The United Nations University

UNU/IIST

International Institute for Software Technology

Lecture Notes on Programming Concurrent Computer Systems

Zhiming Liu

May 2005

UNU/IIST Report No. 325

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macau or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

Lecture Notes on Programming Concurrent Computer Systems

Zhiming Liu

Abstract

Concurrent programming is the activity of constructing a program containing multiple processes that execute in parallel. The aim of the course is to introduce the basic concepts, principles and techniques in programming concurrent computing systems, and to provide practice in solving problems and skill in writing concurrent programs. The course will also provide methods for evaluating systems, algorithms and languages from a broad perspective.

The course provides the students with understanding of the notations of concurrency, nondeterminism, synchronization, deadlock, livelock, safety and liveness. It provides students with the history of the development different language mechanisms for the realization of synchronization and interactions, including synchronization without using synchronization primitives, synchronization by semaphores, conditional regions, monitors, hand-shake communication and remote invocation. It covers communication via shared variables and message passing.

We will take an informal but prices approach in the discussion of of requirements, design and validation of concurrent programs.

Zhiming Liu is a research fellow at UNU/IIST. His research interests include theory of computing systems, including sound methods for specification, verification and refinement of fault-tolerant, real-time and concurrent systems, and formal techniques for OO development. His teaching interests are Communication and Concurrency, Concurrent and Distributed Programming, Software Engineering, Formal specification and Design of Computer Systems. E-mail: Z.Liu@iis.unu.edu.

Copyright © **2005** by UNU/IIST, Zhiming Liu

Contents

1 PREFACE

1.1 Aims & Objectives

The module will focus on basic concepts, principles and techniques in the programming of concurrent and distributed computing systems, and not on specific systems or languages. The students who master the material offered will be prepared not only to write concurrent programs or to read the research literature, but also to evaluate systems, algorithms and languages from a broad perspective.

One cannot learn any programming technique without practising it. The practical classes will give students practice in solving problems, methods in design and skill in writing concurrent programs in a variety of language models exemplifying different programming paradigms.

This module will provide the essential background for further study and research in distributed systems, real-times systems and fault-tolerant systems.

1.2 Syllabus

Break Away from Single Thread Computation: Sequential programming, single processor system, sequential programming as total ordering (single-tread computation), multi-thread computation and its advantages, multi-processor system, concurrent programming as partial ordering, single-processor multi-tasking system, concurrent programming as interleaving.

[3 hours]

Problems in Concurrent Programming: Non-determinism, synchronisation, communication, critical sections, atomic actions.

[3 hours]

Mutual Exclusion and Condition Synchronisation: Communication via shared variables, multiple updating problem, active and passive objects, abstraction of mutual exclusion and condition synchronisation, old-fashioned recipes for synchronisation, notions of busy-waiting, deadlock, livelock, safety, and liveness.

[3 hours]

Semaphores: Motivations and definitions, implementation issues, programming techniques using semaphores (the producers and consumers problem, the readers and writers problem, and the dining philosophers problem), reasoning about semaphore programs.

[4 hours]

Conditional Critical Regions and Monitors: Motivations and definitions, implementation issues, programming techniques using CCRs and Monitors (the producers and consumers problem and the readers and writers problem), reasoning about monitor programs.

[3 hours]

Synchronous Message Passing: Distributed computing, asynchronous vs synchronous, channels for inter-process communication and synchronisation, selective waiting construct for non-determinism, implementation issues, programming techniques using message passing, reasoning about message passing programs.

[3 hours]

Remote Invocation: Motivation and definitions, message passing and synchronisation with remote invocation, client-server paradigm of process interaction, implementation issues, programming techniques using remote invocation.

[3 hours]

Summing Up: summary of the course and industry relevance.

[1 lecture]

1.3 Reading List

A. Concurrent Programming, A. Burns and G.Davies, Addison-Wesley, 1993.

B. Principles of Concurrent Programming, M. Ben-Ari, Prentice-Hall, 1982.

B. Principles of Concurrent and Distributed Programming, M. Ben-Ari, Prentice-Hall, 1990.

B. Concurrent Programming – Principles and practice, G.R. Andrews, The Benjaming/Cummings Publishing Company, Inc.., 1991.

C. Communication and Concurrency, R. Milner, Prentice-Hall, 1989.

C. Communicating Sequential Processes, C.A.R. Hoare, Prentice-Hall, 1985.

2 INTRODUCTION

This chapter also shows how you organize a course and start the first lecture to introduce your course to your students.

2.1 Goals for Chapter 1

- We shall first talk about the Course organisation.
- We shall give a Reading list.
- We shall discuss the Course Assessment.
- We shall discuss the aims and objectives and the materials to be covered in the course.
- Introduce concurrent programming.
- Introduce problems in concurrent programming to motivate the solutions that are provided in later chapters.
- The seat reservation problem – an example.
- We shall discuss some of the basic features of the language that we shall be using for our examples, assignments and exercises throughout the course: Pascal-FC (Functionally Concurrent Pascal).

2.2 Course Organisation

The course is organised as follows: 24 lectures, 10 tutorials, 10 two-hour practical sessions. Tutorials will be based on studying examples and solving problems from the exercises provided in the lecture notes. All course participants should attend these tutorial seminars (see the study

One cannot learn any programming technique without practising it. The practical classes will be based on undertaking programming exercises. All course participants should attend the practical sessions (see the study guide).

For details about the timetable, please see the study guide (I do have a study guide that I always handed out to the students).

2.3 Reading List

- The Main Text:

  A. Burns and G. Davies, *Concurrent Programming*, Addison-Wesley, 1993.

- Recommended Reading:

  M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall International, 1982.

  G.R. Andrews, *Concurrent Programming – Principles and Practice*, The Benjamin/Cummings Publishing Company, Inc., 1991.

  G.R. Andrews and R.A. Olsson, *The SR Programming Language*, Benjaming/Cummings Publishing Company, Inc., 1993.

  J.G.P. Barnes, *Programming in Ada (3rd Edition)*, Addison-Wesley, 1991.

  G. Jones and M. Goldsmith, *Programming in occam 2*, Prentice-Hall, 1988.

  D. Lea, Concurrent Programming in Java, Addison-Wesley, 1997.

  J. Magee and J. Kramer, Concurrency: State Models and Java, Wiley, 1999.

- Background Reading:

  C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

  R. Milner, *Communication and Concurrency*, Prentice-Hall International, 1989.

  A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages*, Addison-Wesley, 1990.

2.4 Course Assessment

- Coursework: 30% of the total credit. Five *assessed worksheets* will be given with explicit deadlines. Late submissions can get marked only when genuine reasons are given and adequate documentary evidence is produced. The final marks on coursework would be forwarded to the resit examination in case one is required (though I sincerely hope this would never happen). There would be no chance to resit on coursework. **For detailed information, please read the study guide**.

- Written Examination: 70% of the total credit. There will be a 3-hour examination in January 1997.

2.5 Overview

Concurrent programming is the activity of constructing a program containing multiple processes that execute in parallel. The aim of the course is to introduce the basic concepts, principles and techniques in programming concurrent computing systems, and to provide practice in solving problems and skill in writing concurrent programs. The course will also provide methods for evaluating systems, algorithms and languages from a broad perspective.

The course will study the key features of concurrent programs, including *concurrency*, *communication* and *synchronization*. It will introduce various language mechanisms including *shared variables*, *semaphores*, *monitors* and *value passing*, to achieve these features.

The course will cover some “classic” concurrent programming problems such as the problems of *mutual exclusion*, *producers and consumers*, *readers and writers*, and the *dining philosophers*.

The Pascal-FC (Functionally Concurrent), which is developed as a *teaching language* for concurrent programming by A. Burns and G. Davies, will be used for practical programming assignments and exercises.

2.6 Concurrent Programming

To talk about the nature and use of concurrent programming, it is better to talk a bit about *sequential programming*, and then to introduce concurrent programming by the way of contrast.

2.6.1 Sequential programming

When we write a program using a sequential programming language such as Pascal, or C language, we always expect the program is to be executed upon a *single processor architecture* (or often called *uniprocessor system*). Otherwise, the use of the programming language will not be successful. For historic reason, this architecture is often referred to the von Neumann architecture. The essence of this architecture (see Figure 1) can be summarized as follows:

- there is a single processing element (CPU), which is connected to Random Access Memory (RAM) and to input/output devices by means of a *bus*;
- both program instructions and data are stored in RAM;
- the processor repeatedly executes the cycle of loading and executing the next command (fetched from RAM) as referenced by the current value of the program counter;
- as there is only one CPU, the system can at most be executing one instruction at any time;
- the process is *deterministic*, i.e. there is only one possible execution sequence and one possible final result (if any) can be produced if you run the program any number of times with the *same* input data; the *thread* of execution in such a computation is the sequence of values of the program counter.

This means that in a sequential programming language, such as Pascal/C, we are giving the system (hardware & software) which runs the program very *strict instructions about the order* in which it may carry out the statements in the program. For example, when we write a sequential program fragment

```
P;
```

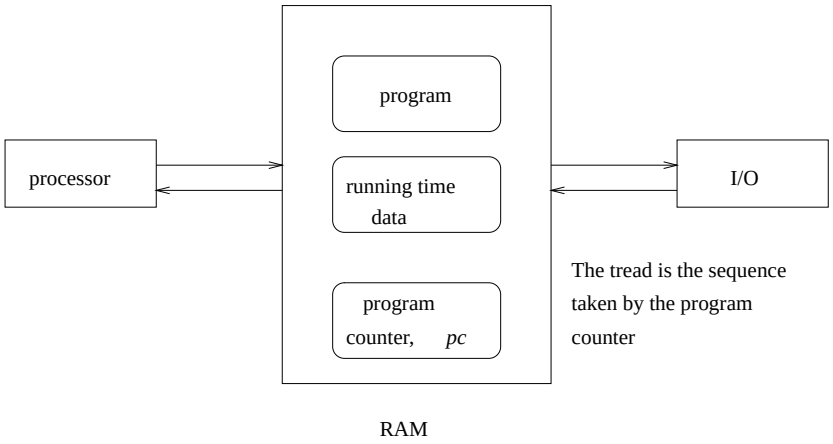


Figure 1: Uniprocessor System

```
Q;
R
```

we actually instruct the system that the execution of P *must* precede the execution of Q, and that, in turn, the execution of Q *must* precede the execution of R. At this level abstraction, what we are saying is that the execution of the fragment can be viewed as the *totally ordered* sequence

```
P, Q, R
```

More concretely let the three statements be

```
x := 1;   (*P*)
y := x+1; (*Q*)
x := y+2  (*R*)
```

It is clear that the final values of the variables, x and y, depend on the order in which the statements are executed. Indeed, the correctness of the algorithm which is written in a sequential languages relies on the assumption that the textual order of the statement is the oder in which they are executed. There is, in effect, a contract between the implementor and the user of the language that the language implementation will behave in that way.

In general, each statement would be compiled into several *machine-code* instructions, e.g.

1. P is treated as a single machine instruction p: store 1 at the address of x.
2. Q is broken into 3 machine operations q1,q2,q3 as follows:
   - q1: load the value of x into a CPU register,
   - q2: increment the value in this register by 1,
   - q3: store the value in this register at the address of y.
3. R is broken into 3 machine operations r1,r2,r3 as follows:
   - r1: load the value of y into a CPU register,
   - r2: increment the value in this register by 2,
   - r3: store the value in this register at the address of x.

Now, what does it mean when we say *the execution of* P *must precede the execution of* Q? It implies that the execution of P must begin before the execution of Q begins, but more precisely, it also means that *the execution of* Q *can only begin after the execution of* P *finishes*. Another way to say this is that there should be no overlap in the execution of the component operations of P and Q. Thus the totally ordered execution sequence at the program level

```
P, Q, R
```

implies the the following totally ordered execution sequence at the system level

```
p, q1,q2,q3, r1,r2,r3
```

Do control structures destroy the total ordering? Most sequential programming languages include control structures for looping and branching, such as **if** and **cases** and **while**. These may at first sight appear to undetermine the assertion that the statements of a sequential program are totally ordered. But these structures mean that the path through a program may vary from one run to another, only depends on various input data. However, if we repeatedly run a sequential program with the *same* input data, then it will *always* trace the same path.

Key nature of sequential programming In summary, the key nature of sequential programming is as follows:

- A sequential program is visualised as being executed up on a single processor architecture.
- A sequential program represents a single threaded computation. Operations of the sequential program are executed in a *total ordering*, i.e. given *any* pair of different machine instructions p and q, the execution of one must precede the other.
- A sequential program is *deterministic*, i.e. repeated runs of a sequential program with the *same* input data *always* produce the *same result* and trace the *same* execution sequence of operations.
- Control structures for looping and branching do not undetermine the above two statements.

Thus, writing a sequential program means finding some strict sequence of steps to achieve the desired end. This odd constraint is a legacy when programs are written only to be executed by sequential computers (uniprocessor systems), and sequential computers are machines that do one thing at a time.

2.6.2 Concurrent programming as partial ordering

Parallel computation

Now the first question: *does the total ordering of the sequential paradigm capture the real nature of computation?*

Let us get an answer to the question from the following story:

Once upon a time long before there was a computer, an intelligent Chinese princess wanted to get married. She announced to the generals of her father, then the emperor of China, that the general who was younger than 30 and could within one month find all the non-trivial factor of 368788194383 would be the prince. Before the deadline, a young general, called Bingxing (meaning *parallel*) brought her the numbers 7, 17, 23, 119, 161, 257, 391, 1799, 2737, 4369, 5911, 30583, 41377, 100487, 524287, 52684027769, 21693423199, 16034269321, 3099060457, 2290609903, 1434973519, 943192313, 204996217, 134741759, 84410207, 62390153, 12058601, 8912879, 3670009, and 703409. 524287. The princess was very happy because she made up the number by the formula

$$368788194383 = 7 \times 17 \times 23 \times 257 \times 524287$$

She asked the general how he found the number. He said that

1. he first calculated $607280 = \lceil \sqrt{368788194383} \rceil$; He then gave each of the numbers between 2 and 607280 to one of his 607279 soldiers, and ordered them to check if the numbers they got were factors of 368788194383,

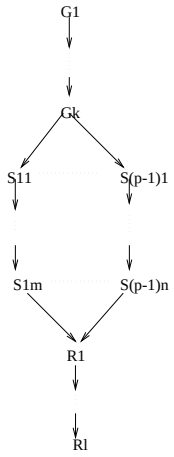


Figure 2: Partial ordering

2. 20 minutes later the soldiers who got 7, 17, 23, 119, 161, 257, 391, 1799, 2737, 4369, 5911, 30583, 41377, 100487, and 524287, reported that their number were factors.

3. Using these, General Binxing then found the rest factors: 52684027769, 21693423199, 16034269321, 3099060457, 2290609903, 1434973519, 943192313, 204996217, 134741759, 84410207, 62390153, 12058601, 8912879, 3670009, and 703409.

4. So he came to the princess with these numbers.

Finally, the princess happily married the young general.

Let $n = 274876858369$ and $p = 607280$. The *computation* in the story was to find all the non-trial factors of n. The computation was carried out by the operations of General Bingxing and his $p - 1$ soldiers. But those operations were **not carried out in a total order** : it could not be said that the operations of soldier s1 *must* precede the operations of soldiers s2, or the other way around. Indeed, the operations of the soldiers were carried out *in parallel*, though the operations of each individual soldier *might be assumed* to be carried out in sequence. The operations in the computation were carried out in a **partial order** as illustrated in Figure 2.

You may wonder what all the other generals in the story were doing with the problem. They were certainly trying hard as well because being the prince of the state meant a lot. But they were not as clever as General Bingxing. They were doing as what a sequential program nowadays does:

```
G1;
 .
 .
 .;
Gk;
```

```
S1;
 .
 .            (*all the operations by a general himself*)
 .;
S(p-1);

R1;
 .
 .
 .;
 Rl
```

Though they were not at all slow in doing divisions, it still needed more one year to do the 607279 required divisions. Thus, they were hopeless to become the prince.

The comparison between the two ways of carrying out the computation shows that sometimes the total ordering of sequential paradigm is not the most appropriate model of computation. It has been wrong (at least infeasible) in sequential programming to *always* constraining the execution in a way that is not required by the *logic* of the algorithm. It has been wrong because writing the statements in *any* particular sequence implies an ordering that is not essential to the algorithm, and thus over-specifying the computation. Therefore, we have to break away from sequential paradigm.

Multiprocessors systems

The second question: *is a parallel computation realizable by a computer*?

For simplicity, let us ignore the operations G1,...,Gk and R1,...,Rl. Suppose that we have a computer system with more than 607279 central processors. We can now replace each General Bingxing's 607279 soldiers by one of the central processor. Then each of the tasks S1,...,S(p-1) has its own private processor, so that these tasks could all be executed in parallel. Computing in this way is usually called *maximum parallel computing*.

Computer systems with multiple processors, such as transputers and computer networks, are also called *parallel computer systems*. Now, parallel computers are becoming more common, and computers are becoming more parallel. Thus, writing programs for parallel computers is becoming more imporatant.

Programming parallel computation

The third question: *how can we program parallel computer systems*?

To program the parallel computation illustrated in the story, we need to introduce some *programming notation* to indicate when parallel execution is sensible and when it is not. We can then have a mixture, within a single program, of sequential and parallel execution. We shall use a **cobegin/coend** structure[1]:

```
G1;
....  (*calculating p =607280*)
Gk;

COBEGIN
   S1; (*BEGIN S11; ...; S1m END*)
   S2;
   .....
   S(p-1) (*BEGIN  S(p-1)1; ...; S(p-1)n END*)
 COEND;
 R1;
 ...      (*reporting to the princess*)
 Rl
```

This notation implies the partial ordering of the computation events represented by Figure 2. It must be noted that the above program terminates *only if* all the processes in the **cobegin/coend** structure terminate.

2.6.3 Concurrent programming as interleaving

The fourth question: *can a uniprocessor system realize a parallel computation*?

For the program outline in the previous subsection, the question can be restated as:

> Does each of the blocks Si within the **cobegin/conend** structure *have* to be executed on its own processor?

Now assume there is a single processor system which supports *multitasking*, such as an UNIX system (Figure 3). We then can treat each of General Bingxing's soldiers as a user of the system with a task Si called a *process*.

During the execution of these processes, each process has its own process counter. The program counter *forks* to produce many process counters, these later *joining* to produce the program counter. In each processor cycle a process is *non-deterministically* chosen, and its next command is loaded and executed. This means that there may be many different possible threads. For example, suppose we have two processes,

[1] Dijkstra first introduced the **parbegin/parend** for a "parallel compound" in 1968.

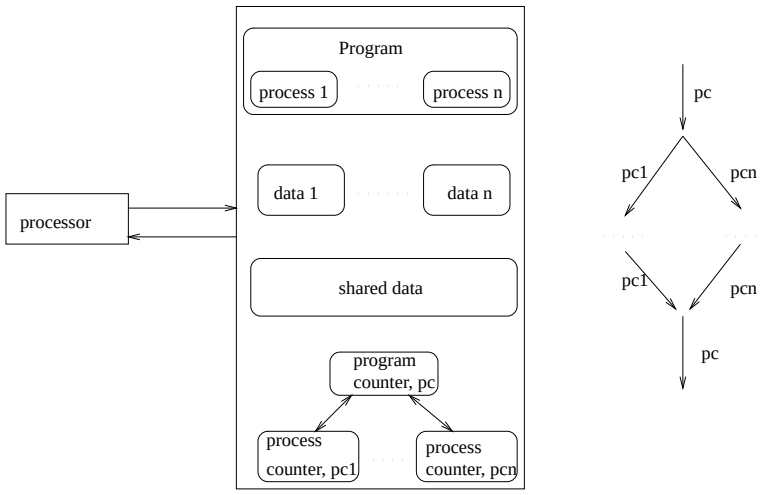


Figure 3: Uniprocessor multitasking system

```
PROCESS p1;            (*process declaration*)
BEGIN A; B; C END;     (*body of process p1*)

PROCESS p2;            (*process declaration*)
BEGIN S; T END;        (*body of process p1*)

BEGIN (*main program*)
  COBEGIN
    p1;
    p2
  COEND
END.
```

Each possible thread is one of the ten possible paths through a lattice (Figure 4). Each thread determines a unique *interleaving* of the statements in the bodies the two processes. One run of the program exhibits one of the ten threads, but *repeated* runs with the same input data may trace different threads.

The *outcome* of a finite concurrent computation is the final values of the variables. The number of outcomes is thus at most the number of possible threads. For

```
PROCESS p;
BEGIN S1; ...; Sm END;
PROCESS q;
BEGIN T1; ...; Tn END;

BEGIN
```

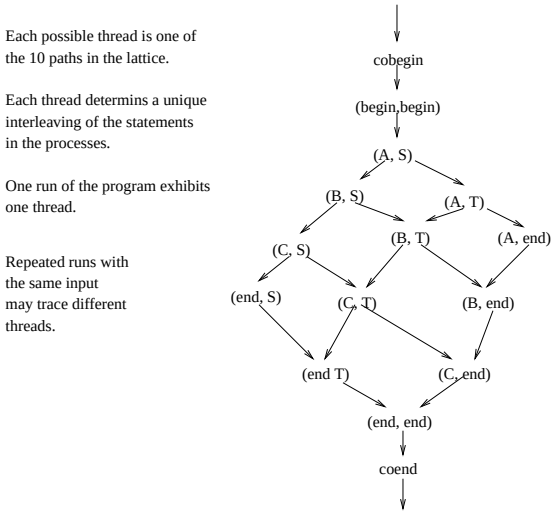


Figure 4: Possible interleavings of p1 and p2

```
  COBEGIN
     p; q
  COEND
END.
```

The number denoted by $f(m,n)$ of possible threads (interleavings) is the combination, i.e. the number of n-choice out of $n+m$.

$$C_m^{n+m} = C_n^{n+m} = \frac{(n+m)!}{n!m!}$$

n	:	1	2	3	4	5	6	7	8	9	10	$\cdots$
$f(n,n)$	:	2	6	20	70	252	924	3432	12870	48620	184756	$\cdots$

Note: Pascal-FC is implemented on a single processor system.

2.6.4 Concurrent computer systems

Computing with maximum parallelism and uniprocessor multitasking are the two extreme cases of concurrent computing. The first requires that each process has its own processor, while in the later all processes in the program compete for a single processor. On one hand, central processors are expensive. Real processors also sometimes break down. We may not, therefore, always have 524286 processors when executing S1 ... S524286. On the other hand, a uniprocessor system can, after all, execute at

most one machine instruction at any time. We certainly cannot expect very much increase in program execution speed from such a system.

The general case is a computer system with one or more processors some of which may be shared by more than one process (task). In our course, we shall call such a computer system a *concurrent computer system*. The programming problems that we will deal with are the problems in writing programs for concurrent computer systems.

However, when writing a concurrent program, we want our source code to be *portable*: we do not want to have to rewrite the **cobegin/coend** structure (perhaps even eliminate it altogether) each time when the program is to be run on a different hardware configuration. This means that the **cobegin/coend** structure provides such a *high level abstraction* so that it has some *fixed* meaning regardless of the number of processors available. The meaning we choose is that the statements in the **cobegin/coend** *may* overlap when the program is executed, but we cannot say that they *must* do so.

In keeping with our aim of using abstractions that are independent of hardware, we can adopt the following two ways (I personally use the first more often).

1. We use the interleaving model. This means that the meaning of

   ```
   PROCESS p1;
   BEGIN S1; ... ; Sm END;
   PROCESS p2;
   BEGIN T1; ...; Tn END;

   BEGIN
    COBEGIN p1; p2 COEND
   END.
   ```

   is define to be the set of all possible interleavings (merges) of the two sequences (S1,...,Sm) and (T1, ... , Tn). This is based on the fact that all concurrent computations can be simulated by a *logical* single-processor multi-tasking system.

2. We use the maximum parallel model, assuming each process is given a **logical processor** which executes one instruction at a time. This means the computation events are in a partial ordering.

How the single logical processor in the first model is broken into more than one *physical processor*, or how the logical processors in the second model are mapped onto physical ones, is an implementation detail in just the same way that implementation of multiplication (by multiply instruction or repeated addition) was. We can safely ignore this problem.

2.7 The motivation for concurrent programming

Here we give three reasons to show why concurrent programming is important.

- Using multiprocessor hardware, we can achieve faster execution speed by assigning different physical processors to different processors, as illustrated in our story of the Chinese princess.

- On a uniprocessor system, we can improve processor utilization by sharing the processor between a number of programs that run as concurrent processes. This is based on the fact that a single process is usually unable to keep a physical processor fully occupied throughout its execution. For example, the process may need to input some data (through the keyboard) before it can carry out any calculation, and in the meantime the CPU is idle, having nothing to do. It certainly makes sense to switch the processor to another process that *can* make use the expensive processor time.

  In this case, individual processes may take even longer to execute. The motivation on running a concurrent program on a uniprocessor is certainly not an increase in program execution speed.

  The exploitation of the possibility for such **multi-tasking** was one of the earliest applications of concurrency and still remains as one of the most important. Thus, historically the problems and solutions to concurrent programming emerged from the design and implementation of multi-tasking operating systems. Multi-tasking was partly motivated by the need to have multi-user access to very expensive hardware. Also there was the need to handle real-time process control and transaction processing as in a bank. These problems generated creative solutions which today form the basis of concurrent programming.

  For further reading, Lister gives a nice introduction to the essential problems and solutions in operating systems in his book:

  A.M. Lister, Fundamentals of Operating Systems, The Macilan press Ltd., 1975, Second Edition 1979.

- Many applications are inherently non-deterministic and concurrent. The sequential paradigm is simply not able to deal with those problems.

  The order in which program operations should occur may be determined at run time by events external to the computer system. This is because that it may not be possible to predict in what order these external events may occur.

  For example, if we are producing software to control traffic lights at a crossroads, we cannot predict in what order the various sensors will be triggered by vehicles approaching the junction, because that will depend on who decides to drive where, and at what time.

  For further exploitation with an example, read Burns & Davies' book, pp14-19.

2.8 Problems in concurrent programming

Although concurrent programming was in the earliest motivated by the problems of constructing operating systems, it is not the study of operating systems. Concurrent programming is the study of *abstract* programming problems posed under certain rules.

In our definition, a concurrent program consists of several sequential processes whose execution sequences are interleaved. In consequence, a concurrent program could produce different output results on different runs, even with the same input data. The sequential processes are not totally independent

– if they were so there would be nothing to study. They must *communicate* with each other in order to *synchronize* or to exchange data.

