chapter 9

# Process Synchronization

To fulfill a common goal, interacting processes need to share data or coordinate their activities with each other. Data access synchronization ensures that shared data do not lose consistency when they are updated by several processes. It is implemented by ensuring that accesses to shared data are performed in a mutually exclusive manner. Control synchronization ensures that interacting processes perform their actions in a desired order. Computer systems provide *indivisible instructions* (also called *atomic instructions*) to support data access and control synchronization.

We begin this chapter with a discussion of *critical sections*, which are used to access shared data in a mutually exclusive manner. Following this discussion, we introduce some classic problems in process synchronization. These problems are abstractions of practical synchronization problems faced in various application domains. We analyze synchronization requirements of these problems and study important issues involved in their implementation.

In the remainder of the chapter, we discuss various synchronization facilities provided in programming languages and operating systems. These include *semaphores*, *conditional critical regions*, and *monitors*. We discuss their use to fulfill the process synchronization requirements in the classic problems.

## 9.1 DATA ACCESS SYNCHRONIZATION AND CONTROL SYNCHRONIZATION

In Chapter 3 we discussed how implementing an application using a set of interacting processes may provide execution speed-up and improved response times. To work towards a common goal, interacting processes need to coordinate their activities with respect to one another. We defined two kinds of synchronization, called data access synchronization (see Def. 3.6) and control synchronization (see Def. 3.7) for this purpose. Table 9.1 summarizes their main features.

In Section 3.6.1, we defined a race condition as follows (see Def. 3.5): Let $a_i$ and $a_j$ be operations on shared data $d_s$ performed by two processes $P_i$ and $P_j$. $f_i(d_s)$

Materiał chroniony prawem autorskim

**Table 9.1** Features of data access synchronization and control synchronization

| | |
|---|---|
| Data access synchronization | Race conditions (see Def. 3.5) arise if processes access shared data in an uncoordinated manner. Race conditions are not reproducible, hence they make debugging difficult. Data access synchronization is used to access shared data in a mutually exclusive manner. It avoids race conditions and safeguards consistency of shared data. |
| Control synchronization | Control synchronization is needed if a process should perform some action $a_i$ only after some other processes have executed a set of actions $\{a_j\}$ or only when a set of conditions $\{c_k\}$ hold. |

represents the value of $d_s$ resulting from changes, if any, caused by operation $a_i$. $f_j(d_s)$ is defined analogously. A race condition arises if the result of execution of $a_i$ and $a_j$ in processes $P_i$ and $P_j$ is other than $f_i(f_j(d_s))$ or $f_j(f_i(d_s))$. In Example 3.9, we saw how a race condition arises in the airline reservations system when processes update the value of *nextseatno* in an uncoordinated manner—its value increased by only 1 even though two processes added 1 to it!

We also discussed some examples of control synchronization in Section 3.6.2. The real time system of Figure 3.3 uses three processes to copy samples received from a satellite onto a disk. Here, the main process waits for its child processes to complete before terminating itself. The child processes synchronize their activities such that a new sample is copied into a buffer entry only when the buffer entry is empty and contents of a buffer entry are copied to the disk only when the buffer entry contains a new sample.

**Implementing synchronization** The basic technique used to implement synchronization is to block a process until an appropriate action is performed by another process or until a condition is fulfilled. Thus, data access synchronization is implemented by blocking a process until another process finishes accessing shared data. Control synchronization is implemented by blocking a process until another process performs a specific action.

While implementing process synchronization, it is important to note that the effective execution speed of a process cannot be estimated due to factors like time slicing, process priorities and I/O activities in a process. It is similarly impossible to know the relative execution speeds of processes, so a synchronization scheme must function correctly irrespective of the relative execution speeds of processes.

## 9.2 CRITICAL SECTIONS

Race conditions on shared data $d_s$ (see Def. 3.5) arise because operations on $d_s$ are performed concurrently by two or more processes. The notion of a *critical section*

Materiał chroniony prawem autorskim

(CS) is introduced to avoid race conditions.

**Definition 9.1 (Critical Section (CS))** *A* critical section *for a data item $d_s$ is a section of code that should not be executed concurrently either with itself or with other critical section(s) for $d_s$.*

If some process $P_i$ is executing a critical section for $d_s$, another process wishing to execute a critical section for $d_s$ will have to wait until $P_i$ finishes executing its critical section. Thus, a CS for a data item $d_s$ is a mutual exclusion region with respect to accesses to $d_s$—at most one process can execute a CS for $d_s$ at any time.

Race conditions on a data item are avoided by performing all its update operations inside a CS for the data item. Further, to ensure that processes see consistent values of a data item, all its uses should also occur inside a CS for the data item.

We mark a CS in a piece of code by a dashed rectangular box. Note that processes may share a single copy of code, in which case a single CS for $d_s$ may exist in an application. Alternatively, the code for each process may contain one or more CSs for $d_s$. Definition 9.1 covers both situations. A process that is executing a CS is said to be 'in a CS'. We also use the terms 'enter a CS' and 'exit a CS' for situations where a process begins and ends the execution of a CS.

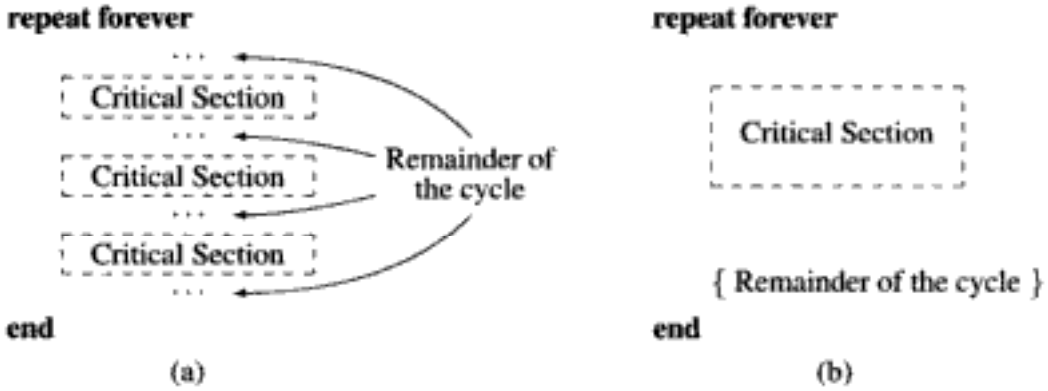


**Fig. 9.1** (a) A process with many critical sections, (b) A simpler way of depicting this process

Figure 9.1(a) shows a process containing several critical sections. These critical sections may be used to access the same or different shared data items. Thus, a process may contain several critical sections for the same data item. The process contains a cycle. In each iteration, it enters a critical section when it needs to access a shared data item. At other times, it executes other code in its logic, which together constitutes "remainder of the cycle". For simplicity, whenever possible, we use the simple process form shown in Figure 9.1(b) to depict a process.

Example 9.1 illustrates use of a CS to avoid race conditions in the airline reservation system.

**Example 9.1** Figure 9.2 shows use of critical sections in the airline reservation system of Figure 3.22. Each process contains a CS in which it accesses and updates

Materiał chroniony prawem autorskim

the shared variable *nextseatno*. Absence of race conditions can be shown as follows: Let $f_i(nextseatno)$ and $f_j(nextseatno)$ represent the value of *nextseatno* resulting from changes, if any, caused by execution of critical sections in $P_i$ and $P_j$, respectively. Let $P_i$ and $P_j$ attempt to execute their critical sections concurrently. From the definition of CS, it follows that only one of them can execute its CS at any time, so the resulting value of *nextseatno* will be either $f_i(f_j(nextseatno))$ or $f_j(f_i(nextseatno))$. From Def. 3.5, a race condition does not arise.

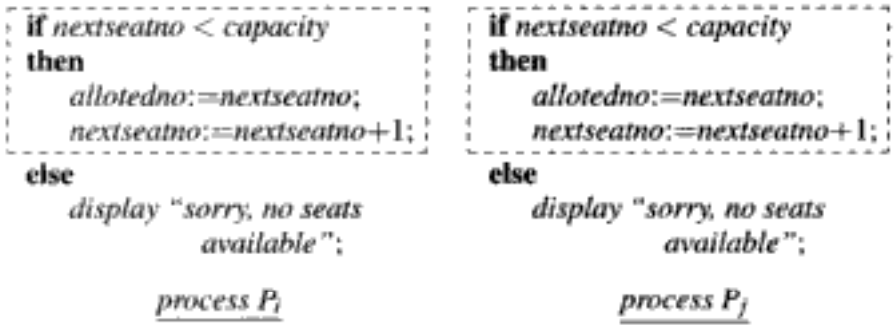


**Fig. 9.2** Use of CS in airline reservation

If processes in an application make frequent use of a shared data item $d_s$, execution of critical sections for $d_s$ may become a performance bottleneck. This would cause delays in the execution of processes, and would adversely affect performance of the application. The severity of this problem is reduced if processes do not spend too much time inside a CS. Both processes and the kernel must cooperate to ensure this. A process must not execute for too long inside a CS. While in a CS, it must avoid making system calls that might put it in a *blocked* state. The kernel must not preempt a process that is engaged in executing a CS. This condition is hard to meet as it requires the kernel to know whether a process is inside a CS at any moment. It cannot be met if processes implement critical sections on their own, i.e., without involving the kernel. Nevertheless, in this Chapter we will assume that a process spends only a short time inside a critical section.

### 9.2.1 Properties of a CS Implementation

A CS implementation for a data item $d_s$ is like a scheduler for a resource. It must keep track of all processes that wish to enter a CS for $d_s$ and select a process for entry in a CS in accordance with the notions of mutual exclusion, efficiency and fairness. Table 9.2 summarizes the essential properties an implementation of CS must possess. We discuss these properties in the following.

*Mutual exclusion* guarantees that more than one process would not be in a CS for $d_s$ at any time. Apart from correctness, a CS implementation should also guarantee that any process wishing to enter a CS would not be delayed indefinitely, i.e., *starvation* would not occur. This guarantee is in two parts, and is provided by the other two properties.

