Concurrent Programming with SCOOP

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 an object-oriented programming model for concurrency called *SCOOP* (Simple Concurrent Object-Oriented Programming). 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 the SCOOP programming model,
- program simple concurrent programs using SCOOP.

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. At 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.

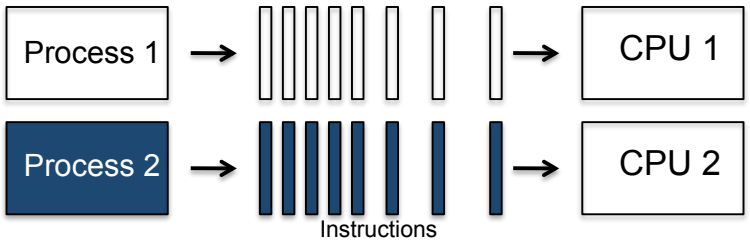


Figure 1: Multiprocessing: instructions are executed in parallel

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.

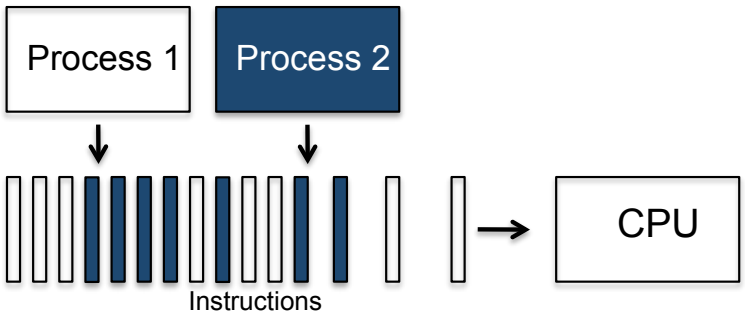


Figure 2: Multitasking: instructions are interleaved

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 picture any concurrent execution as the set of all its potential interleavings. In doing so, you

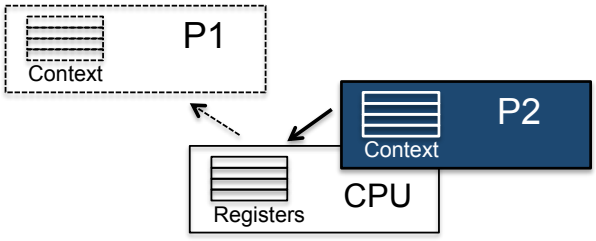


Figure 3: Context switch: process P1 is removed from the CPU and P2 is assigned to it

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 Processors

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 processor

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

```
compute
    do
        t1.do_task1
        t2.do_task2
    end
```

Assume further that it takes m time units to complete the call *do_task1* on the object attached to entity *t1* and n time units to complete *do_task2* on the object attached to entity *t2*. If *compute* is executed sequentially, we thus have to wait m time units after the call *t1.do_task1* before we can start on *t2.do_task2*, 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 *do_task1* on the object attached to entity *t1* by one of the CPUs and *do_task2* on the object attached to entity *t2* by the other CPU, such that the overall computation takes only $max(m,n)$ time units, as shown in Figure 5.

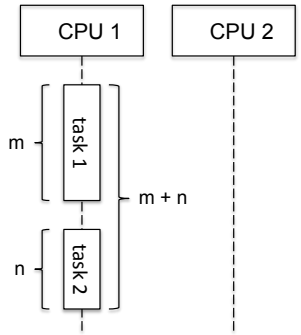


Figure 4: Sequential execution: the overall computation takes $m+n$ time units

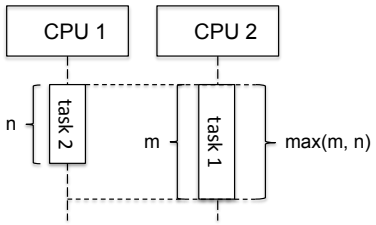


Figure 5: Parallel execution: the overall computation takes $max(m,n)$ time units

In order to be able to associate computation with different execution units, we introduce the abstract notion of a processor. An (abstract) *processor* can best be understood as a kind of virtual CPU: an entity supporting sequential execution of instructions on one or several objects. Each processor has a *request queue* which holds the instructions that it has to execute, and works them off one by one.

