must be of reference type. Thus, the following two examples are equivalent:

```
14 class Test
15 {
16     int count;
17     synchronized void bump() {
18         count++;
19     }
20
21     static int classCount;
22     static synchronized void classBump() {
23         classCount++;
24     }
25 }
26
27 class Test
28 {
29     int count;
30     void bump() {
31         synchronized(this) {
32             count++;
33         }
34     }
35
36     static int classCount;
37     static void classBump() {
38         try {
39             synchronized (Class.forName("Test")) {
40                 classCount++;
41             }
42         } catch (ClassNotFoundException e) {
43             ...
44         }
45     }
46 }
```

The ‘wait’, ‘notify’, and ‘notifyAll’ methods provide efficient transfer of control from one thread to another. A thread that is ‘wait’ing will be suspended until another thread will wake it up by calling either ‘notify’ or ‘notifyAll’.

**Note:** As actual parameters to methods and local variables are private to a thread, access to those variables has not to be synchronized.

**Note:** Neither does Java perform deadlock detection nor does it require prevention of deadlocks. Appropriate measures have to be taken to prevent deadlock from occurring.

**Note:** A thread that has already acquired a lock on an object may claim that lock more than once. Thus, the following example is valid and will *not* result in a deadlock situation at the second 'synchronized' block:

```
47 class Test
48 {
49     public static void main(String[] args)
50     {
51         Test t = new Test();
52         synchronized(t) {
53             synchronized(t) {
54                 System.out.println("made it!");
55             }
56         }
57     }
58 }
```

### 3 Concurrency and Race Conditions

#### 3.1 Introduction

Besides from their own private data, threads (legally) share resources (e. g. variables accessible to other threads as well) with each other. Conceptually, threads may have different relationships between each other:

**producer-consumer** In this kind of relationship, threads actively cooperate in achieving a common goal. For example, one thread might produce data that another thread is about to consume and use in further computations.

**mutual exclusion** In this kind of relationship, several threads operate on shared data independently of each other. Synchronization is only used to prevent conflicts and corruption of data.

**unrelated** In this kind of relationship, no resources are shared, thus synchronization is not an issue.

**Note:** Whenever there is a resource that is accessed by more than one thread (shared resource), *race conditions* might occur. A race condition is a timing dependent error involving shared resources as the error may or may not occur depending on how the threads have been scheduled.

Race conditions can be prevented by inhibiting that more than one thread executes a critical section at the same time, thus *mutually excluding*<sup>1</sup> them from accessing the shared resources by synchronization. Nevertheless it is very hard since all possible schedules have to be safe. The number of possible schedule permutations is huge.

**Definition 2 (Lock)** A *lock* is a resource that can only be held by one thread at a time. If a lock is acquired by one thread, other threads have to wait until the first thread releases the lock. □

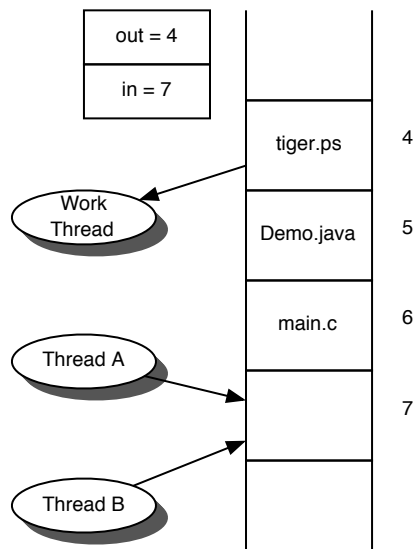
**Definition 3 (Synchronization)** Using atomic operations to ensure cooperation between threads. □

**Definition 4 (Mutual Exclusion)** Ensuring that only one thread is doing a particular thing at a time. One thread doing it excludes the other, and vice versa. □

**Definition 5 (Critical Section)** The pieces of code that access a shared resource are called *critical sections*. □

<sup>1</sup> Wechselseitiger Ausschluß

Figure 7: A Print Spool



**Definition 6 (atomicity, atomic unit)** An *atomic unit* is a sequence of instructions that is guaranteed to execute indivisibly. No two threads are allowed to execute an atomic unit at the same time. If one thread has already entered that atomic unit (*critical section*), others threads that were about to enter it, too, are suspended until the first thread currently in that atomic unit has left the critical section. □

Debugging programs that contain race conditions is very difficult, as most of the time, no symptoms are visible, and very seldom, unexplainable behaviour can be observed. The results are also said to be *timing-dependent*.

