Concurrent Programming with Java Threads

Almost all computer systems on the market today have more than one CPU, typically in the form of a multi-core processor. The benefits of such systems are evident: the CPUs can share the workload amongst themselves by working on different instructions in parallel, making the overall system faster. This work sharing is unproblematic if the concurrently executing instructions are completely independent of each other. However, sometimes they need to access the same region of memory or other computing resources, which can lead to so-called *race conditions* where the result of a computation depends on the order of nondeterministic system events. Therefore concurrent processes have to be properly *synchronized*, i.e. programmed to wait for each other whenever necessary, and this calls for specialized programming techniques.

Today, you will learn about the background and techniques of *concurrent programming*. In particular, you will get to know the thread library approach to concurrent programming using the example of the *Java Threads* API. You might be familiar with Java Threads through other courses or previous self-study, in which case you should use this material to review your knowledge. At the end of this lesson, you will be able to

- explain the basics of concurrent execution of processes in modern operating systems, in particular multiprocessing and multitasking,
- understand some of the most important problems related to concurrent programming, in particular race conditions and deadlocks,
- distinguish between different types of process synchronization, in particular mutual exclusion and condition synchronization,
- understand how these types of synchronization are realized in Java Threads,
- program simple concurrent programs using Java Threads.

The lesson consists entirely of self-study material, which you should work through in the usual two lecture hours. You should have a study partner with whom you can discuss what you have learned. A the end of each study section there will be exercises that help you test your knowledge; solutions to the exercises can be found on the last pages of the document.

1 Concurrent execution

This section introduces the notion of concurrency in the context of operating systems. This is also where the idea of concurrent computation has become relevant first, and as we all have to deal with operating systems on a daily basis, it also provides a good intuition for the problem. You may know some of this content already from an operating systems class, in which case

you should see this as a review and check that you are familiar again with all the relevant terminology.

1.1 Multiprocessing and multitasking

Up until a few years ago, building computers with multiple CPUs (Central Processing Units) was almost exclusively done for high-end systems or supercomputers. Nowadays, most end-user computers have more than one CPU in the form of a multi-core processor (for simplicity, we use the term CPU also to denote a processor core). In Figure 1 you see a system with two CPUs, each of which handles one process.

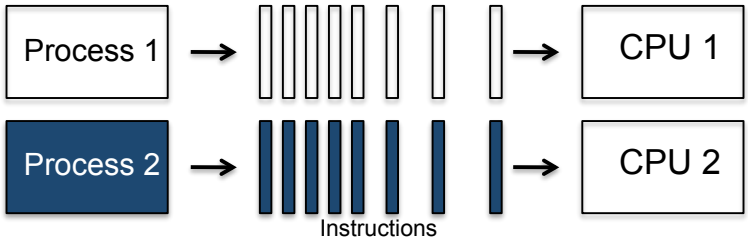


Figure 1: Multiprocessing: instructions are executed in parallel

The situation where more than one CPU is used in a single system is known as *multiprocessing*. The processes are said to execute *in parallel* as they are running at the same time.

However, also if you have a computer with a single CPU, you may still have the impression that programs run "in parallel". This is because the operating system implements *multitasking*, i.e. makes a single CPU appear to work at different tasks at once by switching quickly between them. In this case we say that the execution of processes is *interleaved* as only one process is running at a time. This situation is depicted in Figure 2. Of course, multitasking is also done on multiprocessing systems, where it makes sense as soon as the number of processes is larger than the number of available CPUs.

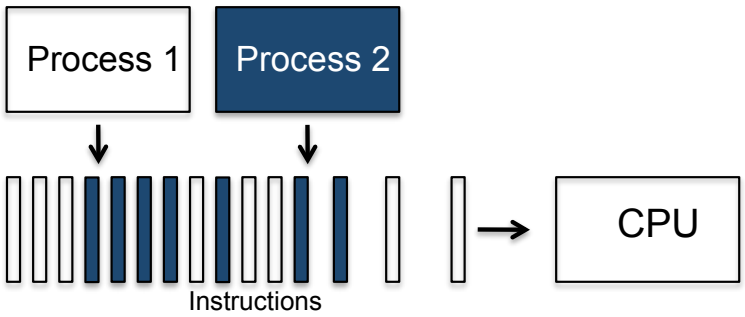


Figure 2: Multitasking: instructions are interleaved

Both multiprocessing and multitasking are examples of *concurrent execution*. In general, we say that the execution of processes is *concurrent* if they execute either truly in parallel or interleaved. To be able to reason about concurrent executions, one often takes the assumption that any parallel execution on real systems can be represented as an interleaved execution at a fine enough level of granularity, e.g. at the machine level. It will thus be helpful for you to