The *progress* property ensures that if some processes are interested in entering

Materiał chroniony prawem autorskim

**Table 9.2** Essential properties of a CS implementation

| Property | Description |
|---|---|
| Correctness | At any moment, at most one process may execute a CS for a data item $d_s$. |
| Progress | When a CS is not in use, one of the processes wishing to enter it will be granted entry to the CS. |
| Bounded wait | After a process $P_i$ has indicated its desire to enter a CS for $d_s$, the number of times other processes can gain entry to a CS for $d_s$ ahead of $P_i$ is bounded by a finite integer. |

a CS for a data item $d_s$, one of them will be granted entry if no process is currently inside a CS for $d_s$—that is, the privilege to enter a CS for $d_s$ will not be reserved for some process that is not interested in entering a CS at present. If this property were not satisfied, processes wishing to use a CS may be delayed until some specific process $P_s$ decides to use a CS. Process $P_s$ may never use a CS for $d_s$, so absence of this property might delay a process indefinitely.

The progress property ensures that the CS implementation will necessarily choose one of the requesting processes to enter a CS. However, even this is not enough to ensure that a requesting process $P_i$ gains entry to a CS in finite time because a CS implementation may ignore $P_i$ while choosing a process for entry to a CS. The *bounded wait* property ensures that this does not happen by limiting the number of times other processes can be favored over $P_i$. Since execution of a CS takes a short time, this property ensures that every requesting process will gain entry to its CS in finite time.

### 9.2.2 The Problem of Busy Wait

A process may implement a CS for a data item $d_s$ using the following simple code:

```
while (some process is in a CS for d_s)
        { do nothing }
    Critical
    section
```

In the **while** loop, the process checks if some other process is in a CS for the same data item. If so, it keeps looping until the other process exits its CS.

A *busy wait* is a situation in which a process repeatedly checks if a condition that would enable it to get past a synchronization point is satisfied. It ends only when the condition is satisfied. Thus a busy wait keeps the CPU busy in executing a process

Materiał chroniony prawem autorskim

even as the process does nothing! Lower priority processes are denied use of the CPU, so their response times suffer. System performance also suffers.

A busy wait also causes a curious problem in a uniprocessor system. Consider the following situation: A high priority process $P_i$ is blocked on an I/O operation and a low priority process $P_j$ enters a CS for data item $d_s$. When $P_i$'s I/O operation completes, $P_j$ is preempted and $P_i$ is scheduled. If $P_i$ now tries to enter a CS for $d_s$ using the **while** loop described earlier, it would face a busy wait. This busy wait denies the CPU to $P_j$, hence it is unable to complete its execution of the CS and exit. This prevents $P_i$ from entering its CS. Processes $P_i$ and $P_j$ now wait for each other indefinitely. This situation is called a *deadlock*. Because a high priority process waits for a process with a low priority, this situation is also called *priority inversion*. The priority inversion problem is addressed using the *priority inheritance protocol*, wherein a low priority process that holds a resource temporarily acquires the priority of the highest priority process that needs the resource. In our example, process $P_j$ would temporarily acquire the priority of process $P_i$, which would enable it to get scheduled and exit from its critical section. However, use of the priority inheritance protocol in these situations is impractical because it would require the kernel to note minute details of the operation of processes.

To avoid busy waits, a process waiting for entry to a CS should be put into the *blocked* state. Its state should be changed to *ready* only when it can be allowed to enter its CS.

### 9.2.3 History of CS Implementations

Historically, implementation of critical sections has gone through three important stages—algorithmic approaches, software primitives and concurrent programming constructs. *Algorithmic approaches* depended on a complex arrangement of checks to ensure mutual exclusion between processes using a CS. Correctness of a CS implementation depended on correctness of these checks, and was hard to prove due to the logical complexity of the checks. This was a serious deficiency in the development of large systems containing concurrent processes. However, a major advantage of this approach was that no special hardware, programming language or kernel features were required to implement a CS.

A set of *software primitives* for mutual exclusion (e.g., the P and V primitives of Dijkstra) were developed to overcome the logical complexity of algorithmic implementations. These primitives were implemented using some special architectural features of a computer system, and possessed useful properties for implementing a CS. These properties could also be used to construct proofs of correctness of a concurrent system. However, experience with such primitives showed that ease of use and proof of correctness still remained major obstacles in the development of large concurrent systems.

Development of *concurrent programming constructs* was the next important step in the history of CS implementations. These constructs used data abstraction and

Materiał chroniony prawem autorskim

encapsulation features specifically suited to the construction of concurrent programs. They had well defined semantics that were enforced by the language compiler. This feature made construction of large concurrent systems more practical.

We discuss algorithmic approaches to CS implementation in Section 9.7. Sections 9.8–9.9 discuss programming language primitives and constructs. Section 9.10 discusses the abstraction and encapsulation facilities for concurrent programming provided by monitors.

## 9.3 RACE CONDITIONS IN CONTROL SYNCHRONIZATION

Processes use control synchronization to coordinate their activities with respect to one another. A frequent requirement in process synchronization is that a process $P_i$ should perform an action $a_i$ only after process $P_j$ has performed some action $a_j$. A pseudo-code for such processes is shown in Figure 9.3. This synchronization requirement is met using the technique of *signaling*.

| {Perform operation $a_i$ only after $P_j$ performs operation $a_j$} | Perform operation $a_j$ |
|---|---|
| ... | ... |
| *Process* $P_i$ | *Process* $P_j$ |

**Fig. 9.3** Processes requiring control synchronization

**Signaling** Figure 9.4 shows how signaling is performed. The synchronization data consists of two boolean variables. Variable *action_aj_performed* is a flag to indicate whether process $P_j$ has performed action $a_j$. Variable *pi_blocked* is a flag to indicate whether process $P_i$ has blocked itself pending execution of action $a_j$ by process $P_j$. It is assumed that the kernel supports system calls to block and activate a process.

Process $P_i$ consults variable *action_aj_performed* to check if process $P_j$ has already performed action $a_j$. If this is not the case, it sets *pi_blocked* to *true* and makes a system call to block itself. It goes ahead to perform action $a_i$ if $a_j$ has already been performed. Process $P_j$ performs action $a_j$ and checks if process $P_i$ has blocked itself before performing action $a_i$. If so, it makes a system call to activate $P_i$. Otherwise, it sets *action_aj_performed* to *true* so that process $P_i$ will know that $P_j$ has performed action $a_j$.

Table 9.3 shows an execution of the system in which process $P_i$ faces indefinite blocking due to a race condition. Process $P_i$ checks the value of *action_aj_performed* and finds that action $a_j$ has not been performed. It is poised to set the variable *pi_blocked* to *true* when it gets preempted. Process $P_j$ is now scheduled. It performs action $a_j$ and checks whether process $P_i$ is blocked. Since *pi_blocked* is *false*, it simply sets *action_aj_performed* to *true* and continues its execution. Sometime later, $P_i$ is scheduled. It sets *pi_blocked* to *true* and makes a system call to block itself. Process $P_i$ now sleeps for ever!

Consider the **if** statements in processes $P_i$ and $P_j$ as the operations $f_i$ and $f_j$ on

Materiał chroniony prawem autorskim

```
var
        operation_aj_performed : boolean;
                pi_blocked : boolean;
begin
        operation_aj_performed := false;
        pi_blocked := false;

Parbegin
    ...                                     ...
    if operation_aj_performed = false       {perform operation a_j}
    then                                    if pi_blocked = true
        pi_blocked := true;                 then
        block(P_i);                             pi_blocked := false;
    {perform operation a_i}                     activate(P_i);
    ...                                     else
    ...                                         operation_aj_performed := true;
    ...                                     ...
Parend;
end.

        process P_i                             process P_j
```

**Fig. 9.4** An attempt at signaling through boolean variables

**Table 9.3** Race condition in process synchronization

| Time | Actions of process $P_i$ | Actions of process $P_j$ |
|---|---|---|
| $t_1$ | **if** *action_aj_performed = false* **then** | |
| $t_2$ | | {*perform action* $a_j$} |
| $t_3$ | | **if** *pi_blocked = true* **then** |
| $t_4$ | | *action_aj_performed := true* |
| ⋮ | | |
| $t_{20}$ | *pi_blocked := true;* | |
| $t_{21}$ | *block* ($P_i$); | |

the state of the system. The result of their execution should have been one of the following: process $P_i$ blocks itself, gets activated by $P_j$ and performs action $a_i$; or process $P_i$ finds that $P_j$ has already performed $a_j$ and goes ahead to perform action $a_i$. However, as seen in Table 9.3, process $P_i$ blocks itself and does not get activated. From Def. 3.5, this is a race condition.

The race condition arises because process $P_i$ is preempted after it checks if *action_aj_performed = true*, but before it can set *pi_blocked = true*. The race condition can be avoided if we can guarantee that $P_i$ would be able to complete both these actions

Materiał chroniony prawem autorskim

before getting preempted. We introduce the notion of an indivisible operation (also called an atomic operation) to achieve this.

**Definition 9.2 (Indivisible operation)** *An* indivisible operation *on a set of data items $\{d_s\}$ is an operation that cannot be executed concurrently with any other operation involving a data item included in $\{d_s\}$.*

The race condition shown in Table 9.3 would not arise if the **if** statements in Figure 9.4 are implemented as indivisible operations, because if process $P_i$ finds *action_aj_performed* = *false*, it would be able to set *pi_blocked* = *true* without getting preempted. We could achieve this by defining two indivisible operations *check_action_aj_performed* and *post_action_aj_performed* to perform the **if** statements of processes $P_i$ and $P_j$, respectively. Figure 9.5 shows a signaling arrangement using these indivisible operations. Figure 9.6 shows details of the indivisible operations *check_action_aj_performed* and *post_action_aj_performed*. When *action_aj_performed* is *false*, indivisible operation *check_aj* is deemed to be complete after process $P_i$ is blocked; it would enable process $P_j$ to perform operation *post_aj*.

```
var
    action_aj_performed : boolean;
          pi_blocked : boolean;

begin
    action_aj_performed := false;
    pi_blocked := false;
    Parbegin
      ...                                  ...
      check_aj;                            {perform action a_j}
      {perform action a_i}                 post_aj;
    Parend;
    end.

          process P_i                          process P_j
```

**Fig. 9.5** Control synchronization by signaling using indivisible operations

An indivisible operation on $\{d_s\}$ is like a critical section on $\{d_s\}$. However, we differentiate between them because a CS has to be explicitly implemented in a program, whereas the hardware or software of a computer system may provide indivisible operations as its primitive operations. In the next Section we discuss how indivisible operations on semaphores can be used to implement data access synchronization and control synchronization without race conditions.