### 3.2 The Lost Update Problem

Consider a printer spool which can be used to send the contents of files to a printer. As a printer is only capable of printing one file at a time (otherwise the output would be corrupted), the access to the printer is *synchronized* through a printer spool. That printer spool provides an array of ‘String’ objects, and treats the ‘String’ objects stored there to be names of files to be printed. Additionally, the integer variables ‘in’ and ‘out’ are provided, which point to the next free position and the position that will be printed next, respectively. Positions less than 4 are empty, as they were occupied with files that have already been printed. The next free position is 7. This is shown in figure 7.

Now assume that two threads want to print a file. First, *Thread A* reads the ‘in’ variable to determine the next free array position. At this moment, the scheduler terminates *Thread A*, and continues with *Thread B*, also accessing ‘in’. *Thread B* stores its filename in position 7 of the array, increments ‘in’ to 8 and continues. Then, execution continues with *Thread A*, which still has the already read value of 7, stores its own filename in position 7 (thus overwriting the value stored by *Thread B*, and also increments ‘in’ to 8 and continues. The *schedule* is shown in figure 8 on the next page

The file to be printed for *Thread B* will never be printed, nor will the printer spool determine an error condition, since the spool is in a consistent state. What happened is the so called *lost-update problem*, and problems of this kind are caused by *race conditions*<sup>2</sup> [see Connolly et al., 1996, pp. 583–586, 652].

<sup>2</sup> Zeitkritischer Ablauf

**Figure 8:** The Lost Update Problem.

Time	Thread <sub>A</sub>	i <sub>A</sub>	Thread <sub>B</sub>	i <sub>B</sub>	in
t <sub>1</sub>			do something		7
t <sub>2</sub>	do something				7
t <sub>3</sub>	read in → i	7			7
t <sub>4</sub>		7	read in → i	7	7
t <sub>5</sub>		7	write "b" → spool[i]	7	7
t <sub>6</sub>	write "a" → spool[i]	7		7	7
t <sub>7</sub>	i += 1	8		7	7
t <sub>8</sub>	write i → in	8		7	8
t <sub>9</sub>		8	i += 1	8	8
t <sub>10</sub>		8	write i → in	8	8
t <sub>11</sub>		8	do something	8	8
t <sub>12</sub>	do something	8			8

**Figure 9:** The Uncommitted Dependency Problem.

Time	Thread <sub>A</sub>	n <sub>A</sub>	Thread <sub>B</sub>	n <sub>B</sub>	next_number
t <sub>1</sub>			do something		9
t <sub>2</sub>			read next_number → n	9	9
t <sub>3</sub>			n++	10	9
t <sub>4</sub>			write n → next_number	10	10
t <sub>5</sub>	do something			10	10
t <sub>6</sub>	read next_number → n	10		10	10
t <sub>7</sub>		10	throw Exception	10	10
t <sub>8</sub>	n += 2	12		10	10
t <sub>9</sub>		12	catch Exception	10	10
t <sub>10</sub>		12	n--	9	10
t <sub>11</sub>		12	write n → next_number	9	9
t <sub>12</sub>		12	do something		9
t <sub>13</sub>	write n → next_number	12			12
t <sub>14</sub>	do something				12

### 3.3 The Uncommitted Dependency Problem

The *uncommitted dependency* problem is caused by one thread seeing the intermediate results of another thread that will be revoked (e. g. because of an exception handler undoing some actions). The problem is shown in figure 9.