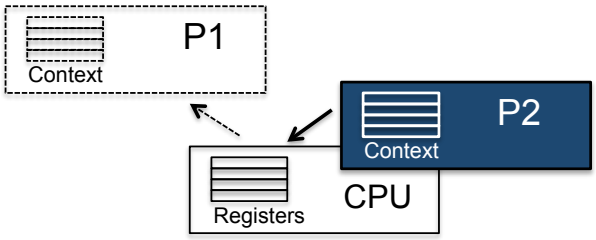


Figure 3: Context switch: process P1 is removed from the CPU and P2 is assigned to it

picture any concurrent execution as the set of all its potential interleavings. In doing so, you will be able to detect any inconsistencies between different executions. We will come back to this point in Section 3.1.

In the following section we will see how operating systems handle multitasking, and thus make things a bit more concrete.

1.2 Operating system processes

Let's have a closer look at processes, a term which we have used informally before. You will probably be aware of the following terminology: a (sequential) *program* is merely a set of instructions; a *process* is an instance of a program that is being executed. The exact structure of a process may change from one operating system to the other; for our discussion it suffices to assume the following components:

- *Process identifier*: the unique ID of a process.
- *Process state*: the current activity of a process.
- *Process context*: the program counter and the values of the CPU registers.
- *Memory*: program text, global data, stack, and heap.

As discussed in Section 1.1, multiple processes can execute at the same time in modern operating systems. If the number of processes is greater than the number of available CPUs, processes need to be scheduled for execution on the CPUs. The operating system uses a special program called the *scheduler* that controls which processes are *running* on a CPU and which are *ready*, i.e. waiting until a CPU can be assigned to them. In general, a process can be in one of the following three states while it is in memory:

- *running*: the process's instructions are executed on a processor.
- *ready*: the process is ready to be executed, but is not currently assigned to a processor.
- *blocked*: the process is currently waiting for an event.

The swapping of process executions on a CPU by the scheduler is called a *context switch*. Assume a process P1 is in the state *running* and should be swapped with a process P2 which is currently *ready*, and consider Figure 3. The scheduler sets the state of P1 to *ready* and saves its context in memory. By doing so, the scheduler will be able to wake up the process at a later

time, such that it can continue executing at the exact same point it had stopped. The scheduler can then use the context of P2 to set the CPU registers to the correct values for P2 to resume its execution. Finally, the scheduler sets P2's process state to *running*, thus completing the context switch.

From the state *running* a process can also get into the state *blocked*; this means that it is currently not ready to execute but waiting for some system event, e.g. for the completion of some prerequisite task by another process. When a process is *blocked* it cannot be selected by the scheduler for execution on a CPU. This can only happen after the required event triggers the state of the blocked process to be set to *ready* again.

Exercise 1.1 Explain the difference between parallel execution, interleaved execution, and concurrent execution.

Exercise 1.2 What is a context switch? Why is it needed?

Exercise 1.3 Explain the different states a process can be in at any particular time.

2 Threads

Concurrency seems to be a great idea for running different sequential programs at the same time: using multitasking, all programs appear to run in parallel even on a system with a single CPU, making it more convenient for the user to switch between programs and have long-running tasks complete "in the background"; in the case of a multiprocessing system, the computing power of the additional CPUs speeds up the system overall.

Given these conveniences, it also seems to be a good idea to use concurrency not only for executing different sequential programs, but also within a single program. For example, if a program implements a certain time-intensive algorithm, we would hope that the program runs faster on a multiprocessing system if we can somehow *parallelize* it internally. A program which gives rise to multiple concurrent executions at runtime is called a *concurrent program*.

