CAMBRIDGE

Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information

1

Concurrent Processes

1.1 BASIC CONCEPTS

A sequential program describes how to solve a computational problem in a sequential computer. An example is the traveling salesman problem in which the number of cities and the distances between each pair of cities are given. These are the input data on which the output data are determined, which from the solution to the problem. The solution is a closed route of the minimum length of the salesman passing through every city exactly once. More precisely, a sequential program is a sequence of instructions that solves the problem by transforming the input data into the output data. It is assumed that a sequential program is executed by a single processor.

If more processors are to be used to solve the problem, it must be partitioned into a number of subproblems that may be solved in parallel. The solution to the original problem is a composition of solutions to the subproblems. The subproblems are solved by separate components that are the parts of a concurrent program. Each component is a traditional sequential program called a **computational task**, or, in short, **task**. A **concurrent program** consists of a number of tasks describing computation that may be executed in parallel. The concurrent program defines how the tasks cooperate with each other applying partial results of computation, and how they synchronize their actions.

Tasks are executed in a parallel computer under supervision of the operating system. A single task is performed as a **sequential (serial) process**, that is as a sequence of operations, by a conventional processor that we call a **virtual processor**. In a sequential process, resulting from execution of a single instruction sequence, the next operation commences only after completion of the previous operation. Thus, the order of operations is clearly defined. Let $\underline{o_i}$ and $\overline{o_i}$ denote the events of beginning and end of an operation o_i. Then in a sequential process the following relation between the times of termination of operation o_i and commencement of operation o_{i+1} holds: $t(\overline{o_i}) \leq t(\underline{o_{i+1}})$. Figure 1.1 shows the sequential processes $\mathcal{P}$ and $\mathcal{P}'$ that differ in the commencement and termination times of operations $o_1, o_2, \ldots, o_i, o_{i+1}$. If the operations of the processes are identical, in terms of their arguments and results, the results of computation of $\mathcal{P}$ and $\mathcal{P}'$ will also be identical, although their execution times will be different.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information

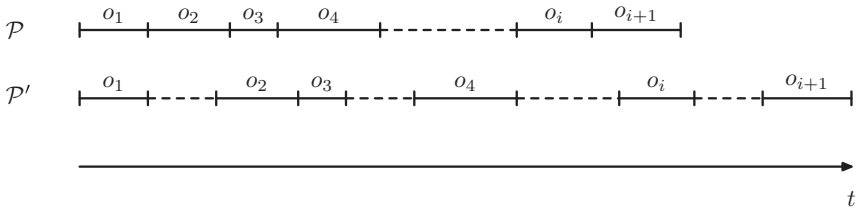

Figure 1.1. Sequential processes $\mathcal{P}$ and $\mathcal{P}'$ that are equivalent with respect to the results of computation; t denotes the time axis.

Sequential processes that are performed simultaneously and asynchronously, in which execution of operations can overlap in time, are called **concurrent processes**. Due to asynchronicity there are many possible implementations or scenarios of execution of concurrent processes. Considering these scenarios, we cannot determine in advance which operation of a given process is preceded or followed by an operation of another process, provided the processes do not synchronize their action.[1] In other words, operations of concurrent processes can be executed in any relative order[2] in different implementations. Figure 1.2 shows two of many scenarios of execution of concurrent processes $\mathcal{P}_1$ and $\mathcal{P}_2$. In the left part of the figure, operation $o_{1,2}$ of $\mathcal{P}_1$ is executed before operation $o_{2,2}$ of $\mathcal{P}_2$, whereas in the right part the execution of operations overlap in time. The admission of an arbitrary relative order of execution of operations in various implementations means in fact that no assumptions about the speed of virtual processors are made. It is justified because this speed depends on the speed and the number of real processors[3] available in a computer in which the concurrent program will be implemented.