After T<sub>2</sub> has written the new value for 'next\_number', T<sub>1</sub> reads it. But T<sub>2</sub> then catches an exception. The changes made by T<sub>2</sub> are being undone, but T<sub>1</sub> already has used the values and continues. The problem can be avoided by preventing T<sub>1</sub> from reading 'next\_number' until T<sub>2</sub> has completed successfully.

### 3.4 The Inconsistent Analysis Problem

The *inconsistent analysis* problem can occur in threads that are only reading data. When they are allowed to read intermediate results of other executing threads, they may yield incorrect results. See figure 10 on the next page for a schedule.

**Figure 10:** The Inconsistent Analysis Problem.

Time	Thread <sub>A</sub>	Thread <sub>B</sub>	d <sub>B</sub>	drafts	finals
t <sub>1</sub>		do something		10	30
t <sub>2</sub>		read drafts → d	10	10	30
t <sub>3</sub>	do something		10	10	30
t <sub>4</sub>	drafts--		10	9	30
t <sub>5</sub>	finals++		10	9	31
t <sub>6</sub>		d += finals	41	9	31
t <sub>7</sub>		do something	41	9	31
t <sub>8</sub>	do something			9	31

T<sub>2</sub> has to read the values ‘drafts’ (which contains the number of draft documents) and ‘finals’ (number of final version documents) to get the total number of documents. Simultaneously, T<sub>1</sub> decrements the number of ‘drafts’ by one and increases the number of ‘finals’ (a document has reached its final version). When T<sub>2</sub> reads the data at certain points in time, wrong results may be achieved.

### 3.5 The Producer-Consumer Problem

Assume another typical *producer-consumer* problem. A producer generates integer numbers, and puts them in a simple data structure, that is either empty or stores exactly one value. The consumer gets this value from the data structure. Both the producer and the consumer run in different threads. The data structure is shared between the two threads. While the producer is deliberately delayed a random amount of time, the consumer expects values as quickly as possible.

#### 3.5.1 First Implementation

A first implementation is given below:

```
1 public void run() {
2     for (int i = 0; i < 10; i++) {
3         cubbyhole.put(i);
4         try {
5             sleep((int)(Math.random() * 100));
6         } catch (InterruptedException e) { }
7     }
8 }

1 public void run() {
2     int value = 0;
3     for (int i = 0; i < 10; i++) {
4         value = cubbyhole.get();
5         ...
6     }
7 }
```

Implementation of the data structure ‘CubbyHole’ is critical, since it contains the critical sections that have to be synchronized to prevent race conditions:

```
1 class CubbyHole {
2     private int contents;
3     private boolean available = false;
4
5     // synchronized causes a lock on this object
6     // at most one thread may claim a lock on this object
```

```
7     public synchronized void put(int value) {
8         if (available == false) {
9             available = true;
10            contents = value;
11        }
12    }
13
14    // synchronized causes a lock on this object
15    // at most one thread may claim a lock on this object
16    public synchronized int get() {
17        if (available == true) {
18            available = false;
19            return contents;
20        } else
21            return -1;
22    }
23 }
```

Unfortunately, this does not yield the desired effect. Whenever the data structure is empty, the value of '-1' is returned. So, the consumer takes values faster out of the data structure than the producer can generate them. On the other hand, if the consumer were delayed, it could potentially lose values, as the producer were faster.