In contrast to physical CPUs, the number of processors is not bounded. We imagine that processors can be assigned to physical CPUs via multitasking, just as operating systems processes are. In the following we will use the term processor only in this abstract sense, and use the term CPU to denote a physical CPU.

The fundamental idea of abstract processors in SCOOP is their relationship to objects: each object is assigned to exactly one processor, called the *handler* of the object. On the other hand, a processor can handle multiple objects.

If a new object is created, the runtime system decides which handler it is assigned to or whether a new processor is created for it, and this assignment remains fixed over the course of the computation. The assignment is guided by an extension of the type system, as we will see later. Assume for now that *t1* is handled by a processor p, and *t2* and *t3* are handled by a processor q. We can depict this with the diagram shown in Figure 6.

We frequently use such diagrams as they give us an idea of the associations of processors and objects. Each region tagged by a processor name contains the objects this processor is handling; processor regions are separated by a dashed line.

2.2 Synchronous and asynchronous feature calls

What does the *handling* of an object imply? It means that all operations on the given object are executed by its handling processor; there is no sharing of objects between processors. For example, assume that the following feature calls

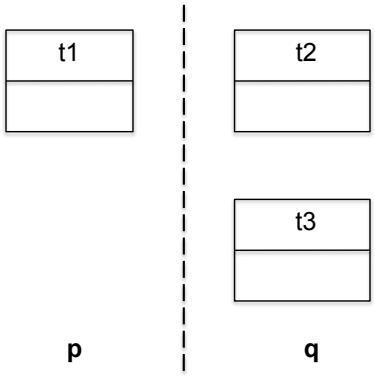


Figure 6: Processor regions: *t1* is handled by processor *p*, and *t2* and *t3* are handled by a processor *q*

t1.do_task1
t2.do_task2
t3.f

are encountered by the current processor *q*, and that the processor association are as in Figure 6. Then *q* doesn't execute the call *t1.do_task1* by itself, but asks *p* to do it by appending the call to *p*'s request queue. The benefit is that processor *q* doesn't have to wait for the call *t1.do_task1* to complete: in contrast to the sequential case, *q* can just continue with other calls, e.g. *t2.do_task2*, which it is handling by itself. Hence the two tasks can be executed concurrently. Lastly, the call *t3.f* is handled once again by processor *q*, therefore it is only started after the task *t2.do_task2* has been completed.

A feature call on an object which is handled by a different processor than the current one is called an *asynchronous* feature call or a *separate call* (e.g., *do_task1*). In this case the current processor can proceed after issuing the call to the other processor, and doesn't have to wait for the call to return. In contrast, a feature call on an object handled by the current processor is called a *synchronous* feature call or a *non-separate call* (e.g. *do_task2*). This is the situation well-known from ordinary sequential programming, and the current processor has to wait for the call to return before continuing.

2.3 Separate entities

We have left open the question of how the runtime system determines whether a particular object is handled by one processor or another. The answer is that the type system is extended to guide the runtime system in this decision, thus giving the programmer control over whether a call is executed synchronously or asynchronously.

To this end, a new keyword is introduced in the language SCOOP: **separate**. Along with the usual

$x : X$

to denote an entity x that can be attached to objects of type X, we can now also write

x : **separate** X

to express that at runtime, x may be attached to objects handled by a different processor. We then say that x is of type **separate** X, or that it is a *separate entity*.

The value of a separate entity is called a *separate reference*, and an object attached to it is called a *separate object*. To emphasize that a certain reference or object is not separate, we use the term *non-separate*. We also extend our diagrams to include references to objects by drawing arrows to the referenced object. If an arrow crosses the border of a processor's domain, it is a separate reference. The diagram in Figure 7 shows two objects *x* and *y* which are handled by different processors, where *x* contains a separate reference to *y*.