2.1 The notion of a thread

Imagine the following method *compute* which implements a computation composed of two tasks:

```
void compute() {
    t1.doTask1();
    t2.doTask2();
}
```

Assume further that it takes m time units to complete the call *t1.doTask1*() and n time units to complete *t2.doTask2*(). If *compute*() is executed sequentially, we thus have to wait m time units after the call *t1.doTask1*() before we can start on *t2.doTask2*(), and the overall computation will take $m+n$ time units, as shown in Figure 4.

If we have two CPUs, this seems rather a waste of time. What we would like to do instead is to execute *t1.doTask1*() on one of the CPUs and *t2.doTask2*() on the other CPU, such that the overall computation takes only $max(m,n)$ time units, as shown in Figure 5.

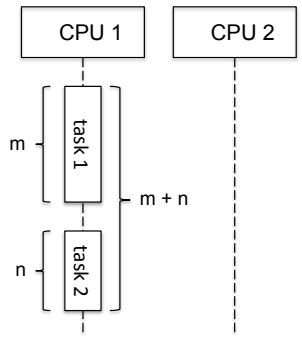


Figure 4: Sequential execution: the overall computation takes $m+n$ time units

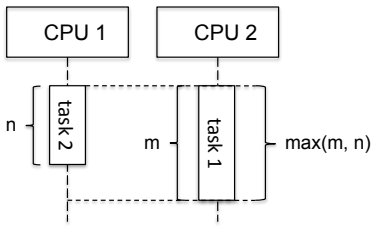


Figure 5: Parallel execution: the overall computation takes $max(m,n)$ time units

In order to be able to associate computation with different execution units, we introduce the notion of a *thread*. A thread can best be understood as a "lightweight process". This means that each thread has its own thread ID, program counter, CPU registers, and stack, and can thus support independent execution of instructions. However, threads are contained within processes, meaning that code and data memory sections and other resources belong to the containing process and are shared by all its threads. A process that has more than one such thread of control is called a multithreaded process. Threads can be assigned to physical CPUs via multitasking, just as operating systems processes are.

2.2 Creating threads

A concurrent program gives rise to a multithreaded process at execution time, so the question is how we can create multiple threads in a programming language. In most programming languages, this is done via thread libraries, which provide the programmer with the API for managing threads. In this lecture we will use the the *Java Threads* API, however the concepts of libraries of other languages are quite similar.

Every Java program consists of at least one thread, which executes the *main*() method. In addition there is the possibility to define user threads; one way to do so is to inherit from the class *Thread* and to override its *run*() method. For example, the *run*() methods of the following two classes provide the implementations of *doTask1*() and *doTask2*() from above.

```
class Worker1 extends Thread {
    public void run() {
        // implement doTask1() here
    }
}
```

```
class Worker2 extends Thread {
    public void run() {
        // implement doTask2() here
    }
}
```

To create threads from these classes, one first creates a *Thread* object and then invokes the *start*() method on this object. This causes the *run*() method of the object to be executed in a new thread. Continuing the example, the following implementation of the method *compute*() creates two threads so that the two tasks from the example can be executed concurrently, and might finish in $max(m, n)$ time units.

```
void compute() {
    Worker1 worker1 = new Thread1();
    Worker2 worker2 = new Thread2();
    worker1.start();
    worker2.start();
}
```

2.3 Joining threads

Let's assume that the classes *Worker1* and *Worker2* from above are extended with the following method and attribute:

```
private int result;

public void getResult() {
    return result;
}
```

This allows the threads to save the final results of the computation in a variable, which can later be read out. For example, we might imagine that the results of the two tasks need to be combined in the *compute*() method:

```
return worker1.getResult() + worker2.getResult();
```

Clearly, we have to wait for both threads to be finished, before we can combine the results. This is done using the *join*() method which, when invoked on a supplier thread, causes the caller thread to wait until the supplier thread is terminated. For our example, this looks as follows:

```
int compute() {
    worker1.start();
    worker2.start();

    worker1.join();
    worker2.join();

    return worker1.getResult() + worker2.getResult();
}
```

Hence the main thread will first wait for *worker1* to finish, and then for *worker2*. Since we have to wait for both threads to finish, the order of the *join*() calls is arbitrary.