## 9.4 IMPLEMENTING CRITICAL SECTIONS AND INDIVISIBLE OPERATIONS

Process synchronization requires critical sections or signaling operations that are indivisible. These are implemented using indivisible instructions provided by com-

Materiał chroniony prawem autorskim

```
procedure check_aj
begin
    if action_aj_performed = false
     then
        pi_blocked := true;
        block (Pi)
end;

procedure post_aj
begin
    if pi_blocked = true
     then
        pi_blocked := false;
        activate(Pi)
    else
        action_aj_performed := true;
end;
```

**Fig. 9.6** Indivisible operations for signaling

puter systems, and lock variables.

**Indivisible instructions** In Section 3.6.1 we saw how race conditions can arise if processes execute a load-add-store instruction sequence on shared data. If a computer system contains more than one CPU, race conditions can arise even during execution of a single instruction that makes more than one access to a memory location, for example, an instruction that increments the value of a variable.

Since mid 1960's, computer systems have provided special features in their architecture to avoid race conditions while accessing a memory location containing shared data. The basic theme is that all accesses to a memory location made by one instruction should be implemented without permitting another CPU to access the same location. Two popular techniques used for this purpose are locking the memory bus (e.g., in Intel 80x86 processors) and providing special instructions that avoid race conditions (e.g., in IBM/370 and M68000 processors). We will use the term *indivisible instruction* as a generic term for all such features.

**Use of a lock variable** A *lock variable* is used to bridge the gap between critical sections or indivisible operations, and indivisible instructions provided in a computer system. The basic idea is to close a lock at the start of a critical section or an indivisible operation and open it at the end of the critical section or the indivisible operation. A close-the-lock operation fails if the lock is already closed by another process. This way only one process can execute a critical section or an indivisible operation. Race conditions on the lock variable are avoided by using indivisible instructions to access the lock variable.

A process that fails to close a lock must retry the operation. This scheme involves

Materiał chroniony prawem autorskim

a busy wait; however, it does not last for long—it lasts only until the process that has closed the lock finishes executing a critical section or an indivisible operation, both of which require only a short time.

Figure 9.7 illustrates how a critical section is implemented using an indivisible instruction and a lock variable. The indivisible instruction performs the actions indicated in the dashed box—it tests the value of the lock and loops back to itself if the lock is *closed*, else it closes the lock. We illustrate use of two indivisible instructions—called test-and-set and swap instructions—to implement critical sections and indivisible operations in the following.

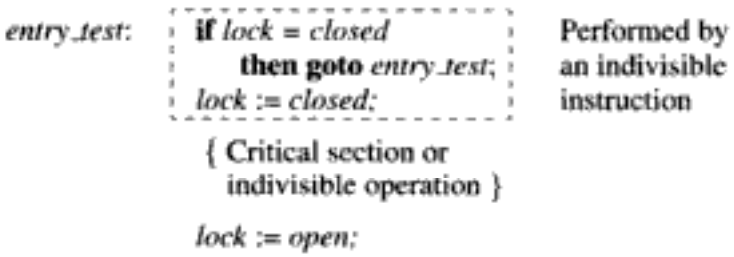


**Fig. 9.7** Implementing a critical section or indivisible operation using a lock variable

**Test-and-set (TS) instruction** This indivisible instruction in the IBM/370 performs two actions. It 'tests' the value of a memory byte such that the condition code indicates whether the value was zero or nonzero. It also sets all bits in the byte to 1s. No other CPU can access the memory byte until both actions are complete. This instruction can be used to implement the statements enclosed in the dashed box in Figure 9.7.

```
LOCK          DC    X'00'           Lock is initialized to open
ENTRY_TEST    TS    LOCK            Test-and-set lock
              BC    7, ENTRY_TEST   Loop if lock was closed

              ...                   { Critical section or
                                      indivisible operation }

              MVI   LOCK, X'00'     Open the lock (by moving 0s)
```

**Fig. 9.8** Implementing a critical section or indivisible operation using test-and-set

Figure 9.8 is an IBM/370 assembly language program that shows how indivisibility of *wait* and *signal* operations is achieved. The first line in the assembly language program declares variable LOCK and initializes it to hexadecimal 0. LOCK is used as a lock variable with the convention that a non-zero value in LOCK implies that the lock is *closed*, and a zero value implies that it is *open*. The TS instruction sets the condition code according to the value of LOCK and also changes its value to *closed*. The condition code indicates whether the value of lock was *closed* before the instruction was executed. The branch instruction BC 7, TEST checks the condition code

Materiał chroniony prawem autorskim

and loops back to the TS instruction if the lock was *closed*. This way a program (or programs) that find the lock to be *closed* execute the loop in a busy wait until lock is set to *open*. The MVI instruction puts 0s in all bits of LOCK, i.e., it opens the lock. TS is an indivisible instruction, so opening the lock would enable only one process looping at TEST to proceed.

**Swap instruction** The swap instruction exchanges contents of two memory locations. It is an indivisible instruction, hence no other CPU can access any of the locations during swapping. Figure 9.9 shows how a critical section or an indivisible operation can be implemented using the swap instruction. (For convenience, we use the same coding conventions as used for the TS instruction.) The temporary location TEMP is set to a non-zero value and its contents are swapped with LOCK. This action closes the lock. Old value of the lock is now available in TEMP. It is tested to see if the lock was already closed. If so, the process loops on the swap-and-compare action until LOCK is set to *open*. The process executing the critical section or indivisible operation opens the lock at the end of the operation. This action enables one process to get past the BC instruction.

```
TEMP         DS     1              Reserve one byte for TEMP
LOCK         DC     X'00'          Lock is initialized to open
             MVI    TEMP, X'FF'    X'FF' is used to close the lock
ENTRY_TEST   SWAP   LOCK, TEMP
             COMP   TEMP, X'00'    Test old vaue of lock
             BC     7, ENTRY_TEST  Loop if lock was closed

             ...                   { Critical section or
                                     indivisible operation }

             MVI    LOCK, X'00'    Open the lock
```

**Fig. 9.9** Implementing a critical section or indivisible operation using a swap instruction

## 9.5 CLASSIC PROCESS SYNCHRONIZATION PROBLEMS

As discussed in Sections 9.2 and 9.3, critical sections and signaling are the key elements of process synchronization. A solution to a process synchronization problem should use a suitable combination of these elements. It must also possess three important properties:

- Correctness
- Maximum concurrency
- No busy waits.

Correctness criteria depend on the nature of a problem. These criteria project requirements concerning data access synchronization and control synchronization of interacting processes. Maximum concurrency within an application is needed to provide execution speed-up and good response times. To achieve it, processes should be

Materiał chroniony prawem autorskim

able to execute freely when they are not blocked due to synchronization requirements. As discussed in Section 9.2.2, busy waits are undesirable because they lead to degradation of system performance and response times, so synchronization should be performed by blocking a process rather than through busy waits.

In this Section, we analyze some classic problems in process synchronization and discuss issues (and common mistakes) in designing their solutions. In later Sections we implement their solutions using various synchronization features provided in programming languages.

### 9.5.1 Producers–Consumers With Bounded Buffers

A producers–consumers system with bounded buffers consists of an unspecified number of producer and consumer processes and a finite pool of buffers (see Figure 9.10). Each buffer is capable of holding one record of information—it is said to become *full* when a producer writes into it, and *empty* when a consumer copies out a record contained in it; it is empty to start with. A producer process produces one record at a time and writes it into the buffer. A consumer process consumes information one record at a time.

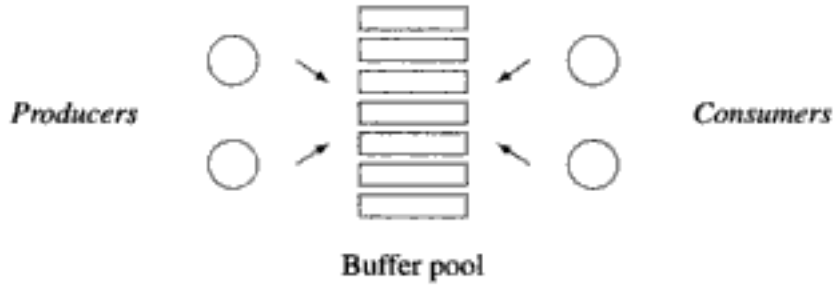


**Fig. 9.10** Producers and consumers

A producers–consumers system with bounded buffers is a useful abstraction for many practical synchronization problems. A print service is a good example in the OS domain. A print daemon is a consumer process. A fixed sized queue of print requests is the bounded buffer. A process that adds a print request to the queue is a producer process.

A solution to the producers–consumers problem must satisfy following conditions:

1. A producer must not overwrite a full buffer.
2. A consumer must not consume an empty buffer.
3. Producers and consumers must access buffers in a mutually exclusive manner.

The following condition is sometimes imposed:

4. Information must be consumed in the same order in which it is put into the buffers, i.e., in FIFO order.

Materiał chroniony prawem autorskim

```
begin
Parbegin
   var produced : boolean;
   repeat
      produced := false;
      while produced = false
         if an empty buffer exists
         then
            { Produce in a buffer }
            produced := true;
      { Remainder of
        the cycle }
   forever;
Parend
end.
```

*Producer*

```
var consumed : boolean;
repeat
   consumed := false ;
   while consumed = false
      if a full buffer exists
      then
         { Consume a buffer }
         consumed := true;
   { Remainder of
     the cycle }
forever;
```

*Consumer*

**Fig. 9.11** Solution outline for producers–consumers containing busy waits

Figure 9.11 shows an outline for the producers–consumers problem. Producer and consumer processes access a buffer inside a critical section. A producer enters its CS and checks to see whether an empty buffer exists. If so, it produces into that buffer, else it merely exits from its CS. This sequence is repeated until it finds an empty buffer. The boolean variable *produced* is used to break out of the **while** loop after the producer produces in the empty buffer. Analogously, a consumer makes repeated checks until it finds a full buffer to consume from.

An undesirable aspect of this outline is the presence of busy waits. To avoid a busy wait, a producer process should check for existence of an empty buffer only once. If none exists, the producer should be blocked until an empty buffer becomes available. When a consumer consumes from a buffer, it should activate a producer that is waiting for an empty buffer. Similarly, a producer must activate a consumer waiting for a full buffer. A CS cannot provide these facilities hence we must look elsewhere for a solution to the problem of busy waits.

When we re-analyze the producers–consumers problem in this light, we notice that though it involves mutual exclusion between a producer and a consumer using the same buffer, it is really a signaling problem. After producing a record in a buffer, a producer should signal a consumer that wishes to consume the record from the same buffer. Similarly, after consuming a record in a buffer, a consumer should signal a producer that wishes to produce a record in that buffer. These requirements can be met using the signaling arrangement shown in Figure 9.5.