### 3.5.2 Second Implementation

Given the previous example, it seems as though the 'synchronized' declaration is not enough. The both threads have to be coordinated in a more sophisticated way. The producer has to wait for the consumer to get an eventually stored value, before it can put the newly created value into the data structure. Similarly, the consumer has to wait for a new value to be available before trying to get it. As the 'available' field holds this information, it can be used in this way in an improved version of the data structure:

```
1 // Deadlock
2 class CubbyHole {
3     private int contents;
4     private boolean available = false;
5
6     // synchronized causes a lock on this object
7     // at most one thread may claim a lock on this object
8     public synchronized void put(int value) {
9         System.out.println("CubbyHole.put() entered");
10        while (available == true)
11            ;
12
13        available = true;
14        contents = value;
15        System.out.println("CubbyHole.put exited");
16    }
17
18    // synchronized causes a lock on this object
19    // at most one thread may claim a lock on this object
20    public synchronized int get() {
21        System.out.println("CubbyHole.get() entered");
22        while (available == false)
23            ;
24
25        available = false;
26        System.out.println("CubbyHole.get() exited");
```

```
27     return contents;
28 }
29 }
```

Both the ‘get’ and the ‘put’ method now wait until the necessary condition is met. But unfortunately, this implementation causes a *deadlock*. As soon as the ‘get’ method is entered and the data structure is empty, the ‘get’ method will wait until a new value is available. But a new value can only be made available when the ‘put’ method runs. As both methods are declared ‘synchronized’, upon entry of the ‘put’ method, the data structure is locked. When the producer wants to store a newly created value calling the ‘put’ method, the Java runtime prevents entry of that method, as there is already a lock associated with the data structure.

### 3.5.3 Third Implementation

To prevent the deadlock situation which occurs while waiting for a condition to change, the methods ‘wait’ and ‘notifyAll’ are used:

‘wait’	Causes this thread to wait until another thread invokes the ‘notifyAll’ (or ‘notify’) method. First, all locks acquired for this object are relinquished, and this thread is placed in the wait set for this object. This thread stays <i>Waiting</i> until another thread calls the ‘notifyAll’ method. As soon as this happens, and this thread is chosen for execution, all its locks are restored to the state they were before the call to ‘wait’.
‘notifyAll’	Wakes all threads that are in the wait set for this object. They will not be able to proceed until the current thread relinquishes the currently held locks.

The implementation shown below correctly solves the problem:

```
30 public synchronized void put(int value) {
31     while (available == true) {
32         try {
33             // release all locks from this thread on this object
34             // and enable other threads to execute, and wait to
35             // re-claim a lock on this object
36             wait();
37             // re-claim all locks to the state just before wait();
38         } catch (InterruptedException e) { }
39     }
40     contents = value;
41     available = true;
42     // wake up all threads that are waiting for a lock on this
43     // object
44     notifyAll();
45 }
46
47 // synchronized causes a lock on this object
48 // at most one thread may claim a lock on this object
49 public synchronized int get() {
50     while (available == false) {
51         try {
52             // release all locks from this thread on this object
53             // and enable other threads to execute, and wait to
54             // re-claim a lock on this object
55             wait();
56         } catch (InterruptedException e) { }
```

```

57     }
58     available = false;
59     // wake up all threads that are waiting for a lock on this
60     // object
61     notifyAll();
62     return contents;
63 }
    
```

### 3.5.4 Analysis

Coordination between consumer and producers involved three constraints:

1. The consumer must wait for a value to be available.
2. The consumer must wake up the producer if no value is available.
3. The producer must wait for the consumer to take the value.
4. The producer must wake up the consumer if a value is available.
5. Only one thread is allowed to manipulate the data structure.

The constraints correspond exactly with the synchronization statements used in the third (and correct) implementation. The last constraint is satisfied by the method modifier ‘synchronized’, at the first four constraints are satisfied via the calls to ‘wait’ and ‘notifyAll’ methods.

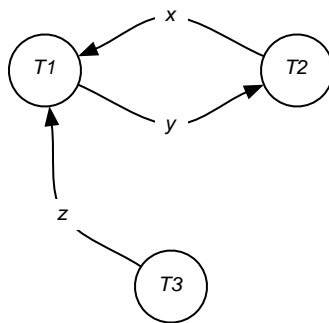
### 3.6 Detecting Deadlocks

Deadlock detection can be performed by constructing a WFG (Wait-for Graph). The wait-for graph is constructed with the following steps:

1. Create a node for each thread
2. Create a directed edge  $T_i \rightarrow T_j$ , if thread  $T_i$  is waiting to lock an item that is currently locked by  $T_j$ .