Exercise 2.1 Consider that the *run*() methods of threads *t1-t4* contain the following code

t1: *worker1.doTask1*(); *worker2.doTask2*();
t2: *manager.evaluate*();
t3: *worker3.doTask3*();
t4: *manager.finish*();

and that the following program fragment is executed in the *main*() thread:

t1.start();
t2.start();
t2.join();
t3.start();
t1.join();
t3.join();
result := *worker2.getValue*() + *worker3.getValue*();
t4.start();

Assume that the call *worker1.doTask1*() takes 20 time units until it returns, *worker2.doTask2*() 30 time units, *manager.evaluate*() 40 time units, *worker3.doTask3*() 20 time units, *manager.finish*() 20 time units; the queries *worker2.getValue*() and *worker3.getValue*() return immediately. What is the minimum time for execution of this program? Draw a sequence diagram to justify your answer.

3 Mutual exclusion

Up until now, concurrency seems easy enough to handle. If we want to execute instructions concurrently with the rest of the program, we put these instructions in the *run*() method of a new class inheriting from *Thread*, create a corresponding object and call the *start*() method on it. At runtime, this gives rise to a new thread executing our instructions, and we are done. However, what happens if different threads interfere with each other, for example access and modify the same objects? We will see that this might change the results of computations in unexpected ways, and we thus have to avoid these situations by using a special type of synchronization called mutual exclusion. Luckily, Java has a simple mechanism for ensuring mutual exclusion.

3.1 Race conditions

Consider the following class *Counter* which only has a single attribute *value*, and features to set and increment *value*.

```
class Counter {
    private int value = 0;

    public int getValue() {
        return value;
    }
```

```
    public void setValue(int someValue) {
        value = someValue;
    }

    public void increment() {
        value++;
    }
}
```

Now assume that an entity *x* of type *Counter* is created and consider the following code:

```
x.setValue(0);
x.increment();
int i = x.getValue();
```

What is the value of *i* at the end of this execution? Clearly, if this code was part of a sequential program, the value would be 1. In a concurrent setting where we have two or more threads, the value of *x* can be read/modified by all of them. For example consider the following call executed concurrently by another thread:

```
x.setValue(2);
```

What is the value of *i* now?

The answer is that, if these are the only threads running concurrently and *x* references the same object in both cases, *i* could have any of the values 1, 2, or 3. The reason for this is easily explained by looking at the thread interleavings that could be taken:

x.setValue(2) *x.setValue*(0) *x.increment*() **int** *i* = *x.getValue*()	*x.setValue*(0) *x.setValue*(2) *x.increment*() **int** *i* = *x.getValue*()	*x.setValue*(0) *x.increment*() *x.setValue*(2) **int** *i* = *x.getValue*()	*x.setValue*(0) *x.increment*() **int** *i* = *x.getValue*() *x.setValue*(2)
i == 1, *x.value* == 1	*i* == 3, *x.value* == 3	*i* == 2, *x.value* == 2	*i* == 1, *x.value* == 2

This is not really what we intended. The result of our computation has become arbitrary, and depends on the scheduling that determines a particular interleaving. Remember that we have no control over the scheduling.

The situation that the result of a concurrent execution is dependent on the nondeterministic scheduling is called a *race condition* or a *data race*. Data races are one of the most prominent problems in the domain of concurrent programming, and you can imagine that it gives rise to errors which can be quite hard to detect. For example, when you are running a program such as the above, say, 100 times, it might be that, because of a specific timing of events, you always obtain the values *i* == 1 and *x.value* == 1. But when you run the program for the 101st time, one of the other results arises. This means that such errors can stay hidden for a long time, and might never be detected during testing.

The question is now how to avoid data races. Java has a specific mechanism for this, which will be explained in the next section.

3.2 Synchronized methods

To avoid data races we have to *synchronize* different computations such that they don't interfere with each other. Let's think about the main reason for the problem to occur. In the above example, two computations *shared* a resource, namely the object referenced by x. A part of a program that accesses a shared resource is called a *critical section*. The problem would not have occurred if, at any time, at most one computation would be in its critical section. The form of synchronization ensuring this property is called *mutual exclusion*.