An improved outline using this approach is shown in Figure 9.12 for a simple producers–consumers system that consists of a single producer, a single consumer and a single buffer. The operation *check_b_empty* performed by the producer blocks it if the buffer is full, while the operation *post_b_full* sets *buffer_full* to *true* and activates

Materiał chroniony prawem autorskim

```
var
        buffer : ... ;
        buffer_full : boolean;
        producer_blocked, consumer_blocked : boolean;
begin
        buffer_full := false;
        producer_blocked := false;
        consumer_blocked := false;
Parbegin
   repeat                                    repeat
      check_b_empty;                            check_b_full;
      { Produce in the buffer }                 { Consume from the buffer }
      post_b_full;                              post_b_empty;
      { Remainder of the cycle }                { Remainder of the cycle }
   forever;                                  forever;
Parend;
end.
```

*Producer* *Consumer*

**Fig. 9.12** An improved outline for a single buffer producers–consumers system using signaling

the consumer if the consumer is blocked for the buffer to become full. Analogous operations *check_b_full* and *post_b_empty* are defined for use by the consumer process. The boolean flags *producer_blocked* and *consumer_blocked* are used by these operations to note if the producer or consumer process is blocked at any moment. Figure 9.13 shows details of the indivisible operations. This outline will need to be extended to handle multiple buffers or multiple producer/consumer processes. We discuss this aspect in Section 9.8.2.

### 9.5.2 Readers and Writers

A readers–writers system consists of a set of processes using some shared data. A process that only reads the data is a *reader*; a process that modifies or updates it is a *writer*. We use the terms *reading* and *writing* for accesses to the shared data made by reader and writer processes, respectively. The correctness conditions for the readers–writers problem are as follows:

1. Many readers can perform reading concurrently.
2. Reading is prohibited while a writer is writing.
3. Only one writer can perform writing at any time.

Conditions 1–3 do not specify which process should be preferred if a reader and a writer process wish to access the shared data at the same time. The following additional condition is imposed if it is important to give a higher priority to readers in order to meet some business goals:

Materiał chroniony prawem autorskim

```
procedure check_b_empty
begin
    if buffer_full = true
    then
        producer_blocked := true;
        block (producer);
end;

procedure post_b_full
begin
    buffer_full := true;
    if consumer_blocked = true
    then
        consumer_blocked := false;
        activate (consumer);
end;
```

Operations of producer

```
procedure check_b_full
begin
    if buffer_full = false
    then
        counsumer_blocked := true;
        block (consumer);
end;

procedure post_b_empty
begin
    buffer_full := false;
    if producer_blocked = true
    then
        producer_blocked := false;
        activate (producer);
end;
```

Operations of consumer

**Fig. 9.13** Indivisible operations for the producers–consumers problem

4. A reader has a non-preemptive priority over writers, i.e., it gets access to the shared data ahead of a waiting writer, but it does not preempt an active writer.

This system is called a *readers preferred readers–writers system*. A *writers preferred readers–writers system* is analogously defined.

Figure 9.14 illustrates an example of a readers–writers system. The readers and writers share a bank account. The reader processes *print statement* and *stat analysis* merely read the data from the bank account, hence they can execute concurrently. *credit* and *debit* modify the balance in the account. Clearly only one of them should be active at any moment and none of the readers should be active when they modify the data.

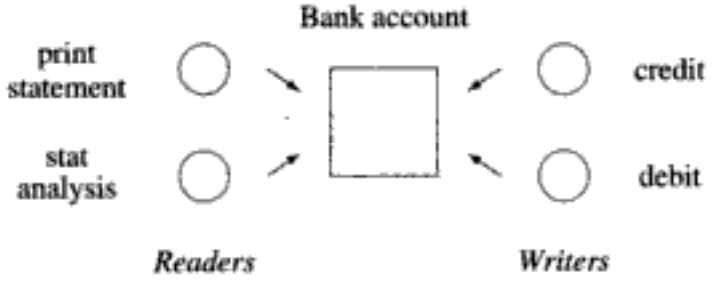


**Fig. 9.14** Readers and writers in a banking system

We determine the synchronization requirements of a readers–writers system by analyzing its correctness conditions. Condition 3 requires that a writer should perform writing in a critical section. When it finishes writing, it should activate one

Materiał chroniony prawem autorskim

waiting writer or activate all waiting readers. This can be achieved using a signaling arrangement. From condition 1, concurrent reading is permitted. We should maintain a count of readers reading concurrently. When the last reader finishes reading, it should activate a waiting writer.

Figure 9.15 contains an outline for a readers–writers system. Writing is performed in a CS. A CS is not used in a reader as that would prevent concurrency between readers. A signaling arrangement is used to handle blocking and activation of readers and writers. It is complicated by the fact that a writer may have to signal many readers. So we do not specify its details in the outline; we shall discuss it in Section 9.8.3. The outline of Figure 9.15 does not satisfy the bounded wait condition for both readers and writers, however it provides maximum concurrency.

```
Parbegin
    repeat                                  repeat
        If a writer is writing                  If reader(s) are reading, or a
        then                                        writer is writing then
            { wait };                               { wait };
        { read }                                [ { write } ];
        If writer(s) waiting                    If reader(s) or writer(s) waiting
        then                                    then
            activate a writer if no                 activate all readers
            other readers reading                   or activate one writer
    forever;                                forever;
Parend
            Reader(s)                               Writer(s)
```

**Fig. 9.15** Solution outline for readers-writers without writer priority

### 9.5.3 Dining Philosophers

Five philosophers sit around a table pondering philosophical issues. A plate of spaghetti is kept in front of each philosopher, and a fork is placed between each pair of philosophers (see Figure 9.16). To eat, a philosopher must pick up the two forks placed between him and his immediate neighbors on either side, one at a time. The problem is to design processes to represent the philosophers such that each philosopher can eat when hungry and none dies of hunger.

The correctness condition in the dining philosophers system is that a hungry philosopher should not face indefinite waits when he decides to eat. The challenge is to design a solution that does not suffer from either *deadlocks*, where processes become blocked waiting for each other, or *livelocks*, where processes are not blocked but defer to each other indefinitely. Consider the outline of a philosopher process $P_i$ shown in Figure 9.17, where details of process synchronization have been omitted. This solution is prone to deadlock, because if all philosophers simultaneously lift their left forks, none will be able to lift the right fork! It also contains race con-

Materiał chroniony prawem autorskim

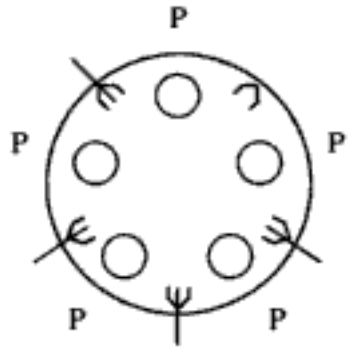


**Fig. 9.16** Dining philosophers

ditions because neighbors might fight over a shared fork. We can avoid deadlocks by modifying the philosopher process so that if the right fork is not available, the philosopher would defer to his left neighbor by putting down the left fork and repeating the attempt to take the forks sometime later. However, this approach suffers from livelocks because the same situation may recur.

```
repeat
    if left fork is not available
    then
        block (Pi);
    lift left fork;
    if right fork is not available
    then
        block (Pi);
    lift right fork;
    { eat }
    if left neighbor is waiting for his right fork
    then
        activate (left neighbor);
    if right neighbor is waiting for his left fork
    then
        activate (right neighbor);
    { think }
forever
```

**Fig. 9.17** Outline of a philosopher process $P_i$

An improved outline for the dining philosophers problem is given in Figure 9.18. A philosopher checks availability of forks in a CS and also picks up the forks in the CS. Hence race conditions cannot arise. This arrangement ensures that at least some philosopher(s) can eat at any time and deadlocks cannot arise. A philosopher who

Materiał chroniony prawem autorskim

cannot get both forks at the same time blocks himself. However, it does not avoid busy waits because the philosopher gets activated when any of his neighbors puts down a shared fork, hence he has to check for availability of forks once again. This is the purpose of the **while** loop. Some innovative solutions to the dining philosophers problem prevent deadlocks without busy waits (see Problem 20 in Exercise 9). Deadlock prevention is discussed in detail in Chapter 11.

```
repeat
    successful := false;
    while (not successful)
        if both forks are available then
            lift the forks one at a time;
                successful := true;
        if successful = false
        then
            block (Pi);
    { eat }
    put down both forks;
    if left neighbor is waiting for his right fork
    then
        activate (left neighbor);
    if right neighbor is waiting for his left fork
    then
        activate (right neighbor);
    { think }
forever
```

**Fig. 9.18** An improved outline of a philospher process

## 9.6 STRUCTURE OF CONCURRENT SYSTEMS

A concurrent system is a system containing concurrent processes. It consists of three key components:

- Shared data
- Operations on shared data
- Processes.

Shared data can be of two kinds—data used and manipulated by processes and data defined and used for synchronization between processes. An operation is a convenient unit of code, typically a procedure in a programming language, which manipulates shared data.

A *synchronization operation* is an operation on synchronization data. Semantics of synchronization operations determine the ease, logical complexity and reliability of a concurrent system implementation. In the following Sections we introduce programming language features for synchronization and discuss semantics of the syn-

Materiał chroniony prawem autorskim

chronization operations provided by them. We also illustrate their use in concurrent systems with the help of snapshots of the system taken at different times.

**Snapshot of a concurrent system** A *snapshot* of a concurrent system is a view of the system at a specific time. It shows relationships between shared data, operations and processes at that moment. We use the pictorial conventions shown in Figure 9.19 to depict a snapshot. A process is shown as a circle. A circle with a cross in it indicates a blocked process. A data item is represented by a rectangular box. The value of the data item, if known, is shown inside the box. An oval shape enclosing a data item indicates that the data item is shared.