If the actual number of real processors is at least equal to the number of virtual processors, that is to the number of processes, then the processes can be executed **in parallel** (Figure 1.3; the sequence of events is: $\underline{o_{i,1}}, \underline{o_{k,1}}, \underline{o_{j,1}}, \overline{o_{i,1}}, \underline{o_{i,2}}, \overline{o_{k,1}}, \underline{o_{k,2}}, \overline{o_{i,2}}, \underline{o_{i,3}}, \ldots$). In this case, each process is performed by an appropriate real processor. There is also a case in which only one real processor is available. Then, the individual processes are performed by **interleaving** (Figure 1.4; the sequence of events is: $\underline{o_{i,1}}, \overline{o_{i,1}}, \underline{o_{j,1}}, \overline{o_{j,1}}, \underline{o_{k,1}}, \overline{o_{k,1}}, \underline{o_{j,2}}, \overline{o_{j,2}}, \ldots$). If the concurrent program is designed correctly, then both these executions, by a single or a larger number of processors, will give the same results. The scenarios depicted in Figure 1.3 and 1.4 are extreme in terms of the number of real processors available, p, and the number of virtual processors, v. For these scenarios the following relations hold: $p \geq v$, and $p = 1$ with any v, respectively. In other scenarios we may have $1 < p < v$, where there is more real processors available but its number is less than the number of processes to be executed.

1.1.1 Communication between Processes

In general, concurrent processes are executed independently, but they can communicate with each other in specific points in time. Communication of processes can be

[1] Synchronization operations may impose a partial (as well as a total) order among operations executed in processes.
[2] Recall that within each process operations are executed sequentially.
[3] Throughout this book, by a (real) processor we mean a uniprocessor or a core as a physical device.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information

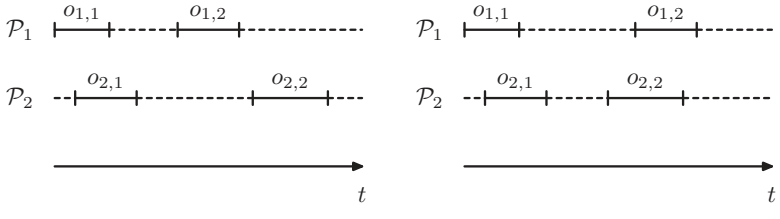

Figure 1.2. Two possible scenarios of execution of concurrent processes $\mathcal{P}_1$ and $\mathcal{P}_2$.

accomplished in two ways: employing **shared memory** or by **message passing**. In the first way, it is assumed that a shared memory is available and all virtual processes performing concurrent computation have access to it. Every processor can perform some operations on a number of variables located in a shared memory. These can be simple operations **read** or **write** on shared variables, but also more advanced ones, such as **exchange**, **test-and-set**, **compare-and-swap**, **fetch-and-add**. By making use of shared variables concurrent processes transfer to each other the results of computation, as well as synchronizing their actions. For the purpose of synchronization, it is important that these operations are atomic (see Exercise 3).

In the second way of communication, virtual processors exchange messages over **communication channels** (shared memory is not available). Each channel provides typically a two-way connection between a pair of processors. A set of communication channels makes the **interconnection network** of a system. Processors located in vertices of the network have the ability to perform computations of a concurrent program, as well as to send and receive messages from neighboring processors. In most circumstances no assumption is made concerning the latency of message delivery between a source and target processor.[4] Consequently, computations are fully **asynchronous** because both the exact points in time of execution of concurrent process operations, as well as sending and receiving messages, cannot be identified in advance.

1.1.2 Concurrent, Parallel, and Distributed Program

Concurrent processes that solve a computational problem can be implemented in three basic ways:

(i) processes are executed by a single processor by interleaving;
(ii) each process is executed by a separate processor and all the processors have access to a shared memory;
(iii) processes are executed by separate, distributed processors interconnected by communication channels.

We assume that the two processes are **concurrent**, if it is possible to execute them in parallel. These processes are **parallel**, if at any time both of them are simultaneously executed. In this context, only processes from cases (ii) and (iii) can be regarded as parallel. A **concurrent program** specifies the processes that may be executed in parallel. It also describes how the issues of synchronization and communication between

[4] Sometimes the upper limit of a latency is specified.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information

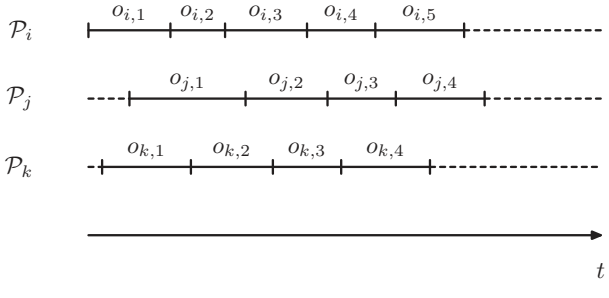

Figure 1.3. Parallel execution of operations of processes $\mathcal{P}_i$, $\mathcal{P}_j$, and $\mathcal{P}_k$.

processes are solved. Whether or not the processes are actually executed in parallel depends on the implementation. If a sufficient number of (physical) processors is available, then each process is executed by a separate processor and the concurrent program is executed as the **parallel program**. Parallel programs executed by distributed processors, for example by processors contained in a computing cluster, are called **distributed programs** (see Section 1.5, Notes to the Chapter, p. 24). Excluding cases (ii) and (iii), it is also possible to execute v processes using p processors where $p < v$; in particular, $p = 1$ may hold, as in case (i). Then some of the processes, or all of them, must be executed by interleaving. Such a way of execution can be viewed as **pseudo-parallel**.