Java provides the programmer with a simple way to ensure mutual exclusion. Each object in Java has a *mutex lock*, i.e. a lock that can be held by only one thread at a time. Thus to create a new lock, any object will do:

```
Object lock = new Object();
```

A thread can acquire and release a lock using synchronized blocks:

```
synchronized (lock) {
    // critical section
}
```

When a thread reaches the start of the block, it tries to acquire the lock of the object referenced by *lock*. If the lock is held by another thread, the thread blocks until the lock is finally available. It will then acquire the lock and hold it until the control reaches the end of the block, where the lock is automatically released. As an example, imagine that the instructions from above are put into synchronized blocks, so in one thread we have

```
synchronized (lock) {
    x.setValue(0);
    x.increment();
    int i = x.getValue();
}
```

and in the other thread we have

```
synchronized (lock) {
    x.setValue(2);
}
```

where we make sure that the object referenced by *lock* is the same in both cases.

As explained above, since the same object referenced by *lock* acts as the lock in both synchronized blocks, the critical sections can be executed in mutual exclusion. This means that the state of the object referenced by x can only be modified by one of the threads at a time, and hence upon completion of the first block we always obtain the result $i == 1$.

Besides having an explicit synchronized block, a method can also be decorated with the keyword **synchronized**. This has the same effect as enclosing the method body in a synchronized block where the current object **this** provides the lock, as shown in Figure 6.

Exercise 3.1 Explain the terms data race and mutual exclusion. How can we ensure mutual exclusion in Java Threads?

Exercise 3.2 Recall the *Counter* class from above, and imagine a class *SynchronizedCounter* which has all its methods declared as **synchronized**, but is otherwise identical to *Counter*. Find

synchronized *type method*(*args*) {
 	
  // body
 	
}
	type method(*args*) {
	  synchronized (**this**) {
	  // body
	  }
	}

Figure 6: Correspondence between synchronized blocks and synchronized methods

a simple example involving two threads where the result of the computation is nondeterministic when the methods from *Counter* are used, but not when the ones from *SynchronizedCounter* are used. Explain how these results come about.

4 Condition synchronization

Protecting access to shared variables is not the only reason why a thread has to synchronize with other threads. For example, assume that a thread continuously takes data items out of a buffer to process them. Hence, the thread should only access the buffer if it holds at least one element; if it finds the buffer empty, it therefore needs to wait until another thread puts a data item in. Delaying a thread until a certain condition holds (as in this case, until the "buffer is not empty") is called *condition synchronization*. As you will see, in Java condition synchronization is enabled by the methods *wait*() and *notify*() which can be called on any synchronized object and allow a thread to release a previously acquired object lock and to notify other threads that a condition may have changed.

As an example of a problem that requires threads to use condition synchronization, we describe the so-called *producer-consumer problem*, which corresponds to issues found in many variations on concrete systems. Devices and programs such as keyboards, word processors and the like can be seen as *producers*: they produce data items such as characters or files to print. On the other hand the operating system and printers are the *consumers* of these data items. It has to be ensured that these different entities can communicate with each other appropriately so that for example no data items get lost.

On a more abstract level, we can describe the problem as follows. We consider two types of threads, both of which execute in an infinite loop:

- *Producer*: At each loop iteration, produces a data item for consumption by a consumer.
- *Consumer*: At each loop iteration, consumes a data item produced by a producer.

Producers and consumers communicate via a shared buffer implementing a queue; we assume that the buffer is unbounded, thus we only have to take care not to take out an item from an empty buffer, but are always able to insert new items. Instead of giving the full implementation we just assume to have a class *Buffer* to implement an unbounded queue:

Buffer buffer = **new** *Buffer*();

Producers append data items to the back of the queue using a method **void** *put*(**int** *item*), and consumers remove data items from the front using **int** *get*(); the number of items in a queue is queried by the method **int** *size*().

As part of the consumer behavior, we might for example want to implement the following method for consuming data items from the buffer:

```
public void consume() {
    int value;
    synchronized (buffer) {
        value = buffer.get(); // incorrect: buffer could be empty
    }
}
```

In this method we acquire a lock on the buffer using the synchronized block, and try to get an item from the buffer. The problem is that the buffer might be empty, i.e. *buffer.size*()== 0, which would result in a runtime error in this case. What we would like instead is that the thread waits before accessing the buffer, until the condition "buffer is not empty" is true, and then get the value from the buffer.

