

Principles, Algorithms and Models of Concurrency and Synchronization

Summer term 2026

Prof. Dr. David Sabel

Last update: July 7, 2026

Theoretische Informatik
Fachbereich DCSM
Hochschule RheinMain
Unter den Eichen 5
65195 Wiesbaden

Email: david.sabel@hs-rm.de

Contents

1	Introduction	1
1.1	Why Concurrent Programming Matters	1
1.2	Terms of Concurrent Programming	3
1.3	Planned Contents	4
1.4	Literature	5
1.5	Model Assumptions	6
1.5.1	The Fairness Assumption	7
1.5.2	Known Processes and Atomic Actions	8
2	Synchronization	9
2.1	The Mutual Exclusion Problem	9
2.1.1	Correctness Criteria	13
2.2	Mutual Exclusion Algorithms for Two Processes	14
2.2.1	Dekker's Algorithm	14
2.2.2	Peterson's Algorithm	16
2.2.3	Kessels' Algorithm	17
2.3	Mutual Exclusion Algorithms for n Processes	18
2.3.1	Lamport's Fast Algorithm	19
2.3.2	The Bakery Algorithm	23
2.4	Three Complexity Results for the Mutual Exclusion Problem	29
2.5	Stronger Memory Operations	33
2.5.1	Mutual Exclusion with Test-and-Set Bits	37
2.5.2	A Mutual Exclusion Algorithm with RMW Object	43
2.5.3	The MCS Algorithm with Queue	43
2.6	Consensus and the Herlihy Hierarchy	49
2.6.1	Process Model with Crashes	49
2.6.2	The Consensus Problem	50
2.6.3	The Consensus Number	54
2.7	References	55
3	Programming Primitives	56
3.1	Extensions of the Process Model	56
3.2	Semaphores	57
3.2.1	Mutual Exclusion Using Semaphores	59
3.2.2	Further Variants of Semaphores	61
3.3	Application Examples for Semaphores	62
3.3.1	Coordination of Parallel Merge Sort	62
3.3.2	Producer-Consumer Problems	63
3.3.3	The Dining Philosophers	65
3.3.4	The Sleeping Barber Problem	68
3.3.5	The Cigarette Smoker's Problem	71

3.3.6	Barriers	75
3.3.6.1	An example application: Conway's Game of Life	77
3.3.7	The Readers & Writers Problem	79
3.4	Monitors	84
3.4.1	Monitors with Condition Variables	86
3.4.1.1	Monitor solution for the producer/consumer problem	89
3.4.2	Different Types of Monitors	90
3.4.3	Monitors with Condition Expressions	92
3.5	Some Application Examples with Monitors	94
3.5.1	The Readers & Writers Problem	94
3.5.2	The Dining Philosophers	94
3.5.3	The Sleeping Barber Problem	97
3.5.4	Barriers	97
3.6	Channels	99
3.6.1	Definition of Channels	100
3.6.2	Mutual Exclusion with Channels	101
3.6.3	Modeling Shared Memory Using Channels	102
3.6.4	Selective Input	104
3.6.5	Dining Philosophers with Channels	104
3.7	References	105
4	Access to Multiple Resources	106
4.1	Deadlocks with Multiple Resources	106
4.2	Deadlock Prevention	108
4.3	Deadlock Avoidance	110
4.4	Transactional Memory	115
4.4.1	Basic Primitives for Transactional Memory	117
4.4.1.1	Atomic Blocks	117
4.4.1.2	The abort command	118
4.4.1.3	The retry command	118
4.4.2	The orElse Command	119
4.4.3	Properties of TM Systems	120
4.4.3.1	Weak / Strong Isolation	120
4.4.3.2	Handling of nested transactions	120
4.4.3.3	Granularity	121
4.4.3.4	Direct and delayed update	121
4.4.3.5	Time of conflict detection	121
4.4.3.6	Conflict management	121
4.4.4	Correctness Criteria for STM Systems	121
4.4.5	The TL2 Algorithm	130
4.4.6	Conclusion	133
4.5	References	134

5	The π-Calculus: Mobile Processes with Message-Passing	135
5.1	The Lambda Calculus	136
5.1.1	Call-by-Name Evaluation	138
5.1.2	Equality of programs	139
5.2	A message-passing model: The π -calculus	140
5.2.1	The synchronous π -calculus	140
5.2.1.1	Syntax of the synchronous π -calculus	140
5.2.1.2	Operational semantics of the synchronous π -calculus . .	143
5.2.1.3	Turing power of the synchronous π -calculus	144
5.2.2	The asynchronous π -calculus	146
5.2.2.1	Encodability of the synchronous π -calculus in the asyn- chronous π -calculus	147
5.2.3	Extensions and Variants of the π -Calculus	149
5.2.3.1	Non-Deterministic Selection	149
5.2.3.2	Polyadic π -calculus	150
5.2.3.3	Recursive Definitions	152
5.2.4	Process Equality in π -Calculus	153
5.2.4.1	Strong Bisimulation	154
5.2.5	Weak Bisimulation	156
5.2.5.1	Barbed congruence	157
5.2.5.2	May- and Must-Testing	158
5.3	References	159
	Bibliography	160

1

Introduction

In this chapter, we motivate why concurrency and concurrent programming play an important role in computer science. We introduce some basic terms and assumptions, and provide a brief overview of the lecture content along with some literature recommendations.

1.1 Why Concurrent Programming Matters

Programs, as covered in the first semesters of study, are sequential programs: at least at the machine code level, they consist of a sequence of (machine) instructions executed one after the other, thereby changing the state of the computer (e.g., the main memory and the registers).

Concurrent or parallel programs that execute several instructions (quasi-)simultaneously, on the other hand, are rarely covered in introductory courses. Nevertheless, there are good reasons to learn about such programs and the associated programming methods and abstractions.

A first motivation is that current real systems are almost never purely sequential. For example, operating systems perform different tasks simultaneously from the user's point of view: a document can be printed while browsing web pages, moving the mouse, and listening to music. All these tasks are carried out concurrently (at least from the user's perspective) and require concurrent programming. Similarly, applications with a graphical user interface require concurrency to enable the simultaneous execution of various tasks (window updating, input processing, etc.).

Concurrent programs can even be faster than purely sequential programs when only one processor is available, because they can better utilize the system. As a simple example, consider a program with two tasks: writing a large file to a slow tape drive and performing large scientific calculations. A sequential program must perform both tasks one after the other, leading to poor processor utilization, since the processor sits idle while waiting for data to be written to the tape. A concurrent program can be faster: while the write operations are taking place, another process can perform the time-consuming calculations.

Another area is web programming. In the client-server model, the server provides services for clients (e.g., retrieving a web page). The server must be able to serve several clients virtually simultaneously—for example, it should not stop communicating with other clients if one client loses its connection—which requires concurrent programming.

Another area of application for concurrent or parallel programming are problems whose instances can only be solved by parallel or distributed computation on many computers (e.g., using GRID computing).

Finally, concurrent programming is becoming increasingly important due to developments in hardware design. Since clock frequencies have reached their physical limits, chip manufacturers have switched to integrating multiple processors into a single system; the trend towards multi-processor and multi-core systems is likely to continue.

In order to use these systems to their advantage, concurrent programming is indispensable.¹

It remains to be clarified what distinguishes concurrent programming from conventional programming, and why concurrent programming is ‘harder’ than sequential programming. On the one hand, it is often not obvious how to parallelize a problem with a known sequential solution (e.g., parallel sorting is non-trivial, though generally considered one of the easier problems to parallelize). On the other hand, some problems are inherently concurrent in nature, such as an operating system, where individual tasks must run in a quasi-parallel fashion. An implementation in a purely sequential language would have to simulate concurrency and would therefore likely be harder to write than a direct implementation in a language with concurrent programming support.

However, the main difficulty of concurrent programming lies elsewhere: the individual processes must communicate with each other (exchange data) or synchronize (wait for each other). This is where most programming errors occur. Incorrect synchronization—or even a lack of synchronization—leads to errors, particularly when shared resources are accessed. As an example, consider a memory location that represents the balance of a bank account. If two processes access the account balance (e.g., one makes a debit, the other a credit), if the programming is incorrect (in this case naive), the account balance can be given an incorrect value after both actions:

(The value of account before execution is 100)

process P:

(P1) X:= account;
(P2) account:= X-10;

process Q:

(Q1) X := account;
(Q2) account := X+10;

Executing the commands in the order (P1),(P2),(Q1),(Q2) gives the desired result (the value of account is 100), just like the sequence (Q1),(Q2),(P1),(P2). However, the sequences (P1),(Q1),(P2),(Q2) and (Q1),(P1),(P2),(Q2) result in an account balance that is too high (the value of account is 110 at the end), while the sequences (P1),(Q1),(Q2),(P2) and (Q1),(P1),(Q2),(P2) result in an account balance that is too low (the value of account is 90).

¹An interesting article on this is

“The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”

by Herb Sutter (to be found at <http://www.gotw.ca/publications/concurrency-ddj.htm>), who calls for a paradigm shift for programmers and programming languages due to the trend towards multiprocessor systems.

The programs should be modified so that only the first two sequences are possible, while the other execution sequences are prohibited. One should also consider how many processes can access the same resource simultaneously: the example involves two processes, but the solution should work for any number of processes.

This simple example also illustrates that traditional debugging and testing are nearly impossible for concurrent programs, since the erroneous results cannot be reliably reproduced. For instance, the error may not appear during debugging if only the first two execution sequences are executed. It is not uncommon for an unwanted sequence not to occur on computer A with compiler C_A , while it does occur on computer B with compiler C_B . Therefore, testing alone is not sufficient; correctness must be established for all possible sequences, independent of the computer and compiler.

Another problem of concurrency is the organization of data exchange. For example, if a ‘producer’ process sends results to a ‘consumer’ process, the producer may calculate results much faster than the consumer can process them. In this case, the producer should be able to start computing the next result as soon as the current one is ready.

For this purpose, a buffer is required in which the producer can store results. Access to this buffer must be protected to prevent inconsistent memory states. Concurrent programming deals with such structures.

1.2 Terms of Concurrent Programming

In this section, some basic terms from the field of concurrent programming are defined, explained, and distinguished from each other.

Parallelism and concurrency In general, *parallel programs* are programs executed on multiple processors simultaneously, so that individual computations overlap. In concurrent programming, parallel execution is only one possible scenario; instead, it is assumed that computations may overlap but need not. Concurrent systems do not necessarily require multiple processors; they can also run on a single processor, in a quasi-parallel fashion. The *model* of concurrent systems used throughout goes one step further and assumes that the smallest computation steps are atomic and that only one atomic computation step can be performed at a time, i.e., the overlapping execution of atomic computation steps is prohibited.

Concurrent and distributed systems A “distributed system” refers to a system that is based on several (often also “locally” separated) processors and has no shared memory. The exchange of data between the processes is carried out solely by sending and receiving messages. Concurrent systems, on the other hand, can also run on a single processor and processes can use shared memory.

Processes and threads The term “process” is commonly used in the theory of concurrent programming, whereas programming languages tend to use the word “thread”.

A further distinction is sometimes made based on who controls the execution: if it is the operating system, it is typically called a process; if it is a program (itself running as a process), it is typically called a thread. In what follows, both terms are used interchangeably without adhering to this strict distinction. Programming languages that provide constructs for creating and managing threads support so-called *multi-threading*. A related term frequently used in the context of operating systems is *multi-tasking*, which enables the simultaneous execution of different tasks (e.g., moving the mouse, printing, etc.). On a single-processor system, these tasks are not truly executed in parallel (but alternately, piece by piece), yet they appear parallel to the user.

Message passing and shared memory model The message-passing model assumes that individual processes use disjoint memory areas and that all communication is carried out via the exchange of messages, i.e., by sending and receiving. As already mentioned, this model is the basis of distributed systems.

Communication between processes can be further classified: In *synchronous* communication, the message exchange between two processes takes place directly (instantaneously), whereas in *asynchronous* communication, one process sends its message but the recipient need not receive it immediately (i.e., time may pass between sending and receiving). For many systems, e.g., the transmission of a web page to a client, asynchronous communication is more natural. A familiar analogy is the distinction between phone calls (synchronous) and email (asynchronous).

The shared memory model, by contrast, assumes that processes share a common memory area that all processes can read and write. Communication takes place directly via this shared memory. Care must be taken to prevent conflicts, such as one process reading a value before another has finished writing it. In a later chapter, various primitive operations for accessing shared memory are introduced, which may differ depending on the model or the actual hardware.

1.3 Planned Contents

The following content is planned for the lecture:

- **Synchronization and mutual exclusion with different assumptions about memory operations.** It is initially assumed that only atomic reading and writing to main memory is possible. The mutual exclusion problem for exactly two concurrent processes is considered first, and the terms deadlock-freedom and starvation-freedom are introduced. Three classical algorithms are discussed: Dekker's, Peterson's, and Kessels' algorithms. The problem is then extended to any number (n) of processes, and Lamport's algorithm and the Bakery algorithm are presented.

After three complexity results for mutual exclusion, stronger memory operations—such as test-and-set bits, read-modify-write objects, and compare-and-swap objects—are considered. These lead to simpler solutions to the mutual

exclusion problem. Finally, the consensus number and the so-called Herlihy hierarchy are discussed to categorize and classify memory primitives.

- **Programming primitives for concurrent programming.** This part discusses common programming primitives for concurrent programming as provided by libraries of many programming languages. Semaphores, monitors, channels, and tuple spaces are presented, along with solutions to some classical problems of concurrent programming using these abstractions: the Dining Philosophers Problem, the Sleeping Barber Problem, the Cigarette Smokers Problem, producer-consumer problems, the implementation of barriers, and the Readers- & Writers problem. Implementations are given in both pseudo-code and the Java programming language.
- **Access to multiple resources.** This chapter deals with the problem of multiple processes competing to access multiple resources, and the possibility of resulting deadlocks. Solutions for deadlock prevention (in particular the two-phase locking protocol) and for deadlock avoidance (in particular the Banker's algorithm) are discussed. Finally, transactional memory is introduced as a programming model and several distinguishing criteria of such models are discussed.
- **Semantic models of concurrent programming languages.** After a general introduction to the semantics of programming languages, the π -calculus is introduced as a message-passing model and its variants are analyzed.

1.4 Literature

Some literature references are listed below.

- On basic principles of concurrent programming:

Ben-Ari, M. (2006). *Principles of concurrent and distributed programming*. Addison-Wesley, Harlow, UK.

Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA.

Raynal, M. (2013). *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Berlin Heidelberg.

Reppy, J. H. (2007). *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England.

Taubenfeld, G. (2006). *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc, Upper Saddle River, NJ, USA.

- On process calculi, especially the π -calculus:

Milner, R. (1999). *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, Cambridge, England.

Sangiorgi, D. & Walker, D. (2001). *The π -Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA.

1.5 Model Assumptions

The most important assumption for models of concurrent programming is that only one computation step can be performed at a time, i.e., two computation steps never take place simultaneously or overlap in time. What constitutes a computation step depends on the model and is therefore not defined universally. The computation steps are carried out sequentially, but almost any interleaving of the computation steps is permitted, i.e., the order in which each process performs its steps is not fixed, but arbitrary. The "almost" refers to the fact that a certain fairness in the execution is assumed, which is explained later.

The following definition of concurrent programs incorporates the interleaving assumption.

Definition 1.5.1 (Concurrent program, Interleaving assumption). *A concurrent program consists of a finite set of processes, where each process consists of a sequential program executed in atomic computation steps. The overall execution consists of a sequence of atomic computation steps of the individual processes, which can be arbitrarily interleaved.*

The terms 'atomic command' or 'atomic instruction' are avoided in the definition since not only imperative programs are considered; functional programs, for example, evaluate by computing values of expressions. The definition covers such programs as well, since only the computation of a value needs to be defined in atomic steps.

In what follows, the interleaving assumption is discussed and reasons are given why this initially very restrictive (and sometimes unrealistic) assumption is nonetheless appropriate.

Granularity Since the interleaving assumption makes no statement about the size of a computation step, steps can be chosen arbitrarily small. One can therefore construct a model in which steps are so small that the non-parallel execution of steps is imperceptible.

Multitasking systems For single-processor systems that perform multitasking, the interleaving assumption fits very well, since only one step can be performed at a time anyway. In practice, however, each process is typically executed for a period of time before being interrupted by a global scheduler. The assumption of arbitrary interleaving is technically too strong here, since more than one instruction is generally executed per process per scheduling slot. Nevertheless, this behavior can be simulated by choosing an appropriate granularity.

The most important reason for assuming arbitrary interleaving is that formalizing the notion of "for a period of time" would complicate the formal model considerably.

If the model depends on the scheduler implementation and the hardware, new, faster hardware could require the entire model to be revised, causing previously correct algorithms to become incorrect. Another consideration is that the duration “for a period of time” may be affected by external circumstances, environmental factors, or transient phase fluctuations.

If correctness is proved for all possible interleavings, such circumstances need not be taken into account.

Multiprocessor systems For systems with multiple processors, the interleaving assumption may initially seem unrealistic, since computation steps can in principle take place in parallel. However, shared resources such as shared memory are generally protected against simultaneous access at the hardware level, enforcing a sequentialization for such steps. Autonomous computations (without access to shared resources) of individual processes can still overlap in time, but in this case the interleaving assumption poses no problem: from the outside, it is not observable whether two processes perform their computations in parallel or sequentially with interleaving.

1.5.1 The Fairness Assumption

The arbitrary interleaving permitted by Definition 1.5.1 is generally restricted by a further assumption. It is required that the execution observes a certain kind of fairness. There are different notions of fairness (see e.g., (Varacca & Völzer, 2006)). In what follows, a simple, rather weak concept of fairness is assumed.

Definition 1.5.2 (Fairness assumption). *Each process for which a computation step (of any length) is possible, makes this step in the overall execution sequence after a finite number of computation steps.*

The following example is intended to illustrate which execution sequences are prohibited by the fairness assumption:

(The value of X before the execution is 0)

process P :

(P1) **if** $X \neq 10$ **then goto** (P1);

process Q :

(Q1) $X := 10$;

Process P loops continuously until the variable X is assigned the value 10. Process Q sets the variable X to 10.

An unfair execution sequence is the infinite sequence (note that the fairness assumption applies only to *infinite* execution sequences): $(P1), (P1), (P1), \dots$. This sequence is

unfair because Q can perform a computation step, yet never does so after a finite number of steps. Every sequence $\underbrace{(P1), (P1), \dots, (P1)}_{n\text{-times}}(Q1)$ for fixed n is fair, though trivially

so since these sequences are all finite. Since no other sequences are possible for this example, the fairness assumption even forces the termination of all processes in this case. This is not generally the case, as the following slightly modified example shows:

(The value of X before the execution is 0)

process P :

(P1) **if** $X \neq 10$ **then goto** (P1);
(P2) **goto** (P1)

process Q :

(Q1) $X := 10$;

Now the fairness assumption still prohibits the infinite sequence $(P1), (P1), (P1), \dots$,
but the (also infinite) sequences $\underbrace{(P1), (P1), \dots, (P1)}_{n\text{-times}} \underbrace{(Q1) (P1), (P2), (P1), (P2), \dots}_{\text{infinitely often}}$ are
all fair (for any fixed n).

The phrase “of any length” in Definition 1.5.2 is intended to clarify that steps that are only enabled occasionally, but not continuously, are not enforced by the fairness assumption. With simple memory and process models, steps are always possible; but in more complex models, certain commands (e.g., acquiring a lock) can only be executed when no other process holds the lock. The fairness assumption does not guarantee that a process gets to execute at precisely the right moment (and thus acquire the lock).

1.5.2 Known Processes and Atomic Actions

Another assumption is that a program consists of a fixed number of processes and that no new processes can be created. This assumption will be relaxed later. While rather unrealistic, it is initially used to keep problems and solutions as simple as possible. A further initial assumption is that processes execute programs in a small imperative language (some commands have already appeared in the examples, e.g., **if-then**, **goto**, assignments $X := 10$, etc.). The syntax and semantics of the language are not formally defined here, as they are assumed to be known. It is assumed that each command line (labeled with line numbers of the form (P1), (P2), etc.) represents an instruction that is executed atomically, i.e., in one computation step.

2

Synchronization

In Section 2.1 we consider the mutual exclusion problem. After a motivating example the problem is formalized and requirements for a solution to the problem are defined. Section 2.2 explains several algorithms that solve the mutual exclusion problem for two processes under the assumption that only read and write operations on the shared memory are permitted as primitive indivisible (i.e., atomic) operations. In Section 2.3 the problem is extended to an arbitrary number of processes. Section 2.4 explains some complexity results on the mutual exclusion problem. In Section 2.5 atomic memory operations different from the basic read and write operations are considered. Several algorithms for solving the mutual exclusion problem using these new operations are presented. Finally, the atomic memory operations are classified qualitatively using the so-called consensus number in Section 2.6. Here, the consensus problem is introduced and the consensus hierarchy is explained.

2.1 The Mutual Exclusion Problem

We discuss the fundamental problem of mutual exclusion. As an introductory example, let us consider the “too much milk” problem:

Alice and Bob live together and want to always have milk in their refrigerator. Therefore, each of them goes to buy milk when they see that there is no milk in the refrigerator. However, this does not work because they always have too much milk in the refrigerator, e.g., when the following happens:

	Alice	Bob
5:00 p.m.	comes home	
5:05 p.m.	notices: no milk in the refrigerator	
5:10 p.m.	goes to the supermarket	
5:15 p.m.		comes home
5:20 p.m.		notices: no milk in the refrigerator
5:25 p.m.		goes to the supermarket
17:30	returns from the supermarket, puts milk in the refrigerator	
17:40		returns from the supermarket, puts milk in the refrigerator

Too much milk in the refrigerator

To avoid the “too much milk” problem, Alice and Bob have agreed to leave notes on the refrigerator that both of them can read. This is to ensure that:

- At most one of the two people goes to buy milk when there is no milk in the refrigerator.
- One of the two people goes to buy milk when there is no milk in the refrigerator.

The solution that, for example, only Bob goes to buy milk is not possible, because then, for example, Alice could come home, find that there is no milk, and then wait indefinitely for Bob to buy milk. This violates the second condition. Another (unrealistic) assumption is that Alice and Bob cannot see each other even when they are both at home at the same time.

First attempt at a solution: If there is no note on the refrigerator and no milk in the refrigerator, then write a note on the refrigerator, go buy milk, put the milk in the refrigerator, and then remove the note from the refrigerator.

Written as processes:

Program for Alice:

```
(A1) if no note then  
(A2) if no milk then  
(A3)   write note;  
(A4)   buy milk;  
(A5)   remove note;
```

Program for Bob:

```
(B1) if no note then  
(B2) if no milk then  
(B3)   write note;  
(B4)   buy milk;  
(B5)   remove note;
```

Unfortunately, this approach does not work because there is still too much milk in the refrigerator: Consider the sequence (A1), (A2), (B1), (B2), ... In this case, Alice and

Bob each find that there is no note on the refrigerator and no milk in the refrigerator. Both leave notes and go to buy milk, and then remove their notes.

Second solution approach: First leave a note on the refrigerator, then check whether there is a note from the other person. Only if there is no other note, check whether there is milk and go buy milk if there is no milk. Then remove the note. Written as processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)   buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) if no note from Alice then  
(B3)   if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

However, this solution does not work either. It is possible that no milk will be bought. Consider the sequence (A1), (B1), (A2), (B2), (A5), (B5). Both leave their notes, then both notice that there is a note from the other person. Therefore, neither looks in the refrigerator, but both remove their notes.

Third attempt at a solution: Alice does the same as before. Bob: Write a note, wait until there is no note from Alice on the refrigerator, then check whether the milk is empty, go buy milk, remove his own note. As processes:

Program for Alice:

```
(A1) write note "Alice";  
(A2) if no note from Bob then  
(A3)   if no milk then  
(A4)   buy milk;  
(A5) remove note "Alice";
```

Program for Bob:

```
(B1) write note "Bob";  
(B2) while note from Alice do skip;  
(B3) if no milk then  
(B4)   buy milk;  
(B5) remove note "Bob";
```

The key difference from the previous solution is that as soon as Bob has written a note on the refrigerator, he waits for Alice to remove her note. This solution works if we assume that Alice is not too fast: Consider the sequence (A1), (B1), (B2), (A2), (A5), (A1), (B2). In this case, milk will never be bought because

infinitely often

Alice removes her note, but Bob does not notice this (more precisely: never notices this) because Alice is quick enough to write a new note on the refrigerator.

A correct solution: For a correct solution, Alice and Bob each use two notes, labeled Alice1, Alice2, Bob1, and Bob2, respectively.

```
(A1) write note Alice1;  
(A2) if Bob2  
(A3)   then write note Alice2;  
(A4)   else remove note Alice2;  
(A5) while Bob1 and ((Alice2 and Bob2) or (not Alice2 and not Bob2))  
(A6)   do skip;  
(A7) if no milk  
(A8)   then buy milk;  
(A9) remove Alice1;
```

Program for Alice

```
(B1) write note Bob1;  
(B2) if not Alice2  
(B3)   then write note Bob2;  
(B4)   else remove note Bob2  
(B5) while Alice1 and ((Alice2 and not Bob2) or (not Alice2 and Bob2))  
(B6)   do skip;  
(B7) if no milk  
(B8)   then buy milk;  
(B9) remove Bob1;
```

Program for Bob

To illustrate the correctness of the algorithm, the most important observation is that for the notes Alice2 and Bob2, the following applies: Bob gets preference if either both notes are present or both are not present. In both other cases, Alice is given preference. However, the two notes Alice2 and Bob2 are only taken into account if both really want to look in the refrigerator, i.e., the notes Alice1 and Bob1 are present.

At this point, the correctness of the algorithm will not be discussed further or even proven, as various algorithms for the problem of mutual exclusion will be discussed later. The “too much milk” example was only intended to illustrate how quickly incorrect algorithms can be constructed in the context of concurrency and to give a first impression of the conditions that should be set in order to speak of a correct solution. These conditions are defined below using various terms.

The mutual exclusion problem arises from the guarantee that exclusive access to a shared resource is ensured, i.e., while a single process is accessing such a resource, no other processes may access it. Access to such a resource (or the associated program code) is referred to as a *critical section*.

Access to shared resources occurs frequently, e.g., in operating systems, databases, networks, etc. As already explained in the introduction, unprotected access can lead to undesirable results. Situations in which multiple processes access a shared resource and in which the value of the resource depends on the order in which the processes are executed are called *race conditions*. The main goal of the mutual exclusion problem is to avoid or eliminate race conditions.

2.1.1 Correctness Criteria

To formalize the mutual exclusion problem, the following assumption is made about the concurrent processes: The code of each process has the form

```
loop forever
    remaining code
    initialization code
    critical section
    exit code
end loop
```

This means that a process runs in an endless loop, first executing some code (referred to as “remaining code”) that is not of interest here, and then entering the critical section in three phases: First, a code for initialization or synchronization is executed (initialization code), then the critical section is entered, and finally an exit code is executed, which is used, for example, to inform the other processes that the critical section has been left.

The mutual exclusion problem is solved by specifying the code for the initialization code and the exit code, whereby the following requirements must be met:

- *Mutual exclusion*: Two or more processes are never in their critical section at the same time.
- *Deadlock freedom*: If a process wants to enter its critical section, then some process eventually enters the critical section.

Note that deadlock freedom only requires that *some* process enter the critical section, but not necessarily the process *mentioned* by the definition of deadlock freedom that wants to enter the critical section. Deadlock freedom guarantees that the concurrent program (as a whole) will not “hang”, i.e., that a situation will never arise in which all processes wait indefinitely for each other to enter the critical section. However, deadlock freedom does *not* guarantee that a waiting process will enter its critical section after a finite amount of time. It could be, for example, that another process repeatedly enters the critical section, thereby guaranteeing deadlock freedom.

Starvation freedom is a stronger requirement than deadlock freedom:

- *Starvation freedom*: If a process wants to enter its critical section, it must enter it after a finite number of computation steps.

Sometimes starvation freedom can be dispensed with if there are only “few” situations in the overall program flow in which individual processes want to access a critical section.

Obviously, the following applies:

Lemma 2.1.1. *A starvation-free algorithm is also deadlock-free.*

The property of mutual exclusion is a so-called safety property, a property that must always be satisfied. The other two properties, on the other hand, are so-called liveness properties, which are properties that must become true at some point. With the help of temporal logic, it is possible to provide proof that such properties (formulated as temporal logic formulas) are satisfied for algorithms. Model checking can be used to provide this proof. In what follows, these proofs are carried out in a rather informal manner (without the use of temporal logic).

For a more precise specification of the mutual exclusion problem, the following assumptions are made:

- In the remaining code, a process cannot influence other processes. Otherwise, no assumptions are made about the remaining code (it may contain infinite loops or terminate, etc.).
- Resources (program variables) used in the initialization code or exit code must not be changed by the code in the critical section and the remaining code.
- No errors occur during the execution of the initialization code, the code in the critical section, and the exit code.
- The code in the critical section and in the exit code consists only of a finite number of execution steps; in particular, the code does not contain any infinite loops and is “wait-free”, i.e., there are no wait loops that could potentially run indefinitely. This implies (based on the fairness assumption made) that the process leaves the critical section after entering it after a finite number of computation steps.

2.2 Mutual Exclusion Algorithms for Two Processes

This section describes some well-known algorithms for the mutual exclusion problem. It assumes *two* concurrent processes under the assumption that there are read and write commands for memory registers that are executed atomically.

The command **await** *condition*; is used as an abbreviation for **while** $\neg$ *condition* **do skip**; where the condition may only be evaluated once per computation step.

2.2.1 Dekker’s Algorithm

Historically, Dekker’s algorithm is the first algorithm to solve the mutual exclusion problem for two processes. The code for both processes is shown in Figure 2.1.

The processes P and Q each use a Boolean variable *wantp* (or *wantq*), to indicate that they want to enter the critical section. Furthermore, they use a shared variable *turn*, which specifies which process – in the event of simultaneous access to the critical section – is allowed to enter the section. If *turn* has the value 1, then P will enter the critical section, otherwise (*turn* = 2) Q will enter the section.

Initial: wantp = False, wantq = False, turn = 1	
<u>Process P:</u> loop forever (P1) remaining code (P2) wantp := True; (P3) while wantq do (P4) if turn=2 then (P5) wantp := False; (P6) await turn=1; (P7) wantp := True; (P8) Critical section (P9) turn := 2; (P10) wantp := False end loop	<u>Process Q:</u> loop forever (Q1) remaining code (Q2) wantq := True; (Q3) while wantp do (Q4) if turn=1 then (Q5) wantq := False; (Q6) await turn=2; (Q7) wantq := True; (Q8) Critical section (Q9) turn := 1; (Q10) wantq := False end loop

Figure 2.1: Dekker's algorithm

Lemma 2.2.1. *Dekker's algorithm satisfies mutual exclusion.*

Proof. Suppose the statement is false. Then there is a state of the program such that both P and Q are in their critical sections. This implies that both processes have “jumped over” the **while**-loop at some point, i.e., they have passed through the condition (in lines (P3) and (Q3), respectively). The following cases can be distinguished:

- Neither process has ever passed through the loop body. This is impossible, since then wantp and wantq must be false. Let Q be the second process that checks the **while** condition. Then P has previously set wantp to true and can no longer set wantp to false afterwards (and since the critical section must not change wantp, wantq, and turn!).
- One process has run through the loop body while the other is in the critical section. Then it is easy to verify that it can no longer enter the critical section.
- Both processes run through the loop body. Then one of the two processes must get stuck at the **await** statement (since turn is only changed after running through the critical section).

The cases show that the assumption is false, i.e., the algorithm satisfies mutual exclusion. □

Lemma 2.2.2. *Dekker's algorithm is starvation-free.*

Proof. Assume that the statement is false. Then one of the two processes enters the initialization code (lines 3-7), but never enters the critical section. Since both processes

behave symmetrically, assume without loss of generality that P starves. There can only be two reasons for this: The **while** loop is run infinitely many times, or the process P cannot exit an **await** construct (line (P6)). First, consider the second case. Then turn must have the value 2. The following cases can be distinguished, depending on where Q is located:

- Q is in its remaining code. This is impossible, since want_q must have been true when P entered the **while** loop. At this point, Q must have been in lines (Q2) to (Q9). To return to (Q1) afterwards, Q would have had to set the value of turn to 1 in line (Q9). However, it was assumed that turn has the value 2.
- Q is in the critical section or in the exit code (lines (Q8) to (Q10)). Then the value of turn is set to 1 (in line (Q9)). Since Q can no longer change the value of turn afterwards, P must leave the **await** statement after a finite number of steps.
- Q is in the initialization code. Since P is in line (P6), want_p must be false. Q will therefore enter the critical section (and cannot starve at its **await** statement in line (Q6), since turn has the value 2). Now the same conclusion can be drawn as before.

Now consider the first case: P runs through the **while** loop endlessly, but does not get stuck endlessly at the **await** construct. This leads to the conclusion that after a finite sequence of computation steps, turn must have the value 1 and this value is not changed (this can only be done by P in the exit code, which P never reaches by assumption). This implies that either want_q must be false (which is not possible, since P runs through the **while** loop), or that Q must wait at its **await** construct in line (Q6). However, since want_q was previously set to false in line (Q5), it is impossible for P to run through its **while** loop indefinitely. This means that the assumption was false and the algorithm is starvation-free. $\square$

Since Dekker's algorithm satisfies mutual exclusion and is starvation-free (and thus also deadlock-free), the following applies:

Proposition 2.2.3. *Dekker's algorithm solves the mutual exclusion problem.*

2.2.2 Peterson's Algorithm

Peterson's algorithm is similar to Dekker's algorithm, but it only requires one **await** statement (without any additional **while** loops) and is therefore somewhat simpler. The algorithm is shown in Figure 2.2.

It is straightforward to verify the following:

Proposition 2.2.4. *Peterson's algorithm solves the mutual exclusion problem. It guarantees mutual exclusion and is starvation-free.*

Proof. The algorithm guarantees mutual exclusion: Assume this is false. Then there is a state such that P and Q are both in the critical section. The last two tests in lines (P4)

Initial: wantp = False, wantq = False, turn irrelevant	
<u>Process P:</u> loop forever (P1) remaining code (P2) wantp := True; (P3) turn := 1; (P4) await wantq = False or turn = 2 (P5) Critical section (P6) wantp := False; end loop	<u>Process Q:</u> loop forever (Q1) Remaining code (Q2) wantq := True; (Q3) turn := 2; (Q4) await wantp = False or turn = 1 (Q5) Critical section (Q6) wantq := False; end loop

Figure 2.2: Peterson's algorithm

and (Q4) cannot have been executed directly after each other, since then only one of the two processes would have been allowed through and wantp and wantq must both be true. This means that the second process that entered the critical section must have performed at least one assignment after the first process entered the critical section. However, the last of these assignments sets the value of turn so that the setting process cannot pass the **await** condition, since it always gives priority to the other process.

The algorithm is starvation-free: Suppose this is false. Then one process remains in its initialization code forever. Only the **await** construct can be considered here, since there are no other loops. Three cases can apply to the other process:

1. It remains in its remaining code forever. Then its want variable is false and thus the supposedly starving process cannot starve.
2. It also remains in its initialization code forever. This cannot be the case, since the **await** condition is always true for one of the two processes.
3. It repeatedly runs through its critical section. This cannot be the case, because after the first run, when the initialization code is executed for the second time, it sets the turn variable so that the supposedly starving process must receive the value "true" for its **await** condition. □

It should also be noted that the condition of the **await** construct does not need to be evaluated atomically: the correctness of the algorithm can also be established if the two sub-conditions are evaluated sequentially.

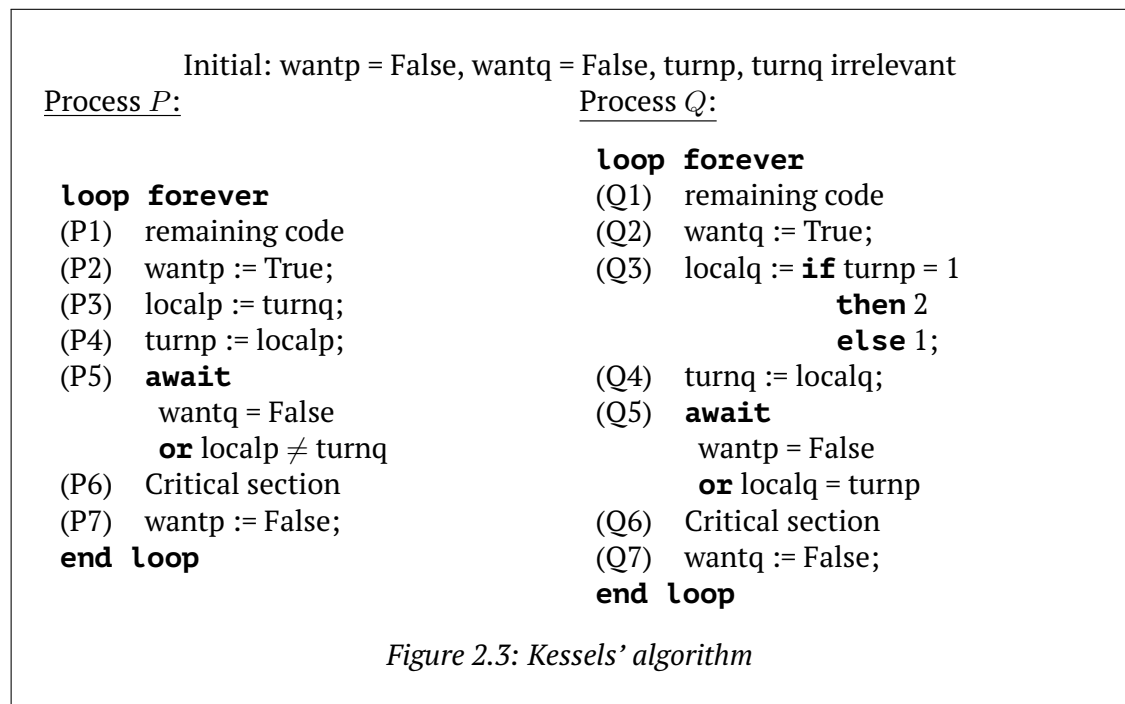
2.2.3 Kessels' Algorithm

The third algorithm is Kessels' algorithm, which, unlike the previous two, does not require variables (or memory locations) written to by multiple processes. Such single-

writer/multiple-reader algorithms are considered easy to implement in message-passing models. Kessels' algorithm is a modification of Peterson's algorithm inspired by the too-much-milk solution: instead of a single shared variable `turn` written by both processes, two variables `turnp` and `turnq` are used, satisfying

$$\begin{aligned} \text{turn} &= 1, & \text{if } \text{turnp} = \text{turnq} \\ \text{turn} &= 2, & \text{if } \text{turnp} \neq \text{turnq} \end{aligned}$$

Both processes also use a local variable (`localp` and `localq`, respectively). The code for Kessel's algorithm is shown in Figure 2.3. Kessels' algorithm guarantees mutual exclu-



sion and is starvation-free. The proof is similar to the correctness proof of Peterson's algorithm.

2.3 Mutual Exclusion Algorithms for n Processes

This section considers algorithms for the mutual exclusion problem that work for any number of processes. The model assumption that only (atomic) read and write operations on memory are available and that the number n of processes is known is retained.

A simple solution for n processes extends the two-process algorithms via divide-and-conquer: divide the n processes into two groups of $n/2$, determine a "winner" from each group recursively, and then use a 2-process mutual exclusion algorithm to decide which winner enters the critical section.

A starvation-free 2-process algorithm used in this way yields a starvation-free n -process algorithm.

The drawback is overhead: regardless of whether contention actually occurs, each process must win $\lceil \log_2 n \rceil$ competitions before entering (and leaving) the critical section.

Since simultaneous contention by all n processes is rare in practice, more efficient solutions are desirable.

2.3.1 Lamport's Fast Algorithm

We now consider Lamport's algorithm, which provides a faster solution. The program of the i th process is shown in Figure 2.4.

Initial: $y=0$, for all $i \in \{1, \dots, n\} : \text{want}[i]=\text{False}$, value of x irrelevant

Program of the i th process

```

loop forever
(1)  remaining code
(2)  want[ $i$ ] := True;
(3)   $x := i$ ;
(4)  if  $y \neq 0$  then
(5)    want[ $i$ ] := False;
(6)    await  $y = 0$ ;
(7)    goto (2);
(8)   $y := i$ ;
(9)  if  $x \neq i$  then
(10)   want[ $i$ ] := False;
(11)   for  $j := 1$  to  $n$  do await  $\neg \text{want}[j]$ ;
(12)   if  $y \neq i$  then
(13)     await  $y = 0$ ;
(14)     goto (2);
(15) Critical section
(16)  $y := 0$ ;
(17) want[ $i$ ] := False;
end loop

```

Figure 2.4: Lamport's algorithm for n processes

For clarity, a simplified flowchart of the algorithm is shown in Figure 2.5.

The individual parts of the algorithm can be explained as follows:

In line (2), process i first sets its want variable to True to indicate that it wants to

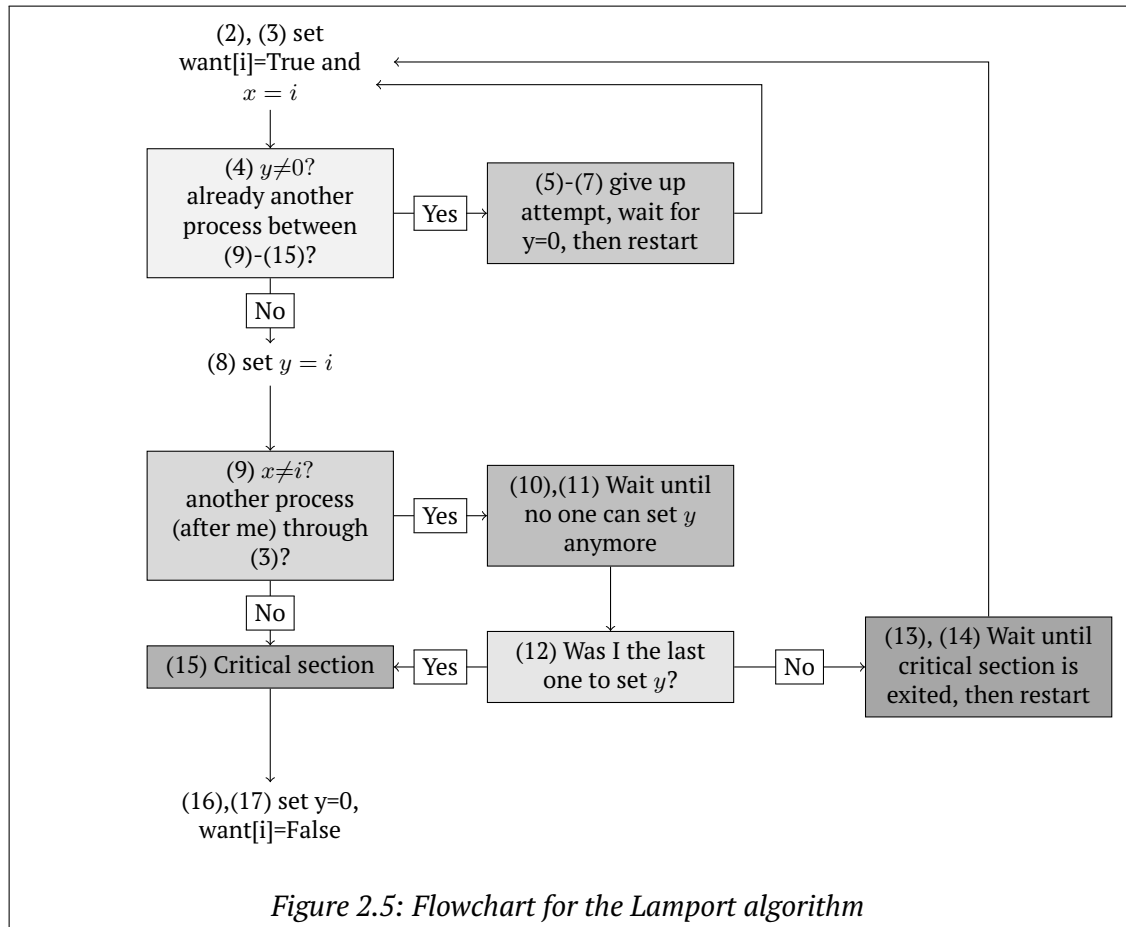


Figure 2.5: Flowchart for the Lamport algorithm

enter the critical section. Immediately afterwards (line (3)), it sets x to the value i so that it can later observe whether other processes want to enter the critical section. The query in line (4) serves to determine whether another process is already in the critical section or is attempting to reach it. If this is the case, process i aborts the attempt to reach the critical section directly: it sets its `want` variable to false, waits until $y = 0$ applies (then a process has successfully left the critical section) and then starts a new attempt (all this happens in lines (5)-(7)).