Synchronization In general, we do not want a *chaotic* program which could produce anything as a result. We only want a program to produce a result compatible with the purpose of the program. This implies that we *do* need some language facilities which provide us with a way of constraining the possible interleavings, so that a program will only produce the *acceptable* results (though there may be many). In other words, we must be provided with mechanisms to *synchronise* certain computation events whenever necessary.

For example, suppose we have a *writing process* and a *reading* process running concurrently, the writing process write a value to a variable and the reading process read the value from the *same* variable. It is often the case that a value can be read (by the reading process) only after the value has been written (by the writing process).

The way of constraining execution of concurrent programs is called **process synchronization**. The languages facilities, and how we use them to eliminating all the unacceptable interleavings, will be the central theme of the study of concurrent programming.

Communication Concurrent processes often need to communicate information with each other, to carry out cooperative computation. There are various ways in achieving proper communication between processes. Some languages use *common memory* as the means of passing information from one process to another. In our abstraction, common memory will be represented simply by *global variables* (*shared variables*) accessible to all processes. In this case, the synchronization facilities must be properly used to allow processes to know when valid information is ready. This is prefered for single processor computers.

With the introduction of distributed computing, it is not valid to assume that a common central memory exists. We have to use sending and receiving signals (often called *message passing*) as the means of process communication. In this case synchronization is provided through communication.

Therefore, synchronization and communication are complementary concepts and equally important in concurrent programming.

2.9 Seat Reservation Problem

Problem description: A central computer connected to remote terminals via communication links is used to automate seat reservation for a concert hall. The terminals are located in booking offices in different cities. When a client enters a booking office, the clerk displays the current state of the reservation on a screen. The client chooses one of the free seats and the clerk enters the number of the chosen seat at the terminal. A ticket for that seat is then issued.

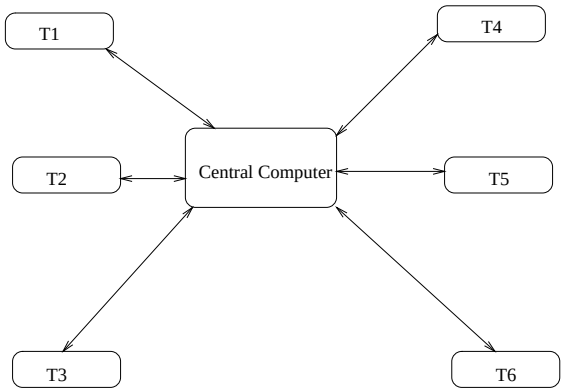


Figure 5: Seat reservation problem

Requirements:

1. no multiple booking for any seat;
2. allow the client a free choice among the available seat.

A first attempt We write a concurrent program which consists of a number of processes called **terminal handlers**, one for each terminal:

```
COBEGIN

  HANDLER1;
  HANDLER2;
  HANDLER3;
     .
     .
     .
  HANDLERn

COEND
```

The pseudo code for process HANDLERi:

```
REPEAT
   display seat on screen;
   read(client_choice);
   seat[client-choice]:=reserved;
   issue ticket
FOREVER
```

Problem: This obviously violates Requirement 1. If two clients were shown the seating plan at the approximately same time, say in different booking offices, they could both choose the same seat and tell the clerks to book it.

A second attempt Now in the terminal handler, we insert a *check* before committing a reservation to ensure that the position has not changed since the display was presented.

```
REPEAT
   IF sold-out THEN
   BEGIN
     present a suitable display;
     await further instructions
   END
   ELSE
   BEGIN
     success:=false;
     REPEAT
       display seat plan on screen;
       read(client_choice);
       IF seat[client_choice]=free THEN  (*check before reservation*)
       BEGIN
          seat[client_choice]:=reserved;
          success:=true;
          issue ticket
       END
       ELSE
          give_error_message
     UNTIL success
    END
FOREVER
```

However, this has not solved the problem, and multiple bookings are still possible. For simplicity, consider the case of two terminal handers HANDLER1 and HANDERLER2. We may still have the following interleaving.

1. HANDLER1 check seat[k] and finds it is free,
2. HANDLER2 check seat[k] and find it is free,
3. HANDLER1 marks seat[k] reserved,
4. HANDLER2 marks seat[k] reserved.

2.9.1 Critical sections and Mutual exclusion

To essentially see why both of the two solutions to the seat reservation problem have failed, we take the second attempt and look at its machine level execution. The compiler would translate the **if** statement into a sequence of machine instructions:

```
o1:  load seat[client_choice] to CPU register
o2: test value of CPU register
o3: go to L1 if reserved
o4: set seat[client_choice] to reserved
o5: set success to true
o6: issue ticket
o7: go to L2
L1: code to output error message
L2: code following if statement
```

Each machine instruction of a terminal handler is executed **atomically**, i.e. no other process can observe its effect or interfere with its execution. But between any two successive machine instructions of one handler interference from another handler may well possibly occur. Indeed, Requirement 1 was violated because the sequence o1,o2,o3,o4 in one handler may be interleaved with the same sequence of another handler.

To meet Requirement 1, what we require is that the sequence of instructions o1,o2,o3,o4 in one terminal handler must not interleave with that of another terminal handler. In other words, the executions of this sequence in different terminal handlers must be under **mutual exclusion**. This means that at any time no more than one handler is within this *critical section*. From the interleaving point of view, these operations from different terminal handlers cannot be interleaved. From the partial ordering point of view, executions of these sequences in different terminal handlers cannot be overlapped. Any terminal handler which wants to execute o1 must wait until no other is executing any of these four operations.

In general, a *critical section* is a section of code in a process that accesses one or more *shared variables* in a *rea-update-write* fashion. Thus a critical section is always associated with a set of shared variables representing shared resources. We say two critical section in two processes *correspond* to each other if they are associated with same shared variables.

Mutual exclusion requires that at any time no two or more processes can execute their corresponding critical sections.

What we need now is some language synchronization facilities by which we can tell the compiler the corresponding critical sections in different processes. In our example, we want the complier to generate the following machine instructions, where those contained in angle brackets form a critical section of a terminal handler.

<

```
o1:  load seat[client_choice] to CPU register
o2: test value of CPU register
o3: go o L1 if reserved
o4: set seat[client_choice] to reserved
>
o5: set success to true
o6: issue ticket
o7: goto L2
L1: code to output error message
L2: code following if statement
```

2.10 Concurrent programming in Pascal-FC

Sequential Pascal (and the subset used in this course) must be augmented by concurrent programming constructs. The concurrent processes are declared *as* Sequential Pascal procedures, but with a reserved word **process** rather than **procedure**. The declared and only the declared process *names* (identifiers) must be listed within the **cobegin/coend** structure for concurrent execution, separated by ";". Here is a simple schema outline of a concurrent Pascal-FC program:

```
PROGRAM schema1;  (*program header*)
  TYPE
  (*type declarations as in the sequential Pascal*)
  VAR
  (*global variable declarations as in the sequential Pascal*)
  PROCESS p1; (*process declaration for p1*)
     VAR
     (*local variable declarations*)
     BEGIN
      ...     (*body of process p1*)
     END;
  PROCESS p2; (*process declaration for p2*)
     VAR
     (*local variable declarations*)
     BEGIN
      ...    (*body of p2*)
     END;
  BEGIN      (*the main program*)
      (*sequential execution prior to *)
      (*execution of concurrent processes*)
      COBEGIN
         p1;
         p2 (*the textual order of p1
             and p2 is inessential*)
      COEND
      (*sequential execution after concurrent  *)
      (*processes have all terminated*)
    END.
```

A concurrent program terminates *only if* all the concurrent processes terminate.

Example 1

```
 PROGRAM example1; VAR x,y,
y: integer; PROCESS p1; VAR l: integer; BEGIN  l:=1; x:=x+l END;
PROCESS p2; VAR l: integer; BEGIN l:=2; y:=y+l END; BEGIN (*main
program*)
  x:=0; y:=0;
  COBEGIN p1; p2 COEND;
  z:=x+y
END.
```

Notes on example 1:

- the program sequentially assigns 0 to x and y, then increments x and y concurrently, finally assigns $x + y$ to z;
- x y, and z are global variables whose scope in the whole program, declared in the main program, and initialized in the main program body before COBEGIN;
- the two l's are local, each to one process, whose scope is limited to that process,
- local variables in different processes are different, even if they happen to have the same name.

Process type Sometimes the processes in a concurrent program can be declared as instances of a certain **type**. Pascal-FC allows us to declare a **process type** in the following way.

```
 PROGRAM schema2;
  (*definition of process types*)
  PROCESS TYPE PTYPE(i:integer);
  BEGIN
  .....
  END;
  VAR
    (*definition of process objects*)
    p1,p2,p3: PTYPE;
    ........
BEGIN (*main program}
   ......
   COBEGIN
      p1(1);
      p2(2);
      p3(3)
   COEND
   .......
END.
```

Using process type like this, we can use *loop* within **cobegin/coend** to make programs simpler.

```
PROGRAM simple;
 PROCESS TYPE PTYPE(i: integer);
 BEGIN
 .....
 END;
 VAR
   count: integer;
   p: ARRAY[1..3] of PTYPE;
 BEGIN
   COBEGIN  (*this is semantically the same as *)
      FOR count:=1 TO 3 DO
      (*COBEGIN p[1](1);p[2](2);p[3](3) COEND*)
      p[count](count)
   COEND
  END.
```

Example 2

```
 PROGRAM example2;
(*parallel initialize all
                  *elements of a[1:3] to 0)
  VAR a:ARRAY[1..3] OF integer;
  PROCESS TYPE PTYPE(i: integer);
  BEGIN a[i]:=0 END;
  VAR
    count: integer;
    p: ARRAY[1..3] of PTYPE;
  BEGIN
    COBEGIN
       FOR count:=1 TO 3 DO
       p[count](count)
    COEND
   END.
```

Standard Pascal features not supported by Pascal-FC The following facilities are not supported by Pascal-FC.

1. **Files**. There are no files, apart from the standard input and output. Therefore, the only form of the program header is:

   ```
   program_header ::= PROGRAM identifier;
   ```

2. The **with statement.** Record types may be declared as in standard Pascal, but there is no **with** statement.

3. **Sets.** There are no set types.

4. **Subrange types.** Subrange types are not provided.

5. **Dynamic storage.** There are no pointer types, and the standard procedures **new** and **dispose** are not provided.

6. **Packed data.** The language does not support the **packed** quantifier, and consequently there is no facility for string variables (though string literals in **write** and **writeln** procedures are supported).

The reason for removing these features is just for simplicity. Pascal-FC is designed as a *teaching language*, not as a language to be used in the software industry. It intends to provide more concurrent facilities by loosing efficiency in the language implementation.

Useful extensions to standard Pascal When writing a sequential process, we are allowed to use the following abstractions.

1. **Extension to the repeat ... until loop.** In a concurrent program, cyclic process may execute indefinitely. Such a process can be written by using a loop REPEAT...FOREVER, which is semantically equivalent to the loop REPEAT ... UNTIL false.
2. The **null statement.** The **null** statement has been added as an alternative to the Pascal's empty statement. Its execution has no effect.
3. The **random function.** This function returns a pseudorandom integer. A call has the form:

   ```
   i:=random(n)
   ```

   which returns an integer in the range 0...abs(n).

In addition to these, we certainly need extensions for process communication and synchronization. We will introduce them along the progress in the course.

Other characteristics

1. **Case of alphabetic in the source program.** Case is not significant, except in string and **char** literals. I will use upper cases for reserved words as a convention, while the book uses bold lower case.
2. **Order of declarations.** No like the original Pascal, there is no restrictions on the order of declarations for **label, const, type, var, procedure** and **function**.

Further reading: Visit the website for Pascal-FC at

http://www-users.cs.york.ac.uk/ burns/pf.html.

2.11 Exercises

1. In Pascal-FC, give the pseudocode of the program for the seat reservation problem. You do not have to care about the multiple booking problem in this program.
2. **Parallel sort:** Write a concurrent program in Pascal-FC to sort an array of n integers. The program should have two processes which are to sort the halves of the array in parallel; and finally merges the two sorted halves.
3. Write a two-process concurrent program to find the mean of n numbers.

4. Assume that each of the assignments in the following program, except for $x := 1$ which is implemented as one single machine instruction, is implemented by three machine instructions that load a register, add a value to the register, then store the result. What are the all possible outputs of the following programs?

```
PROGRAM P1; VAR
  x: integer;
PROCESS p1;
BEGIN
x:=x+1; x:=x+2
END;
PROCESS p2;
BEGIN
x:=x+1;x:=x+1
END;
BEGIN
  x:=0;
  COBEGIN
    p1;p2
  COEND
END.
```

```
PROGRAM P2;
VAR
  x: integer;
PROCESS p1;
BEGIN
x:=1
END;
PROCESS p2;
BEGIN
x:=x+1;x:=x+1
END;
BEGIN
  x:=0;
  COBEGIN
    p1;p2
  COEND
END.
```

5. Consider the concurrent program:

```
PROGRAM P2;
VAR
  x,y: integer;
PROCESS p1;
BEGIN
x:=x+1;y:=y+x
END;
PROCESS p2;
```

```
BEGIN
x:=x+2;y:=y-x
END;
BEGIN
  x:=0;y:=0;
  COBEGIN
    p1;p2
  COEND
END.
```

(a) Suppose the above assignments are implemented by single machine instructions and hence are atomic. How many possible interleavings are there? What are the possible final values of x and y?

(b) Suppose each of the above assignments is implemented by three *atomic* instructions: load a register, add or subtract a value from that register, then store the result. How many possible interleavings are there? What are the possible final results of x and y?

6. Write a three-process concurrent program to multiply two 3×3 matrices.

7. Write a program to find the maximum value in an integer array $a[1..n]$ by searching even and odd subscripts of a in parallel.

8. Can a multiprocessor system allow us to solve computational problems that *cannot* be solved with a uniprocessor system, or does it merely permit us to solve the same problems more quickly?

3 MUTUAL EXCLUSION

The goals of the chapter

- Understand the two different classes of entities in a concurrent program: *active* ones and *passive* ones.
- Discuss the mutual exclusion and condition synchronization and their solutions without using synchronization primitives.
- Understand some important concepts of concurrent programming, including *multiple update, busy-waiting, livelock, deadlock, safety,* and *liveness*.

3.1 Process interaction

All concurrent languages, though may have various constructs, must in one way or another provide three fundamental facilities, which allow:

1. the expression of concurrency
2. inter-process synchronization
3. inter-process communication

We have seen in Chapter 1, Pascal-FC provides us with the **cobegin/coend** structure to express concurrent execution of processes. The rest of the course is main concerned with process interactions: inter-process synchronization and communication. As discussed in Chapter 1, the key nature of processes behaviour in a concurrent program is:

- processes may behave *independently* from each other – parallel execution,
- processes may *cooperate* with each other by means of synchronization and communication to achieve a common goal.

There is a trade-off between the two types of processes behaviour. First, programs that consist of *totally independent* processes are rare and not very much interesting. Second, too much interaction means too little parallel computation for *significant processing*, since synchronization and communication both take time. Thus, the meaningful concurrency is possible only if the processes are *loosely connected*. This means that for most of the execution time, processes behave independently of each other, but they must also compete for resources and coordinate their activities in order to meet the program requirement. They achieve this interaction by means of synchronization and communication.

3.2 Active and passive entities

In concurrent programs, there are two kinds of *entities* (called **objects** in object-oriented program paradigm): active entities and passive entities.

Active entities in a program are those that undertake spontaneous actions to make the computation of the program proceed. Clearly, these active entities will be expressed as processes.

Passive entities in the program are those which perform actions only when 'invoked' ('requested') by an active entity. They are needed to encapsulate resources needed by processes, and they *must control* how the resources are accessed. For example, a bounded data buffer cannot have an element removed if it is empty, or an element added if it is full. We will see that passive objects can also been represented by processes, and other primitives as well.

Certainly, there are other non-active entities, such as the usual data variables, that can be accessed without any restriction. We call such an entity a *neutral entity*.

3.3 Communication via shared variable

The simplest way for two or more processes to communicate is via a *shared variable* x that is in the scope of the communicating processes. One process p *sends* messages by *writing* (*updating*) the variable, e.g.

```
x:=e   (*p writes x*)
```

Another process q *receives* a message sent by p by *reading* the variable, i.e. assigning the valued of the variable to another variable. More generally, we may have an assignment in q

```
y:=e'

(* e' contains x:
*q receives the value of x and uses it *)
```

A special case of this is that a shared variable is updated by two (or more) processes, for example

```
PROCESS p;
  BEGIN ...; x:=x+1; ... END;
PROCESS q;
  BEGIN ...; x:=x+1; ...END;
BEGIN
  ......
  COBEGIN p;q COEND;
```

```
  ......
END.
```

To illustrate the problems of shared variables and *multiple updating*, we consider an example called the *Ornamental Gardens Problem*.

3.3.1 The Ornamental Gardens problem

A large ornamental garden is open to members of the public. However, to enter the garden, one has to pay an admission fee. Entry is gained by *two* tunstiles. The management of the garden want to be able to determine, at any time, the *total* number of visitors *in* the garden. They propose that a computer system should be installed which has connections to each turnstile and a terminal from which the management can get the current total. We shall attempt to construct some prototype software for the proposed system.

To count the visitors as they enter and leave the garden, we suppose that the turnstiles are able to send a signal to the computer to indicate each arrival and each departure. To develop a software prototype, we will simply construct a simulation of the system and ignore the hardware details.

As there are two turnstiles behaving in parallel and the order of the their events should not be predictable: we do not know in advance when or from which turnstile visitors will arrive or leave. It is obviously an application of concurrent programming. We simulate the system as follows:

- each turnstile is simulated by a process in the program, the two processes run in parallel;
- a shared (global) integer variable will represent the current number of visitors in the garden,
- a terminal handler process (which will not be considered further) will provide the management information.

We only want to illustrate the multiple updating problem. Thus, we further simplify the problem by only counting the visitors as they enter. Furthermore, we only simulate an experiment in which 20 people enter the garden from each turnstile: no-one is allowed to leave until the experiment has been completed. The global counter should show that there are 40 people in the garden at the completion.

We provide the following program for the simulation:

```
 PROGRAM gardens1; VAR count: integer;
(*shared variable*)

PROCESS turnstile1;
VAR
  loop: integer;    (*local variable of turnstile1*)
```

```
BEGIN
  FOR loop:= 1 to 20 DO
  count:=count+1
END; (*turnstile1*)

PROCESS turnstile2;
VAR
  loop: integer;    (*local variable of turnstile2*)
BEGIN
  FOR loop:= 1 to 20 DO
  count:=count+1
END; (*turnstile2*)

BEGIN (*the main program*)
   count:=0;        (*initialization*)
   COBEGIN
      turnstile1; turnstile2
   COEND;
   WRITELN('Total admitted: ', count)
           (*output the result*)
END.
```

Does the program gardens1 work for our purpose?

The answer is NO. If you run the program many times, you may find that the following values are all possible outputs (i.e. the final value of variable $count$):

```
   25, 29, 31, 20, 21, 26, 27, 28, 18, 31, 35, 40
```

But 40 is the only one we want. Where did it go wrong then?

It is certainly *not* because one of the turnstile processes may not execute all its statements. Bear in mind that the program terminates only after both processes terminate. Thus, they both do execute all their statements.

The problem is fundamentally the same as illustrated in the Seat Reservation Example. At the machine level execution, the update action on the variable $count$ is broken into three instructions:

1. load the value of $count$ into a processor register,
2. increment the value in the register,
3. store the value in the register at the address for $count$.

Since we have written the program in the way that the two tunstile processes are *total independent* of one another. It is possible for their actions to interfere with each other. Consider the case that when $count$

has the value 0, and the process counters are both at the beginning of 'count := count+1'. The following ordering of operations is possible:

1. turnstile1 loads the value 0 of $count$ into a CPU register,
2. turnstile2 loads the value 0 of $count$ into a CPU register,
3. turnstile2 increments the value in its register (becomes 1),
4. turnstile1 increments the value in its register (becomes 1),
5. turnstile2 stores the value 1 in its register at the address of $count$ (the value of $count$ becomes 1),
6. turnstile1 stores the value 1 in its register at the address of $count$ (the value of $count$ becomes 1)

At the completion of step (6), each processes has executed its assignment $count := count + 1$ once. But the value of $count$ has had one increment though it *appears* to have had two increments. If we run the program gardens1 from the beginning to the end, we may have much fewer increments than we *appear* to have in the program. In other words, by using this program we have lost a lot of increments that we wanted.

The mistake we have made in this program is that we have treated $count$ as a *neutral* object. We should treat it as a *passive* object so that the access to it should be controlled under certain rules. The solution to the problem is the same for the Seat Reservation Problem. What we need is to code the updates to the shared variable in a Critical Section which must be accessed under mutual exclusion.

Condition synchronization Consider a case that a shared variable of a program can only be *updated* by *one* process (while it can be *read* by another). There is certainly no multiple updating problem here. Do we still have to treat such a shared variable as a passive object? In order words, is synchronization still needed for the access to that variable? The answer is yes. Consider a simple version of the Sender-Receiver problem:

```
PROGRAM passing1;
VAR value: integer;

   PROCESS sender;
   VAR message : integer;
   BEGIN
      .......
      message:=42;
      value:= message;
      (*updates the shared variable*)
      .......
    END;

    PROCESS receiver;
```

```
   VAR data: integer;
   BEGIN
      ......
      data:=value;     (*reads the shared variable*)
      write(data);
      ......
   END;
BEGIN  (*main program*)
   COBEGIN sender; receiver COEND
END.
```

Error: If the $receiver$ reads the $value$ before the $sender$ has written into it, then it will get an *undefined* value rather than 42 that is intended.

So we need synchronization to guarantee that, in this situation, the *updating* action of $sender$ *must* precede the *reading* action of $receiver$.

Condition synchronization is the name given to the required property that one process should not perform an event until some other process has undertaken a designated action to make a *condition occur*.

The Sender-Receiver problem is also a special case of the general Producer-Consumer problem: the sender *produces* an item by writing into the shared variable, and the receiver *consumes* the written item by reading it from the shared variable.

3.4 Old-fashioned recipes for synchronization

We have seen that the naive use of shared variables for inter-process communication is not sufficient. We need some way of providing inter-process synchronization. As far as shared variables are concerned, there are two forms of synchronization in concurrent programs: *mutual exclusion* and *condition synchronization*. Mutual exclusion is concerned with ensuring that *critical sections* of statements that access shared objects are not executed at the same time. Condition synchronization is concerned with ensuring that a process delays if necessary until a given condition occurs. It is usually the case that events in another process will make the required condition occur. For example, communication between a sender process and a receiver process is often implemented using a shared buffer. Mutual exclusion is used to ensure that the sender and receiver do not access the buffer at the same time – hence a partially written message is not read. Condition synchronization is used to ensure that a message is not received before it has not been sent and that a message is not overwritten before it has been received.

We said that we would need new language features (primitives) for synchronization. But *is it possible to program synchronization by using only the data types normally provided in a sequential language?* Or *is it possible to achieve mutual exclusion and condition synchronization without using new language features?*

We shall show that the answer is YES. But we want also to show the solutions by this old-fashioned

recipe are inefficient, overcomplicated, and very much error-prone. This will motivate the need of high level primitives and abstractions.

3.4.1 An old-fashioned recipe for condition synchronization

Consider the Sender-Receiver (Producer-Consumer) problem again. We modify the program $passing1$ by introducing another shared variable $flag$, which delays the receiver process when its value is false:

```
PROGRAM passing2;
VAR value: integer;
    flag: boolean;

    PROCESS sender;
    VAR message: integer;
    BEGIN
       message:=42;
       value:= message;
       flag:=true (*to indicate receiver to read*)
    END;

    PROCESS receiver;
    VAR data: integer;
    BEGIN (*busy waiting loop*)
       WHILE NOT flag DO NULL;
       (*read after flag becomes true*)
       data:=value;
       write(data)
    END;
  BEGIN
     flag:=false;
     COBEGIN sender;receiver COEND
  END.
```

The program passing2 has the following feature:

- no use of any new primitive instructions, and the shared variables $value$ and $flag$ which are just simple variables in sequential Pascal;
- *busy waiting* is used: this means that a process that finds itself unable to proceed immediately remains executable and continues to execute instructions while it is waiting;
- the program is very inefficient, as the process is using up valuable processor cycles while doing nothing useful.

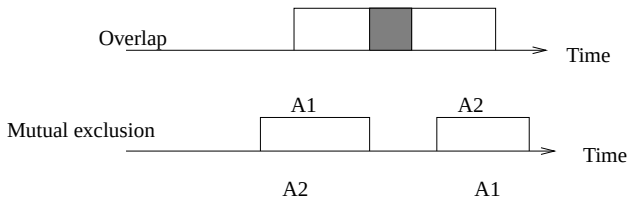


Figure 6: Mutual exclusion and overlap

3.4.2 Old-fashioned recipes for mutual exclusion

From the earlier discussions, we must have got the feeling that mutual exclusion is a basic problem in concurrent programming. It is the abstraction of many synchronization problems. It turns out, as we will show, a very difficult problem as well.

We say that activity A_1 of process P_1 and activity A_2 of process P_2 *must exclude* each other if the execution of A_1 may not overlap with the execution of A_2. This means that if P_1 and P_2 simultaneously attempt to execute their respective activities A_1 and A_2, then we must ensure that only one of them succeeds. The losing process must be blocked, that is it must not proceed until the winning process P_i completes the execution of its activity A_i. This is illustrated in Figure 6

The most common example of the need for mutual exclusion in real systems is resource allocation. Obviously, two tapes cannot be mounted simultaneously on the same tape drive. Some provision must be made for deciding which process will be allocated a free drive and some provision must be made to block processes which request a drive when none is free. There is an obvious solution: run only one job at a time. But this defeats one of the main aims of concurrent programming – parallel execution of several processes.

The *abstract* mutual exclusion problem will be expressed:

```
non-critical section1
entry-protocol
critical section
exit-protocol
non-critical section2
```

The *non-critical sections* represents some sets of actions in a process that can be executed concurrently with any part of the other processes in the program without the danger of interference. After the completion of non-critical section1, the process needs to enter the *critical section.* It will execute certain sequences of instructions, called *protocols* before and possibly after the critical section. These protocols will ensure that the critical section is in fact executed so as to exclude the critical sections in the other processes of the program. Thus, we are mainly concerned with these protocols.