1.2 CONCURRENCY OF PROCESSES IN OPERATING SYSTEMS

An execution of three processes by a single processor via interleaving is illustrated in Figure 1.4. Although such an execution is possible, in practice it is inefficient because the processor while passing between processes has to make a **context switch**. It involves saving the necessary data regarding the state (context) of a process, such as the contents of arithmetic and control registers, including the program counter, etc., so that execution of the process can be resumed from the point of interruption. A context switch is generally time-consuming,[5] therefore interleaving in modern operating systems is accomplished in the form of **time-sharing**.[6] It includes allocating a processor to perform more operations of a process, during a given period of time with some maximum length, for example 1 ms. In general, the tasks executed as processes do not have to be the parts of a single concurrent program. They can be independent, sequential computational tasks that should be performed on a computer.

Let us investigate an example in which the following tasks should be carried out by a single processor: the optimization of a given function, printing a file, and editing a document. Execution of these tasks may be done through assignments of consecutive periods of computation time of a processor to the optimization task, with the ability to perform character operations in the tasks of printing and editing the files. An I/O character operation does not require much computation time of the processor. Therefore, it can perform the optimization task, probably computationally intensive, interrupting its work at the times when the I/O devices (the printer and keyboard)

[5] Especially in superscalar processors equipped with pipelined instruction implementation units and large caches (see Section 5.1).
[6] Also called time-slicing.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information

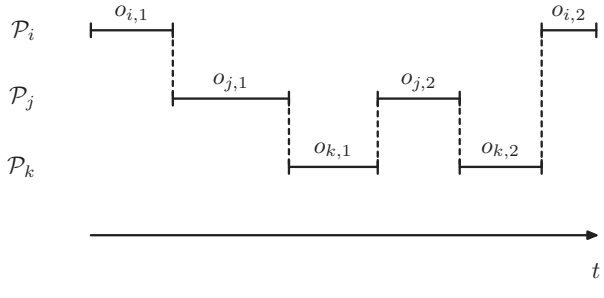

Figure 1.4. Interleaving of operations of processes—one real processor "implements" three virtual processors.

need to be serviced. Commonly, the service demands from I/O devices are requested via **interrupts**. In practice, the allocation of computation time of processors is more complex since tasks and interrupts are assigned priorities affecting the order in which computations are performed.

Note that traditional **personal computers** (PCs) equipped with a single processor operate in the way described above. The operating system of a computer supervises the computation by assigning the processor the concurrent processes to implement by time-sharing, as well as handling interrupts taking into account their priorities. Processes communicate with each other by employing the operating memory of the computer with a single address space shared by all processes.

Pseudo-parallel execution of several tasks by a single processor is referred to as **multitasking**. This way of operation is also used in computing servers with many processors, where the number of tasks greatly exceeds the number of processors. The order of tasks execution in the systems with both a single and multiple processors is determined by an operating system module designed to manage the processor's computation time, called a **task scheduling module**. In order to make best use of processors this module solves the problem of load balancing among processors (see Section 4.5).

1.2.1 Threads

As mentioned earlier, tasks that are components of a parallel program are executed as sequential processes under supervision of an **operating system**. From the implementation point of view a process is an execution entity created, supervised, and destroyed by the operating system. In order to execute, the operating system allocates to a process certain resources, such as computation time of a processor, memory space to store instructions, data, and the process stack, the set of registers, including the program counter and stack pointer.