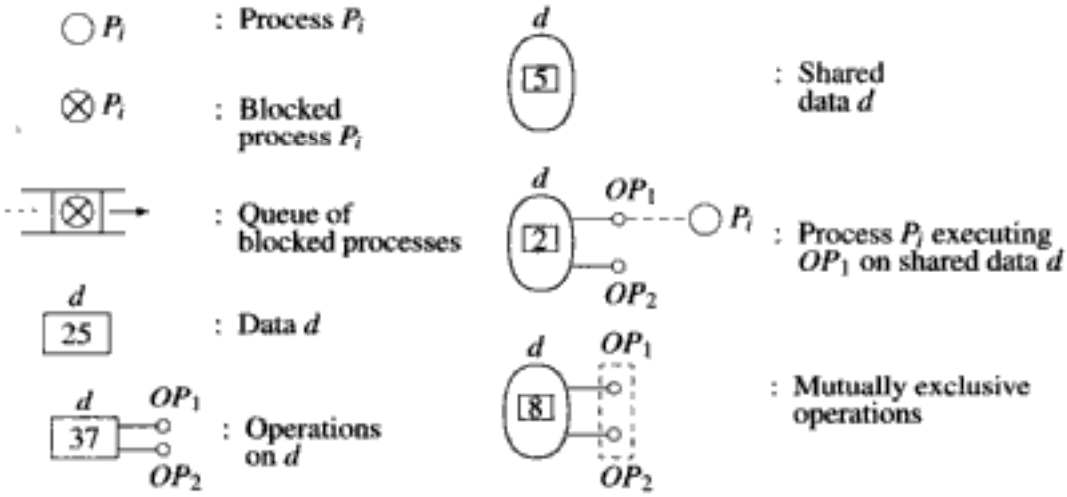


**Fig. 9.19** Pictorial conventions for snapshots of concurrent systems

Operations on data are shown as connectors or sockets joined to the data. An oval shape enclosing a data item indicates that the data item is shared. A dashed line connects a process and an operation on data if the process is currently engaged in executing the operation. We have been using a dashed rectangular box to enclose code executed as a critical section. We extend this convention to operations on data. Hence mutually exclusive operations on data are enclosed in a dashed rectangular box. A queue of blocked processes is associated with the dashed box to show the processes waiting to perform one of the operations.

The execution of a concurrent system is represented by a series of snapshots.

**Example 9.2** Consider the system of Figure 9.3, where process $P_i$ performs action $a_i$ only after process $P_j$ performs action $a_j$. We assume that operations $a_i$ and $a_j$ operate on some shared data items $X$ and $Y$, respectively. Let the system be implemented using the operations *check_aj* and *post_aj* of Figure 9.6. This system comprises of the following components:

| | | |
|---|---|---|
| Shared data | : | Boolean variables *operation_aj_performed* and *pi_blocked*, both initialized to *false*, and data items $X$ and $Y$. |
| Operations on application data | : | Operations $a_i$ and $a_j$. |

Materiał chroniony prawem autorskim

when synchronization involves more than two processes.

We show an algorithm for CS implementation as a pseudo-code with the following features:

1. The **Parbegin–Parend** control structure is used to enclose code that is to be executed in parallel. This control structure has the syntax **Parbegin** *<list of statements>* **Parend**. If *<list of statements>* contains $n$ statements, execution of the **Parbegin–Parend** control structure spawns $n$ processes, each process consisting of the execution of one statement in *<list of statements>*. For example, **Parbegin** $S_1, S_2, S_3, S_4$ **Parend** initiates four processes that execute $S_1, S_2, S_3$ and $S_4$, respectively.

   The statement grouping facilities of a language such as **begin–end**, can be used if a process is to consist of a block of code instead of a single statement. For visual convenience, we depict concurrent processes created in a **Parbegin–Parend** control structure as follows:

   **Parbegin**

   $$\begin{array}{cccc} S_{11} & S_{21} & \cdots & S_{n1} \\ \vdots & \vdots & & \vdots \\ S_{1m} & S_{2m} & \cdots & S_{nm} \end{array}$$

   **Parend**

   $$\underline{\textit{process } P_1} \quad \underline{\textit{process } P_2} \qquad \underline{\textit{process } P_n}$$

   where statements $S_{11} \cdots S_{1m}$ form the code of process $P_1$, etc.
2. Declarations of shared variables are placed before a **Parbegin**.
3. Declarations of local variables are placed at the start of a process.
4. Comments are enclosed within braces '{ }'.
5. Indentation is used to show nesting of control structures.

We begin by discussing CS implementations for use by two processes. Later we extend some of these schemes for use by $n$ processes.

### 9.7.1 Two Process Algorithms

**Algorithm 9.1** *First attempt*

```
var    turn : 1 .. 2;
begin
      turn := 1;
Parbegin
   repeat                                repeat
      while turn = 2                        while turn = 1
         do { nothing };                       do { nothing };
      { Critical Section }                  { Critical Section }
```

Materiał chroniony prawem autorskim

```
        turn := 2;                          turn := 1;
        { Remainder of                      { Remainder of
          the cycle }                         the cycle }
    forever;                            forever;
Parend
end.
```

*Process $P_1$* *Process $P_2$*

Variable *turn* is a shared variable. The notation 1 .. 2 in its declaration indicates that it takes values in the range 1–2, i.e., its value is either 1 or 2. It is initialized to 1 before processes $P_1$ and $P_2$ are created. Each process contains a CS for some shared data $d_s$. The shared variable *turn* is used to indicate which process can enter its CS next. Let process $P_1$ wish to enter its CS. If *turn* = 1, $P_1$ can enter straightaway. After exiting its CS, it sets *turn* to 2 so that $P_2$ can enter its CS. If $P_1$ finds *turn* = 2 when it wishes to enter its CS, it waits in the **while** loop until $P_2$ exits from its CS and executes the assignment *turn* := 1. Thus the correctness condition is satisfied. However, processes may encounter a busy wait before gaining entry to the CS.

Use of shared variable *turn* leads to a problem. Let process $P_1$ be in its CS and process $P_2$ be in the remainder of the cycle. If $P_1$ exits from its CS, finishes the remainder of its cycle and wishes to enter its CS once again, it would encounter a busy wait until after $P_2$ uses its CS. This situation does not violate the bounded wait condition, since $P_1$ has to wait for $P_2$ to go through its CS exactly once. However, the progress condition is violated since $P_1$ is currently the only process interested in using its CS, but it is unable to do so. Algorithm 9.2 tries to eliminate this problem.

**Algorithm 9.2** *Second attempt*

```
var     c1, c2 : 0 .. 1;
begin
        c1 := 1;
        c2 := 1;
Parbegin
    repeat                              repeat
        while c2 = 0                        while c1 = 0
            do { nothing };                     do { nothing };
        c1 := 0;                            c2 := 0;
        { Critical Section }                { Critical Section }
        c1 := 1;                            c2 := 1;
        { Remainder of                      { Remainder of
          the cycle }                         the cycle }
    forever;                            forever;
Parend
end.
```

*Process $P_1$* *Process $P_2$*

Materiał chroniony prawem autorskim

Variable *turn* of Algorithm 9.1 has been replaced by two shared variables $c_1$ and $c_2$. These variables can be looked upon as status flags for processes $P_1$ and $P_2$, respectively. $P_1$ sets $c_1$ to 0 while entering its CS, and sets it back to 1 after exiting from its CS. Thus $c_1 = 0$ indicates that $P_1$ is in its CS and $c_1 = 1$ indicates that it is not in CS. Process $P_2$ checks the value of $c_1$ to decide whether it can enter its CS. This check eliminates the progress violation of Algorithm 9.1 because processes are not forced to take turns using their CSs.

Algorithm 9.2 violates the mutual exclusion condition when both processes try to enter their CSs at the same time. Both $c_1$ and $c_2$ will be 1 (since none of the processes is in its CS), hence both processes will enter their CSs. To avoid this problem, the statements **while** $c_2 = 0$ **do** { *nothing* }; and $c_1 := 0$; in process $P_1$ could be interchanged. (Similarly, in process $P_2$ **while** $c_1 = 0$ **do** { *nothing* }; and $c_2 := 0$; could be interchanged.) This way $c_1$ will be set to zero before $P_1$ checks the value of $c_2$, hence both processes cannot be in their CSs at the same time. However, if both processes try to enter their CSs at the same time, both $c_1$ and $c_2$ will be 0 hence both processes will wait for each other indefinitely. This is a deadlock situation.

Both, the correctness violation and the deadlock possibility, can be eliminated if a process defers to the other process when it finds that the other process also wishes to enter its CS. This can be achieved as follows: if $P_1$ finds that $P_2$ is also trying to enter its CS, it can set $c_1$ to 0. This will permit $P_2$ to enter its CS. $P_1$ can wait for some time and try to enter its CS after turning $c_1$ to 1. Similarly, $P_2$ can set $c_2$ to 0 if it finds that $P_1$ is also trying to enter its CS. However, this approach may lead to a situation in which both processes defer to each other indefinitely. This situation is called a *livelock*.

**Dekker's algorithm** Dekker's algorithm combines the useful features of Algorithms 9.1–9.2 to avoid a livelock situation. A variable named *turn* is used to avoid livelocks. If both processes try to enter their CSs, *turn* indicates which process should be allowed to enter. Many other features of Dekker's algorithm are analogous to those used in the previous algorithms.

**Algorithm 9.3** *Dekker's Algorithm*

```
var     turn : 1 .. 2;
        c1, c2 : 0 .. 1;
begin
        c1 := 1;
        c2 := 1;
      turn := 1;
Parbegin
   repeat                                  repeat
      c1 := 0;                                c2 := 0;
      while c2 = 0 do                         while c1 = 0 do
         if turn = 2 then                        if turn = 1 then
         begin                                   begin
```

Materiał chroniony prawem autorskim

```
            c1 := 1;                          c2 := 1;
            while turn = 2                    while turn = 1
               do { nothing };                   do { nothing };
            c1 := 0;                          c2 := 0;
         end;                              end;
      { Critical Section }              { Critical Section }
      turn := 2;                        turn := 1;
      c1 := 1;                          c2 := 1;
      { Remainder of                    { Remainder of
        the cycle }                       the cycle }
   forever;                          forever;
Parend
end.
```

*Process $P_1$* *Process $P_2$*

Variables $c_1$ and $c_2$ are used as status flags of the processes. The statement **while** $c_2 = 0$ **do** in $P_1$ checks whether it is safe for $P_1$ to enter its CS. To avoid the correctness problem of Algorithm 9.2, the statement $c_1 := 0$ in $P_1$ precedes it. If $c_2 = 1$ when $P_1$ wishes to enter a CS, $P_1$ skips the **while** loop and enters its CS straightaway. If both processes try to enter their CSs at the same time, value of *turn* is used to force one of them to defer to the other. For example, if $P_1$ finds $c_2 = 0$, it defers to $P_2$ only if *turn* = 2, else it simply waits for $c_2$ to become 1 before entering its CS. Process $P_2$, which is also trying to enter its CS at the same time, is forced to defer to $P_1$ only if *turn* = 1. In this manner the algorithm satisfies mutual exclusion and also avoids deadlock and livelock conditions.