In line (8), process i sets the variable y to the value i . The variable y acts like a barrier: as soon as it is set to a value other than 0, no other process will be able to pass the query in line (4). If several processes successfully pass query (4) and then execute the assignment $y := i$ in line (8), then (as will be shown later) the process that set the value of y last will enter the critical section.

By querying the value of x in line (9), a process can enter the critical section directly: If a process i reads in line (9) that $x = i$, then it knows that no other process j , which could still change y , will read $x = j$ in its line (9). Therefore, in this case, process i can safely enter the critical section.

If the query in line (9) is false, then there are other processes that have changed x . In this case, the for loop in line (11) waits until all processes are waiting (or do not want to enter the critical section at all). This ensures that no one can change the value of y . Finally, in line (12), process i asks whether it was the last one to set y . If this is the case, it is allowed to enter the critical section; otherwise, it aborts its attempt (14), but first waits for another process to leave the critical section (hence the `await` in line (13)). The exit code in lines (16) and (17) sets the variable y to 0, thereby indicating that the critical section has been exited, and finally, the want variable of process i is set to False.

Some simple observations about Lamport's algorithm are:

- If only a single process wants to enter the critical section, only a constant number of operations are performed in the initialization and exit code: In the initialization code, 5 operations: (2), (3), (4), (8), and (9); in the exit code, two: (16) and (17).
- The algorithm is not starvation-free: If a process enters one of the two termination branches, it will start over from the beginning. This can happen to it any number of times, since other processes will always reach the critical section.
- The algorithm offers process i two ways to reach the critical section. Either via (9), or via (12).
- Another interesting aspect is that if process i reaches the critical section via (12), it has set its want variable to false while it is in the critical section.

The following shows that Lamport's algorithm guarantees mutual exclusion and is deadlock-free.

Proposition 2.3.1. *Lamport's algorithm guarantees mutual exclusion.*

Proof. A process is said to *reach the critical section via path α* if it evaluates the `if` condition in line (9) to True (i.e., it is allowed to jump directly to (15)). If a process reaches the critical section after evaluating the `if` condition in line (12) to True, then the process is said to *reach the critical section via path β* . As already mentioned, these are the only two ways to reach the critical section.

To prove the statement, we assume that (at least) two processes are in the critical section at the same time. Let process i be the first in the critical section and process j the second. Two cases can be distinguished:

- Process i reaches the critical section via path α . When process i executes the query in line (9), the following must apply: $x = i$ and $y \neq 0$. This implies that one of the following two cases must apply:
 - Process j has not yet executed line (3). Since process j will then read $y \neq 0$ when executing line (4), process j will abort its attempt and cannot have entered the critical section at the same time as process i .

- Process j executed line (3) *before* process i executed line (3). Then, when executing (9), the following applies to process j : $x \neq j$, since process j can no longer set the value of x and process i has already changed this value to $x = i$. Thus, process j cannot reach the critical section via path α . However, process j can only reach the critical section via path β if process i sets its want variable to False (as long as process j does not terminate the for loop). Since process i only sets its want variable to False *after* it has reached the critical section, it is also impossible in this case for both processes i and j to be in the critical section at the same time.
- Process i reaches the critical section via path β . After process i has run through the for loop in line (11), the value of y can only be changed again when process i executes the exit code: Since process i had waited until all want entries are false, every other process must remain stuck at an `await` construct. If another process remains stuck in line (6), it has not changed y and can no longer do so. If a process leaves the remaining code later, it is always directed to the `await` in line (6) in line (4), since $y \neq 0$ applies. If another process hangs in line (11) or (13), it has first changed y and then set its want variable to False. However, since process i waits in line (11) for all want entries to be set to False, y will not change afterwards.

This implies in particular that process j cannot change the value of y and therefore will not reach the critical section along path β as long as process i is in the critical section.

Process j also cannot reach the critical section via path α : Consider the evaluation sequence at the moment when process i executes line (12). If process j has not yet executed (4), it will wait in line (6) until process i leaves the critical section. If process j has executed (4) but has not yet executed (8), then process i cannot have executed the for loop (since `want[j]=True`). If process j has executed (8), then it is impossible for $y = i$ to have been true for process i when executing line (12).

Since all cases lead to a contradiction, the assumption that processes i and j are in the critical section at the same time is false. Thus, Lamport's algorithm satisfies mutual exclusion. $\square$

To prove deadlock freedom, the notation $(i_1, j_1), (i_2, j_2), (i_3, j_3), \dots$ is used for a computation sequence, where for (i_k, j_k) :

- i_k is the number of the process that performs a computation step.
- j_k is the line of code that process i_k executes.
- it is assumed that the current code pointer is known for every other process and, in addition, the assignment of the program variables is known. The entire description of this assignment is referred to as "state k ".

Proposition 2.3.2. *Lamport's algorithm is deadlock-free.*

Proof. Assume that the statement is false. Let $\mathcal{P} = (i_1, j_1), (i_2, j_2), (i_3, j_3), \dots$ be a sequence of computations that refutes deadlock freedom.

- Since $\mathcal{P}$ refutes deadlock freedom, the following must hold: There exists $k_1 \in \mathbb{N}$ such that for all $k'_1 \geq k_1$, $j_{k'_1} \neq 15$, i.e., no process reaches the critical section in the sequence $(i_{k'_1}, j_{k'_1}), (i_{k'_1+1}, j_{k'_1+1}), \dots$ and for at least one $l \geq k'_1$ it holds that $j_l \in \{2, \dots, 14\}$ i.e., at least one process wants to enter the critical section.
- Due to the fairness assumption, it also follows that at some point all processes must have completed their exit code, i.e., there is $k_2 \geq l$ such that for all $k'_2 \geq k_2$, $j_{k'_2} \in \{1 - 14\}$.
- Since no process is in the exit code and in the critical section from k_2 onwards, the value of y cannot be set to 0 for all successor states. Since process i_l is also in the initialization code, y must take on a value other than 0 at some point, because either $y \neq 0$ or process i_l executes the assignment for y in line (8) at some point. This means that there is a $k_3 \geq k_2$ from which the variable y always has a value other than 0.
- For all states from state k_3 onwards, the following applies: If the program pointer of a process is not directly before line (8), then the program pointer can no longer reach it in any subsequent states. This is obvious, because from state k_3 onwards, all executions of line (4) evaluate the if condition to false (y is not equal to 0 and can no longer be set to 0).
- Due to the fairness condition, all processes whose program pointer is directly before line (8) will execute the code for (8) after a finite many steps. That is, there is a state $m \geq k_3$ in which line (8) is executed for the last time. That is, process i_m sets y to the value i_m . After that, after a finite number of steps, all processes must set their want entry to False (without being able to change it back), whereby the value of y cannot be changed (all processes are either in line (1) or in (5)-(6) or in (9)-(14)).
- The last step implies that process i_m must run through the for loop after a finite number of steps. Since $y = i_m$ always applies, process i_m must enter the critical section after a finite number of steps. This leads to a contradiction. $\square$

2.3.2 The Bakery Algorithm

Finally, another algorithm for the mutual exclusion problem for n processes is explained, the so-called *Bakery Algorithm*. Its special feature is that it is not only starvation-free, but also satisfies the so-called FIFO property (also known as “0-bounded waiting”). To recap, here is the definition of starvation-freedom:

Starvation-freedom: If a process wants to enter its critical section, then it must enter it after a finite number of computation steps.

However, this property does not guarantee the process how long it must wait before it is allowed to enter the critical section, or how many other processes before it are allowed to enter the critical section and how often.

In what follows, a *waiting process* refers to a process that performs a “busy wait”, i.e., it constantly checks a condition until it is fulfilled by another process changing a variable value. (Until now, `await` constructs have mostly been used for this purpose). The initialization code in mutual exclusion algorithms can be divided accordingly: The code before the wait (which must be wait-free) is referred to as the “doorway”, and the code for waiting is referred to as “waiting”. Figure 2.6 illustrates this division for mutual exclusion algorithms.

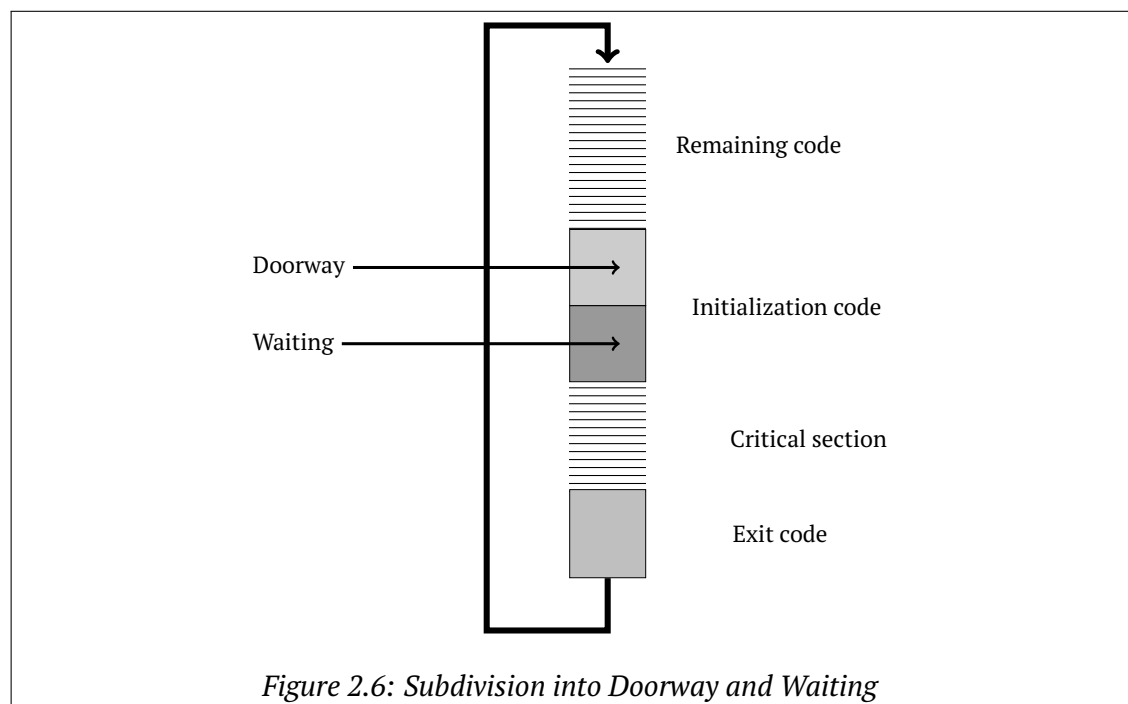


Figure 2.6: Subdivision into Doorway and Waiting

The term “bounded waiting” and other related terms are defined as follows:

Definition 2.3.3. A mutual exclusion algorithm satisfies

r -bounded waiting, if the following applies to every process i : If process i leaves the doorway before process j (with $j \neq i$) enters the doorway, then process j enters the critical section at most r times before process i enters the critical section.

bounded waiting, if there is an $r \in \mathbb{N}$ such that the algorithm satisfies r -bounded waiting.

the FIFO property, if it satisfies 0-bounded waiting. (FIFO = first-in-first-out)

Note that none of these properties imply deadlock freedom (and therefore also starvation freedom). Informally, the FIFO property means that the first process to cross the doorway is also the first to enter the critical section. Lamport’s algorithm does not

satisfy bounded waiting, since a process can enter the critical section any number of times before another process is allowed to enter the critical section. The tournament algorithms also do not satisfy bounded waiting.

The Bakery algorithm explained below, on the other hand, satisfies 0-bounded waiting. The idea behind the Bakery algorithm is relatively easy to explain. Each process draws a number in the doorway that is greater than (or equal to, see below) all numbers assigned so far. The process with the smallest number is always allowed to enter the critical section. The name of the Bakery algorithm comes from the fact that this number assignment procedure is used in some bakeries (in Germany, the procedure is more commonly known from government offices). First, the simplest variant of the Bakery algorithm is explained. The code for process i is shown in Figure 2.7.

Initial: for $i = 1, \dots, n$ number[i]=0

Program of the i th process

```

loop forever
(1)  remaining code
(2)  number[ $i$ ] := 1+max(number);
(3)  for  $j:=1$  to  $n$  do
(4)    await number[ $j$ ]=0 or (number[ $j$ ], $j$ )  $\geq_{lex}$  (number[ $i$ ], $i$ )
(5)  Critical section
(6)  number[ $i$ ]=0
end loop

```

Figure 2.7: Simple variant of the Bakery algorithm

Here, $\geq_{lex}$ refers to lexicographic order, i.e., $(a, b) \geq_{lex} (c, d)$ if and only if $a > c$ or $(a = c \text{ and } b \geq d)$. If we assume for the simple algorithm that the maximum of the number field can be calculated atomically (including the assignment), we can provide proof that it guarantees mutual exclusion, is starvation-free, and satisfies the FIFO property. Note that if the maximum is calculated atomically, but the assignment only takes place in the next step, then the algorithm does not satisfy mutual exclusion: Two processes i and j with $i < j$ can both calculate the same value for number[i] and number[j], respectively. Process j then continues its execution and passes line (4) because number[i] is still set to 0, i.e., j is in the critical section. If process i now continues its computation, then i also passes line (4) because number[j] = number[i] and $i < j$.

However, even the atomic computation of the maximum is very unrealistic. This assumption is not used in the extended Bakery algorithm, but the algorithm is somewhat more complicated. The code for process i is shown in Figure 2.8. The figure also shows the code for calculating the maximum.

The choosing field is used to mark entry into and exit from the doorway. The addi-

Initial: for $i = 1, \dots, n$ number[i]=0, choosing[i]=False

Program of the i th process

loop forever

(1) remaining code

(2) choosing[i] := True;

(3) number[i] := 1+maximum(number);

(4) choosing[i] := False;

(5) **for** $j:=1$ **to** n **do**

(6) **await** choosing[j]=False;

(7) **await** number[j]=0 **or** (number[j], j) $\geq_{lex}$ (number[i], i);

(8) Critical section

(9) number[i]=0

end loop

The maximum is calculated as follows:

(1) max := 0

(2) **for** $j:=1$ **to** n **do**

(3) current := number[j];

(4) **if** max < current **then** max := current;

(5) number[i] := 1+max;

Figure 2.8: Extended variant of the Bakery algorithm

tional `await` query on the `choosing` field serves to wait until the respective process in the doorway has set its number value.

The following proves that the extended Bakery algorithm guarantees mutual exclusion, is starvation-free, and satisfies the FIFO property. To do this, we first prove a few auxiliary theorems.

It can be easily verified that the following applies:

Lemma 2.3.4. *If the value of `number[k]` is not changed while process i calculates the maximum, then the value `number[i]` is subsequently greater than the value of `number[k]`.*

A process i is in the doorway if its program pointer is between lines (2) and (4). It is in the bakery if it is in the waiting or critical section (lines (5)-(8)). A direct consequence of Lemma 2.3.4 is:

Lemma 2.3.5. *If process i and process k are both in the bakery and i enters the bakery before k enters the doorway, then `number[i] < number[k]`.*

The next lemma implies mutual exclusion:

Lemma 2.3.6. *If process i is in the critical section and process k is in the bakery, then `(number[i], i) <lex (number[k], k)`.*

Proof. Let T_6^i be the step in which process i passes through line (6) via the `await` statement for $j = k$ (i.e., process i reads `choosing[k]` for the last time). Similarly, let T_7^i be the step in which process i reads `number[k]` for the last time. Then T_6^i must precede T_7^i (noted as $T_6^i < T_7^i$). For process k , let T_2^k, T_3^k, T_4^k be the steps in which it last executed program lines (2), (3), (4), respectively. It must hold that $T_2^k < T_3^k < T_4^k$.

In step T_6^i , `choosing[k]` had the value `False`, and in steps T_2^k, T_3^k , and T_4^k , `choosing[k]` had the value `True`. This implies that one of the following two cases must hold:

1. $T_6^i < T_2^k$: In this case, Lemma 2.3.5 implies that `number[i] < number[k]` must hold, and the statement is proven.
2. $T_4^k < T_6^i$: In this case, $T_3^k < T_4^k < T_6^i < T_7^i$ applies, i.e., at the point in time when process i executes line (7) for the last time and reads `number[k]`, its value cannot be 0 (it was set at the point in time T_3^k !). Since the `await` condition in line (7) is true, `(number[i], i) <lex (number[k], k)` must have been true.

□

Proposition 2.3.7. *The extended bakery algorithm guarantees mutual exclusion, is starvation-free, and satisfies the FIFO property.*

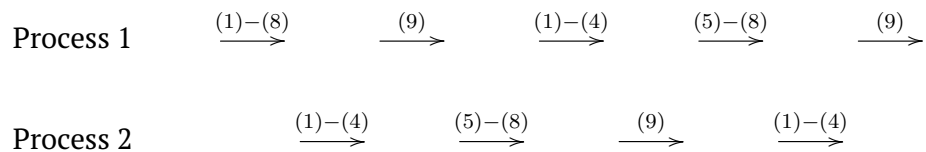
Proof. Mutual exclusion follows directly from Lemma 2.3.6, since otherwise a contradiction can be derived. The FIFO property follows from Lemmas 2.3.5 and 2.3.6. The proof of starvation-freedom is simple: First, consider deadlock-freedom: If, in an infinitely long sequence of computations, at some point no more processes enter the critical section, but at least one process is still in the bakery, then there is a point in time

when no process enters the doorway and the bakery. From this point on, a process must be able to run through the `for` loop. Starvation freedom follows from deadlock freedom and the FIFO property. $\square$

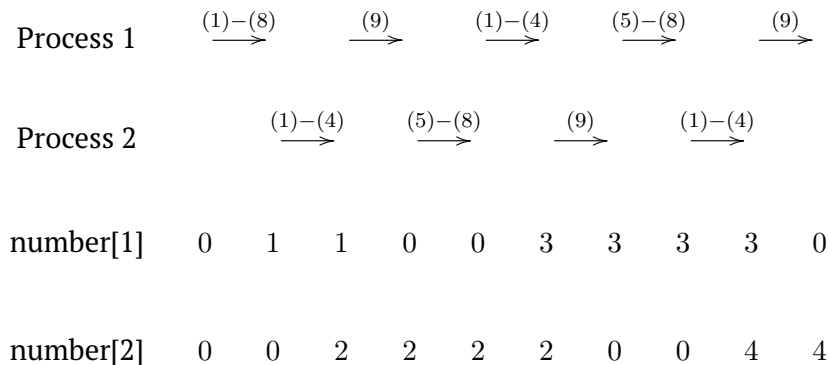
In addition to these advantages of the Bakery algorithm, there are some disadvantages:

Disadvantage 1: The algorithm is not fast in the case where only one process wants to enter the critical section. It must perform $O(n)$ computation steps in the initialization code.

Disadvantage 2: The presented algorithm requires unlimitedly large numbers for the number field. This is easy to see if we consider only two processes: Process 1 and process 2 visit the critical section alternately, with the execution mixed as follows:



It is important to note that line (9) of one process is always executed only after the other process has calculated the maximum. If we add the values of number, we can see the effect:



However, there are (more complicated) variants of the Bakery algorithm that work with a limited number size (e.g., the so-called Black-White Bakery algorithm).

As a final note on the Bakery algorithm, we would like to mention that care must also be taken when calculating the maximum of the number values. If the previous algorithm is replaced by the following code:

```
(1) maxpos := i
(2) for j:=1 to n do
(3)   if number[maxpos] < number[j] then maxpos := j
(4) number[i] := 1+number[maxpos];
```

then the extended Bakery algorithm becomes incorrect, as mutual exclusion is no longer guaranteed. A counterexample can be constructed with three processes 1, 2,

and 3. Assume that processes 2 and 3 initially calculate their new number value in an interleaved manner, so that $\text{number}[2] = 1$ and $\text{number}[3] = 1$ apply afterwards. Then process 2 is in the critical section, while process 3 waits for it to leave (i.e., for the $\text{number}[3]$ value to be set to 0). If process 1 now performs all steps of the maximum computation except step 4, then $\text{maxpos} = 2$. If process 2 now leaves the critical section, sets $\text{number}[2] = 0$, and process 3 enters the critical section and then process 1 executes line (4) of the maximum computation, this execution sets $\text{number}[1] = 1 + \text{number}[2] = 1 + 0 = 1$. Process 1 can then enter the critical section even though process 3 is still in it.

2.4 Three Complexity Results for the Mutual Exclusion Problem

This section presents some statements on the complexity of mutual exclusion algorithms. For the most part, the corresponding proofs are omitted. Note that these results are still based on the same model, i.e., there are only atomic read and write commands for the shared memory.

In 1980, J. Burns and N.A. Lynch provided proof of a lower bound for the required space in shared memory:

Theorem 2.4.1. *Every deadlock-free mutual exclusion algorithm for n processes requires at least n shared memory locations.*

In fact, it can be shown that these n memory locations are actually sufficient, whereby such a memory location only needs to be able to store bits (true or false):

Theorem 2.4.2. *There is a deadlock-free mutual exclusion algorithm for n processes that uses n shared bits.*

The proof consists of the so-called one-bit algorithm, which is shown in Figure 2.9.

It was developed by both J.E. Burns in 1981 and L. Lamport in 1986. There is a proof that this algorithm is deadlock-free and guarantees mutual exclusion. The procedure of the algorithm from the perspective of the process with the identification i can be explained as follows:

In lines (2)-(10), process i sets its want entry to True to indicate that it wants to enter the critical section. In doing so, it checks the want entries of all processes that have a smaller identification number. If these entries are all false (and its own entry is true), it successfully exits the loop and continues with line (11). If one of these want entries is true, process i sets its own entry to False, waits until the other entry becomes false, and then restarts the entire loop. In lines (11)-(12), process i checks the want entries of all processes that have a larger identification number than it does. If these are all false, then process i is allowed to enter the critical section. It has then read all want entries at least once as false since it entered the loop in line (2). In the exit code (line (14)), process i sets its want entry to false.

It should also be noted that the algorithm does not satisfy starvation freedom, is not fast (if only one process wants to enter the critical section, it must read all n bits), and is also not symmetric, since processes with smaller process identifications are given

Initial: for $i = 1, \dots, n$ want[i] = False,

Program of the i th process

```

loop forever
(1)  remaining code
(2)  repeat
(3)    want[i] := True;
(4)    local := 1;
(5)    while (want[i] = True) and (local < i) do
(6)      if want[local] = True then
(7)        want[i] := False;
(8)        await want[local] = False;
(9)        local := local + 1;
(10)   until want[i] = True;
(11)   for local := i+1 to n do
(12)     await want[local] = False;
(13)   Critical section
(14)   want[i] = False
end loop

```

Figure 2.9: One-bit algorithm

preference in entering the critical section (e.g., process 1 will only ever run through the **repeat** loop once).

The next theorem (by R. Alur and G. Taubenfeld) states that processes must always “wait” for an arbitrary amount of time before they can enter the critical section:

Theorem 2.4.3. *Assuming that there are only atomic read and write operations, the following applies: There is no (deadlock-free) mutual exclusion algorithm for 2 (or even n) processes that has an upper bound for the number of memory accesses (of the shared memory) that a process must perform before it is allowed to enter the critical section.*

Proof. Let $\mathcal{M}$ be a mutual exclusion algorithm for 2 processes. Let P_1 and P_2 be the two processes. The *computation tree* T_M of M is a binary tree whose nodes correspond to the states of the program. The tree is defined by the following rules:

- The root of T_M is the initial state, whereby the remaining code is ignored, i.e., both processes P_1 and P_2 are directly before entering the initialization code.
- The left child of a node is the state of the node after executing the next step of P_1 on the shared memory.

- The right child of a node is the state of the node after executing the next step of P_2 on the shared memory.
- A node is a leaf if and only if one of the two processes is in its critical section.

This means that the computation tree records all states after entering the initialization code until the critical section is reached, whereby “process-internal steps”, which only read or change local variables, are not explicitly shown. The nodes of the computation tree are marked with 1, 2, or (1 and 2), whereby the following conditions must be met:

- A leaf is marked with 1 or 2, depending on which process is in the critical section
- An inner node is marked with the markings of its two children (seen as a set, therefore with 1, 2, or (1 and 2))

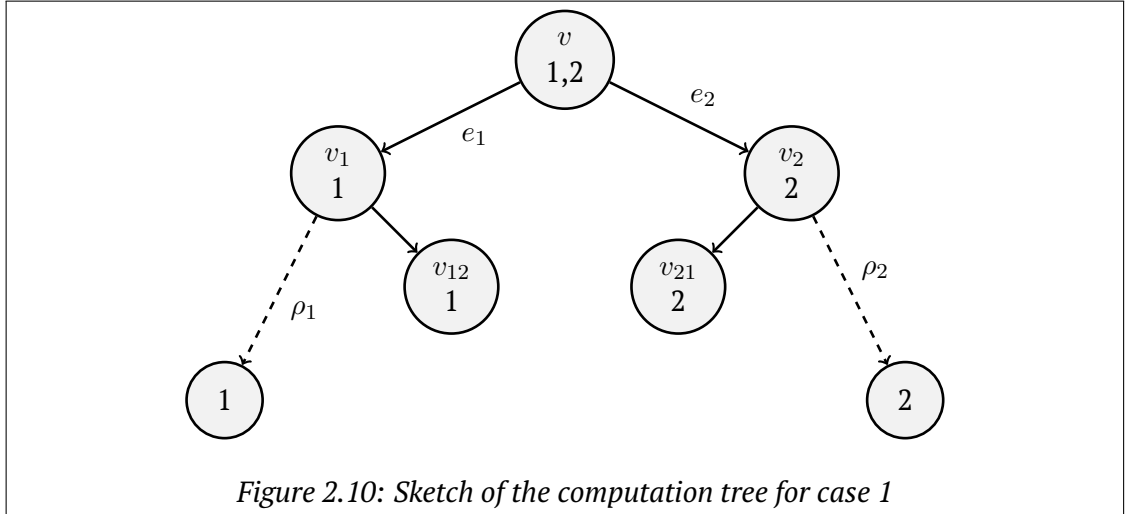
Two nodes v and w are *similar with respect to P_i* (written as $v \langle P_i \rangle w$) if

1. the sequence of steps that process P_i takes on the path from the root to v is exactly the same as the sequence of steps that process P_i takes on the path from the root to w , and
2. For the states associated with v and w , the following applies: The shared variables and the local variables of P_i are assigned the same values for v and w .

To prove the theorem, it suffices to show that for every $n > 0$ and $i \in \{1, 2\}$ there is a leaf v with label i such that more than n steps are executed for process P_i on the path from the root to v . This is sufficient because it ensures that no upper bound can be specified for the (active) waiting period until entering the critical section.

If there is an infinitely long path in T_M that contains infinitely many nodes marked with i , and process i takes infinitely many steps on the path, then it immediately follows that the above statement holds by choosing a prefix of the path that contains more than n steps from P_i and then extending the prefix to a leaf in which process i is in the critical section. From now on, only trees T_M in which such paths do not exist will be considered. For reference, this is assumption A. Since the critical section must be reached (deadlock freedom), the following applies: There is a node v marked with 1 and 2, such that one child of v is marked with 1 and the other child of v is marked with 2. Let v_1 be the left child and v_2 the right child of v , and let e_1 and e_2 be the computation steps from v to v_1 and from v to v_2 , respectively. Two cases are considered, and it is shown that both cases lead to a contradiction. Therefore, it follows that assumption A is never satisfied, and thus the theorem is proven.

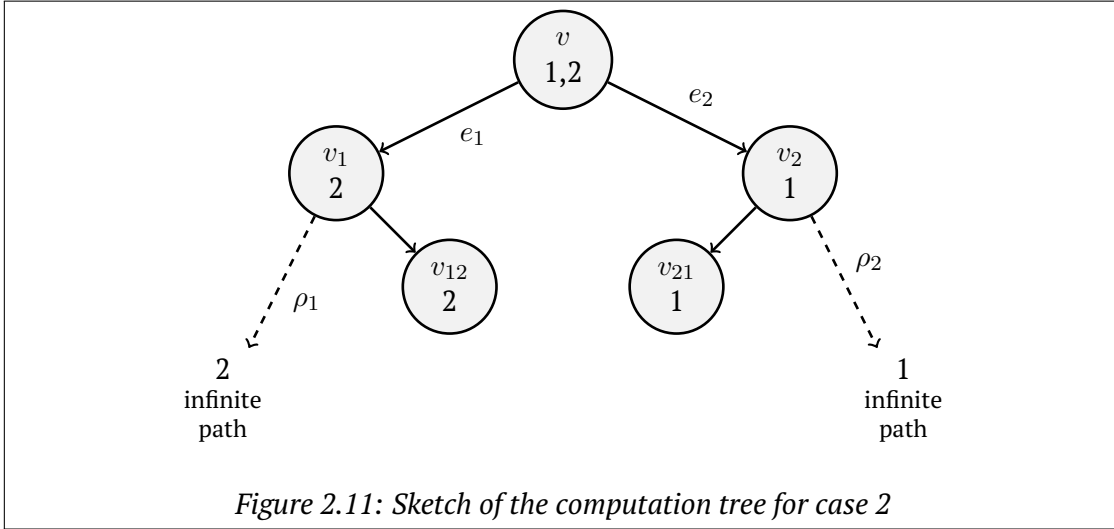
Case 1: v_1 is marked with 1, v_2 is marked with 2. The tree is sketched in Figure 2.10. The leftmost path starting from v_1 must be finite in length due to assumption A, since only P_1 computation steps are performed on this path. This path must also end with a leaf marked with 1. Let ρ_1 be the sequence of computation steps from v_1 to the leaf (this can be thought of as the edge marking of the path). For the rightmost path starting from v_2 , the analogous conclusion applies only to process 2: This path is finite and ends with a leaf marked 2. Let ρ_2 be the corresponding sequence of computations from v_2 to the rightmost leaf. The following cases can be distinguished:



- The step e_1 is a read operation: Then the values of the shared variables in the states v and v_1 are identical and thus $v \langle P_2 \rangle v_1$. This immediately implies $v_2 \langle P_2 \rangle v_{12}$. Since ρ_2 only performs steps from P_2 and ends in a leaf with label 2, it follows that v_{12} must also end in a leaf with label 2 on its rightmost path. This is a contradiction, since v_{12} is only labeled with 1.
- e_2 is a read operation: Then the argument can be made symmetrically to the previous case.
- e_1 and e_2 are write operations to different variables: Then for $i = 1, 2$: $v_{12} \langle P_i \rangle v_{21}$. However, it is then impossible for v_{12} to be marked only with 1 and v_{21} to be marked only with 2.
- e_1 and e_2 are write operations to the same variable. Then $v_2 \langle P_2 \rangle v_{12}$ applies, since executing e_2 in state v_1 reverses the write operations of e_1 . Now we can argue exactly as in the first case.

Case 2: v_1 is marked with 2 and v_2 is marked with 1. The computation tree is sketched in Figure 2.11. Since P_2 is not in the critical section in v , P_2 is also not in the critical section in v_1 (on the path from v to v_1 , P_1 performs one step!). This implies that every path starting from v_1 that ends in a leaf must perform at least one computation step for P_2 . Therefore, the leftmost path starting from v_1 must be infinitely long (no steps are performed for P_2 there!). Let ρ_1 be the sequence of steps on this path. Similarly, the following can be concluded for v_2 : The rightmost path is infinitely long. Let ρ_2 be the corresponding sequence of computation steps. Distinction between different cases:

- e_1 is a read operation. Then $v_2 \langle P_2 \rangle v_{12}$ applies. Therefore, ρ_2 can also be executed from v_{12} . Since ρ_2 only performs computations for P_2 , we obtain an infinitely long path consisting exclusively of P_2 steps and in which all nodes are marked with 2. This violates assumption A and results in a contradiction.



- e_2 is a read operation: Symmetrical to the previous case.
- e_1 and e_2 are write operations in different variables. Then, for $i = 1, 2$, the following applies: $v_{12} \langle P_i \rangle v_{21}$. This is impossible because v_{12} and v_{21} are each marked with different numbers.
- e_1 and e_2 are write operations to the same variable. Then $v_{12} \langle P_2 \rangle v_2$ applies, since the write operation e_1 from v to v_1 is effectively made invisible again by the operation e_2 . Now the same argument can be made as in the first case. $\square$

2.5 Stronger Memory Operations

In this section, we depart from the previous model, in which there are only atomic read and write commands to the shared memory. Various other so-called primitive operations are presented, whose atomic execution is supported by some hardware architectures. Deadlock-free mutual exclusion algorithms are relatively easy to implement with these more powerful primitives.

The other atomic (or indivisible) operations are defined as functions that receive a shared memory register (its address) as their first argument, which is explicitly identified by specifying the type **Register** for the argument. Depending on the operations, the functions receive additional arguments (of the types **register** (shared memory register), **local register** (local memory register), **value** (a value such as True, 1, etc.), **function** (a side-effect-free function on values)). The type of the returned result is also specified (using **returns:type**) if the function returns a value. The following applies to the execution of the functions: All steps after the call until the function is exited are executed atomically, i.e., to ensure that these steps are not mixed with other steps (from other processes), the entire function call is defined as *one* computation step (according to definition 1.5.1).

In what follows, no explicit separation between register addresses and the value of registers is used. As before, the register name is used to access the value (which corresponds to atomic reading). To write to a register, the assignment $x := \text{value}$ is used as before.

First, the known operations of reading and writing can be defined as functions in this pattern:

- $\text{read}(r)$ reads the value of the register with address r and returns the read value, i.e.,

```
function  $\text{read}(r : \text{register})$  returns : value  
  return( $r$ ) :  
end function
```

- $\text{write}(r, v)$ receives a register and a value and writes the value to the register, i.e.,

```
function  $\text{write}(r : \text{register}, v : \text{value})$   
   $r := v$   
end function
```

Since the *write* operation does not return a result, the **returns** part is omitted.

Note that reading and writing have not been explicitly noted by function calls in the algorithms so far. For example, this could now be done with the functions *read* and *write*. For example, instead of **if** $x = 0$ **then** ..., one could write **if** $\text{read}(x) = 0$ **then** ... and instead of $x := 10$, one could write $\text{write}(x, 10)$. However, for the sake of clarity, this will continue to be omitted. The function notation clearly shows, for example, that the assignment $x := y$, where x and y are registers, is not atomic, but contains two atomic operations: $\text{write}(x, \text{read}(y))$.

Now, some new atomic operations are defined:

- The *test-and-set* operation *tests* the value of a register (which corresponds to reading) and simultaneously sets a new value (which corresponds to writing). More precisely, the call is $\text{test-and-set}(r, v)$, where r is a register and v is a value. The value v is assigned to the register and the *old* value of the register is returned by the function, i.e.,

```
function  $\text{test-and-set}(r : \text{register}, v : \text{value})$  returns : value  
   $\text{temp} := r$ ;  
   $r := v$ ;  
  return( $\text{temp}$ );  
end function
```

In some other definitions or implementations of the test-and-set operation, the value v may only take the value 1 (and thus the second argument of the function *test-and-set* can be omitted). The memory location is then one bit (which can only take the values 0 or 1).

- $\text{swap}(r, l)$ receives a register and a local register and exchanges the values of both registers (atomically), i.e.,

```
function swap( $r$  : register,  $l$  : local register)  
     $temp := r$ ;  
     $r := l$ ;  
     $l := temp$ ;  
end function
```

The *swap* operation is sometimes also referred to as a *fetch-and-store* operation.

- $\text{fetch-and-add}(r, v)$ receives a register r and a value v . The value of r is increased by the value v . The old value of the register r is returned, i.e.,

```
function fetch-and-add( $r$  : register,  $v$  : value) returns : value  
     $temp := r$ ;  
     $r := temp + v$ ;  
    return( $temp$ );  
end function
```

There are also variants of this operation that do not use the value as an argument and always add exactly 1 (this operation is sometimes referred to as *fetch-and-increment*).

- $\text{read-modify-write}(r, f)$ receives a register and a function. The function is applied to the value of the register and then the resulting value is adopted as the new value of the register. The function returns the old value of the register. That is

```
function read-modify-write( $r$  : register,  $f$  : function) returns : value  
     $temp := r$ ;  
     $r := f(temp)$ ;  
    return( $temp$ );  
end function
```

Note that the operations *read*, *write*, *test-and-set*, and *fetch-and-add* can also all be expressed using *read-modify-write* by using suitable functions as arguments for *read-modify-write*. Note also that the function f should be side-effect-free. In particular, it would be problematic to use an atomic operation (e.g., *fetch-and-increment*) for the function, since then two atomic operations would be made into a new atomic operation by definition. For this reason, functions that manipulate memory are prohibited as arguments for *read-modify-write*.

- $\text{compare-and-swap}(r, old, new)$ expects a register r and two values *old* and *new*. If the value of the register is equal to *old*, then r is set to the value *new* and *True* is returned as the result of the function. Otherwise, the value of r is not changed

and False is returned, i.e.,

```
function compare-and-swap(r : register, old : value, new : value)  
    returns : value  
  
    if r = old then  
        r := new;  
        return(True);  
    else  
        return(False);  
end function
```

- *sticky-write*(*r*, *v*) expects a register and a value. The initial value of *r* is undefined (written as $\perp$). If the value of the register *r* is undefined (i.e., $\perp$) or equal to *v*, then the value is set to *v*. In this case, the operation was successful and returns True. Otherwise, False is returned. The *sticky-bit-write* operation describes the special case in which *r* can only take the values 0, 1, and $\perp$.
- *move*(*r*₁, *r*₂) expects two registers and sets the value of the first register to the value of the second register, i.e.,

```
function move(r1 : register, r2 : register)  
    temp := r2;  
    r1 := temp;  
end function
```

The assignment $x := y$ for two shared variables (which is prohibited in the atomic read and write model) is therefore precisely a *move* operation.

- *shared-swap*(*r*₁, *r*₂) expects two registers and swaps the values of the registers, i.e.,

```
function shared-swap(r1 : register, r2 : register)  
    temp1 := r1;  
    temp2 := r2;  
    r1 := temp2;  
    r2 := temp1;  
end function
```

The implementation of such atomic operations is not trivial. In what follows, it is assumed that the hardware provides such operations and executes them atomically.

It remains to be defined exactly what the objects, i.e., the data structures, look like (e.g., it makes little sense to assume a register with all of the above operations as a data structure). Therefore, some such objects are now defined, i.e., (concurrent) data structures with associated atomic operations:

Atomic register: A shared memory register that supports (atomic) *read* and *write* operations.

Test-and-set object: A shared memory register that supports (atomic) *write* and *test-and-set* operations. The *reset* operation is often added, which corresponds to writing a 0.

Test-and-set bit: A special test-and-set object: Only 0 and 1 are allowed as values of the register, whereby the *test-and-set* operation may only write a 1 and the *write* operation may only write a 0 (i.e., it only supports *reset*). Almost all hardware architectures support test-and-set bits.

Test-and-test-and-set object: A test-and-set object that additionally supports the atomic *read* operation.

Fetch-and-increment object: A shared memory register that supports the *fetch-and-increment*, *write*, and *read* operations.

Fetch-and-add object: A shared memory register that supports the *fetch-and-add*, *write*, and *read* operations.

Swap object: A shared memory register that supports the *swap* operation between the register and any local register.

Read-modify-write object: A shared memory register that supports the *read-modify-write*, *write*, and *read* operations.

Compare-and-swap object: A shared memory register that supports the *compare-and-swap*, *write*, and *read* operations.

Sticky bit: A shared memory register that supports the *read* and sticky-bit-write operations. Such a bit can have three values: 0, 1, or $\perp$.

Move object : Several shared memory registers that support the *move* operation in pairs (in addition to *write* and *read*).

Shared swap object : Several shared memory registers that support the *shared-swap* operation in pairs (in addition to *write* and *read*).

This list of objects is not exhaustive; there are other objects that can be found in the literature.

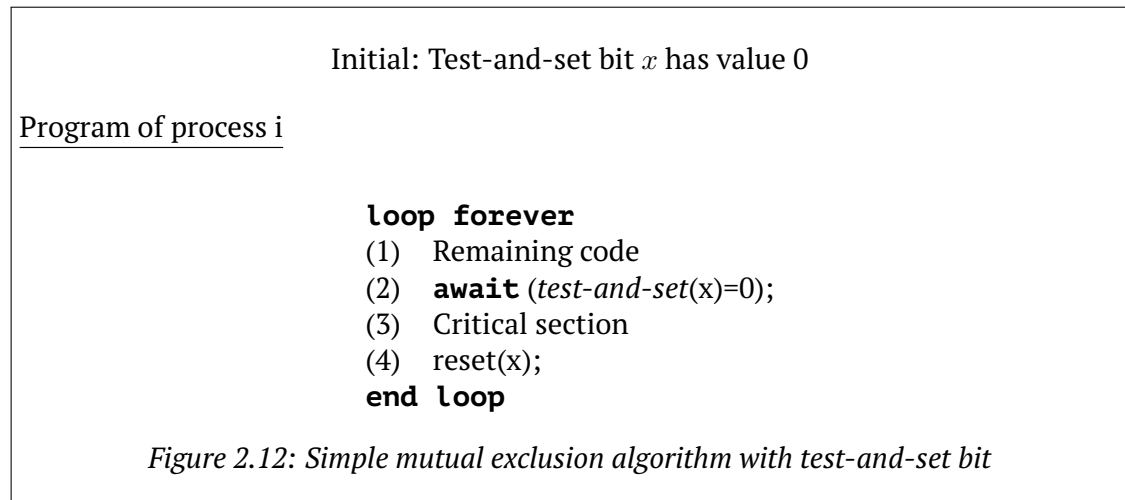
2.5.1 Mutual Exclusion with Test-and-Set Bits

This section introduces mutual exclusion algorithms that use test-and-set bits as objects. Since these objects only support 0 and 1 as values and the atomic operations *reset* (set to 0) and *test-and-set* with the value 1, *test-and-set(r)* is used instead of *test-and-set(r, 1)*.