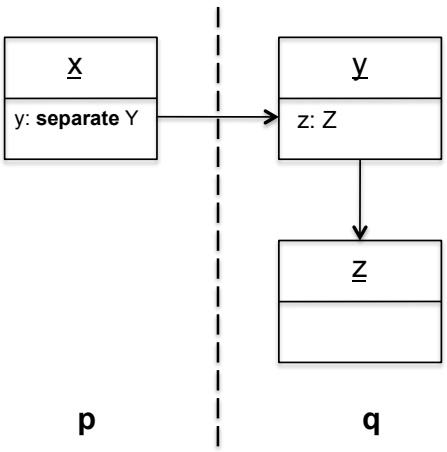


Figure 7: Separate reference: *y* references an object on a different processor

2.4 Wait-by-necessity

We generalize the example of Section 2.1 by defining the following class *WORKER*:

```
class WORKER
    feature
        output: INTEGER
        do_task (input: INTEGER) do ... end
end
```

The idea is that a worker can run *do_task* (*input*) and will store the result in the feature *output*. Let's assume that two workers are defined in the class *MANAGER* as follows:

```
class MANAGER
    feature
        worker1 : separate WORKER
        worker2 : WORKER

        -- in some routine:
            do
                . . .
                worker1.do_task (input1)
                worker2.do_task (input2)
                result := worker1.output + worker2.output
            end
end
```

We have learned before that separate calls are spawned off asynchronously, so the current processor doesn't have to wait for them to return. The call *worker1.do_task* (*input1*) is therefore executed on a different processor; the second call *worker2.do_task* (*input2*) is synchronous and is executed on the same processor the manager object is running on. Note that the call *worker1.do_task* (*input1*) is a *command* and thus just transforms the target object *worker1*, without returning a result. But what about the call *worker1.output*? This is a *query* and thus returns a result. As we are interested in the result, we clearly have to wait for the call to return; furthermore, we would also like to wait until previous computations on the object are finished before retrieving the information.

This waiting happens automatically in SCOOP, and the corresponding synchronization principle is called *wait-by-necessity*:

> "If a client has started one or more calls on a certain separate object, and it executes on that object a call to a query, that call will only proceed after all the earlier ones have been completed, and any further client operations will wait for the query to terminate."

This rule ensures that after completion of the call *worker2.do_task* (*input2*), the processor will also wait for the asynchronous completion of call *worker1.do_task* (*input1*) before combining the results. This mechanism is completely automatic, so you as a programmer don't have to worry about this. However, when trying to optimize programs, it is important to know that queries on an object will act as a *barrier*, i.e. a program point where execution waits for all previously spawned calls on that object before proceeding.

Exercise 2.1 How does the execution of an asynchronous feature call differ from a synchronous one? How are asynchronous feature calls expressed in SCOOP?

Exercise 2.2 Consider that the following SCOOP program fragment is executed on a processor *p*:

worker1.do_task1
worker2.do_task2
manager.evaluate
worker3.do_task3
result := *worker2.value* + *worker3.value*
manager.finish

The object-processor associations are given as follows: *worker1* and *worker2* are handled by processor *q*, *manager* by processor *p*, and *worker3* by processor *r*. The call *worker1.do_task1* takes 20 time units until it returns, *worker2.do_task2* 30 time units, *manager.evaluate* 40 time units, *worker3.do_task3* 20 time units, *manager.finish* 20 time units; the queries return immediately. What is the minimum time for execution of this program? Draw a sequence diagram to justify your answer.

Exercise 2.3 Consider classes *A* and *B*

class *A*
  feature
  b: **separate** *B*
  c: *C*
 	
  set_b (*b1*: **separate** *B*) **do** *b* := *b1* **end**
  set_c (*c1*: *C*) **do** *c* := *c1* **end**
end
	class *B*
	  feature
	  a: **separate** *A*
	
	  set_a (*a1*: **separate** *A*) **do** *a* := *a1* **end**
	
	end

and assume that the following program fragment is executed

a.*set_b* (*b*)
a.*set_c* (*c*)
b.*set_a* (*a*)

where *a* and *c* are handled by processor *p*, and *b* is handled by processor *q*. Draw a diagram showing the association of objects with processor regions and any separate or non-separate references.

Exercise 2.4 Under what conditions does wait-by-necessity become applicable?