This section considers the mutual exclusion problem for two processes as follows:

Two processes P_1 and P_2 are each in an infinite loop consisting of two sections, a critical section and a non-critical section. Let $critic_1$ and $critic_2$ represent the critical sections of P_1 and P_2, and $nocritic_1$ and $noncritc_2$ for their non-critical sections, respectively. The execution of The critical sections $critic_1$ and $critic_2$ must be under mutual exclusion, i.e. must not overlap.

We present Dijkstra's (Dijkstra, 1968) development step-by-step from failed attempts to the elegant Dekker's solution, to show how difficult the problem is.

Each of the attempts will be a program outline of the following form:

```
 PROGRAM outline;
   VAR .....;
   PROCESS P1;
     VAR.....;
     BEGIN
        REPEAT
          entry-protocol1
          critic1
          exit-protocol1
          noncritic1
        FOREVER;
   PROCESS P2;
     VAR.....;
     BEGIN
        REPEAT
          entry-protocol2
          critic2
          exit-protocol2
          noncritic2
        FOREVER;
BEGIN (*main program*)
  .......
  COBEGIN P1; P2 COEND
END
```

First attempt Similar to the idea of the busy-wait condition synchronization algorithm in the Sender-Receiver problem, a process uses a *flag* to delay the other process.

```
PROGRAM attempt1;
  VAR flag1, flag2: boolean;
  PROCESS P1;
    BEGING
       REPEAT
         WHILE flag2 DO NULL; (*busy-wait*)
         flag1:=true; (*announce*)
         critic1;
```

```
        flag1:=false; (*exit protocol*)
        noncritic1
       FOREVER
    END;
  PROCESS P2;
    BEGIN
       REPEAT
         WHILE flag1 DO NULL; (*busy-wait*)
         flag2:=true; (*announce*)
         critic1;
         flag2:=false; (*exit protocol*)
         noncritic2
        FOREVER
    END;
BEGIN (*main program*)
  flag1:=false; flag2:=false;
  COBEGIN P1;P2 COEND
END.
```

This violates the mutual exclusion requirement: consider the following interleaving

```
                          flag1     flag2
      Initially           false     false
      P1 checks flag2     false     false
      P2 checks flag1     false     false
      P1 sets   flag1     true      false
      P2 sets   flag2     true      true
      P1 enters critic1   true      true
      P2 enter2 critic2   true      true
```

Second attempt

```
PROGRAM attempt2;
  VAR flag1,flag2: boolean;
  PROCESS P1;
    BEGIN
      REPEAT
        flag1:=true; (*announce intent to enter*)
        WHILE flag2 do NULL;
        (*busy wait if P2 is in its *)
        (*critical section*)
        critic1;
        flag1:=false; (*exit protocol*)
        noncritic1
      FOREVER
     END;
  PROCESS P2;
```

```
    BEGIN
      REPEAT
        flag2:=true; (*announce intent to enter*)
        WHILE flag1 do NULL;
        (*busy wait if P1 is in its *)
        (*critical section*)
        critic2;
        flag1:=false;(*exit protocol*)
        noncritic2
      FOREVER
     END;
BEGIN (*main program*)
  flag1:=true;
  flag2:=true;
  COBEGIN P1;P2 COEND
END.
```

NB:

- The mutual exclusion requirement is met by the program: when P1 is in $critc_1$, $flag1$ must be true, P2 has to wait, similar for the case when P2 is in $critic_2$.
- But, the program may **deadlock**: both P1 and P2 keep remaining in busy waiting forever without making any progress. This happens because the program allows the following interleaving:

```
                    flag1      flag2
 Initially          false      false
 P1 sets flag1      true       false
 P2 sets flag2      true       true
 P1 checks flag2    true       true
 P2 checks flag1    true       true
 .....both busy-wait without making progress......
```

In this case no process can enter its critical section only after that it could set its flag to false. *Processes have no other way to withdraw their announcement of their intent to enter the critical sections.* The program hopelessly deadlocked.

In general, **deadlock** means that the execution of the program is no longer *possibly* doing any (useful) work.

We need to add a requirement for the program to rule out deadlock:

Progress Requirement: if both processes continuously request for entry into their critical sections, one of them must *eventually* succeed to do so.

Third attempt In the previous solution, when P_1 sets $flag1$ to indicate its intention to enter its critical section, it also turns out that it is *insisting* on its right to enter the critical section. Deadlock occurs when both announced their intention and no one wants to back off. In our next attempt, we correct this *stubborn* behaviour by having a process *temporarily* give up its intention to enter its critical section to give the other process a chance to do so.

```
PROGRAM
attempt3; VAR flag1,flag2: boolean; PROCESS P1; BEGIN
  REPEAT
    flag1:=true;        (*announce its intention*)
    WHILE flag2=true DO
       BEGIN
         flag1:=false;  (*give P2 a chance*)
         flag1:=true    (*reannounce its intention*)
        END;
    critic1;
    flag1:=false;
    noncritic1
   FOREVER
END;
PROCESS P2;
BEGIN
  REPEAT
    flag2:=true;
    WHILE flag1=true DO
       BEGIN
         flag2:=false;  (*give P2 a chance*)
         flag2:=true    (*reannounce its intention*)
        END;
    critic2;
    flag2:=false;
    noncritic2
   FOREVER
END;
BEGIN (*main program*)
   flag1:=false; flag2:=false;
   COBEGIN P1;P2 COEND
END.
```

This program, like the previous one, meets the mutual exclusion requirement. But we may have the following situation:

```
                    flag1    flag2

Initially           false    false
 P1 sets flag1      true    false
 P2 sets flag2      true    true
```

```
P1 checks flag2     true    true
P2 checks flag1     true   true
P1 downs flag1      false   true
P2 downs flag2      false  false
P1 sets flag1       true   false
P2 sets flag2       true   true
... REPEAT FOREVER    ..............
```

In this situation, both processes are looping on a protocol which is certainly not useful computation and this situation appears to be similar to the previous attempt. However, it is a quite different situation and we call it situation a **livelock**. In the previous attempt the solution is hopeless: from the instant that the program is deadlocked, all future execution sequences remain deadlocked (no hope for breaking the lock at all). In this solution however, the possibility of remaining in the loop forever is *very very* small. In both practice and principle, keep the execution running, one of the processes will be eventually freed. In such a sense, the loop of on this protocol is still *useful*.

In general, **livelock** is a situation in which all the processes are *indefinitely delayed*.

Livelock is less serious than deadlock since the computer is still doing (presumably) useful work. However, livelock is difficult to discover and correct (it is hopeless to discover livelock by testing) since it can only happen in complex scenarios.

Precisely speaking, attempt2 indeed meets the progress requirement since the processes do *not continuously* request for the entry to their critical sections, though they may request *infinitely often* without success. Therefore, we need to strengthen the progress requirement as follows.

New progress requirement: whenever both processes request *infinitely often* for the entry to the critical sections, one should eventually enter its critical section.

Fourth attempt The problem with the above three attempts seems to be at the setting of one's own flag and the check on he other's cannot be done as one indivisible action. The fourth attempt is to use just one flag that indicates whose *turn* it is to enter the critical section next.

```
PROGRAM attempt4;
   VAR  turn: integer;
   PROCESS P1;
     BEGIN
        REPEAT
           WHILE turn=2 DO NULL;
           critic1;
           turn:=2; (*next is P2's turn*)
           noncritc1
        FOREVER
      END;
   PROCESS P2;
```

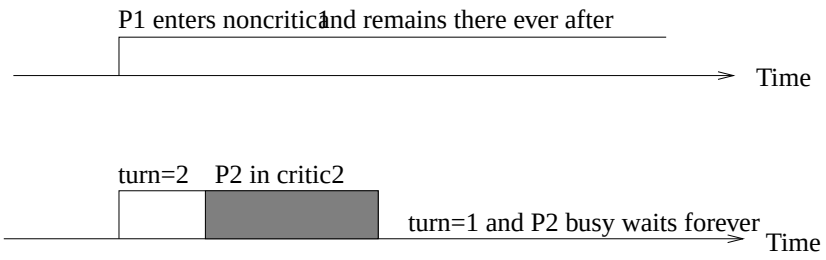


Figure 7: Starvation

```
     BEGIN
        REPEAT
           WHILE turn=1 DO NULL;
           critic2;
           turn:=1; (*next is P1's turn*)
           noncritc2
        FOREVER
      END;
BEGIN (*main program*)
  turn:=1; (*can also be 2*)
  COBEGIN P1;P2 COEND
END.
```

This program meets both the mutual exclusion requirement and the new progress requirement (WHY?).

But the program may suffer from **Starvation** (or **indefinite postponement**, or **lockout**). If one process, say P1 is indefinitely postponed in its non-critical section (e.g. an infinite loop in noncritic1, or P1's processor crashed during the execution of critic1), it will never intent to enter the critical section again. At the same time P2 keeps trying to enter its critical section thus, $turn$ will be eventually be 1 and remains as 1 forever since P1 will never change it to 2 again. This will block P2 from entering its critical section again even though it keeps trying (see the time diagram in Figure 7).

Starvation is a condition that while the program as a whole still making progress, but a set of processes do not because their request for some conditions will never be fulfilled.

In this case, starvation is *local deadlock* in the sense that the program as whole still making progress but a subset of processes cannot. Starvation is sometimes classified into livelock or deadlock according to the nature of the 'lock'.

Note that attempt4 does meet both the mutual exclusion and the new progress requirements, since not all processes request infinitely often but only one does. The new progress requirement needs further strengthened.

Liveness requirement: If one process requests for the entry infinitely often, it will eventually enter its critical section.

Dekker's solution Dekker's solution is an ingenious combination of the third and fourth attempted solution.

In the third attempt, each process behaves like a *gentleman*. When a process wants to enter its critical section, it announces it intention and then checks on another's intention. If it finds that the other intents to enter as well, it kindly give up its right to enter *but only temporarily*. This may unfortunately lead to an indefinite loop on giving up and reclaiming the right to enter the critications.

The fourth attempt could be viewed as processes passing a shared key ($turn$) to the critical sections between them. Unfortunately, if a process passed the key to another process which will not come back to return the key again after leaving its critical section (stays in the noncritical section or terminates), the first process will be ever since *lockedout*

Dekker's solution solves the problem by using two flags $flag1$ and $flag2$, and a shared key $turn$ which will be only used when both processes intent to enter the critical sections. A process intents to enter its critical section sets its flag to announce its intention. It then checks on another's intention:

- if the other does not intent to enter, then it enters its critical section (the gate is open to him),
- if the other intents to enter as well (the door is locked to it), then it needs to see whether it has the key,
- if it has, then it can enter, if not it gives up its right and wait for the key to be returned to it before reclaiming its right.

```
PROGRAM Dekker; VAR      turn: integer;
         flag1,flag2: boolean;
PROCESS   P1;
BEGIN
   REPEAT
     flag1:=true;               (*announce intention*)
     WHILE flag2 DO             (*check on flag2*)
        IF turn =2 THEN         (*if no key   *)
           BEGIN
             flag1:=false;       (*give up right*)
             WHILE turn=2 DO NULL;(*busy wait*)
             flag1:=true         (*reclaim right*)
            END;
       critic1;
       turn:=2;                  (*return the key*)
       flag1:=false;             (*exit*)
       noncritic1
     FOREVER
END;
PROCESS   P2;
BEGIN
   REPEAT
```

```
     flag2:=true;                (*announce intention*)
     WHILE flag1 DO              (*check on flag1*)
        IF turn =1 THEN          (*if no key   *)
           BEGIN
             flag2:=false;        (*give up right*)
             WHILE turn=1 DO NULL;(*busy wait*)
             flag2:=true          (reclaim right*)
            END;
       critic2;
       turn:=1;                   (*return the key*)
       flag2:=false;              (*exit*)
       noncritic2
     FOREVER
END;
BEGIN (*main program*)
   flag1:=false;
   flag2:=false;
   turn:=1;                       (*can be 2 as well*)
   COBEGIN P1;P2 COEND
END.
```

3.5 Correctness of concurrent programs

As you have seen, when we write a program, we always start with a problem description followed by the requirement(s) of the program. Then we try to reach a solution. And after we get a solution, we need to analyse whether the program meets the requirements.

We say a **property** is some assertion on the program execution: i.e. an assertion on all possible interleavings that the program may exhibit. A program **satisfies** (or **has**) a property iff the property is *true* of *every* interleaving (execution) of the program.

A **requirement** of a program is just stated as a property. A program is **correct** with respect to a requirement if the program satisfies the requirement property. For example, the mutual exclusion requirement and the liveness requirement are properties that every execution of the program Dekker satisfies.

In general, (useful) properties of programs are classified (by Owicki & Lamport, 1982) as **safety** properties and **liveness** properties:

- A *safety property* states that nothing *bad* will ever happen during an execution.

  More precisely, a safety property asserts that the program execution will never enter a *bad state*. A bad state can be a state, for examples, in which two or more processes are in their critical sections, or a state at which the program deadlocks.

- A *liveness property* states that something *good* will *eventually* happen duration an execution.

Again in other words, a liveness property asserts that an execution will eventually enter an *expected state*. Such an expected state can be a state in which a process is in its critical section, or a state in which the execution outputs the result, or a state in which the execution normally terminates.

It is noted that a liveness property is often under a condition. Thus, in full a liveness property states that "if a property (condition) holds in an execution, then something good will eventually happen in the execution". For example, if a process requests infinitely often for a resource, it will eventually get the resource infinitely many times.

A program violating safety property is dangerous to use: a banking system should never allow a customer to enter another's account. In this sense, safety properties are treated more seriously.

On the other hand, a program without liveness property can be useless. For example, a program terminates without doing anything satisfies any safety property.

Program Verification **Program verification** is a procedure of reasoning about a program against its requirements. For example, we claimed the program Dekker is correct w.r.t. the mutual exclusion and liveness requirements. Can we prove this?

Without a formal notation, it would be very complicated to prove that Dekker's solution is correct. At most we can only give some informal explanation. It would be very confusing and ambiguous. We rather not enter this topic in this course.

Formal methods in specification and verification have been developed for tackling this problem effectively.

3.6 Exercises

1. In Pascal-FC, $clock$ is a function of no argument, which returns an integer that represents the number of system clock units (seconds) elapsed since some arbitrary zero (not necessarily the start of the execution of the concurrent program), and $sleep$ is a procedure whose call is of form $sleep(n)$, where n is an integer constant, that causes the calling process delayed for n seconds. Read Pascal-FC User Guide for Sun Systems and consider the following program.

```
PROGRAM sleeper;
  CONST period =4;
  VAR   start: integer;

  PROCESS A;
     VAR time: integer;
  BEGIN
     REPEAT
       time:= clock;
       IF (time-start) MOD 4 < 2 THEN
         writeln('Uninterruptable write by A');
```

```
      sleep(1)
    FOREVER
END;


PROCESS B;
    VAR time: integer;
BEGIN
    REPEAT
      time:= clock;
      IF (time-start) MOD 4 >= 2 THEN
        writeln('Uninterruptable write by B');
      sleep(1)
    FOREVER
END;

BEGIN
  start:=clock;
  COBEGIN
     A;B
  COEND
END.
```

Are the **writeln** statements uninterruptable as the program claims?

2. With the program in Exercise 1, is the fair scheduler actually fair? What is the output of the program likely to look like with the unfair scheduler? If the two **sleep** statements are removed (or replaced by **null**), what will the behaviour of the program be with the fair and unfair schedulers?

3. Assume that the **clock** and **sleep** statements work on a one-second time constant. Write a program that consists of five processes. Each process is periodic, but the periods differ. One process should run every 2 seconds, one every 3, one every 5, one every 7 and the last every 9 second. Use the unfair scheduler. Each process should simulate an intensive computational load by executing:

```
FOR i:=1 to MAX do
    temp:=163*10 MOD 975
```

where $temp$ is a local variable (to each process) and MAX is a global constant. Experiment with different values of MAX. When is the computational load too excessive for each process to complete its executions before its next period starts? (Hint: see pages 54-55 of Burns& Davies book.)

4. We said that the update of a variable would be accomplished on most current hardware by three separate operation: load the value of the variable to a register, update the value in the register, and store the value of the register at the address of the variable. How many possible interleavings are there when two processes currently execute the update on a uniprocessor? How many of them lead to the 'loss' of an increment?

5. Given that the Pascal-FC compiler generates these three instructions when the variable $count$ is updated by a process in the program **garden1** for the Ornamental Gardens problem, what is the

minimum value that you would expect $count$ to have when the program completes? What is the maximum for $count$?

6. The following program is a solution originally proposed by Peterson (1981) to the mutual exclusion problem for two processes.

```
PROGRAM peterson; VAR turn: integer;
    flag1, flag2: boolean;
PROCESS P1;
BEGIN
  REPEAT
     flag1:=true;  (*announce intent to enter*)
     turn:=2;      (*give priority to other process*)
     WHILE flag2 AND (turn=2) DO null;
     critic1;
     flag1:=false;
     noncritic1
  FOREVER
END;

PROCESS P2;
BEGIN
  REPEAT
     flag2:=true;  (*announce intent to enter*)
     turn:=1;      (*give priority to other process*)
     WHILE flag1 AND (turn=1) DO null;
     critic2;
     flag2:=false;
     noncritic2
  FOREVER
END;

BEGIN(*main program*)
  flag1:=false; flag2:=false;
  turn:=1;     (*arbitrary value, could be 2*)
  COBEGIN
    P1; P2
  COEND
END.
```

Show that program peterson meets the both safety and liveness requirements.

7. Adapt Peterson's algorithm so that three processes wish to gain access to the critical sections.

8. The following attempts to show how two processes can pass data (without blocking) using two slots for the data:

```
PROGRAM twoslots;
  TYPE data=RECORD
       A: integer;
       B: integer;
```

```
     C: integer
  END;

  VAR twoslot: ARRAY[false..true] OF data;
      slot: boolean;

  PROCESS writer;
      VAR I:integer;
          D:data;
  BEGIN
      FOR I:=1 TO 30 DO
          BEGIN
             twoslot[slot].A:=I;
             twoslot[slot].B:=I;
             twoslot[slot].C:=I;
             slot:=NOT slot
          END
   END;

   PROCESS reader;
      VAR I: integer;
          D: data;
   BEGIN
      FOR I:=1 TO 40 DO
         BEGIN
          write(twoslot[NOT slot].A);
          write(twoslot[NOT slot].B);
          write(twoslot[NOT slot].C);
          writeln
         END
    END;

BEGIN (*main program*)
  slot:=true;
  twoslot[false].A:=0;
  twoslot[false].B:=0;
  twoslot[false].C:=0;
  COBEGIN
    writer;
    reader
  COEND
END.
```

The data is coded as a record so that atomic updates are clearly impossible. Boolean variables can, however, be assumed to be updated atomically. Does this program possess safety and liveness?

9. The following complete Pascal-FC program implements Simpson's 4-slot algorithm for information exchange between a single reader process and a single writer process. Neither process busy waits; instead the reader process will get immediate access to an old version if the writer process is currently updating the 'shared' data. Four slots for the data are needed so that no interleaving will break mutual exclusion on any copy of the data:

```
PROGRAM simpson; TYPE data=RECORD
      A: integer;
      B: integer;
      C: integer
 END;

 VAR fourslot: ARRAY[false..true,false..true] OF data;
     slot: ARRAY[false..true] OF boolean;
     reading, latest: boolean;

 PROCESS writer;
     VAR I:integer;
         D:data;
         pair, index: boolean;
 BEGIN
     FOR I:=1 TO 30 DO
         BEGIN
            pair:= NOT reading;
            index:= NOT slot[pair];
            fourslot[pair,index].A:=I;
            fourslot[pair,index].B:=I;
            fourslot[pair,index].C:=I;
            slot[pair]:=index;
            latest:=pair
         END
  END;

  PROCESS reader;
     VAR I: integer;
         D: data;
         pair, index: boolean
  BEGIN
     FOR I:=1 TO 40 DO
        BEGIN
         pair:=latest;
         reading:=pair;
         index:=slot[pair];
         write(fourslot[pair,index].A);
         write(fourslot[pair,index].B);
         write(fourslot[pair,index].C);
         writeln
        END
   END;

BEGIN (*main program*)
  reading:=false;
  latest:=false;
slot[false]:=false;
  fourslot[false,false].A:=0;
  fourslot[false,false].B:=0;
  fourslot[false,false].C:=0;
```

```
  COBEGIN
    writer;
    reader
  COEND
END.
```

Study the program. Does it possess safety? Can you find two liveness properties that this program meets? Can you find a liveness property that this program does not meet?

10. The following program is an attempted solution to the mutual exclusion problem for two processes. Discuss the correctness of the solution: if it is correct, then prove it. If not, write scenarios that show that the solution is incorrect.

```
PROGRAM attempt; VAR   c1,c2: integer; PROCESS
p1; BEGIN
   REPEAT
     noncritic1;
     REPEAT
       c1:=1-c2
     UNTIL c2<>0;
     critic1;
     c1:=1
    FOREVER
END;
PROCESS p2;
BEGIN
   REPEAT
     noncritic2;
     REPEAT
       c2:=1-c1
     UNTIL c1<>0;
     critic2;
     c2:=1
    FOREVER
END;

BEGIN  (*main program*)
  c1:=1; c2:=1;
  COBEGIN
     p1;p2
  COEND
END.
```

11. The IBM 360/370 computers have a primitive instruction called TST (Test an Set). There is a system global variable called c (condition Code). Executing $TST(l)$ for a local variable l is equivalent to the following two assignments:

```
l:=c;
c:=1
```

(a) Discuss the correctness (safety, deadlock, lockout) of the solution of mutual exclusion problem shown in the program $testandset$.

(b) Generalize to n processes.

(c) What would happen if the primitive TST instruction were replaced by the two assignments?

```
 PROGRAM testandset; VAR c:integer; PROCESS
p1; VAR l: integer; BEGIN
  REPEAT
    noncritic1;
    REPEAT
       TST(l)
    UNTIL l=0;
    critic1;
    c:=0
  FOREVER
END;
PROCESS p2;
VAR l: integer;
BEGIN
  REPEAT
    noncritic2;
    REPEAT
       TST(l)
    UNTIL l=0;
    critic2;
    c:=0
  FOREVER
END;
BEGIN   (*main program*)
   c:=0;
   COBEGIN
     p1;p2
   COEND
END.
```

12. The EX instruction exchanges the contents of two memory locations. $EX(a, b)$ is equivalent to an *indivisible* execution of the following assignment statements:

```
temp:=a;
a:=b;
b:=temp
```

(a) Discuss the correctness (safety, deadlock, lockout) of the solution for the mutual exclusion shown in program $exchange$.

(b) Generalize to n processes.

(c) What would happen if the primitive EX instruction were replaced by the three assignments?

```
PROGRAM exchange; VAR  c: integer; PROCESS
p1; VAR  l: integer; BEGIN
   l:=0;
```

```
   REPEAT
     noncritc1;
     REPEAT
       EX(c,l)
     UNTIL l=1;
     critic1;
     EX(c,l)
   FOREVER
END;
PROCESS p2;
VAR  l: integer;
BEGIN
   l:=0;
   REPEAT
     noncritc2;
     REPEAT
       EX(c,l)
     UNTIL l=1;
     critic2;
     EX(c,l)
   FOREVER
END;
BEGIN  (*main program*)
  c:=1;
  COBEGIN p1;p2 COEND
END.
```

4 SEMAPHORES

The goals of the chapter

- Introduce the synchronization primitives called *semaphores*.
- Apply semaphores to mutual exclusion and condition synchronization.

4.1 Introducing Semaphores

As we have seen in Chapter 2, the outline of the solutions to mutual exclusion is of the form:

```
entry-protocol
critical section
exit-protocol
non-critical section
```

We have shown that the problem can be solved by directly programming the **entry** and **exit** synchronization protocols, without using new language facilities. In general, these protocols are sequences of statements whose executions can overlap in time.

However, synchronization protocols that use only busy waiting can be difficult to design, understand, and reason about. We saw in the last chapter, most busy-waiting protocols are quite complex. Also there is no clear separation between variables that are used for synchronization and those that are used for computing results. A consequence of these attributes is that one has to be very careful to ensure that processes are correctly synchronized. For example the order of setting one's own flag and checking another's is crucial and needs serious consideration. This implies that such an old-fashioned approach is very much error-prone.

A further deficiency of busy waiting is that it is inefficient when processes are implemented by multiprogramming (multitasking). A processor executing a spinning process can usually be more productively employed executing another process.

From Chapter 1, we should have had the feeling that all the **entry**/**exit** protocols have a common feature: a process's entry protocol forces the process to **wait** to enter its critical section until no other processes are in the their critical sections; a process's exit protocol gives a **signal** to all the relevant processes about its exit from the critical section. Because synchronization is fundamental to concurrent programming, it is desirable to have special tools that aid in the design of synchronization protocols and that can be used to block processes that must be delayed.

Would it not be much nicer if we can always represent the entry and exit protocols by single statements?

```
WAIT;
critical section;
SIGNAL;
non-critical section
```

Semaphores are one of the first such tools and certainly one of the most important to provide such an abstraction. The introduction of semaphores by Dijkstra (1968) gave a decisive thrust to the scientific study of concurrent programming.

4.2 Definition of semaphores

A **semaphore** is represented as a program *variable* which can take on only non-negative integers.

Then what is the **data type** of a semaphore variable? Can it be **Integer**? To answer the questions, we must first understand what is a data type.

For computer scientists, a data type has *two* attributes:

- a set of **permissible values**,
- a set of **permissible operations** on the values.

For example, the type BOOLEAN has only two values, true and false, and has the boolean operations such as **and**, **or** and **not**. We are not allowed to assign an integer, such as 5, to a boolean variable, or apply the operation '+' on boolean values.

Therefore, a semaphore *cannot* be of type **integer**. The first reason is that it does not take on all integers. The second reason it does not allow the conventional integer operations.

Pascal-FC provides a new *standard* type, **semaphore** which has:

- the set of non-negative integers as its permissible values,
- two priciple permissible operations **wait** and **signal** (the original Dijkstra's notation is P for **wait** and V for **signal** which are the initials of the corresponding words in Dutch).

Given a semaphore s, meanings of these two definitions are given as follows:

- wait(s): **If** $s > 0$ **then** $s := s - 1$ **else** the execution of the process calling wait(s) is suspended (blocked on s).
- signal(s): **If** some processes have been blocked by a previous wait(s) on s **then** unblock one of them **else** $s := s + 1$.