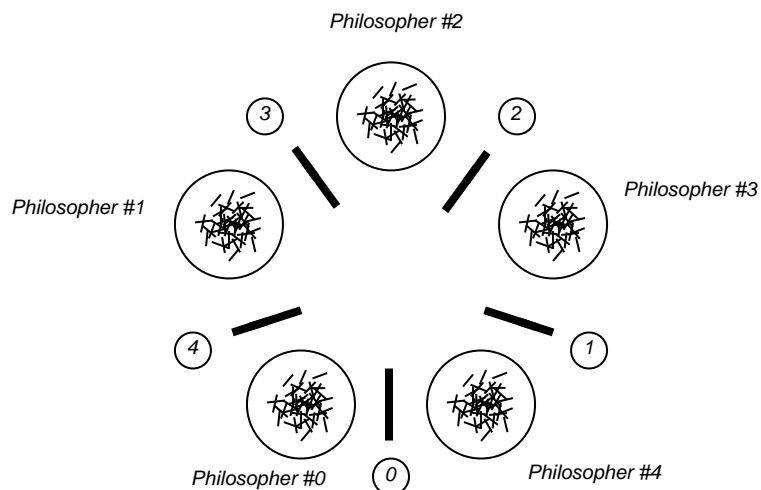
Deadlock exists if and only if the WFG contains a cycle. Figure 11 shows an example for a wait-for graph.

Figure 11: A Wait-for Graph for Deadlock Detection.



### 3.7 The Dining Philosophers Problem

A classical problem to discuss synchronization issues is “The Dining Philosophers Problem”. These five philosopher’s live consists of thinking and eating, especially chinese food. Whenever they get hungry, they grab two chopsticks (from their left and right), and start eating rice until they are sated and continue to think again. Figure 12 on the next page depicts the situation.

**Figure 12:** The Table of the Five Dining Philosophers

### 3.7.1 First Implementation

Class 'Philosopher' models the behaviour of a philosopher in the run method:

```
59 class Philosopher extends AbstractPhilosopher implements Runnable
60 {
61     ...
62     public void run()
63     {
64         try {
65             while (!stopRequested) {
66                 think();
67                 takeLeftChopstick();
68                 ...
69                 takeRightChopstick();
70                 eat();
71                 putLeftChopstick();
72                 ...
73                 putRightChopstick();
74             }
75         } catch (InterruptedException e) {
76             ...
77         }
78     }
79     ...
80 }
```

As the chopsticks are the resources that are accessed concurrently, the implementation of class 'Chopstick' is critical, and synchronization is performed for the two basic methods 'grab' and 'put'. 'grab' will 'wait' if the chopstick is already in use, and 'put' will 'notify' other threads (philosophers) whenever it gets available.

```
81 class Chopstick
82 {
83     ...
84     public synchronized void grab() throws InterruptedException
85     {
86         while (this.taken)
```

```

87     wait();
88     this.taken = true;
89 }
90
91 public synchronized void put()
92 {
93     this.taken = false;
94     notify();
95 }
96 }
    
```

The complete implementation is contained in ‘threads05-Dining’, and a typical run resulting in a deadlock situation might produce the following output.

```

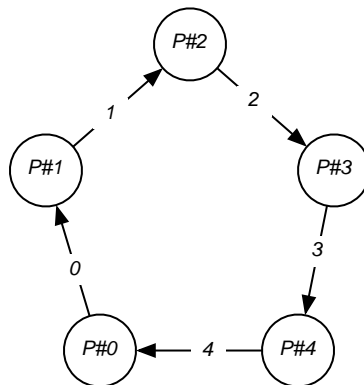
97     0. 1. 2. 3. 4. 3L 4L 0L 2L 1L 1R 0R 3R 2R 4R
    
```