Waiting can be achieved in Java using the method *wait*(), which can be invoked on any object which is already locked, i.e. inside a synchronized block which has the object as lock; *wait*() then blocks the current thread (i.e. setting the thread state to *blocked*) and releases the lock. Continuing the example, we adapt the code as follows:

```
public void consume() throws InterruptedException {
    int value;
    synchronized (buffer) {
        while (buffer.size() == 0) {
            buffer.wait();
        }
        value = buffer.get();
    }
}
```

Note that the *wait*() call can throw an *InterruptedException* which we have to note in the method header (or otherwise in a try-catch block). Let's assume that the buffer is indeed found empty by the current thread. Thus the *wait*() call gets executed, the current process gets blocked and releases the lock on the object referenced by *buffer*.

Now the lock can be acquired by another thread, which might change the condition. To notify a waiting thread that the condition has changed, the thread can then execute the method *notify*() which unblocks one waiting thread (i.e. setting the thread state to *ready*), but doesn't yet release the lock. However, eventually the thread will release the lock by leaving the synchronized block, such that the unblocked thread can acquire it eventually and continue. Note that also the method *notify*() can only be called within a synchronized block which locks the object that it is called on.

This signaling step is part of the implementation of the method *produce*():

```
public void produce() {
    int value = random.produceValue();
    synchronized (buffer) {
```

```
            buffer.put(value);
            buffer.notify();
        }
    }
```

In this method the producer thread first creates a random integer value, locks the buffer and puts the value into the buffer. Then the thread uses *notify*() to signal a waiting consumer thread (if there is any) that the condition *buffer.size*()== 0 is no longer true. Note however that the signaled consumer cannot take the truth of the condition for granted, as yet another interleaved consumer thread could have taken out the item before the signaled consumer was able to acquire the lock. This is why the checking of the condition *buffer.size*()== 0 takes place within a while-loop, and not within an if-then-else: the unblocked consumer thread might need to block itself again.

It is important to note that a thread cannot know that a notification corresponds to the change of the condition it was interested in. For example, if we had a bounded buffer, we might also want to notify processes that the "buffer is not full". As *notify*() only unblocks a single process, we cannot be sure whether a process has been unblocked that waits for the condition "buffer is not full" or for "buffer is not empty". For this reason there is the method *notifyAll*() which unblocks all currently waiting processes. While we usually want to avoid using *notifyAll*() for efficiency reasons, and therefore we should use different lock objects corresponding to different conditions whenever possible.

Exercise 4.1 You are to implement a controller for a device which can be accessed with the following interface:

```
class Device {
    public void startup() { ... }
    public void shutdown() { ... }
}
```

There are also two sensors, one for heat and one for pressure, which can be used to monitor the device.

```
class Sensor extends Thread {
    Device device;
    private int value;

    public Sensor(Device d) {
        device = d;
    }

    public int getValue() {
        return value;
    }

    public void updateValue() { ... }

    public void run() { ... }
}
```

Write a class *Controller* in Java Threads that can poll the sensors concurrently to running the device. You should implement its *run*() method such that it starts the device and then monitors it by waiting for and examining any new sensor values. The controller shuts down the device if the heat sensor exceeds the value 70 or the pressure sensor the value 100. Also complete the *run*() method in the class *Sensor* which calls *updateValue*() continuously and signals the controller if its value has changed.

Exercise 4.2 What is the difference between *notify*() and *notifyAll*()? When is it safe to substitute one with the other?

Exercise 4.3 Name and explain three forms of synchronization used in Java Threads.

Exercise 4.4 Write down three possible outputs for the Java Threads program shown below:

```
public class Application
extends Thread {
  public static X x;
  public static Y y;
  public static Z z;

  public void run() {
    z = new Z(); x = new X(z);
    y = new Y(z);
    System.out.print("C");
    execute1();
    z.h();
    execute2();
  }

  public void execute1() {
    System.out.print("A");
    x.start();
  }

  public void execute2() {
    y.start();
    System.out.print("L");
  }
}
```

```
class X
extends Thread {
  public Z z;

  public X(Z zz) {
    z = zz;
    z.n = 0;
  }

  public void run() {
    synchronized (z) {
      z.n = 1;
      z.notify();
      System.out.print("K");
    }
  }
}
```

```
class Y
extends Thread {
  public Z z;

  public Y(Z zz) {
    z = zz;
  }

  public void run() {
    System.out.print("J");
    synchronized (z) {
      while (z.n == 0) {
        try {
          z.wait();
        } catch (
            InterruptedException
            e) {};
      }
      System.out.print("Q");
    }
  }
}
```

```
public class Root {
  public static void main(String[]
      args) {
    Application app = new
        Application();
    app.start();
  }
}
```

```
class Z {
  public int n;
  public void h() {
    System.out.print("P");
  }
}
```