**Peterson's algorithm** Peterson's algorithm is simpler than Dekker's algorithm. It uses a boolean array *flag*, which contains one flag for each process. These flags are equivalent to status variables $c_1, c_2$ of earlier algorithms. A process sets its flag to *true* when it wishes to enter a CS and sets it back to *false* when it exits from the CS. Processes are assumed to have the ids $P_0$ and $P_1$. A process id is used as a subscript to access the status flag of a process in the array *flag*. Variable *turn* is for avoiding livelocks, however it is used differently than in Dekker's algorithm.

**Algorithm 9.4** *Peterson's Algorithm*

```
var     flag : array [0 .. 1] of boolean;
        turn : 0 .. 1;
begin
     flag[0] := false;
     flag[1] := false;

Parbegin
   repeat                                repeat
      flag[0] := true;                      flag[1] := true;
      turn := 1;                            turn := 0;
```

Materiał chroniony prawem autorskim

by Lamport.

**Algorithm 9.5** *An n Process Algorithm*

```
const   n = ...;
var     flag : array [0..n   1] of (idle, want-in, in-CS);
        turn : 0 .. n   1;
begin
        for j := 0 to n   1 do
            flag[j] := idle;
Parbegin
    process Pi :
        repeat
            repeat
                flag[i] := want-in;
                j := turn;
                while j ≠ i
                    do if flag[j] ≠ idle
                        then j := turn { Loop here! }
                        else j := j + 1 mod n;
                flag[i] := in-CS;
                j := 0;
                while (j < n) and (j = i or flag[j] ≠ in-CS)
                    do j := j + 1;
            until (j ≥ n) and (turn = i or flag[turn] = idle);
            turn := i;
            { Critical Section }
            j := turn +1 mod n;
            while (flag[j] = idle) do j := j + 1 mod n;
            turn := j;
            flag[i] := idle;
            { Remainder of the cycle }
        forever
    process Pk : ...
Parend
end.
```

The variable *turn* is still used to indicate which process may enter its CS next. Each process has a 3-way status flag which takes the values *idle*, *want_in* and *in_CS*. The flag has the value *idle* when a process is in the remainder of its cycle. A process turns its flag to *want-in* whenever it wishes to enter a CS. It now makes a few checks to decide whether it may change the flag to *in_CS*. It checks the flags of other processes in an order that we call the modulo order. The modulo order is $P_{turn}$, $P_{turn+1}$, $\ldots P_{n\ 1}$, $P_0$, $P_1$, $\ldots P_{turn\ 1}$. In the first **while** loop, the process checks whether any

Materiał chroniony prawem autorskim

process ahead of it in the modulo order wishes to use its CS. If not, it turns its flag to *in-CS*.

Since many processes may make this check concurrently, more than one process may simultaneously reach the same conclusion. Hence another check is made to ensure correctness. The second **while** loop checks whether any other process has turned its flag to *in-CS*. If so, the process changes its flag back to *want-in* and repeats all the checks. All other processes, which had changed their flags to *in-CS* also change their flags back to *want-in* and repeat the checks. These processes will not tie for entry to a CS again because they have all turned their flags to *want-in*, hence only one of them can get past the first **while** loop. This feature avoids the livelock condition. The process earlier in the modulo order from $P_{turn}$ will get in and enter its CS ahead of other processes.

This solution contains a certain form of unfairness since processes do not enter their CSs in the order in which they request entry to a CS. This unfairness is eliminated in the Bakery algorithm by Lamport [1974].

**Bakery algorithm** When a process wishes to enter a CS, it chooses a token such that the number contained in the token is larger than any number issued earlier. *choosing* is an array of boolean flags. *choosing*[*i*] is used to indicate whether process $P_i$ is currently engaged in choosing a token. *number*[*i*] contains the number in the token chosen by process $P_i$. *number*[*i*] = 0 if $P_i$ has not chosen a token since the last time it entered the CS. Processes enter CS in the order of numbers in their tokens.

**Algorithm 9.6** *Bakery Algorithm*

```
const    n = ... ;
var   choosing : array [0 .. n   1] of boolean;
       number : array [0 .. n   1] of integer;
begin
      for j := 0 to n   1 do
         choosing[j] := false;
         number[j] := 0;
Parbegin
   process P_i :
      repeat
         choosing[i] := true;
         number[i] := max (number[0], ..., number[n   1])+1;
         choosing[i] := false;
         for j := 0 to n   1 do
         begin
            while choosing[j] do { nothing };
            while number[j] ≠ 0 and (number[j],j) < (number[i],i)
               do { nothing };
         end;
```

Materiał chroniony prawem autorskim

#### 9.8.1.2 Bounded Concurrency

We use the term *bounded concurrency* for the situation in which upto $c$ processes can concurrently perform an operation $op_i$, where $c$ is a constant $\geq 1$. Bounded concurrency is implemented by initializing a semaphore *sem_c* to $c$. Every process wishing to perform $op_i$ performs a *wait*(*sem_c*) before performing $op_i$, and a *signal*(*sem_c*) after performing it. From the semantics of *wait* and *signal* operations, it is clear that upto $c$ processes can concurrently perform $op_i$. Semaphores used to implement bounded concurrency are called *counting semaphores*.

Figure 9.24 illustrates how a set of concurrent processes share five printers.

```
var printers : semaphore := 5;
   Parbegin
      repeat                                   repeat
         wait(printers);                          wait(printers);
         { Use a printer }            ...         { Use a printer }
         signal(printers);                        signal(printers);
         { Remainder of the cycle }               { Remainder of the cycle }
      forever;                                 forever;
   Parend
   end.
```

Process $p_1$ ... Process $p_n$

**Fig. 9.24** Bounded concurrency using semaphores

#### 9.8.1.3 Signaling Between Processes

Consider the synchronization requirements of processes $P_i$ and $P_j$ shown in Figure 9.4. A semaphores can be used to achieve this synchronization as shown in Figure 9.25. Here process $P_i$ performs a *wait*(*sync*) before executing action $a_i$ and $P_j$ performs a *signal*(*sync*) after executing action $a_j$. Semaphore *sync* is initialized to 0 hence $P_i$ will get blocked on *wait*(*sync*) if $P_j$ has not already performed a *signal*(*sync*). It would proceed to perform action $a_i$ only after process $P_j$ performs a *signal*. Unlike the solution of Figure 9.4, race conditions cannot arise because the *wait* and *signal* operations are indivisible. The signaling arrangement can be used repetitively; it does not require resetting as in the signaling arrangement of Figure 9.12.

### 9.8.2 Producers–Consumers Using Semaphores

As discussed in Section 9.5.1, the producers–consumers problem is a signaling problem. After consuming a record from a buffer, a consumer signals to a producer that is waiting to produce in the same buffer. Analogously, a producer signals to a waiting consumer. Hence we should implement producers–consumers using the signaling arrangement shown in Figure 9.25.

For simplicity, we first discuss the solution for the single buffer case shown in Figure 9.26. The buffer pool is represented by an array of buffers with a single

Materiał chroniony prawem autorskim

```
var sync : semaphore := 0;
Parbegin
        ...                              ...
        wait(sync);                      {Perform action a_j}
        {Perform action a_i}             signal(sync);
Parend
end.
```

*Process $P_i$* *Process $P_j$*

**Fig. 9.25** Signaling using semaphores

element in it. Two semaphores *full* and *empty* are declared. They are used to indicate the number of full and empty buffers, respectively. *full* is initialized to 0 and *empty* is initialized to 1. A producer performs a *wait*(*empty*) before starting the produce action and a consumer performs a *wait*(*full*) before a consume action.

```
type    item = ...;
var
          full : Semaphore := 0; { Initializations }
        empty : Semaphore := 1;
        buffer : array [0] of item;
begin
Parbegin
   repeat                                  repeat
      wait(empty);                            wait(full);
      buffer[0] := ...;                       x := buffer[0];
          { i.e. produce }                        { i.e. consume }
      signal(full);                           signal(empty);
      { Remainder of the cycle }              { Remainder of the cycle }
   forever;                                forever;
Parend
end.
```

*Producer* *Consumer*

**Fig. 9.26** Producers–consumers with a single buffer

Initially consumer(s) would get blocked on a consume and only one producer would get past the *wait*(*empty*) operation. After completing the produce action it performs *signal*(*full*). This enables one consumer to enter, either immediately or in future. When the consumer finishes a consume, it performs a *signal*(*empty*) operation that enables a producer to enter and perform a produce action. This solution avoids busy waits since semaphores are used to check for empty or full buffers, hence a process gets blocked if it cannot find a full or empty buffer. The total concurrency is 1, sometimes a producer executes and sometimes a consumer executes. Example 9.3

Materiał chroniony prawem autorskim

```
const          n = ...;
type        item = ...;
var
          buffer : array [0..n  1] of item;
            full : Semaphore := 0; { Initializations }
           empty : Semaphore := n;
prod_ptr, cons_ptr : integer;
begin
        prod_ptr := 0;
        cons_ptr := 0;
Parbegin
  repeat                                    repeat
    wait(empty);                              wait(full);
    buffer[prod_ptr] := ...;                  x := buffer[cons_ptr];
      { i.e. produce }                          { i.e. consume }
    prod_ptr := prod_ptr+1 mod n;             cons_ptr := cons_ptr+1 mod n;
    signal(full);                             signal(empty);
  forever;                                  forever;
Parend
end.
```

*Producer* *Consumer*

**Fig. 9.28** Bounded buffers using semaphores

### 9.8.3 Readers–Writers Using Semaphores

Key features of the readers–writers problem are evident from the outline of Figure 9.15. These are as follows:

- Any number of readers can read concurrently.
- Readers and writers must wait if a writer is writing. When the writer exits, either all waiting readers should be activated or one waiting writer should be activated.
- A writer must wait if readers are reading. It must be activated when the last reader exits.