Test-and-set bits are often also referred to as *lock*. The two operations *test-and-set* and *reset* are then referred to as *lock* and *unlock*. When a process notices that a lock is set, it often proceeds by waiting in a loop until it is allowed to set the lock, i.e., it

executes a loop of the form **while** *test-and-set*(lock)=1 **do skip**;. In other words, the process "spins" around the lock. This is why test-and-set bits are often referred to as *spinlocks*.

This technique makes it easy to implement a mutual exclusion algorithm. Figure 2.12 shows the code for process *i* for this simple algorithm.



The algorithm guarantees mutual exclusion and is deadlock-free. Both are easy to see: Suppose two processes *i* and *j* are in the critical section at the same time. Let process *i* be the first in the critical section. Then process *i* set the value of *x* to 1 when last executing (2). Then, before entering the critical section, process *j* must have executed line (2), so that *test-and-set*(*x*) returned the value 0. However, this is only the case if *x* has the value 0, which is impossible because *x* can only be set back to the value 0 by process *i* (using *reset*) after process *i* has left the critical section. This means that mutual exclusion is guaranteed. Regarding deadlock freedom: Since *x* only has the value 1 for a finite number of steps (only as long as a process is in the critical section and has not yet executed line (4)), a process will always pass the **await** condition after a finite number of steps. However, the simple algorithm also has disadvantages: It does not satisfy starvation freedom, since a waiting process can be repeatedly "overtaken" by another process. Another disadvantage is that the test-and-set operations constantly reset the value of *x* (again and again to 1) while waiting, even if 1 is the result. The disadvantage here is that changing the shared memory is costly from a hardware perspective. For example, the caches of different processors must be constantly updated.

A (more practical, less theoretical) improvement to the algorithm can be achieved by using test-test-and-set objects and thereby attempting to perform the wait with a *read* instruction instead of a *test-and-set* operation. Figure 2.13 shows this variant as a program for process *i*. It has the advantage that waiting processes rarely perform test-and-set operations, since they actively wait using read operations. Just like the previous algorithm, this algorithm guarantees mutual exclusion and is deadlock-free but not starvation-free.

Initial: Test-and-test-and-set bit x has value 0

Program of process i

```

loop forever
(1)  remaining code
(2)  await (x=0);
(3)  while (test-and-set(x)=1) do
(4)    await (x=0);
(5)  Critical section
(6)  reset(x);
end loop

```

Figure 2.13: Mutual exclusion algorithm with test-test-and-set bit

From a practical point of view, this algorithm is still rather poor for many processes, because as soon as one process performs a *reset*, all other waiting processes perform a *test-and-set* operation. In practice, so-called "exponential backoff" is often used. The idea here is that a process waits for a period of time after its attempt to set the lock fails. This time can be adjusted dynamically. In the exponential method, this time grows exponentially with each failed attempt, whereby it makes sense to use upper limits for the waiting period. Most program libraries provide a **pause** (or **delay**) operation that allows a single thread to wait for a number of milliseconds. This makes it easy to implement such algorithms. Figure 2.14 shows the program of the i th process of such an algorithm. Here, the waiting period is only increased if the lock was free once and the waiting process was unable to set it. This variant seems more practical in terms of practical experience compared to increasing the waiting period in the **await** statement.

Next, we discuss a fast algorithm for the mutual exclusion problem that guarantees starvation freedom. It was developed by R. Alur and G. Taubenfeld. Figure 2.15 shows the program code for the i th process.

The idea of the algorithm or the sequence of the program for process i can be explained as follows: In the initialization code, the process signals that it is entering the waiting phase by setting the entry `waiting[i]` (line (2)). Process i has two options for exiting the wait: Either it sets the lock on the test-and-set bit `lock` (in which case the local variable `key` is assigned the value 0), or another process sets the entry for process i (i.e., `waiting[i]`) to False. Before entering the critical section, process i signals that it has left the waiting area by setting `waiting[i]` to False. In short, the exit code has the task of not resetting the lock if possible, but rather passing it on to the next waiting process. To do this, the shared variable `turn` comes into play (`lturn` is only a local auxiliary variable): `turn` determines the next process that—if it is waiting—is allowed to enter the critical section. If `turn` has the value i , then the value is increased by 1 (since process i is leav-

Initial: Test-and-test-and-set bit x has value 0, delay: local variable, minDelay, maxDelay: constants

Program of the i th process

```

loop forever
(1)  remaining code
(2)  delay := minDelay;
(3)  repeat
(4)    delay := min(2*delay,maxDelay)
(5)    while (x=1) do pause(delay);
(6)  until (test-and-set(x)=0)
(7)  Critical section
(8)  reset(x);
end loop

```

Figure 2.14: Mutual exclusion algorithm with "exponential backoff"

ing the critical section). The modulo operation is used to set turn from n to 1 (lines (8)-(10)). In lines (11)-(15), it is first checked whether the process determined by turn is waiting. If this is the case, its waiting entry is set to False, i.e., the waiting process is allowed to enter the critical section, and the lock is transferred directly (without reset). If the process determined by turn is not waiting, the lock is reset so that all waiting processes can access it again, and turn is increased again because the process associated with turn is not waiting.

It is easy to verify that the following applies

Theorem 2.5.1. *The algorithm from Figure 2.15 guarantees mutual exclusion and is deadlock-free.*

However, the algorithm fulfills stronger properties. It is starvation-free and fulfills the so-called n -fairness, which is defined as:

Definition 2.5.2. *A mutual exclusion algorithm satisfies r -fairness if and only if the following always holds:*

A waiting process has the opportunity to enter the critical section before all other processes together can enter the critical section $r + 1$ times.

In contrast to r -bounded waiting, r -fairness counts how often all processes together (summed) can enter the critical section before a waiting process is allowed to enter the critical section. However, the terms are closely related:

Proposition 2.5.3. *The property r -fairness implies the property r -bounded waiting. For n processes, the property r -bounded waiting implies the property $((n - 1) \cdot r)$ -fairness.*

Initial:

turn: atomic register, value at the beginning between $1 \dots n$;

lock: test-and-set bit value, value at the beginning 0;

waiting: field (index: $1 \dots n$) of atomic registers, value for $i = 1, \dots, n$: False;

lturn, key: local registers, initial value irrelevant

Program of the i th process

```

loop forever
(1)  remaining code
(2)  waiting[i]:=True;
(3)  key := 1;
(4)  while (waiting[i] and key=1) do
(5)      key := test-and-set(lock);
(6)  waiting[i]:=False;
(7)  Critical section
(8)  if turn=i
(9)      then lturn := 1 + (turn mod n);
(10) else lturn := turn
(11) if waiting[lturn]
(12) then turn := lturn;
(13)     waiting[lturn] := False;
(14) else turn := 1+(lturn mod n);
(15)     reset(lock);
end loop

```

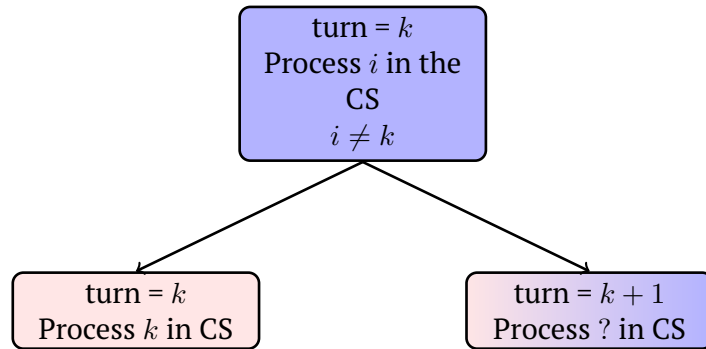
Figure 2.15: Starvation-free algorithm with test-and-set bit

Proof. The first implication is obvious: If only r processes in total are allowed to enter the critical section, then each individual process can also only enter the critical section r times. The second implication can be calculated analogously: If all other $(n - 1)$ processes are allowed to enter the critical section at most r times each, then there are at most $((n - 1) \cdot r)$ entries into the critical section in total. $\square$

Theorem 2.5.4. *The algorithm from Figure 2.15 is starvation-free and guarantees the n -fairness property for n processes.*

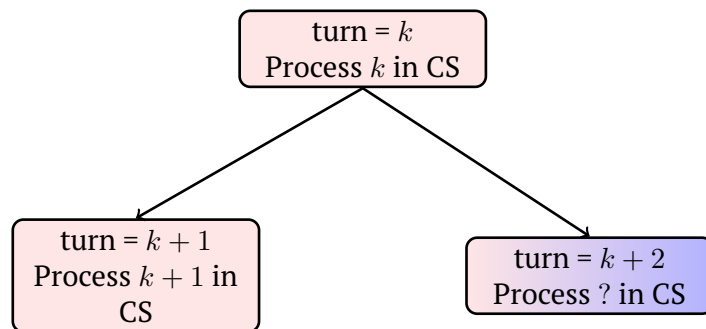
Proof. First, we prove n -fairness by considering two cases:

- If process i leaves the critical section and $\text{turn} \neq i$, then there can be two possibilities for the subsequent process in the critical section:



Either process k enters the critical section and turn is not changed beforehand, or any process enters the critical section, but turn is increased by 1.

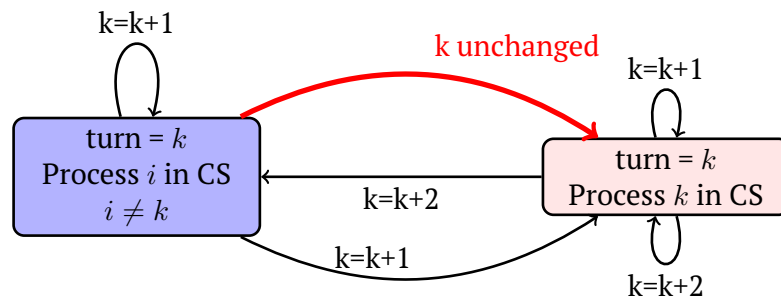
- If process k leaves the critical section and $\text{turn} = k$, there can also be two possibilities for the next process in the critical section:



Either turn is increased by 1 to $k+1$ and process $k+1$ is then in the critical section, or turn is increased twice because process $k+1$ does not wait. Then any process is in the critical section.

The coloring of the nodes is not random: states in which the value of turn is equal to the process in the critical section are red, others are blue. In mixed states, both cases may apply.

If we combine the cases, we can create the following diagram:



Let process l now be a waiting process, i.e., $\text{waiting}[l] = \text{True}$. Then the diagram shows that after leaving the critical section n times, at some point $\text{turn} = l$ must have been true. It is important to note that whenever the red arrow was chosen, it can only be chosen again after turn has been increased by 2. This is important because the critical section can be exited via the red arrow without increasing turn . Since $\text{turn} = l$ was true at some point and $\text{waiting}[l] = \text{True}$ is true, process l must enter the critical section at the latest after executing the exit code n times. Note: Leaving n times (instead of $n - 1$) may be necessary if the red arrow is selected at the beginning.

Starvation freedom follows from deadlock freedom and n -fairness. $\square$

2.5.2 A Mutual Exclusion Algorithm with RMW Object

In the following, another algorithm for the mutual exclusion problem is presented – the *ticket algorithm*. The algorithm was developed by M.J. Fischer, N.A. Lynch, J.E. Burns, and A. Borodin in 1989 and uses a read-modify-write object. The idea behind the algorithm is similar to that of the bakery algorithm: In the doorway, a process draws a ticket (a number) and increases the number for the next process. With the help of the read-modify-write object, this step can be performed atomically. In contrast to the bakery algorithm, however, the numbers for the tickets are not arbitrarily large, but are only increased modulo n .

The program for the i th process of the ticket algorithm is shown in Figure 2.16.

Note that $\text{ticket} = \text{valid}$ must be true at the beginning so that the very first process can enter the critical section. Mutual exclusion applies because valid is always increased only when the exit code is executed. Deadlock freedom is also easy to see. In addition, the algorithm satisfies the FIFO property, i.e., the order of entry into the critical section corresponds to the order of ticket drawing.

Theorem 2.5.5. *The ticket algorithm guarantees mutual exclusion, is starvation-free, and has the FIFO property.*

With regard to the value range of the RMW object, the ticket algorithm uses n^2 values (each n for ticket and n for valid). This is the best known algorithm with regard to this size if the (strong) FIFO property is required.

2.5.3 The MCS Algorithm with Queue

The last algorithm presented in this chapter is the MCS algorithm. It was published by J.M. Mellor-Crummey and M.L. Scott in 1991. The idea is that the waiting processes line up in a queue. The construction and deconstruction of the queue is accomplished with the help of a compare-and-swap object, which must also support the swap operation. Another special feature of the algorithm is that the exit code does not quite correspond to our assumptions, because it is not “wait-free”.

The algorithm uses *pointers* to create a linked list. The following notation is used for a pointer p :

Initial:

(ticket,valid): Read-Modify-Write object for a pair of numbers in the range $1 \dots n$ where initially ticket = valid

(ticket_{*i*}, valid_{*i*}) for $i = 1, \dots, n$: local register

Program of the *i*th process

loop forever

(1) remaining code

// increase ticket (read old values):

(2) (ticket_{*i*},valid_{*i*}) := read-modify-write((ticket,valid), inc-fst)

(3) **while** ticket_{*i*} ≠ valid_{*i*} **do**

(4) valid_{*i*} := valid;

(5) Critical section

(6) read-modify-write((ticket,valid), inc-snd) // increase valid

end loop

Functions

function inc-fst((a,b))

return (1+(a mod *n*), b)

end function

function inc-snd((a,b))

return (a, 1+(b mod *n*))

end function

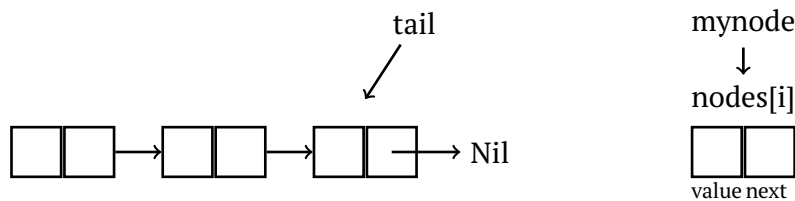
Figure 2.16: Ticket algorithm

- $*_p$ as an indirection operation, i.e., $*_p$ returns the object pointed to by the pointer.
- $\&p$ gives the memory address of an object and can be used to create pointers to the object.

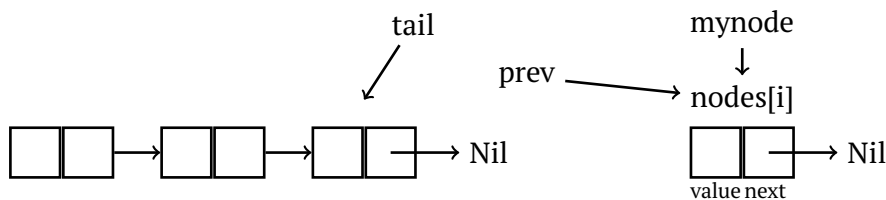
The algorithm builds linked lists, whereby a list element has two components (and can be represented using a record): The value of the element and a pointer to the next element. Let e be a list element, then let $e.value$ be the value and $e.next$ be the pointer to the next element. Pointers in the MCS algorithm point either to list elements or to nil (empty list, i.e., for the end of the list, a pointer points to nil).

Figure 2.17 shows the program of the i th process and an "overview" of the variables. The individual phases of the algorithm are explained in detail. The pointer tail always points to the last list element. If the list is empty (e.g., at the beginning), tail points to nil. The field entries $nodes[i]$ are list elements for the i th process. If a process wants to enter the critical section, it appends its list element to the list.

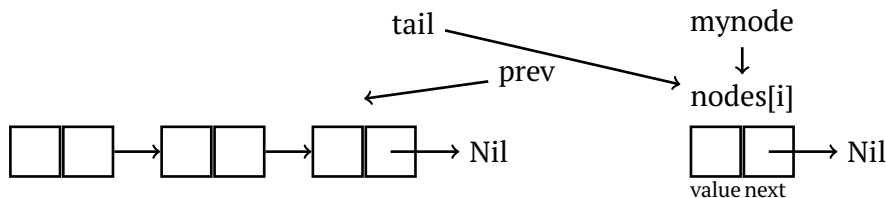
In the initialization code, the next pointer of the $nodes[i]$ entry is first set to nil (line (2)). This nil will later mark the new end of the list. In the next step, the local pointer prev is created, which points to $nodes[i]$ just like mynode. In line (4), the tail pointer is now exchanged with the prev pointer. The swap operation ensures that this happens atomically in one step. From this step onwards, tail points to $nodes[i]$ – the new last list element, although the actual appending to the list is still missing. These steps can be illustrated with the following box-and-pointer diagrams:



After executing lines (2) and (3):



After executing line (4)



Types:

- Record Element: two attributes: value:Bool, and next: pointer to an element

Shared variables:

- nodes[i]: Field entry: Content is a record of type Element
- tail: Swap and Compare-and-Swap object of type: Pointer to an element, initial value: Nil

Local variables:

- mynode: pointer to an element, initially pointing to nodes[i]
- prev, succ: pointers to elements

Program of the i th process

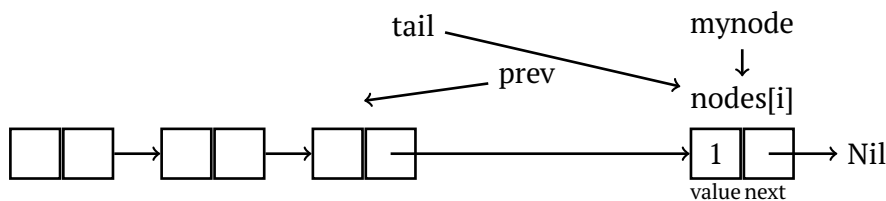
```

loop forever
(1)  remaining code
(2)  *mynode.next := Nil;
(3)  prev := mynode;
(4)  swap(tail,prev);
(5)  if prev  $\neq$  Nil then
(6)    *mynode.value := 1;
(7)    *prev.next := mynode;
(8)    await *mynode.value = 0;
(9)  Critical section
(10) if mynode.next = Nil then
(11)   if compare-and-swap(tail,mynode,Nil) = False then
(12)    await *mynode.next  $\neq$  Nil;
(13)    succ := *mynode.next;
(14)    *succ.value := 0;
(15) else
(16)   succ := *mynode.next;
(17)   *succ.value := 0;
end loop

```

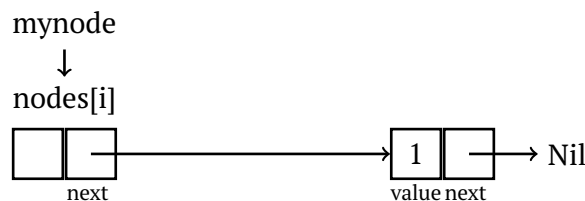
Figure 2.17: The MCS algorithm

Line (5) checks whether there are elements in the list (if so, you have to wait). If the pointer `prev` points to `nil`, then it is clear that `tail` previously pointed to `nil`, meaning that the list was empty. In this case, the critical section can be entered. If this is not the case, lines (6)-(8) are executed: the value entry of `nodes[i]` is set to 1, the element `nodes[i]` is appended to the list (by setting the next pointer of the previous element), and then the program waits until the value entry of `nodes[i]` is set to 0. In the illustration, this results in the following situation:

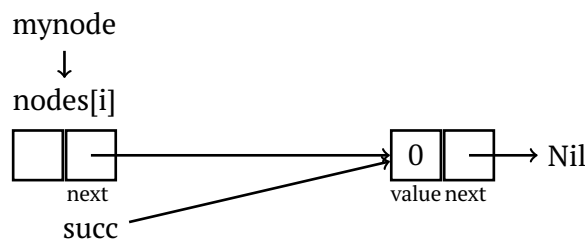


In the exit code, it is first checked whether the next entry points to `nil` or an element. If the pointer points to an element, then another process has been pending in the meantime. Its value is then set to 0 in lines (16) and (17) so that it can leave the wait.

This can be illustrated as follows: When querying in line (10)



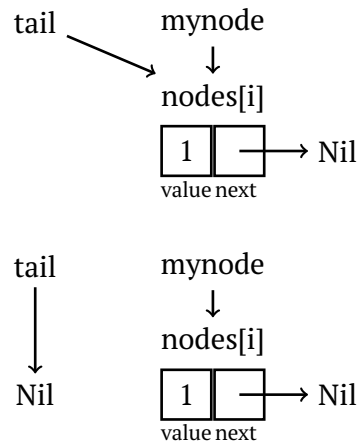
Since a successor exists, its value is set:



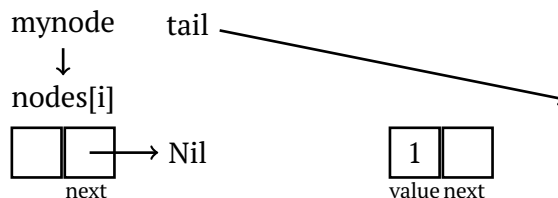
Note that the process in the exit code does not have to explicitly depend on the list, as it must be at the top of the list (i.e., the old pointer entries are irrelevant).

If the **if** condition in line (10) is satisfied, two cases may apply: Either the process in the exit code is really the last list element and there is no further process in the initialization code, or there are further processes in the initialization code, but they have not yet changed the next pointer of the process in the exit code (this happens in line (7)). The distinction between these two cases is made by a compare-and-swap

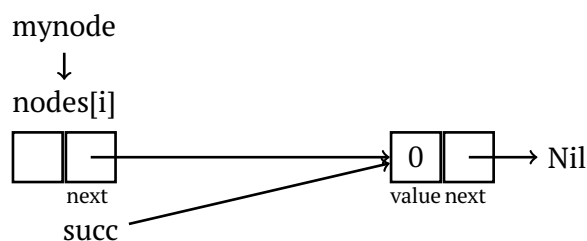
operation: If both pointers *tail* and *mynode* point to the same element (namely *nodes[i]*, if process *i* is in the exit code), then no other process has performed the swap operation in line (4). In this case, the compare-and-swap operation sets the *tail* pointer to *nil*, and the process in the exit code hangs itself; the list is empty. This case can be illustrated by the two box-and-pointer diagrams:



In the other case, the *compare-and-swap* operation in line (11) fails, i.e., the *tail* pointer is not changed. Now the process waits in the exit code until its own next pointer is changed by the process in the initialization code: Such a process must exist because the *tail* pointer has been changed and is no longer equal to *mynode*. Once this has happened, the process continues exactly as before: The value entry of the successor is set from 1 to 0. The waiting situation can be illustrated by the following image: First, *tail* no longer points to the current element *nodes[i]*, but the next pointer still points to *nil*.



Now we wait until the next pointer is changed, then the value entry of the successor is set to 0:



It can be shown that the following applies:

Theorem 2.5.6. *The MCS algorithm guarantees mutual exclusion, is starvation-free, and fulfills the FIFO property.*

Another property of the MCS algorithm is that the individual processes always execute their wait operations on independent memory locations, i.e., no memory entries are repeatedly queried by multiple processes. This has a positive effect in implementations, as changes to these memory locations rarely need to be communicated to all processes. This is particularly useful in situations where many processes are created. Here, it can be seen that the MCS algorithm performs very well, as there is no waiting on the lock variable (here tail).

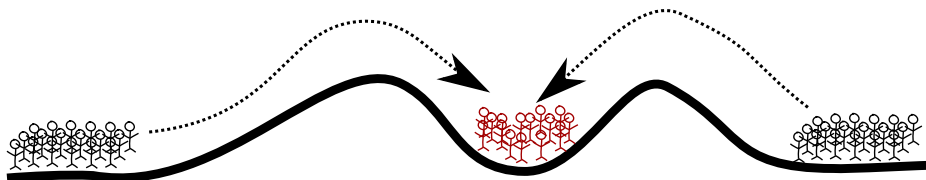
2.6 Consensus and the Herlihy Hierarchy

In the previous sections, various memory operations and corresponding concurrent objects were introduced and used. The question arises as to whether the different objects can be compared in terms of their *expressiveness*. Intuitively, objects that only provide *read* and *write* as atomic operations are the weakest (e.g., the mutual exclusion problem is rather difficult to solve with them), and RMW objects are strong because they can be used to express many other objects. Maurice Herlihy (Herlihy, 1991) developed a general approach to comparing expressiveness and then established a hierarchy of objects. This section briefly presents Herlihy's results.

2.6.1 Process Model with Crashes

First, Herlihy's approach uses a slightly different model, which is adopted for this section: Processes can crash at any time: Each process can hang at any command forever, but individual commands are executed atomically, i.e., either completely or not at all. Note that none of the mutual exclusion algorithms presented work for this model, since processes can crash even in the critical section or in the exit code, for example, and thus cause a deadlock.

A good example in this model is the so-called "coordinated attack" problem or "two generals" problem: The situation of the problem is as follows: Two divisions want to attack an enemy division. The enemy division is located in the valley between two hills. The attacking divisions are located to the left and right behind the hills.



The attacking divisions will certainly lose if they do not attack at the same moment. Therefore, they must coordinate in order to attack at the same moment. The generals

only have messengers at their disposal for communication, which they can send to the opposing division. The messenger must pass through enemy territory and may therefore be captured by the enemy, meaning that it is not certain whether the messenger will actually reach the other general. The general on the right sends a messenger to inform the general on the left that the attack is to take place at 12 noon. Since it is not certain whether the messenger has arrived, the right general will wait for the left general to send a messenger back to confirm that the message has arrived. However, the left general cannot be sure that his messenger has actually arrived and therefore cannot be sure that the attack will actually take place. As a possible solution, the right-wing general then sends another messenger to confirm receipt. But even he cannot be sure that the message will actually arrive. This back-and-forth communication therefore continues indefinitely, and the attack will not take place. In fact, it is relatively straightforward to prove that the coordinated attack problem cannot be solved in this model. Note that the difficulty/unsolvability arises solely from the fact that messengers can fail. Applied to the process model, such a messenger would be a process that could crash under certain circumstances.

2.6.2 The Consensus Problem

To investigate the expressiveness of concurrent objects, Herlihy considered the so-called *consensus problem*. Let there be n processes. Each process i receives an input value $x_i \in \{0, 1\}$. The consensus problem consists of programming the processes in such a way that all processes decide on a common value $d \in \{0, 1\}$ (this value is called the *decision value*). The decision value must satisfy the following conditions:

- *Agreement*: All non-crashed processes decide on the same value d .
- *Validity*: $d \in \{x_1, \dots, x_n\}$, i.e., d corresponds to one of the input values.

If there were no crashing processes, it would be easy to solve the consensus problem: All processes share all their input values with each other (via shared memory), then each process calculates $f(x_1, \dots, x_n) = d$, where f is a fixed function known to all processes, e.g., $f(x_1, \dots, x_n) = x_1$. However, since processes are allowed to crash, the problem is no longer easy to solve. Note that the "consistency condition" also implies that the algorithm to be designed must be wait-free, i.e., regardless of whether the other processes crash or not, the following must apply: If process i does not crash, it decides on the decision value d after a finite number of steps, or conversely: Process i does not run the risk of getting stuck in an infinite loop.

With the help of a three-valued RMW object, it is very easy to solve the consensus problem. The corresponding code can be found in Figure 2.18. The RMW object can take the values $\perp$, 0, or 1 and initially has the value $\perp$. Each process attempts to write its input value to the RMW object, but only if it has the value $\perp$, otherwise the value remains unchanged. This means that the first successful process sets the value and all other processes adopt the value. In particular, this also means that the value d_i is identical for all non-crashing processes. Crashing processes are not a problem here,

as they do not affect the value and access to the RMW object. This means that the algorithm from Figure 2.18 represents a wait-free solution to the consensus problem for any number of processes.

Objects and initialization:

x : Read-modify-write object with the possible values $\perp, 0, 1$, initially $\perp$

x_i : Input value of process i

d_i : Decision value made by process i .

Program of process i

- (1) $d_i := \text{read-modify-write}(x, f_i)$;
- (2) **if** $d_i = \perp$ **then**
 $d_i := x_i$;

Function f_i of process i

```
function  $f_i(v)$ 
  if  $v = \perp$  then return  $x_i$ 
  else return  $v$ 
end function
```

Figure 2.18: Consensus with a three-valued RMW object

However, an important prominent result (also known as the “FLP impossibility result”, named after its authors Fischer, Lynch, and Paterson, (Fischer et al., 1985)) is that with atomic *read* and *write*, the consensus problem cannot be solved, even for two processes and only one crash:

Theorem 2.6.1. *There is no consensus algorithm for atomic read and write that can tolerate a crash.*

Proof. We outline the proof, essentially using the presentation and arguments from (Herlihy & Shavit, 2008). We do not assume that processes crash, but that there is a wait-free consensus algorithm for 2 processes that only uses atomic registers and refute its existence.

Similar to the proof of Theorem 2.4.3, we use computation trees for the execution of two processes that perform a consensus computation, where the nodes represent states of the processes and the memory, and the edges represent *essential computation steps* of the processes P_1, P_2 . An *essential step* is an operation on a shared memory object. Since we are only arguing with atomic registers here, these are read and write operations. An edge to the left child represents an essential step of process P_1 , and an edge to the right child represents an essential step of process P_2 . A leaf is a state in which both processes have decided on a (shared) decision value. An initial state is a state before any process has taken an essential step. Since the consensus algorithm must be wait-free, the computation tree is actually a finite tree. Since leaves have a unique decision value,

we mark them with this value, which is either 0 or 1. A node is *bivalent* if the decision value has not yet been determined (i.e., leaves with marks 0 and 1 are reachable from it). Otherwise, we call a node *univalent* or *1-valent* if the decision value is 1, and *0-valent* if 0 is the decision value.

Claim 1: Every 2-process consensus algorithm has a bivalent initial state. Proof of Claim 1: Assume P_1 has input value $x_1 = 0$ and P_2 has input value $x_2 = 1$. If only P_1 takes steps (leftmost path in the tree), then P_1 must compute 0 as the decision value; conversely, if P_1 never takes a step, then P_2 must compute 1 as the decision value.

A node is *critical* if it is bivalent and each successor is univalent.

Claim 2: Every computation tree for a consensus algorithm that starts with a bivalent initial state has a critical node.

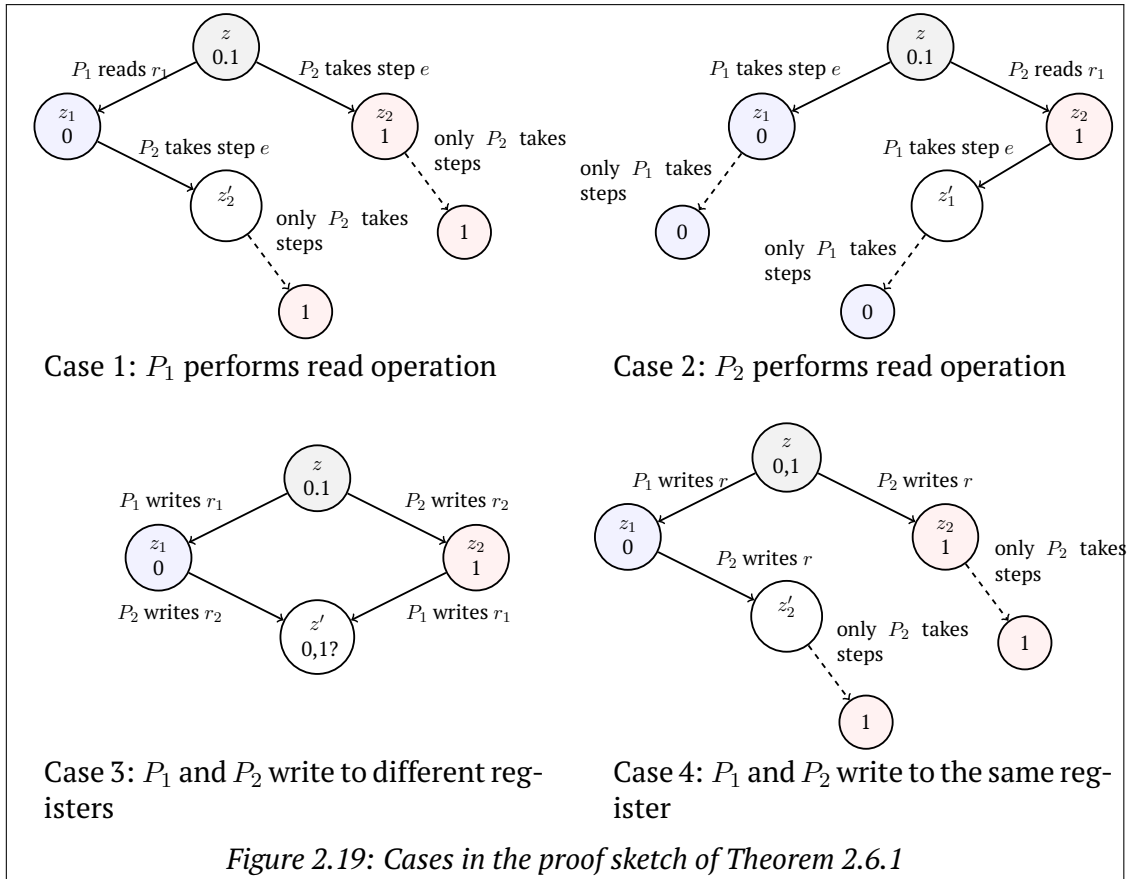
Proof of Claim 2: Suppose this is not true. Start with a bivalent initial state. As long as there are steps that lead to a bivalent state, execute these steps. If this is possible indefinitely, then the algorithm is not wait-free, which is a contradiction.

Now assume there is a consensus algorithm for 2 processes. We execute the algorithm up to a critical bivalent state z . The children of this state are univalent, one 0-valent and one 1-valent. Let the left child z_1 be 0-valent and the right child z_2 be 1-valent. We consider the steps from P_1 and P_2 to their children (see Figure 2.19)

- If P_1 performs a read operation and P_2 performs a read or write operation e , then P_2 can also perform the operation e from z_1 . The resulting state is identical to z_2 from the perspective of P_2 . Therefore, P_2 can execute the entire rightmost path to the leaf from there. This leaf must be 1-valent. However, then z_1 cannot be 0-valent. Contradiction.
- If P_2 performs a read operation and P_1 performs a read or write operation e , then P_1 can also execute the operation e from z_2 . The resulting state is identical to z_1 from the perspective of P_1 . Therefore, P_1 can execute the complete leftmost path to the leaf from there. This leaf must be 0-valent. However, then z_2 cannot be 1-valent. Contradiction.
- If P_1 and P_2 each perform write operations on different registers, then P_2 can also perform the write operation from z_1 and P_1 from z_2 , and the resulting state z' is the same in each case. However, this cannot be univalent! Contradiction.
- If P_1 performs a write operation and P_2 performs a write operation on the same register, then P_2 can also perform the write operation from z_1 and then follow the same steps as on the rightmost path. This path must end in a leaf that is 1-valent. Then z_1 cannot be 0-valent. Contradiction.

□

A more detailed proof can be found in the literature. As a first conclusion, it can be stated that the consensus problem for atomic *read* and *write* is unsolvable, but for a three-valued RMW object it is easily solvable for any number of processes and crashes. One consequence is that an RMW object obviously cannot be implemented (or simulated) by atomic *read* and *write* operations.



2.6.3 The Consensus Number

Based on the consensus problem, the following *consensus number* can be defined for a concurrent object:

Definition 2.6.2 (Consensus number). *For a concurrent object of type o , the consensus number $\mathbb{CN}(o)$ is the largest number of processes n for which the consensus problem for n processes can be solved by using any number of objects of type o and any number of atomic registers (with read and write). If the number is unlimited, then $\mathbb{CN}(o) = \infty$.*

The following table shows some consensus numbers

$\mathbb{CN}(o)$	Object o
1	atomic register with <i>read</i> and <i>write</i>
2	test-and-set object, fetch-and-increment object, fetch-and-add object, swap object, read-modify-write bit
$\Theta(\sqrt{m})$	swap ^{m} object
$2m - 2$	m -register with m -fold assignment ($m > 1$)
∞	(three-valued) RMW object, compare-and-swap object, sticky bit

It is worth mentioning that the algorithm from Figure 2.18 can be easily adapted to a compare-and-swap object and also to a sticky bit and therefore these objects, just like the three-valued RMW object, have the consensus number ∞ . The table also contains some multi-objects: A swap ^{m} -object allows m memory locations to be swapped in an atomic step, an m -register with m -fold assignment allows the atomic (multiple) assignment of m shared registers.

An important result regarding this hierarchy is:

Theorem 2.6.3. *If o_1 and o_2 are two objects with $\mathbb{CN}(o_1) < \mathbb{CN}(o_2)$, then the following applies for a system with $\mathbb{CN}(o_2)$ processes:*

- *There is no wait-free implementation of object o_2 using only objects of type o_1 and atomic registers.*
- *There is a wait-free implementation of object o_1 using only objects of type o_2 and atomic registers.*

Sketch. The first statement follows easily by contradiction: Suppose an object of type o_2 can be implemented wait-free using objects of type o_1 and atomic registers. Let $n = \mathbb{CN}(o_2)$. Then there is a consensus algorithm for n processes with objects of type o_2 and atomic registers. Replace all o_2 objects there with the implementation using o_1 objects. This results in a correct consensus algorithm for n processes with objects of type o_1 and atomic registers. Therefore, $\mathbb{CN}(o_1) = n$, which contradicts the assumption $\mathbb{CN}(o_1) < \mathbb{CN}(o_2)$.

The second statement is not so easy to prove. It follows from the next theorem 2.6.4, since this implies that for $n = \mathbb{CN}(o_2)$ processes, every concurrent object can be implemented wait-free with a sequential description using only o_2 -objects and atomic registers. This means that o_1 -objects for $n = \mathbb{CN}(o_2)$ processes can also be encoded with it. $\square$

This shows that objects with higher consensus numbers are more expressive. Another result is that objects with a consensus number of at least n are so-called *universal objects*:

Theorem 2.6.4. *An object o with $\text{CN}(o) \geq n$ is universal in a system with n processes. This means that every wait-free object (with a sequential description, such as the object descriptions at the beginning of Section 2.5) can be implemented by objects of type o and atomic registers in a system with n processes.*

We will refrain from providing the exact proof of the theorem (this can be found in the original in (Herlihy, 1991) or, for example, in the textbooks (Taubenfeld, 2006) or (Herlihy & Shavit, 2008) can be found in the original). The essential idea of the proof is that each process stores the accesses to the concurrent objects in a large table and each process calculates the successor states of the objects until its access takes place. This computation itself is not a real problem due to the sequential description. The two hurdles in the proof are that, on the one hand, an order must be specified for when which operations of which process are executed and, on the other hand, it must be ensured that this execution cannot be delayed indefinitely (in order to obtain a wait-free implementation). The consensus algorithm is used for the first hurdle to determine which process is allowed to execute the next operation. The second hurdle is overcome by cleverly processing all operations by all processes.

The universality of an object means that sequentially writable objects (which guarantee wait-freedom) can be encoded by the object. In other words, the special property of the consensus number also makes statements about the object in general. The conclusion from the theorem is that objects with a consensus number of n (or greater) are to be preferred, and that one type of such objects is sufficient.

2.7 References

The presentation of the assumptions and algorithms mostly follows (Taubenfeld, 2006). Parts (e.g., the variable naming of some algorithms) follow (Ben-Ari, 2006). The original publications of the presented algorithms, correctness proofs, and complexity results can all be found in (Taubenfeld, 2006). A comprehensive presentation of the consensus problem and consensus algorithms (from which the proof sketch for Theorem 2.6.1 also originates) can be found in (Herlihy & Shavit, 2008).

3

Programming Primitives

After examining primitive operations provided by hardware in the previous chapter, this chapter discusses programming abstractions, i.e., primitives provided by programming libraries of concurrent programming languages. These programming constructs generally make programming more convenient. Alongside the presentation of these constructs (and corresponding model adaptations), some (classical) examples/problems of concurrent programming are explained and solutions are developed using programming abstractions.

3.1 Extensions of the Process Model

To describe the properties of the various programming abstractions, the model is first modified slightly by assigning different states (or phases) to processes. Control over which phase a process is in, or how the change from one state to another occurs, generally lies with the operating system (for operating system processes) or with the runtime environment of the programming languages.

If there are enough processors available for all running processes, then normally all processes are in the *running* state. If there are too few resources (e.g., processors) available, not all processes can be running. A process that wants to run but cannot do so due to a lack of resources is referred to as *ready*. During the execution of a concurrent program, at least one process should always be running and any number of other processes should be ready. Even if there is no actual process that wants to perform work, an idle process usually runs.

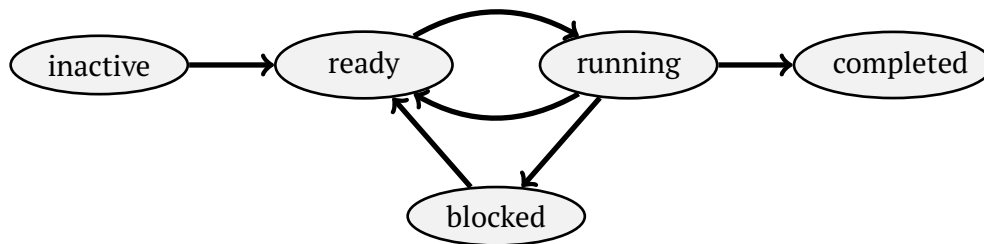
The scheduler is responsible for stopping running processes (and thus putting them into the “ready” state) to allow other ready processes to run. This switching of the state of individual processes is called a “context switch”. As before, no assumptions are made about how the scheduler behaves: All “interleavings” are possible with the sole restriction that fairness must apply, i.e., every ready process must be set to the “running” state after a finite number of computation steps. To identify or change the state of processes, the notation $P.state$ is used, where P is a process and the attribute state stores the current state of the process. This means, for example, that if the scheduler sets a running process p to the “ready” state, the value of $p.state$ changes from running to ready.

For the programming constructs described in the following sections, it is assumed that there is another state for processes: the state *blocked*. A blocked process is not ready, so it cannot be set to the “running” state by the scheduler. A blocked process

is unblocked (or awakened) when an operation sets it from the “blocked” state to the “ready” state. This operation is not performed by the scheduler, but by a program instruction. Subsequently, the (ready) process can be set back to the “running” state by the scheduler through a context switch.

Two further states can be defined for processes: A process is initially “inactive” until it is ready at some point. When a process is finished, i.e., no further computation steps are to be performed, it is “completed”.

Overall, the different states of the process status and their possible transitions can be summarized in a diagram as follows:



3.2 Semaphores

The term semaphore actually originates from railway traffic, where they are or were used as mechanical signal transmitters. In computer science, a semaphore is an abstract data type with (usually) two components:

- A non-negative integer V .
- A set of processes M (or their identifications).

Let S be a semaphore, then $S.V$ denotes the non-negative integer and $S.M$ denotes the set of processes.

Let **newSem**(k) be an operation that creates a new semaphore S and initializes $S.V$ with k and $S.M$ with the empty set. For a semaphore S , the two *atomic* operations **wait** and **signal** are available. Atomic here means that the programming language (the compiler, interpreter, runtime environment) ensures that the corresponding operations are performed in an indivisible computation step. The compiler itself can use the known primitives of the computer architecture to implement the semaphore data type.