The output ‘3L 4L 0L 2L 1L’ (each philosopher grabs the chopstick to its left) results in the following resource allocation:

- Philosopher #3 grabs chopstick #2
- Philosopher #4 grabs chopstick #3
- Philosopher #0 grabs chopstick #4
- Philosopher #2 grabs chopstick #1
- Philosopher #1 grabs chopstick #0

No resources are left. Then, each philosopher is reaching for the chopstick on its right (‘1R 0R 3R 2R 4R’). But these are taken, already, resulting for each philosopher to wait until that chopstick becomes available again, effectively the philosophers are waiting for each other. Constructing the WFG (see figure 13) visualizes the deadlock situation, as a cycle is contained.

**Figure 13:** A Wait-for Graph for a Deadlock Situation.



### 3.7.2 Second Implementation

Deadlock can be prevented via a slight modification when taking the chopsticks. First, a philosopher will take the chopstick on the left of him, and then will only take the chopstick on his right if it is available. This has to be implemented in a ‘synchronized’ method to avoid introducing a new race condition. If the right chopstick was not available, the left chopstick will be released again.

Shown below is the newly created ‘grabIfAvailable’ method. Note that this method will not wait until the chopstick becomes available.

```
98 class Chopstick
99     ...
100     public synchronized boolean grabIfAvailable()
101     {
102         if (this.taken) {
103             return false;
104         }
105         else {
106             this.taken = true;
107             return true;
108         }
109     }
110     ...
111 }
```

Shown below is the modified 'run' method:

```
112 class Philospher
113     ...
114     public void run()
115     {
116         try {
117             while (!stopRequested) {
118                 think();
119                 takeLeftChopstick();
120                 reportHungryCount++;
121
122                 if (takeRightChopstickIfAvailable()) {
123                     eat();
124                     reportEatCount++;
125                     putLeftChopstick();
126                     putRightChopstick();
127                 } else {
128                     putLeftChopstick();
129                     reportDisappointedCount++;
130                 }
131             }
132         } catch (InterruptedException e) {
133             System.out.print("\n#" + this.number + " IRQ!");
134             stopRequested = true;
135         }
136     }
137     ...
138 }
```

Although this implementation will not cause a deadlock situation, *starvation*, when a process is waiting indefinitely for a resource, might occur. When all philosophers take the chopsticks on their left, and finding that the chopstick on their right is already taken, they will release the already taken chopstick, again waiting for some time.

### 3.7.3 Third Implementation

A final and correct implementation of the 'run' method is given below. The only difference is that philosopher number 0 will first reach for the right chopstick, then the left chopstick. All other philosophers will still first reach for the left, then the right chopstick. This strategy does neither cause deadlock nor starvation:

```
139 class Philospher
140     ...
141     public void run()
```

```
142     {
143         try {
144             while (!stopRequested) {
145                 think();
146                 if (this.number == 0) {
147                     takeLeftChopstick();
148                     delay();
149                     takeRightChopstick();
150                 } else {
151                     takeRightChopstick();
152                     delay();
153                     takeLeftChopstick();
154                 }
155                 eat();
156                 reportEatCount++;
157                 putLeftChopstick();
158                 delay();
159                 putRightChopstick();
160             }
161         } catch (InterruptedException e) {
162             System.out.print("\n# " + this.number + " IRQ!");
163             stopRequested = true;
164         }
165     }
166     ...
167 }
```

#### 4 Using Threads with Swing

Using threads with Swing involves some additional precautions to be taken to ensure that an already constructed GUI (Graphical User Interface) will remain in a valid state. As long as the GUI is only modified initially and its state is only updated in the event-dispatching thread, Swing ensures proper synchronization. But if the GUI state has to be updated in response to non-standard events (such as those needed to perform animations), some rules have to be followed.

The class `javax.swing.SwingUtilities` provides two methods that ensure proper usage of threads that work with Swing components, namely `invokeAndWait` and `invokeLater`. Both methods schedule some code to be executed in the event-dispatching thread of Swing and thus ensure proper synchronization.