We implement these features by keeping track of the number of readers and writers that wish to read or write at any moment. The following counters are introduced for this purpose:

| | |
|---|---|
| *runread* | count of readers currently reading |
| *totread* | count of readers waiting to read or currently reading |
| *runwrite* | count of writers currently writing |
| *totwrite* | count of writers waiting to write or currently writing |

Values of these counters are incremented and decremented at appropriate times by the processes. For example, a reader process increments *totread* when it decides

Materiał chroniony prawem autorskim

to read and it increments *runread* when it actually starts reading. It decrements both *totread* and *runread* when it finishes reading. Thus *totread* − *runread* indicates how many readers are waiting to begin reading. Similarly *totwrite* − *runwrite* indicates the number of writers waiting to begin writing. Values of these counters are used as follows: A reader is allowed to begin reading when *runwrite* = 0 and a writer is allowed to begin writing when *runread* = 0 and *runwrite* = 0. The value of *totread* is used to activate all waiting readers when a writer finishes writing (note that *runread* = 0 when a writer is writing).

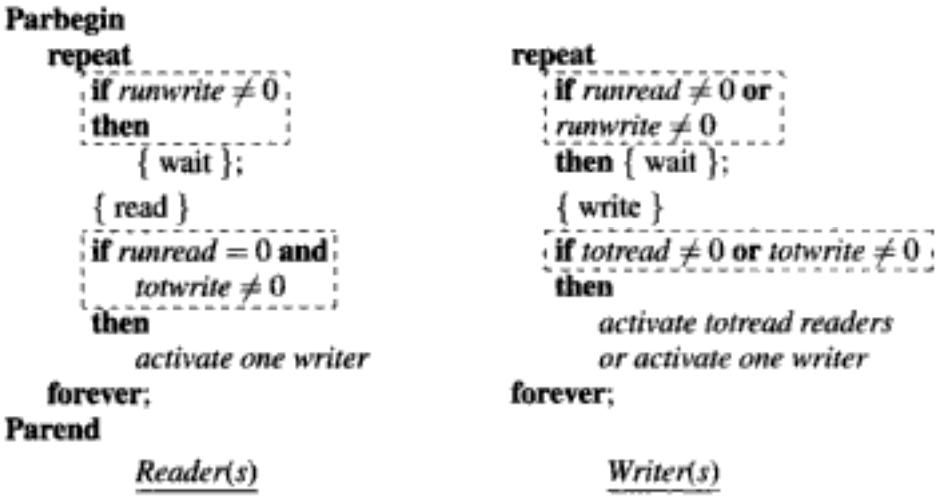


**Fig. 9.29** Refined solution outline for readers-writers without writer priority

The outline of Figure 9.15 is refined to obtain the solution shown in Figure 9.29. Values of the counters are examined inside critical sections. It is assumed that their values are incremented or decremented in other critical sections at appropriate places. This solution does not use explicit critical sections for readers or writers. Instead they are blocked until they can be allowed to start reading or writing.

Blocking of readers and writers resembles the blocking of producers and consumers in the producers–consumers problem, so it is best handled using semaphores for signaling. We introduce two semaphores named *reading* and *writing*. A reader process performs *wait*(*reading*) before starting to read. This action blocks the reader process if conditions permitting it to read are not currently satisfied, else the reader can get past the *wait* operation and start reading. Similarly, a writer process performs a *wait*(*writing*) before writing.

Who should perform *signal* operations to activate the reader and writer processes blocked on their *wait* operations? Obviously conditions governing the start of reading or writing were not satisfied when the reader or writer processes were blocked. These conditions change when any of the counter values change, i.e., when a reader finishes reading or a writer finishes writing. Hence processes must perform appropriate *signal* operations after completing a read or a write operation.

Materiał chroniony prawem autorskim

```
var
    totread, runread, totwrite, runwrite : integer;
                    reading, writing : semaphore := 0;
                                mutex : semaphore := 1;
begin
        totread  := 0;
        runread  := 0;
        totwrite := 0;
        runwrite := 0;
Parbegin
  repeat                                    repeat
    wait(mutex);                              wait(mutex);
    totread := totread+1;                     totwrite := totwrite+1;
    if runwrite = 0 then                      if runread = 0 then
      runread := runread+1;                     runwrite := 1;
      signal(reading);                          signal(writing);
    signal(mutex);                            signal(mutex);
    wait(reading);                            wait(writing);
    { Read }                                  { Write }
    wait(mutex);                              wait(mutex);
    runread := runread  1;                    runwrite := runwrite  1;
    totread := totread  1;                    totwrite := totwrite  1;
    if runread = 0 and                        while (runread<totread) do
      totwrite > runwrite                     begin
      then                                      runread := runread+1;
        runwrite := 1;                          signal(reading);
        signal(writing);                      end;
    signal(mutex);                            if runread = 0 and
  forever;                                    totwrite > runwrite then
                                                runwrite := 1;
                                                signal(writing);
                                              signal(mutex);
                                            forever;
Parend
end.
          Reader(s)                                 Writer(s)
```

**Fig. 9.30** Readers and writers using semaphores

Materiał chroniony prawem autorskim

The *wait* operation has a very low failure rate in most systems using semaphores, i.e., processes performing *wait* operations are seldom blocked. This characteristic is exploited in some methods of implementing semaphores to reduce the overhead. In the following, we describe three methods of implementing semaphores and examine their overhead implications. Recall that we use the term process as a generic term for both processes and threads.

**Kernel-level implementation** The kernel implements the *wait* and *signal* procedures of Figure 9.31. All processes in a system can share a kernel-level semaphore. However, every *wait* and *signal* operation results in a system call; it leads to high overhead of using semaphores. In a uniprocessor OS with a non-interruptible kernel, it would not be necessary to use a lock variable to eliminate race conditions, so the overhead of busy waits in the *Close_lock* operation can be eliminated.

**User-level implementation** The *wait* and *signal* operations are coded as library procedures, which are linked with an application program so that processes of the application can share user-level semaphores. The *block_me* and *activate* calls are actually calls on library procedures, which handle blocking and activation of processes themselves as far as possible and make system calls only when they need assistance from the kernel. This implementation method would suit user-level threads because the thread library would already provide for blocking, activation and scheduling of threads. The thread library would make a *block_me* system call only when all threads of a process are blocked.

**Hybrid implementation** The *wait* and *signal* operations are again coded as library procedures, and processes of an application can share the hybrid semaphores. *block_me* and *activate* are system calls provided by the kernel and the *wait* and *signal* operations make these calls only when processes have to be blocked and activated. In principle, it is possible to implement hybrid semaphores even if such kernel support is not available. For example, the system calls *block me* and *activate* can be replaced by system calls for receiving and sending messages or signals. However, message passing is subject to availability of buffer space, and signals can be ignored by processes, so it is best to use a separate *block_me* system call for semaphore implementation.

## 9.9 CONDITIONAL CRITICAL REGIONS

The conditional critical region (CCR) is a control structure in a higher level programming language. It provides two features for process synchronization—it provides mutual exclusion over accesses to shared data, and it permits a process executing CCR to block itself until a specified boolean condition becomes *true*.

Figure 9.32 shows a concurrent program using the CCR construct **region** *x* **do** .. Variable *x* is declared with the attribute **shared**. It can be used in any process of the program. However, it must be used only within a CCR, i.e., within a **region** *x* **do**

Materiał chroniony prawem autorskim

```
const    n =  ... ;
type  item =  ... ;
var
    buffer_pool :  Shared record
                       buffer : array [0..n   1] of item;
                         full : integer := 0;
                    prod_ptr : integer := 0;
                    cons_ptr : integer := 0;
                   end
begin
Parbegin
    var produced_info : item;
    repeat
        { Produce in produced_info }
        region buffer_pool do
        begin
            await (full < n);
            buffer[prod_ptr]
                    := produced_info;
            prod_ptr :=
                    prod_ptr+1 mod n;
            full := full+1;
        end;
        { Remainder of the cycle }
    forever;
Parend
end.
```

*Producer*

```
var for_consumption : item;
repeat
    region buffer_pool do
    begin
        await (full > 0);
        for_consumption :=
                buffer[cons_ptr];
        cons_ptr :=
                cons_ptr+1 mod n;
        full := full   1;
    end;
    { Consume from for_consumption }
    { Remainder of the cycle }
forever;
```

*Consumer*

**Fig. 9.33** Bounded buffer using conditional critical region

any changes to the code of the producer and consumer processes.

Figure 9.34 contains a solution to the readers–writers problem using conditional critical regions. *read_write* is a shared variable containing the counters *runread* and *runwrite*. A reader should read only when no writer is writing. Hence a reader process performs an **await** (*runwrite* = 0) before starting to read. A writer performs a write inside **region** *read_write* **do**, hence it is adequate to use an **await** (*runread* = 0) before starting to write. When a writer completes, a reader blocked on *runwrite* = 0 in an **await** statement gets activated and exits the region. The condition *runwrite* = 0 still holds, so other readers blocked on this condition also get activated. Thus there is no need for the counter *totread* of Figure 9.30. The counter *totwrite* is not needed due to a similar reason.

Note that the count *runwrite* of this solution is redundant. Its purpose is to block a reader from entering its CS if a writer is writing. However, since writing is performed inside a CS, no reader can enter its CS while a writer is writing! Hence *runwrite*

Materiał chroniony prawem autorskim

```
type item = ... ;
var
      read_write : shared record
                     runread : integer := 0;
                     runwrite : integer := 0;
                   end
begin
Parbegin
    repeat                                    repeat
       region read_write do                      region read_write do
       begin                                     begin
          await (runwrite = 0);                     await (runread = 0);
          runread := runread+1;                     runwrite := runwrite+1;
       end;                                      { Write }
       { Read }                                  runwrite := runwrite  1;
       region read_write do;                  end;
          runread := runread  1;              { Remainder of the cycle }
       end;                                  forever;
       { Remainder of the cycle }
    forever;
Parend
end.
            Reader(s)                                 Writer(s)
```

**Fig. 9.34** Readers and Writers without writer priority

is bound to be 0 every time a reader executes the statement **await** (*runwrite* = 0). A more compact and elegant solution is obtained by eliminating *runwrite* and all statements incrementing or testing its value.

### 9.9.1 Implementation of CCR

Implementation of the **await** statement is straightforward. The code generated by the compiler can evaluate the condition and, if *false*, make a *block me* system call to block the process. How and when should the blocked process be activated? This is best achieved by linking the condition in the **await** statement with an event known to the scheduling component of the kernel. In Figure 9.33, conditions in the **await** statements (viz. *full* $< n$ and *full* $> 0$) involve fields of *buffer_pool*. Each CCR is implemented as a CS on *buffer_pool*, so these conditions can change only when some process executes the CCR. Hence every time a process exits a CCR, conditions in all **await** statements in the CCR on which processes are blocked can be checked. A blocked process whose **await** condition is satisfied can now be activated.