Remark 3.2.1. In Dijkstra’s original article, the operation is not called **wait**(S), but **P**(S), and the operation **signal**(S) is called **V**(S). The designations **P** and **V** come from Dutch terms: “ V ” stands for “verhoog” (Dutch for “increase”), and “ P ” stands for “prolaag”, which is a made-up word introduced by Dijkstra, since “verlaag” (Dutch for “lower”) also begins with the letter “ V ”. The terms **down** and **up** are often used instead of **wait** and **signal**.

Let P be the process that calls **wait** or **signal** for the semaphore S . Then the semantics of the two operations can be described by the following program code: First, the **wait** operation:

```
procedure wait( $S$ )  
  if  $S.V > 0$  then  
     $S.V := S.V - 1$ ;  
  else  
     $S.M := S.M \cup \{P\}$ ;  
     $P.state := \text{blocked}$ ;
```

The **wait** operation first reads the V component. If it is greater than 0, it is decremented by 1. Otherwise ($S.V$ is 0), the calling process P is added to the set $S.M$ of processes and its status is set to “blocked”.

The procedure for the **signal** operation can be described as follows:

```
procedure signal( $S$ )  
  if  $S.M = \emptyset$  then  
     $S.V := S.V + 1$ ;  
  else  
    select an element  $Q$  from  $S.M$ ;  
     $S.M := S.M \setminus \{Q\}$ ;  
     $Q.state := \text{ready}$ ;
```

The **signal** operation unblocks a blocked process (and sets its state to “ready”) if there are blocked processes, i.e., if the set of processes is not empty. If the set contains several processes, any process is selected from the set. If there are no blocked processes, the counter is incremented.

Note that for such (general) semaphores, the **newSem** operation could also be defined so that it is always initialized with $k = 0$ and then (before the semaphore is actually used) calls **signal** k times (this sets the counter to k).

The following shows an invariant that applies to (general) semaphores, which first requires the following definition: For a finite evaluation sequence $\mathcal{P}$ and a semaphore S , $\#_{\text{wait}}(S, \mathcal{P})$ is the number of **wait**(S) operations in $\mathcal{P}$ and $\#_{\text{signal}}(S, \mathcal{P})$ is the number of **signal**(S) operations in $\mathcal{P}$.

Lemma 3.2.2. *Let S be a general semaphore initialized with the number $k \geq 0$. Then, after each finite evaluation sequence $\mathcal{P}$, the following applies:*

- $S.V \geq 0$
- $S.V = k + |S.M| + \#_{\text{signal}}(S, \mathcal{P}) - \#_{\text{wait}}(S, \mathcal{P})$

Proof. The first statement is trivial, since $S.V$ is initialized with $k \geq 0$, **signal** increases the value $S.V$ or leaves it unchanged, and **wait** only decreases the value $S.V$ by 1 if it is greater than 0. The second statement can be shown by induction over the length of $\mathcal{P}$: If $\mathcal{P}$ is the empty sequence, then $S.V = k$ obviously holds, since **newSem** was programmed that way, and furthermore, $S.M = \emptyset$, i.e., $|S.M| = 0$, must hold. For the induction step, let $\mathcal{P}_1$ be a sequence of length $n - 1$. The induction hypothesis is that the statement holds after executing $\mathcal{P}_1$, i.e., $S.V = k + |S.M| + \#_{\text{signal}}(S, \mathcal{P}_1) - \#_{\text{wait}}(S, \mathcal{P}_1)$. It must be shown that the statement holds for $\mathcal{P}_1 e$, where e is another operation.

- If the operation e has nothing to do with the semaphore S , then the statement holds.
- The operation e is a **wait**(S) operation. If $S.V$ previously had the value 0, then $S.V$ is not changed, but $|S.M|$ increases by 1. Since then $\#_{\text{wait}}(S, \mathcal{P}_1 e) = 1 + \#_{\text{wait}}(S, \mathcal{P}_1)$ holds, the statement is true. If $S.V$ was previously greater than 0, then $S.V$ is decreased by one and the equation is again true.
- The operation e is a **signal**(S) operation. Then either $S.V$ is increased by 1, or $|S.M|$ is decreased by one. Since $\#_{\text{signal}}(S, \mathcal{P}_1 e) = 1 + \#_{\text{signal}}(S, \mathcal{P}_1)$ then applies, the statement follows. $\square$

A distinction is made between general semaphores, as described above, and *binary semaphores*. For binary semaphores, the value attribute $S.V$ may only take the values 0 and 1. The **newSem** operation can initialize the semaphore with a value of 0 or 1. The **wait** operation works the same as before, but the **signal** operation must be changed (since for $S.V = 1$ the value cannot be increased further). More precisely, a **signal** operation is not allowed if $S.V = 1$, i.e., the operation leads to an error or an undefined state. The code for the **signal** operation of a binary semaphore is therefore:

```

procedure signal( $S$ )
  if  $S.V = 1$  then
    undefined
  else if  $S.M = \emptyset$  then
     $S.V := 1$ 
  else
    select an element  $Q$  from  $S.M$ ;
     $S.M := S.M \setminus \{Q\}$ ;
     $Q.\text{state} := \text{ready}$ ;

```

Binary semaphores are also often referred to as “mutexes”. Lemma 3.2.2 does not generally apply to binary semaphores (since too many **signal** operations can lead to an undefined state). However, if there is a **wait** operation in the sequence $\mathcal{P}$ before a **signal** operation is performed, then the lemma still applies. In this case, another invariant $S.V \leq 1$ applies.

3.2.1 Mutual Exclusion Using Semaphores

The mutual exclusion problem can be solved very easily using binary semaphores. Figure 3.1 shows the program code for the i th of n processes.

Theorem 3.2.3. *The algorithm from Figure 3.1 guarantees mutual exclusion and is deadlock-free.*

Proof. Let $\#_{CS}(\mathcal{P})$ be the number of processes in the critical section after execution of the finite sequence $\mathcal{P}$.

Initial: S is a binary semaphore, initialized with 1

Program of the i -th process

```

loop forever
(1)  remaining code
(2)  wait( $S$ )
(3)  Critical section
(4)  signal( $S$ )
end loop

```

Figure 3.1: Simple mutual exclusion algorithm with binary semaphore

Since the algorithm always executes **wait** before **signal**, the invariants from Lemma 3.2.2 apply and the undefined state cannot be reached. Obviously, $\#_{CS}(\mathcal{P}) = \#_{wait}(S, \mathcal{P}) - \#_{signal}(S, \mathcal{P}) - |S.M|$ holds: All processes that have executed a **wait** and **signal** have left the critical section again, but of the remaining processes that have executed **wait** but not yet **signal**, the waiting processes in $S.M$ must be subtracted ($S.M$ here refers to the set after execution of $\mathcal{P}$). With the second invariant from Lemma 3.2.2, this results in

$$\begin{aligned} \#_{CS}(\mathcal{P}) + S.V &= \#_{wait}(S, \mathcal{P}) - \#_{signal}(S, \mathcal{P}) - |S.M| \\ &\quad + k + |S.M| + \#_{signal}(S, \mathcal{P}) - \#_{wait}(S, \mathcal{P}) = k \end{aligned}$$

Since the semaphore is initialized with $k = 1$, this shows that $\#_{CS}(\mathcal{P}) = 1 - S.V$, i.e., at most one process can be in the critical section after executing any sequence.

For deadlock freedom, first assume the opposite. Let $\mathcal{P}$ be an infinitely long evaluation sequence such that at least one process wants to enter the critical section, but none enters the critical section. Then there is a finite prefix $\mathcal{P}_1$ of $\mathcal{P}$ such that no process is in the critical section ($\#_{CS}(\mathcal{P}_1) = 0$) and this remains the case for every longer prefix of $\mathcal{P}$. In addition, a process must be blocked by **wait** forever (i.e., $S.V = 0$ applies from $\mathcal{P}_1$). However, this is impossible because the equation $\#_{CS}(\mathcal{P}) = 1 - S.V$ just proven is violated. $\square$

Note that the algorithm does not guarantee starvation freedom for any number of processes. The following sequence illustrates this for 3 processes, where process 3 starves.

$S.V$	$S.M$	Process 1	Process 2	Process 3
1	$\emptyset$	remaining code	remaining code	remaining code
1	$\emptyset$	wait (S)	remaining code	remaining code
0	$\emptyset$	Critical section	wait (S)	remaining code
0	{2}	Critical section	(blocked)	wait (S)
0	{2, 3}	Critical section	(blocked)	(blocked)
0	{2, 3}	signal (S)	(blocked)	(blocked)
0	{3}	Remaining code	Critical section	(blocked)
0	{1, 3}	wait (S)	Critical section	(blocked)
0	{1, 3}	(blocked)	signal (S)	(blocked)
0	{3}	Critical section	remaining code	(blocked)
0	{3}	Critical section	wait (S)	(blocked)
0	{2, 3}	Critical section	(blocked)	(blocked)
...				

However, it can be proven that the algorithm is starvation-free with a maximum of 2 concurrent processes.

3.2.2 Further Variants of Semaphores

The semaphores defined so far are generally referred to as *weak semaphores*, since the selection of the process to be unblocked in a **signal** operation is arbitrary. In contrast, there are so-called strong semaphores, which process the waiting (blocked) processes in first-in-first-out order. For their implementation, a queue (or simply a list) $S.L$ is used instead of the set $S.M$ of processes. When the semaphore is initialized, the empty list is created. The implementation of the operations is then changed to:

<pre> procedure wait(S) if $S.V > 0$ then $S.V := S.V - 1$; else $S.L := \text{append}(S.L, P)$; $P.\text{state} := \text{blocked}$; </pre>	<pre> procedure signal(S) if $\text{isEmpty}(S.L)$ then $S.V := S.V + 1$; else $Q := \text{head}(S.L)$; $S.L := \text{tail}(S.L)$; $Q.\text{state} := \text{ready}$; </pre>
---	---

Here, *append* adds an element to the end of a list, *head* returns the first element of a list, *tail* returns the rest of the list without the first element, and *isEmpty* checks whether a list is empty.

If strong semaphores are used in the mutual exclusion algorithm from Figure 3.1, the algorithm guarantees starvation freedom and the FIFO property.

Another class of semaphores are so-called *unfair semaphores*. These are even weaker than weak semaphores, since no assumption is made at all about when a blocked process will be unblocked. More precisely, these semaphores can be implemented using busy-wait and do not have the $S.M$ component. The operations can be implemented with a fetch-and-add object, for example:

procedure wait(S)**await** $S.V > 0$; $S.V := S.V - 1$;**procedure signal(S)** $S.V := S.V + 1$;

If such an unfair semaphore is used in the previous mutual exclusion algorithm, it is no longer starvation-free even for two processes.

3.3 Application Examples for Semaphores

This section considers some classic problems of concurrent programming and discusses solutions (i.e., implementations) to these problems with the help of semaphores.

3.3.1 Coordination of Parallel Merge Sort

When writing concurrent or parallel programs, it is usually not possible to execute all steps in parallel. Often, individual steps must be executed sequentially without interleaving. This problem has already been solved with mutual exclusion algorithms. However, some problems require that a single process may only start calculating once other processes have completed their calculations. Consider Mergesort as an example. Recall that Mergesort is a recursive algorithm that proceeds as follows:

- While more than one element remains to be sorted:
 - Divide the input into two equal halves.
 - Sort each half recursively.
 - Merge the two sorted halves into a single sorted sequence.

To parallelize this algorithm, the two recursive calls can naturally be executed concurrently. A similar parallelization strategy applies to almost all divide-and-conquer algorithms. However, it is also obvious that the step of merging the sorted subsequences can only be performed once the (recursive) sorting of both halves is complete. The merge process must therefore wait until the two sorting processes finish. Two binary semaphores suffice for this coordination. Figure 3.2 shows the implementation for the merge process and for the two recursive sorting processes. The details of sorting and merging are omitted; only the semaphore-based coordination is shown.

The implementation is easy to describe: Since the two binary semaphores are initially set to 0, the merge process is blocked at both semaphores. As soon as both sorting processes have performed their **signal** operation for the two halves, the merge process is ready again and can mix.

It should also be noted that this implementation of merge sort requires two semaphores per recursion step, i.e., for n elements to be sorted, $O(\log n)$ binary semaphores are required.

Initial: left, right: Binary semaphores initialized with 0

Process for left half:

- (1) sort left half;
- (2) **signal**(left);

Process for right half:

- (1) sort right half;
- (2) **signal**(right);

merge process:

- (1) **wait**(left);
- (2) wait(right);
- (3) merge the two halves;

Figure 3.2: Coordination of merge sort using semaphores

3.3.2 Producer-Consumer Problems

In the producer-consumer problem, processes are divided into two groups: producers that generate data, and consumers that process it. Classic examples include the keyboard (producer) and the operating system (consumer), or a web browser and a web server, where both roles may be exchanged. To decouple producers and consumers, a shared buffer (a queue) is typically used. If the producer generates data faster than the consumer can process it, the buffer should be bounded to avoid overflow. Some applications also require a bounded buffer because data can only be processed in complete blocks—for instance, a decompression program can only start once the full zip file has been downloaded.

The challenge is protecting concurrent access to the buffer. We distinguish between infinite and bounded buffers.

First, let us consider the producer/consumer problem with an infinite buffer. In this case, it must be ensured that the buffer is accessed atomically for reading and writing. Furthermore, it must be implemented that the consumer process waits when reading an empty buffer until the buffer is filled with data again. Figure 3.3 shows an implementation of the producer/consumer problem with arbitrarily large buffer memory. It uses two semaphores: The binary semaphore `mutex` ensures that reading and writing are always performed atomically. The semaphore `notEmpty` is a general semaphore; it is used to store the size of the buffer memory and to block the consumer in the event of an empty buffer.

`mutex` ensures that only one process can access the list at a time. The consumer must first check the semaphore `notEmpty` to see if there are any elements in the buffer. If this is the case, the **wait** operation decreases the value `notEmpty.V` by 1 and the consumer may take the first element of the list. If the value of `notEmpty.V` is 0, the consumer is blocked: it must wait until a producer process performs a **signal** operation for the semaphore `notEmpty`. The producer appends the element it has created to the end of the list (this is always possible, since the buffer can be as large as desired) and then signals that the list is not empty.

Initial: notEmpty: General semaphore, initialized with 0
 mutex: Binary semaphore, initialized with 1
 l: List

Producer (produces e):

- (1) produce e;
- (2) **wait**(mutex);
- (3) l := append(l,e);
- (4) **signal**(notEmpty);
- (5) **signal**(mutex);

Consumer (consumes e):

- (1) **wait**(notEmpty);
- (2) **wait**(mutex);
- (3) e := head(l);
- (4) l := tail(l);
- (5) **signal**(mutex);
- (6) consume e;

Figure 3.3: Semaphore implementation of producer/consumer with infinite buffer

Assuming that the list is empty at the beginning, it is easy to see that the equation $\text{notEmpty}.V = \text{length}(l)$ (where length calculates the length of the list) is always true, i.e., it is an invariant of the buffer memory implementation.

Now let us consider a buffer of limited size. In this case, in addition to the previous requirements (atomic reading and writing to the buffer, and protection when a consumer accesses the empty buffer), it must also be ensured that a producer cannot write to a full buffer and must instead wait until the buffer is no longer full. Figure 3.4 shows an implementation for this case. An additional general semaphore notFull is used, which is initially created with the maximum size of the buffer N .

Initial: notEmpty: General semaphore, initialized with 0
 notFull: General semaphore, initialized with N
 mutex: Binary semaphore, initialized with 1
 l: List

Producer (produces e):

- (1) produce e;
- (2) **wait**(notFull);
- (3) **wait**(mutex);
- (4) l := append(l,e);
- (5) **signal**(notEmpty);
- (6) **signal**(mutex);

Consumer (consumes e):

- (1) **wait**(notEmpty);
- (2) **wait**(mutex);
- (3) e := head(l);
- (4) l := tail(l);
- (5) **signal**(notFull);
- (6) **signal**(mutex);
- (7) consume e;

Figure 3.4: Semaphore implementation of producer/consumer with bounded buffer

Compared to the implementation of the infinite buffer, only minor changes have been made: The additional semaphore `notFull` monitors the call of the producer. If `notFull.V` has the value 0, the buffer is completely full and the producer is blocked. The consumer signals that it has removed an element from the buffer by executing the **signal**(`notFull`) operation.

As an invariant of the two semaphores, it can be easily shown that $\text{notEmpty.V} + \text{notFull.V} = N$ always applies. Such a data structure consisting of two semaphores, which acts “in both directions” like a semaphore, is generally referred to as a “split semaphore”.

3.3.3 The Dining Philosophers

The dining philosophers problem is probably the best-known classic problem in concurrent programming. The formulation using philosophers was devised by Tony Hoare.

In this problem, several philosophers are seated around a round table. Each philosopher has a plate, and in the middle of the table is a large bowl of spaghetti. Each philosopher symbolizes a process. A philosopher alternates between thinking and eating. He does this indefinitely. The problem arises from the fact that a single philosopher needs two forks to eat (one in the left hand, one in the right hand), but there is always only one fork on the table between two philosophers. Figure 3.5 illustrates the situation for five philosophers.

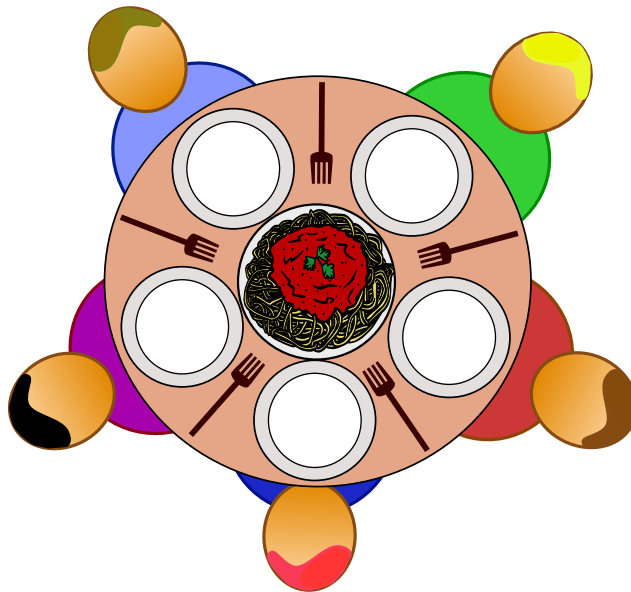


Figure 3.5: Five dining philosophers

An important assumption in the dining philosophers problem is that a philosopher

can only pick up one fork at a time, or conversely, he cannot pick up both forks at the same time.

Solutions to the problem should guarantee deadlock freedom, i.e., at least some philosopher can switch between thinking and eating infinitely often. As a second requirement, a kind of starvation freedom should be guaranteed, whereby starvation can be taken literally here: No philosopher starves, i.e., after a finite number of steps, every philosopher eats.

To model the problem, the N philosophers are numbered from 1 to N . The forks are modeled by binary semaphores (initialized with 1 at the beginning). For the philosopher with number i , let $\text{fork}[i]$ be his left fork and $\text{fork}[i + 1]$ his right fork (the addition is implicitly modulo N).

The solution to the philosopher problem now consists of specifying a program for philosopher i . A first attempt is shown in Figure 3.6. Each philosopher first attempts to

forks: Field (dimension $1 \dots N$) of binary semaphores, initially all initialized with 1

Philosopher i :

loop forever

(1) Philosopher thinks;

(2) **wait**(fork[i]); // left fork

(3) **wait**(fork[$i+1$]); // right fork

(4) Philosopher eats

(5) **signal**(fork[$i + 1$]);

(6) **signal**(fork[i]);

end loop

Figure 3.6: Attempt to solve the philosopher's problem

take his left fork and then his right fork (using the **wait** operation). After the philosopher has eaten, he puts down both forks one after the other (using the **signal** operation). However, this approach does *not* work, as the following example sequence for three philosophers shows:

Philosopher 1	Philosopher 2	Philosopher 3
wait (forks[1]) "has left fork"	wait (forks[2]) "has left fork"	wait (forks[3]) "has left fork"
wait (forks[2]) blocked	wait (forks[3]) blocked	wait (forks[1]) blocked

All philosophers manage to pick up their left fork one after the other. Afterwards, there

are no more forks on the table. Since all philosophers now try to pick up their right fork, they are blocked one after the other. A deadlock has occurred because no philosopher can be unblocked anymore.

One way to prevent such deadlocks is to use an additional binary semaphore mutex, which regulates access to the forks exclusively. Figure 3.7 shows the corresponding code for the i th philosopher. Before a philosopher is allowed to touch the forks, he must pass through the mutex semaphore. He then takes the forks, eats, and puts the forks down. After that, he releases the mutex semaphore.

forks: Field (dimension $1 \dots N$) of binary semaphores, initially all initialized with 1
mutex: Binary semaphore, initialized with 1

Philosopher i :
loop forever
 (1) Philosopher thinks;
 (2) **wait**(mutex);
 (3) **wait**(fork[i]); // left fork
 (4) **wait**(fork[$i+1$]); // right fork
 (5) Philosopher eats
 (6) **signal**(fork[$i+1$]);
 (7) **signal**(fork[i]);
 (8) **signal**(mutex);
end loop

Figure 3.7: 2nd attempt to solve the philosopher problem

It is easy to see that this solution is deadlock-free, since only one philosopher has access to *all* forks. However, the algorithm is only starvation-free if mutex is a strong semaphore that satisfies the FIFO property. If mutex is a weak semaphore, a philosopher can starve because when blocked, he can always be overtaken by other philosophers. Another problem with this proposed solution is that at most one philosopher can eat at a time. Conversely, this also means that $N - 2$ forks are always on the table. If we apply this to real-world problems, we see a waste of resources: just to ensure correct implementation, $N - 2$ resources are “blocked” even though they are not being used. For this reason, this solution is generally unacceptable. However, if we consider the case where only two philosophers are sitting at the table, we can conclude that the resources are being used well and that even when using a weak semaphore for mutex, starvation-free conditions apply.

The next solution extends the good properties of the 2-process solution to N processes: maximize fork utilization while preserving starvation-freedom even with weak semaphores. The code is shown in Figure 3.8. The idea is to use a guard that allows at most $N - 1$ philosophers into the room simultaneously. Even if all $N - 1$ admitted philosophers pick up their left fork first, at least one of them can always pick up the

right fork (the left fork of the excluded philosopher is free). This rules out deadlock. The guard is implemented by the general semaphore `room`, initialized with $N - 1$, so the N th philosopher is blocked outside.

```

        Initialize all forks with 1
room: general semaphore, initialized with  $N - 1$ 

Philosopher  $i$ 
  loop forever
    (1) Philosopher thinks;
    (2) wait(room);
    (3) wait(fork[ $i$ ]); // left fork
    (4) wait(fork[ $i+1$ ]); // right fork
    (5) Philosopher eats
    (6) signal(fork[ $i+1$ ]);
    (7) signal(fork[ $i$ ]);
    (8) signal(room);
  end loop

```

Figure 3.8: Solution to the philosopher's problem with guards

Even with a weak semaphore for `room`, this solution is starvation-free: at most one process can be blocked outside the room at any time, and it will be unblocked at the next `signal(room)`. Inside the room, at most two processes compete for any given fork (one holds it as a left fork, the other wants it as a right fork). A blocked process will obtain the fork after a finite number of steps, since binary weak semaphores guarantee starvation-freedom for two processes. Global deadlock is impossible—this is precisely what the room semaphore prevents.

The following section shows another way to make global deadlock impossible. To do this, we must note that the deadlock (all have the left fork) occurs because the resources are cyclically dependent on each other. The solution in Figure 3.9 breaks this dependency by having the N th process proceed in reverse to all other processes: it first attempts to pick up its *right* fork and then, in the next step, only the *left* fork. It can be shown that neither deadlocks nor starvation are possible for this algorithm either. The absence of deadlocks is a direct consequence of the so-called theorem of total order, which will be discussed later (but only in the following chapter).

3.3.4 The Sleeping Barber Problem

The Sleeping Barber Problem consists of the situation depicted in Figure 3.10 of a barber shop.

The shop has $N - 1$ waiting places, one barber chair, and one barber. The barber and the customers are to be modeled as processes, whereby the following conditions must

Initially, all forks are initialized with 1

Philosopher $i, i < N$:

loop forever

- (1) Philosopher thinks;
 - (2) **wait**(fork[i]); // left fork
 - (3) **wait**(fork[$i+1$]); // right fork
 - (4) Philosopher eats
 - (5) **signal**(fork[$i+1$]);
 - (6) **signal**(fork[i]);
- end loop**

Philosopher N

loop forever

- (1) Philosopher thinks;
 - (2) **wait**(fork[$i+1$]); // right fork
 - (3) **wait**(fork[i]); // left fork
 - (4) Philosopher eats
 - (5) **signal**(fork[i]);
 - (6) **signal**(fork[$i+1$]);
- end loop**

Figure 3.9: Solution to the philosopher's problem with asymmetric N -process

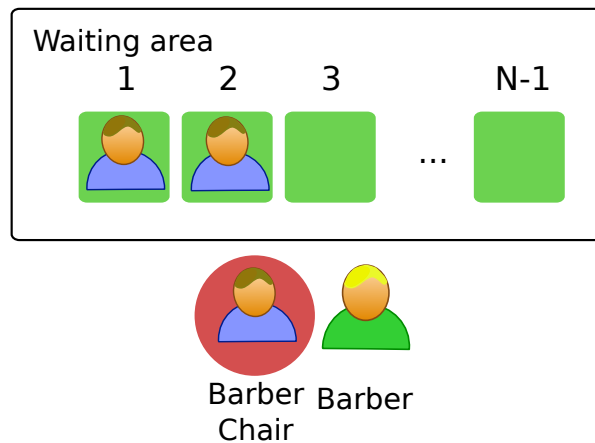


Figure 3.10: Situation of the Sleeping Barber Problem

apply: If there are no customers in the shop, the barber sleeps. The first customer in the shop wakes the barber and has his hair cut. If there are several customers in the shop, a new customer takes a seat in one of the waiting areas. If all waiting areas are occupied, the customer leaves the shop immediately. After the barber has given a customer a new hairstyle, the barber waits until the customer has left the shop, then the barber takes the next waiting customer.

A semaphore-based solution is shown in Figure 3.11. The atomic register `waiting` counts the number of waiting customers (initially 0). The general semaphore `customers` also tracks the number of customers and is used to wake up the barber. The binary semaphore `mutex` protects updates to `waiting` (making them atomic). The semaphore `synch` makes the barber wait until the current customer has left before serving the next one. Finally, the semaphore `barber` signals to a waiting customer that the barber chair is free.

```

Initial:  waiting: atomic register, initially 0
          customers: general semaphore, initially 0
          mutex: binary semaphore, initially 1
          synch,hairstresser: binary semaphore, initially 0

Hairstresser:
loop forever
(1) wait(customers); //sleep as long as there are no customers
(2) wait(mutex);
(3) waiting := waiting - 1;
(4) signal(hairstresser); //take next customer
(5) signal(mutex);
(6) cut hair;
(7) wait(synch); //wait until customer leaves shop
end loop

Customer:
(1) wait(mutex);
(2) if waiting < N then
(3)   waiting := waiting + 1;
(4)   signal(customers); //Wake up hairstresser (or increase number of customers)
(5)   signal(mutex);
(6)   wait(hairstresser); //Wait until hairstresser is ready
(7)   get haircut;
(8)   signal(synch); //leave shop
(9) else signal(mutex); //leave immediately

```

Figure 3.11: Solution to the Sleeping Barber Problem with Semaphore

The customer's program can be explained as follows: In line (1), the semaphore mu-

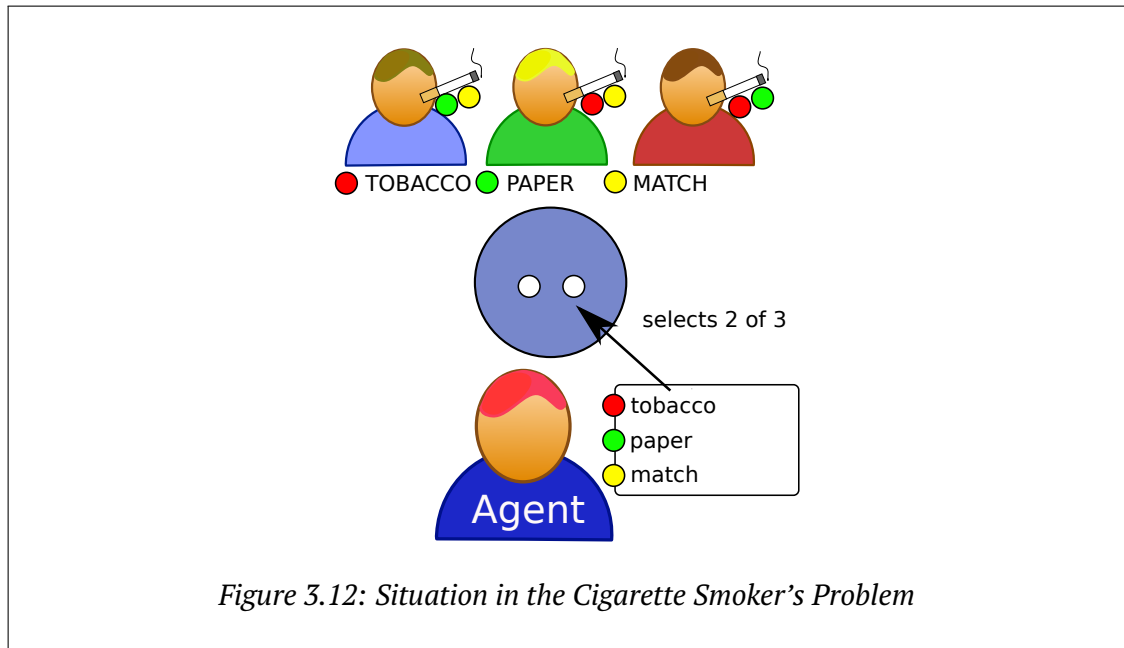
tex is occupied in order to obtain secure access to the variable waiting (this resource is then released either in line (5) or in line (9)). In line (2), the customer checks whether the store is too full. If this is the case, it jumps to line (9) and leaves the store (and the algorithm) immediately. Otherwise, the number of waiting customers is first increased and the general semaphore customers is also increased using the **signal** operation. If the current customer is the first customer in the shop, the hairdresser will wait at the semaphore customers and will be awakened (unblocked) by the **signal** operation of the customer. In line (5), the mutex is released again (so that other customers can enter the shop). In line (6), customer a waits, semaphore hairdresser, until the hairdresser serves him (the hairdresser will inform him of this with a **signal** operation). After the customer has received his haircut in line (7), he leaves the shop in line (8), again signaling this to the hairdresser with a **signal** operation for the semaphore synch.

From the hairdresser's point of view: Unlike the customer, the hairdresser runs in an endless loop (he styles hair all the time, while a customer has to be content with one hairstyle). First, the hairdresser waits at the semaphore kunden. If there are no customers, the hairdresser is blocked, i.e., he sleeps. After the hairdresser has been woken up, he uses the semaphore mutex in lines (2) to (5) to atomically change the variable waiting: In line (3), he decreases the number of waiting customers, since he takes the next customer in line (4) he takes the next customer (he signals this via the semaphore hairdresser). In line (6) he creates the customer's new hairstyle, and finally in line (7) he waits (at the semaphore synch) for the customer he has just served to leave the shop.

3.3.5 The Cigarette Smoker's Problem

In the Cigarette Smoker's Problem, the following problem must be correctly modeled as a concurrent program: There are three smokers and one agent. Three ingredients are needed to smoke a cigarette: tobacco, paper, and a match. However, each of the three smokers has only one of the ingredients (of which he has an infinite supply). The agent has all three ingredients. The agent always selects two of the ingredients at random and places them on the table. The smokers now try to smoke a cigarette using the ingredients on the table. To do this, they must take the ingredients from the table one after the other. The agent only places ingredients back on the table once the previous ones have been used up. The graphic in Figure 3.12 symbolically represents the situation.

The agent's program is given by the problem and is shown in Figure 3.13, where four semaphores are used: $S[1]$, $S[2]$, $S[3]$ are binary semaphores for tobacco, paper, and matches. Initially, the semaphores are initialized with 0. If one of these semaphores is set to 1, this means that the corresponding ingredient is on the table. The agent uses another semaphore, agent, which indicates whether the agent should place new ingredients on the table again (i.e., the agent waits at this semaphore using **wait**). The semaphore agent is initially set to 1, because the agent should immediately place ingredients on the table at the beginning. The agent's program consists of first selecting two numbers from the set $\{1, 2, 3\}$ in line (1). These two numbers later indicate which two of the three ingredients the agent will place on the table. However, the agent first



waits in line (2) until it receives the signal to place ingredients on the table. In lines (3) and (4), it places the ingredients on the table by executing a **signal** operation for the two corresponding semaphores.

A solution to the Cigarette Smoker's Problem now consists of specifying the code for the smokers, whereby no (global) deadlock should occur: The appropriate smoker takes the ingredients and smokes.

```

Initial:  wait: binary semaphore set to 1
          S[1], S[2], and S[3]: binary semaphores for tobacco, paper
                                and matches occupied with 0

loop forever
  (1)  select i and j randomly from {1, 2, 3};
  (2)  wait(agent);
  (3)  signal(S[i]);
  (4)  signal(S[j]);
end loop
  
```

Figure 3.13: Agent program

The difficulty of the problem is obvious: if a smoker takes a suitable ingredient, but the other ingredient on the table does not fit, a deadlock is more or less inevitable. Figure 3.14 shows a code for smokers that takes a naive approach, but is unfortunately

wrong.

<u>Smoker with tobacco:</u> loop forever (1) wait (S[2]); (2) wait (S[3]); (3) “smoke”; (4) signal (agent); end loop	<u>Smoker with paper:</u> loop forever (1) wait (S[1]); (2) wait (S[3]); (3) “smoke”; (4) signal (agent); end loop	<u>Smoker with match:</u> loop forever (1) wait (S[1]); (2) wait (S[2]); (3) “smoke”; (4) signal (agent); end loop
---	---	---

Figure 3.14: Naive attempt at the Cigarette Smoker’s Problem

Consider the following execution (underlined commands are executed next):

Agent	Smoker m. Tobacco	Smoker m. Paper	Smoker m. Match	S[1]	S[2]	S[3]	agent
<u>chooses</u> <u>i = 1, j = 2</u>	wait (S[2])	wait (S[1])	wait (S[1])	0	0	0	1
<u>wait(agent)</u>	wait (S[2])	wait (S[1])	wait (S[1])	0	0	0	1
<u>signal(S[1])</u>	wait (S[2])	wait (S[1])	wait (S[1])	0	0	0	0
<u>signal(S[2])</u>	wait (S[2])	wait (S[1])	wait (S[1])	1	0	0	0
<u>wait(agent)</u>	<u>wait(S[2])</u>	wait (S[1])	wait (S[1])	1	1	0	0
<u>wait(agent)</u>	wait (S[3])	<u>wait(S[1])</u>	wait (S[1])	1	0	0	0
<u>wait(agent)</u>	wait (S[3])	wait (S[3])	wait (S[1])	0	0	0	0
blocked	<u>wait(S[3])</u>	wait (S[3])	wait (S[1])	0	0	0	0
blocked	blocked	<u>wait(S[3])</u>	wait (S[1])	0	0	0	0
blocked	blocked	blocked	<u>wait(S[1])</u>	0	0	0	0
blocked	blocked	blocked	blocked	0	0	0	0

The execution shows that a deadlock is possible: all processes are blocked. In fact, solutions to the Cigarette Smoker’s Problem are not easy. One possibility is to add three more processes to the existing four processes, which are used as helpers. Six additional binary semaphores $R[1], \dots, R[6]$ (initialized with 0) are added to the existing semaphores, as well as an atomic register t (initially set to 0) and a binary semaphore mutex (initially 1), which is used to protect the register t . Figure 3.15 shows the new implementation: In addition to the programs for the three smokers, it also includes three programs for the three helpers.

The solution can be explained as follows: The smokers no longer access the table directly themselves. They are unblocked by the helpers. There is one helper for each of

Additional objects: $R[i]$, $i = 1, \dots, 6$, binary semaphores (initially 0),
 mutex: binary semaphore (initial 1),
 t: atomic register (initial 0)

Helpers (tobacco)

```
loop forever
(1) wait(S[1]);
(2) wait(mutex);
(3)  $t := t + 1$ ;
(4) if  $t \neq 1$  then
(5)   signal(R[t]);
(4) signal(mutex);
end loop
```

Helper (paper)

```
loop forever
(1) wait(S[2]);
(2) wait(mutex);
(3)  $t := t + 2$ ;
(4) if  $t \neq 2$  then
(5)   signal(R[t]);
(4) signal(mutex);
end loop
```

Helper (matchstick)

```
loop forever
(1) wait(S[3]);
(2) wait(mutex);
(3)  $t := t + 4$ ;
(4) if  $t \neq 4$  then
(5)   signal(R[t]);
(4) signal(mutex);
end loop
```

Smoker with tobacco

```
loop forever
(1) wait(R[6]);
(2)  $t := 0$ ;
(3) "smoke";
(4) signal(agent);
end loop
```

Smoker with paper

```
loop forever
(1) wait(R[5]);
(2)  $t := 0$ ;
(3) "smoke";
(4) signal(agent);
end loop
```

Smoker with match

```
loop forever
(1) wait(R[3]);
(2)  $t := 0$ ;
(3) "smoke";
(4) signal(agent);
end loop
```

Figure 3.15: Solution to the Cigarette Smoker's Problem with Helpers

the ingredients. Each helper waits for its corresponding resource. As soon as it holds the ingredient in the form of the semaphore, it modifies the value of the atomic register t , doing so atomically: To do this, it occupies the semaphore mutex, thus ensuring that only it can change the value of t . There is a little number game involved here: The helper for the tobacco increases t by 1, the helper for the paper increases t by 2, and the helper for the match increases t by 4. The initial value of t is 0. The helper then checks in line (4) whether it is the first to increase the value of t first. If this is the case, the helper releases the mutex and does nothing else. If the helper is the second to increase t , then the other ingredient is already off the table. In this case (line (5)), he wakes up the corresponding smoker via the semaphore $R[t]$. The trick is that the helper who receives the second ingredient (and thus increases t second) is responsible for waking up the corresponding smoker. This is where the number game comes in: With a table, you can calculate that the "right" thing happens:

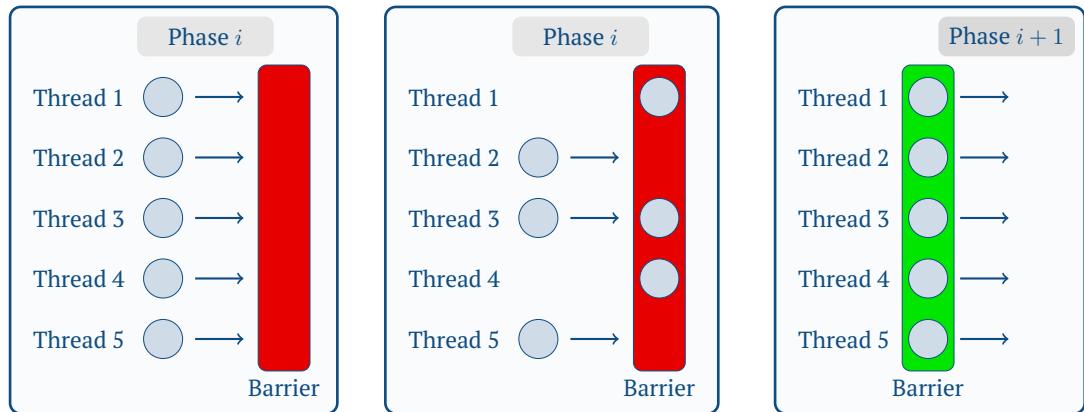
Ingredients on the table	first helper	second (=awakening) helper	Value of t	awakened smoker
Tobacco & Paper	(Tobacco)	(Paper)	$1+2 = 3$	$R[3]$ (=matches)
Tobacco & Paper	(Paper)	(Tobacco)	$2+1 = 3$	$R[3]$ (=matches)
Tobacco & matches	(tobacco)	(matches)	$1+4 = 5$	$R[5]$ (=paper)
Tobacco & matches	(matches)	(tobacco)	$4+1 = 5$	$R[5]$ (=paper)
Paper & matches	(paper)	(matches)	$2+4 = 6$	$R[6]$ (=tobacco)
Paper & matches	(matches)	(paper)	$4+2 = 6$	$R[6]$ (=tobacco)

The algorithm for smokers is simple: a smoker waits until he is woken up by the helper. He then resets the value of t for the next round and smokes. When he has finished smoking, he signals to the agent that the next round can begin.

3.3.6 Barriers

As a further example, we consider the construction of so-called "barriers". These were already used in the implementation of the parallelized merge sort. Barriers serve to synchronize a set of processes at a certain point, i.e., they wait until all (or a certain set of) processes have crossed the barrier before continuing. In merge sort, the merge process waited until the two sorting processes for the halves had finished sorting. The two semaphores left and right served as barriers. In what follows, it is assumed that all processes wait for each other (this situation did not exist in merge sort, since, for example, the sorter of the left half did not wait for the sorter of the right half).

The problem of barrier coordination can also be illustrated visually.



The processes run until a certain point in their calculation. Once they have reached this point, they must wait until all processes have reached the barrier. Only then may the processes continue their calculation.

For two processes, the barrier problem can be solved relatively easily using two binary semaphores. The code for both processes is shown in Figure 3.16. As soon as a process has finished its calculation before the barrier, it signals this on “its” semaphore. It then waits at the “other” semaphore for the other process to finish and signal this.

Initial: `p1ready`, `p2ready`: binary semaphores are 0 at the start

Program for process 1:

- (1) Calculation before the barrier;
- (2) **signal**(`p1ready`);
- (3) **wait**(`p2ready`);
- (4) Calculation after the barrier;

Program for process 2:

- (1) Calculation before the barrier;
- (2) **signal**(`p2ready`);
- (3) **wait**(`p1ready`);
- (4) Calculation after the barrier;

Figure 3.16: Barrier with semaphore for two processes

For n processes, two binary semaphores and one atomic register (for counting the processes) are sufficient. Figure 3.17 shows the implementation.

As soon as processes arrive at the barrier, they increase the counter of the waiting processes and (with the exception of the n th process) then wait at the semaphore leave in line (7). For the n th process, the **if** condition in line (4) becomes false. Therefore, it releases the semaphore (a process is unblocked). When leaving the barrier (from line

Initial: arrive: binary semaphore initialized with 1
 leave: binary semaphore initialized with n
 counter: atomic register initialized with 0

Program for process i:

```
(1) Calculation before the barrier;
(2) wait(arrive);
(3) counter := counter + 1;
(4) if counter < n then
(5)   signal(arrive)
(6) else signal(leave)
(7) wait(leave);
(8) counter := counter - 1;
(9) if counter > 0 then
(10)  signal(leave)
(11) else signal(arrive)
(12) Calculation after the barrier;
```

Figure 3.17: Barrier with semaphore for n processes

(8) onwards), the processes decrease the counter one after the other. The last process to leave the barrier executes the **signal**(arrive) operation in line (11). This returns the barrier to its original state and the next “phase” can begin. It is important to note that between the entry of the n th process and the exit of the last process, no other processes can re-enter the barrier. However, this is ensured by the fact that the n th process performs a **wait**(arrive) operation at the beginning (in line (2)) (all subsequent processes are blocked here) and the arrive semaphore is only released by the very last process that leaves the barrier (in line (11)).