3 Mutual exclusion

Up until now, concurrency seems easy enough to handle. If we want a feature to be evaluated concurrently, we have to declare its corresponding target **separate**. At runtime, this gives rise to an asynchronous feature call, and we are done. However, what happens if different calls 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, SCOOP 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
    feature
        value : INTEGER

        set_value (a_value: INTEGER)
            do
                value := a_value
            end

        increment
            do
                value := value + 1
```

end

end

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

x.set_value (0)
x.increment
i := *x.value*

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 processors, the value of *x* can be read/modified by all processors though that handle objects owning a separate reference to *x*. For example consider the following call executed concurrently by another processor (different from the processor executing the above code):

x.set_value (2)

What is the value of *i* now?

The answer is that, if these are the only feature calls running concurrently and *x* is attached to the same object in both cases, *i* could have any of the values 1, 2, or 3. The reason for this is easily explained. Assume that processor *p* is handling the object associated with *x*. This processor will receive feature calls for evaluation from concurrently executed code parts, and will interleave them. The following interleavings could be taken:

x.set_value (2) *x.set_value* (0) *x.increment* *i* := *x.value*	*x.set_value* (0) *x.set_value* (2) *x.increment* *i* := *x.value*	*x.set_value* (0) *x.increment* *x.set_value* (2) *i* := *x.value*	*x.set_value* (0) *x.increment* *i* := *x.value* *x.set_value* (2)
$i = 1$ and $x.value = 1$	$i = 3$ and $x.value = 3$	$i = 2$ and $x.value = 2$	$i = 1$ and $x.value = 2$

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. SCOOP has a specific mechanism for this that eliminates these types of errors at compile-time (before you even run the program!), which will be explained in the next section.

3.2 The separate argument rule

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 attached to 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*.

SCOOP has a simple way to ensure mutual exclusion: its runtime system automatically *locks* the processors which handle separate objects passed as arguments of a routine. If a processor is locked, no other computation can use it to evaluate a feature call; the processor becomes private to whoever locked it. Let's make an example to see how that helps us.

Recall the above example, but let's extend it to see the routine the code has been taken from:

```
compute (x: separate COUNTER)
    do
        x.set_value (0)
        x.increment
        i := x.value
    end
```

Consider now the call *compute* (x) and assume that x is handled by processor p. As explained above, since x is a separate argument to the routine, the processor p must be locked. The current processor, which is about to execute the *compute* feature, waits until the underlying runtime system locks processor p. As soon as p is locked, the body of the routine can be executed without interference (multiple locks on a processor are not possible), and hence upon completion of the routine we always obtain the result $i = 1$ and $x.value = 1$.

This is so important that SCOOP forces us to make separate entities which we want to access an argument of the enclosing routine. This is formulated as the *separate argument rule*:

> "The target of a separate call must be a formal argument of the routine that contains the separate call."

In other words, all calls on separate objects must be wrapped in a procedure that makes it possible to pass the target as argument. Hence only one of the following two examples is correct:

`x : separate X`
`compute`
`  do`
`  x.f`
`  end`
	`x : separate X`
	`compute (x1: separate X)`
	`  do`
	`  x1.f`
	`  end`

Incorrect: Target x is declared separate, but not an argument of the enclosing routine *compute*.	*Correct*: Target *x1* is separate and therefore has to be an argument of the enclosing routine. In order to execute *x.f*, we use the call *compute* (x).

Note that if an argument of a separate type is passed, the corresponding formal argument must also be of separate type. Thus, in the example above on the right hand side, *x1* must be declared of type **separate** *X*, since *x*, which we want to pass as an argument is also of type **separate** *X*. This type system restriction avoids that entities declared as non-separate can become attached to separate objects, which would compromise the correctness of the SCOOP model.

An analogous requirement holds also for assignments. For example, if *x1* := *x* and *x* is of type **separate** *X* (or might just be attached to an object on a separate processor), then *x1* must be of type **separate** *X* too. This can be remembered by "*nonsep* := *sep*" being disallowed, and is also summarized in the following typing rule:

> "If the source of an attachment (assignment or argument passing) is separate, its target must be separate too."

Note that an assignment the other way around ("*sep* := *nonsep*", i.e. non-separate source, separate target) is however admissible.

Exercise 3.1 Explain the terms data race and mutual exclusion. How does SCOOP ensure mutual exclusion?

Exercise 3.2 Consider the following class *MOTORBIKE* that models a motorbike with engine, wheels, and status display. The class doesn't compile properly. Find all errors and fix them, assuming that the type declarations of all attributes are correct and that the omitted classes *ENGINE*, *DISPLAY*, *WHEEL* have the features mentioned.