However, activation of a process is not always so simple. Semantics of the CCR construct permit any boolean condition to be used in an **await** statement. If the condition does not involve a field of the CCR control variable, execution of some

Materiał chroniony prawem autorskim

statement outside a CCR may also change the truth value of an **await** condition. For example, conditions involving values returned by system calls like time-of-day can become true even if no process executes a CCR. Such conditions will have to be checked periodically to activate blocked processes. These checks increase the overhead of implementing CCRs.

## 9.10 MONITORS

A *monitor* is a programming language construct that supports both data access synchronization and control synchronization. A monitor type resembles a class in a language like C++ or Java. It has the four aspects summarized in Table 9.5—declaration and initialization of shared data, operations on shared data and synchronization operations. A concurrent program creates monitor objects, i.e., objects of a monitor type, and uses them to perform operations on shared data and implement process synchronization using synchronization operations. We refer to an object of a monitor type as a monitor variable, or simply as a monitor.

**Table 9.5** Aspects of a monitor type

| Aspect | Description |
|---|---|
| Data declaration | Shared data and condition variables are declared here. Copies of this data exist in every object of a monitor type. |
| Data initilization | Data are initialized when a monitor, i.e. an object of a monitor type, is created. |
| Operations on shared data | Operations on shared data are coded as procedures of the monitor type. The monitor ensures that these operations are executed in a mutually exclusive manner. |
| Synchronization operation | Procedures of the monitor type use synchronization operations **wait** and **signal** over *condition variables* to synchronize execution of processes. |

Operations on shared data are coded as procedures of the monitor. To ensure consistency of shared data and absence of race conditions in their manipulation, the monitor ensures that its procedures are executed in a mutually exclusive manner. Calls on procedures of a monitor are serviced in a FIFO manner to satisfy the bounded wait property. This feature is implemented by maintaining a queue of processes that wish to execute monitor procedures. Process synchronization is implemented using the **signal** and **wait** operations on synchronization data called *condition variables*.

**Condition variables** The process synchronization support in monitors resembles the event mechanism described in Section 3.3.6 in many respects. Events are called *conditions* in monitors. A condition variable, which is simply a variable with the attribute **condition**, is associated with each condition in a monitor. Thus,

Materiał chroniony prawem autorskim

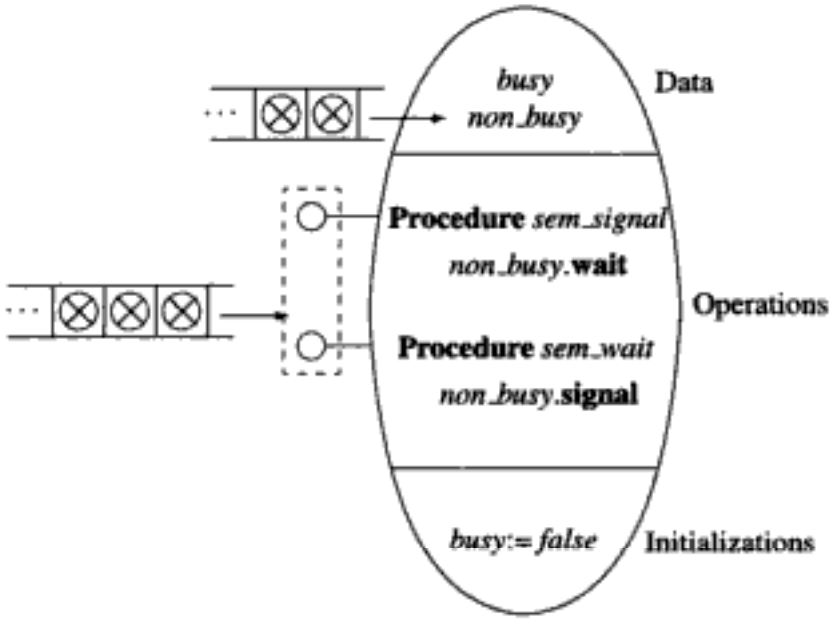


**Fig. 9.36** A monitor implementing binary semaphore

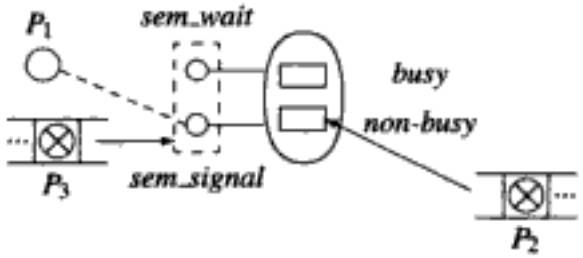


**Fig. 9.37** Snapshot of the system of Example 9.4

Scheduling process $P_2$ would delay the signaling process $P_1$, which seems unfair. Scheduling $P_1$ would imply that $P_2$ is not really activated until $P_1$ leaves the monitor. Hoare (1974) proposed the first alternative. Brinch Hansen (1973) proposed that a **signal** statement should be the last statement of a monitor procedure; this way, the process executing **signal** exits the monitor procedure immediately and the process activated by the **signal** statement can be scheduled.

As mentioned earlier, use of condition variables has two advantages: First, a **signal** has no effect if no process is blocked on the condition variable. This feature simplifies the logic of operations on shared data. Example 9.5 illustrates this fact in the context of solution to the producers–consumers problem. Second, unlike in conditional critical regions, processes are activated by execution of **signal** statements. Hence the implementation of monitors need not repeatedly check whether a blocked process can be activated; this reduces the process synchronization overhead.

Figure 9.38 shows a solution to the producers–consumers problem using monitors.

Materiał chroniony prawem autorskim

```
type Disk_Mon_type : monitor
    const n = ... ; { Size of IO list }
          m = ... ; { Number of user processes }
    type   Q_element = record
                           proc_id : integer;
                           track_no : integer;
                       end;
    var
              IO_list : array [1..n] of Q_element;
            proceed : array [1..m] of condition;
          whose_IO_in_progress, count : integer;
    procedure IO_request (proc_id, track_no : integer);
    begin
        { Enter proc_id and track_no in IO_list }
        count := count + 1;
        if count > 1 then proceed[proc_id].wait;
        whose_IO_in_progress := proc_id;
    end;
    procedure IO_complete (proc_id : integer);
    var id : integer;
    begin
        if whose_IO_in_progress ≠ proc_id then { indicate error }
        { Remove request of this process from IO_list }
        count := count    1;
        if count > 0 then
        begin
            { Select the IO operation to be initiated from IO_list }
            id := id of requesting process;
            proceed[id].signal;
        end;
    end;
    begin { Initializations }
        count := 0;
        whose_IO_in_progress := 0;
end;
```

**Fig. 9.39** A monitor for implementing the disk scheduler

The disk scheduling arrangement is as follows: In the code of a user process, each I/O operation is preceded by a call to the disk monitor procedure *IO_request*. Process id and details of the I/O operation are parameters of this call. *IO_request* enters these parameters in *IO_list*. If an I/O operation is in progress, it blocks the user process until its I/O operation can be initiated. This blocking is performed by issuing a **wait** on a condition variable.

When the disk scheduler decides to schedule the I/O operation, it executes a **signal** statement to activate the user process. The user process now exits the monitor procedure and starts its I/O operation. After the I/O operation, it invokes the monitor

Materiał chroniony prawem autorskim

(2,85), (3,40) and (4,100). Processes $P_1$ $P_4$ are blocked on different elements of *proceed*, which is an array of condition variables. *IO_complete* is called by $P_6$. It selects the I/O operation on track number 85, and obtains the id of the process that made this request from *IO_list*. This id is 2, hence it executes the statement *proceed*[2].**signal**. Process $P_2$ is now activated and proceeds to perform its I/O operation.

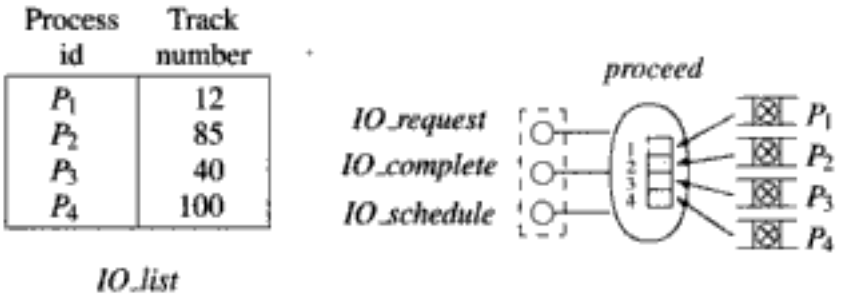


**Fig. 9.42** Snapshot of the disk monitor

This disk monitor makes the simplifying assumption that a process can be allowed to perform its own I/O operations. If this is not the case, the I/O operation should be performed by the disk monitor itself (see Problem 23 of Exercise 9).

### 9.10.2 Monitors in Java

A Java class becomes a monitor type when the attribute `synchronized` is associated with one or more methods in the class. An object of such a class is a monitor. The Java virtual machine ensures mutual exclusion over the `synchronized` methods in a monitor. Each monitor contains a single unnamed condition variable. A thread waits on the condition variable by executing the call `wait()`. The `notify()` call is like the **signal** operation described in Section 9.10. It wakes one of the threads waiting on the condition variable. The Java virtual machine does not implement FIFO behavior for the wait and notify calls. Thus it does not provide the bounded wait property. The `notifyall()` call activates all threads waiting on the condition variable.

Provision of a single condition variable in a monitor has the effect of clubbing together all threads that require synchronization in the monitor. It leads to busy waits in an application if threads need to wait on different conditions. This problem is encountered in a readers–writers system. When a writer is active, all readers wishing to read and all writers wishing to write will have to wait on the same condition variable. If the writer executes a `notify()` call when it finishes writing, either one reader or one writer will be activated. However, concurrent reading is desirable, so it is not enough to activate only one reader. The writer would have to use a `notifyall()` call to activate all waiting threads. If readers are preferred, all writer threads will have to perform `wait()` calls once again. If writers are preferred, all reader threads and some writer threads will have to perform `wait()` calls once again. Thus, a reader or writer thread may be activated many times before it gets an opportunity to perform reading or writing. A producers–consumers system would also suffer from a similar

Materiał chroniony prawem autorskim