5 Deadlock

While we have seen that locking is necessary for the proper synchronization of processes, it also introduces a new class of errors in concurrent programs: deadlocks. A *deadlock* is the situation where a group of processors blocks forever because each of the processors is waiting for resources which are held by another processor in the group. In thread libraries, a common class of resources are mutex locks. As explained in Section 3, locks are requested using synchronized blocks, and held for the duration of the method.

As a minimal example, consider the following class:

```
public class C extends Thread {
    private Object a;
    private Object b;

    public C(Object x, Object y) {
        a = x;
        b = y;
    }

    public void run() {
        synchronized (a) {
            synchronized (b) {
                ...
            }
        }
    }
}
```

Now imagine that the following code is executed, where *a1* and *b1* are of type *Object*:

```
C t1 = new C(a1, b1);
C t2 = new C(b1, a1);
t1.start();
t2.start();
```

Since the arguments are switched in the initialization of *t1* and *t2*, a sequence of calls is possible that lets the threads first acquire the locks to *a1* and *b1*, respectively, such that they end up in a situation where each of them requires a lock held by the other handler.

Note that there is no built-in mechanism of Java Threads that prevents deadlocks from happening, and it is the programmers responsibility to make sure that programs are deadlock-free.

Exercise 5.1 Explain in detail how a deadlock can happen in the above example by describing a problematic interleaving and the locks taken.

Answers to the exercises

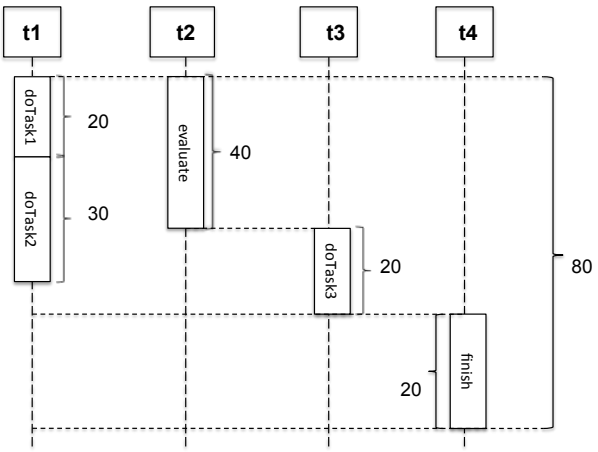
Answer 1.1 If all processes in a group are running at the same time, their execution is said to be *parallel*. If all processes of a group have started to execute but only one process is running at a

time, their execution is said to be *interleaved*. We say that the execution of a group of processes is *concurrent* if it is either parallel or interleaved. □

Answer 1.2 A context switch is the exchange of one process's context (its program counter and CPU registers) with another process's context on a CPU. A context switch enables the sharing of a CPU by multiple processes. □

Answer 1.3 A process can be in one of three states: *running*, *ready*, and *blocked*. If a process is *running*, its instructions are currently executed on a processor; if a process is *ready*, it is waiting for the scheduler to be assigned to a CPU; if a process is *blocked*, it is currently waiting for an event which will set its state to *ready*. □

Answer 2.1 The computation takes at least 80 time units, as can be seen from the following sequence diagram.



□

Answer 3.1 A data race is the situation where the result of a concurrent computation depends on scheduling. Mutual exclusion is a form of synchronization to avoid the simultaneous use of a shared resource (such as a shared object) by multiple processes.