Barriers are also used below as an abstract data type (e.g., as a 4-tuple $(n, \text{arrive}, \text{leave}, \text{counter})$) supporting the following operations:

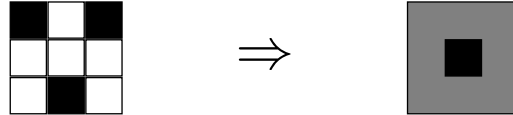
- **newBarrier**(k): Creates a barrier for k processes, i.e., two semaphores are created for arrive and leave, an atomic register counter is created and initialized accordingly, and a 4-tuple $(n, \text{arrive}, \text{leave}, \text{counter})$ is returned.
- **synchBarrier**(B): Synchronizes at the barrier, i.e., lines (2) to (12) from Figure 3.17 are executed. for the 4-tuple B .

3.3.6.1 An example application: Conway’s Game of Life

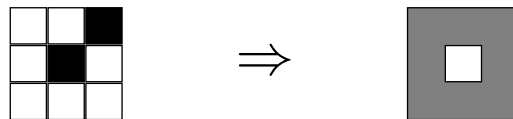
The playing field of Conway’s Game of Life consists of a two-dimensional $N \times N$ matrix, where fields are either *inhabited* or *uninhabited*. In the Game of Life, this matrix is

changed from generation to generation. For a field, the value in the next generation is determined depending on the number of neighbors:

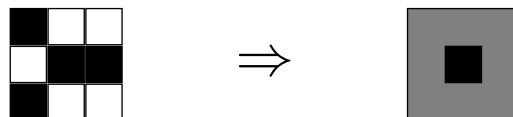
- If the field is uninhabited, it will only be inhabited in the next generation if it had exactly three neighbors. For example:



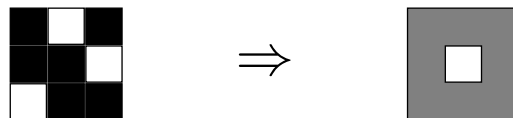
- If the field is inhabited, the following applies to the next generation:
 - If it had no neighbors or only one neighbor, it will subsequently be uninhabited due to underpopulation. For example:



- If the field had two or three neighbors, it remains inhabited. For example:



- If the field had four or more neighbors, it is then uninhabited due to “overpopulation”.



Let the playing field be represented by a two-dimensional array with Boolean entries. Furthermore, let function `nextValue(i, j, array)` already be implemented, which calculates for the field in row *i* and column *j* whether it will be uninhabited or inhabited in the next generation: The function reads all values of the neighboring fields of field (*i*, *j*) as well as the value of field (*i*, *j*) one after the other and then calculates **True** or **False** according to the above rules.

A sequential algorithm that calculates the *k*th subsequent generation is as follows:

array: Initialized $N \times N$ array representing the playing field

Algorithm:

```

for g:=1 to k do
  for i=1 to N do
    for j=1 to N do
      array2[i,j] := nextValue(i,j,array);
  for i=1 to N do
    for j=1 to N do
      array[i,j] := array2[i,j];
  
```

Assuming you now want to parallelize this algorithm (or program it concurrently) so that one process is available per field to calculate the update of a field, a naive approach would be the following algorithm:

```
array: Initialized  $N \times N$  array representing the playing field  
( $N \times N$ ) processes: one per playing field  
  
Program for process (i,j):  
for g:=1 to k  
  v := nextValue(i,j,array);  
  array[i,j] := v;
```

Unfortunately, however, the algorithm is incorrect, because when process (i, j) calculates the value of the l th generation, there is no guarantee that it will actually only read the values of the $l - 1$ th generation for the neighbors in the array: Other processes may be slower or faster and update the field at their positions too early or too late. The processes must synchronize. A barrier is suitable for this purpose: Then the processes can be programmed as follows:

```
array: Initialized  $N \times N$  array representing the playing field  
barrier: Barrier for  $N \times N$  processes  
( $N \times N$ ) processes: one per playing field  
  
Program for process (i,j):  
for g:=1 to k  
  v := nextValue(i,j,array);  
  synchBarrier(barrier);  
  array[i,j] := v;  
  synchBarrier(barrier);
```

Now the calculation works: Before the processes are allowed to write the new value to the field, they must all wait for each other at the barrier (first **synchBarrier** call) and then, after writing, they wait again for each other again (second **synchBarrier** call) to ensure that everyone has actually written to the field before calculating the next generation.

3.3.7 The Readers & Writers Problem

Like the barrier problem, the readers & writers problem is more of a general concept. Here, the processes are divided into two groups. One group is the readers (i.e., reading processes), while the other group is the writers (i.e., writing processes). Such a division can be applied, for example, to database systems or database access. There are processes that only access the data in read mode, while other processes want to change

the entries in the database. The key point here is that multiple processes can access the database simultaneously as long as they are only reading the data. The writing processes, on the other hand, require exclusive access to ensure data consistency. The goal of the readers & writers problem is therefore to enable parallel read access as much as possible and to allow write access only sequentially.

Two different strategies can be pursued for this problem, depending on which of the two groups of processes should have higher priority when accessing the data. If readers are given higher priority than writers, the writing processes must wait (they are blocked) as long as there are still reading processes present or arriving. If the priority is reversed, writing processes are given preference and take precedence over reading processes.

Depending on the application, one priority selection may be more advantageous than another. Taking a flight reservation system as an example, from the operator's point of view, it is probably more advantageous to give priority to writers over processes that only search for possible flights but do not (yet) want to book flights: On the one hand, the operator earns more money with booked flights, and on the other hand, the reading requests want to be informed as quickly as possible about current flights (i.e., flights that are not fully booked).

The following section explains solutions with semaphores for both priority variants.

Figure 3.18 contains the code for reader and writer processes, whereby reader processes have higher priority than writer processes.

The implementation uses four data structures: An atomic register `countR`, which stores the number of current reader processes. Consequently, `countR` has an initial value of 0. The semaphore `mutexR` is used to protect access to `countR` (and to the semaphore `w`), with an initial value of 1. Furthermore, the semaphore `w` is used to prevent access by writing processes if reading processes are present or another writing process is already active. The semaphore `w` is initialized with 1. The semaphore `mutex` ensures that at most one writer can wait at `w`.

The program for the writing processes is simple: In line (2), the process checks whether the semaphore `w` is occupied. If this is the case, the process is blocked and waits for release. Otherwise, the writing process is allowed to enter the critical section (line (4)). After leaving the critical section, the writing process signals this by means of a **signal** operation on the semaphore `w`. As will be shown later, the semaphore `w` can be occupied either by a reading process or by writing processes. It is already obvious that at most one writing process can enter the critical section at any given time. Occupying the semaphore `mutex` in line (1) and releasing it in line (3) ensures that at most one writing process waits at the semaphore `w`. This gives readers preference over writers.

The code of a reading process can be commented as follows: First, it occupies the semaphore `mutexR` in order to gain exclusive access to the register `countR`. In line (2), the stored number of reading processes is increased by one. In lines (3) and (4), the first reading process is assigned a special role (it had previously increased `countR` from 0 to 1): This first process occupies the semaphore `w`. It then either waits for all writing processes to leave the critical section and then (or immediately) ensures by occupying `w` that no further writing processes can enter the critical section, but are blocked at `w`.

Initial: countR: atomic register, initially 0
mutex, mutexR, w: binary semaphore, initially 1

Program for reader:

```
(1)  wait(mutexR);
(2)  countR := countR + 1;
(3)  if countR = 1 then
(4)    wait(w);
(5)  signal(mutexR);
(6)  Critical section
(7)  wait(mutexR);
(8)  countR := countR - 1;
(9)  if countR = 0 then
(10)    signal(w);
(11) signal(mutexR);
```

Program for Writer:

```
(1)  wait(mutex);
(2)  wait(w);
(3)  signal(mutex);
(4)  Critical section;
(5)  signal(w)
```

Figure 3.18: Readers & Writers with Priority for Readers

At this point, other reading processes cannot interfere: When it is at line (4), the first reading process still occupies the semaphore `mutexR`, which means that other reading processes cannot get past line (1).

In line (5), the reading process releases the semaphore `mutexR`. This allows other readers to enter the critical section. After leaving the critical section, a reader process uses the semaphore `mutexR` again to change the value of `countR` exclusively. In line (8), it decreases the value by 1, while lines (9) and (10) again play a special role: they are only relevant for the last reader. This ensures that any waiting writer process is unblocked (using the **signal**(w) operation in line (10)).

This algorithm gives the reader processes priority over the writer processes, because as long as reading processes are in the critical section, other reading processes can also enter the critical section. Waiting writer processes must wait until the last reader unblocks the semaphore `w`.

Figure 3.19 shows another implementation for the readers & writers problem, which gives writer processes priority over reader processes.

In contrast to the previous algorithm, an atomic register `countW` is now also used, which counts the number of waiting writer processes and whose access is protected by the semaphore `mutexW`. The counter `countR` for the number of reader processes is protected by the semaphore `mutexR` as before. As before, the semaphore `w` is used to prevent writer processes from entering the critical section if another writing or reading process is already in the critical section. New is the semaphore `r`, which is used to block reading processes. The semaphore `mutex` now has a different task than before. This will be explained later.

The program for writer processes can be explained as follows: Lines (5) to (7) are exactly the same as before: Only if the writing process holds the semaphore `w` is it allowed to enter the critical section. In lines (1) to (4), the counter `countW` (the number of writing processes that want to enter the critical section) is increased exclusively by protection using the semaphore `mutexW`. The first writing process is given a special role in line (3): It waits at the semaphore `r`, on the one hand, to ensure that there are no readers in the critical section and, on the other hand (as soon as it is no longer blocked), it ensures that no more writing processes enter the critical section as long as there are still writing processes present. In lines (8)-(10), the number of writing processes (exclusively protected by `mutexW`) is reduced. The last writing process releases access for reading processes using the **signal**(r) operation.

When looking at the reader program, it can first be noted that, except for lines (1), (2), (7), and (8), the functionality is the same as before. The **wait**(r) and **signal**(r) calls serve to block reading processes if writing processes are present. The two nested calls for the semaphore `mutex` ensure that at most one reading process waits at the (blocked) semaphore `r`. This gives writing processes priority over reading processes: When the first writer process performs the **wait**(r) operation in its line (3), at most one reader can already be waiting blocked at `r`. If the operations for `mutex` were omitted, any number of reading processes could wait there and compete with the first writing process. Furthermore, it can be seen that as long as there are writing processes, they always have priority over reading processes, since the semaphore `r` is only released when

Initial: countR, countW: atomic registers, initially 0
mutexR, mutexW, mutex, w, r: binary semaphores, initially 1

Program for reader:

```
(1)  wait(mutex);
(2)  wait(r);
(3)  wait(mutexR);
(4)  countR := countR + 1;
(5)  if countR = 1 then wait(w);
(6)  signal(mutexR);
(7)  signal(r);
(8)  signal(mutex);
(9)  Critical section;
(10) wait(mutexR);
(11) countR := countR - 1;
(12) if countR = 0 then signal(w);
(13) signal(mutexR);
```

Program for Writer:

```
(1)  wait(mutexW);
(2)  countW := countW + 1;
(3)  if countW = 1 then wait(r);
(4)  signal(mutexW);
(5)  wait(w);
(6)  Critical section;
(7)  signal(w);
(8)  wait(mutexW);
(9)  countW := countW - 1;
(10) if countW = 0 then signal(r);
(11) signal(mutexW);
```

Figure 3.19: Readers & Writers with Priority for Writers

the last writing process leaves the critical section.

3.4 Monitors

This section explains another programming abstraction – so-called *monitors*. Although semaphores can be used to solve many problems of concurrent programming relatively elegantly and easily, semaphores have some disadvantages. For example, they contradict the principle of structured programming: as soon as a **signal** call is missing at the end, a (global) deadlock of the program can occur, or vice versa: if a **wait** call is missing before accessing critical sections, race conditions can occur. The problem here is that it is the programmer's job to ensure that the appropriate **wait** and **signal** operations are always present. Therefore, it would be advantageous if the program structure (syntax) forced the programmer to use the appropriate operations correctly. This is where monitors come in. They follow the paradigm of structured programming, as found in object-oriented programming: a monitor consists of data (attributes) and methods that are allowed to read and change the values of the attributes.

Figure 3.20 shows a monitor definition for a bank account. The keyword **monitor** is used to introduce a monitor definition. This is followed by the attributes and then the methods.

```
monitor Account {  
    int Balance;  
    int AccountNumber;  
    int CustomerId  
  
    withdraw(int x) {  
        balance := balance - x;  
    }  
  
    deposit(int x) {  
        balance := balance + x;  
    }  
}
```

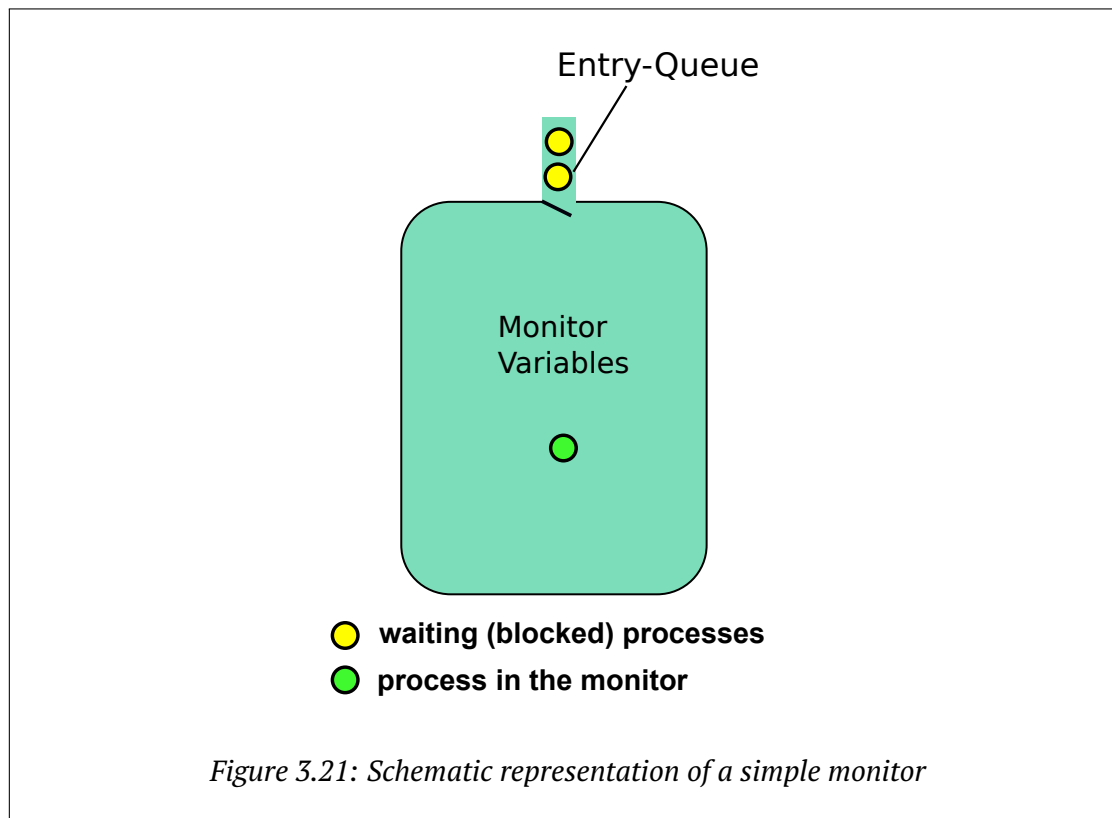
Figure 3.20: Monitor for a bank account

The account monitor has attributes for the account balance, the account number, and a customer identification number. There are two methods: one to credit an amount to the account and another to debit an amount.

When a monitor method is called (e.g., `account.withdraw(10)`), it is implicitly ensured that only one method is executed atomically at a time. For this purpose, there is a monitor lock that only one process can occupy at a time. When the monitor method

is called, the process attempts to occupy the monitor lock. If it is free, it is occupied, the method is executed, and the lock is then released again. If the lock is occupied, the process joins the set of waiting processes (and is blocked). When the lock is released, one of the waiting processes is unblocked. Depending on the implementation, the set of waiting processes can be implemented as a FIFO queue or as a simple set.

The graphic in Figure 3.21 symbolically represents a monitor. At the input, there is a queue of waiting processes, and only one process has access to the monitor variables. It should be noted that access to the monitor attributes is only permitted via monitor methods (i.e., direct access is prohibited).



In general, mutual exclusion can be implemented using a monitor, by having all variables of the critical section “monitored” by the monitor, i.e., defined as attributes of the monitor. Figure 3.22 shows such an implementation: The monitor mutex contains the variables of the critical section and the method `inCriticalSection` (this may receive parameters to pass corresponding values). The monitor ensures that only one process is in the critical section at a time, since only one method call is executed at a time. The implementation is also deadlock-free, since one of the concurrent calls is always executed. Starvation-free operation is only guaranteed if the monitor lock is managed by a FIFO queue, which is not always the case.

The monitors defined so far solve the problem of semaphores, since no explicit **sig-**

```

monitor Mutex {
    ...
    Variables of the critical section
    ...
    inCriticalSection(...) {
        Critical section;
    }
}

```

```

Process i:
loop forever
(1) Remaining code
(2) Mutex.inCriticalSection()
end loop

```

Figure 3.22: Mutual exclusion with a monitor

nal and **wait** are necessary to program mutual exclusion, but some important functionalities provided by semaphores are still missing. Many solutions to the problems of concurrent programming require the explicit blocking of processes until a condition is met. An example of this is the bounded buffer for the producer/consumer problem: The buffer must be programmed so that a producer is blocked if the buffer is full, and similarly, a consumer must be blocked if the buffer is empty. The previous monitor definition does not allow for implementation here. Therefore, monitors provide additional constructs to accomplish the explicit blocking and unblocking of processes. A distinction is made here between two different types of monitors: On the one hand, there are monitors that provide so-called *condition variables*, and on the other hand, there are monitors with so-called *condition expressions*. Here, we will mainly deal with the more commonly used condition variables and later briefly discuss condition expressions.

3.4.1 Monitors with Condition Variables

A *condition variable* is a FIFO queue that can be accessed in the monitor using the predefined operations **waitC**, **signalC**, and **empty**. A condition variable is defined in the attribute definition block with the keyword **condition**. The keyword is followed by the name of the condition variable to be defined. This name is often chosen to correspond to the condition that is being waited for. It should be noted that this is only a variable name, i.e., there is no logical connection between the name and the condition.

The semantics of the three available operations for condition variables are described below. It should be noted at this point that there are differences between the operations known from semaphores **wait** and **signal** and the operations **signalC** and **waitC** for condition variables.

The **empty** operation checks whether the queue for the condition variable is empty,

i.e., whether there are waiting processes. In pseudo-code, this can be written for the condition variable `cond` as:

```
empty(cond) {  
    return(cond = empty);  
}
```

The **waitC** operation causes the calling process to be blocked and added to the queue of the condition variable. Since the operation is executed within a monitor, the calling process has occupied the lock of the monitor at the time of the call. Since the process is now blocked, it releases the lock through the **waitC** operation.

Let P be the calling process, `cond` a condition variable in the monitor, and `monitor.lock` the implicit lock of the monitor (the lock on the entry queue). The semantics of **waitC** can be described by the following pseudo-code (which is executed atomically):

```
waitC(cond) {  
    cond := append(cond,P); //Process is added to the queue  
    P.state := blocked;     //Process blocked  
    monitor.lock := release; //Monitor lock is released  
}
```

The **signalC** operation unblocks the first process in the queue. If there are no processes, the **signalC** operation has no effect. In pseudo-code, this can be written as:

```
signalC(cond) {  
    if cond ≠ empty then  
        Q := head(cond); //First process in the queue  
        cond := tail(cond); //removed from queue  
        Q.state := ready;   //and unblocked  
}
```

However, this creates a problem. The monitor ensures that only one process at a time is allowed to perform calculations in a monitor method. With the semantics specified so far, both processes—the process calling **signalC** and the unblocked process—can continue to calculate. The problem is solved by assuming that the process calling **signalC** releases the lock to the unblocked process. This means that the process calling **signalC** is blocked and must reacquire the monitor lock. For most algorithms, this is not a major issue, since **signalC** is the last operation of a method. However, the problem and various solutions to it will be discussed in more detail later.

As an example of the use of monitors, we will consider the implementation of a semaphore using monitors. The code is shown in Figure 3.23.

The $S.V$ component of the semaphore is represented by the monitor variable `s`. The condition variable `notZero` is used to wait for `s` to take on a value unequal to 0. The implemented wait operation tests whether `s` has the value 0. If this is the case, the process waits at the condition variable `notZero`. If `s` has a value greater than 0 and after unblocking a previously blocked process, the value `s` is decreased by 1. Decreasing

```

monitor Semaphore {
    int s := k;
    condition notZero;

    wait() {
        if s = 0 then
            waitC(notZero)
        s := s - 1
    }

    signal() {
        s := s + 1
        signalC(notZero)
    }
}

```

Figure 3.23: Semaphore simulation with a monitor

without prior blocking corresponds to the behavior of a semaphore; the other case initially appears to be incorrect. However, the correctness can be seen when explaining the signal implementation: Here, the value of s is always increased first, and then a blocked process is unblocked. Note that **signalC** has no effect if there are no blocked processes. Since signal also increases the value of s when a process is unblocked, and the unblocked process (within **wait**()) decreases the value of s again, the overall result is that the value of s remains the same in this case. This means that the implementation of the semaphore is correct.

If we compare the **wait** and **signal** operations of classic semaphores and the **waitC** and **signalC** operations of monitors, the differences can be summarized in the following table:

Semaphores Sem	Monitors (Condition Variable cond)
wait (Sem) can cause blocking, but not necessarily (if $S.V > 0$, no blocking, but lowering of $S.V$)	waitC (cond) always blocks the calling process
signal (Sem) always has an effect: Unblocking a process from $S.M$ or Increasing Sem.V	signalC (cond) can be ineffective: Either a process in the queue cond is unblocked, or – if cond is empty – no effect is observed.

3.4.1.1 Monitor solution for the producer/consumer problem

In this section, we consider a solution to the producer/consumer problem using a monitor. The code for a bounded buffer is shown in Figure 3.24.

```

monitor BoundedBuffer {
  bufferType buffer := empty;
  condition notEmpty;
  condition notFull;

  produce(v) {
    if length(buffer) =  $N$  then
      waitC(notFull);
    buffer := append(v,buffer);
    signalC(notEmpty);
  }

  consume() {
    if length(buffer) = 0 then
      waitC(notEmpty)
    w := head(buffer);
    buffer := tail(buffer);
    signalC(notFull)
    return(w);  }
}

```

Producer code:

```

loop forever
(1) create e
(2) BoundedBuffer.produce(e)
end loop

```

Consumer code:

```

loop forever
(1) e := BoundedBuffer.consume();
(2) consume e;
end loop

```

Figure 3.24: Bounded Buffer as Monitor

The monitor has a list of buffers (of type `bufferType`) as monitored objects. The maximum size of the buffer is encoded directly as a number N . Two condition variables are used. The condition variable `notFull` is waited on if the buffer is full, and the condition variable `notEmpty` is waited on if the buffer is empty. The `produce` method first checks whether the length of the buffer is equal to the maximum size N . If this is the case, the buffer is full and the producing process waits at the condition variable `notFull` until the buffer is no longer full. The producer then appends its produced element to the list and finally calls `signalC(notEmpty)` to signal that the buffer is not empty.

This unblocks a waiting consumer (if there is at least one). The `consume` method first checks whether the buffer is empty. If this is the case, the calling process is blocked and waits at the condition variable `notEmpty` for elements to be added to the buffer again. Then (or immediately), the first element is removed from the list, a **signalC** operation is performed for the condition variable `notFull` to wake up a waiting producer, and finally the first value of the buffer is returned as the result.

The programs for the producer and consumer are easy to explain. They use the corresponding methods of the `BoundedBuffer` monitor to add elements to the buffer or remove them from the buffer.

3.4.2 Different Types of Monitors

Before presenting further monitor solutions to classic problems of concurrent programming, the problem already mentioned after a **signalC** operation is considered: On the one hand, it is unclear which process obtains the monitor lock after unblocking. On the other hand, it is not clear where (i.e., at which queue) the process that does not receive the monitor lock must wait.

To explain the problem in more detail, the model of monitors is refined. A condition variable no longer consists of just one queue, but uses three queues: the condition queue (there is one per condition variable), the wait queue, and the signal queue (there is one per monitor). queue, but uses three queues: the condition queue (there is one per condition variable), the wait queue, and the signal queue (there is one per monitor). These are used as follows:

Condition queue: The processes blocked by **waitC** wait at the condition queue. This queue is the one we are already familiar with and used implicitly in the previous monitor specification.

Wait queue: If a **signalC** operation is executed and the condition queue is not empty, the first waiting process in the condition queue is added to the wait queue.

Signal queue: The process that executes **signalC** is placed in the signal queue *if* it has unblocked a process in the condition queue.

There is one condition queue for each condition variable, and one wait queue, one signal queue, and one entry queue per monitor. The processes that want to enter the monitor for the first time wait at the entry queue.

Figure 3.25 illustrates the monitor with the three global queues and the condition queues.

The resulting model now assumes that after an unblocking **signalC** operation, no one automatically obtains the monitor lock, but rather all processes waiting at the monitor lock can reoccupy it (these are the processes at the entry, wait, and signal queues). Priorities can be assigned to these three different queues, which specify which processes in which queue are allowed to occupy the monitor lock first. The priorities are designated as follows:

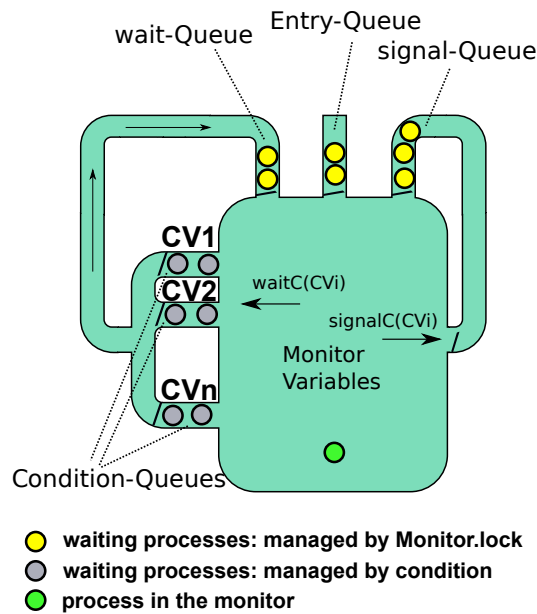


Figure 3.25: Illustration of a monitor with wait and signal queues for the condition variables

- E for the entry queue
- W for the wait queue
- S for the signal queue

A higher priority means that the corresponding queue takes precedence over the other queue. Comparing the three priorities, 13 different constellations can be examined. Figure 3.26 shows a table with all 13 possibilities.

Combinations in which the entry queue has a higher priority than the wait and signal queues (options 7, 12, and 13 in the table) are not very useful, since processes that have already entered the monitor may have to wait for any length of time because newly arriving processes can always “overtake” them. Options 8-11 are also not very useful, since in these cases the entry queue has higher priority than one of the other two queues. This means that “old” processes can be “overtaken” by “new” processes as often as desired.

The classic definition of monitors can be described by the sixth option, i.e., $E < S < W$. In this variant, after a **signalC** operation, the monitor lock is immediately “handed over” to the unblocked process. When this process leaves the monitor, the signaling process is allowed to continue computing. New processes have the lowest priority. Monitors with this strategy are referred to as *Signal and Urgent Wait* monitors.

1	$E = W = S$	
2	$E = W < S$	Wait and Notify
3	$E = S < W$	Signal and Wait
4	$E < W = S$	
5	$E < W < S$	Signal and Continue
6	$E < S < W$	Signal and Urgent Wait, classic definition
7	$E > W = S$	not useful
8	$E = S > W$	not meaningful
9	$S > E > W$	not meaningful
10	$E = W > S$	not meaningful
11	$W > E > S$	not meaningful
12	$E > S > W$	not meaningful
13	$E > W > S$	not meaningful

Figure 3.26: 13 possibilities for distributing priorities

Alternatively, one could also say that the monitor fulfills the *Immediate Resumption Requirement*.

The names (as far as they are known) for other combinations can be found in the table. The second constellation, i.e., $E = W < S$, is also worth mentioning. In this case, the signaling process may continue to compute. The unblocked process is treated in the same way as a new process. This strategy is used in Java (this will be explained in more detail later). The problem with this variant is that at the time when the unblocked process is allowed to continue computing, the condition that was true at the time of unblocking may now be false again. For this reason, such monitors perform a circular check of the condition. If the condition has become false again in the meantime, the process waits again. Consider, for example, a method that only decreases the value of the variable s if it is not equal to zero. The condition variable `sIsNotZero` is used to wait until s is not equal to zero. The circular checking of the condition can then be expressed as:

```
while s = 0 do
  waitC(sIsNotZero);
  s := s - 1;
```

The conclusion of these different monitor variants should be that, depending on the monitor implementation of the respective programming languages, slightly different programs must be written. One should therefore be aware that not all monitor implementations have exactly the same behavior.

3.4.3 Monitors with Condition Expressions

As already mentioned, there are monitors that do not have condition variables, but rather so-called *condition expressions*. These are Boolean expressions whose truth value

can be tested directly with the **waitCE** operation. The operation is called **waitCE** to distinguish it from the other wait operations. If the Boolean expression is false, the process waits (it is blocked and inserted into a corresponding queue).

There is no signal operation for condition expressions. Instead, waiting processes are automatically unblocked by the runtime system as soon as the Boolean expression becomes true. As with condition variables, there are several possibilities for when these processes obtain the monitor lock. However, this will not be discussed further here. To conclude the discussion of condition expressions, an implementation for the producer/consumer problem with a bounded buffer using condition expressions is presented. Here, it is assumed that blocked processes immediately obtain the monitor lock when the corresponding Boolean expression becomes true.

Figure 3.27 shows the implementation of the buffer and the code for the producer and the consumer.

```

monitor BoundedBuffer {
    bufferType buffer := empty;

    produce(v) {
        waitCE(length(buffer) < N); // Buffer full?
        buffer := append(v,buffer);
    }

    consume() {
        waitCE(length(buffer) > 0); // Buffer empty?
        w := head(buffer);
        buffer := tail(buffer);
        return(w);    }
}

```

Producer code:

```

loop forever
(1) create e
(2) BoundedBuffer.produce(e)
end loop

```

Consumer code:

```

loop forever
(1) e := BoundedBuffer.consume();
(2) consume e;
end loop

```

Figure 3.27: Bounded Buffer as Monitor with Condition Expressions

3.5 Some Application Examples with Monitors

This section explains implementations for classic problems of concurrent programming with monitors. Monitors with condition variables and the “Immediate Resumption Requirement” are used, i.e., the priorities $E < S < W$ are assumed.

3.5.1 The Readers & Writers Problem

As already explained in Section 3.3.7, the Readers & Writers problem consists of granting parallel read access to as many processes as possible, while writing processes require exclusive access to the critical section (the database, etc.). Figure 3.28 shows a monitor implementation for the readers & writers problem. The monitor uses two variables of type `int` to count the number of reading and writing processes (variables `countR` and `countW`). In addition, two condition variables are used: `okToWrite` and `okToRead`. Writer processes wait at the first condition variable `okToWrite`, while reading processes wait at the second condition variable `okToRead` to start reading.

The monitor contains four methods: one method each for starting read and write access and one method each for ending read and write access. Note that the actual reading must not take place within the monitor, as the monitor properties would then mean that a maximum of one process would be allowed to access the data simultaneously. An explanation for the two methods for writer processes is as follows: In the `startWrite()` method, a writer process first checks whether both counters `countR` and `countW` have the value 0. If this is the case, the counter `countW` is set to 1 and the writer process is then allowed to enter the critical section. In other cases, the writer process must wait at the condition variable `okToWrite`. After leaving the critical section, a writer process calls the `endWrite()` method. Within this method, the count variable `countW` is first decremented by 1. Then the **empty** method is used to determine whether there are any waiting reader processes. If this is not the case, the next writer process is allowed to enter the critical section. Otherwise, a reader process is unblocked.

Now let’s look at the methods for reader processes: In the `startRead()` method, a reading process first tests whether a writing process is already in the critical section (in which case `countW  $\neq$  0`) or whether there are any waiting writer processes. In both cases, the reading process waits at the condition variable `okToRead`. Then (or immediately, if there was no need to wait), the counter `countR` is incremented. Using the last command of the `startRead()` method, a reader that is about to enter the critical section unblocks another reader. This continues in a cascading manner, so that all waiting read processes can enter the critical section at the same time. The `endRead()` method is called after leaving the critical section. The counter `countR` is decremented and the last process to leave unblocks any writer processes that may be present.

3.5.2 The Dining Philosophers

A possible solution to the dining philosophers problem using monitors is shown in Figure 3.29. Since no semaphores are to be (or can be) used, the forks are not explicitly

Monitor definition:

```
monitor RW {  
  int countR, countW := 0;  
  condition okToWrite, okToRead;  
  
  startRead() {  
    if countW  $\neq$  0 or not(empty(okToWrite))  
    then waitC(okToRead);  
    countR := countR + 1;  
    signalC(okToRead);  
  }  
  
  endRead() {  
    countR := countR - 1;  
    if countR = 0 then signalC(okToWrite);  
  }  
  
  startWrite() {  
    if countR  $\neq$  0 or countW  $\neq$  0  
    then waitC(okToWrite);  
    countW := countW + 1;  
  }  
  
  endWrite() {  
    countW := countW - 1;  
    if empty(okToRead)  
    then signalC(okToWrite);  
    else signalC(okToRead);  
  }  
}
```

Reader program:

- (1) RW.startRead();
- (2) Read;
- (3) RW.endRead();

Writer program:

- (1) RW.startWrite();
- (2) Write;
- (3) RW.endWrite();

Figure 3.28: Readers & Writers with monitor

shown. Instead, each philosopher receives an entry in the forks field. The entry `forks[i]` indicates how many of the maximum of two forks are currently available to the philosopher with number i . A philosopher can only eat if both of his forks are available, i.e., if `forks[i]=2` is true.

The operations $i + 1$ and $i - 1$ are meant to be “modulo N ”, so that $i - 1$ for $i = 1$ returns the number N and $i + 1$ for $i = 1$ returns the number 1.

```

monitor Forks {
    constant numberPhil :=  $N$ ;
    int array [1,...,numberPhil] forks := [2,...,2];
    condition array [1,..., numberPhil] okToEat;

    takeForks(i) {
        if forks[i]  $\neq$  2 then waitC(okToEat[i]);
        forks[i+1] := forks[i+1]-1;
        forks[i-1] := forks[i-1]-1;}

    releaseForks(i) {
        forks[i+1] := forks[i+1]+1;
        forks[i-1] := forks[i-1]+1;
        if forks[i+1] = 2 then signalC(okToEat[i+1]);
        if forks[i-1] = 2 then signalC(okToEat[i-1]);}

}

```

Program of philosopher i :

```

loop forever
(1) Think;
(2) Forks.takeForks(i);
(3) Eat;
(4) Forks.releaseForks(i);
end loop

```

Figure 3.29: Monitor implementation for the dining philosophers

A philosopher calls the monitor method `takeForks( $i$ )` before he can eat. Within the method, it is first tested whether both forks are available. If this is the case, the philosopher grabs the forks and lowers the fork entries of his left and right neighbors. If there are not enough forks available, the philosopher blocks at the condition variable `okToEat[i]`.

After the philosopher has finished eating, he calls the `releaseForks` method. First, he puts down the forks and increases the fork entries of his neighbors (who now each have one more fork again). In the last two **if** queries, the philosopher checks whether his neighbors now have two forks available. If this is the case, the corresponding philosopher is unblocked. Note that the immediate resumption requirement is assumed, i.e., when the first of the two neighbors is unblocked, it will modify the forks entries before

the signaling process queries the forks entries of its other neighbor.

It can be proven that this implementation is indeed deadlock-free. However, it is not starvation-free.

3.5.3 The Sleeping Barber Problem

The Sleeping Barber problem has already been described in Section 3.3.4. Figure 3.30 shows an implementation of the problem using a monitor.

The barber runs in an endless loop and first calls the monitor method `takeCustomer()`. Here, he uses the count variable `waiting` to check whether there are any customers waiting. If this is not the case, the barber waits at the condition variable `customersAvailable`. As soon as he is unblocked, the hairdresser decrements the number of waiting customers and signals to a customer that he is ready to serve him (by means of a `signalC` operation on the condition variable `hairdresserReady`). After the hairdresser has cut the customer's hair, he calls the method `beendeKunden()`. Here, he waits at the condition variable `ladenVerlassen` until the customer whose hair he has just cut signals that he has left the shop.

The customer processes are numbered consecutively for this solution, so that each customer is assigned a unique number. The customer first calls the monitor method `getHaircut`. First, the customer checks whether n customers are already waiting. If this is the case, he leaves the shop immediately. Otherwise, they increase the number of waiting customers, signal to the hairdresser that there are customers, and wait at the condition variable `hairdresserReady` for the hairdresser to take them. Depending on whether all waiting places were occupied or not, the method `getHairstyle()` returns `False` or `True`. Only if the value `True` is returned does the customer get his new hairstyle and then call the method `leaveShop` to signal to the hairdresser (at the condition variable `leaveShop`) that he has left the shop.

3.5.4 Barriers

Barriers are used for algorithms that run in several phases and where it is necessary to wait until all processes have completed the previous phase before all processes enter the next phase. Figure 3.31 shows an implementation of a barrier using a monitor. This uses a new operation on condition variables that was not previously defined:

The operation **`signalAllC(cond)`** unblocks all processes waiting at the condition queue of the condition variable `cond` and adds them to the wait queue. This step is atomic.

The monitor implementation uses a condition variable `alleAngekommen`, a variable `angekommen` to count the processes that have arrived at the barrier, and a variable `maxProzesse` that specifies the number of processes to be synchronized.

At the end of its phase, a process executes a call to the `barrier()` method. Within this method, `arrived` is first incremented. Then the processes wait at the condition variable `allArrived` until the last process: This process sets `arrived` to 0 and calls the **`sig-`**

```
monitor Barbershop {  
  int waiting:= 0;  
  condition customerPresent;  
  condition barberReady;  
  condition leaveShop;  
  
  // Methods for the barber  
  
  takeCustomer() {  
    if waiting = 0 then  
      waitC(customerPresent);  
    waiting := waiting - 1;  
    signalC(barberReady)}  
  
  endCustomers() {  
    waitC(leaveShop)}  
  
  // Methods for the customer  
  
  getHairstyle() {  
    if waiting < N then  
      waiting := waiting + 1;  
      signalC(customerPresent)  
      waitC(barberReady)  
      return(True);  
    else return(False);}  
  
  leaveShop(){  
    signalC(leaveShop)}  
}
```

Barber:

```
loop forever  
  Barbershop.takeCustomer();  
  CutHair;  
  Barbershop.finishCustomer();
```

Customer:

```
if Barbershop.getHaircut()  
then  
  getHaircut;  
  Barbershop.leaveShop();
```

Figure 3.30: Sleeping Barber with Monitor

nalAllC operation on the condition variable `alleAngekommen`, allowing all waiting processes to enter the next phase.

```

monitor Barrier {
  int arrived := 0;
  int maxProcesses := N;
  condition allArrived;

  barrier () {
    arrived := arrived + 1;
    if arrived < maxProcesses then
      waitC(allArrived);
    else
      arrived := 0;
      signalAllC(allArrived);
  }
}

Process i:
loop forever
  Code before the barrier
  Barrier.barrier()
  Code after the barrier
end loop

```

Figure 3.31: Barrier implementation with monitor

3.6 Channels

The programming constructs examined so far – semaphores and monitors – use shared memory to enable communication between processes. This section looks at so-called *channels*, which are used to communicate without shared memory, but rather by exchanging messages. Such channels can also be implemented in systems with shared memory, but the key point is that they do not necessarily require shared memory. This makes channels suitable for distributed systems, for example. This chapter considers *synchronous* communication (i.e., synchronous channels), meaning that data exchange between two processes occurs in one step and not *asynchronously*. If a sending process wants to send data but the receiving process is not yet ready, the sending process is blocked until the transmission can take place. Conversely, receivers are blocked until a sending process that is not yet ready is ready to send.

Channels were introduced by C.A.R. Hoare in the so-called “Communicating Sequential Processes” (CSP) formalism. This type of concurrent programming has influenced some programming languages and is at least provided as a library for some

programming languages. Channels are built into the Go programming language (<http://golang.org>) natively.

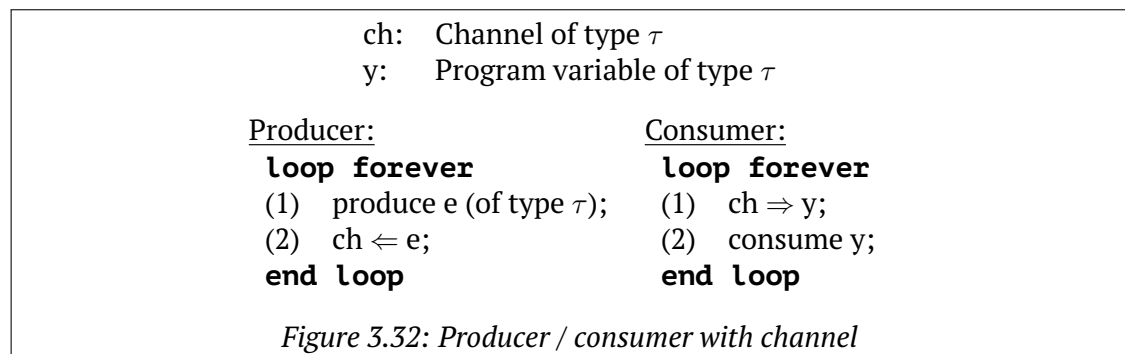
3.6.1 Definition of Channels

A channel connects a sending process to a receiving process. Channels are generally typed, i.e., only elements (i.e., messages) of the same type may be sent via a channel. Channels have a name, and there are two operations on channels. Let ch be a channel, then we write

- $ch \Leftarrow w$ if the value w is to be sent via the channel ch . Here, w must be a value of the appropriate type. For simplicity, we also write $ch \Leftarrow x$ for a program variable x , meaning that the value of the variable x is sent via the channel ch .
- $ch \Rightarrow y$ means that the calling process wants to receive an element via the channel ch . The received value is assigned to the program variable y . Note that in the construct $ch \Rightarrow e$ only program variables are allowed at the position of e .

The execution of concurrent programs with channel operations then requires that communication between the sending and receiving processes only takes place when both program pointers have reached the corresponding operations.

For clarification, consider the following example for solving the producer/consumer problem, whereby not any number of elements can be stored in the buffer, but rather this buffer cannot actually temporarily store any elements at all. Figure 3.32 shows the program code for the producer and consumer processes.



The two processes must be in the corresponding lines of code to communicate via the channel ch (the producer in line (2), the consumer in line (1)). The execution of these two lines then takes place atomically in one step, i.e., the value is sent from the consumer to the producer in one step. For example, if the producer is in line (2) but the consumer is still consuming a previously received element (i.e., also in its line (2)), then the producer process is automatically blocked and must wait until the consumer is in its line (1). A similar case occurs when the producer is still in line (1), but the consumer is in its line (1) and wants to receive: In this case, the consumer is blocked until the producer is in line (2). This blocking and unblocking happens automatically.