A process can be executed by a single **thread** or by a team of cooperating threads. A thread is an execution entity able to run independently a stream of instructions.[7] If more than one tread executes a process, then the concurrent computation occurs as the result of execution of a number of instruction streams. All the threads

[7] Both a process and thread are executions entities of the operating system. In view of this similarity, a thread is sometimes referred to as a **lightweight process**, and a traditional process as a **heavyweight process**.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information

executing a process share the resources allocated to it, in particular the address space of a designated area of memory. In this memory the threads can store computation results and messages that can be read by other threads. So it allows the threads to communicate with each other. In addition to shared resources, each thread has likewise a few resources for its exclusive use, such as the stack, program counter register, memory to store private data, etc. Multiple threads can be executed concurrently by a single processor or core applying time-sharing (see p. 4). In such a case we talk about **multithreaded** execution. Multiple threads can also be executed in parallel by several processors or a multicore processor.

1.3 CORRECTNESS OF CONCURRENT PROGRAMS

Correctness of a concurrent program is more difficult to prove than correctness of a sequential program, because it is necessary to demonstrate additional properties that should have the concurrent program, namely the properties of safety and liveness. We will discuss these properties later.

The proofs of correctness of terminating[8] concurrent programs are conducted in a similar way as for sequential programs.[9] In general, a sequential program is correct if the desired relations are satisfied between the input data and output data that are results of computation. The correctness of a sequential program is expressed by a sentence of the form

$$\{p\}\, S\, \{q\},$$

where S denotes a program, and p and q are assertions called **precondition** and **postcondition**, respectively. The precondition specifies the conditions that are satisfied by the input data, or, in other words, by the state of memory with which execution of the program begins. The postcondition specifies the desired conditions to be met by the results of computation, or, in other words, by the state of memory once the program has been executed. Thus it can also be said that the program transforms the computer's memory from a specified initial state to a required final state.

The correctness of a sequential program is formulated in two stages. The sequential program S is **partially correct**, if for every terminating execution of S with the input data satisfying precondition p, the output data satisfy postcondition q. The sequential program S is **totally correct**, if it is partially correct and every execution of S with the input data satisfying precondition p terminates.

Partial correctness does not take into account whether the computation of program S terminates or not. Answering the question of whether the program will eventually terminate for all valid input data satisfying precondition p, is called a **halting problem**. Solving this problem is equivalent to showing that all iterative instructions (loops) in the program always come to an end, as it is assumed that execution times of other instructions involving primitive operations are finite. Thus in practice, the proof

[8] We consider here concurrent programs whose processes terminate. There are also concurrent programs with nonterminating processes. An example may be processes of a computer operating system, or processes of a program that monitors a continuously working device by receiving and processing data sent by its sensors.

[9] In this section we analyze the conditions of correctness of concurrent and sequential *programs*. These conditions also apply to *algorithms* underlying those programs, because it may be assumed that algorithms and programs, which are their implementations, are semantically equivalent.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information

of total correctness of the sequential program includes proving that the program is partially correct and satisfies the halting condition.

As stated earlier, a concurrent program consists of components called tasks. Components are sequential programs, so to prove the correctness of a concurrent (or parallel[10]) program, in the first place the correctness of its components must be demonstrated. Since the concurrent processes participating in the execution of a concurrent program cooperate with each other, additional properties that relate to **safety** and **liveness** must be proved. Safety properties are the conditions that should *always* be satisfied, that is for all possible implementations (or execution scenarios) of concurrent processes. In contrast, liveness properties are conditions that should be satisfied *eventually*, which means that if a given condition should be satisfied, then for every possible realization of concurrent processes, at some point of time it actually will hold.

Concurrent processes during their execution often compete with each other trying to gain access to shared resources. These may be variables in shared memory, files, disk storage, I/O devices. In such a case we are dealing with the safety property whereby in no time of the concurrent program execution a particular resource is used by more than one process. This property is called **mutual exclusion**. Another safety property is freedom from **deadlock**. The deadlock is a situation when two or more processes do not terminate, because they cannot continue their action. Let us examine one of the classic problems where the deadlock may arise. Suppose there are two concurrent processes $\mathcal{P}_1$ and $\mathcal{P}_2$, and two resources A and B. Each process needs access to both resources to perform its computation. Assume that processes $\mathcal{P}_1$ and $\mathcal{P}_2$ issued the requests and got access to resources A and B, respectively. Clearly, they are now deadlocked when asking for the second resource, because process $\mathcal{P}_1$ holds resource A, that is needed by process $\mathcal{P}_2$, and process $\mathcal{P}_2$ holds resource B that is needed by process $\mathcal{P}_1$. As a result, the processes will not terminate because none of them can continue its action.

Informally, the liveness property means that during each concurrent implementation of a program, a required condition will be satisfied at some point. Consider an example in which a number of processes compete for access to a shared resource, for example a printer. The liveness property says that if any process issues a print request, then at some point it will be assigned the printer. In other words, there will be no individual **starvation** of the process as a result of not allocating the resource to it.