Remarks on semaphores

- *Remark 1*: If a semaphore only assumes values 0 and 1 in a program, it is called a **binary semaphore**, otherwise it is called a **general** (or **counting**) semaphore.
- *Remark 2*: wait(s) and signal(s) are the only operations allowed. In particular, assignment to s or tests on s are prohibited except for an assignment to s of an initial non-negative value in the *main* program by the procedure **initial**(s,n), where n is a non-negative integer.
- *Remark 3*: The standard procedures wait(s) and signal(s) are implemented as single indivisible operation. This means that they exclude one another just as Load and Store operations exclude one another. But semaphore operations on *distinct* semaphores need not exclude one another.
- *Remark 4*: The definitions of wait(s) and signal(s) here uses *block-waiting* rather than busy-waiting, a blocked process was taken off from its processor by the scheduler, so that the processor can be switched to another process.

Question: Can you give the busy-waiting definitions for wait(s) and signal(s)?

Declarations of semaphores

Since a semaphore is represented as variable of type SEMAPHORE, it must be declared before being used. Pascal-FC declarations of semaphore variables are as for any other Pascal's unstructured data types. We can have the following declarations:

```
TYPE
  semtable=ARRAY[1..10] OF semaphore;

VAR
   s1, s2: semaphore;
   stab:   semtable;
   semarray: ARRAY[1..20] OF semaphore;
   semrc:
        RECORD
           i: integer;
           s: semaphore
         END;
```

But we have the following two restrictions:

- a semaphore may only be declared in the main program declaration part;
- they may be parameters of processes, procedures or functions, but they must be always be *formally* declared as **var** parameters.

Question: Why these restrictions are needed?

4.3 Mutual exclusion with semaphores

Solving the two-process mutual exclusion problem is trivially easy now:

```
 PROGRAM two-process-mutual-exclusion; VAR
s: semaphore; PROCESS P1; BEGIN
  REPEAT
     wait(s);
     critic1;
     signal(s);
     noncritic1
   FOREVER
END;
PROCESS P2;
BEGIN
   REPEAT
      wait(s);
      critics2;
      signal(s);
      noncritic2
    FOREVER
END;
BEGIN (*main program*)
  initial(s,1);
  COBEGIN P1; P2 COEND
END.
```

You can easily imagine that we can add as many processes as we want to the program and the program will still guarantee the mutual exclusion. Let us ignore the differences in the the critical sections and noncritical sections among different processes. We have have a program like

```
 PROGRAM mutualexclusion; PROCESS TYPE
proc; BEGIN
  REPEAT
    wait(s);
    critic;
    signal(s);
    noncritic
  FOREVER
END;
VAR s: semaphore;
    P: ARRAY[1..N] OF proc;
CONST: N= 50; (*for example*)
BEGIN
  initial(s,1);
  COBEGIN
    FOR i:=1 to N DO P[1]
```

```
  COEND
END.
```

About $initial(s, n)$

Notice that in *this case* we have used $initial(s, 1)$ which initialized s with value 1. Can we chose another $n \geq 0$, say 0, or 2, or 1000, as the initial value of s? The answer is NO!

Consider the case that we have three processes P1, P2 and P3, and the initial value of s is 0:

```
                  s     P1        P2       P3
Initially         0    begin     begin    begin
P1 executes wait  0    blocked   begin    begin
P2 executes wait  0    blocked   blocked  begin
P3 executes wait  0    blocked   blocked  blocked
         ...... Deadlock.......
```

If the initial value is greater than 1, say 2:

```
                  s     P1        P2       P3
Initially         2    begin     begin    begin
P1 executes wait  1    critic    begin    begin
P2 executes wait  0    critic    critic    begin

         ......mutual exclusion violated .......
```

Does this mean that we must always initialize the semaphores with 1? If so, an implementation of semaphores could automatically assign 1 to all semaphores in their declarations and there would have no need to provide this initialization primitive. In some other applications, 1 turned out not to be an appropriate initial value.

4.4 Implementation of semaphores

The definitions of **wait** and **signal** operations do not specify which of the blocked processes is unlocked. This is a matter taken care by the scheduler. There are several scheduling policies in practice for the implementation of semaphores, which have different effect on the liveness properties of a program. Here we consider three of them.

- FIFO. Any process attempting a **wait** on a semaphore s with a value of 0 is blocked on a FIFO queue. When a **signal** is executed on s (by an unblocked process), the first process in the queue (i.e. at the head of the queue) is unblocked (allowed to complete its **wait**).

- PRIORITY. Each process is associated (either statically or dynamically) with a *priority* which is usually an integer. Any process attempting a **wait** on s with a value of 0 is blocked and joins a set ordered by according to the priorities of processes. When a **signal** is executed on s, the blocked process with the highest priority in the set will be unblocked.
- RANDOM. Any process that attempts a **wait** on s with a value of 0 is blocked and joins an unordered set of processes. When a **signal** is executed on s, one member of the set is chosen at random and completes its **wait**.

The FIFO scheduling is a **fair** in the sense that *a blocked process will eventually be unblocked*. And it is even more than fair that the process which is blocked earlier will be unblocked earlier. The priority-based scheduling is certainly not fair because "lower class" processes have to give way to "higher class" processes. The random scheduling is sometimes said **weak fair** (but I personally do not) in the sense that if a process stays in the blocked set long enough it will be unblocked (with probability 1). But note that it is possible that an event with probability 1 may never happen. So in both of the last two scheduling policies, starvation of some processes may be allowed, though this is *unlikely* to happen for the random scheduling.

4.5 Analysing semaphore programs

In analysing the behaviour of semaphore programs, it is important to know the following two properties called *semaphore invariants* because they are satisfied at all times during the execution of a program.

- (1). $s \geq 0$: at any time the value of s is no less than 0.
- (2). $s = s_0 + \#siginal - \#wait$: at any time the value of s equals the summation of the initial value s_0 and the number of **signal** operations $\#signal$ carried out up to the time **minus** the number of *completed* **wait** operations $\#wait$ up to the time. In other words, the value of a semaphore always equals to the summation of its initial value and the number of increments (by **signal** operations) minus the number of decrements (carried out by **wait** operations).

Note the fact that a **signal** operation may not actually increment s to unblock a blocked process, but in this case the unblocked process *completes* its **wait** without decrementing s either.

Analysis of program *mutualexclution*

Using the two semaphore invariants, we can prove the mutual exclusion property of program $mutualexclution$: the number $\#CS$ of processes in their critical sections is at all times no more than 1 during the execution:

$$\textbf{Invariant}: \quad \#CS \leq 1$$

The proof outline is:

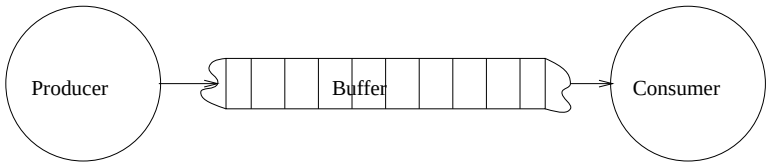


Figure 8: Producer and consumer–unbounded buffer

i). $\#CS = \#wait(s) - \#signal(s)$: this can be shown by examination of the program text (can you prove it as an exercise?);

ii). $s = 1 + \#signal(s) - \#wait(s)$: this is semaphore invariant (2) since s is initialized as 1;

iii). $\#CS = 1 - s \leq 1$: this is from (i) and (ii) and the semaphore invariant (1).

The liveness requirement depends on the semaphore implementation. It is satisfied by the FIFO scheduling policy. In general, it is difficult to rigourously reason about a liveness property without using a formal notation, such a Temporal Logic.

4.6 Condition synchronization with semaphores

We consider a number of examples to show further applications of semaphores.

4.6.1 Producer and consumer with an unbounded buffer

Problem description
Consider two processes, a *producer* which repeatedly *produces* items, and a *consumer* which *consumes* the produced items one-by-one. The producer sends its produced items to the consumer by *placing* them into a unbounded FIFO buffer, and the consumer receives these items by *taking* them from the FIFO buffer (see Figure 8).

We assume that we have an implementation of an unbounded buffer with two operations **place** and **take**, which guarantees a FIFO discipline on buffer accesses.

Requirements

1. the producer may produce a new item at any time;
2. the consumer may only consume an item when the buffer is not empty.

3. all items produced are eventually consumed —liveness requirement.

To meet the requirements, the consumer must be blocked when the buffer is empty. We propose to use a semaphore whose value records the current number of items in the buffer: the number of items that have been produced and have not yet been consumed. The initial value of the semaphore should be 0 since nothing has been produced at the beginning. After the producer executes a **place**, it should signal the semaphore.

```
PROGRAM producer-consumer1; VAR
ItemsReady: semaphore; PROCESS producer; VAR item: sometype; BEGIN
   REPEAT
      produce(item); (*may be a sequence of statements*)
      place(item);   (*place the newly produced item *
                     (*into the buffer*)
      signal(ItemsReady);
   FOREVER
END;
PROCESS consumer;
VAR item: sometype;
BEGIN
   REPEAT
     wait(ItemsREady);  (*blocked if buffer is empty*)
     take(item);        (*take an item from the buffer*)
     consume(item)      (*may be a sequence of statements*)
   FOREVER
END;

BEGIN
   initial(ItemsReady,0);
   COBEGIN
     producer; consumer
   COEND
END.
```

This program meets the three requirements that we gave for the problem. And it has the property that the order in which items are consumed is the same of the order in which they are produced because of the use of the FIFO buffer.

Question: *Can you give a proof of requirement 2 of the problem in the similar way as we did for the mutual exclusion program*?

However, the above program is not entirely satisfactory. The problem is that when the buffer is *not* empty, there is nothing to stop overlapping of a **place** and a **take**. We might have assumed that these two operations could be safely overlapped, but it is not always necessarily so. It depends on how the buffer has been implemented, e.g. on how the bits are shift from one side to another. Moreover, if we consider a general case in which there are multiple producers and consumers sharing the same buffer, then overlapping buffer operations would be undesirable.

Now we introduce another semaphore enforce the mutual exclusion on the buffer operations. This can be easily done following the mutual exclusion solution.

```
PROGRAM producer-consumer2; VAR
ItemsReady, MutEx: semaphore; PROCESS producer; VAR item:
sometype; BEGIN
  REPEAT
    produce(item);
    wait(MutEx);
    place(item);  (*a critical section*)
    signal(MutEx);
    signal(ItemsReady)
  FOREVER
END;
PROCESS consumer;
VAR item: sometype;
BEGIN
  REPEAT
    wait(ItemsReady);
    wait(MutEx);
    take(item);    (*a critical section*)
    signal(MutEx);
    consume(item)
  FOREVER
END;
BEGIN(*main program*)
  initial(ItemsReady,0);
  initial(MutEx,1);
  COBEGIN
    producer; consumer
  COEND
END.
```

Note that $ItemsReady$ is a general semaphore which may take on any negative integers.

4.6.2 Producer-consumer with bounded buffers

In practice, there is no implementation of an unbounded buffer. We now consider the use of a buffer of finite capacity. The requirement 1) for the producer-consumer with an unbounded buffer has to be changed as

- the producer must never attempt to **place** an item into the buffer when the buffer is full.

All the other requirements for the unbounded buffer case remain as requirements for this case.

We solve this problem by using another (in addition to $ItemsReady$ and $MutEx$) semaphore $SpacesLeft$ which records the number of spaces left in the buffer. The producer must be blocked whenever there is not space left, i.e. whenever the value of this semaphore is 0.

We give a full executable Pascal-FC program in which the buffer is a record, and **place** and **take** are two procedures.

```
PROGRAM producerconsumer3;  (*with a
bounded buffer*) CONST
   BuffSize=5;              (*for example*)
   BuffInxMax=4;            (*BufferSize-1*)
VAR
   MutEx: semaphore;        (*binary*)
   ItemsReady: semaphore;   (*general*)
   SpacesLeft:  semaphore;   (*general*)
   Buffer: RECORD
            products: ARRAY[0..BuffInxMax] OF char;
            NextIn:   integer;
            NextOut:  integer;
           END;
PROCEDURE place(ch:char);
BEGIN
  Buffer.products[Buffer.NextIn]:=ch;
  Buffer.NextIn:= (Buffer.NextIn+1) MOD BuffSize
END; (*place*)
PROCEDURE take(VAR ch:char);
BEGIN
   ch:=Buffer.products[Buffer.NextOut];
   Buffer.NextOut:=(Buffer.NextOut+1) MOD BuffSize
END;  (*take*)
PROCESS producer;
VAR
  item: char;
BEGIN
  FOR item:='a' TO 'z' DO
      BEGIN
        wait(SpacesLeft);
        wait(MutEx);
        place(item);
        signal(MutEx);
        signal(ItemsReady)
      END
END;  (*producer*)
PROCESS consumer;
VAR
  item: char;            (*local variable*)
BEGIN
  REPEAT
     wait(ItemsReady);
```

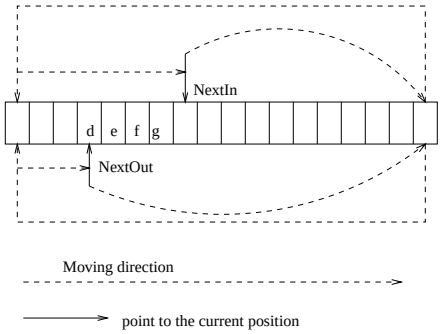


Figure 9: Behaviour of the buffer

```
     wait(MutEx);
     take(item);
     signal(MutEx);
     Signal(SpacesLeft);
     write(item)
  UNTIL item='z';
  writeln
END;   (*consumer*)

BEGIN
  Buffer.NextIn:=0;   (*it is safe to initialize  *)
  Buffer.NextOut:=0;  (*these two variables    *)
  initial(SpacesLeft, BuffSize);
  initial(ItemsReady,0);
  initial(MutEx,1);
  COBEGIN
     producer;
     consumer
  COEND
END.
```

Figure 9 can hopefully help the illustration of the behaviour of the buffer, especially the **place** and **take** procedures.

NB:

- In general a global variable such as $Buffer.NextIn$ and $Buffer.NextOut$ should be initialized. But it is mysterious (to me) the above program works fine without initializing $Buffer.NextIn$ and $Buffer.NextOut$.
- It is always safe to initialized the global variables in the main program. But they must be initialized correctly.

QUESTIONS:

1. Can we swap the order of the two signal operations in each processes?
2. Would it be correct if we initialized $Buffer.NextIn$ and $Buffer.NextOut$ to 3?
3. Would it be correct if we initialized these two variables with different values, say one with 0 and the other with 2? What would happen with the program executions if in this case?
4. Try the above questions by running the corresponding programs.

4.7 More examples

4.7.1 The Readers and Writers Problem

A generalization of the mutual exclusion problem is the problem of the readers and writers. The prototype for the abstract problem is an on-line transaction system such as a banking system which does not require mutual exclusion among several processes which *only read* the data. However, an update or any operation that *writes* data must be considered to be a critical section which should be carried out under mutual exclusion with all other operations on the data (including reading operations). We abstract our problem as follows.

Problem description: Consider a number of processes accessing to some shared data. Any process requiring access to the data must first call a procedure **open**, indicating whether the access is for reading or writing. When the process finishes its access, it must call a procedure **close**, again indicating whether it had been reading or writing. We ignore the details of how the data are given their initial values.

Requirement: Any number of readers can be concurrently active while write accesses to the data must exclude *all* other accesses.

So we have two types of processes: readers and writers, whose life cycles are as follows.

```
PROCESS TYPE reader; VAR
  local: SomeType;
BEGIN
  REPEAT
     open(ReadAccess);
     rs_1;                (*access data, copying to local*)
     close(ReadAccess);
     rs_2                 (*use local data*);
  FOREVER
END; (*reader*)

PROCESS TYPE writer;
```

```
VAR
  local: SomeType;
BEGIN
  REPEAT
     ws_1;              (*produce new local data*);
     open(WriteAccess);
     ws_2;              (*write local values to data*);
     close(WriteAccess)
  FOREVER
END; (*writer*)
```

To meet the requirement, we make the following design decisions for **open** and **close**:

1. *Any* process attempting an access to the data has to **wait** when there is writer currently accessing the data. So we need a semaphore $writing$ whose value should be 0 when there is such a writer.
2. A reading process does not have to **wait** at $writing$ if there is already a reader is currently reading the data.
3. A writing process attempting an access to the data has to wait at $writing$ when there is *any* process is currently accessing the data. So $writing$ can be **signalled** on by a process which is the only one currently accessing the data.
4. Decisions 2&3 implies that we need to keep a record on the number of readers currently accessing to the data. Since this number is *commonly* updated by all the readers, we have to use another semaphore $MutEx$ to ensure the updating operations are carried out under mutual exclusion.
5. $writing$ and $MutEx$ must both have an initial value 1.

These decisions lead to the following solutions for the procedures **open** and **close** (Courtois et al, 1971):

```
 PROCEDURE open(readwrite: boolean); (*true
for read and false for write*) BEGIN
  IF readwrite THEN        (*if it is a reading access*)
     BEGIN
       wait(MutEx);        (*wait if another process  *)
                           (*is accessing nr*)
       nr:=nr +1;          (*update the shared variable*)
       IF nr =1 THEN       (*only the first reader has *)
          wait(writing);   (*wait for no writer condition*)
       signal(MutEx)       (*allow other readers to update nr*)
    END ELSE               (*if it is a writing access then*)
      wait(writing)        (*wait for no access condition*)
END; (*open*)

PROCEDURE close(readwrite:boolean);
BEGIN
```

```
  IF readwrite THEN        (*if it was a reading access*)
  BEGIN
     wait(MutEx);          (*wait if another process is *)
                           (*accessing nr*)
     nr:=nr-1;
     IF readers =0 THEN    (*if it is the only process *)
                           (*left in accessing to the data*)
        signal(writing);   (*allow a writer/reader to enter*)
     signal(MutEx)         (*allow another to access nrs*)
  END ELSE
     signal(writing)       (*allow a writer/reader to enter*)
END;
```

Characteristics of the program

Analysing the behaviour of the above procedures, we find:

- the procedures meets the safety requirement that a writing access operation excludes *all* other accesses;
- writers can be locked out indefinitely: as long as a reader is active, no writers can gain access, but other readers are allowed in.

A Reader-and-Writer program with these properties is called **readers' preference** protocols. In applications where there are frequent read accesses and it is important to make frequent updates, such a readers' preference feature becomes unsuitable. For this reason, Courtois *et al.* (1971) presented a **writers' preference** version for the Readers-and-Writers problem, in which incoming readers become blocked when a writer is waiting to access the data. This thus guarantees that the number of active readers will eventually decreased to 0 and a blocked writer will then be unblocked. The solution uses five binary semaphores, and two shared variables nr and nw which record the numbers of currently *active* readers and currently *blocked/active* writers respectively. These variables are declared as:

```
VAR
  nr, nw: integer;  (*initially 0*)
  MutEx1,   (*for mutual exclusion of accesses to nr*)
  MUtEx2,    (*for mutual exclusion of accesses to nw*)
  r,        (*for blocking the first reader when *)
            (* there is a blocked writer)
  Mutex3,   (* for blocking the second and subsequent readers *)
            (* when there is a blocked reader*)
  writing: semaphore ;
  (*for blocking all accesses to the data if there is *)
  (*an active writer*)
```

All the five semaphores must have an initial value 1.

The **open** and **close** procedures are:

```
PROCEDURE open(readwrite:boolean); BEGIN
IF readwrite THEN  (*if it is a read access*) BEGIN
  wait(MutEx3);  (*wait if there is already a reader *)
                 (*blocked because of a blocked a reader*)
  wait(r);       (*wait if there is a writer active/blocked*)
  wait(MutEx1);  (*wait if nr is being accessed*)
  nr:=nr+1;      (*one more reader attempting*)
  IF nr =1 THEN  (*if no reader is already active*)
  wait(writing); (*wait if a writer is active*)
  signal(MutEx1);(*allow another reader to access nr*)
  signal(r);     (*unblock one which is blocked on r*)
  signal(MutEx3) (*unblock one which is blocked on MutEx3*)
END ELSE         (*if it is write access then*)
BEGIN
  wait(MutEx2);  (*wait if another writer access nw*)
  nw:=nw+1;      (*one more writer attempting*)
  IF nw=1 THEN   (*if no writer is blocked*)
  wait(r);       (*a first incoming writer has equal right*)
                 (*as the incoming reader*)
  signal(MutEx2);(*allow another writer to access nw*)
  wait(writing)  (*wait if there is an active reader/writer*)
END
END;


PROCEDURE close(readwrite: boolean);
BEGIN
IF readwrite THEN (*if it was a read access*)
BEGIN
  wait(MutEx1);    (*wait if nr is currently accessed*)
  nr:=nr-1;       (*number of active/blocked readers reduced by one*)
  IF nr=0 THEN     (*If this was the last reader accessing the data*)
    signal(writing);(*Allow a reader/writer to enter*)
  signal(MutEx1);   (*Allow another access to nr*)
END ELSE            (*If it was a write access*)
BEGIN
  signal(writing);  (*allow another process to enter*)
  wait(MutEx2);     (*wait if nw is being accessed*)
  nw:=nw-1;         (*the No. of active/blocked writers reduced by one*)
  IF nw=0 THEN      (*If this is the only one being active/blocked*)
     signal(r);     (*allow a reader/writer to enter*)
  signal(MutEx2)    (*allow another writer to access nw*)
END
END;
```

We can see that though semaphore primitives have made the life of concurrent programmers much easier, we still face the challenge and difficulties in both understanding semaphores and using them in our design

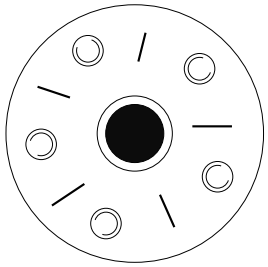


Figure 10: Philosophers dining table

tasks.

Questions

1. Do we have to use semaphore $MutEx3$ in our above procedures?
2. Can you illustrate why an attempting writer will eventually gain access to the data?
3. Can a reader be blocked on semaphore $writing$?
4. Can a reader be blocked indefinitely?

4.7.2 The Dining Philosophers problem

The problem of the dining philosophers (originally proposed by Dijkstra) is of great importance in concurrent programming research. The problem allows all of the pitfalls of concurrent programming to be demonstrated in a vividly graphical situation (as we will see in later chapters).

Problem description: In a monastery, there are five Chinese monks who are dedicated philosophers. Each philosopher would be happy to engage only in *thinking* were it not occasionally necessary to eat. Thus the life of a philosopher is an endless cycle: **repeat** think; eat **forever**.

The communal dining arrangement is shown in Figure 10. In the center of the table there is a bow of rices that is endlessly replenished; there are five plates and five chopsticks. A philosopher wishing to eat enters the dining room, takes a seat (which reserved for him), eats and then returns to his cell to think. However, it is hopeless (even for Chinese philosophers) to get any rice by only using one chopstick. Philosophers are too polite to reach across the table and pick up a spare chopstick or to eat with their hands.

Requirement:

- **Safety requirement:** A philosopher cannot be eating concurrently with his neighbour.
- **Liveness requirement:** A hungry philosopher will eventually eat assuming that a eating philosopher will be eventually full and go back his cell to think.

We use a binary semaphore to represent a chopstick: the chopstick is free if the semaphore is 1, it is not free if the semaphore is 0. So the life cycle of philosopher (represented as a process) should be:

```
BEGIN
  REPEAT
    think;
    wait(LeftChop);
    wait(RightChop);
    eat;
    signal(LeftChop);
    signal(RightChop)
  FOREVER
END;
```

We simulate the activities of thinking and eating by the process calling **sleep** procedure. The **sleep** procedure in Pascal-FC is provided mainly for timing features in real-time systems. The call of the procedure is of the form: **sleep(n)**, where **n** can be any integer. The meaning of this call is:

- When $n > 0$, the calling process is blocked (delayed) for n seconds of time.
- When $n \leq 0$, the calling process can be considered to make an instantaneous transition from ”executable” to ”blocked” and back to ”executable”.

We assume that a philosopher may think or eat for a random time between 0 to 10 seconds.

We can have the following program for the dining philosopher problem.

```
PROGRAM philsem1; (*Dining
philosophers with semaphores*) CONST
   N=5;          (*5 philosophers*)
VAR
   chopsticks: ARRAY[1..N] OF semaphore;
   I: integer;
PROCESS TYPE philosophers(name:integer);
BEGIN
```

```
   REPEAT
     sleep(random(10));     (*thinking*)
     wait(chopsticks[name]);(*picking up the left chopstick*)
     wait(chopstick[(name MOD N)+1]); (*the right chopstick*)
     sleep(random(10));      (*eating*)
     signal(chopsticks[name]); (*release the left chopstick*)
     signal(chopsticks[(name MOD N)+1]); (*release the right*)
    FOREVER
END; (*philosopher*)
VAR
   phils: ARRAY[1..N] OF philosophers;
BEGIN
   FOR I:=1 TO N DO
      initial(chopsticks[I],1);
   COBEGIN
      FOR I:=1 TO N DO phils[I](I)
    COEND
END.
```

This program meets the safety requirement for a solution. But it does not meet the liveness requirement because deadlock may occur. The following situation may happen: each philosopher has picked his left chopstick and waiting for his right one, but no one will release its left until he finishes eating.

```
                          chop1  chop2  chop3 chop4 chop5
Initially                   1      1      1     1     1
phils[1](1)                 0      1      1     1     1
phils[2](2)                 0      0      1     1     1
phils[3](3)                 0      0      0     1     1
phils[4](4)                 0      0      0     0     1
phils[5](5)                 0      0      0     0     0
.............Deadlock from here.....................
```

A deadlock free solution uses an additional semaphore (general) which ensures that no more than N-1 (4 in this case) philosophers are ever simulated seated at the table (to attempt picking up chopsticks). Then the circular pattern requests for chopsticks is broken since at least one philosopher can get his two chopsticks.

```
PROGRAM philsem2; (*Dining philosophers
with semaphores*) CONST
   N=5;          (*5 philosophers*)
VAR
   chopsticks: ARRAY[1..N] OF semaphore;  (*binary*)
   freechairs: semaphore;   (*general*)
   I: integer;
PROCESS TYPE philosophers(name:integer);
BEGIN
```

```
   REPEAT
     sleep(random(10));     (*thinking*)
     wait(freechairs);
     wait(chopsticks[name]);(*picking up the left chopstick*)
     wait(chopstick[(name MOD N)+1]); (*the right chopstick*)
     sleep(random(10));      (*eating*)
     signal(chopsticks[name]); (*release the left chopstick*)
     signal(chopsticks[(name MOD N)+1]); (*release the right*)
     signal(freechairs)
    FOREVER
END; (*philosopher*)
VAR
   phils: ARRAY[1..N] OF philosophers;
BEGIN
   FOR I:=1 TO N DO
      initial(chopsticks[I],1);
      initial(freechairs, N-1); (*there are only N-1 chairs*)
   COBEGIN
      FOR I:=1 TO N DO
          phils[I](I)
    COEND
END.
```

4.8 Final remarks on semaphores

1. In theory, we do not have to use general semaphores. A general semaphore can be simulated by using binary semaphores. But in practice, such a simulation would be very inefficient. See pages 183-184 in Burns&Davies book for further discussions.