The question arises as to what happens if, for example, two processes want to send on the same channel to which a receiver is connected. In this case, it is not immediately clear which of the sending processes is allowed to send its message to the receiver. In fact, this is decided quasi randomly by the scheduler. Any possibility can occur. Consider the following example with three processes:

<u>Process 1:</u>	<u>Process 2:</u>	<u>Process 3:</u>
<code>ch $\Leftarrow$ True</code>	<code>ch $\Leftarrow$ False</code>	<code>ch $\Rightarrow$ x</code>
		<code>print x;</code>

Processes 1 and 2 both attempt to send a message via the channel `ch`. Depending on the order of execution, process 3 will receive the message from process 1 or the message from process 2. This means that depending on how the program runs, either `True` or `False` will be printed. This non-determinism naturally also occurs when there are multiple receivers but only one sender for a channel. On the one hand, one could argue that this is a race condition, since the execution of the program influences the result. On the other hand, such an effect is often desirable. Consider, for example, a situation where there are several servers, all offering the same service (and waiting on the same channel to process a request from a client): If many clients access the servers, a free server is almost always selected at random.

3.6.2 Mutual Exclusion with Channels

Now we will consider the mutual exclusion problem when using channels. Mutual exclusion can be implemented with channels, for example, by defining a process as a guard for the critical section. This guard process is connected via a channel to all processes that want to enter the critical section. Only after the guard has sent a message to one of the other processes is that process allowed to enter the critical section. In the exit code, the process sends a message to the guard, letting it know that the critical section is free again. Figure 3.33 shows the code for this idea. However, after leaving the critical section, the message can also be sent to another waiting process (it does not necessarily have to arrive at the watcher), which can then enter the critical section directly (i.e., the watcher is bypassed, so to speak). However, this is not a problem. The guard is necessary, however: Consider, for example, the case where only one process wants to enter the critical section. Without a guard, it would neither be able to enter the critical section nor exit it again.

The solution satisfies mutual exclusion and is deadlock-free, since a process will always receive a message. However, it is not starvation-free, since a process can always be "ignored" and never receive a message. Note that this case is not excluded by the fairness assumption, since the waiting process cannot perform an evaluation step at any time (it can only do so when the guard is ready to send).

```
mutex: channel "of type Bool
local: local variable, initial value irrelevant
```

```
Process i
```

```
loop forever
```

```
(1) Remaining code;
```

```
(2) mutex  $\Rightarrow$  local;
```

```
(3) Critical section;
```

```
(4) mutex  $\Leftarrow$  True;
```

```
end loop
```

```
Watcher
```

```
loop forever
```

```
(1) mutex  $\Leftarrow$  True;
```

```
(2) mutex  $\Rightarrow$  local;
```

```
end loop
```

Figure 3.33: Mutual exclusion with channels

3.6.3 Modeling Shared Memory Using Channels

Although there is no shared memory in the channel model through which the processes can communicate, it can be simulated using channels. For this purpose, a single memory cell is considered that can be read and written to. There is a server process that manages the cell and accepts read and write operations. Figure 3.34 shows the program code for the server and the implementation of the access methods read and write. The implementation uses two channels: The channel requestChannel is used to send requests to the cell, while the channel replyChannel is used during a read request to send the result (the value of the cell) back to the requesting process. The requestChannel channel expects pairs, where the first component is a Boolean value (this determines whether it is a read or write request (True = write access, False = read access)) and the second component is of the type of the cell content (abstractly as CellType). The second component only plays a role in write accesses, because it contains the new value to be set for the cell. The server program waits for a message on the requestChannel. Depending on whether the first component of the message is true or false, either the value of the cell is reset or the previous value is sent to the requesting process via the replyChannel. The programs for the two access methods are analogous, whereby a read request must also send a dummy value for the second component of the pair.

The local variable x in the server process can be omitted if the server process is defined recursively. This variant is shown in Figure 3.35.

Data types:

CellType is the type of the cell, e.g., Bool, Int, ...

RequestChannelType = (Bool, CellType)

Channels:

requestChannel : Channel "of type RequestChannelType

replyChannel: Channel "of type CellType

Variables:

x: Local variable of the server process of type CellType

dummy: Any variable of type CellType

Process for the cell (server process):

```
loop forever
  requestChannel  $\Rightarrow$  r;
  if fst(r) then // write operation
    x := snd(r);
  else // read operation
    replyChannel  $\Leftarrow$  x;
end loop
```

Methods for accessing the cell:

```
read(requestChannel, replyChannel) {
  requestChannel  $\Leftarrow$  (False, dummy);
  replyChannel  $\Rightarrow$  x;
  return(x);
}

write(reqCh, replyCh, x) {
  requestChannel  $\Leftarrow$  (True, x)
}
```

Figure 3.34: Implementation of a cell with channels

Process for the cell (server process):

```
cell(x) {
  requestChannel  $\Rightarrow$  r;
  if fst(r) then // write operation
    cell(snd(r));
  else // read operation
    replyChannel  $\Leftarrow$  x;
  cell(x); }
```

Figure 3.35: Process for the cell with recursion

3.6.4 Selective Input

Many languages based on CSP have another construct that makes programming more convenient. This is known as “selective input”: Previously, a message-receiving process had to wait for a message on exactly one channel. With “selective input”, it is possible to wait on multiple channels and, as soon as a message is received on one channel, to continue. This can be written as a “or” operation of the receive primitives:

```
either
  ch1  $\Rightarrow$  var1
or
  ch2  $\Rightarrow$  var2
or
  ch3  $\Rightarrow$  var3
```

If a process executes this command, it can receive messages on channels `ch1`, `ch2`, or `ch3`. If there are several different senders, the evaluation randomly (or non-deterministically) selects one of the possible communications. This allows many problems to be solved more elegantly. For example, a server service can wait on multiple channels to receive a request from a client. As soon as a request arrives on one of the channels, the server processes only that request. Implicitly, this blocks other clients on other channels until the server is ready again (and provides the non-deterministic selection).

In a later chapter, more detailed theoretical foundations for such message-passing models will be discussed.

3.6.5 Dining Philosophers with Channels

We consider a solution to the dining philosophers problem using channels. Here, each fork is represented by a process that is connected to both the left and right philosopher processes via a channel. Figure 3.36 shows the code for the philosophers and the forks.

A philosopher attempts to obtain both forks one after the other by receiving something (True) via the associated channels. If this is successful, he can eat. He then puts the forks back by sending True back via the channels associated with the forks. The fork processes do nothing but alternately send True and then receive something. If forks are already occupied, the corresponding philosopher is automatically blocked because he cannot receive a message.

This implementation is initially only a model and not yet a solution to the problem, as a deadlock can occur (each philosopher could have the left fork, then none can obtain the right one). However, the programs can be easily modified to achieve a correct solution. For example, the last philosopher could lift the forks in reverse order.

```
forks : field of channels of type Bool
x:      local variables
```

Philosopher

loop forever

```
(1) Think;
(2) forks[i] ⇒ x
(3) forks[i+1] ⇒ x
(4) Eat;
(5) forks[i] ⇐ True
(6) forks[i+1] ⇐ True
```

end loop

Fork i

loop forever

```
(1) forks[i] ⇐ True
(2) forks[i] ⇒ x
```

end loop

Figure 3.36: Dining philosophers with channels (not deadlock-free)

3.7 References

The description of semaphores, monitors, channels, and Linda is essentially based on (Ben-Ari, 2006), although some of the syntax and algorithms were taken from (Taubenfeld, 2006). The classification of different types of monitors was examined in more detail in (Buhr et al., 1995). The implementation of cells using synchronous channels is based on (Reppy, 2007), with some modifications. An overview of Linda with many examples can be found in (Carriero & Gelernter, 1992; Gelernter, 1985), from which some of the examples were also taken.

4

Access to Multiple Resources

4.1 Deadlocks with Multiple Resources

This section considers deadlocks that can occur when processes want to acquire *multiple* resources. In Chapter 2, deadlocks and deadlock freedom were already defined for the mutual exclusion problem. However, in that setting, all processes had a similar program and entered only one critical section, which is equivalent to acquiring *one* resource. Here deadlocks are defined more generally, and the problem is considered that processes want to acquire multiple resources and block other processes from accessing them using locks (with semaphores, monitors, etc.).

Definition 4.1.1. *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

The event is generally the release of a resource.

Some situations in which such a deadlock can occur are:

- In the mutual exclusion problem, a deadlock occurs when it is no longer possible for any process to enter the critical section, even though there are processes that want to enter the critical section.
- Two processes that want to transfer funds from one bank account to another bank account. To ensure that the balance of the accounts remains consistent, the processes block access to the accounts by other processes during their transaction. The accounts are locked one after the other: first, the account from which money is debited is locked, then the account to which the amount is credited. Process P wants to transfer funds from bank account A to bank account B, while process Q wants to transfer funds from account B to account A. To make the example explicit, let `accountA` and `accountB` be binary semaphores to lock the two accounts. Processes P and Q then execute the following programs:

Process P

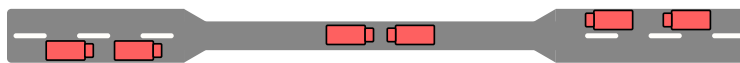
```
wait(AccountA);  
wait(AccountB);  
transfer from A to B  
signal(AccountA);  
signal(AccountB);
```

Process Q

```
wait(AccountB);  
wait(AccountA);  
transfer from B to A  
signal(AccountB);  
signal(AccountA);
```

In an unfortunate execution sequence, the program can end up in a deadlock: Process P occupies semaphore AccountA, then process Q occupies semaphore AccountB. Now both processes wait at the other semaphore, which is never released. A better solution to this problem would be for both processes to attempt to lock the accounts in the same order. In fact, then no deadlock can occur.

- A similar example to the previous one has already been discussed in the incorrect solutions to the dining philosophers problem. These programs can end up in a deadlock if all philosophers have the left fork.
- A clear example is the crossing (by vehicles) of bridges or other bottlenecks that only have one lane. If the vehicles drive into the bottleneck in an uncontrolled manner, a deadlock can occur. The following image illustrates the situation



In general, there are four ways to deal with deadlocks:

Ignore: Take no precautions and assume that deadlocks are rare and therefore harmless. In practice, this approach is not uncommon.

Deadlock detection and elimination: Assume that the runtime system can detect deadlocks and resolve them. The task is to find an algorithm that detects deadlocks in the system and to specify a procedure for resolving them (e.g., releasing all locks, etc.).

Deadlock avoidance: Assume that the system can detect potential deadlocks when resources are requested and refuse such requests proactively. The goal is to implement a resource management algorithm that ensures deadlock-free access to resources.

Deadlock prevention: The programmer designs the programs so that deadlocks cannot occur.

The last approach is clearly the simplest from the perspective of the runtime system, and the safest from the programmer's point of view. The interesting question is therefore how to prevent deadlocks during programming.

Definition 4.1.2 (Necessary conditions for a deadlock). *For a deadlock to occur, the following four conditions must all be met simultaneously:*

Mutual exclusion: Only one process can access a resource at a time.

Hold and Wait: A process can request a resource (wait for a resource) while it already occupies another resource.

No Preemption: A resource can only be released (unlocked) by the process that occupied it.

Circular Wait: There is a cyclic dependency between waiting processes: Each waiting process wants access to the resource occupied by the next process in the cycle.

These four conditions are also called *Coffmann conditions*, named after Edward Coffman, Jr. and the paper (Coffman Jr. et al., 1971) which was the first paper that formulated the conditions.

To prevent deadlocks, it therefore suffices to ensure that at least one of the four conditions is never satisfied. The next two sections discuss methods for deadlock prevention and then a procedure for deadlock avoidance.

4.2 Deadlock Prevention

To prevent deadlocks during programming/design, it suffices to address one of the four necessary conditions from Definition 4.1.2.

Mutual exclusion can be eliminated in some situations by inserting a spooler between the processes. For example, access to the printer can be made non-exclusive by inserting a print server that receives print jobs from individual processes. However, there are many situations in which such solutions do not apply.

Eliminating the “no preemption” condition makes little sense, as this would mean, for example, revoking the printer from a process that has not yet finished printing.

The “hold and wait” condition can be addressed by requiring that a process request all the resources it needs at the outset. The problem, however, is that it is often difficult to know in advance which resources will be needed. This approach may result in far too many resources being blocked simultaneously. The philosopher’s program with an exclusive mutex is one such solution: only one philosopher at a time can access the forks, which, as we have seen, leads to poor resource utilization (only two of the N forks are in use at any one time).

A variation of this approach is the so-called two-phase locking protocol: Each process works in two phases:

1. Phase: The process attempts to acquire all required resources. If a required resource is not available, the process releases all acquired resources and restarts from phase 1.
2. Phase: The process holds all required resources. It completes its computations and then releases all resources.

To implement this with semaphores, non-blocking operations are needed that, instead of blocking when a semaphore cannot be acquired, return a catchable exception.

For the philosopher’s problem, this means: in phase 1, all philosophers attempt to pick up both forks. If a philosopher cannot get both forks, he puts back any fork he may have received. He only tries again when phase 1 begins again. The phases can be synchronized using barriers, for example, though global phase synchronization across all processes is not strictly necessary.

It can be shown that no deadlock occurs in this case. However, another problem may arise: a so-called *livelock*, where no process successfully acquires all required resources in phase 1, and this repeats indefinitely. For the philosopher's problem, a livelock can occur when all philosophers pick up their left fork and put it back because none can obtain the right fork. Unlike a deadlock, in a livelock no process is blocked—but no process makes progress either.

Another method of addressing the hold-and-wait condition is the so-called timestamp ordering procedure. The idea is to impose an ordering on processes: if two processes want to acquire the same resource, the one with the smaller timestamp wins. Before a process begins acquiring resources (as in two-phase locking), it receives a unique timestamp; each subsequent process receives a strictly larger timestamp. The two-phase locking protocol with timestamping then proceeds as follows:

1. Phase: The process attempts to lock all required resources at once. If a required resource is already held by a process with a smaller timestamp, the process releases all its resources and restarts from phase 1. However, if its own timestamp is smaller than that of the holding process, the process waits to acquire the remaining resource. This wait always succeeds, as all processes comply with the protocol.
2. Phase: Once the process has successfully entered this phase, it holds all required resources. It uses them and then releases them.

More precisely, a new timestamp is assigned only after a process has completed successfully; no new timestamp is assigned when restarting in phase 1. For the philosopher's problem, each philosopher receives a timestamp and communicates it to the forks it holds. If a fork is already held, the philosopher must determine whether its own timestamp is smaller than that of the holding process and act accordingly.

This procedure guarantees deadlock freedom and starvation freedom; livelocks cannot occur either. However, the implementation effort is substantial. Starvation is defined here more generally (not restricted to the mutual exclusion problem) as:

Definition 4.2.1. *Starvation is a situation in which a process is never (after any number of computation steps) able to occupy all the resources it needs.*

One drawback of the two-phase locking protocol (with timestamping) is that each process must know in advance which resources it needs in order to lock all of them in the first phase.

The most effective method of preventing deadlock is to eliminate the “circular wait” condition. In the philosopher's problem, the appropriate solution was to have the N -th philosopher pick up forks in reverse order. This breaks the cyclic dependency. A total order is imposed on the forks, with fork 1 being the smallest and fork N the largest. All philosophers acquire forks in ascending order (which is why the last philosopher acquires the right fork (fork 1) first, then the left fork (fork N)). The following theorem shows that a total order on resources prevents deadlocks:

Theorem 4.2.2 (Total Order Theorem). *If all shared resources are ordered by a total order and each process occupies its required resources in ascending order with respect to the total order, then a deadlock is impossible.*

Proof. The proof is by contradiction. The assumption is therefore initially that there is a deadlock, even though the resources have been occupied according to the total order. Since there is a deadlock, there must be circular waiting, i.e., there are resources $R_1, \dots, R_n$ and processes $P_1, \dots, P_n$ such that

- for $i = 2, \dots, n$: Process P_i has occupied resource R_{i-1} , and P_1 has occupied resource R_n
- for $i = 1, \dots, n$: Process P_i is waiting for resource R_i

Let R_j be the smallest resource from $\{R_1, \dots, R_n\}$ with respect to the total order. Then process P_j waits for resource R_j , having already occupied resource R_{j-1} (or, for $j = 1$, resource R_n). However, this is impossible because process P_j must occupy the resources in ascending order with respect to the total order. This is a contradiction, and the assumption was therefore incorrect. $\square$

As mentioned at the beginning of this section, this method can also solve the bank account transfer problem, by having processes acquire accounts in ascending order of account number.

It can be shown that a deadlock-free system in which the allocation of each individual resource is starvation-free is also starvation-free overall. In particular: if a total order on resources exists, resources are acquired in that order, and individual resource allocation is starvation-free, then the overall system is starvation-free.

4.3 Deadlock Avoidance

As already mentioned, deadlock avoidance involves having an algorithm monitor resource allocation so that a potential deadlock situation is detected early and prevented in advance. Consider the so-called “deadly embrace” problem formulated by E.W. Dijkstra, which can be solved using the banker’s algorithm.

Assume there are m identical resources and each process requires a certain maximum number of resources during its execution. This maximum is assumed to be known in advance, but processes request the resources incrementally. It is also assumed that once a process has received all the resources it needs, it uses them and then releases them. The task is to insert an algorithm (a resource manager) that decides for each request whether it should be granted or whether the process must wait.

Dijkstra’s banker’s algorithm takes its name from an analogy: There is a banker with a fixed supply of money. Customers wish to borrow up to a specified maximum amount, requesting funds incrementally. Once a customer has received their maximum, they repay everything. The banker’s task is to lend money only when he can be certain of eventually recovering all funds.

For example, suppose the banker has EUR 98 in total and there are two customers, each needing a maximum of EUR 50 and each having already received EUR 48. If both customers now request EUR 1 each and the banker grants both requests, he is left with no funds, both customers have EUR 49, and neither can ever reach their maximum. A deadlock has occurred. An algorithm is needed to prevent such situations. An obvious approach would be to serve customers in a fixed total order: serve the first customer fully before serving the next. However, this enforces complete sequentialization—a parallel problem solved by a sequential algorithm, which contradicts the problem’s intent. Therefore, an additional requirement is that the algorithm should allow as much parallelism as possible.

More precisely, every request should be granted as long as it does not *necessarily* lead to a deadlock. Following Dijkstra’s approach, requests are admitted as long as they do not result in a situation where a process can only continue by preempting another process (hence the name “deadly embrace”).

The following section presents an extended variant of the banker’s algorithm that handles multiple different types of resources (in the banker analogy: different currencies). The resources currently available are denoted by the vector $\vec{A}$, where each component of $\vec{A}$ is a non-negative integer indicating the available amount of the corresponding resource.

Let $\mathcal{P}$ be the set of processes. For $P \in \mathcal{P}$, let:

- $\vec{M}_P$ be the vector of resources that can be requested by process P at most
- $\vec{C}_P$ be the vector of resources already allocated to process P

A state (with all its vectors) is *safe* if there is a permutation π of the processes $P_1, \dots, P_n \in \mathcal{P}$ such that for each process P_i according to the order of the permutation there are sufficient resources when it is its turn. Sufficient resources means that $\vec{A} + \left(\sum_{\pi(j) < \pi(i)} \vec{C}_{P_{\pi j}} \right) - \vec{M}_{P_i} + \vec{C}_{P_i} \geq \vec{0}$ holds. This means that the currently allocated resources may first be added to the currently available resources $\vec{A}$, whose associated processes are fully served according to the permutation before P_i . Subsequently, the maximum resources requested by process P_i must be covered, whereby the resources already allocated to P_i can be offset accordingly.

The banker’s algorithm receives a resource request $\vec{L}_P$ from a process $P \in \mathcal{P}$ as input. Based on the assumptions, it must hold that $\vec{L}_P + \vec{C}_P \leq \vec{M}_P$ (otherwise, the process would request more resources than it previously specified). The banker’s algorithm then proceeds as follows:

- If there are not enough resources available, it immediately rejects the request, i.e., process P must wait.
- Otherwise, it adjusts the current state to the request, i.e., it executes:

$$\begin{aligned} \vec{A} &:= \vec{A} - \vec{L}_P \\ \vec{C}_P &:= \vec{C}_P + \vec{L}_P \end{aligned}$$

- Now it checks whether the state obtained in this way is safe (see below).
- If the state is safe, P receives the requested resources; otherwise, the request is rejected and P must wait, and the previous state is restored.

When a process reaches its maximum and returns the resources, the vector $\vec{A}$ is (automatically) adjusted.

The algorithm remains to be specified. The algorithm is shown in Figure 4.1.

```

function testState( $\mathcal{P}$ ,  $\vec{A}$ ):
  if  $\mathcal{P} = \emptyset$  then
    return "safe"
  else
    if  $\exists P \in \mathcal{P}$  with  $\vec{M}_P - \vec{C}_P \leq \vec{A}$  then
       $\vec{A} := \vec{A} + \vec{C}_P$ ;
       $\mathcal{P} := \mathcal{P} \setminus \{P\}$ ;
      testState( $\mathcal{P}$ ,  $\vec{A}$ )
    else
      return "unsafe"

```

Figure 4.1: Banker's algorithm: Is a state safe?

The algorithm always selects a process from the set whose remaining resource requirements can be fully satisfied. Assuming this process will subsequently return its resources, the algorithm places it in the permutation (removing it from $\mathcal{P}$) and updates $\vec{A}$. Once all processes have been placed in the sequence ($\mathcal{P} = \emptyset$), the state is safe. If some processes cannot be placed, the state is unsafe and a deadlock is possible.

The worst-case runtime of the algorithm is $O(|\mathcal{P}|^2)$: there are at most $|\mathcal{P}|$ recursive calls, and in each call, finding an eligible process (the $\exists$ quantifier) requires examining all remaining processes.

It is noteworthy that $O(|\mathcal{P}|^2)$ suffices even though there are $|\mathcal{P}|!$ possible permutations. The reason is that if a process can be served with currently available resources, it does not matter in which position in the permutation it appears.

One minor disadvantage of this approach is that it does not optimize concurrency, i.e., it does not consider the quality (in terms of parallelism) of the chosen execution order. Another disadvantage is that the maximum values $\vec{M}_P$ must be known from the outset, which is often not the case in practice.

Finally, let us consider an example with four resources EUR, USD, JYN, SFR and four processes A, B, C, D. The current state is described by the following vector assignments:

Maximum values	Values obtained	Available resources
$\vec{M}_A = (4, 7, 1, 1)$	$\vec{C}_A = (1, 1, 0, 0)$	$\vec{A} = (2, 2, 3, 3)$
$\vec{M}_B = (0, 8, 1, 5)$	$\vec{C}_B = (0, 5, 0, 3)$	
$\vec{M}_C = (2, 2, 4, 2)$	$\vec{C}_C = (0, 2, 1, 0)$	
$\vec{M}_D = (2, 0, 0, 2)$	$\vec{C}_D = (1, 0, 0, 1)$	

First, we check whether this state is really safe: Start with $\mathcal{P} = \{A, B, C, D\}$

- First call of testState with $\mathcal{P} = \{A, B, C, D\}$ and $\vec{A} = (2, 2, 3, 3)$
 - Since $\mathcal{P} \neq \emptyset$, check whether there is a process $P \in \mathcal{P}$ that can still be executed:
 - A cannot be executed because $\vec{M}_A - \vec{C}_A = (4, 7, 1, 1) - (1, 1, 0, 0) = (3, 6, 1, 1) \not\leq (2, 2, 3, 3) = \vec{A}$
 - B cannot be executed because $\vec{M}_B - \vec{C}_B = (0, 8, 1, 5) - (0, 5, 0, 3) = (0, 3, 1, 2) \not\leq (2, 2, 3, 3) = \vec{A}$
 - C can be executed because $\vec{M}_C - \vec{C}_C = (2, 2, 4, 2) - (0, 2, 1, 0) = (2, 0, 3, 2) \leq (2, 2, 3, 3) = \vec{A}$ so adjust $\vec{A}$ (as if C receives all resources and then returns them):

$$\begin{aligned}\vec{A} &:= \vec{A} + \vec{C}_C = (2, 2, 3, 3) + (0, 2, 1, 0) = (2, 4, 4, 3) \\ \mathcal{P} &:= \mathcal{P} \setminus \{C\} = \{A, B, D\}\end{aligned}$$

- Call testState recursively
- Call testState with $\mathcal{P} = \{A, B, D\}$ and $\vec{A} = (2, 4, 4, 3)$
 - Since $\mathcal{P} \neq \emptyset$, check whether there is a process $P \in \mathcal{P}$ that can still be executed:
 - A cannot be executed because $\vec{M}_A - \vec{C}_A = (4, 7, 1, 1) - (1, 1, 0, 0) = (3, 6, 1, 1) \not\leq (2, 4, 4, 3) = \vec{A}$
 - B can be executed, because $\vec{M}_B - \vec{C}_B = (0, 8, 1, 5) - (0, 5, 0, 3) = (0, 3, 1, 2) \leq (2, 4, 4, 3) = \vec{A}$ so adjust $\vec{A}$ (as if B receives all resources and then returns them):

$$\begin{aligned}\vec{A} &:= \vec{A} + \vec{C}_B = (2, 4, 4, 3) + (0, 5, 0, 3) = (2, 9, 4, 6) \\ \mathcal{P} &:= \mathcal{P} \setminus \{B\} = \{A, D\}\end{aligned}$$

- Call testState recursively
- Call testState with $\mathcal{P} = \{A, D\}$ and $\vec{A} = (2, 9, 4, 6)$
 - Since $\mathcal{P} \neq \emptyset$, check whether there is a process $P \in \mathcal{P}$ that can still be executed:

- A can be executed because $\vec{M}_A - \vec{C}_A = (4, 7, 1, 1) - (1, 1, 0, 0) = (3, 6, 1, 1) \leq (2, 9, 4, 6) = \vec{A}$ so adjust $\vec{A}$ (as if A receives all resources and then returns):

$$\begin{aligned}\vec{A} &:= \vec{A} + \vec{C}_A = (2, 9, 4, 6) + (1, 1, 0, 0) = (3, 10, 4, 6) \\ \mathcal{P} &:= \mathcal{P} \setminus \{A\} = \{D\}\end{aligned}$$

- Call testState recursively
- Call testState with $\mathcal{P} = \{D\}$ and $\vec{A} = (3, 10, 4, 6)$
 - Since $\mathcal{P} \neq \emptyset$, check whether there is a process $P \in \mathcal{P}$ that can still be executed:
 - D can be executed because $\vec{M}_D - \vec{C}_D = (2, 0, 0, 2) - (1, 0, 0, 1) = (1, 0, 0, 1) \leq (3, 10, 4, 6) = \vec{A}$ so adjust $\vec{A}$ (as if D receives all resources and then returns):

$$\begin{aligned}\vec{A} &:= \vec{A} + \vec{C}_D = (3, 10, 4, 6) + (1, 0, 0, 1) = (4, 10, 4, 7) \\ \mathcal{P} &:= \mathcal{P} \setminus \{D\} = \emptyset\end{aligned}$$

- Call testState recursively
- Call testState with $\mathcal{P} = \emptyset$ and $\vec{A} = (4, 10, 4, 7)$
 - Returns safe because $\mathcal{P} = \emptyset$.

The state is therefore safe.

Now the request $L_A = (2, 2, 0, 0)$ is processed, i.e., process A wants to allocate two more EUR and two more USD.

After updating ($\vec{A} := \vec{A} - \vec{L}_A$ and $\vec{C}_A := \vec{C}_A + \vec{L}_A$) the state is obtained:

Maximum values	Values obtained	Available resources
$\vec{M}_A = (4, 7, 1, 1)$	$\vec{C}_A = (3, 3, 0, 0)$	$\vec{A} = (0, 0, 3, 3)$
$\vec{M}_B = (0, 8, 1, 5)$	$\vec{C}_B = (0, 5, 0, 3)$	
$\vec{M}_C = (2, 2, 4, 2)$	$\vec{C}_C = (0, 2, 1, 0)$	
$\vec{M}_D = (2, 0, 0, 2)$	$\vec{C}_D = (1, 0, 0, 1)$	

and it must be checked whether this state is still safe.

- Call testState with $\mathcal{P} = \{A, B, C, D\}$ and $\vec{A} = (0, 0, 3, 3)$
 - Since $\mathcal{P} \neq \emptyset$, check whether there is a process $P \in \mathcal{P}$ that can still be executed:
 - A cannot be executed because $\vec{M}_A - \vec{C}_A = (4, 7, 1, 1) - (3, 3, 0, 0) = (1, 4, 1, 1) \not\leq (0, 0, 3, 3) = \vec{A}$
 - B cannot be executed because $\vec{M}_B - \vec{C}_B = (0, 8, 1, 5) - (0, 5, 0, 3) = (0, 3, 1, 2) \not\leq (0, 0, 3, 3) = \vec{A}$

- C cannot be executed because $\vec{M}_C - \vec{C}_C = (2, 2, 4, 2) - (0, 2, 1, 0) = (2, 0, 3, 0) \not\leq (0, 0, 3, 3) = \vec{A}$
- D cannot be executed because $\vec{M}_D - \vec{C}_D = (2, 0, 0, 2) - (1, 0, 0, 1) = (1, 0, 0, 1) \not\leq (0, 0, 3, 3) = \vec{A}$
- This means that there is no process in $\mathcal{P}$ with $\vec{M}_P - \vec{C}_P \leq \vec{A}$ and the algorithm returns unsafe.

The request should therefore be rejected.

4.4 Transactional Memory

This section introduces a relatively recent programming approach that lets programmers write essentially sequential code while the system ensures freedom from deadlocks and other concurrency errors.

First, it is worth asking why existing mechanisms and programming abstractions are insufficient.

Consider again the bank account transfer problem. One solution requires all transfer processes to lock accounts in a fixed total order, execute the transfer, and release the locks. Suppose we extend this so that a debit is only executed if the account has sufficient funds. One option is to abort and retry the transaction if the balance is too low—but this leads to busy-waiting and, without starvation-freedom guarantees, may loop indefinitely. Another option is to wait on a condition variable, but care must be taken not to hold the account lock while waiting, since other transactions must be able to proceed. Even in this small example, a correct solution requires careful coordination of multiple locks and condition variables.

S.L. Peyton Jones cites (Peyton-Jones, 2007, Chapter 24) several reasons why lock-based programming is bad:

- Setting too few locks: It can easily happen that the programmer forgets to set a lock, resulting in memory space being modified unprotected by two processes.
- Setting too many locks: If too many locks are set because the programmer is overly cautious (or does not understand the problem well enough), there may (in the best case) be less concurrency than would otherwise be possible, i.e., the programmer forces unnecessary sequentialization. In the worst case, the programmer creates a deadlock.
- Setting the wrong locks: In lock-based programming approaches, there is usually no clear assignment between locks and the data they are supposed to protect. Consider, for example, condition variables: Here, the programmer should choose a name for the variable that corresponds to the condition, but he does not have to. This approach means that the (first) programmer knows why he is setting which locks, but neither the compiler nor other programmers know this. Subsequent

maintenance work on the code can easily lead to incorrect locks being set or necessary locks not being set.

- Setting locks in the wrong order: The total order theorem can only be adhered to if every programmer knows what the order of the locks is and sets them accordingly.
- Error handling can be difficult, as it must always be taken into account that an error does not result in locks remaining set forever. For example, when intercepting errors, care must be taken to ensure that set locks are removed and a consistent state of the data is established.
- Forgotten **signal** operations or forgotten rechecking of conditions leads to faulty systems. It is easy to make such programming errors.

The main counterargument against lock-based programming is that it is not modular: in general, it is not possible to assemble a large, correct program from small, correct components. Consider account transfers: suppose a new form of transfer is introduced that debits an amount from account A1 or A2 to account B, depending on which has sufficient funds. In this case, waiting on a condition variable for account A1 no longer works; the entire program must be restructured.

The idea of transactional memory is based on the observation that similar problems arise in concurrent programming when accessing databases. However, in the database domain, parallel access has not been a real problem for user programmers for many years, as the database manager ensures consistent access. The basic idea of transactional memory is to treat shared memory like a database: accesses to memory are handled like database transactions, and a manager ensures that the memory state remains consistent. From a programming perspective, this is attractive because the programmer can write operations as if programming sequentially, without setting locks, and instead delegates the operation to the transaction manager, which either executes the transaction, aborts and restarts it, or aborts and reports failure.

The semantics of database transactions can be described as follows: each transaction is executed as if it were the only action taking place on the database. The database manager guarantees that the transaction's externally observable behavior conforms to this specification.

For the discussion of transactional memory (i.e., transactions on memory rather than on databases), database transactions are first considered, and it is explained how they transfer to the memory model. Each database transaction must satisfy four properties, the so-called *ACID properties*:

- Atomicity: All operations within a transaction are either all performed or none is performed. A failed transaction leaves no observable change in the database; a successful transaction commits, a failed transaction aborts.
- Consistency: A committed transaction changes the database state consistently. An aborted transaction leaves the state unchanged and thus consistent. The consistency of a committed transaction is defined by the application (e.g., a bank

account balance cannot be arbitrarily negative, or a newly added account must have a unique number).

- Isolation: A transaction produces a correct result regardless of the number of other concurrent transactions.
- Durability: The result of a committed transaction is permanent, i.e., stored persistently on disk or similar media.

The first three properties are also required for memory transactions. Durability cannot generally be guaranteed for memory transactions (and is therefore not required), since main memory is volatile.

However, there are some differences between accessing databases and the state of memory:

- Data in a database is usually stored on hard disks. Since hard disk access takes a few milliseconds, which on the other hand allows millions of CPU operations, database programming allows computing time and data access to be well balanced. Since transactional memory operates on much faster main memory, not many computing operations can be performed during memory access, i.e., balancing is no longer so easy. As already mentioned, transactional memory is not permanent, since the data in the main memory is gone after the program terminates. This generally makes the implementation of transactional memory easier than with databases.
- Transactional memory must be integrated into existing programming languages and must not represent a completely new language in order for it to be used and accepted by programmers.

4.4.1 Basic Primitives for Transactional Memory

This section describes some operations that can be considered the basis for transactional memory.

4.4.1.1 Atomic Blocks

The keyword **atomic** can be used to mark code blocks as transactions that are to fulfill the aforementioned ACI properties.

```
atomic {  
    Transaction code  
}
```

The exact implementation is ignored for now. The advantage over monitors is that the variables to be modified do not have to be specified explicitly; all variables are automatically protected and none can be accidentally omitted.

Nevertheless, it is still possible to write incorrect programs with atomic blocks, especially if the code inside a transaction does not terminate. This is not further restricted (as doing so would sacrifice Turing completeness), but the semantics are adjusted accordingly: the execution of a transaction (atomic block) has three possible outcomes:

- commit, if the transaction was successful,
- abort, if the transaction was aborted
- undefined, if the transaction did not terminate

Note that transactions in the transactional memory model can be aborted in two ways: by the system (the transaction manager), because a conflict occurs, or by the programmer (who defines an abort option within the transaction). If the system aborts the transaction, the transaction manager will attempt to execute the transaction again, unless the programmer has specified what should happen in this case.

Unlike database transactions, however, there is another problem: The variables that are changed within a transaction can also be accessed outside of transactions, which can cause unwanted behavior that violates the ACI conditions. Due to these differences, the semantics of **atomic** are often specified in such a way that the execution of such a code block is as if all **atomic** blocks were protected by a global lock. This does not necessarily mean that the implementation does this; it only specifies the behavior. If variables are accessed outside of transactions, this semantics allows for undesirable results. It is the programmer's responsibility to mark all transactions as such.

4.4.1.2 The abort command

The **abort** operation can be used to abort a transaction. The original state is restored.

4.4.1.3 The retry command

Most transactional memory systems have a **retry** command, which allows transactions to be coordinated. When **retry** is executed, the transaction is aborted (with consistent state restoration!) and an attempt is made to execute the transaction again from the beginning. This allows behavior similar to that of condition variables to be generated. Consider, for example, the programming of a bounded buffer in the producer/consumer problem. Here, it must be ensured that the consumer only accesses the buffer if it is not empty. This can be programmed with **retry** as follows:

```
atomic{  
    if isEmpty(buffer) then retry;  
    Read first element of buffer, etc.  
}
```

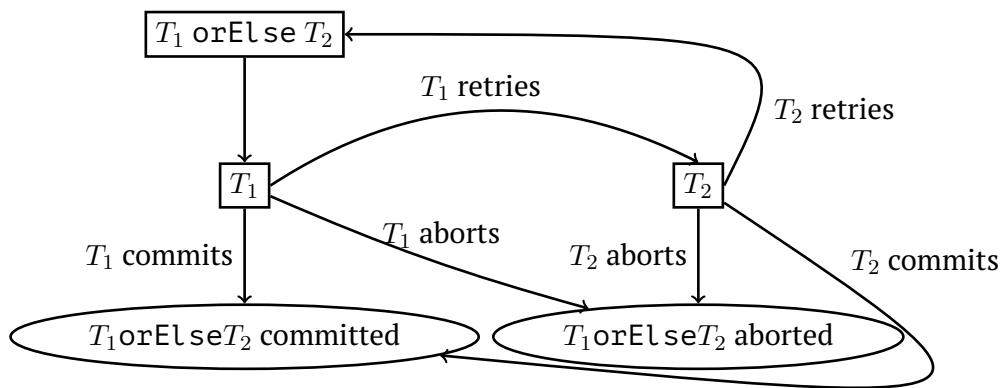
If the transaction is executed, it is aborted and restarted if the buffer is empty.

4.4.2 The orElse Command

The **orElse** statement can be used to specify an alternative if the transaction is aborted. It combines two transactions. Let T_1 and T_2 be transactions, then $T_1 \text{orElse} T_2$ is the transaction that:

- first executes T_1 . If T_1 is committed or T_1 is explicitly aborted via command, then T_2 is *not* executed. The transaction $T_1 \text{orElse} T_2$ is committed or aborted, depending on what the execution of T_1 does.
- If T_1 is aborted by the transaction manager or **retry** is executed, then the transaction manager does not restart T_1 , but attempts to execute T_2 .
- If T_2 was explicitly aborted or successfully committed, then the entire transaction is complete.
- If T_2 performs a **retry** (or is aborted by the system), then $T_1 \text{orElse} T_2$ is restarted.

In summary, the semantics of **orElse** can be represented by the following diagram:



With the **orElse** command, it is easy to implement debiting account A1 or account A2 to account B:

```

atomic {
  {
    B := B + amount;
    A1 := A1 - amount;
    if A1 ≤ 0 then retry;
  }
  orElse
  {
    B := B + amount;
    A2 := A2 - amount;
    if A2 ≤ 0 then retry;
  }
}
  
```

4.4.3 Properties of TM Systems

This section presents various properties that can be used to classify and describe existing transactional memory systems. There are some software TM systems that are implemented within different programming languages. However, there are now also hardware TM systems, i.e., TM instructions are supported at the hardware level.

4.4.3.1 Weak / Strong Isolation

Weak isolation refers to systems in which memory locations are only protected when access is via transactions. Operations outside of transactions can violate the isolation condition and cause inconsistencies. Systems that support strong isolation automatically ensure that variables protected by transactions always remain protected, e.g., by having **atomic** blocks automatically inserted by the compiler.

4.4.3.2 Handling of nested transactions

TM systems differ in their handling of nested **atomic** blocks. Different behaviors can be specified for what should happen if an inner transaction fails. There are three different approaches: *smoothed* transactions, *closed* transactions, and *open* transactions.

If transactions are *smoothed*, this means that aborting an inner transaction leads to the abortion of the entire transaction and, conversely, that a committed inner transaction only becomes visible to the outside world when the outer transaction is also successful.

For example, in the following transaction, the variable x will retain the value 1:

```
x := 1
atomic {
  x:=2;
  atomic {
    x:= 3;
    abort;
  }
}
```

In a closed transaction, an internal aborting transaction does not cause the outer transaction to abort. For the above program, x will retain the value 2. In the case of a successful transaction, the program behaves in the same way as with smooth transactions. In particular, changes made by inner transactions are only visible to the outside world once the outer transaction has been committed.

In the case of open transactions, successful internal transactions become visible externally. Even if the external transaction is aborted, the changes made by the internal transaction are retained. In the following example, x would have the value 3 even though the external transaction is aborted:

```
x := 1
atomic {
  x:=2;
  atomic {
    x:= 3;
  }
  abort;
}
```

4.4.3.3 Granularity

TM systems also differ in terms of the granularity at which conflicts are detected. A distinction is made between *object granularity*, i.e., conflicts due to changes to objects are observed, and *word or block granularity*, whereby changes relate directly to memory words. Note that object granularity is coarser: as soon as two transactions modify the same object, a conflict is detected and one of the transactions is aborted. This also happens if the transactions modify different attributes of the object.

4.4.3.4 Direct and delayed update

If a TM system uses *direct update*, then executing transactions modify the objects immediately, and the system ensures that concurrent processes do not do this at the same time. The old values (before the update) are stored by the system and written back in the event of a termination. With a delayed update, the opposite is true: only local copies are modified, and the original is only overwritten by the local copy when the commit is made.

4.4.3.5 Time of conflict detection

TM systems differ in when conflicts are detected. They can detect conflicts as soon as an object is first accessed, in between, when they have changed some objects, or only at the very end, when the transaction is to be committed.

4.4.3.6 Conflict management

When conflicts occur, transactions must be aborted. However, it should be ensured that the entire system takes a step forward (i.e., not all transactions are aborted). There are numerous strategies for determining which transactions are executed.

4.4.4 Correctness Criteria for STM Systems

Several research papers have investigated and continue to investigate the requirements and properties that a TM system must have in order to be considered correct. A fairly comprehensive overview of 16 different terms can be found in (Dziura et al., 2015).

Here, we explain a selection of correctness criteria, following the presentation in (Guerraoui & Kapalka, 2008) with adaptations from (Dziurma et al., 2015).

For a formal modeling of STM systems, transaction execution can be viewed as step-by-step processing, where a step corresponds to a call to an operation on a concurrent object (such as an atomic register, compare-and-swap object, etc.) and additional local calculations. We limit our discussion to atomic registers. To further abstract, we use histories of transaction execution:

Definition 4.4.1 (History and events). *A history H is a sequence of events, where an event can be:*

- Transaction T_i performs an operation call. This can be an operation on an atomic register: $T_i.\text{read}(x)$ (read the transactional variable x), $T_i.\text{write}(x, v)$ (writing the transactional variable x with the value v), or $T_i.\text{commit}$ (T_i wants to commit) and $T_i.\text{abort}$ (T_i wants to abort).
- Transaction T_i receives a return value for one of the four operations read, write, commit, abort. For read, this is the event $T_i.w$, where w is the value; for write, this is the event $T_i.\text{Ok}$. However, the return value can always be the special value A_{T_i} (written as event $T_i.A_{T_i}$), which indicates that transaction T was aborted. The return value of abort is always A_{T_i} , while the return value of commit is Ok or A_{T_i} .

Consider, for example, a TM system where the values of x, y, z are 1 at the beginning and the two transactions:

$$T_1 : x := y + 1 \qquad T_2 : y := z + 1$$

Events (which are also calls) of transaction T_1 are then $T_1.\text{read}(y)$, $T_1.\text{write}(x)$ and $T_1.\text{commit}$. Returns are events such as $T_1.1$ (the last call of T_1 returns 1) or $T_1.\text{Ok}$ (the last call of T_1 was completed successfully). $T_1.A_{T_1}$ is also possible if transaction T_1 was aborted.