Related to the liveness property is a notion of **fairness**, which is intended to limit all possible executions of concurrent processes to the fair executions. The fairness constraint[11] is the condition that prevents such execution scenarios, in which a request issued by a process continually or infinitely often, is unserved, because at all times requests of other processes are handled. There are two basic[12] types of fairness:

[10] By a parallel program we mean a concurrent program implemented by a sufficient number of physical processors (see p. 3).

[11] The fairness constraint is not the property of a concurrent program. The constraint is imposed on the system that schedules execution of concurrent processes.

[12] In addition, the **fairness in expected linear time** can be formulated in which the request issued by a process will be handled before requests of any other process are handled more than once. One can also formulate the **FIFO fairness**, where requests are handled in the order of their submissions.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information

weak and **strong**. The weak fairness condition states that if a process *continually* issues a request, then it will eventually be handled. For strong fairness this condition has the form: if a process issues its request *infinitely often*, then it will eventually be handled. Consider a process, being a part of a concurrent program P, that issues and constantly sustains the request for allocating the printer. If during each execution of P such a request will eventually be handled, then these executions are weakly fair. Suppose now that a process issues a print request, but after some time, when it is unhandled, the process withdraws the request to perform other work. If we assume that the request is issued and withdrawn infinitely often, then in the weakly fair executions it may never be handled. This occurs when the printer control process, to which requests are directed, checks whether it is a print request in the moments when it is just withdrawn. To prevent this kind of situation the condition of strong fairness is formulated, which requires that the request issued infinitely often by a process of P will eventually be handled.

1.4 SELECTED PROBLEMS IN CONCURRENT PROGRAMMING

When designing and implementing concurrent systems the problems that do not occur in sequential systems must be solved. They are concerned mainly with cooperation of processes (tasks), which are essential for proper operation of systems. In this section, we discuss some problems of concurrent programming and ways to solve them.

1.4.1 The Critical Section Problem

The critical section problem occurs when a group of processes compete for the resource, wherein at any given time only one process can have access to it. Resources of this type[13] are variables in shared memory, database records, files, physical devices. Note that the simultaneous use of a printer by several processes would lead to illegible printing reports. Similarly, modifying the same database records by multiple processes at the same time can cause data inconsistency. The fragment of a process, or more accurately, of a task in which it makes use of a resource is called a **critical section**. Exclusive use of a resource is achieved by ensuring that at any time only one task executes its critical section. In other words, instructions of two or more critical sections cannot be interleaved.

For example, let T_1 and T_2 be tasks. If task T_1 wants to use the resource, it must acquire permission for its use, which means the resource must be assigned to the task. After the assignment of the resource, the task uses it in its critical section, and after completing the section it releases the resource. When the resource is released by task T_1, it can be assigned to task T_2. If during the use of the resource by task T_1, task T_2 issues the request for this resource, it must wait until the resource is released by task T_1.

The critical section problem can be solved employing semaphores. A **semaphore** s is a compound data structure with two fields: $s.w$ and $s.q$, where the field $s.w$ takes

[13] Typically, only one process at a time can use a resource. However, there are resources that can be shared by multiple processes. These resources are composed of a number of units, such as memory cells, disk sectors, printers. Processes can apply for allocation of one or more units of the resource.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information


```
 1  s: semaphore := (1, ∅); -- binary semaphore
 2  task T1; -- task specifications
 3  task T2;
 4  task body T1 is
 5  begin
 6      loop
 7          wait(s); -- pre-protocol
 8          CRITICAL SECTION
 9          signal(s); -- post-protocol
10          -- remainder of the task
11      end loop;
12  end T1;
13  task body T2 is
14  begin
15      loop
16          wait(s); -- pre-protocol
17          CRITICAL SECTION
18          signal(s); -- post-protocol
19          -- remainder of the task
20      end loop;
21  end T2;
```

Figure 1.5. Solving the critical section problem with a binary semaphore.

nonnegative integer values, and the values of the field $s.q$ are sets of tasks (processes). On semaphore s the following operations are defined:

wait(s): If $s.w > 0$, then $s.w := s.w - 1$, otherwise suspend execution of the task performing operation *wait* and add it to set $s.q$. The task added to set $s.q$ is said to be blocked on semaphore s.

signal(s): If set $s.q$ of tasks is nonempty, then delete and unblock one of the tasks from $s.q$ and resume its execution, otherwise $s.w := s.w + 1$.