```
class MOTORBIKE
    create
        make
    feature
        engine: separate ENGINE
        display: DISPLAY
        front_wheel: separate WHEEL
        back_wheel: separate WHEEL

        make
            do
                create engine; create display
                create front_wheel; create back_wheel
            end
        initialize
            do
                engine.initialize
                initialize_wheels
                display.show ("Ready")
            end

        initialize_wheels
            do
                display.show ("Initializing wheels ...")
```

```
            front_wheel.initialize
            back_wheel.initialize
        end

    switch_wheels
        local
            wheel: WHEEL
        do
            wheel := front_wheel
            front_wheel := back_wheel
            back_wheel := wheel
        end
end
```

4 Condition synchronization

Protecting access to shared variables is not the only reason why a process has to synchronize with other processes. For example, assume that a process continuously takes data items out of a buffer to process them. Hence, the process 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 process puts a data item in. Delaying a process until a certain condition holds (as in this case, until the "buffer is not empty") is called *condition synchronization*. As you will see, SCOOP has an elegant way of expressing condition synchronization by reinterpreting the preconditions of a routine as *wait conditions*.

As an example of a problem that requires processes 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 processes, 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 generic class *BUFFER*[*T*] to implement an unbounded queue:

buffer: **separate** *BUFFER*[*INTEGER*]

Producers append data items to the back of the queue using a routine *put*(*item*: *INTEGER*), and consumers remove data items from the front using *get*: *INTEGER*; the number of items in a queue is determined by the feature *count*: *INTEGER*.

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

```
consume (a_buffer: separate BUFFER[INTEGER])
    require
        not (a_buffer.count == 0)
    local
        value: INTEGER
    do
        value := a_buffer.get
    end
```

Note that we have used a precondition to ensure that if we attempt to get a value from the buffer, it is not currently empty. However, what should happen if the buffer is indeed found empty? In a sequential setting, we would just throw an exception. However, this is not justified in the presence of concurrency: eventually a producer will put a value into the buffer again, allowing the consumer to proceed; the consumer will just have to wait a while. To implement this behavior, the runtime system first ensures that the lock on *a_buffer*'s processor is released (which was locked to allow the precondition to be evaluated); this allows values to be put in the buffer. The call is then reevaluated at a later point.

This means that the semantics of preconditions is reinterpreted: they are now treated as wait conditions, meaning that the execution of the body of the routine is delayed until they are satisfied. We can summarize this behavior in the *wait rule*:

> "A routine call with separate arguments will execute when all corresponding processors are available and the precondition is satisfied. The processors are held exclusively for the duration of the routine."

We complete the producer-consumer example by showing the code of the producer's main routine:

```
produce (a_buffer: separate BUFFER[INTEGER])
    local
        value: INTEGER
    do
        value := random.produceValue
        a_buffer.put (value)
    end
```

Since the buffer is unbounded, a wait condition is not necessary. It is however easily added and then makes the solution completely symmetric.

Exercise 4.1 What is the difference between the **require** clause in SCOOP and in ordinary Eiffel?

Exercise 4.2 Imagine a SCOOP routine has a precondition such as $n > 0$, that doesn't involve any separate targets. What do you think should happen in this case?

Exercise 4.3 You are to implement a controller for a device which can be accessed with the following interface:

```
class DEVICE
    feature
        startup do ... end
        shutdown do ... end
end
```

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

```
class SENSOR
    feature
        value: INTEGER
        device: DEVICE
end
```

Write a class *CONTROLLER* in SCOOP that can poll the sensors concurrently to running the device. You should implement two routines: *run* starts the device and then monitors it with help of a routine *emergency_shutdown*, which shuts the device down if the heat sensor exceeds the value 70 or the pressure sensor the value 100.

Exercise 4.4 Name and explain three forms of synchronization used in SCOOP.

Exercise 4.5 Write down three possible outputs for the SCOOP program shown below:

```
class APPLICATION
create make
feature
  x: separate X
  y: separate Y
  z: Z

  make
    do
      create x; create y; create z
      print ("C")
      run1 (x)
      z.h
      run2 (y)
    end
  run1 (xx: separate X)
    do
      print ("A")
      xx.f
    end

  run2 (yy: separate Y)
    do
      yy.g (x)
      print ("L")
      yy.g (x)
    end
end
```