A possible history H_1 for the above scenario is:

$T_1.\text{read}(y)$
 $T_2.\text{read}(z)$
 $T_2.1$
 $T_2.\text{write}(y, 2)$
 $T_1.1$
 $T_2.\text{Ok}$
 $T_1.\text{write}(x, 2)$
 $T_1.\text{Ok}$
 $T_1.\text{commit}$
 $T_1.\text{Ok}$
 $T_2.\text{commit}$
 $T_2.A_{T_2}$

Histories can also be represented along the time axis as intervals, with calls and returns being the beginning and end of the intervals. Figure 4.2 shows a corresponding representation.

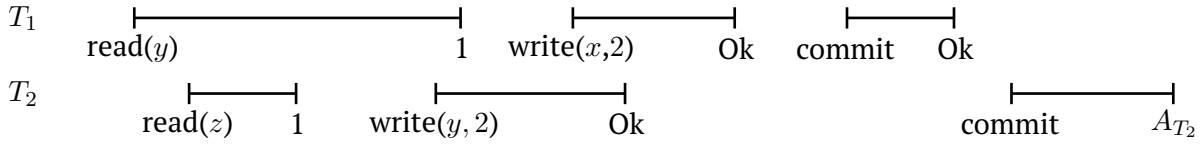


Figure 4.2: Interval representation of a history

We introduce further notation and terminology for histories before using them to define the various correctness criteria.

Definition 4.4.2 (Restrictions of a history). *For a history H and a transaction T , we write $H|T$ for the restriction of H to T : This is obtained from the history H by deleting all events that do not belong to T . Similarly, we write $H|\{T_1, \dots, T_n\}$ for the history H after deleting all events that do not belong to the transactions $T_1, \dots, T_n$. For a transactional variable x , let $H|x$ denote the history H after deleting all events that do not belong to x .*

For example, $H_0|T_2$ (with H_0 as defined above) is:

$T_2.\text{read}(z)$
 $T_2.1$
 $T_2.\text{write}(y,2)$
 $T_2.\text{Ok}$
 $T_2.\text{commit}$
 $T_2.A_{T_2}$

Since H_0 only contains transactions T_1 and T_2 , the following applies $H_0|\{T_1, T_2\} = H_0$. As a further example, we show the history $H_0|y$:

$T_1.\text{read}(y)$
 $T_2.\text{write}(y,2)$
 $T_1.1$
 $T_2.\text{Ok}$

Next, we define the well-formedness of histories. This requires that the history per transaction contain events in a valid order. It does not require that every transaction has been completed.

Definition 4.4.3. *A history H is well-formed if the following applies to every transaction T : $H|T$ consists alternately of calls and returns, $H|T$ does not contain any events after the event $T.A_T$, if $H|T$ contains the event $T.\text{commit}$, then either no further events follow or the event $T.A_T$ or the event $T.\text{Ok}$ follows, and if $H|T$ contains the event $T.\text{abort}$, then either no more events follow or only the event $T.A_T$ follows.*

In what follows, we will only consider well-formed histories. TM systems that do not provide well-formed histories are therefore considered invalid (incorrect).

Based on the history, we can classify transactions as successfully completed (committed), aborted, or ongoing:

Definition 4.4.4. A transaction T is committed in a history H if $H|T$ ends with $T.commit$, $T.Ok$. It is aborted if $H|T$ ends with $T.A_T$. A transaction is complete in a history H if it is committed or aborted in H . We call a non-complete transaction ongoing in H .

We write $comm(H)$ for $H|\{T_1, \dots, T_n\}$, where $T_1, \dots, T_n$ are all transactions in H that are committed, i.e., $comm(H)$ is H after removing all aborted and running transactions.

Definition 4.4.5 (Sequential History). A history is sequential if all events of a transaction are in a subsequence.

Example 4.4.6. The history H_0 is not sequential. The history H_1 , defined as:

$T_1.read(y)$
 $T_1.1$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$
 $T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_2.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

is sequential.

Definition 4.4.7 (Equivalence of histories). Two histories H, H' are equivalent (written as $H \sim H'$), if they use the same transactions and for every transaction T $H|T = H'|T$ (the histories per transaction are identical).

Example 4.4.8. Consider the following histories

History H_1 :	History H_2 :	History H_3 :	History H_4 :
$T_1.read(x)$	$T_1.read(x)$	$T_1.read(x)$	$T_2.read(x)$
$T_2.read(x)$	$T_2.read(x)$	$T_1.v$	$T_2.v$
$T_1.v$	$T_2.v$	$T_1.write(x, v')$	$T_2.read(y)$
$T_2.v$	$T_1.v$	$T_1.Ok$	$T_2.v''$
$T_2.read(y)$	$T_1.write(x, v')$	$T_1.commit$	$T_2.commit$
$T_1.write(x, v')$	$T_1.Ok$	$T_1.Ok$	$T_2.Ok$
$T_1.Ok$	$T_1.commit$	$T_2.read(x)$	$T_1.read(x)$
$T_1.commit$	$T_2.read(y)$	$T_2.v'$	$T_1.v$
$T_2.v''$	$T_1.Ok$	$T_2.read(y)$	$T_1.write(x, v')$
$T_1.Ok$	$T_2.v''$	$T_2.v''$	$T_1.Ok$
$T_2.commit$	$T_2.commit$	$T_2.commit$	$T_1.commit$
$T_2.Ok$	$T_2.Ok$	$T_2.Ok$	$T_1.Ok$

Then histories H_3 and H_4 are sequential, histories H_1 and H_2 are equivalent, H_1 and H_4 are equivalent, but H_1 and H_3 are not equivalent.

A history implies a partial order of transactions: If the time of completion (with A_T or Ok) of transaction T_i precedes the first event of transaction T_j , then transaction T_i genuinely precedes transaction T_j :

Definition 4.4.9 (Real-time order $\prec_H$). For a history H , the real-time order of transactions $\prec_H$ is the partial order defined by: If T_i is complete in H and the first event of T_j occurs after the last event of T_i in H , then $T_i \prec_H T_j$.

For sequential histories H whose transactions are all complete in H , the real-time order is always a total order.

Definition 4.4.10 (Legal sequential histories of committed transactions). A sequential history H , where all transactions are committed except possibly the last transaction, is legal if the following applies to all transactional variables: $H|x$ corresponds to the sequential description of atomic registers (the read and write operations and their returns behave as if they had been executed in a sequential program).

Example 4.4.11. The following history H_5 is not legal because $H_5|x$ does not correspond to the sequential description of atomic registers. The last read operation should have returned 4

<i>History H_5:</i>	<i>History $H_5 x$:</i>	<i>History $H_5 y$:</i>
$T_1.read(y)$	$T_1.write(x,4)$	$T_1.read(y)$
$T_1.3$	$T_1.Ok$	$T_1.3$
$T_1.write(x,4)$	$T_1.read(x)$	$T_2.read(y)$
$T_1.Ok$	$T_1.4$	$T_2.3$
$T_1.read(x)$	$T_2.read(x)$	
$T_1.4$	$T_2.2$	
$T_1.commit$		
$T_1.Ok$		
$T_2.read(x)$		
$T_2.2$		
$T_2.read(y)$		
$T_2.3$		

For a sequential history H (whose transactions are all completely in H), the above concept of legality cannot be used directly, since after aborted transactions, the read and write operations may behave differently than in a sequential program. In order to define a concept of legality for such histories, all transactions are considered individually and, for each transaction T , only the events of those transactions T' in the history that are committed before the start of T are considered:

Definition 4.4.12 (Legal complete, sequential histories). *A sequential history H , whose transactions are all complete in H , is legal if for every transaction T_i in H , the history H_i is legal, where H_i is created from H by transferring exactly those events $T_j.e$ from H to H_i with*

- $j = i$, or
- T_j is committed in H and $T_j \prec T_i$ in H .

Note that the history H_i satisfies the requirements of the previously defined concept of legality: All transactions except possibly the transaction T_i are committed in H_i .

Example 4.4.13. *Consider the history H_6 and the histories $H_{6,1}$, $H_{6,2}$ and $H_{6,3}$:*

<i>History H_6</i>	<i>History $H_{6,1}$</i>	<i>History $H_{6,2}$</i>	<i>History $H_{6,3}$</i>
$T_1.write(x, 3)$	$T_1.write(x, 3)$	$T_3.read(x)$	$T_3.read(x)$
$T_1.Ok$	$T_1.Ok$	$T_3.4$	$T_3.4$
$T_1.read(x)$	$T_1.read(x)$	$T_3.write(y, 2)$	$T_3.write(y, 2)$
$T_1.3$	$T_1.3$	$T_3.Ok$	$T_3.Ok$
$T_1.abort$	$T_1.abort$	$T_3.commit$	$T_3.commit$
$T_1.A_{T_1}$	$T_1.A_{T_1}f$	$T_3.Ok$	$T_3.Ok$
$T_3.read(x)$		$T_2.read(y)$	
$T_3.4$		$T_2.2$	
$T_3.write(y, 2)$		$T_2.read(x)$	
$T_3.Ok$		$T_2.4$	
$T_3.commit$			
$T_3.Ok$			
$T_2.read(y)$			
$T_2.2$			
$T_2.read(x)$			
$T_2.4$			

Since $H_{6,1}$, $H_{6,2}$ and $H_{6,3}$ are legal sequential histories of committed transactions, H_6 is a legal complete sequential history.

The most important properties of a correct STM system are:

- All transactions that are committed appear as if they were performed indivisibly in a single step.
- Aborted transactions appear as if they were never performed.

Two other aspects that are usually required are:

Preservation of real-time order: The point in time at which the effects of a transaction appear is somewhere within the runtime of the transaction. This means that the transaction itself should not read outdated memory states. This effect can actually occur in incorrect implementations, for example, if old values of transactional variables are still stored due to the use of caches. This should be avoided. Rather, if a transaction T_1 modifies and commits an object x , and then a transaction T_2 starts and reads x , then T_2 should read the value written by T_1 and not an older value.

Formally, the following is defined for histories: A history H' *preserves the real-time order* of a history H if $\prec_H \subseteq \prec_{H'}$.

Exclusion of inconsistent views: One could argue that running transactions that are neither committed nor aborted should also be allowed to read inconsistent memory states, since they can simply be aborted later anyway. However, this can lead to various problems, since transactions do not run in a completely isolated environment: Let x and y be two (transactional) variables that must always satisfy the

invariant $y = x^2$ and $x \geq 2$. The programmer satisfies the invariant by ensuring that all transactions satisfy it. Let $x = 4$ and $y = 16$ and now consider the transactions

$$T_1 : \quad x := 2; \quad T_2 \quad z := 1/(y - x) \\ y := 4;$$

Assume that T_2 first reads x , then runs through T_1 , and then continues calculating T_2 and is allowed to see the updated values. Written as history

```

T2.read(x)
T2.4
T1.write(x, 2)
T1.Ok
T1.write(y, 4)
T1.Ok
T1.commit
T1.Ok
T2.read(y)
T2.4
...
```

In this case, T_2 crashes with an error (division by 0) because inconsistent values were read. Therefore, T_2 should either receive $T_2.16$ as the return value or $T_2.A_{T_2}$ to abort immediately.

We will now explain the correctness criteria:

Definition 4.4.14 (Linearizability). *An STM system satisfies the property linearizability if the following applies: Each committed transaction acts like an atomic function call on the transactional variables, which is executed in one step. The successful transactions act like a sequence of such steps.*

Closely related is the concept of sequentializability:

Definition 4.4.15 (Sequentializability). *An STM system is sequentializable if the following applies to every history H of the TM system: There is a sequential and legal history S with $S \sim \text{comm}(H)$. This means that the history of successful transactions is equivalent to the history of a sequential sequence of transactions.*

Strict sequentializability additionally requires that the sequential history sought preserves the real-time order of the given history:

Definition 4.4.16 (Strict sequentializability). *An STM system is strictly sequentially orderable if the following applies to every history H of the TM system: There is a sequential and legal history S with $S \sim \text{comm}(H)$ and $\prec_{\text{comm}(H)} \subseteq \prec_S$.*

So far, we have assumed (as does the sequentializability in the literature) that there are only read and write operations. The approach of so-called “global atomicity” also

allows other concurrent objects and copies of these objects. The approach of (Guerraoui & Kapalka, 2008) also allows arbitrary concurrent objects with sequential description. We will not extend our formalism at this point and refer to the literature.

One criticism of the criteria used so far is that they say almost nothing about aborted transactions or transactions that are still running.

Recoverability requires that if a transaction T_i writes to a transactional variable, then no other transaction T_j may read the same transactional variable before T_i has been successfully completed or aborted. On the one hand, this requirement is too strong, as it sometimes forces too much sequentialization of transaction execution, and on the other hand, it is too weak, as it (even in combination with global atomicity) does not prohibit aborting transactions from reading inconsistent states (see (Guerraoui & Kapalka, 2008)).

In (Guerraoui & Kapalka, 2008), the so-called opacity is therefore proposed as a correctness criterion: This requires an equivalent sequential and legal history to the given history, which is completed so that ongoing transactions are aborted and transactions that have executed a commit but are not yet committed are either committed or aborted. The completion is not unique; it is sufficient if a completion has the required properties.

Definition 4.4.17. *A TM system satisfies opacity if every history of the TM system is opaque:*

A history H is opaque if there is a complete, sequential, legal history S that is equivalent to a history in $\text{complete}(H)$ such that S respects the real-time order of H .

Here, $\text{complete}(H)$ provides a set of histories H' such that all transactions occurring in H are complete in H' , where

- *H' is well-formed,*
- *H' contains only additional commit and abort events compared to H , and only for transactions that are ongoing in H .*
- *running transactions T from H for which $T.\text{commit}$ occurs in H are committed or aborted in H'*
- *running transactions T from H for which no $T.\text{commit}$ occurs in H are aborted in H' .*

This concept of correctness implies that the operations of each successful transaction are performed atomically in a single step. In particular, this means that if all steps of aborted or ongoing transactions are removed, the (concurrent) execution of the successful transactions is equivalent to a sequential execution of these transactions. In addition, sequential execution must adhere to real-time ordering. The effects of aborted transactions are never visible to other transactions. Every transaction (whether successful or aborted) sees only consistent states, i.e., those that have been created by committed transactions.

Further terms and explanations for this concept of correctness can be found in the relevant literature. The TL2 algorithm presented in the next section satisfies opacity.

4.4.5 The TL2 Algorithm

This section introduces the so-called “transactional locking” algorithm by Dice, Shalev, and Shavit (Dice et al., 2006), following the presentation in (Guerraoui & Kapalka, 2010) for the most part. This algorithm implements the transaction manager and is often used in implementations of STM systems. The **retry** and **orElse** operations, as well as nested transactions, are not initially provided for in this algorithm.

The TL2 algorithm uses a global counter `gc`, which is incremented when committing transactions. This counter is implemented by a fetch-and-increment object.

The transactional variables $x_1, x_2, \dots$ (i.e., the real shared memory locations) are in an array `TVar`, and each memory location `TVar[i]` is assigned a compare-and-swap object `C[i]`, which contains a pair `(ver, lock)`, where

- The component `ver` is a timestamp (of the global counter) that makes a statement about the last write access to the memory location.
- The component `lock` is a Boolean value that indicates whether the memory location is considered locked.

The local data structures of each transaction are

- A set `readset`, in which the addresses of the memory locations read by the transaction are stored.
- A set `writeset`, in which the addresses written by the transaction are stored.
- Local memory locations `writelog[i]`, such that `writelog[i]` contains the value written by the transaction for `TVar[i]`. All write operations of the transactions are recorded in these local memory locations and are not initially executed in the real memory. The new contents are only written to the shared memory upon (successful) commit.
- A timestamp `readver`, which receives the value of the global counter during the transaction’s first read access.
- A set `lockset`, in which the addresses of the memory locations currently locked by the transaction are stored.

The operations of the TL2 algorithm for reading, writing a transactional variable, as well as for aborting and committing are shown in Figure 4.3. We explain the individual operations.

Executing a read operation for TVar x_m The first read operation sets `readver` to the current value of the global counter.

First, it is checked whether the address of the transactional variable is already in the `writeset` (i.e., has already been written to by the transaction). If this is the case, the new value from the `writelog` array is returned.

Common data structures:

gc (global counter), fetch-and-increment object, initial gc=0
 TVar[1..] array of atomic registers, initial TVar[i] = 0
 C[1..] array of compare-and-swap objects with pairs as content, initial C[i]=(0,false)

Local data structures:

readver local counter value, initial readver = $\perp$
 readset, writeset, lockset sets of addresses, initial all = $\emptyset$
 writelog[1..] array of values

Operation $T_k.read(x_m)$

```

if readver =  $\perp$  then readver := read(gc);
if  $m \in \text{writeset}$  then return writelog[m];
(ver1,locked1) := read(C[m]);
result := read(TVar[m]);
(ver2,locked2) := read(C[m]);
if ver1  $\neq$  ver2 or locked2 or ver2 > readver
then  $T_k.abort()$ ;
readset := readset  $\cup \{m\}$ ;
return result;

```

Operation $T_k.write(x_m, val)$

```

writeset := writeset  $\cup \{m\}$ ;
writelog[m] := val;
return Ok;

```

Operation $T_k.abort()$

```

for all  $m \in \text{lockset}$  do
  (ver,locked) := read(C[m]);
  write(C[m],(ver,False));
readset :=  $\emptyset$ ;
writeset :=  $\emptyset$ ;
lockset :=  $\emptyset$ ;
readver :=  $\perp$ ;
return  $A_{T_k}$ ;

```

Operation $T_k.commit()$

```

for all  $m \in \text{writeset}$  do
  (ver,locked) := read(C[m]);
  if locked then  $T_k.abort()$ ;
  lock := compare-and-swap(C[m],(ver,False),(ver,True));
  if not lock then  $T_k.abort()$ ;
  lockset := lockset  $\cup \{m\}$ ;
writever := 1+fetch-and-increment(gc);
if writever  $\neq$  readver+1 then
  for all  $m \in \text{readset}$  do
    (ver,locked) := read(C[m]);
    if ver > readver
      or (locked and  $m \notin \text{writeset}$ )
      then  $T_k.abort()$ ;
  for all  $m \in \text{writeset}$  do
    write(TVar[m],writelog[m]);
    write(C[m],(writever,False))
readset :=  $\emptyset$ ;
writeset :=  $\emptyset$ ;
lockset :=  $\emptyset$ ;
readver :=  $\perp$ ;
return Ok;

```

Figure 4.3: The TL2 algorithm

If m is not in the writeset, the pair $(\text{ver}_1, \text{locked}_1)$ is first read from the associated compare-and-swap object, then the content of x_m from $\text{TVar}[m]$, and finally the compare-and-swap object is read again (this results in $(\text{ver}_2, \text{locked}_2)$)

In the following cases, the transaction is completely aborted and restarted because a conflict has been detected:

- The memory location is locked because locked_2 is set to `True`.
- $\text{ver}_2 > \text{readver}$, because then x_m was changed after the transaction started.
- $\text{ver}_1 \neq \text{ver}_2$, because then reading the value of $\text{TVar}[m]$ was not atomic, i.e., another transaction changed $\text{TVar}[m]$ in the meantime.

If no conflict was detected, the address m is added to the readset and the read value is the return value of the read operation.

Executing a write operation for TVar x_m The write operation only takes place in the writeset and in the writelog array, i.e., the new value is stored in $\text{writelog}[m]$ (or it overwrites a previous value) and m is added to the writeset.

Commit phase Once a transaction has processed all STM commands, the commit phase begins. The following steps are performed:

- Lock all memory locations from the writeset: The transaction sets the truth values of the compare-and-swap objects to `True` to lock them. If the transaction does not obtain all locks, it aborts, releases all locks it holds, and restarts. To implement this step, the transaction must remember which memory locations have already been (lockset is used for this). To lock the memory locations, it is recommended to lock them in a total order to avoid deadlocks.
- Increment the global counter. The new value of the counter is stored in the local variable `writever`.
- Validating the readset: The readset contains the addresses of the read memory locations. For each of these addresses, the current timestamp is checked to see whether $\text{readver} \leq \text{ver}$, i.e., whether the object has been written to in the meantime. If one of these tests fails, or if one of the memory locations is locked, the transaction is aborted and restarted. Verification of the readset can be omitted if $\text{writever} = \text{readver} + 1$, because then no other transaction has written to the transactional variables between the first read and the commit phase.
- Writing the entries belonging to the writeset to the shared memory: The new values from the writelog array are written to the corresponding memory cells.
- Unlocking and setting the timestamp: The locked memory locations are unlocked and the value `writever` is set as the new timestamp.

Checking the readset and locking the compare-and-swap objects belonging to the writeset ensures that the effect of the transaction acts like an atomically executed function from read memory locations to written memory locations.

Aborting a transaction When aborting, all local data structures are reinitialized. If the abort occurs during the commit, the locked memory locations are unlocked again using the lockset. A special feature of the TL2 algorithm is that few locks are used. For example, the lock on the memory locations is only set during commit, but not when reading.

The algorithm still lacks the creation of new transactional memory locations, but this can be easily accomplished by adding another set of new memory locations that are only written to the shared memory at commit time.

A further optimization treats transactions that only read differently: These can dispense with the administration of the writeset, but also the readset. They validate read accesses only by means of the readver timestamp and the comparison of readver with ver_2 for each read access. If these do not result in an abort, it is ensured that the transaction has read a consistent view of the transactional variables.

Whether a transaction only reads can be determined in various ways :

- The programmer specifies this by means of an annotation.
- A program analysis determines this.
- Each transaction is initially executed as a read-only transaction. However, as soon as it wants to perform a write action, it is restarted as a read-write transaction.

Semantically, the TL2 algorithm has a weakness: non-terminating transactions are not aborted, even if the non-termination only occurs conditionally. Consider, for example, the two transactions that access the transactional memory location x :

<u>Transaction 1</u>	<u>Transaction 2</u>
if x then	$x := \text{False};$
loop forever	
else return	

If x is assigned True and transaction 1 is executed first, and runs into an infinite loop, and transaction 2 is executed in the meantime, transaction 1 should be restarted because the value read for x is no longer current. However, the TL2 algorithm does not detect this. Semantically correct handling would require that the readset be checked for validity at regular intervals and that the transaction be aborted if the readset is invalid.

4.4.6 Conclusion

In conclusion, it should be noted that transactional memory is still under development, but is often used. A major problem with TM systems is that only operations that can be reversed should be used within transactions. For example, in a system with direct updates, a `print "Hello"`-command will be executed immediately and "Hello" will appear on the screen. This command is difficult to reverse (clear the screen?).

4.5 References

Sections 4.1 – 4.3 are essentially taken from (Taubenfeld, 2006), section 4.4 "Transactional Memory" is based on (Larus & Rajwar, 2006; Larus & Rajwar, 2010) and (Peyton-Jones, 2007, Chapter 24). The presentation of the correctness criteria from Section 4.4.4 is a summary of the presentation from (Guerraoui & Kapalka, 2008). Further criteria and a comprehensive overview can be found in (Dziura et al., 2015). The TL2 algorithm was described in (Dice et al., 2006).

5

The π -Calculus: Mobile Processes with Message-Passing

In this chapter, we consider a famous model for concurrent programming languages. In general, such *program calculi* consist of the *syntax*, which specifies which expressions may be formed in the language, and the *semantics*, which specifies the meaning of the expressions. These calculi can be seen as the core of real programming languages where the full syntax is translated into the simpler language during compilation, as these are easier to understand and, among other things, easier to handle mathematically. With the help of semantics, one can provide statements about programs and their properties. The field of formal semantics essentially distinguishes between three approaches:

- An *axiomatic semantics* describes properties of programs using logical axioms or rules of inference. Further properties of programs can then be derived using logical inference rules. In general, axiomatic semantics do not cover all properties, but only individual properties of the programs. A prominent example of axiomatic semantics is the Hoare calculus (Hoare, 1969).
- *Denotational semantics* use mathematical spaces to define the meaning of programs. A *semantic function* maps programs to the corresponding mathematical space. For functional programming languages, domains, i.e., partially ordered sets, are often used as mathematical spaces. For non-deterministic languages, this formalism is often extended to so-called power domains, i.e., programs themselves are mapped to sets. Denotational semantics are mathematically elegant, but often difficult to define for more complex languages. This becomes even more difficult when concurrent (or non-deterministic) languages are used.
- An *operational semantics* defines the meaning of programs by defining how programs are *evaluated*. There are various ways to define operational semantics: State transition systems specify how the state of the machine (memory) changes during evaluation, abstract machines specify a machine model for evaluating programs, and substitution systems define how the value of programs can be calculated through term substitutions. Operational semantics can be further divided into *big-step* and *small-step* semantics. While small-step semantics specify evaluation in small steps, i.e., programs are evaluated step by step, big-step semantics specify this evaluation in larger (usually one) steps. Big-step semantics often allow freedom in the implementation of an interpreter based on them, whereas with small-step semantics, an interpreter can usually be read directly from the rules.

We will consider operational semantics that can be understood as rewriting systems and are small-step semantics.

We consider the π -calculus, which is a model for message-passing concurrency, i.e., processes communicate exclusively via the exchange of messages.

5.1 The Lambda Calculus

In this section, we consider the lambda calculus as a model for (in particular, functional) sequential programming languages. We will present various evaluation strategies (i.e., operational semantics). The lambda calculus was introduced by Alonzo Church in the 1930s (Church, 1941). Expressions of the lambda calculus can be formed with the non-terminal **Expr** using the following context-free grammar :

$$\mathbf{Expr} ::= V \mid \lambda V. \mathbf{Expr} \mid (\mathbf{Expr} \mathbf{Expr})$$

Here, V is a non-terminal that generates a variable (from an infinite set of variable names). $\lambda x.s$ is referred to as an *abstraction*. The lambda binder λx binds the variable x within the subexpression s , i.e., conversely: The scope of x is s . An abstraction acts like an anonymous function, i.e., like a function without a name. For example, the identity function $id(x) = x$ can be represented in lambda calculus by the abstraction $\lambda x.x$.

The construct $(s t)$ is referred to as an *application*. Applications can be used to apply functions to arguments. In this case, both the function (the expression s) and the argument t can be any expression in lambda calculus. This means, in particular, that abstractions themselves can be arguments of applications. This is why we also refer to higher-order lambda calculus. For example, the identity function can be applied to itself, i.e. the application $id(id)$ can be written in lambda notation as $(\lambda x.x) (\lambda x.x)$.

In order to formally define the domains of variables, we define the functions FV and BV . For an expression t , $BV(t)$ is the set of its *bound variables* and $FV(t)$ is the set of its *free variables*, which are defined (inductively) by the following rules:

$$\begin{array}{ll} FV(x) &= x & BV(x) &= \emptyset \\ FV(\lambda x.s) &= FV(s) \setminus \{x\} & BV(\lambda x.s) &= BV(s) \cup \{x\} \\ FV(s t) &= FV(s) \cup FV(t) & BV(s t) &= BV(s) \cup BV(t) \end{array}$$

If $FV(t) = \emptyset$ holds for an expression t , we say that t is *closed* or that t is a *program*. If an expression t is not closed, we call t *open*.

To define the operational semantics of the lambda calculus, we need the concept of *substitution*: We write $s[t/x]$ for the expression that results from replacing all free occurrences of the variable x in s with the expression t . To avoid name conflicts in this substitution, we assume that $x \notin BV(s)$ holds. Under this assumption, substitution

can be formally defined by the following equations:

$$\begin{aligned} x[t/x] &= t \\ y[t/x] &= y, \text{ if } x \neq y \\ (\lambda y.s)[t/x] &= \lambda y.(s[t/x]) \\ (s_1 s_2)[t/x] &= (s_1[t/x] s_2[t/x]) \end{aligned}$$

For example, $(\lambda x.z x)[(\lambda y.y)/z]$ yields the expression $(\lambda x.((\lambda y.y) x))$.

Contexts C are expressions in which exactly one subexpression is replaced by a hole (represented by $[\cdot]$). Contexts can also be defined using the following context-free grammar:

$$\mathbf{C} = [\cdot] \mid \lambda V.\mathbf{C} \mid (\mathbf{C} \text{ Expr}) \mid (\text{Expr } \mathbf{C})$$

Expressions can be inserted into contexts to obtain a new expression. Let C be a context and s an expression. Then $C[s]$ is the expression obtained by inserting the expression s into C in place of the hole. This insertion can capture variables. Consider, for example, the context $C = \lambda x.[\cdot]$. Then $C[\lambda y.x]$ is the expression $\lambda x.(\lambda y.x)$. The free variable x in $\lambda y.x$ is captured during substitution.

Now we can define α -renaming: An α -renaming step has the form:

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s[y/x]] \text{ if } y \notin BV(\lambda x.s) \cup FV(\lambda x.s)$$

The reflexive-transitive hull of such α -renamings is called α -equivalence. We do not distinguish between α -equivalent expressions. Rather, we assume that all bound variables in an expression have different names and that the names of bound variables are always different from the names of free variables. With the help of α -renaming, this convention can always be adhered to.

The classic reduction rule of lambda calculus is β -reduction, which evaluates the function application:

$$(\beta) \quad (\lambda x.s) t \rightarrow s[t/x]$$

If $s_0 = (\lambda x.s) t \xrightarrow{\beta} s[t/x] = t_0$, then we also say that s_0 reduces directly to t_0 . However, it is not sufficient to perform β -reductions at the top level only, as this would prevent further reduction of expressions such as $((\lambda x.x) (\lambda y.y)) (\lambda z.z)$. To define the operational semantics, it is therefore necessary to specify where in the expression the β -reductions should be applied. Consider, for example, the expression $((\lambda x.x x)((\lambda y.y)(\lambda z.z)))$ as an example. It contains two different positions that could be reduced (the corresponding subexpressions are called *relexes*): $((\lambda x.x x)((\lambda y.y)(\lambda z.z))) \rightarrow ((\lambda y.y)(\lambda z.z)) ((\lambda y.y)(\lambda z.z))$ or $((\lambda x.x x)((\lambda y.y)(\lambda z.z))) \rightarrow ((\lambda x.x x)(\lambda z.z))$.

This specification is also called a reduction strategy. We introduce two different strategies with call-by-name and call-by-value evaluation.

5.1.1 Call-by-Name Evaluation

Call-by-name evaluation always searches for the reindex that is furthest up and furthest to the left. Formally, call-by-name evaluation can be defined using reduction contexts.

Definition 5.1.1. *Call-by-name reduction contexts R are defined by the following grammar:*

$$R ::= [\cdot] \mid (R \text{ Expr})$$

If $s_0 \xrightarrow{\beta} t_0$, then $R[s_0] \xrightarrow{\text{name}} R[t_0]$ is a call-by-name reduction step.

One property of call-by-name reduction is that it is always *deterministic*, i.e., for an expression s , there is at most one expression t such that $s \xrightarrow{\text{name}} t$. There are also expressions for which no reduction is possible. On the one hand, these are expressions in which the evaluation encounters a free variable (e.g., $(x (\lambda y.y))$ is not reducible). On the other hand, these are values: For the call-by-name lambda calculus, these are abstractions. This means that as soon as we have achieved an abstraction using $\xrightarrow{\text{name}}$ reductions, the evaluation is complete. Expressions that can be converted into an abstraction in this way *converge* (or *terminate*). We define this formally, where $\xrightarrow{\text{name},+}$ is the transitive hull of $\xrightarrow{\text{name}}$ and $\xrightarrow{\text{name},*}$ is the reflexive-transitive hull of $\xrightarrow{\text{name}}$ ¹.

Definition 5.1.2. *An expression s (call-by-name) converges if and only if there is a sequence of call-by-name reductions that transforms s into an abstraction, which we write as $s \downarrow_{\text{name}}$. That is, $s \downarrow_{\text{name}}$ if $\exists \text{abstraction } v : s \xrightarrow{\text{name},*} v$. If s does not converge, we write $s \uparrow_{\text{name}}$ and say s diverges.*

The programming language Haskell has call-by-name lambda calculus as its semantic basis, where implementations use the improved strategy of call-by-need evaluation, which avoids double evaluations of arguments. The following property applies to the call-by-name strategy:

Proposition 5.1.3. *Let s be a lambda expression and s can be converted to an abstraction v using arbitrary β -reductions (at arbitrary positions). Then the following applies: Let s be a lambda expression and s can be converted to an abstraction v using arbitrary β -reductions (at arbitrary positions). Then $s \downarrow_{\text{name}}$ applies.*

This statement shows that call-by-name evaluation is an optimal strategy with regard to termination.

¹Formally, we can define:

$$\begin{aligned} \xrightarrow{\text{name},0} &:= \{(s, s) \mid s \text{ is an expression of the lambda calculus}\}, \\ \xrightarrow{\text{name},i+1} &:= \{(s, t) \mid s \xrightarrow{\text{name}} t' \text{ and } t' \xrightarrow{\text{name},i} t\} \\ \xrightarrow{\text{name},*} &:= \bigcup_{i \in \mathbb{N}_0} \xrightarrow{\text{name},i} \\ \xrightarrow{\text{name},+} &:= \bigcup_{i \in \mathbb{N}, i > 0} \xrightarrow{\text{name},i} \end{aligned}$$

5.1.2 Equality of programs

So far, we have introduced call-by-name and call-by-value lambda calculus, defined the syntax and the respective operational semantics. We can use them to execute (i.e., evaluate) programs, but we still lack a concept of program equality. This is necessary, for example, to verify whether a compiler optimizes correctly, i.e., whether an original program and an optimized program are equal.

According to Leibniz's principle, two things are equal if they have the same properties with respect to all properties. Then the things are interchangeable in any context. For program calculi, this can be summarized as follows: Two expressions s, t are equal if they cannot be distinguished, regardless of the context in which they are used. More formally, s and t are equal if, for all contexts C , $C[s]$ and $C[t]$ behave identically. Here, we still need a term to describe the behavior we want to observe. For deterministic languages, it is sufficient to observe termination, which we have already defined as convergence for both calculi.

Following this pattern, we define *contextual equality*, whereby one can first define an approximation and then equality represents the symmetrization of the approximation.

Definition 5.1.4 (Contextual approximation and equality). *For the call-by-name lambda calculus, we define contextual approximation $\leq_{c,name}$ and contextual equality $\sim_{c,name}$ as:*

- $s \leq_{c,name} t$ if and only if $\forall C : C[s] \downarrow_{name} \implies C[t] \downarrow_{name}$
- $s \sim_{c,name} t$ if and only if $s \leq_{c,name} t$ and $t \leq_{c,name} s$

Contextual equality can be regarded as the coarsest equality (i.e., as many programs as possible are equal) that distinguishes between obviously different programs. An important property of contextual equality is the following:

Proposition 5.1.5. *$\sim_{c,name}$ is a congruence, i.e. they are equivalence relations and compatible with contexts, i.e. $s \sim t \implies C[s] \sim C[t]$.*

Proof. Reflexivity $s \sim_{c,name} s$ and symmetry $s \sim_{c,name} t \implies t \sim_{c,name} s$ obviously hold. For transitivity, let $s \sim_{c,name} t$ and $t \sim_{c,name} r$. Let C be a context such that $C[s] \downarrow_{name}$. Then it follows from $s \sim_{c,name} t$ that $C[t] \downarrow_{name}$ and from $t \sim_{c,name} r$ follows $C[r] \downarrow_{name}$. Similarly, we can conclude that $C[r] \downarrow_{name}$ also implies $C[s] \downarrow_{name}$. Thus, $s \sim_{c,name} r$.

For the congruence property, assume $s \sim_{c,name} t$ and let C be a context. We must show that $C[s] \sim_{c,name} C[t]$. Let C' be a context such that $C'[C[s]] \downarrow_{name}$, then it follows from $s \sim_{c,name} t$ that $C'[C[t]] \downarrow_{name}$ (use the context $C'[C[\cdot]]$ in the definition of $\sim_{c,name}$). Conversely, one can also conclude $C'[C[t]] \downarrow_{name} \implies C'[C[s]] \downarrow_{name}$ in the same way. Since this holds for every context C' , it follows that $C[s] \sim_{c,name} C[t]$. $\square$

The congruence property allows, among other things, subprograms of a large program to be transformed (or optimized) while maintaining the same overall program (with the requirement that the local transformation preserves contextual equality).

Contextual equality provides a concept of program equality that is recognized. One disadvantage of equality is that equality proofs are generally difficult, since all contexts must be considered. In general, contextual equality is undecidable, since the question of whether $s \sim_c \Omega$ holds would solve the halting problem. We will not go into more detail about the proof techniques. However, it is worth mentioning that inequality is generally easier to provide proof for, since in this case one only has to specify a context that distinguishes expressions with respect to their convergence.

It can be provided a proof that the (β) reduction preserves equality in the call-by-name lambda calculus, i.e., $(\beta) \subseteq \sim_{c, name}$.

5.2 A message-passing model: The π -calculus

In the π -calculus, everything is represented as a process, and the only operation is communication between processes. The π -calculus was developed by Robin Milner, Joachim Parrow, and David Walker; it is primarily a theoretical model and is not used for real programming. However, it has been highly influential, and many variants of the calculus have found applications in areas such as bioinformatics.

5.2.1 The synchronous π -calculus

We will first introduce the syntax of the synchronous π -calculus, then we will describe the asynchronous π -calculus, which has a more restricted syntax. From now on, we will simply write π -calculus when it is clear from the context which of the two variants is meant.

5.2.1.1 Syntax of the synchronous π -calculus

We describe a variant of the synchronous π -calculus. There are various other variants that contain, for example, sums of processes or recursive process definitions (the reader is referred to the further literature for more information).

Let $\mathcal{N}$ be a countably infinite set of *names*. In the π -calculus, processes can perform various *actions* that are noted by a so-called *action prefix* π :

$$\pi ::= x(y) \mid \bar{x}y \quad \text{where } x, y \in \mathcal{N}$$

The syntax of π -calculus processes P is given by the grammar:

$$P ::= \pi.P \mid P_1 \mid P_2 \mid !P \mid \mathbf{0} \mid \nu x.P \quad \text{for } x \in \mathcal{N}$$

We first describe the actions for processes: The process $x(y).P$ *receives* the name y via the channel named x and then behaves like P , except that y is replaced by the received name in P . The process $\bar{x}y.P$ *sends* the name y via the channel named x and then behaves like P .

The other constructs for processes can be explained informally as follows:

- 0 is the *inactive process*, which does nothing more.
- $P_1 \mid P_2$ is the *parallel composition* of the two processes P_1 and P_2 . The two processes run concurrently and can communicate via shared names (or channels).
- $\nu z.P$ is the *restriction* that limits the scope of the name z to P .
- $!P$ is the *replication*, whereby the process P is multiplied any number of times, i.e., $!P$ can be regarded as an abbreviation for $\underbrace{P \mid P \mid \dots \mid P}_{\text{infinitely many times}}$.

Thus, in the π -calculus, there are two constructs that *bind* names and thus also define *scopes* of names: The restriction $\nu z.P$ binds the name z with scope P , and for the process with input prefix $x(y).P$, the name y with scope P is introduced. Note that an output prefix $\bar{x}y$ binds neither x nor y . Names that are not in any scope are called *free names*, other names are *bound*. We define this formally for processes:

For a process P , the *set of its free names* $\text{fn}(P)$ is inductively defined by the equations:

$$\begin{aligned} \text{fn}(x(y).P) &= \{x\} \cup (\text{fn}(P) \setminus \{y\}) \\ \text{fn}(\bar{x}y.P) &= \{x, y\} \cup \text{fn}(P) \\ \text{fn}(P_1 \mid P_2) &= \text{fn}(P_1) \cup \text{fn}(P_2) \\ \text{fn}(0) &= \emptyset \\ \text{fn}(\nu x.P) &= \text{fn}(P) \setminus \{x\} \\ \text{fn}(!P) &= \text{fn}(P) \end{aligned}$$

The *set of bound names* $\text{bn}(P)$ of a process P is defined inductively by the equations:

$$\begin{aligned} \text{bn}(x(y).P) &= \{y\} \cup \text{bn}(P) \\ \text{bn}(\bar{x}y.P) &= \text{bn}(P) \\ \text{bn}(P_1 \mid P_2) &= \text{bn}(P_1) \cup \text{bn}(P_2) \\ \text{bn}(0) &= \emptyset \\ \text{bn}(\nu x.P) &= \{x\} \cup \text{bn}(P) \\ \text{bn}(!P) &= \text{bn}(P) \end{aligned}$$

We denote all names of a process by $\text{n}(P)$ (i.e., $\text{n}(P) := \text{fn}(P) \cup \text{bn}(P)$). An occurrence of a name in a process is *bound* if it is within the scope of the corresponding binder, otherwise the occurrence is *free*. For a process P , $P[x/y]$ is the process that results from P by replacing all free occurrences of the name y with the name x , whereby no unwanted name capture may occur, i.e., a previously free occurrence of y is replaced by a bound occurrence of x .

Name capture can be prevented by renaming bound names. However, before defining renaming, we define contexts: A process context D is like a process that has a hole (which we denote with $[\cdot]$) at one position instead of a process. Process contexts can also be formally defined using the following grammar, where π and P are defined as above.

$$D ::= [\cdot] \mid \pi.D \mid D \mid P \mid P \mid D \mid !D \mid \nu x.D \quad \text{for } x \in \mathcal{N}$$

Definition 5.2.1 (Alpha-Equivalence). A renaming of bound names of a process P is an α -renaming step defined as:

$$\begin{aligned} D[x(y).P] &\xrightarrow{\alpha} D[x(z).P[z/y]] \text{ if } z \notin n(x(y).s) \\ D[\nu y.P] &\xrightarrow{\alpha} D[\nu z.P[z/y]] \text{ if } z \notin n(\nu y.P) \end{aligned}$$

The relation $=_{\alpha}$ is the reflexive-transitive hull of $\xrightarrow{\alpha}$. Processes P_1, P_2 are called α -equivalent if they can be made equal by renaming bound names.

We do not distinguish between α -equivalent processes; rather, we require that the following convention always hold:

Convention 5.2.2. For a process P , we always assume that its bound names are pairwise distinct and that bound and free names are disjoint.

It is easy to adhere to this convention, since it can always be satisfied by renaming bound names, and α -equivalent processes are not distinguished.

In what follows, we define the relation $\equiv$, which is *structural congruence* for processes of the π -calculus.

α -equivalent contexts are *not* considered equal, i.e., for example, $\nu x.[\cdot]$ and $\nu y.[\cdot]$ are genuinely different contexts.

A binary relation $\mathcal{R}$ on processes is *compatible with contexts* if for all $(a, b) \in \mathcal{R}$ and all contexts D , $(D[a], D[b]) \in \mathcal{R}$ holds. An equivalence relation on processes that is compatible with contexts is called *congruence*.