The operations *wait* and *signal*[14] are atomic, which means that the actions within these operations cannot be interleaved with any other instructions. In the sequel, we assume that the set of blocked tasks on a semaphore is organized in a FIFO queue. Such a semaphore is called **blocked-queue semaphore**. Before a semaphore s is used, it must be initialized by assigning to the field $s.w$ any nonnegative integer, $s.w \geq 0$, and to component $s.q$ the empty queue $\emptyset$. If the component $s.w$ of a semaphore may take any nonnegative integer value, then the semaphore is called **general**. If this component may take only values 0 or 1, then the semaphore is called **binary**.[15] Note that performing the operation *signal* on a binary semaphore with the integer component equals 1 is an error. If the semaphore queue is empty, then the component should be increased by 1, which is unacceptable (other versions of semaphores are discussed in the notes to this chapter).

An example of solving the critical section problem for two tasks T_1 and T_2 executed independently of each other is depicted in Figure 1.5. The tasks are written

[14] It is also said that the operation *wait* and *signal* lowers and raises a semaphore, respectively.

[15] A binary semaphore is also called **mutex**. This term is used in the Pthreads and java.util.concurrent libraries.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-17439-9 — Introduction to Parallel Computing
Zbigniew J. Czech
Excerpt
More Information

with the syntax of Ada language.[16] In line 1, semaphore s is declared by applying the type *semaphore* defined as follows:

```
type semaphore is
    record
        w: natural;
        q: queue;
    end record;
```

The specifications of tasks T_1 and T_2 are given in lines 2–3, and their bodies, respectively, in lines 4–12 and 13–21. Both tasks use the infinite loop of the syntax: **loop** … **end loop**. In each run of the loop the tasks perform the critical sections (lines 8 and 17). Suppose that in one of the tasks, say T_1, the critical section should be performed. Then, operation *wait*(s) in line 7 is executed. If $s.w = 1$ the integer component of semaphore s is set to 0 and the task begins executing the critical section. After its execution, component $s.w$ is set back to 1 by operation *signal*(s) in line 9. If task T_2 tries to execute its critical section before the completion of the critical section by task T_1, then because $s.w = 0$ task T_2 will be blocked and placed in the queue by operation *wait*(s) in line 16. Only when task T_1 completes its critical section, operation *signal*(s) in line 9 will the execution of task T_2 be resumed.

Now we will show that solution in Figure 1.5 has the property of mutual exclusion. Suppose that tasks T_1 and T_2 want simultaneously execute critical sections. To this end, they perform operation *wait*(s). By definition, this operation is atomic so only one of the tasks will enter the critical section. The second one will be blocked and put to the queue. Similarly, if during execution of the critical section by one of the tasks, the second task will also attempt to execute the critical section, it will be blocked, because the integer component of semaphore s will be equal to 0.

The solution in Figure 1.5 is also free of deadlock. Assume that both tasks T_1 and T_2 have been blocked by operation *wait*(s). Then $s.w = 0$ must hold, which means that one of the tasks executes the critical section. This is contrary to the assumption that both tasks are blocked. In the analyzed solution there is also no starvation. Suppose that task T_1 is in the queue, and task T_2 executes the critical section. After completing the critical section, task T_2 performs operation *signal*(s) resulting in resumption of execution of task T_1. Therefore it is impossible that task T_1 will continue to be blocked and task T_2 again will be able to execute its critical section.

From the above considerations it appears that the safety and liveness conditions are met, and so the solution in Figure 1.5 is correct. This solution can be easily generalized to n tasks for $n > 2$ of the form depicted in Figure 1.6 with unchanged specification of the semaphore. The generalized solution has the same properties as the previous one. The property of freedom from starvation of competing tasks results from the use of a FIFO queue within the semaphore. In the worst case, any task will receive a resource after it is used by $n - 1$ tasks that may precede it in the queue of blocked tasks (see Exercise 4).

Note that the solution to the critical section problem in Figure 1.5 relies on suitable synchronization of execution of critical sections by tasks T_1 and T_2. The structure of synchronization is the same for both tasks. Before entering the critical section, the

[16] In Ada a comment begins with characters – –, and ends with a newline character. In this book, comments will also be enclosed within characters { and }, or /* and */.

© in this web service Cambridge University Press www.cambridge.org