```
class X
feature
  n: INTEGER

  f
    do
      n := 1
      print ("K")
    end
end
```

```
class Y
feature
  g (x: separate X)
    require
      x.n = 1
    do
      print ("Q")
    end
end
```

```
class Z
feature
  h
    do
      print ("P")
    end
end
```

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 SCOOP, the resources are the locks of the processors. As prescribed by the wait rule, a lock on processor p is requested when executing a call to a routine with a separate argument handled by p; the lock is held for the duration of the routine.

As a minimal example, consider the following class:

```
class C
    creation
        make

    feature
        a : separate A
        b : separate A

        make (x : separate A, y : separate A)
            do
                a := x
                b := y
            end

        f do g (a) end
        g (x : separate A) do h (b) end
        h (y : separate A) do ... end
end
```

Now imagine that the following code is executed, where *c1* and *c2* are of type **separate** *C*, *a* and *b* are of type **separate** *A*, and *a* is handled by processor *p*, and *b* by processor *q*:

```
create c1.make (a, b)
create c2.make (b, a)
c1.f
c2.f
```

Since the arguments are switched in the initialization of *c1* and *c2*, a sequence of calls is possible that lets their handlers first acquire the locks to *p* and *q* respectively, such that they end up in a situation where each of them requires a lock held by the other handler.

Deadlocks are currently not automatically detected by SCOOP, and it is the programmers responsibility to make sure that programs are deadlock-free. An implementation of a scheme for preventing deadlocks is however underway, and is based on locking orders that prevent cyclical locking.

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

Answers to the exercises

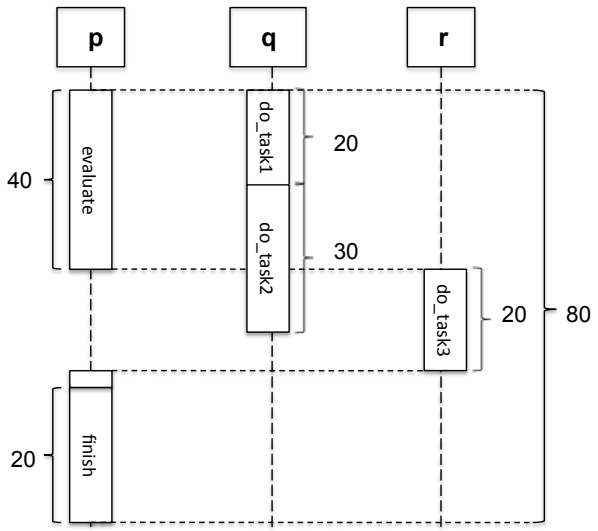
Answer 1.1 If all processes in a group are running at the same time, their execution is said to be *parallel*. If all processes of a group have started to execute but only one process is running at a time, their execution is said to be *interleaved*. We say that the execution of a group of processes 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 external event which will set its state to *ready*. □

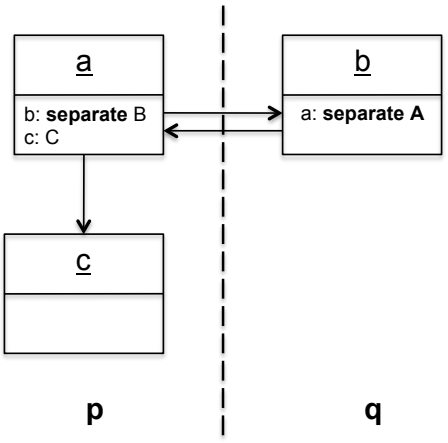
Answer 2.1 An asynchronous feature call is executed on a different processor than the current one. This means it runs concurrently with other computations that are subsequently executed on the current processor. Ordinary sequential feature calls which are executed on the current processor are called synchronous. In SCOOP, a feature call $t.f$ where t is separate (of some type **separate** X) will be executed asynchronously; if t's type is non-separate, it will be executed synchronously. □

Answer 2.2 The computation takes at least 80 time units, as can be seen from the following sequence diagram.



□

Answer 2.3 The following diagram depicts the object-processor associations and the references after execution of the program fragment.



□

Answer 2.4 A statement will be executed with wait-by-necessity semantics if it contains a query on a separate target. □

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.

In SCOOP, an object can only be accessed by its handler, and this handler must be locked before it can be used to execute calls on the object. Mutual exclusion follows from the fact that only one processor can have a lock on another processor at any time. A lock on the handler of some object is taken by passing this object as an argument to the routine it is used in. SCOOP enforces this argument passing by the separate argument rule. □

Answer 3.2 The class contains numerous violations of the separate argument rule. These violations are reported and fixed in the following code:

```
class MOTORBIKE
    feature
        engine: separate ENGINE
        front_wheel: separate WHEEL
        back_wheel: separate WHEEL
        display: DISPLAY

        initialize (e: separate ENGINE) -- Added separate argument
            do
                e.initialize -- Fixed: engine.initialize was incorrect as 'engine' is a
                    separate target, but not argument of the routine 'initialize'
                initialize_wheels(front_wheel, back_wheel)
                display.show ("Ready") -- This is correct: display is non-separate
            end

        initialize_wheels (f, b: separate WHEEL) -- Added separate arguments
            do
                display.show ("Initializing wheels ...")
                f.initialize -- Fixed
```

```
            b.initialize −− Fixed
        end

    switch_wheels
        local
            wheel: separate WHEEL −− Fixed: changed type from WHEEL to
                separate WHEEL...
        do
            wheel := front_wheel −− ...otherwise this would violate the typing rule:
                a separate source is assigned to a non−separate target
            front_wheel := back_wheel
            back_wheel := wheel
        end
end
```

Answer 4.1 In ordinary Eiffel, a precondition that evaluates to false gives rise to an exception. In SCOOP no exception is thrown and instead the call is scheduled for reevaluation at a later point. □

Answer 4.2 A precondition that doesn't involve any separate targets will always evaluate to the same value, as the objects involved cannot be changed concurrently. If such a precondition evaluates to false, an exception is therefore thrown, just as in the sequential case. □

Answer 4.3 The controller can be implemented in the following manner:

```
class CONTROLLER
    create
        make

    feature
        device: DEVICE
        heat: separate SENSOR
        pressure: separate SENSOR

        make (d: DEVICE; h, p: separate SENSOR)
            do
                device := d
                heat := h
                pressure := p
            end

        run (d: DEVICE)
            do
                d.startup
                emergency_shutdown (d, heat, pressure)
            end
```

```
        emergency_shutdown (d: DEVICE; h, p: separate SENSOR)
            require
                h.value > 70 or p.value > 100
            do
                d.shutdown
            end
    end
```

Note that the wait conditions on *emergency_shutdown* ensure that the shutdown is initiated only if the sensors exceed their threshold values. Observe that the separate argument rule is correctly abided by. □

Answer 4.4 There are three major forms of synchronization provided in SCOOP: mutual exclusion, condition synchronization, and wait-by-necessity. Mutual exclusion for object access is ensured by the separate argument rule. Condition synchronization (waiting until a certain condition is true) is provided via the reinterpretation of preconditions as wait conditions. Wait-by-necessity is provided for queries on separate targets and ensures that an object is only queried after all previous calls have been finished and causes the caller to wait for this. □

Answer 4.5 Three possible output sequences are:

- CAKPLQQ
- CAPKQLQ
- CAKPQLQ

In routine *make* "C" is always printed at the beginning. Then there are three non-separate calls, which will be worked off one after the other. In *run1*, "A" is always printed first, but then the call *xx.f* is separate, i.e. will execute asynchronously. Hence, "K" might be printed after "A", but also after "P" has been printed as a result of the call *z.h*. The call *yy.g* (*x*) proceeds only if *x.n* = 1 is true, i.e. after "K" has been printed. Since both calls to *yy.g* (*x*) are asynchronous, but *print*("*L*") is synchronous "L" may be printed before or after the first "Q", but must be printe before the second "Q". □

Answer 5.1 The following sequence of calls can happen. First *c1.f* is executed, leading to the call *g* (*a*). Since *a* is a separate argument of routine *g*, its handler *p* gets locked. Then *c2.f* is executed, leading to the call *g* (*b*), since the roles of *a* and *b* are switched in *c1* and *c2*; this means that *q* is locked. On processor *p*, the call *h* (*b*) is issued, thus requesting a lock on *q*; on processor *q*, the call *h* (*a*) is issued, thus requesting a lock on *p*: a deadlock has occurred as none of the processors can proceed any further. □