2. Semaphore primitives are powerful enough for program all kinds of synchronizations in the shared variable framework.

3. Semaphore are easy to be implemented.

4. But semaphores are still at a too low level of abstraction, and, therefore they are difficult to use reliably. Missing a **signal**, for example, is likely to lead to deadlock. And missing a **wait** may lead to a violation of safety requirement, such as mutual exclusion. The more difficult issue is to put **wait** and **signal** operations in the correct place. Putting them in wrong places can be equally dangerous as missing them.

5. The next chapter will introduce primitives at a higher level of abstraction.

Further Reading: Burns&Davies book, pp171–199.

Are binary semaphores sufficient?

- Any general semaphore can be simulated by binary semaphores.

- This means that the **wait** and **signal** operations on a general semaphore can be simulated by operations on some binary semaphores.
- How can we define the simulation?

4.9 Exercises

1. Can a process be simultaneously blocked on more than one semaphore? Give reason for your answer.
2. If a semaphore has a non-zero value, is it possible for processes to be currently blocked on it?
3. Generalize the solution to the mutual exclusion problem so that the programmer can specify for k, the number of processes that can be simultaneously in their 'critical' section, any possitive integer.
4. In attempting to solve the Producer-Consumer problem (bounded buffer), suppose that the consumer process had been written as follows:

```
 PROCESS consumer; VAR
   item: sometype;
BEGIN
   REPEAT
     wait(MutEx);
     wait(ItemsReady);
     take(item);
     signal(MutEx);
     signal(SpacesLeft);
     consume(item)
    FOREVER
END;  (*consumer*)
```

   Would this then have been a satisfactory solution? What about the following?

```
 PROCESS consumer; VAR
   item: sometype;
BEGIN
   REPEAT
     wait(ItemsReady);
     wait(MutEx);
     take(item);
     signal(SpacesLeft);
     signal(MutEx);
     consume(item)
   FOREVER
END;  (*consumer*)
```

5. Write a scenario that shows that $signal$ must be a primitive instruction and not simply $s := s + 1$.

6. A set of processes have *precedence* relationships if there are restrictions on the order in which they can execute. Consider six processes. P1 and P2 must both run before P3; P3 runs before P4 and P5; and P6 runs after all other processes. Use semaphores to enforce these precedence relationships. Allow each process to be cyclic so that, for example, the first execution of P6 can be concurrent with the second execution of P1 and P2.

7. *The Unisex Bathroom.* Suppose there is one bathroom in your department. It can be used by both men and women, but not at the same time.

   (a) Develop a solution which allow any number of men or women to be in the bathroom at the same time. Your solution should ensure the required exclusion and avoid deadlock, but it need not to be fair.

   (b) Modify your answer to (a) so that at most four people are in the bathroom at the same time.

   (c) Modify your answer to (a) to ensure fairness (i.e. any person who wants to use the bathroom will eventual get it.)

8. In the lecture notes, we presented a solution to the Dinning Philosophers problem (**philsem2**) which prevented deadlock by removing the circular wait condition. Another possible approach is to remove the hold-and-wait condition, whereby processes hold on to resources already allocated to them while waiting for others. This can be done by allowing a philosopher to acquire both chopsticks, or none at all. Use semaphore to solve the problem using this strategy.

9. In the program **philsem2**, can any semaphore have more than one process simultaneously blocked on it?

10. Write a program that implements a lift (elevator) control system. A server process accepts calls on floor buttons and moves the lift to the requesting floor. The lift is very small and so can only take a single person at a time. In the lift are buttons that allow the passenger to choose the destination floor. The program should contain a number of passenger processes that make calls on the lift.

11. Modify the previous example so that there are now m lifts (each only carrying a single person).

5 CONDITIONAL CRITICAL REGIONS AND MONITORS

The goals of the chapter

- Understand the ideas of using *critical regions* for mutual exclusion.
- Understand the ideas of using general *conditional critical regions* for condition synchronization.
- Introduce the synchronization primitives called *monitors*.
- Apply monitors to mutual exclusion and condition synchronization.

5.1 Introduction

Semaphores are fundamental synchronization mechanisms. As shown in chapter 3, semaphores are elegant synchronization primitives that can be efficiently implemented. They can be used systematically to solve any synchronization problem. But they are at a rather low level of abstraction that leaves too much for the application programmer to do. The programmer must be careful not to omit or add a wait or signal operation accidently, not to employ the wrong semaphore, or to fail to protect all critical sections or to fail to meet a liveness property. Thus, semaphore algorithms can be difficult to design, understand and reason about.

A final problem with semaphores is that one programs both mutual exclusion and condition synchronization using the same pair of primitives. This makes it difficult to identify the purpose of a given wait or signal without looking at other operations on the same semaphore. Since mutual exclusion and condition synchronization are distinct concepts, they should ideally be programmed in different ways.

The consequences of these disadvantages are:

1. low programmer productivity,
2. non-reliable programs: a system built on semaphores alone is subject to disaster if
   - even one occurrence of a semaphore operation is omitted, or
   - one occurrence of semaphore operation is mistaken anywhere in the program.

These *motivate the need of a more structured tool* (i.e. primitives at a high level of abstraction). For example, a programming language with the provision of procedures and functions is more abstract and easier to use and understand than one without these facilities.

Beyond semaphores

Since semaphores have the above disadvantages that may cause various errors in programming, attempts were made to make those errors less likely by providing language constructs of high level abstraction, which allowed the compiler to partially automatically look after the necessary controls on access to shared data objects. This chapter will look at two such constructs: *conditional critical regions* (CCRs) (Brinch Hansen, 1972) and *monitors* (Hoare, 1974).

Pascal-FC provides an implementation of monitors, but not of CCRs. Therefore, the examples of CCRs here will not be executable programs.

5.2 Critical regions

Before looking at CCRs, we first look at a simpler and less powerful construct called the *critical regions* (CRs).

Similar to the *critical section*, a CR is a piece of code. However, while a critical section *should be executed* under mutual exclusion to get the program work correctly and this must be ensured by the programmer by proper use of semaphores, a CR *is always* executed under mutual exclusion, and this is ensured by the implementation of CRs.

Notation and semantics of CRs

A shared variable (resource) which should be accessed only under mutual exclusion *must* declared in such a way that the compiler can know this requirement. This can be done by the declaration:

```
VAR
  V: SHARED sometype
```

where V is the name of the variable of type $sometype$, the reserved word SHARED indicates that V must be accessed in a CR.

In some other books, the following declaration is used:

```
RESOURCE V: sometype;
```

A statement S which accesses the shared V should be written in a CR in the form

```
REGION V DO
   S
```

where S can refer to the shared variable V as well as other variables which can be accessed concurrently without danger of interference.

This construct provides a secure way of programming mutual exclusion, because:

- a shared variable V is declared such that it should be accessed in a CR tagged with the name V - the compiler can flag as an error any attempt to access it outside the CRs;
- all CRs tagged with the same variable name V are executed under mutual exclusion, but statements in CRs tagged with distinct variables can be executed concurrently,
- in effect, the **wait** and **signal** operations which would be required to protect a CS when using semaphores are automatically generated by the compiler, so that they cannot be overlooked.

An Example

For the ornamental gardens problem, we can easily have the following solution:

```
PROGRAM GARDENS; VAR
   count: SHARED integer;
PROCESS Turnstile1;
VAR loop:integer;
BEGIN
   FOR loop:=1 To 20 DO
      REGION count DO
         count:=count+1
END;
PROCESS Turnstile2;
VAR loop:integer;
BEGIN
   FOR loop:=1 To 20 DO
      REGION count DO
         count:=count+1
END;
BEGIN (* main program*)
  REGION count DO
       count:=0;
  COBEGIN Turnstile1; Turnstile2 COEND
END.
```

5.3 Conditional critical regions

CRs provide a more structured and securer way of implementing mutual exclusion than semaphores. However, they are not expressive enough to be as widely applicable as semaphores: CRs are not capable of simulating semaphores. They cannot solve the condition synchronization problem. Therefore, *conditional Critical regions* (CCRs) are introduced to meet such requirements.

Notation and semantics for CRs

In a language with CCRs:

- The declaration of a SHARED variable V may be done as for CRs.
- But the structured statement needs to be extended to allow for a condition to be expressed. The following is one possible form:

```
REGION V  WHEN B DO
      S
```

  Another possible form could be:

```
REGION V WHEN B => S
```

  Here B is a boolean expression (called the *guard* or the *condition* of the CCR). Both B and S can refer to V as well as variables local to the executing process.

- The semantics (i.e. the execution) of a CCR is as follows: execution of CCR for V delays (i.e. blocks) the executing process until B is true; then S is executed. The access to the same SHARED variable in different CCRs is mutually exclusive; in particular B is guaranteed to be true when execution of S begins.
- This semantics can be implemented as follows:
  - A process wishing to enter a CCR for V must obtain mutual exclusion lock on V. A queue can be used to implement blocked waiting.
  - Once the mutual exclusion lock has been obtained, the guard B can be evaluated. If it evaluates to true, the process can proceed to execute S under mutual exclusion. Otherwise it must release the mutual exclusion and become blocked. It cannot execute S until it has again obtained mutual exclusion on V and found B to be true.
  - A process that completes the execution of S must release its mutual exclusion on V.

Expressiveness of CCRs

Now we show that CCRs are as expressive as semaphores. We use a CCR to simulate a semaphore as follows: given a semaphore s

```
VAR
  s: SHARED integer;

  REGION s WHEN s>0 DO s:=s-1;    (*simulate wait(s)*)

  REGION s DO s:=s+1;             (*simulate signal(s)*)

  REGION s DO s:=n ;              (*simulate initial(s,n)*)
```

So CCRs are of no less expressive power than semaphores. Obviously, they have no more expressive power than semaphores.

Question: Can you simulate a CCR by using semaphores?

5.3.1 Example - condition synchronization

We apply CCRs to the producer-consumer problem:

```
PROGRAM PCCCR; CONST
  BuffSize = ...;
TYPE
  ITEM = ...;
  BUFFTYPE = RECORD
               NextIn: integer;
               NextOut: integer;
               Count: integer;
               elements: ARRAY[0..BuffSize-1] OF ITEM
             END;
VAR
    Buff: SHARED BUFFTYPE;
PROCESS producer;
BEGIN
  REPEAT
    produce(product);
    REGION Buff WHEN Buff.Count < BuffSize DO
       place(product)
   FOREVER
END;
PROCESS consumer;
BEGIN
   REPEAT
      REGION Buff WHEN Buff.Count <> 0 DO
           take(product)
    FOREVER
END;
BEGIN (*main program*)
   (*initialize Buff*)
   COBEGIN
     producer; consumer
   COEND
END.
```

Drawbacks of CCRs

Though CCRs are an improvement on semaphores, they still suffer from shortcomings. For examples:

- They may be dispersed in the program text. Thus, to understand how shared variables are used, one must examine the entire program. Ideally, all code that manipulates a particular shared variable (whose access needs to be under mutual exclusion) should be collected together in one place.

- The integrity (consistency) of a shared data structure is easily to be damaged because there is no control over what operations are carried out by the application programmer inside a CCR. The programmer himself/herself has to guarantee the correctness of the operations. So if a new process is added to the program, the programmer must verify that the process uses the shared variables correctly.

  Ideally, a set of approved operations (which can be designed by a senior person) on the data structure should be provided and all processes should be restricted to those *official* operations. This can be illustrated with a simple example.

  Suppose that a data structure represents the accounts of customers in a bank. The data structure might, for example, be an array of records:

```
CONST
   CustomerMax = ...;
TYPE
   AccountNumber = 1..CustomerMax;
   AccountRec = RECORD
                   .....
                   balance: integer;
                   .....
                END;
VAR
  Accounts: SHARED ARRAY[1..CustomerMax] Of AccountRec;
```

  The data structure would be shared by processes running on behalf of customers at automatic service machines, clerks in branches, the manager at the customer's home branch, and so on, and clearly need controls for concurrent access.

  The accounts are manipulated by various kinds of transactions. Each transaction could be modelled as a procedure; one would be the procedure to transfer funds from one customer to another.

```
 Procedure XFER(FromAcc, ToAcc:
AccountNumber; Sum: integer); BEGIN
   REGION Accounts DO
      BEGIN
        Accounts[FromAcc].balance:=Accounts[FromAcc].balance-Sum;
        Accounts[ToAcc].balance:=Accounts[ToAcc].balance+Sum;
      END
END;
```

Ideally, this procedure should be provided so that all parts of the program that need to perform transfers could call it. However, with CCRs we cannot prevent programmer from writing code elsewhere which attempts to carry out the operation. Suppose that a programmer did this and by mistake (or malice) omitted the debit of the first customer's account. We have now lost the integrity of the data structure: its state is inconsistent.

Also CCRs are difficult to implement efficiently (see Page 207 in Burns & Davies's book). To improve CCRs, monitors were proposed (Hoare, 1974).

5.4 Monitors

5.4.1 The ideas behind monitors

The first idea behind the monitor is the use of procedures and functions in sequential programming. Special tasks are programmed as procedures or functions that can be called by the main program. Procedures and functions enable a programmer to write more robust programs because a change in a procedure or a function does not affect other parts of the program.

This idea led to the use of the *monolithic monitor* used in current operating systems. Most of these systems are in effect single programs that centralized all critical functions. If a message must be passed from P_1 to P_2 then P_1 passes it to a 'big brother' monitor M with a request to forward it to P_2, or at least P_1 requests permission from M to pass the message. Such monitors are supported by hardware facilities that ensure the privileged position of the monitor: M runs in an uninterruptable mode thus guaranteeing mutual exclusion; only M can access certain areas of memory; only M can execute certain instructions such as I/O instructions.

The monitors in concurrent programming can be viewed as decentralized versions of the monolithic monitors. Each monitor will be entrusted with a specific task and in turn it will have its own privileged data and instructions. Thus, if M_1 is the only monitor that can access variable x_1 then we are ensured of mutual exclusion of access to x_1 because M_1 will be uninterruptable or as we say in the abstraction: entry (or call) to the monitor by one process excludes entry by any other process. In addition, since the only processing that can be done on x_1 is the processing programmed into the monitor M_1, we are assured that no other assignments or test are accidentally made on x_1. We can design different monitors (like that we can design different procedures and functions in sequential Pascal) for different task. Thus, the system is both more efficient because execution of distinct monitors can be done concurrently, and more robust because a change in one monitor cannot surreptitiously change a variable in another monitor.

The other idea behind monitors is that of structuring data and structuring accesses to data in a programming language. As we have said before that a data type has two attributes:

- a set of **permissible values** and
- a set of **permissible operations** on the values.

Structuring data by *typing* was first used successfully in Pascal. The main purpose of data typing is to prevent indiscriminate mixing of data that have no purpose being mixed even though their representations may be identical, and to prevent applying 'meaningless' operations on data. For example, the Pascal complier flags a syntactic error for a statement which adds a number to a character.

Although typing in Pascal is successful in dealing with the safe use of data, it has not gone far enough. For example, we have no way in Pascal to define (or declare) a type which consists of *integers* that may *only* be added or subtracted. Though this type would have the same permissible values as the type **integer**, it would not have the same set of permissible operations.

The idea of *typing by the permissible operations* is found in the language Simula 67. A *class* in Simula 67 is a data declaration together with a set of procedures which define the only legal operations that may be performed on the data. The Simula class has been combined with the Pascal data type in the Ada *package* feature which provides a carefully designed mechanism that allows the programmer to structure his/her program to reflect his/her knowledge of the properties of the data.

A *monitor* in concurrent programming can be viewed as a class that can executed by several processes under mutual exclusion. For example, although the buffer is an array, there is no reasonable operations to do with a buffer except to *place* a new element or *take* an old one.

5.4.2 Definition of monitors

A monitor is written as a set of global variable declarations followed by a set of procedures (which may be parameterized). The monitor has a *body* (**begin** .... **end**) which is a sequence of statements that is executed immediately when the program is initiated. The body is used to give initial values to the monitor variables. Therefore, the monitor exists only as a package of data and procedures. The syntax of a monitor declaration is outlined as:

```
monitor_declaration::=

MONITOR identifier;  (*name the monitor*)
   export_list (*::= EXPORT procedure_identifier_list; *)
               (*{EXPORT  procedure_identifier_list; }*)
   {
    const_declaration
   |type_declaration
   |variable_declaration
   |procedure_declaration (*include the exported procedures*)
   |function_declaration
   }
   [BEGIN                   (*the body of the monitor*)
   (*statement_sequence*)]
   END;
```

Now let's make some remarks about monitors.

1. The declaration in a monitor may include constants, types, variables, procedures and functions, but not processes or other monitors.

2. All the declarations except for the identifiers appearing in the **export** list, are only in the scope within the monitor.

3. Only the names of procedures in the **export** list can be called by a process statement of the form:

```
monitor_identifier.export_procedure_identifier[actual parameters]
```

4. The monitor body (**begin** statement-sequence), which is optional (i.e. not all monitor need a body), is executed immediately when the program is initiated, to give initial values to the monitor variables. It is just executed once during the program execution.

5. The compiler guarantees that access to the code within a monitor is done under mutual exclusion. A process that tries to execute a monitor procedure when there is already a process executing one of the procedures in the *same* monitor becomes blocked on a what is called a monitor **boundary queue**. In general, several processes may be blocked on this queue by the time an occupying process completes its monitor procedure call. Mutual exclusion is then passed to *one* of the blocked processes. In Pascal-FC, monitor boundary queues are defined to be FIFO.

5.4.3 Mutual exclusion with monitors

As an example of application of monitors to mutual exclusion, we consider the Ornamental Gardens problem with monitor. A general ornamental gardens problem is the case that we have a number of turnstiles (rather than only two) concurrent updating the number $count$ of people in the garden. The solution of the problem is now very easy: we define a monitor to control the updating of $count$ under mutual exclusion.

```
PROGRAM
GARDMON; (*Ornament gardens - monitor version*) CONST
  max=10;  (*number of turnstiles*)
MONITOR Tally;
  EXPORT
    inc, print;   (*export list*)
 VAR
    count: integer; (*global variable*)
 PROCEDURE inc;
  BEGIN
    count:=count+1
  END; (*inc*)
  PROCEDURE print;
  BEGIN
    writeln(count)
  END;  (*print*)
  BEGIN (*body of Tally*)
    count:=0
  END;   (*Tally*)
```

```
PROCESS TYPE  turnstiletype;
VAR
  loop:integer;
BEGIN
  for loop:=1 TO 20 DO
    Tally.inc     (*call monitor procedure inc*)
END;    (*turnstiletype*)
VAR
   turnstile: ARRAY[1..max] OF turnstiletype;
   procloop:integer;
BEGIN  (*main*)
  COBEGIN
     FOR procloop:=1 TO max DO
         turnstile[procloop]
  COEND;
  Tally.print
END.
```

The key feature of the program:

1. The shared data structure, such as $count$, is declared in the monitor; the code in the monitor body is executed to give initial values of the monitor variables before processes begin to call the monitor.
2. The monitor procedures inc and $print$ sit passively until called from a process.
3. If a process wishes to increment $count$, all it needs to do is to call the monitor procedure inc.
4. The monitor data structure is not directly visible from outside the monitor; it can only be accessed by executing one of the 'official' operations implemented by the exported procedures. For example, it is necessary to call the monitor procedure $print$ from the main program in order to view the final result, even though the monitor $Tally$'s mutual exclusion is not required any more after the completion of concurrent phase of execution.
5. The mutual exclusive access to the shared variable $count$ is enforced automatically by the complier when it generates code for a call to an exported monitor procedure: it is not possible to access the data except under mutual exclusion.
6. In comparison with semaphores, monitors are more structured:
   - easier to understand: all the code that manipulates a given data structure must be located in one place;
   - easier to modify: a change in a monitor does not lead to change in other parts;
   - safer to use: a monitor can be written by a senior person with unimpeachable competence and trustworthiness; then the users of the monitor need only call a procedure.

5.4.4 Condition synchronization with monitors

As we saw in the case of the Producers-Consumers Example, the problem with condition synchronization is that a shared data can only be accessed when some condition holds. For example in the producers-consumers problem with a bounded buffer, a producer can *place* a new item into the buffer *only when the buff is not full*; and a consumer can *take* an item out of the buffer *only when the buffer is not empty*. This means that a producer wishing to place an item to the buffer must be **delayed** until the condition that the buffer is not full holds *even when no process is currently accessing the buffer*. Similarly, a consumer wishing to consume an item must be delayed until the condition that the buffer is not empty *even when no process is currently accessing the buffer*.

We can program the buffer as a monitor with two exported procedures *place* and *take* as follows.

```
MONITOR Buffer; EXPORT
   place, take;
CONST
   BuffSize=5;
   BuffMax=4;
VAR
  b: ARRAY[0..BuffMax] Of char;
  n, in, out: integer;
PROCEDURE place(ch: char);
BEGIN
  IF n>= BuffSize THEN
  "wait until not full";
  b[in]:=ch;
  n:=n+1;
  in:=(in+1) MOD BuffSize;
  "signal that the buffer is  not empty"
END; (place*)
PROCEDURE take(VAR ch:char);
BEGIN
  IF n=0 THEN
    "wait until not empty";
  ch:=b[out];
  n:=n-1;
  "signal that the buffer is notfull";
END; (*take*)

BEGIN  (*body of Buffer*)
n:=0;
in:=0;
out:=0
END; (*Buffer*)
```

The problem is then how we represent the statements ”wait until condition” and ”signal condition”. As we saw in Chapter 1, we can of course use busy waiting: “wait until condition” is represented as

```
WHILE NOT condition DO null
```

And "signal condition" will be simply an empty statement. But as we pointed out before, busy waiting is inefficient and thus not desirable in concurrent programming. Furthermore if a process is busy waiting in a monitor procedure, the monitor will never be released. This would lead to deadlock (or at least starvation). We prefer an implementation using blocked waiting so that a process waiting for a condition is taken from the processor to allow other executable process to be executed. For this purpose, Hoare's suggested (in 1974) the use of **condition** variables for condition synchronizations. Variables of type **condition** have no values accessible to the programmer but instead are FIFO queues which are initialized automatically to the empty queue on declaration. The declarations of condition variables are exemplified by:

```
VAR
 c: condition;
 carray: ARRAY[1..10] OF condition
```

To block and unblock a process on a condition, Pascal-FC introduced two principle operations called **delay** and **resume** respectively. Many discussions of monitors elsewhere use terms **wait** and **signal** respectively, which has been reserved for the semaphore operations in this course.

The delay operation

The delay operation on a condition is the counterpart of the semaphore wait operation. In Pascal-FC, it is implemented as a (standard) procedure, which is called in the following way:

```
delay(c)
```

where c is a condition variable. A process calling this procedure can be understood to announce 'I am waiting for c to occur'. So it looks as if we should define the semantics of delay(c) as follows:

delay(c): the calling process is blocked
and is entered into a queue of processes blocked on c.

However, remember that it is the monitor that controls the mutual exclusion. If we take the above semantics, a process executing the delay operation inside the monitor would still hold up the monitor and no further process could gain access to the monitor. This would be undesirable because that on the one hand it might lead to waste of execution time, and on the other hand that is more dangerous, this would lead to deadlock when only such a further access to the monitor could make the condition c to occur. Therefore, the full semantics of $delay(c)$ should be:

delay(c): the calling process releases the monitor
the calling process is blocked
the calling process is entered into a queue of processes blocked on c

The differences between delay and semaphore wait

There are important differences between **delay** on a condition and **wait** on a semaphore.

1. **delay** always causes the calling process to be blocked, whereas the semaphore **wait** only does so if the value of the semaphore is 0.
2. **delay** always release the current monitor when blocking a process. If semaphores were used rather than conditions in a monitor, **wait** on a semaphore does not release the current monitor when blocking a process. So using a semaphore rather than a condition in a monitor is likely to lead deadlock.

The resume operation

This is the counterpart of the **signal** operation on semaphores. This operation is again a procedure in Pascal-FC, and is called as follows:

```
resume(c)
```

A process calling $resume(c)$ can be understood as to announce 'I am signalling that c has occurred', so that the first process waiting for c to occur can carry on its execution. A first description of the semantics of this operation is;

resume(c): unblock the first process waiting on c

What happens if no *process is blocked on* c *when* **resume** *is blocked?* It is defined that executing resume(c) when there are no processes waiting in the queue for c is a non-operation. So the full semantics of **resume** should be

resume(c): IF the queue for c is not empty THEN
unblock the first process on the queue

The difference between resume and semaphore signal

If the queue for the condition is empty, *resume* has no effect at all. But **signal** always has some effect: either it unblocks a process, or increments the semaphore value.

A note on the operations of conditions

Consider the following problem with the use of **resume** to unblock a process.

- Process P is blocked when executing **delay(c)** in monitor M,
- Process Q unblocks P when executing **resume(c)** in monitor M.

The problem is that when P is unblocked, it will re-enter the M from the point just after the **delay(c)**. However, Q is already inside M. To have two processes in one monitor at the same time would violate the mutual exclusion. So we cannot permit this to happen. Here we consider two solutions to this problem.

1. The resumed process P must wait until the resumer Q leaves the monitor (by completing its call to the monitor procedure or by executing a **delay**) before re-entering M.
2. The resumer Q must immediately 'step outside' the monitor M, handling the mutual exclusion to the resumed process Q.

Pascal-FC has chosen the second solution.

5.4.5 More examples

This section gives more illustrative examples starting with a full program for the Producer-Consumer with a bounded buffer.