Java Threads allows to protect critical sections by **synchronized** blocks. Two blocks guarded by the same lock object are guaranteed to execute in mutual exclusion. To protect a shared variable of an object, a common pattern is to declare the variable private in the class and declare all public methods accessing the shared variable **synchronized**. Then all method bodies will be executed with mutual exclusion (having the common lock object **this**), and the shared variable is protected against data races as it can only be accessed through these methods. □

Answer 3.2 The following simple example proves the point:

t1: *increment*();
t2: *increment*();

If we assume that the value of the counter is 0 at first, the value after both threads have finished will always be 2 in the case of the *SynchronizedCounter* methods, and 1 or 2 if *Counter* methods are used. The reason for this is that at the byte code level, the increment instruction consists of the following steps

```
temp = value;
temp = temp + 1;
value = temp;
```

which can be interleaved in the case of *Counter*. In *SynchronizedCounter* this cannot happen as the **synchronized** block ensures that this group of instructions is executed atomically. □

Answer 4.1 The controller and the sensor can be implemented in the following manner, together with an appropriate root class:

```
class Root {
  public static void main(String[] args) {
      Device d = new Device();
      Sensor h = new Sensor(d);
      Sensor p = new Sensor(d);
      Controller c = new Controller(d,h,p);
      h.start();
      p.start();
      c.start();
  }
}

class Controller extends Thread {
  Device device;
  Sensor heat;
  Sensor pressure;

  public Controller(Device d, Sensor h, Sensor p) {
    device = d;
    heat = h;
    pressure = p;
  }

  public void run() {
    device.startup();
    synchronized (device) {
      while (heat.getValue() <= 70 && pressure.getValue() <= 100) {
        try { device.wait(); } catch (InterruptedException e) {}
      }
    }
    device.shutdown();
  }
}

class Sensor extends Thread {
  Device device;
  private int value;
```

```
    public Sensor(Device d) {
      device = d;
    }

    public int getValue() {
      return value;
    }

    public void updateValue() { ... }

    public void run() {
      while (true) {
        synchronized (device) {
          int oldValue = value;
          updateValue();
          if (value != oldValue) {
            device.notify();
          }
        }
      }
    }
  }

  class Device {
    public void startup() { ... }
    public void shutdown() { ... }
  }
```

Note that condition synchronization is used to check on the emergency shutdown conditions: whenever the sensor obtains a new value, it will signal the controller, upon which the controller rechecks the condition and blocks itself if it is not yet fulfilled. □

Answer 4.2 The call *notify*() wakes up exactly one waiting thread, the call *notifyAll*() wakes up all waiting threads. A typical pattern for condition synchronization is

```
    while (!condition) {
        lock.wait();
    }
```

In this case it is safe to substitute any calls to *notify*() with *notifyAll*(), although this is inefficient: many threads which have been unblocked will just have to block themselves again as they find the condition invalidated once they get there.

On the other hand, *notifyAll*() typically cannot be substituted by *notify*() without semantic changes in other parts of the program. One reason is that threads can be blocked on various conditions within a **synchronized** block and that notify-calls cannot distinguish between them. The call *notifyAll*() will force all blocked processes to recheck their conditions, allowing at least one to proceed. The call *notify*() will only unblock one arbitrary process, whose condition might still be false; in this case a deadlock can happen, causing processes to wait for an event which will never manifest itself (see Section 5). □

Answer 4.3 There are three major forms of synchronization provided in Java Threads: mutual exclusion, condition synchronization, and thread join. Mutual exclusion for object access can be ensured by the use of **synchronized** blocks. Condition synchronization (waiting until a certain condition is true) is provided via the methods *wait*() and *notify*() that can be called on synchronized objects at the point where the condition is known to be false or true, respectively. Joining threads is provided by the method *join*() which, called on a thread, causes the caller to wait until the thread has completed execution. □

Answer 4.4 Three possible output sequences are:

- CAPLKJQ
- CAPLJKQ
- CAKPLJQ

In method *run* of class *Application* "C" is always printed at the beginning. Upon calling *execute1*(), "A" is printed next and a new thread is started, which will run concurrently to the application thread. Hence "K" can be printed before or after "P" (which results from the call *z.h*()). "J" will be printed at the start of the thread *y*, after "P". Because of the waiting on the condition $z.n == 0$, "Q" will always be printed after "K". □

Answer 5.1 The following sequence of events can happen. First thread *t1* is executing, and obtains *a*'s lock at the start of its synchronized block. After a context switch, thread *t2* is executing and obtains *b*'s lock, since the roles of *a* and *b* are switched in *t1* and *t2*. After this *t1* will request *b*'s lock (currently held by *t1*), while *t2* requests *a*'s lock (currently held by *t2*): a deadlock has occurred as none of the threads can proceed any further. □