A higher-level class that supports creating animations is `javax.swing.Timer`. See also section “How to use Threads” in “Creating a GUI with JFC/Swing” [Walrath and Campione, 1999].

A typical example is given in project `threads06-Swing`: The `MouseListener` will recognize mouse click events, and for each event, will create an instance of `CountDown`.

```
168 class DrawPanel extends JPanel {
169     ...
170     public DrawPanel(Draw controller) {
171         ...
172         // disable layout manager
173         setLayout(null);
174         ...
175         addMouseListener(new MouseAdapter() {
176             public void mousePressed(MouseEvent e) {
177                 int x = e.getX();
178                 int y = e.getY();
179                 if (point == null) {
```

```
180         point = new Point(x, y);
181     } else {
182         point.x = x;
183         point.y = y;
184     }
185     createCountDown(point);
186     repaint();
187 }
188 });
189 }
```

A 'CountDown' is associated with two other objects. One 'JLabel' that represents its state in the GUI, and a timer that fires an event every 1000 milliseconds, thus calling 'actionPerformed'.

```
190 private void createCountDown(Point point)
191 {
192     System.out.println("Timer started");
193
194     Insets insets = getInsets();
195     JLabel label = new JLabel();
196     label.setBounds(point.x + insets.left, point.y+insets.top, 75, 20);
197     add(label);
198
199     countdown = new Countdown(this, label);
200     Timer timer = new Timer(1000, countdown);
201     timer.setInitialDelay(10);
202     timer.setCoalesce(true);
203     timer.start();
204
205     countdown.setTimer(timer);
206 }
207 ...
208 }
```

As 'CountDown' implements 'ActionListener', it has to implement 'actionPerformed'. Every 1000 milliseconds, that method is called. After the counter has been decremented, 'repaint' is called on the labels container, namely 'DrawPanel'. If the countdown is completed, the timer will be stopped and the label will be removed from its container. Finally, all unneeded references are set to 'null'.

```
209 class Countdown implements ActionListener
210 {
211     ...
212     public void actionPerformed(ActionEvent e)
213     {
214         counter--;
215         if (counter < 0) {
216             timer.stop();
217             System.out.println("Timer stopped");
218             timer = null;
219             parent.remove(label);
220             label = null;
221             parent.repaint();
222             parent = null;
223         } else {
224             label.setText(": " + counter + ":");
225             parent.repaint();
226         }
227     }
228 }
```

## List of Tables

## List of Figures

1	Typical Memory Layout . . . . .	2
2	Threads . . . . .	3
3	Thread Scheduling . . . . .	4
4	Thread Classes . . . . .	5
5	Thread Classes available in 1.3 . . . . .	7
6	Thread States . . . . .	8
7	A Print Spool . . . . .	11
8	The Lost Update Problem. . . . .	12
9	The Uncommitted Dependency Problem. . . . .	12
10	The Inconsistent Analysis Problem. . . . .	13
11	A Wait-for Graph for Deadlock Detection. . . . .	16
12	The Table of the Five Dining Philosophers . . . . .	17
13	A Wait-for Graph for a Deadlock Situation. . . . .	18

## List of Acronyms

AWT.....	Abstract Window Toolkit
C/S.....	Client/Server
CPU.....	Central Processing Unit
GUI.....	Graphical User Interface
IPC.....	Inter Process Communication
JVM.....	Java Virtual Machine
OS.....	Operating System
WFG.....	Wait-for Graph

## References

- Thomas M. Connolly, Carolyn E. Begg, and Anne D. Strachan. *Database Systems*. International Computer Science. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1996. ISBN 0-201-42277-8.
- Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Carl Hanser Verlag, München; Wien, 2 edition, 1995. ISBN 3-446-18402-3.
- Katy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs (With CD-ROM)*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, July 1999. ISBN 0-201-43321-4. URL <http://web2.java.sun.com/docs/books/swing/>.