5.4.6 Producer-Consumer with a bounded buffer

We simple replace "wait until not full" by $delay(notfull)$, "signal not empty" by $resume(notempty)$, etc. in our monitor given at the beginning of Section 4.

```
PROGRAM PCMON;
    (*Producer-Consumer - monitor version*)
MONITOR Buffer;
EXPORT
   place, take;
CONST
   BuffSize=5;
   BuffMax=4;
VAR
  b: ARRAY[0..BuffMax] Of char;
  n, in, out: integer;
  notfull, notempty: condition;

PROCEDURE place(ch: char);
BEGIN
  IF n>= BuffSize THEN
```

```
  delay(notfull);
  b[in]:=ch;
  n:=n+1;
  in:=(in+1) MOD BuffSize;
  resume(notempty)
END; (place*)

PROCEDURE take(VAR ch:char);
BEGIN
  IF n=0 THEN
    delay(notempty);
  ch:=b[out];
  out:=(out+1) MOD Buff
  n:=n-1;
  resume(notfull);
END; (*take*)

BEGIN  (*body of Buffer*)
n:=0;
in:=0;
out:=0
END; (*Buffer*)

PROCESS producer;
VAR
 ch: char;   (*local to producer*)
BEGIN
 FOR local:='a' TO 'z' DO
    Buffer.place(ch);
END; (*producer*)

PROCESS consumer;
VAR
  ch: char;    (*local to consumer*)
BEGIN
  REPEAT
    Buffer.take(ch);
    write(ch);
  UNTIL ch='z';
  writeln
END;  (*consumer*)

BEGIN (*main*)
  COBEGIN
    producer;
    consumer
  COEND
END.
```

5.4.7 Readers and writers

In Chapter 3, we presented two semaphore solutions for this problem, both used the protocols **open** and **close** for accessing to the data structure. We now implement these protocols as monitor procedures. The monitor version of the first solution becomes as follows.

```
PROGRAM RWMON1;
MONITOR ReadWrite1;
  EXPORT
    open, close;
VAR
nr: integer;
writing: boolean;
oktowrite: condition;
PROCEDURE open(read: boolean);
BEGIN
  IF read THEN
    BEGIN
      IF nr=0 THEN
         IF writing THEN
            delay(oktowrite);
         nr:=nr+1;
     END ELSE
        IF nr>0 OR writing THEN
           delay(oktowrite);
        writing:=true;
END; (*open*)
PROCEDURE close(read:boolean);
BEGIN
  IF read THEN
    BEGIN
      nr:=nr-1;
      IF nr=0 THEN resume(oktowrite)
    END ELSE BEGIN writing:=false; resume(oktowrite) END
END; (*close*)

BEGIN
 nr:=0; writing:=false
END; (*ReadWrite1*)

PROCESS TYPE reader;
VAR local: SomeType;
BEGIN
  REPEAT
    ReadWrite1.open(true);
    "access data, copy to local";
    ReadWrite1.close(true);
    "use copied data"
  FOREVER
END; (reader)
```

```
PROCESS TYPE writer;
VAR local: SomeType;
BEGIN
  REPEAT
   "produce new local value";
   ReadWrite1.open(false);
   "write local value to data"
   ReadWrite1.close(false)
  FOREVER
END; (*writer*)

VAR
  I,J: integer;
  reasderprs: ARRAY[1..10] OF reader;
  writerprocs: ARRAY[1..10] OF writer;

BEGIN (*main*)
  COBEGIN
     FOR I:=1 TO 10 DO
         readerprocs[I];
     FOR J:=1 TO 10 DO
         writerprocs[J]
   COEND
END.
```

Note that in this program we used a test operation **empty** on the condition $oktowrite$. In general this operation is called in the form

```
   empty(c)     where c is a condition variable
```

It is a Pascal-FC function which returns a boolean result: it returns $true$ if the condition queue is empty (i.e. if no process is blocked on the condition), false otherwise.

This program, as the first semaphore solution to the problem, may lead to starvation of writers. The second semaphore solution in Chapter 3 avoided this liveness problem, but it was quite complex. We shall see that a monitor solution that treats readers and writes fairly is reasonably simple. The following solution was proposed by Hoare (1974):

1. a new reader should not be permitted to start if there is a waiting writer;
2. at the end of a write operations, waiting readers are given preference over waiting writers.

Thus, we need another condition $oktoread$ to block readers.

```
MONITOR ReadWrite2;
  EXPORT
    open, close;
```

```
VAR
  nr: integer;
  writing: boolean;
  oktoread, oktowrite: condition;

PROCEDURE open (read:boolean);
BEGIN
  IF read THEN
  BEGIN
    IF writing OR NOT empty(oktowrite) THEN
       delay(oktoread);  (*Hoare's proposal 1*)
    nr:=nr+1;            (*permitted to read*)
    resume(oktoread);    (*unblock the blocked readers*)
  END ELSE
  BEGIN
    IF writing OR (nr<>0) THEN
       delay(oktowrite)
     writing:=true;       (*permitted to write*)
  END
END; (*open*)

PROCEDURE close(read:boolean);
BEGIN
  IF read THEN
  BEGIN
     nr:=nr-1;           (*one reader leaving*)
     IF nr=0 THEN
       resume(oktowrite)
  END ELSE
  BEGIN
    writing:=false;         (*finished writing*)
    IF NOT empty(oktoread) THEN (*Hoare's proposal 2*)
      resume(oktoread)
    ELSE
      resume(oktowrite)
  END
END; (*close*)

BEGIN
   writing:=false;   (*no one is writing initially*)
   nr:=0;            (*no one is reading initially*)
END;  (*ReadWrite2*)
```

5.4.8 Reasoning about monitor programs

Let us now give semi-formal proofs of some properties of second solution to the readers and writers problem using monitor $ReadWrite2$. Let R be the number of processes currently reading and let W be the number processes currently writing.

The basic safety property required of a solution to the problem of the readers and writes will be proven if we can show that the following formula I is *invariant*, i.e. I holds at any time during the program execution:

$$((R > 0) \Rightarrow W = 0) \wedge ((W > 0) \Rightarrow (W = 1 \wedge R = 0))$$

This property reads

If $R > 0$ then $W = 0$ and if $W > 0$ then ($W = 1$ and $R = 0$)

This means that if there is some process currently reading then no process is current writing, and if there is some process currently writing then there is exactly one process current writing and no process is current reading.

To prove the safety property I, we need to prove the following *monitor invariants*, i.e. the formulae are invariant *outside* the monitor.

(a) $R = nr$;

(b) $W > 0$ iff $writing = true$;

(c) $nonempty(oktoread)$ only if ($writing$ or $nonempty(oktowrite)$);

(d) $nonempty(oktowrite)$ only if ($nr \neq 0$ or $writing$).

To prove these formulae are monitor invariants of monitor $ReadWrite2$, we have to show that

1. each of these statements is initially true (i.e. just after the execution of the body of the monitor);
2. if a statement is true upon entry into a monitor procedure, it is still true when the process exit the procedure.

Thus, it allows these formulae to be false during the execution of a monitor procedure. However, we should be clear about the points of a process entry to and exit from a monitor procedure:

- a process enters a monitor procedure either by a *call* to the procedure, or by a **resume(c)** operation executed by another process;
- a process exits from a monitor procedure by completing the procedure, or by executing a **delay(c)** operation, or by executing a **resume(c)** operation when there is a process waiting for c (see item 1 as well).

We leave the proofs of these monitor invariants as an exercise.

Now we use these monitor invariants to prove the program invariant I by showing that any attempt to describe an execution (interleaving) sequence which falsified I is unsuccessful. We first note that I holds initially since $R = W = 0$.

1. Suppose $R > 0$ and $W = 0$ (so that I holds) and then I is falsified by some process starting to write (so W will be 1).

By (a), $R > 0$ implies $nr > 0$ so the process that wishes to write will **delay** in procedure **open**. Therefore, the only way this scenario could falsify I is if a **resume(oktowrite)** occurs. The **resume** operation in the **close(true)** procedure is only executed only if $nr = 0$, contrary to the assumption that $nr > 0$. The **resume** operation in the **close(false)** will also not be executed since there are no writers by the assumption $W = 0$.

2. Suppose $R = 0$ and $W > 0$ and then some process starts reading so that $R = 1$, falsifying I.

   $W > 0$ implies $writing = true$ by (b), so any process executing **open(true)** will **delay** on *oktoread*. Since $R = 0$, there are no readers so **resume(oktoread)** is not executed in **close(true)**. Now I is assumed true so $W > 0$ implies $W = 1$. Thus, executing **resume(oktoread)** in **close(false)** upon termination of writing occurs when $W = 0$ contradicting the assumption of this scenario.

3. $W = 1$, $R = 0$ and then some process starts writing to falsify the second clause of I.

   This is impossible by the code in **open(false)**. the only possible **resume** is the one from **close(false)**, but $R = W = 0$ so I is not falsified.

We have claimed that the second solution to the readers and the writers problem satisfies the liveness property:

- If a process P wishes to read (or to write), then *eventually* it will be allowed to so.

The proof of this liveness property will use the four monitor invariants (a), (b), (c), and (d), and the assumption that queue of processes that are blocked on a condition is FIFO. We leave the proof of this property as an exercise.

5.4.9 The expressiveness of monitors

We claim here that

1. a program with semaphores can always be simulated by a program with monitors.
2. a program with monitors can always be simulated by a program with semaphores.

This means that the semaphore facilities have exactly the same expressive power. Hence the decision to use monitors can be made solely on the basis of their contribution to *clarity* and *reliability* of the resulting system.

5.5 Exercises

1. Using the following declaration:

   ```
   VAR
    eating : SHARED ARRAY[1..N] Of boolean
   ```

   and CCRs to solve the dining philosophers problem, assuming that acquiring chopsticks involves setting $eating[i]$ to true; releasing chopsticks involves setting $eating[i]$ to false. Your solution should not deadlock.

2. Show how CCRs can be used to solve the problem of Readers and Writers problem. Your solution should have readers to have preference over writers.

3. Show how CCRs can be used to solve the problem of Readers and Writers in which waiting writers have preference over coming readers.
4. (*Atomic Broadcast*) Assume that one producer process and n consumer processes share a bounded buffer of size b. The producer deposits messages in the buffer; consumers fetch them. Every message deposited by the producer is to be received by all n consumers. Furthermore, each consumer is to receive the messages in the order they were deposited. However, different consumers could receive up to b more messages than another if the second consumer is slow.

   Develop a solution to this problem that uses CCRs for synchronization.
5. Use monitor(s) to solve question 1.
6. Use monitors to solve question 4.
7. Use monitors to write a program in which a number of customer processes interact with a bank cash dispensing process. A customer can ask for balance information or request money; however, the account must not go negative. A maximum of 100 pounds can be withdrawn in an one-hour period.
8. Modify the previous exercise so that there are now three cash dispensers. a customer may go to any dispenser. Note that more than one custmomer may share an account.
9. Tow kinds of processes, A's and B's, enter a room. An A process cannot leave until it meet two B processes, and a B process cannot leave until it meets one A process. Either kind of process leaves the room – without meeting any other processes – once it has met the required number of processes.

   (a) Develop solution to the problem that uses a monitor to implment this synchronization.

   (b) Modify your answer to (a) so that the first of the two B processes that meets an A process does not leave the room until after the A process meets a second B process.
10. Prove the four monitor invariants (a), (b), (c), and (d) for $ReadWrite2$.
11. Prove the liveness property of the second solution to the readers and the writers problem.
12. Write a program to implement a binary semaphore by a monitor.
13. Give an algorithm for transforming a program using monitors to a program that uses semaphores.

6 SYNCHRONOUS MESSAGE PASSINGS

The goals of the chapter

- Motivation for message passing
- Understand asynchronous and synchronous message passing
- Understand the use of *channels* for inter-process communication and synchronization
- Understand non-determinism in the message passing model and the use of selective waiting construct for non-determinism.
- Programming techniques using message passing

6.1 Introduction

The synchronization constructs we have examined so far are all based on shared variables. Consequently, they are used in concurrent programs that execute on a hardware in which *processors share memory*.

However, *network architectures*, in which processors share only a communication network, have become increasingly common. Examples of such architectures are networks of workstations or multicomputers. In addition, hybrid combinations of shared-memory and network architectures are sometimes containing workstations and multiprocessor computers. Even on shared memory architectures, it is often necessary or convenient for processes not to share variables; e.g., processes executing on behalf of different users usually have different protection requirements.

To write programs for a network, it is first necessary to define the network interface, i.e., the primitive network operations. In abstraction, there are two primitive operations on a network, **SEND** and **RECEIVE**. These could simply be read and write operations analogous to read and write operations on shared variables. However, this would mean that processes have to employ busy-waiting synchronization. Better approach is to define special network operations that include synchronization, much as semaphore operations are special operations on shared variables. Such network operations are called *message-passing primitives*. In fact, message passing can be viewed as extending semaphores to convey data as well as to provide synchronization.

When message passing is used, networks are typically the only objects that processes share. Thus, every variable is local to and accessible by only one process, its *caretaker*. This implies that variables are never subject to multiple accesses, and therefore no special mechanism for mutual exclusion is required. The absence of shared variables also changes the way in which condition synchronization is programmed since only a caretaker process can examine the variables that encode a condition. This requires using programming techniques different from those employed with shared variables. The final consequence of the absence of shared variables is that processes need not execute on processors that share memory; in particular, processes can be distributed among processors. For this reason, concurrent programs that employ message passing are called *distributed programs*. Such programs can, however, be executed on centralized processors, just as any concurrent program can be executed on a single, multitasking processor. In this case, networks are implemented using shared memory.

6.2 Three forms of communication

With message passing, we are usually concerned with three kinds of synchronization classified according to the nature of the **SEND** and **RECEIVE** operations:

- asynchronous message passing
- synchronous message passing (simple rendezvous)
- remote invocation (extended rendezvous)

If the SENDing process continues executing without being blocked by the **SENDING** operation, the message passing is called *asynchronous*. In other words, the **SEND** operation in asynchronous message passing is a *non-blocking* primitive. However, in contrast to **SEND**, the **RECEIVE** operation is a *blocking* primitive since the receiving process has to be delayed by this operation if there is no message which has already sent and not yet received.

Alternatively, the message passing is called *synchronous* if a sender attempting a **SEND** operation is delayed until the corresponding **RECEIVE** is ready to be executed. In other words, both the **SEND** and **RECEIVE** operations are *blocking* primitives. If the sending (or receiving) process arrives at **SEND** (or **RECEIVE** respectively) earlier than the receiving (or sending respectively) process arriving at **RECEIVE** (or **SEND**), it has to wait for the arrival of the receiving (or sending respectively) process. For this reason, synchronous communication is also called *rendezvous*.

If the SENDing process is delayed further until a reply message is received by it from the RECEIVing process, the value passing is called remote invocation (or extended rendezvous). This will be dealt with in the next chapter.

Obviously, the syntaxes for the **SEND** and **RECEIVE** operations in representing these forms of communication must be different as well as their semantics.

To understand the difference between these three kinds of communication, consider the following analogy:

- The posting and receiving of a letter is an asynchronous message passing - once the letter has been put into the letter box the sender proceeds with his/her life. Only by receiving a reply letter from the receiver can the sender ever know that the first letter actually arrived. From the receiver's point of view, a letter can only inform the receiver about an out-of-date event; it says nothing about the current position of the sender.
- Sending and receiving of messages through fax machines is synchronous communication. The SENDing fax machine waits until contact is made and the identity of the RECEIVing machine verified before the message is transmitted. When the transmission is complete, the sender continues.
- Communication by telephones is an analogy for remote invocation. The sender (caller) waits not only for the message to be transmitted, but also for a reply to be returned.

There is a tradeoff between asynchronous and synchronous message passing. On the one hand, since the **SEND** operation is non-blocking, it has the following consequences.

- A sending process can get arbitrarily far ahead of a receiving process. If process P sends a message to process Q and later needs to be sure Q got it, P needs to wait to receive a reply from Q.

- Message delivery is not guaranteed if failures can occur. If P sends a message to Q and does not get a reply, P has no way of knowing whether the message could not be delivered, Q crashed while acting on it, or the reply could not be delivered.
- Messages have to be buffered, yet buffer space is finite in practice. If too many messages are sent, either the program will crash or **SEND** will be blocked.

Synchronous message passing avoids these consequences. In particular, both **SEND** and **RECEIVE** are blocking primitives. If a process tries to send to another process, it delays until the receiver is waiting to receive from it. Thus, a sender and a receiver synchronize at every communication point. If the sender proceeds, then the message was indeed delivered, and message do not have to be buffered.

On the other hand, synchronous message passing is more difficult and inefficient to program algorithms that in nature need messages to be buffered.

However, synchronous and asynchronous message passing can simulate one another. Thus, they have the same expressive power. We will not cover asynchronous communication in this course. But students should be able to learn it after the lectures on synchronous message passing.

6.3 Naming the destinations and sources of messages

Another important issue in the design of a programming language for message passing is how destinations and sources of messages are designated: the sending process should indicate where (which process) the message goes to, and the sending process should know from which process it receives (waits for) the message. There are basically two independent decisions that the language designer may consider here:

- whether naming is *direct* or *indirect*;
- whether the naming scheme is symmetrical or asymmetrical.

Obviously, different decisions made by the designer lead to different syntactic and semantic definitions of the **SEND** and **RECEIVE** operations.

Direct naming: This is the simplest form of naming. All processes in the system have unique names (identifiers); a **SEND** operation will then directly name the destination processes:

```
SEND message TO ProcessName
```

A **symmetric** form for the **RECEIVE** operation in the receiving process would be

```
RECEIVE message FROM ProcessName
```

Thus, this symmetric form requires the receiver to know the name of any process liable to send it a message. By contrast, an asymmetric form may be used if the receiver is only interested in the existence of a message, rather than from where it came:

```
RECEIVE message
```

Indirect naming: Where the unique naming of all processes is inappropriate, a language may define intermediaries (usually called mailboxes or channels) that are named by both partners in the communication. The naming is then said to be indirect:

```
SEND message TO mailbox
RECEIVE message FROM mailbox
```

6.4 Channels in Pascal-FC

In this chapter, we use a model for synchronous message passing which resembles that found in Occam (IMMOS, 1984) originally proposed in CSP (Hoare, 1985).

Processes communicate with each other via named channels. A channel is an abstraction of a physical communication network; it provides a communication path between processes. With message passing, processes share **channels**. Channels are accessed by means of two kinds of primitives:

```
ch ! e     (*SEND the value of the expression e*)
           (*to the channel ch*)
ch ? x     (*RECEIVE from channel, ch, a value*)
           (*and assign it to variable x*)
```

Operations on the channel are synchronized: whichever process arrives at the channel operation first will be blocked until the other process arrives. When both processes are ready, a rendezvous is said to take place, with data passing from the expression e to the variable x. Thus, a synchronous message passing can be understood a distributed assignment:

```
x:=e
```

where x is in one process (receiver) and e is in another (the sender), with the expression evaluated by the sender and assigned to a variable in the receiver.

We will call the two channel operations $ch?x$ and $ch!e$ on the *same* channel *co-operations*. $ch?x$ is also called a channel read operation, while $ch!e$ a channel write operation.

In this model, each channel can only be used by a single sender and a single receiver; communication is point-to-point. Moreover, a channel can only pass information in one direction.

6.5 The types of channels

To discuss the type of a channel, we should first know the term *message*. In concurrent programming, a message can be a data of any structure, such as a structured data type. Therefore, a message could be complex or simple

(such as a 16-bit word). Therefore, a message is always of some data type.

The type of a channel must indicate what type of messages it can carry. Thus, the type of the message of a channel is the *base type* of the channel. In Pascal-FC, we may have the following declarations;

```
TYPE chan  = CHANNEL OF integer;
     chans = ARRAY[1..n] OF chan;
VAR
     pipeline: chans;
     link:   CHANNEL OF integer;

TYPE packet= RECORD
               (*some suitable structure*)
             END;
 VAR
     network: CHANNEL OF packet;
```

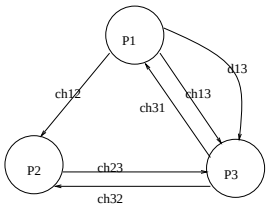
Channels must be declared at the beginning of the program because they are shared among processes.

Graphical representation of programs

A graphical representation for concurrent programs with message passing is quite helpful in the design of programs. In general, a concurrent program consists of a number of processes which are connected by channels that represents the messages flows between processes. Therefore, a concurrent program can be represented as a *labelled directed graph* called the *configuration graph* of the program in which

- a process is represented as a *node* labelled with the name of the process;
- a channel is represented as a directed line labelled with the name of the channel and the line directs from the sending process to the receiving process.

An example of a configuration graph is as follows:



This graph represents a concurrent program of the following sketch:

```
PROGRAM sketch; TYPE CH = CHANNEL
OF sometype; VAR ch, d: ARRAY[1..3,1..3] OF CH; PROCESS P1; VAR x,
.., :sometype; BEGIN
```

```
   .....
   ch[1,2] ! e;
   ......
   d[1,3] ! e;
   ......
   ch[1,3] ! e;
   .......
   ch[3,1] ? x
   ........
END;
PROCESS P2;
VAR x, y, z...: sometype;
BEGIN
   .....
   ch[1,2] ? x;
   ......
   ch[2,3] ! e;
   .......
   ch[3,2] ? y;
   .......
END;

PROCESS P3;
VAR x,y, ..: sometype;
BEGIN
    .....
    ch[1,3] ? x;
    ......
    ch[2,3] ? y;
    ......
    ch[3,2] ! e;
    ......
    ch[3,1] ! e;
    d[1,3] ? z;
    ......
END;
BEGIN
  COBEGIN
     P1;P2;P3
  COEND
END.
```

6.6 A classification of distributed processes

Before getting into a full example, let us first make a classification on the processes in distributed programs.

There are three basic kinds of processes in a distributed program: *filters, clients, and servers*. A *filter* is a data transformer. It receives streams of data values from its input channels, performs some computation on those values, and sends streams of results to its output channels. Because of these attributes, we can design a filter independent of

other processes. Moreover, we can readily connect filters into networks that perform larger computations. All that required is that each filter produces output that meets the input assumptions of the filter(s) that consume that output. Many of the user-level commands in the UNIX operating system are filers, e.g., the text formatting programs tbl, eqn, and $troff$.

A *client* is a triggering process; a *server* is a reactive process. Clients make requests that trigger reactions from servers. A client thus initiates activity, at times of its choosing; it often then delays until request has been served. A server waits for requests to be made, then reacts to them. The specific actions a server takes can depend on the kind of the requests, parameters in the request messages, and the server's state; the server might be able to respond to a request immediately, or it might have to save the request and respond later. A server is often a non-terminating process and often provides service to more than one client. For example, a file server in a distributed systems typically manages a collection of files and services requests from any client that wants to access those files.

6.6.1 An network of filters – Prime number generation

The sieve of Eratosthenes - named after the Greek mathematician who developed it - is a classic algorithm for determining which numbers in a given range are prime. Suppose we want to generate all the primes between 2 and n:

1. First, write down a list with all the numbers:

   $$2,\ 3,\ 4,\ 5,\ 6,\ \ldots,\ n$$

2. Starting with the first uncrossed-out number in the list, 2, go through the list and cross out multiples of that number. If n is odd, this yields the list:

   $$2,\ 3,\ \not{4},\ 5,\ \not{6},\ \ldots,\ n$$

   At this point, crossed-out numbers are not prime; uncrossed-out numbers are still candidates for being prime.

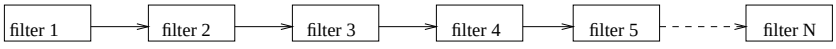
3. Now moving to the next uncrossed-out number in the list, 3, and repeat the above process by crossing out multiples of 3.

4. Continue this process until every number has been considered, the uncrossed-out numbers in the final list will be all the primes between 2 and n.

The essence of this algorithm is that the primes form a sieve that prevents their multiples from falling through.

You can easily write a sequential program for this algorithm. Now consider how we might parallelize this algorithm. One possibility is to assign a different process to each possible prime p and to have each in parallel cross out multiples of p. However, if we can know each p is prime, we do not have to solve this problem anymore.

Now we employ a *pipeline* of filter processes as shown in the following configuration graph:



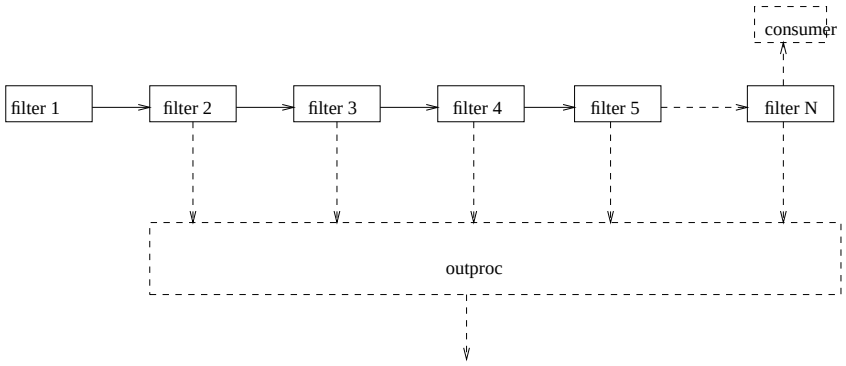
- The first filter process, $filter[1]$, sends the stream of integers starting at 2 (i.e. 2, 3, 4, 5, 6, ...).

- Every other filter process receives a stream of numbers from its left neighbour.
- The first number p that process $filter[i]$ ($i > 1$) receives is the $(i-1)$th prime.
- Each $filter[i]$ subsequently passes on all other numbers it receives that are *not* multiples of its prime p (discards all the multiples of p).
- These N filters generates the first $N-1$ primes.

We can now write a program as follows:

```
PROGRAMM sieve1; CONST N= ... (*N-1 = number of primes to
generate*) TYPE chan = CHANNEL Of integer; VAR pipeline :
ARRAY[1..N] OF chan;
    ploop:integer
PROCESS filter[1];
VAR i: integer;
BEGIN
   i:=2;
   REPEAT
     ch[1] ! i;
     i:=i+1
   FOREVER
END;
PROCESS TYPE filters(i:integer);
VAR p,next: integer;
BEGIN
   ch[i-1] ? p;
   REPEAT
     ch[i-1] ? next;
     IF (next MOD p) <> 0 THEN ch[i] ! next
   FOREVER
END;
VAR
  filter: ARRAY[2..N] OF filter;
BEGIN
   COBEGIN
      filter[1]; FOR ploop:=2 TO N DO filter[ploop](ploop)
   COEND
END.
```

The above program terminates in deadlock. The $filter[N]$ will be blocked on $ch[N]!next$ since no process is ready to consume its output; this in turn will block filter[N-1]. The blocked filter[N-1] will block filter[N-2], and so on. The program does not print out the primes generated either. To solve these two problems, we add two processes, $consumer$ which consumes the integers passing through filter[N], and $outproc$ which receives the prime from each filter and print it out:




```
PROGRAM sieve; CONST N= ... (*N-1 = number
of primes to generate*) TYPE chan = CHANNEL Of integer; VAR
pipeline : ARRAY[1..N] OF chan;
    output: ARRAY[1..N] OF chan;    (*added*)
    ploop:integer
PROCESS filter[1];
VAR i: integer;
BEGIN
   i:=2;
   REPEAT
     ch[1] ! i;
     i:=i+1
   FOREVER
END;
PROCESS TYPE filters(i:integer);
VAR p,next: integer;
BEGIN
   ch[i-1] ? p;
   output[i] ! p;    (*added: send p to outproc*)
   REPEAT
     ch[i-1] ? next;
     IF (next MOD p) <> 0 THEN ch[i] ! next
   FOREVER
END;
PROCESS consumer;
VAR local: integer;
BEGIN
  REPEAT ch[N] ? local FOREVER
END;
PROCESS outproc;
VAR I, Num: integer;
BEGIN
  FOR I:=2 TO N DO
    BEGIN
     output[I] ? num;
     writeln(num)
    END
END;
```

```
VAR
  filter: ARRAY[2..N] OF filter;
BEGIN
   COBEGIN
      filter[1]; consumer; outproc;
      FOR ploop:=2 TO N DO filter[ploop](ploop)
   COEND
END.
```

Then above program will not deadlock. However, it will not terminate. It is certainly desirable that the program should terminate normally after all N-1 primes have been printed. We will come back later to discuss the termination problem in general.

6.7 Synchronous channels

As mentioned earlier, with message passing processes synchronize while communicating. The sending process and receiving process synchronize at the communication point. Exchange of message and synchronization are carried out by the sending and receiving operations which are executed simultaneously. However, sometimes two processes need to synchronize without the need of exchange of a message. In this case a dummy piece of data would have to be communicated. This can lead to confusion for someone reading the program at a later date. Pascal-FC allows the intension of the programmer to be clearly expressed by introducing a special base type for such *contentless* communication.

The type **synchronous** is predefined. There are no values associated with this type. A variable called **any**, of type **synchronous**, is automatically declared by the compiler for every program. The following sketch code illustrates how one process (**starter**) can be used to delay and then release two (**worker**) processes.

```
PROGRAM starters;
TYPE syn = CHANNEL OF synchronous;
     barriers = ARRAY[1..2] OF syn;
VAR barrier: barriers;
PROCESS starter;
VAR I: integer;
BEGIN
....
  FOR I:=1 TO 2 DO
     barrier[I] ! any;
...
END;
PROCESS TYPE worker(num:integer);
BEGIN
 ....
 barrier[num] ? any;
 ...
END;
VAR workers: ARRAY[1..2] of worker;
...
```

Each worker will be delayed at the barrier until the starter releases it (by sending **any**).

6.8 Selective waiting construct

In this section we introduce the important *selective wait* construct. A message passing language without a **select** construct can be compared to a sequential language without **if** statement. We shall illustrate the need for this construct by considering one solution to the Ornamental Gardens Problem.

In the message passing passing model *both active and passive* objects must be represented by processes. Thus, the visitor-counting variable must be embodied in a process that serializes accesses to it. The following program represents a solution to the problem. It is correct *in the sense* that the value of $count$ is always correct. There is, however, a serious problem in the system design.

```
 PROGRAM
gardens5a; (*5 here is for chapter 5*) VAR path: ARRAY[1..2] OF
CHANNEL OF integer; PROCESS TYPE turnstile(i:integer); VAR
loop:integer; BEGIN
  FOR loop:=1 TO 20 DO path[i] ! 1
END; (*turnstile*)
VAR
  turnstiles: ARRAY[1..2] OF turnstile;
PROCESS counter;
VAR count: integer;
    I, temp:integer;
BEGIN
   count:=0;
   FOR I:=1 TO 20 DO
     BEGIN
        path[1] ? temp;
        count:=count + temp;
        path[2] ? temp;
        count:=count+temp
     END;
   writeln('Total admitted:', count)
END;
BEGIN
  COBEGIN counter; turnstile[1]; turnstil[2] COEND
END.
```

Problem of this program: The controller process ($counter$) insists that visitors enter at the same rate through the two turnstiles. Once a ticket has been sold at turnstile[1] another cannot be issued until someone has bought a ticket at turnstile[2]. Also, the first ticket has to be sold at turnstile[1]. This is clearly, in general practice, an unacceptable solution.

The requirement should be such that the controller interact with the other processes ' as required', not in an order predefined and embodied in the controller. If more people come to turnstile[1] then more tickets should be sold

there. The controller needs to react dynamically to the incoming calls (from its environment). To do this, a non-deterministic selective waiting construct is needed.

In general, for one process P in a concurrent program, its *environment* consists of the rest of the processes in the program. It often that a process P has to interact with its environment and its environment in general behaves no-deterministically (if it has more than one process). The problem here then becomes: how can we program P such that it will not restrict the non-determinism of its environment?

The basic selective waiting construct

To deal with nondeterminism, we introduce the select construct to Pascal-FC. We first look at the basic select construct which is simpler and needs to be extended later to increase its expressiveness.

A basic select construct is of the following form:

```
 SELECT
    alternative;
 OR
    alternative;
 OR
    alternative;
  .........
 OR
    alternative
END;
```

where SELECT and OR are reserved words. Each alternative is a statement sequence *starting with a channel operation*, (i.e. ch ? x or ch ! e).

Let us explain the semantics of the basic select construct by using the Ornamental gardens problem. Now we rewrite the process $counter$ as follows:

```
PROCESS counter;
VAR count: integer;
    I,J, temp: integer;
BEGIN
  count:=0;
  FOR I:=1 TO 40 DO
  BEGIN
    SELECT
       paths[1] ? temp;
     OR
       paths[2] ? temp
    END;
    count:=count+temp
  END;
  writeln('Total admitted:',count)
END.
```

The semantics can be explained as follows:

- When a process, $counter$ for example, attempts the SELECT, an alternative, say $paths[1]?temp$, is said to be *ready* if the environment of the process is ready to execute the *co-operation* of the first channel operation of the alternative, ($paths[1]!1$ in this case). In this case, this co-operation is executed by process $turnstile[1]$ of the environment of $counter$.
- If there are some alternatives ready when the process attempts SELECT, then one of the ready alternative will be nondeterministically chosen to execute.
- If there is no ready alternative when the process executing SELECT, the process is suspended until one of the alternatives becomes ready.

Thus, the $counter$ will respond to whichever process wishes to call it. If there is more than one outstanding call then one will be chosen. Pascal-FC uses a random algorithm to choose. However, the point about random *versus* non-deterministic behaviour made before must be borne in mind.

After a channel operation that distinguishes the alternative, any number of further statements can be given. For example, the **select** statement above could have been written as:

```
SELECT
   paths[1] ? temp; count:=count+temp;
 OR
   paths[2] ? temp; count:=count+temp
END;
```

Where the selection is between elements from an array of channels, an shorthand can be used. For example,

```
SELECT
      ch[1] ? v[1];
   OR
      ch[2] ? v[2];
   OR
      ch[3] ? v[3];
   OR
      ch[4] ? v[4];
   OR
      ch[5] ? v[5];
   OR
      another ! e
END;
```

can be written as:

```
 SELECT
    FOR i:=1 TO 5 REPLICATE
        ch[i] ? v[i];
```

```
  OR
    another ! e
END;
```

6.9 Guarded alternatives

Now let us look at how we can deal with condition synchronization in the massage passing model. Condition synchronization requires that a process must be delayed until some condition holds (or occurs). We can express a condition by a boolean expression which will be used as the *guard* of an alternative in the **select** construct. The syntax of a **guard** is as follows:

```
   WHEN boolean_expression =>
```

Thus, the basic select construct is extended to the following form which is called **guarded select** construct:

```
 SELECT
    WHEN boolean_expression => alternative;
 OR
    WHEN boolean_expression => alternative;
 OR
    WHEN boolean_expression => alternative;
  .........
 OR
    WHEN boolean_expression => alternative
END;
```

The semantics of the guarded **select** statement is as follows:

- The guards are evaluated at the start of the execution of the **select** statement.
- An alternative is said to be open if its guard is evaluated to TRUE; otherwise it is said to be *closed*.
- Closed alternatives are ignored for the remainder of *that* execution of the **select** statement.
- An open *and* ready alternative is non-deterministically chosen for execution.
- If there is no one ready among the open alternatives, the process is suspended until some of them becomes ready.
- The guards are evaluated only once per execution of the **select** (at the beginning). They are not re-evaluated when a call comes in.
- It must be the case that at least one alternative is open in every execution of the **select**, otherwise the process would be blocked forever.

An alternative whose guard is always true is equivalent to an alternative without a guard:

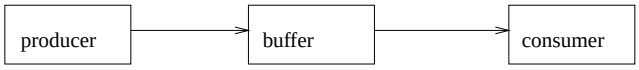
```
WHEN TRUE => alternative
```

is semantically the same as

```
alternative
```

Thus, an alternative without a guard is always open. And the basic select construct is a special case of the guarded select construct.

We take the producer-consumer problem with a bounded buffer as an example. Since the buffer is a passive object, we should represent it as a process in the message passing model. The configuration graph of the program is shown as follows:



The producer process and the consumer process are the clients processes which call the buffer process to receive and send messages respectively. They are trivial to program. We only give the solution to the buffer process below:

```
 PROCESS buffer(var put,get: channel of
integer); VAR BUFF: ARRAY[0..N] OF integer;  (*buffer size is
N+1*)
    NextIn, NextOut, CONTENTS: integer;
BEGIN
    NextIn:=0; NextOut:=0; CONTENTS:=0;
    REPEAT
       SELECT
         WHEN CONTENTS < N+1 => put ? BUFF[NextIn];
            NextIn := (NextIn+1) MOD (N+1);
            CONTENTS:=CONTENTS + 1;
       OR
         WHEN CONTENTS > 0 => get ! BUFF[NextOut];
            NextOut: NextOut:= (NextOut+1) MOD (N+1);
            CONTENTS:=CONTENTS-1
       END;
    FOREVER
END;
```

6.10 The terminate alternative

Recall that a concurrent program contains active and passive objects. With message passing, both active and passive objects are encoded as processes - generally as client processes and server processes respectively.

Active processes control their own execution and hence terminate when their internal states require them to do so. For example, the producer can stop if it does not want to produce anymore, and the consumer can stop if it does not

want to consume anymore. But the issue is more problematic with passive processes, such as the buffer process. Ideally they should terminate when "no longer needed by any active process" – a server should close down his/her business if he/she has no client to serve. If this must be programmed, then each passive process must keep track of which other processes are using it and then must be told by each of them that they guarantee never to call again.

As an example, consider the *counter* process in the Ornamental Gardens program. Previously, we had to know how many times it must loop round before terminating. This is not practical and thus represents poor language structure. But to allow *counter* to know it is no longer needed, each process has to inform the counter that it had finished:

```
 PROCESS TYPE
turnstile(name, num: integer); VAR loop:integer; BEGIN
  FOR loop:=1 TO num DO
    paths[name] ! 1;
    closedown[name] ! any
END;

PROCESS counter;
VAR count, I, J, temp:integer;
    continue: ARRAY[1..2] OF boolean;
BEGIN
  count:=0; continue[1]:=true; continue[2]:=true;
  WHILE continue[1] OR continue[2] DO
    SELECT
        paths[1] ? temp; count:=count+temp;
     OR
        paths[2] ? temp; count:=count+temp;
     OR
        closedown[1] ? any; continue[1]:=false;
     OR
        closedown[2] ? any; continue[2]:=false
   END;  (*select*)
  WRITELN('Total admitted:',count)
END;
```

With a complicated program, this is certainly difficult to achieve. It would be much nicer to programmers if the system can look after this issue. For this purpose, Pascal-FC introduces the **terminate** alternative to the **select** construct. Hence, rather than the simple channel operation:

```
in ? SomeVariable
```

the following code can be used:

```
SELECT
   in ? SomeVariable;
 OR
   TERMINATE
```

The semantics of **terminate** alternative *does not* imply that the 'owner' task will terminate if there is no out standing call. The 'owner' task only terminate when there will never be a call to come. More precisely:

The process executing the select with the **terminate** *alternative will terminate if, and only if, there are no outstanding calls and all the other processes that call are either already terminated or are waiting on a* **select** *statement with a* **terminate** *alternative.*

To understand this semantics, let us look at the possible cases when there *no process executable* in a program:

1. all processes have terminated;
2. some processes are blocked on channel operators or **selects** without the **terminate** alternative;
3. all non-terminated processes are suspended on **select** statements with **terminate** alternatives.

Condition 1 is normal termination and should be accepted. Condition 2 implies system deadlock - the program must be abandoned with error messages. The third condition is not an error; all the suspended processes can be terminated.

Let us use the **terminate** alternative to rewrite the buffer process:

```
 PROCESS buffer(var put,get: channel of integer);
VAR BUFF: ARRAY[0..N] OF integer;  (*buffer size is N+1*)
    NextIn, NextOut, CONTENTS: integer;
BEGIN
    NextIn:=0; NextOut:=0; CONTENS:=0;
    REPEAT
       SELECT
         WHEN CONTENTS < N+1 => put ? BUFF[NextIn];
            NextIn := (NextIn+1) MOD (N+1);
            CONTENTS:=CONTENTS + 1;
       OR
         WHEN CONTENTS > 0 => get ! BUFF[NextOut];
            NextOut: NextOut:= (NextOut+1) MOD (N+1);
            CONTENTS:=CONTENTS-1
       OR
         TERMINATE
       END;
    FOREVER
END;
```

If the producer and consumer both terminate, the buffer process will terminate even if there are still data in BUFF. At least this means that if the consumer has terminated too early, this error is not exacerbated by a deadlock ensuring.

With the **terminate** alternative, the code for the $counter$ process becomes much more straightforward and less error-prone:

```
PROCESS TYPE turnstile(name, num:
```

```
integer); VAR loop:integer; BEGIN
  FOR loop:=1 TO num DO
    paths[name] ! 1
END;

PROCESS counter;
VAR count, temp:integer;
BEGIN
  count:=0;
  REPEAT
    SELECT
        paths[1] ? temp; count:=count+temp;
     OR
        paths[2] ? temp; count:=count+temp;
     OR
        TERMINATE
   END;  (*select*)
FOREVER
  WRITELN('Total admitted:',count)
END;
```

However, there is a serious problem with the above $counter$ process. Although it terminates normally it does not produce any results at all. This is because the WRITELN statement is never reached. There is not, unfortunately, any opportunity for a process to execute any statement after the **select** statement if the process terminates by executing that **select**. When a server process must produce a result just before terminating, Pascal-FC requires the use of a **var** parameter to process. By making $count$ a **var** parameter its result will be returned to the main program after termination. There it can be output:

```
 ..... PROCESS counter(var count: integer);
VAR temp:integer; BEGIN
  REPEAT
    SELECT
      paths[1]?temp; count:=count+temp;
    OR
      paths[2]?temp; count:=count+temp;
    OR
      TERMINATE
   END
  FOREVER
END;
BEGIN
  number:=0;
  COBEGIN
     turnstiles[1](1,20);
     turnstiles[2](2,20);
     counter(number)
  COEND;
  WRITELN('Total admitted:',number)
END.
```

Note that $number$ is now a global variable. This trick does no apply to distributed systems which do not have global variables.

6.11 Else and timeout alternatives

The above discussion has focussed on the main uses and semantics of the selective waiting construct. There are, however, two other features that are useful in certain situations. In particular, there are applications for which a process willing to make a rendezvous on a channel may not wish to commit itself to waiting indefinitely for the partner in the communication. Two alternatives to such indefinite waiting are:

- if a rendezvous is *not immediately* possible (i.e. there is no pending call on any eligible channel), then abort the call and do something else instead;
- wait for a specified maximum period for a rendezvous, but abort the call and do something else if the period expires before such a rendezvous begins.

The **else** alternative caters for the first of these; the **timeout** alternative caters for the second. The form of a **select** with an **else** alternative is as follows:

```
SELECT
    inp ? SomeVariable;
 OR
    out! SomeValue;
 ELSE
     (*arbitrary sequence of statements*)
END
```

The semantics of it can be explained as:

- The **else** part can play the role of the default alternative.
- It is not guarded and is taken immediately if none of the other open alternatives is ready.
- A **select** statement with an **else** part can never lead to the executing process being blocked: it either takes one of the normal alternatives or it starts executing the **else** code.

The **timeout** alternative is of the following form:

```
SELECT
    inp ? SomeVariable;
 OR
    out! SomeValue;
 OR
    TIMEOUT 3;  (*arbitrary sequence of statements*)
END
```

The semantics is as follows:

- If no other open alternative is ready within three seconds (in general TIMEOUT n, where n is an integer), the **timeout** alternative is chosen.
- The alternative TIMEOUT 0 is the same as ELSE.

Final remarks on SELECT

- A **terminate** is only of use to passive processes.
- An **else** is only of use to processes that are hybrids of passive and active entities. They must respond to calls, but have actions of their own to undertake if no calls is available.
- A **timeout** is useful to real-time programs. For example, if a message has not arrived within three seconds an error message must be produced.
- It is not allowed to have mixing of these three alternatives in a single **select**.

6.12 Exercises

1. Process controlling a drinks machine accepts coins followed by a user's request; it then dispenses the drink and any change before waiting a short time and then repeating the behaviour. Assume a user can only choose tea or coffee which cost 25p and 35p respectively; and the machine only accepts 5p, 10p and 20p coins. Write the program of this controlling process which should not cause deadlock to the system.
2. In order to speed up the service of the drinks machine in the previous exercise, the process is decomposed into two concurrent processes; one dealing with the user's inputs, the other controlling the dispensing of drink and change. Write these two processes which use the synchronous message passing for communication.
3. Complete the the prime generating program $sieve$ so that the entire program terminates correctly. First use **select** without the **terminate**. Then, separately, program termination using the full Pascal-FC facilities.
4. Write a simulation of a semaphore in the synchronous message passing model. Then, use the simulation to write a program for the mutual exclusion problem.
5. Solve the Readers-Writers problem in the synchronous message passing model which has the readers have preference over writers.
6. Develop a deadlock free solution to the dining philosophers problem by using message passing. In this solution, a server process should be used to record whether or not a chopstick is being used. With this approach a philosopher would need to send to the server process two messages to request for his chopsticks and two messages to release his chopsticks.
7. Develop a deadlock free solution to the dining philosophers problem. In this solution, a server process should be used to record whether or not each philosopher is eating. With this approach, a philosopher needs to send one message to request or release both chopsticks.
8. Although the message-passing primitives described in this chapter is one-to-one, it is sometimes necessary to broadcast a value to a number of other processes. Write a server process that will enable a single client process to broadcast an integer value to a number of other clients. The server process should not dictate the order in which the receiving processes get their data. Moreover, the broadcasting process should be delayed if it wishes to send a another broadcast before all receivers have got the previous one.

7 REMOTE INVOCATION

The goals of the chapter

- Motivation for remote invocation
- Understand message passing and synchronization with remote invocation
- Understand the client-server paradigm of process interaction
- Understand non-determinism with remote invocation
- Programming techniques using remote invocation

7.1 Introduction

The synchronous message passing model studied in the previous chapter is powerful enough to program all kinds of processes: filters, clients and servers. That model can be characterized by:

- point-to-point communication – a **SEND** operation can only pass a message from a single sending process to a single receiving process,
- one-direction data flow – the taking place of a rendezvous only passes a message in one direction, i.e. from the sender to the receiver, and if a reply message (a result from the receiver) is required by the sender, another rendezvous is needed and carried out by another pair of send and receive operations.

Both of these are ideally suited to programming filters. However, the communication between a client and a server is in nature a two-way information flow between them: a client sends a message to request for a service and expecting the service to be delivered by the server. In the synchronous message passing model, this has to be programmed with two explicit message exchanges using two different message channels. Moreover, different clients may request for the same service from the same server. Then the different clients have to have rendezvous with different **RECEIVE** operations via different channels, and the server has to deliver the same service via different channels and by different **SEND** operations. All these lead to a large number of channels and message passing operations.

In this chapter, we consider a different language model, which is used in the Ada programming language. It has the following characters which differ from those of the synchronous message passing in the followin aspects:

- two-way data flow communication – the taking place of a rendezvous may pass data in both directions, i.e. message sent from the sender (or client) to the receiver (or server) and reply sent from receiver to sender.
- many-to-one communication - different **SEND** operations from different clients only have to have rendezvous with a single **RECEIVE** operation from a server if they request for the same service.

Therefore, this model is ideally suited to the client-server paradigm of process interaction. The remote invocation model combines the aspects of monitors and synchronous message passing. As with monitors, a process (usually a server) exports (owns) operations, and the operations are invoked by **call** statements from other processes (usually clients). As with the synchronous message passing, execution of SENDing a **call** delays the sender (caller) until the

call is accepted (RECEIVED). When a rendezvous takes place, a two-way communication is carried out from the caller to the process that services the called operation and then back to the caller (with the result of the operation).

7.2 The message passing primitives in the remote invocation model

The remote invocation model is still a message passing model and suited to programming distributed systems. Therefore, both active and passive objects are represented as processes. And typically, processes only share the network but not variables. This is also where the remote invocation differs from monitors. Here "remote invocation" means an operation of one process can be called by a "remote" process. The called operation can be executed only when the owner allows it to be executed and is executed by the owner. The result of the operation will be sent to the caller by the owner.

As said in Chapter 5, the first thing in writing a program for a network is to define the network interface, i.e.the **SEND** and **RECEIVE** primitives.

Entry definitions

To minimize confusion with the procedures in monitors, in the remote invocation model, a process that is prepared to accept calls defines one or more named **entries**. In Pascal-FC entries are defined or declared with the process type, as in the following two examples:

```
PROCESS P; PROCESS
TYPE Q;
   ENTRY E1;                    ENTRY E1;
   ENTRY E2;                    ENTRY E2;
    .....                       .......
   ENTRY En;                    ENTRY En;
VAR  "local declaration";      VAR  "local declaration";
BEGIN                          BEGIN
"statement sequence"           "statement sequence"
END;  (*P*)                    END; (*Q*)
```

We can view the **entries** as channels in the synchronous message passing model, except for that channels there must be of some types, and **entries here** are identifiers. Also a channel can have at most one process waiting on it but an entry can have a number of calls outstanding.

We can also view **entries** here as the **export list** in the monitors, except for that the export list there includes several procedure names.

Communication

When procedures and procedure calls are used, a procedure call can involve information passing into, or out of the procedure through the procedure parameters. This is said that data is communicated between procedures via

the parameters. This parameter-passing model used for procedure calls is reused in our entry definition. Hence an **entry** can have any number of (**formal**) parameters; each parameter has a type and is either passed by value (value parameter) or is defined to be a variable (variable parameter). Value parameters are used to allow data to be communicated to the owner of the entry, and variable parameters are used to allow data to be communicated back to the calling process. The following are example declarations:

```
ENTRY place(c: char);  (*value parameter*)
ENTRY take(var c: char); (*variable parameter*); ENTRY
count(i:integer); ENTRY exchange(Indata: integer; var:
OutData:integer); ENTRY lost(A,B:integer; var C,D:integer; E:
boolean);
```

Entry call- the SEND operation

When two process P and Q communicate with each through an entry, the receiving process is the process, say P, which owns (defines) the entry and the sending process is the caller of the entry. The **SEND** operation in Q, i.e. the call of an entry ENTRY E[formal-part] by Q, is of the form

```
   P.E[actual-part]
```

An entry must be declared before called, i.e. the entry must be declared before the calling processes are declared.

Forward declaration

As we have just said that an entry must be declared before the calling processes are declared. However, there are applications in which two process call each other. For this Pascal-FC supports *forward declarations*:

```
PROCESS P
PROVIDES
   ENTRY E;
END;
PROCESS Q;
   ENTRY F;
BEGIN
   ....
   P.E;
   ....
END;
PROCESS P;
   ENTRY E;
BEGIN
   ....
   Q.F;
```

```
   ....
END;
```

A common use of this facility is when we have an array (or pipeline) of processes that wish to call each other, e.g. a process i needs to call process $i + 1$. The following code gives the structure of this type of program:

```
PROCESS TYPE ELEM(NUM:integer) PROVIDES
   ENTRY CALL;
END;

VAR
  ELEMENTS : ARRAY[1..PIPELINE] OF ELEM;

PROCESS TYPE ELEM(NUM:integer);
  ENTRY CALL;
BEGIN
  ....
  IF NUM<> PIPELINE THEN
     ELEMENTS[NUM+1].CALL;
  ....
END;
```

Accept operation – RECEIVE operation

For each **entry** defined, the process must embody *at least* one **accept** statement that corresponds to that entry and which defines the code to be executed when an entry call is accepted (received). An **accept** statement corresponding an entry must have the same name and same parameter profile of that entry:

```
  ACCEPT entry_name[formal-part] DO
     statement
```

The typical function of an **accept** statement is to save the value of any input data passed from the caller and to undertake whatever actions are needed to generate any defined data. For example:

```
PROCESS P;
  ENTRY exchange(Indata: integer; var OutData: integer);
VAR D1,D2; integer;
BEGIN
  ........
  D2:=SomeValue;
  ACCEPT exchange(InData: integer; var OutData:integer) DO
  BEGIN
    D1:=InData;
    OutData:=D2
  END;
```

```
  ......
END;
```

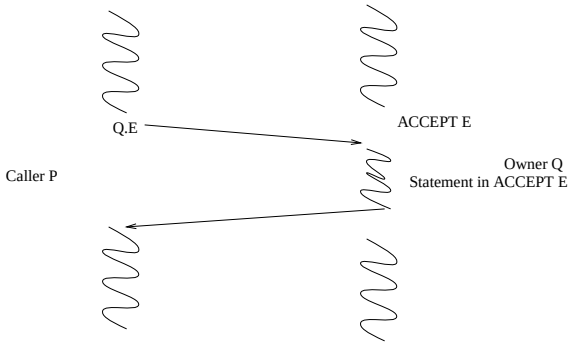
Note that an **accept** for an entry E1 may be nested inside an **accept** for an entry E2, but not within an **accept** for E1 itself (either directly or indirectly).

Synchronization

Two communicating processes are synchronized at the **entry call** and the corresponding **accept** statement:

- A process that attempts an **accept** for an entry will be "suspended" if there is no pending calls.
- A process blocked at an **accept** may become executable when another process makes a call on that entry.
- A process that attempts an **accept** for which there is pending call will start to execute the statement in the **accept** statement.
- A process that makes a call on an entry will be "suspended" on that entry if there no pending **accept**.
- If there are more than one process calling an entry at the same time, one of the *call*s will be non-deterministically chosen to be accepted by the pending **accept** and the other callers are suspended.
- A process that makes a call on an entry for which there is a pending **accept** becomes "suspended" on that entry following the transfer of parameters to the called process and then unblocking of the process suspended at the **accept**
- A process suspended at an entry call may become executable following the completion of an **accept** statement for that entry. (including the transfer of **var** parameters back to the caller.

It is called extended rendezvous message passing because that the caller is blocked until the reply is sent by the called process. This semantics is illustrated by the following figure:



wher the squiggly lines indicate when a process is executing.

Example - Ornamental gardens

As a first example, consider the Ornamental Gardens program. With remote invocation, a **select** construct is not needed, as all the turnstiles can call in on the same entry:

```
 PROGRAM gardens6; PROCESS counter;
   ENTRY path(temp: integer);
VAR count,I: integer;
BEGIN
  count:=0
  FOR I:=1 TO 40 DO
     ACCEPT path(temp:integer) DO (*one accept each iteration*)
        count:=count+temp;
  WRITELN('Total admitted:',count)
END;
PROCESS TYPE turnstil(name,num:integer);
VAR loop:integer;
BEGIN
  FOR loop:=1 TO num DO
     counter.path(1)
END;
VAR turnstiles: ARRAY[1..2] OF turnstile;
BEGIN
  COBEGIN
    counter; turnstiles[1](1,20); turnstiles[2](2,20)
  COEND
END.
```

It is important to note that the code in an **accept** statement can be arbitrarily complex. Indeed it can involve an entry call to another process:

```
 PROGRAM complex; PROCESS
P;
  ENTRY E(var I: integer);
VAR T: integer;
BEGIN
  T:=42;
  ACCEPT E(var I:integer) DO
      I:=T
END;
PROCESS Q;
  ENTRY E(var I: integer);  (*same name can be *)
                            (*used in different processes*)
VAR T: integer;
BEGIN
   ACCEPT E(var I: integer) DO
     BEGIN
       P.E(T); (*while accepting E, a call to P.E is made*)
```

```
      I:=T    (*42 is now passed to I from from P via T*)
    END;
PROCESS S;
VAR T: integer;
BEGIN
   Q.E(T)      (*42 is passed to T of S*)
END;

BEGIN
  COBEGIN P;Q;S COEND
END.
```

At the other extreme, an **accept** can be entirely empty, *being used for synchronization only*:

```
   ACCEPT START DO NULL;
   .......
```

It is necessary to consider the behaviour of **var** parameters as they are used in data communication. Consider the following calling relationship:

```
PROCESS Q;
   ENTRY GET(var I: integer);
VAR
   Store: integer;
BEGIN
 ....
 ACCEPT GET(var I: integer) DO
    I:= Store;
  .....
END;
PROCESS P;
VAR Local: integer;
BEGIN
  ...
  Q.GET(Local);
  ...
END;
```

Note that when process P calls process Q, the effect is for information to flow from Q to P, i.e. the direction of the communication is the opposite of the direction of call. This can be confusing when using terms like ‘message passing’, ‘sender’ and ‘receiver’. Hence, the term ‘remote invocation’ is preferred.

7.3 Selective waiting with remote invocation

There is always a need for a selective wait construct with message-passing semantics. This is certainly true for remote invocation as well. Consider the case that a process P defines two entries which can be called by the

environment, i.e. other processes, non-deterministically. P should accept whichever call that comes first.

The **selective** wait construct in the remote invocation model takes the form of a (possibly guarded) **accept** alternative to a **select** statement. This behaves in an equivalent way to the channel alternatives described in the previous chapter. As an example, consider a passive process, the role of which is to protect access to a single shared variable. This variable can be written to or read from (by other processes):

```
 PROCESS share;
  ENTRY read(var I: integer);
  ENTRY write(I: integer);
VAR value: integer;
BEGIN
  ACCEPT write(I:integer) DO
     value:=I;
  REPEAT
     SELECT
        ACCEPT write(I:integer) DO
           value:=I;
      OR
        ACCEPT read(var I: integer) DO
           I:=value
     END
  FOREVER
END;
```

There are two important points that even this simple example illustrates:

- First, the body of a process can have more than one **accept** statement for an entry. In the particular code above, the first **accept** for $write$ ensures that an initial value is assigned to the variable. After this is done the process loops around and accepts (in any order) calls to $read$ or $write$. In general an entry can have any number of **accept**s associated with it. Moreover, the codes executed the **accept**s need not be the same. This becomes easier to understand if we relate an **accept** for an entry to a **receive** operation for a channel in Chapter 5: a process can have any number of **receive** operations on the same channel and the code executed following one **receive** operation on the channel need not to be the same as the code folloing another **receive** operation on the same channel.
- The second point is the reversal of direction of call associated with the $read$ operation. As data can flow in the opposite direction of the call, the $read$ request is also programmed as a 'call-in'. This has a number of advantages:
  - an asymmetric **select** is adequate (i.e. selecting between **accepts** only rather than between both **accept**s and entry calls);
  - it is more in keeping with the abstraction of a passive entity. (a server process never calls out, it just accepts incoming calls.)

  To illustrate these two points, it is better to compare the above example program with a version by using the synchronous message-passing model:

  ```
  PROCESS ShareSM;
  VAR read, write: CHANNEL OF integer;
  ```

```
        value, I: integer;
    BEGIN
        write ? In;
        value:=In;
        REPEAT
           SELECT
              write ? In; value:=In; (*receive In from *)
            OR                       (*a writer then write*)
              read ! value           (*send value to a reader*)
            END
        FOREVER
    END;
```

The basic guarded form of a **select** statement is thus:

```
   SELECT
        WHEN B1 => ACCEPT E1 DO S1;
     OR
        WHEN B2 => ACCEPT E2 DO S2;
        ............
     OR
        WHEN Bn => ACCEPT En DO Sn
   END
```

In addition to these **accept** alternatives, **timeout**, **else** and **terminate** alternatives are all valid.

The semantics of this statement is the same as the select statement in the previous chapter.

7.4 Examples

7.4.1 Resource allocation problem

Consider a resource controller that allocates a collection of 16 objects. Clients obtain one of these objects by calling $ALLOCATE$; the object is returned by calling $REPLACE$. The following gives an outline to the structure of the $CONTROLLER$. Omitted from the code are the details of how the objects are passed out to the client (and returned). Rather, a simple integer is returned that indicates which object number has been allocated:

```
PROGRAM control; PROCESS CONTROLLER;
   ENTRY ALLOCATE(var num: integer);
   ENTRY REPLACE(num: integer);
CONST size = 16;
VAR objectfree: ARRAY[1..size] Of boolean;
    I: integer;
    freecount: integer;   (*number of free objects*)
PROCEDURE GetObj(var ob:integer); (*find the first free object*)
```

```
VAR j:integer; success: boolean;
BEGIN
  j:=1; success := false;
  WHILE j<= size DO
      IF objectfree[j] and not success THEN
         BEGIN
           objectfree[j]:=false;
           ob:=j;
           success:=true      (*terminate search*)
         END else j:=j+1;
       IF j= size+1 THEN   (*no free object found*)
         writeln('Error on GetObj')
END;

BEGIN  (*process controller*)
  FOR I:= 1 TO size DO
      objectfree[I]:=true;
  freecount:=size;
  REPEAT
    SELECT
       WHEN freecount>0 =>
       ACCEPT ALLOCATE(var num:integer) DO
       BEGIN
          GetObj(num);
          freecount:=freecount-1
       END;
     OR
       ACCEPT REPLACE(num:integer) DO
       BEGIN
          freecount:=freecount+1;
          objectfree[num]:=true
       END;
     OR
       TERMINATE
    END    (*select*)
 FOREVER
END;  (*CONTROLLER*)
PROCESS TYPE CLIENT;   (*example client structure*)
VAR
  ob: integer;
  I: integer;
BEGIN
  FOR I:=1 TO 10 DO
  BEGIN
     CONTROLLER.ALLOCATE(ob);
     sleep(random(5));   (*use resource*)
     CONTROLLER.REPLACE(ob)
  END
END;
VAR
  CLIENTS : ARRAY[1..20] OF CLIENT;
```

```
  ploop: integer;
BEGIN    (*main program*)
  COBEGIN
    CONTROLLER;
    FOR ploop:=1 TO 20 DO
       CLIENTS[ploop]
   COEND
END.
```

With this number of clients, the $freecount$ variable soon drops to zero (which can be illustrated by adding a $write$ statement to the CONTROLLER).

A point of useful detail can be observed in this code by considering the **accept** statement for the $ALLOCATE$ entry. In general the code inside the **accept** should be the minimum required for rendezvous, so that the potential for concurrency can be fully exploited. The assignment to $freecount$ is therefore inappropriately placed; it should be done after the rendezvous, as it only affects a local variable:

```
  WHEN freecount>0 =>
  ACCEPT ALLOCATE(var num:integer) DO GetObj(num);
  freecount:=freecount-1
```

Question:

How can we change this code so that each client can request a number of objects?

7.4.2 Dining philosophers problem

Next, we are going to solve the dining philosophers problem in the remote invocation model. The solution avoid deadlock by allowing at most $N-1$ philosophers sitting at the table. Recalling the corresponding semaphore solution in Chapter 3, the following program should be easy to understand:

```
PROGRAM pilada; CONST N=5; VAR I: integer; PROCESS TYPE chopstick;
  ENTRY pickup; ENTRY putdown;
BEGIN
  REPEAT
    SELECT
         ACCEPT pickup DO NULL;
      OR ACCEPT putdown DO NULL;
      OR TERMINATE
    END
   FOREVER
END;
VAR chopsticks: ARRAY[1..N] OF chopstick;
PROCESS chairs;
```

```
   ENTRY getchair; ENTRY replacechair;
VAR freechairs: integer;
BEGIN
   freechairs:=N-1;
   REPEAT
      SELECT
           WHEN freechairs>0 => ACCEPT getchair DO null;
                freechairs:=freechairs-1;
        OR ACCEPT replacechair DO null; freechairs:=freechairs+1;
        OR TERMINATE
       END
   FOREVER
END;
PROCESS TYPE philosopher(name:integer);
VAR I:integer;
    chop1, chop2: integer;
BEGIN
  chop1:=name;
  IF name =N THEN chop2:=1 ELSE chop2:=name+1;
  FOR I:=1 TO 10 DO
  BEGIN
    sleep(random(5));  (*thinking*)
    chairs.getchair;
    chopsticks[chop1].pickup;
    chopsticks[chop2].pickup;
    sleep(random(5));
    chopsticks[chop1].putdown;
    chopsticks[chop2].putdown;
    chairs.replacechair
   END
END;
VAR phils: ARRAY[1..N} of philosopher;
BEGIN
   COBEGIN
      FOR I:=1 TO N DO
      BEGIN chopsticks[I}; phils[I](I) END;
      chairs
    COEND
END.
```

7.4.3 Process idioms

An important distinction has been made in this book between active and passive processes. In general, passive processes can be classified into a number of process idioms. A partial list is as follows:

1. buffer
2. stack
3. mailbox

4. forwarder.

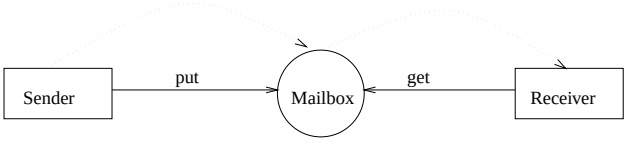
Buffer processes have already been considered. A stack process merely implement LIFO (Last In First Out) data store. The other two idioms introduce processes that act on behalf of the other entities. Each will now be considered.

Mailbox

A mailbox is a process that acts as a temporary buffer between two other processes. It allows the active processes to pass data asynchronously (via the mailbox). The mailbox type simply has two entries for capturing and releasing the message (which is of type $integer$ in the following example):

```
PROCESS TYPE MAILBOX;
   ENTRY PUT(Mess:integer);
   ENTRY GET(VAR Mess: integer);
VAR
  temp:integer;
BEGIN
   REPEAT
     SELECT
         ACCEPT PUT(Mess: integer) DO
         temp:=Mess;
         ACCEPT GET(var Mess: integer) DO Mess:=temp;
       OR
         TERMINATE
     END
   FOREVER
END;
```

The users of this process simply call the appropriate entry. Note that if a message is still 'in' the mail box, then a subsequent call of PUT will block. This program is illustrated in the following figure:



where the dashed arrowed lines represent the directions of the data flows, while the solid arrowed line represent the directions of entry calls.

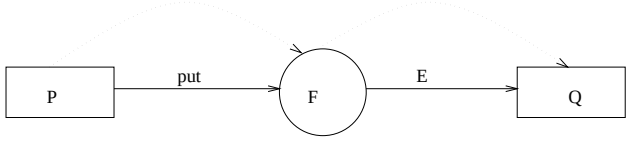
Forwarder

A forwarder acts as a direct intermediary between two processes. Consider a program in which process P calls ENTRY F in process Q, and deposits an integer. If P calls Q directly it will be blocked if Q is not able to accept

the call immediately. This synchronous communication can be altered by P using an agent that calls Q on its behalf, so that P and Q become less tightly coupled. The following code outline illustrates this:

```
PROGRAM USEFORWARDER; PROCESS Q;
   ENTRY E(mess: integer);
BEGIN
   ....
END;
PROCESS TYPE FORWARDER;
   ENTRY PUT(Mess:integer);
VAR
   temp:integer;
BEGIN
   REPEAT
     SELECT
         ACCEPT PUT(Mess:integer) DO
            temp:=Mess;
         Q.E(tem);
      OR
         TERMINATE
      END
    FOREVER
END;
VAR F, ...: FORWARDER;
PROCESS P;
VAR V: integer;
......
BEGIN
  ....
  F.PUT(V);
  .....
END;
```

This program is illustrated in the following figure:



7.4.4 A reactive example - a controlling system

Many concurrent systems are driven by the arrival of data at the system's interface. These systems are often called *reactive*. In the following example an output value (usually a controlling instruction) is produced in response to a new input value. There are three sources of input which are intended to be three alternative measures of the same environmental quantity: for example, they might come from three different devices (usually called *sensors*) that

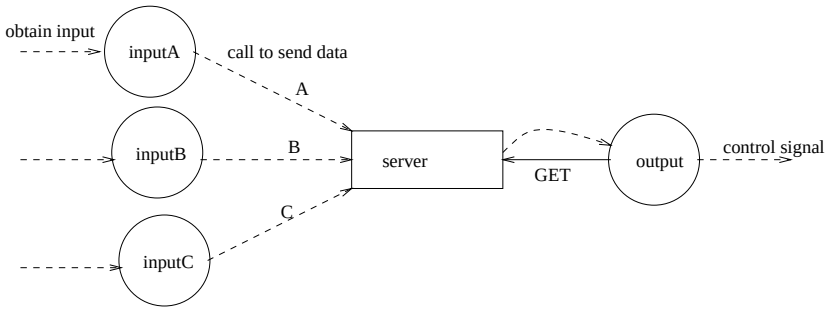
estimate (or sensing) altitude in an aeroplane. As each of the sensors might give different data representations, the program uses three distinct processes for input (i.e. sensors) even though only integers are passed in the example. Whenever a new input value is registered, the output process can run, get a new weighted average of the inputs and produce an appropriate output. However, it is important that if a number of new input arrive before the outputting process (controller) gets around to reading the input values, then it is given an up-to-date average; some data may be lost but the average always uses the latest values from the three sensors.

One way to accommodate the freshness requirement (and to minimizing the number of times the averaging activity is performed) is to call the $average$ procedure only when a request for a data is made. This requires the procedure to be called from within the **accept** statement, thus illustrating why it is often useful to be able to execute code within an accept body:

```
 PROGRAM control; PROCESS server;
  ENTRY A(I:integer);
  ENTRY B(I:integer);
  ENTRY C(I:integer);
  ENTRY GET(var I: integer);
VAR newvalue: boolean;
    tempA, tempB, tempC:integer;
PROCEDURE average(X,Y,Z:integer; var result:integer);
BEGIN
  (*form a weighed average of the input values*)
END;
BEGIN
  newvalue:=false;
  REPEAT
    SELECT
         ACCEPT A(I:integer) DO tempA:=I;
         newvalue:=true;
     OR
         ACCEPT B(I:integer) DO tempB:=I;
         new value:=true;
     OR
         ACCEPT C(I:integer) DO tempC:=I;
         newvalue:=true;
     OR
         WHEN newvalue => ACCEPT GET(var I: integer) DO
              average(tempA,tempB,tempC, I);
         newvalue:=false;
     OR
         TERMINATE
    END
  FOREVER
END;
PROCESS inputA;
VAR V: integer;
BEGIN
   REPEAT
      (*wait for input*)
      (*obtain input value V*)
```

```
      server.A(V)    (*send V to server*)
    FOREVER
END;
PROCESS inputB;
VAR V: integer;
BEGIN
   REPEAT
      (*wait for input*)
      (*obtain input value V*)
      server.B(V)    (*send V to server*)
    FOREVER
END;
PROCESS inputC;
VAR V: integer;
BEGIN
   REPEAT
      (*wait for input*)
      (*obtain input value V*)
      server.C(V)    (*send V to server*)
    FOREVER
END;
PROCESS output;     (*controller*)
VAR V: integer;
BEGIN
   REPEAT
      server.GET(V); (*get the average*)
      (*Use V to produce output controlling command*)
   FOREVER
END;
BEGIN (*main*)
   COBEGIN
      inputA;
      inputB;
      inputC;
      output;
      server
    COEND
END.
```

This program can be illustrated by the following figure:



7.5 The limitation of the Pascal-FC select construct

Earlier in this chapter a program was given that show hows a resource $CONTROLLER$ process can control a set of objects that are needed by a group of competing clients. The program restricted each client to obtaining only one object at a time and the $ALLOCATE$ operation was thus programmed as a single entry. If we now extend the problem to allow each client to ask for a *variable* number of objects, the guarded select structure (as described so far) is inadequate. Let us now show this limitation step-by-step.

First we have to extend the $GetObj$ procedure used in the $CONTROLLER$ process to find the first number of free objects that are requested by a client; and add a $ReturnObjs$ procedure to replace the objects returned by a client:

```
PROGRAM control; CONST size=16; TYPE
OBjectFlags=ARRAY[1..size] OF boolean; PROCESS CONTROLLER;
.......; PROCEDURE GetObjs(number: integer; var: ObF:ObjectFlags);
   (*find the first number free objects*)
VAR J, found:integer;
BEGIN
  j:=1; found:=0;
  WHILE j<= size DO
     BEGIN
       IF objectfree[j] THEN
         BEGIN
           objectfree[j]:=false;
           ObF[j]:=true;         (* jth object is allocated*)
           found:=found+1;
           IF found = number THEN
              j:=size+1          (*terminate search*)
         END;
       j:=j+1
      END;
   IF j=size+1 THEN WRITELN('Error on GetObjs')
END;
PROCEDURE ReturnObjs(VAR ObF:ObjectFlags);
VAR j:integer;
BEGIN
   FOR j:=1 TO size DO
```

```
      IF ObF[j] THEN
         BEGIN
            ObF[j]:=false;
            objectfree[j]:=true
         END
END;
.......
```

After the extension with these two procedures, we might be thinking the body of the $CONTROLLER$ process should be similar to that of the earlier solution:

```
BEGIN   (*body of CONTROLLER*)
  FOR i:=1 to size DO
     objectfree[i]:=true;
     freecount:=size;
   REPEAT
      SELECT
          WHEN freecount >= number =>
          ACCEPT ALLOCATE(number: integer; var ObF: ObjectFlags) DO
          BEGIN
            GetObjs(number,ObF);
             freecount:=freecount-number
           END;
         OR
           ACCEPT REPLACE(number:integer; var ObF:ObjectFlag) DO
           BEGIN
              ReturnObjs(ObF);
              freecount:=freecount+number
            END;
           OR
             TERMINATE
       END
    FOREVER
END;
```

This certainly will not work except for the case when $number$ in the $ALLOCATE$ entry is a constant and each clients can request for only $number$ objects at a time. To allow a client to obtain a variable $number$ of objects, the $CONTROLLER$ must accept the call to $ALLOCATE$ first and then check whether there are enough free objects. Therefore, it is better to have the select statement in the body of $CONTROLLER$ as follows:

```
SELECT
    ACCEPT ALLOCATE(number: integer; var ObF: ObjectFlags)
    WHEN freecount >= number DO
    BEGIN
      GetObjs(number,ObF);
      freecount:=freecount-number
    END;
 OR
```

```
    ACCEPT REPLACE(number:integer; var ObF:ObjectFlag) DO
    BEGIN
      ReturnObjs(ObF);
      freecount:=freecount+number
    END;
 OR
    TERMINATE
END;
```

The semantics of the guarded alternative should be such that the calling process will be locked if the guard does not hold; and the guard should be re-evaluated each time a call is made on the process with the **select**. This, of course increases the run-time complexity. This is supported in the language SR (Andrews, 1981) but not in Pascal-FC.

Section 5.9 in the text book presents a solution to this problem which uses the current Pascal-FC model.

7.6 Exercises

1. Simulate the operations on a channel by ACCEPT and CALL on an entry (or entries). And simulate ACCEPT and CALL on an entry by channel operations.
2. Program the readers and writers problem using the remote invocation model.
3. Write a Pascal-FC program that emulates a simple terminal handler. One process should read characters from'input'. Lower case characters are the normal ones; upper case letters represent control characters. Each normal character must be passed onto a buffer process before another is read. Four control characters should be recognized and acted upon (others should be ignored):

   B - erase last character from the buffer

   L - put end-of-line marker into buffer

   U - remove current incomplete line from buffer

   C - remove content of buffer

   A third process should be coded to read lines of characters from the buffer. This process should then 'output' these lines. The buffer process should only allow the reader process to access complete lines of characters. You can assume a line will have a maximum of 20 characters.
4. Write a program that implements a lift (elevator) control system. A server process accepts calls on floor buttons and moves the lift to the requesting floor. The lift is very small and so can only take a single person at a time. In the lift are buttons that allow the passenger to choose the destination floor. The program should contain a number of passenger processes that make calls on the lift.
5. Modify the previous example so that thee are now m lifts (each only carrying a single person).
6. Consider a collection of n processes. Each has a unique identity and an arbitrary integer value (in the range 1 to 5, say), By communicating with every other process, each process is able to find out how many other processes have the same integer value. Construct a program that implements these communication (without deadlock!).

8 SUMMING UP

8.1 The course

This is a course on basic *concepts, problems, principles, techniques and tools* in *concurrent programming*.

8.1.1 Concepts

Concepts includes: *parallel computation, parallel computing systems, concurrent computer systems, distributed computing systems, concurrent programming, non-determinism, interleaving, concurrency, synchronization, communication (interaction), safety property, liveness property, deadlock, livelock, starvation, blocking waiting (or blocking primitives)*.

8.1.2 Problems an Priciples

We have considered the problems of *multiple updating, critical section and mutual exclusion, producer and consumer, readers and writers, the dining philosophers, resource allocation, and automatic control system*.

These problems and their various solutions are abstractions of real application problems in operating systems, concurrent computation applications, and application of automatic control engineering. They provide general patterns and principles in solving classes of concurrent program problems. Students should learn to use these abstract problems and solutions to solve concrete application problems.

The principles also include how to represent active and passive entities, especially passive entities, in different programming models; how to program mutual exclusion synchronization and condition synchronization in different models.

8.1.3 Techniques and tools

We discussed 6 basic tools and the techniques of using these tools in concurrent programming:

- Busy waiting techniques. These are the old-fashioned techniques in solving synchronization problems.
- Semaphores: These are the first and most fundamental abstract primitives for synchronization used in shared memory based systems. They are very easy to implement efficiently. And they are still widely used in the design of operating systems.
- CCRs and Monitors: They are more abstract than semaphores and easier and safer to use as well. They are popular in the design of operating system and of concurrent programs for client/server paradigm of process interactions on shared memory based systems.
- Synchronous message passing: This is a popular model for theoretical study of concurrent and distributed system (Hoare's CSP, 1985, Milner's CCS, 1989). The model is simple since both active and passive entities

are modelled as processes. Practically, it is a good model for programming networks of filter processes (pipelined systems). Occam (INMOS, 1984) uses this model. Thus, students who have taken this course should have no problem in writing occam programs after reading the language manual.

- Remote invocation: This is a message passing model which is ideally suited to programming the client/server paradigm of process interaction for distributed systems. Ada uses this model. Therefore, the students who have taken this course should have no problem in writing Ada programs after reading the language manual.

We have discussed the language notations and their semantics for each of these models, the techniques in programming mutual exclusion and condition synchronization in these five models, and the ways in which passive entities are represented.

8.2 Possible application of the course

- Programming parallel computation and distributed computation.
- Design and use multitasking operating systems: Many of the problems and solution of these course were from the design of operating systems.
- Design automatic control system – reactive system.
- Design programming language constructs (primitives) for concurrent programming, even a full concurrent programming language. When we introduced a new language facility, we first discussed its motivations, then introduced its notation (syntax), then defined its semantics, and finally discussed how the semantics is implemented. This is the general routine followed by a language designer when he/she designs (or extends) a language.
- Evaluate concurrent programming languages, parallel algorithms, and concurrent programs. Most aspects of a language these are about the expressiveness, semantics, implementation, easy-to-use, and safe to use, and the trade-off between these aspects. Concurrent algorithms and programs are main concerned with their properties (correctness, efficiency, etc.).

8.3 Furthermore ...

This course also provides the backgrounds for the study of real-time and fault-tolerant systems, distributed (communicating) networks.

8.4 What cannot we do?

This course has covered little on reasoning about concurrent programs. We have gained the feeling that this will not be a trivial task. Also, the techniques or methods used here in developing a concurrent program are rather *ad hoc*. So we have the following questions:

- How can we obtain systematic guidelines for the development of a provably correct concurrent program?
- How can we systematically reason about the correctness of a concurrent program?

Answering to these two questions has formed a field called formal methods in concurrent and distributed systems, which is currently one of most exciting research areas in computer science, and which has produced and is still producing many big names in the computer science community.

Comments and corrections on the lecture notes are and will be very much appreciated.