Definition 5.2.3. The structural congruence $\equiv$ is defined as the smallest congruence on processes that satisfies the following axioms.

$$\begin{aligned} P &\equiv Q, \text{ if } P \text{ and } Q \text{ are } \alpha\text{-equivalent} \\ P_1 \mid (P_2 \mid P_3) &\equiv (P_1 \mid P_2) \mid P_3 \\ P_1 \mid P_2 &\equiv P_2 \mid P_1 \\ P \mid \mathbf{0} &\equiv P \\ \nu z.\nu w.P &\equiv \nu w.\nu z.P \\ \nu z.\mathbf{0} &\equiv \mathbf{0} \\ \nu z.(P_1 \mid P_2) &\equiv P_1 \mid \nu z.P_2, \text{ if } z \notin fn(P_1) \\ !P &\equiv P \mid !P \end{aligned}$$

Conversely, if P_1 and P_2 are processes that can be made equal using the axioms (applied to subprocesses) and α -renaming, then P_1 and P_2 are structurally congruent (i.e., $P_1 \equiv P_2$).

An interesting fact about structural congruence defined in this way is that it is still unclear whether it is decidable, i.e., the decision problem is whether $P \equiv Q$ holds for given P and Q . It is known that the problem is at least EXPSPACE-hard (see, for example, (Engelfriet & Gelsema, 2007; Schmidt-Schauß et al., 2013)).

5.2.1.2 Operational semantics of the synchronous π -calculus

In what follows, we define the reduction semantics for the π -calculus, which can be regarded as small-step semantics. The evaluation rules are as follows:

$$\begin{array}{ll}
 \text{(Interact)} & x(y).P \mid \bar{x}v.Q \rightarrow P[v/y] \mid Q \\
 \text{(Par)} & P \mid Q \rightarrow P' \mid Q, \text{ if } P \rightarrow P' \\
 \text{(New)} & \nu x.P \rightarrow \nu x.P', \text{ if } P \rightarrow P' \\
 \text{(StructCongr)} & P \rightarrow P', \text{ if } Q \rightarrow Q' \text{ and } P \equiv Q \text{ and } P' \equiv Q'
 \end{array}$$

The essential reduction rule is (Interact), as it implements communication between two processes. The rules (Par) and (New) enable reduction under parallel composition and under ν -binders. The fourth rule (StructCongr) allows the process to be transformed before and after reduction using structural congruence. Note that reduction does not occur below replication or below action prefixes. This means, for example, that neither the process $x(y).(u(v).\mathbf{0} \mid \bar{u}w.\mathbf{0})$ nor the process $!(u(v).\mathbf{0} \mid \bar{u}w.\mathbf{0})$ can be reduced directly. However, the second process can be reduced with the help of structural congruence.

Let's look at a few examples:

Example 5.2.4. *The process*

$$P \equiv \underbrace{x(y).\bar{y}y.\mathbf{0}}_{P_1} \mid \underbrace{\bar{x}z.\mathbf{0}}_{P_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3}$$

can be reduced as follows. First, P_1 and P_2 communicate via the channel named x , whereby P_2 sends the name z to P_1 :

$$P \rightarrow \underbrace{\bar{z}z.\mathbf{0}}_{P'_1} \mid \underbrace{\mathbf{0}}_{P'_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3} \equiv P'$$

Now P'_1 and P_3 can communicate via the channel named z , whereby P'_1 sends the name z :

$$P' \rightarrow \mathbf{0} \mid \mathbf{0} \mid \mathbf{0} \equiv P''$$

The resulting process is structurally congruent to $\mathbf{0}$ and cannot be further reduced.

Example 5.2.5. *The reduction is non-deterministic, e.g., for the process*

$$P \equiv x(y).\mathbf{0} \mid \bar{x}v.\mathbf{0} \mid x(z).\bar{z}w.\mathbf{0}$$

two reductions are possible. The first two processes can interact:

$$P \rightarrow \mathbf{0} \mid \mathbf{0} \mid x(z).\bar{z}w.\mathbf{0},$$

but also the last two:

$$P \rightarrow x(y).\mathbf{0} \mid \mathbf{0} \mid \bar{v}w.\mathbf{0}$$

Example 5.2.6. ν -binders prevent communication possibilities and thus enable local communication that is protected from external interference. For example, for

$$P \equiv \nu x.(x(y).0 \mid \bar{x}v.0) \mid x(z).\bar{z}w.0$$

only one reduction is possible, since the name x in the last process is not in the binding scope of the ν -binder and must therefore be regarded as different from the name x in the preceding processes. A correct α -renaming of P , which leads to the fulfillment of convention 5.2.2, makes this clear:

$$P \equiv \nu x'.(x'(y).0 \mid \bar{x'}v.0) \mid x(z).\bar{z}w.0$$

Now it is obvious that the last process cannot communicate with the other processes.

However, it should also be mentioned that local names can leave their binding scope through communication. For example, let $P \equiv x(y).P_1 \mid \nu z.\bar{x}z.P_2$, where z does not occur in P_1 . By applying (Interact), the name z , which appears to be known only to P_2 , also becomes known to P_1 , because $P \rightarrow \nu z.(P_1[z/y] \mid P_2)$. This phenomenon is also referred to as extrusion.

Example 5.2.7. There are processes that can communicate infinitely often. A simple example is the process

$$P = !(\bar{x}v.0) \mid !(x(z).0)$$

We show the beginning of the infinitely long reduction sequence, making some transformations explicit with regard to structural congruence :

$$\begin{aligned} P &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid x(z).0 \mid !(x(z).0) \\ &\rightarrow !(\bar{x}v.0) \mid 0 \mid 0 \mid !(x(z).0) \equiv !\bar{x}v.0 \mid !(x(z).0) \equiv P \end{aligned}$$

5.2.1.3 Turing power of the synchronous π -calculus

In this section, we illustrate that the synchronous π -calculus is Turing-powerful by showing how the lambda calculus can be encoded in the π -calculus. We refrain from providing a complete proof of the correctness of this encoding (this can be found in (Milner, 1992)).

The call-by-name lambda calculus is encoded in the π -calculus using the mapping $\llbracket \cdot \rrbracket$, where for a lambda expression s , the translation $\llbracket s \rrbracket$ is not yet a π -calculus process, but rather a function that expects a name and then delivers a process, i.e., for example, $\llbracket s \rrbracket u$ is a process for a name u . The name u is the channel name through which the process associated with s receives its arguments.

The following equations specify how all constructs of the lambda calculus are to be translated. This means that three cases must be defined: the translation of variables, the translation of abstractions, and the translation of applications.

An abstraction $\lambda x.s$ is translated in such a way that it receives two names via the additional name (i.e., u): first, the name of its argument, i.e., x , and then the name that s should use to receive arguments:

$$\llbracket \lambda x.s \rrbracket u := u(x).u(v).\llbracket s \rrbracket v$$

A variable x is translated into a process that sends the name u introduced by the translation via the channel named x :

$$\llbracket x \rrbracket u := \bar{x}u.\mathbf{0}$$

The application $(s\ t)$ is translated in such a way that the argument t is generated as a separate process, quasi like a binding $x = t$. The request for the argument then occurs via x . Each time x is requested, the argument receives a channel name via which it receives its arguments. Since the argument t may be needed more often within s , this process is replicated. We define the abbreviation here:

$$\langle x = t \rangle := !(x(w).\llbracket t \rrbracket w)$$

The application $(s\ t)$ is now translated as follows, where x is a new name, i.e., it does not appear in t :

$$\llbracket s\ t \rrbracket u := \nu v.(\llbracket s \rrbracket v \mid \nu x.\bar{v}x.\bar{v}u.\langle x = t \rangle)$$

The translation is not easy to understand in detail, as some communication takes place via channels. We demonstrate the translation using a few examples:

Consider the lambda expression $(\lambda x.x)\ t$. The translation (where u can be freely chosen) yields:

$$\begin{aligned} \llbracket (\lambda x.x)\ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \end{aligned}$$

Now we can reduce as follows:

$$\begin{aligned} &\nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ \equiv &\nu y.\nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \bar{v}y.\bar{v}u.\langle y = t \rangle) \\ \rightarrow &\nu y.\nu v.(v(w).\bar{y}w.\mathbf{0} \mid \bar{v}u.\langle y = t \rangle) \\ \rightarrow &\nu y.\nu v.(\bar{y}u.\mathbf{0} \mid \langle y = t \rangle) \end{aligned}$$

After expanding the abbreviation for $\langle y = t \rangle$ and a further reduction, this results in:

$$\begin{aligned} \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid \langle y = t \rangle) &= \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid !(y(w).\llbracket t \rrbracket w)) \\ \equiv &\nu y.\nu v.(\bar{y}u.\mathbf{0} \mid (y(w).\llbracket t \rrbracket w) \mid !(y(w).\llbracket t \rrbracket w)) \\ \rightarrow &\nu y.\nu v.(\mathbf{0} \mid (\llbracket t \rrbracket u) \mid !(y(w).\llbracket t \rrbracket w)) \\ \equiv &\llbracket t \rrbracket u \mid \nu y.(! (y(w).\llbracket t \rrbracket w)) \end{aligned}$$

If we interpret the subprocess $\nu y.(! (y(w).\llbracket t \rrbracket w))$ so that it can be removed, since no other process can communicate with it anymore (y is unknown for t). The result is exactly what is expected from the call-by-name lambda calculus, because $(\lambda x.x)\ t \xrightarrow{\beta} t$.

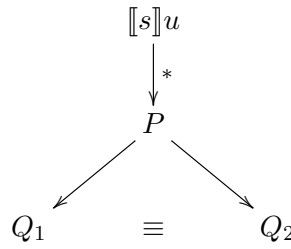
In (Milner, 1992), it was shown that:

Proposition 5.2.8. *Let s be a closed lambda expression, then one of the following conditions holds:*

1. $s \downarrow_{name}$ and $\llbracket s \rrbracket u \xrightarrow{*} P$, such that P is irreducible.
2. $s \uparrow_{name}$ and there is no P such that $\llbracket s \rrbracket u \xrightarrow{*} P$, where P is irreducible.

This proposition shows that call-by-name evaluation of lambda expressions can be faithfully simulated in the π -calculus via the translation $\llbracket \cdot \rrbracket$. The translation $\llbracket \cdot \rrbracket$ satisfies another condition:

Lemma 5.2.9. *Let s be a closed lambda expression, then $\llbracket s \rrbracket u$ is deterministic with respect to $\equiv$, i.e., if $\llbracket s \rrbracket u \xrightarrow{*} P$ and $P \rightarrow Q_1$ as well as $P \rightarrow Q_2$, then $Q_1 \equiv Q_2$ always holds. This statement can be illustrated by the following diagram*



This lemma strengthens Proposition 5.2.8 by showing that lambda expressions are translated into *deterministic* π -calculus processes. Since the lambda calculus is Turing-complete, it follows that:

Theorem 5.2.10. *The synchronous π -calculus is Turing-powerful.*

Note that this does not yet say anything about the equality theories of the lambda calculus and the π -calculus. A stronger connection is proved in (Milner, 1992). We do not pursue this here, since we have not yet defined a concept of equality for the π -calculus; moreover, there are different concepts of equality for the π -calculus, which we will discuss later.

5.2.2 The asynchronous π -calculus

In the synchronous π -calculus presented above, all communication takes place synchronously, since the exchange of messages (the receiving and sending of the name) is carried out directly between the sending and receiving processes in a single step. In the asynchronous π -calculus, this is changed. The essential syntactic difference between the synchronous π -calculus and the asynchronous π -calculus is that in the asynchronous π -calculus, the output prefix may only be followed by the inactive process 0 , i.e., the syntax of processes P of the asynchronous π -calculus is defined by the following grammar:

$$P ::= 0 \mid \bar{x}y.0 \mid x(y).P \mid P_1 \mid P_2 \mid \nu z.P \mid !P$$

The asynchronous character now arises from the view that all processes of the form $\bar{x}y.0$ are to be understood as sent messages that float around in space and can be received by a receiving process after any waiting period. The reduction rules are not changed. Only the view that two processes communicate with each other for the rule (Interact) is rejected, since the sending process can no longer be any process, but rather a process that consists only of the message to be sent and the associated channel (this is one of the reasons why the inactive processes 0 are often not explicitly noted. One often finds the notation $\bar{x}y$ instead of $\bar{x}y.0$). This view also allows for a separation of the processes into sent messages (which are currently in the message medium) and the remaining processes.

5.2.2.1 Encodability of the synchronous π -calculus in the asynchronous π -calculus

In order to encode the synchronous π -calculus in the asynchronous one, synchronous communication must be simulated in the asynchronous π -calculus. The idea is that after the message has been received, the receiver sends a confirmation to the sender. Let $P' \mid Q' \equiv (\bar{x}z.P \mid x(y).Q)$ be a process of the synchronous π -calculus. We call P' the sender and Q' the receiver. Then P and Q are blocked until the interaction between the two processes has taken place. A first approach to translating $\bar{x}z.P$ into the asynchronous π -calculus is $\bar{x}z.0 \mid P$. However, P can now be reduced before communication with Q' has taken place. A next idea is to protect the sender P so that it waits for confirmation from Q before it is allowed to continue computing. Similarly, the receiver Q must send this confirmation after it has received the actual message. This results in the following encodings:

$$\text{Sender } S = \bar{x}z.0 \mid u(v).P \quad \text{Receiver } E = x(y).(\bar{u}v.0 \mid Q)$$

where v does not occur freely in P , and u does not occur freely in Q .

We consider the evaluation of $S \mid E$ in asynchronous π -calculus:

$$(\bar{x}z.0 \mid u(v).P) \mid x(y).(\bar{u}v.0 \mid Q) \rightarrow u(v).P \mid \bar{u}v.0 \mid Q[z/y] \rightarrow P \mid Q[z/x]$$

However, this solution still has weaknesses: The encoding is not context-independent: When the translation is applied to a process $D[P' \mid Q']$, there is no guarantee that other processes from D communicate with the sender or receiver via channel u , thereby disrupting synchronization. As an example, let $D = u(w).0 \mid [\cdot]$, then $D[S \mid E]$ can be reduced as follows:

$$\begin{aligned} & u(w).0 \mid (\bar{x}z.0 \mid u(v).P) \mid x(y).(\bar{u}v.0 \mid Q) \\ \rightarrow & u(w).0 \mid (u(v).P) \mid \bar{u}v.0 \mid Q[z/y] \\ \rightarrow & u(v).P \mid Q[z/y] \end{aligned}$$

In this case, P is still blocked because the acknowledgment sent by E was received by the wrong process. Therefore, a *private* channel name should first be exchanged between the sender and receiver, via which only these two processes can communicate.

Considering a sender of the form $\nu u.(\bar{x}u.\mathbf{0} \mid u(v)P)$, it is easy to verify that the subprocess $u(v)P$ can only continue reducing once the message u has been sent via x from $\bar{x}u.\mathbf{0}$, because before that, the channel name u is not visible to other processes. Here, the phenomenon of extrusion is used to make u visible to the outside world. It is obvious that only the process that receives u can communicate with P via u , since only this process can know the name u .

The translation $\llbracket \cdot \rrbracket_\pi$ from synchronous to asynchronous π -calculus processes is based on (Boudol, 1992) and defined as follows:

$$\begin{aligned} \llbracket \mathbf{0} \rrbracket_\pi &= \mathbf{0} \\ \llbracket \bar{x}y.P \rrbracket_\pi &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}y.\mathbf{0} \mid \llbracket P \rrbracket_\pi)), \text{ where } u, v \notin \text{fn}(P) \\ \llbracket x(y).P \rrbracket_\pi &= x(u).\nu v.(\bar{u}v.\mathbf{0} \mid v(y).\llbracket P \rrbracket_\pi), \text{ where } u, v \notin \text{fn}(P) \\ \llbracket P \mid Q \rrbracket_\pi &= \llbracket P \rrbracket_\pi \mid \llbracket Q \rrbracket_\pi \\ \llbracket !P \rrbracket_\pi &= !\llbracket P \rrbracket_\pi \\ \llbracket \nu x.P \rrbracket_\pi &= \nu x.\llbracket P \rrbracket_\pi \end{aligned}$$

We now consider the evaluation of synchronous communication, i.e., the evaluation of $\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$:

$$\begin{aligned} &\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi \\ &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid x(u').\nu v'.(\bar{u}'v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \\ &\rightarrow \nu u.(\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid \nu v'.(\bar{u}'v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \\ &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid (\bar{v}'z.\mathbf{0}) \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \\ &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid \mathbf{0} \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid \llbracket Q \rrbracket_\pi[z/y])) \\ &\equiv \llbracket P \rrbracket_\pi \mid \llbracket Q \rrbracket_\pi[z/y] \end{aligned}$$

Note that the sender could continue calculating after receiving confirmation from the receiver that it wanted to communicate with it, but before the name z was actually sent. However, this does not matter, since at this point the connection between the two processes has been established, i.e., it is certain who is communicating with whom. No process other than the receiver can receive z .

In (Boudol, 1992), it was shown, among other things, that the encoding $\llbracket \cdot \rrbracket_\pi$ preserves convergence. To do this, however, we must first define which convergence concept Boudol uses (there are also different definitions for the π -calculus). Boudol essentially only considers closed processes, for which he also uses a special term:

Definition 5.2.11. *An output-closed process of the synchronous or asynchronous π -calculus is a process such that for every output prefix $\bar{x}y$: The names x and y are bound by binders.*

The answers or results of a reduction are so-called input-answers:

Definition 5.2.12. *A process P is an input-answer if $P \equiv \nu x_1 \dots \nu x_n.x(y).P' \mid Q$*

Input-answers are therefore processes that can still receive input from the outside. This corresponds to the intuition of an abstraction in lambda calculus, since it can still process inputs (arguments).

Now we can formulate Boudol's result for convergence:

Proposition 5.2.13. *Let P be an output-closed process of the synchronous π -calculus. Then the following applies:*

$$\begin{aligned} & \exists \text{ input-answer } P': P \xrightarrow{*} P' \\ & \text{exactly when} \\ & \exists \text{ input-answer } P'': \llbracket P \rrbracket_{\pi} \xrightarrow{*} P'' \end{aligned}$$

The result states that, for output-closed processes, the translation $\llbracket \cdot \rrbracket_{\pi}$ preserves the attainment of an input-answer — Boudol's convergence concept — between the synchronous and asynchronous π -calculus.

Remark 5.2.14. *Note that this result, i.e., the encodability of the synchronous π -calculus in the asynchronous π -calculus, is no longer valid if sums are added to the synchronous π -calculus (see next section) and requirements are placed on the translation:*

- *The translation should be compositional and translate parallel composition as parallel composition, i.e., $\llbracket P \mid Q \rrbracket = \llbracket P \rrbracket \mid \llbracket Q \rrbracket$*
- *The translation should be well-behaved with respect to renaming, i.e., for renaming σ , $\llbracket \sigma(P) \rrbracket = \sigma(\llbracket P \rrbracket)$ should hold.*

Under these assumptions, no meaningful (i.e., semantics-preserving) translation can be found (see (Palamidessi, 1997; Palamidessi, 2003)).

5.2.3 Extensions and Variants of the π -Calculus

In this section, we introduce some variants and extensions of the synchronous π -calculus.

5.2.3.1 Non-Deterministic Selection

In many variants of the π -calculus, non-deterministic choice is added as an additional construct. Here, $P + Q$ corresponds to a process that can behave either like process P or like process Q . However, this non-deterministic choice is often restricted so that the choice between P and Q must be made simultaneously with the receipt or transmission of a message. We follow (Milner, 1999) and only allow processes with the prefix as summands of the sum: The syntax of processes of the *synchronous π -calculus with sums* is defined by the following grammar, whereby there is a new prefix: τ , which stands for the unobservable action.

$$\begin{aligned} \pi & ::= x(y) \mid \bar{x}y \mid \tau \quad \text{where } x, y \in \mathcal{N} \\ P & ::= \pi.P \mid P_1 \mid P_2 \mid !P \mid \mathbf{0} \mid \nu x.P \mid \sum_i \pi.P_i \quad \text{for } x \in \mathcal{N} \end{aligned}$$

We use the summation sign and $+$ as desired. The definitions for bound and free names and for structural congruence must be adapted accordingly. Sums are considered associative and commutative. In addition, the rule $P + P = P$ applies.

The reduction rules must be adjusted according to the sums and with regard to the new prefix τ :

(Silent)	$\tau.P \rightarrow P$
(SilentSum)	$\tau.P + M \rightarrow P$
(InteractSum)	$x(y).P + M \mid \bar{x}v.Q + N \rightarrow P[v/y] \mid Q$
(Interact)	$x(y).P \mid \bar{x}v.Q \rightarrow P[v/y] \mid Q$
(Par)	$P \mid Q \rightarrow P' \mid Q, \text{ if } P \rightarrow P'$
(New)	$\nu x.P \rightarrow \nu x.P', \text{ if } P \rightarrow P'$
(StructCongr)	$P \rightarrow P', \text{ if } Q \rightarrow Q' \text{ and } P \equiv Q \text{ and } P' \equiv Q'$

Note that no reduction is performed within a sum. For example, $(x(y).P + \bar{x}z.Q)$ cannot be reduced because the summands represent alternatives.

The τ -prefix basically adds a non-observable action. It is effectively removed by the reduction. It was only included in the syntax here so that it could be used as a prefix in the sum, because, for example, $(0 + \pi.P)$ is a syntactically invalid construct, whereas $(\tau.0 + \pi.P)$ is valid (if P is syntactically correct).

Using τ as a prefix, any non-deterministic selection can also be encoded: Let $P \oplus Q := \tau.P + \tau.Q$. Then the evaluation of $P \oplus Q$ behaves in such a way that it decides between P and Q in a non-deterministic manner, i.e., $P \oplus Q \rightarrow P$ as well as $P \oplus Q \rightarrow Q$.

5.2.3.2 Polyadic π -calculus

The variants of the π -calculus presented so far are also called *monadic* π -calculus. In contrast to the monadic π -calculus, the *polyadic* π -calculus allows the sending and receiving of multiple names simultaneously, i.e., instead of one name, a tuple of names is exchanged. Let $\vec{x} = (x_1, \dots, x_n)$ be a tuple of names; we write $|\vec{x}| = n$ for its length.

The action prefixes π in the *polyadic* π -calculus are defined by the grammar

$$\pi ::= x(\vec{y}) \mid \bar{x}\vec{z} \text{ where } x, y_i, z_i \in \mathcal{N}$$

(here, too, the τ -prefix can be added for the polyadic calculus with sums).

The Interact rule for defining reduction is adapted as follows:

$$(\text{Interact}) \quad x(\vec{y}).P \mid \bar{x}\vec{z}.Q \rightarrow P[\vec{z}/\vec{y}] \mid Q \text{ if } |\vec{y}| = |\vec{z}|$$

This means that instead of a name, an entire tuple is sent. The notation $P[\vec{z}/\vec{y}]$ refers to the process $P[z_1/y_1, \dots, z_n/y_n]$, where n is the common length of the two tuples.

The Interact rule requires that the input and output tuples have the same length. This would still allow processes to be formed in which the same channel is used for different lengths. Therefore, this is usually prohibited by so-called *sorts*. This works similarly to a type system and specifies for each channel name the tuple lengths for which the channel can be used exclusively. We will not go into further detail here, but assume in the following that only tuples of the same length are ever sent over each channel.

The polyadic π -calculus can be translated into the monadic π -calculus. The idea here is to send the names from the sent tuple $(z_1, \dots, z_n)$ one after the other (i.e., sequentially instead of in parallel). This means that a process with a (polyadic) input prefix $x(y_1, \dots, y_n).P$ is translated as $x(y_1) \dots x(y_n).P'$, where P' is the translation of P , and a polyadic process with output prefix $\bar{x}(z_1, \dots, z_n).Q$ is translated into $\bar{x}z_1 \dots \bar{x}z_n.Q'$, where Q' is the translation of Q .

Unfortunately, however, this translation does not work if more than two processes use the same communication channel, as the following example shows. Let $P := \bar{x}(z_1, z_2).0 \mid \bar{x}(z_3, z_4).0 \mid x(y_1, y_2).\bar{y}_1 y_2.0$. Then there are two ways to reduce P :

- $P \rightarrow 0 \mid \bar{x}(z_3, z_4).0 \mid \bar{z}_1 z_2.0$
- $P \rightarrow \bar{x}(z_1, z_2).0 \mid \bar{z}_3 z_4.0$

If we translate the process P using the above idea, we obtain $Q := \bar{x}z_1.\bar{x}z_2.0 \mid \bar{x}z_3.\bar{x}z_4.0 \mid x(y_1).x(y_2).\bar{y}_1 y_2.0$. For the translated process, there are four results after reduction:

- $Q \xrightarrow{*} 0 \mid \bar{x}z_3.\bar{x}z_4.0 \mid \bar{z}_1 z_2.0$
- $Q \xrightarrow{*} \bar{x}z_1.\bar{x}z_2.0 \mid 0 \mid \bar{z}_3 z_4.0$
- $Q \xrightarrow{*} \bar{x}z_2.0 \mid \bar{x}z_4.0 \mid \bar{z}_1 z_3.0$
- $Q \xrightarrow{*} \bar{x}z_2.0 \mid \bar{x}z_4.0 \mid \bar{z}_3 z_1.0$

The last two results arise from the fact that the sequentialized operations were mixed up. The translation can thus introduce completely different behavior than the original process had. The pitfall with translation is that the transfer of the individual names of the tuple does not have to arrive at the *same* recipient or originate from the *same* sender. This can be corrected by having the sender and recipient first exchange a private channel name through which the actual communication takes place. Once the name has been exchanged, no other process can interfere with the communication.

The translation $\llbracket \cdot \rrbracket_{poly}$ of the polyadic into the monadic π -calculus is defined as follows:

$$\begin{aligned}
 \llbracket x(z_1, \dots, z_n).P \rrbracket_{poly} &= x(w).w(z_1) \dots w(z_n).\llbracket P \rrbracket_{poly} \\
 \llbracket \bar{x}(z_1, \dots, z_n).P \rrbracket_{poly} &= \nu v.\bar{x}v.\bar{v}z_1 \dots \bar{v}z_n.\llbracket P \rrbracket_{poly} \\
 \llbracket \nu x.P \rrbracket_{poly} &= \nu x.\llbracket P \rrbracket_{poly} \\
 \llbracket P \mid Q \rrbracket_{poly} &= \llbracket P \rrbracket_{poly} \mid \llbracket Q \rrbracket_{poly} \\
 \llbracket !P \rrbracket_{poly} &= !\llbracket P \rrbracket_{poly} \\
 \llbracket 0 \rrbracket_{poly} &= 0
 \end{aligned}$$

This translation behaves correctly.

5.2.3.3 Recursive Definitions

Recursive process definitions are often used instead of replication, i.e., the syntax for processes is changed: replication is deleted and so-called *constant applications* are added:

$$P := \dots \mid \mathcal{L}P \mid \dots \mid A(x_1, \dots, x_n) \text{ where } x_i \in \mathcal{N}$$

Here, A must be defined (outside the program) as: $A(x_1, \dots, x_n) = P$ where $\text{fn}(P) \subseteq \{x_1, \dots, x_n\}$.

The following additional reduction rule is included:

$$\begin{aligned} (\text{Const}) \quad A(y_1, \dots, y_n) &\rightarrow P[y_1/x_1, \dots, y_n/x_n] \\ &\text{if } A(x_1, \dots, x_n) = P \text{ is the definition of } A \end{aligned}$$

We briefly describe why replication and recursion are interchangeable. That is, replication can be defined using recursion and vice versa.

We consider the polyadic π -calculus here, as this simplifies the definitions and translations. However, this can also be transferred to the monadic π -calculus, since we have already seen how the polyadic π -calculus can be translated into the monadic π -calculus.

We first consider the translation of the π -calculus with recursion into the π -calculus with replication. Let $\{A_1, \dots, A_n\}$ be all defined constants, where A_i is defined as $A_i(\vec{x}_i) = P_i$. The translation introduces new channel names $a_1, \dots, a_n$. Each occurrence of a constant application $A_i(\vec{y}_i)$ is replaced by the process $\langle A_i(\vec{y}_i) \rangle := \bar{a}_i \vec{y}_i.0$. The translation $\langle \cdot \rangle$ homomorphically translates all other constructs.

The definitions of the constants A_i are also represented by processes. For this purpose, we define the processes $A'_i := !a_i(\vec{x}_i). \langle P_i \rangle$. The processes A'_i are added to the program so that a process Q with recursion is translated into the following process with replication:

$$\llbracket Q \rrbracket_{\text{rec}} := \nu a_1. \dots \nu a_n. (\langle Q \rangle \mid A'_1 \mid \dots \mid A'_n)$$

Note that this translation is not compositional, since (apart from the dependency of the defined names) the following does not apply, for example: $\llbracket Q_1 \mid Q_2 \rrbracket_{\text{rec}} = \llbracket Q_1 \rrbracket_{\text{rec}} \mid \llbracket Q_2 \rrbracket_{\text{rec}}$.

It can be shown that the translation is correct. We will not go into detail about the concept of correctness here, but we mean that the processes behave essentially the same.

For the inverse translation $\llbracket \cdot \rrbracket_{\text{repl}}$, i.e., the translation of replication into recursion, let us consider a process $!P$. Let $\{x_1, \dots, x_n\} = \text{fn}(P)$. Then choose a new constant A_P and translate as follows:

$$\llbracket !P \rrbracket_{\text{repl}} := A_P(x_1, \dots, x_n)$$

All other constructs are translated homomorphically. The definition for the constant A_P is still missing. It is as follows:

$$A_P(x_1, \dots, x_n) = (\llbracket P \rrbracket_{\text{repl}} \mid A_P(x_1, \dots, x_n))$$

This translation also simulates the behavior correctly.

Finally, let us consider the process $Q = !\bar{x}z.0 \mid x(y).0 \mid x(w).0$ as an example. Then we obtain $\llbracket Q \rrbracket_{repl} = A(x, z) \mid x(y).0 \mid x(w).0$, where $A(x, z) = (\bar{x}z \mid A(x, z))$.

Q reduces to $!\bar{x}z.0$ in two steps. Let us consider an evaluation of $\llbracket Q \rrbracket_{repl}$:

$$\begin{aligned} \llbracket Q \rrbracket_{repl} &= A(x, z) \mid x(y).0 \mid x(w).0 \\ &\rightarrow (\bar{x}z \mid A(x, z)) \mid x(y).0 \mid x(w).0 \\ &\rightarrow A(x, z) \mid 0 \mid x(w).0 \\ &\rightarrow (\bar{x}z \mid A(x, z)) \mid 0 \mid x(w).0 \\ &\rightarrow A(x, z) \mid 0 \mid 0 \end{aligned}$$

5.2.4 Process Equality in π -Calculus

There are various concepts of equality for the π -calculus, which are used depending on the application. In order to define a concept of contextual equality, it would be necessary to define a concept of termination, i.e., convergence for π -calculus processes. However, this is rejected by many researchers because the termination of processes is not considered central, since distributed systems often run in infinite loops and therefore do not terminate.

One approach to defining equality in π -calculus is to use the input and output possibilities of a process as a concept for its behavior. For this purpose, a so-called labeled transition system is often used instead of reduction. This can be thought of as labeled reductions, but with more permissions than the reduction defined so far. The labels represent the input and output possibilities of the process. A distinction is made between internal communication, i.e., communication between two processes, and external communication, i.e., communication with the outside world. Another way of looking at external communication is: How can a process communicate if a communication partner were to be added as an additional process?

Essentially, two types of external communication can be distinguished here: A process of the form $P_0 \equiv \nu z_1 \dots \nu z_n. x(y).P$ with $x, y \neq z_i$ can receive an input, i.e., a process of the form $\bar{x}z$ must be added for communication to take place, where P_0 would be the receiver. A process of the form $P_0 \equiv \nu z_1 \dots \nu z_n. \bar{x}y.P$ with $x, y \neq z_i$ can perform an output, i.e., one would have to add a process $x(z).Q$ parallel to P_0 so that communication can take place, P_0 would be the sender in this case. A slightly different variant is the output of a bound name, i.e., for $P_0 = \nu z_1 \dots \nu z_n. \nu y. \bar{x}y.P$ with $x, y \neq z_i$: Communication also takes place here if a process of the form $x(z).Q$ is added in parallel to P_0 , but a new effect occurs in that the communication pushes the ν -binder for y to the outside. This results in a total of three types of external communication and, in addition, one type for internal communication. Therefore, there are four different labels in the labeled transition system, which are defined by the following grammar.

$$\alpha := \tau \mid x(y) \mid \bar{x}y \mid \bar{x}(y)$$

The label τ refers to an internal (silent) action, $x(y)$ indicates that the process has

performed an input, $\bar{x}y$ indicates that the process has performed an output (where y was not bound), and $\bar{x}(y)$ indicates that the process has performed a bound output. The definition of free, bound, and all names is transferred to such labels as follows:

$$\begin{aligned} \text{fn}(x(y)) &= \{x, y\} & \text{bn}(x(y)) &= \emptyset \\ \text{fn}(\bar{x}y) &= \{x, y\} & \text{bn}(\bar{x}y) &= \emptyset \\ \text{fn}(\bar{x}(y)) &= \{x\} & \text{bn}(\bar{x}(y)) &= \{y\} \\ \text{fn}(\tau) &= \emptyset & \text{bn} &= \emptyset \end{aligned}$$

The labeled transition system for the synchronous π -calculus is now defined by the following rules:

$$\begin{aligned} (\text{In}) \quad & x(y).P \xrightarrow{x(z)} P[z/y] \\ (\text{Out}) \quad & \bar{x}y.P \xrightarrow{\bar{x}y} P \\ (\text{Open}) \quad & \nu y.P \xrightarrow{\bar{x}(y)} Q, \text{ if } P \xrightarrow{\bar{x}y} Q \text{ and } x \neq y. \\ (\text{Interact}) \quad & x(y).P \mid \bar{x}v.Q \xrightarrow{\tau} P[v/y] \mid Q \\ (\text{Par}) \quad & P \mid Q \xrightarrow{\alpha} P' \mid Q, \text{ if } P \xrightarrow{\alpha} P' \text{ and } \text{n}(\alpha) \cap \text{fn}(Q) = \emptyset \\ (\text{New}) \quad & \nu x.P \xrightarrow{\alpha} \nu x.P', \text{ if } P \xrightarrow{\alpha} P' \text{ and } x \notin \text{n}(\alpha) \\ (\text{StructCongr}) \quad & P \xrightarrow{\alpha} P', \text{ if } Q \xrightarrow{\alpha} Q' \text{ and } P \equiv Q \text{ and } P' \equiv Q' \end{aligned}$$

In the following examples, sums and τ -prefixes are also used. The labeled transition system would need to be adjusted accordingly. We omit the formal definition here.

An important statement regarding the transition system is that τ -transitions correspond precisely to reductions:

Lemma 5.2.15 (Harmony Lemma). $P \rightarrow Q$ if and only if $P \xrightarrow{\tau} Q'$ where $Q' \equiv Q$

The so-called *bisimulations* defined for the π -calculus compare the behavior of two processes based on the labels.

5.2.4.1 Strong Bisimulation

Strong bisimulation holds if processes exhibit exactly the same behavior with respect to the labeled transition system:

Definition 5.2.16. A binary relation $\mathcal{R}$ on processes is a strong bisimulation if for all $(P, Q) \in \mathcal{R}$ the following holds

- If $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{R}$
- If $Q \xrightarrow{\alpha} Q'$, then there exists a process P' such that $P \xrightarrow{\alpha} P'$ and $(P', Q') \in \mathcal{R}$.

Two processes P, Q are strongly bisimilar (written $P \sim_{b, \text{strong}} Q$) if there exists a strong bisimulation R with $(P, Q) \in R$. The relation $\sim_{b, \text{strong}}$ is called strong bisimilarity.

Lemma 5.2.17. $\sim_{b, \text{strong}}$ is the largest strong bisimulation.

Proof. The definition of strong bisimilarity implies that $\sim_{b, \text{strong}}$ is the union of all strong bisimulations, i.e., $\sim_{b, \text{strong}} = \bigcup \{ \mathcal{R} \mid \mathcal{R} \text{ is a strong bisimulation} \}$. This immediately implies that the greatest strong bisimulation is contained in $\sim_{b, \text{strong}}$. It remains to be shown that $\sim_{b, \text{strong}}$ itself is a strong bisimulation. However, this follows from the fact that the union $\mathcal{R}_1 \cup \mathcal{R}_2$ of two strong bisimulations $\mathcal{R}_1, \mathcal{R}_2$ is always a strong bisimulation. $\square$

Remark 5.2.18. The following procedure is analogous to the above definition. Define the function on relations of processes that results from the above definition, i.e.,: Let Proc be the set of all processes of the π -calculus. Let $F : (\text{Proc} \times \text{Proc}) \rightarrow (\text{Proc} \times \text{Proc})$ be the following function on binary relations of processes: Let $\eta \subseteq (\text{Proc} \times \text{Proc})$ be a binary relation on processes, then $(P, Q) \in F(\eta)$ if and only if:

- If $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \eta$
- If $Q \xrightarrow{\alpha} Q'$, then there exists a process P' such that $P \xrightarrow{\alpha} P'$ and $(P', Q') \in \eta$.

Strong bisimilarity is then precisely the largest fixed point of the function F . The principle of coinduction then states that every relation η that is dense with respect to F is also contained in the largest fixed point. A relation η is F -dense if $\eta \subseteq F(\eta)$, i.e., the application of F does not reduce the relation. In Definition 5.2.16, the F -dense relations correspond precisely to the relations that are strong bisimulations.

Strong bisimilarity is an equivalence relation, but not a congruence:

Example 5.2.19. Let $P = \bar{z}b.0 \mid a(c).0$ and $Q = \bar{z}b.a(c).0 + a(c).\bar{z}b.0$. Then $P \sim_{b, \text{strong}} Q$, because $\mathcal{R} = \{(P, Q), (a(c).0, a(c).0), (\bar{z}b.0, \bar{z}b.0), (0, 0)\}$ is a strong bisimulation (as can be easily verified). However, $x(z).P \not\sim_{b, \text{strong}} x(z).Q$ (shown below), and therefore $P \sim_{b, \text{strong}} Q \not\Rightarrow C[P] \sim_{b, \text{strong}} C[Q]$ for every context C , i.e., $\sim_{b, \text{strong}}$ is not a congruence.

To show that $x(z).P \not\sim_{b, \text{strong}} x(z).Q$, consider the transition in which the name a is received. Then the following applies: $x(z).P \xrightarrow{x(a)} P[a/z]$ and $x(z).Q \xrightarrow{x(a)} Q[a/z]$. It now suffices to show that $P[a/z] \not\sim_{b, \text{strong}} Q[a/z]$ holds. However, this follows immediately, since $P[a/z] = \bar{a}b.0 \mid a(c).0 \xrightarrow{\tau} 0$, but for $Q[a/z] = \bar{a}b.a(c).0 + a(c).\bar{a}b.0$. no τ -transition is possible.

A meaningful concept of program equality should always be a congruence, since one would also like to apply equality to subprograms. Since strong bisimilarity is not a congruence, it must be rejected as a concept of equality. An improvement is obtained with strong full bisimilarity:

Definition 5.2.20. Two processes P, Q are strongly fully bisimilar (notation $P \sim_{b, \text{strong}, \text{full}} Q$) if, for all substitutions σ that replace names with names, $\sigma(P) \sim_{b, \text{strong}} \sigma(Q)$.

It can be shown that $\sim_{b, \text{strong}, \text{full}}$ is a congruence.

5.2.5 Weak Bisimulation

Strong full bisimulation is often considered too discriminating, i.e., it distinguishes too many processes. For example, the processes $P_1 := \nu x.(x(y).0 \mid \bar{x}z.0)$ and $P_2 := 0$ are not strongly bisimilar, since P_1 can still perform an internal action, but P_2 cannot. On the other hand, this internal behavior is not observable from the outside, and therefore P_1 and P_2 should not necessarily be distinguished.

In order to ignore such differences in internal actions in the equality, the so-called weak bisimulation is defined, which is often simply referred to as bisimulation. Applied to the labels with respect to the transition system, ignoring the internal actions means that τ -transitions play no role for the equality. Let $\xRightarrow{\tau}$ be the reflexive-transitive hull of $\xrightarrow{\tau}$ (i.e., $P \xRightarrow{\tau} Q$ if P can be transferred to Q with 0 or a finite number of τ -transitions) and let $\xRightarrow{\alpha}$ be the relation $\xRightarrow{\tau} \xrightarrow{\alpha} \xRightarrow{\tau}$, i.e., the relation that first executes any number of τ -transitions, then executes the action α and then executes any number of τ -transitions again.

Definition 5.2.21. A binary relation $\mathcal{R}$ on processes is a bisimulation if for all $(P, Q) \in \mathcal{R}$ the following holds

- If $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that $Q \xRightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{R}$
- If $Q \xrightarrow{\alpha} Q'$, then there exists a process P' such that $P \xRightarrow{\alpha} P'$ and $(P', Q') \in \mathcal{R}$.

Two processes P, Q are bisimilar (written $P \sim_b Q$) if there exists a bisimulation $\mathcal{R}$ with $(P, Q) \in \mathcal{R}$. The relation $\sim_b$ is called bisimilarity.

Two processes P, Q are fully bisimilar ($P \sim_{b,full} Q$) if $\sigma(P) \sim_b \sigma(Q)$ holds for all substitutions σ .

While bisimilarity is not congruence (for the processes P, Q from Example 5.2.19, for the (non-strong) bisimilarity $P \sim_b Q$ but $x(z).P \not\sim_b x(z).Q$), full bisimilarity is a congruence.

Obviously, strong (full) bisimilarity implies (full) bisimilarity, but the converse does not hold:

Proposition 5.2.22. For all processes P, Q , the following applies:

- $P \sim_{b,strong} Q \implies P \sim_b Q$
- $P \sim_{b,strong,full} Q \implies P \sim_{b,full} Q$.

Furthermore, $\sim_{b,strong} \neq \sim_b$ and $\sim_{b,strong,full} \neq \sim_{b,full}$.

The proof can be found in the relevant literature.

5.2.5.1 Barbed congruence

For the π -calculus, there is another concept of equality known as *barbed congruence*.

Definition 5.2.23 (Barb). A process P has an input barb on channel x if $P \xrightarrow{x(y)} P'$. In this case, we write $P \Downarrow^x$.

A process P has an output barb at channel x if $P \xrightarrow{\bar{x}y} P'$. In this case, we write $P \Downarrow^{\bar{x}}$.

For $\beta \in \{x, \bar{x}\}$, we write $P \Downarrow_\beta$ if there is a process Q such that $P \xRightarrow{\tau} Q$ and $Q \Downarrow^\beta$. This means that P can perform an input or output action after some τ -actions.

Definition 5.2.24. A binary relation $\mathcal{R}$ on processes is a barbed bisimulation if the following holds for all $(P, Q) \in \mathcal{R}$:

- If $P \Downarrow^x$ (or $P \Downarrow^{\bar{x}}$), then $Q \Downarrow_x$ (or $(Q \Downarrow_{\bar{x}})$)
- If $Q \Downarrow^x$ (or $Q \Downarrow^{\bar{x}}$), then $P \Downarrow_x$ (or $(P \Downarrow_{\bar{x}})$)
- If $P \xrightarrow{\tau} P'$, then there exists Q' with $Q \xRightarrow{\tau} Q'$ and $(P', Q') \in \mathcal{R}$
- If $Q \xrightarrow{\tau} Q'$, then there exists P' with $P \xRightarrow{\tau} P'$ and $(P', Q') \in \mathcal{R}$

Two processes P, Q are barbed bisimilar (written $P \sim_{b, \text{barbed}} Q$) if there exists a barbed bisimulation containing (P, Q) .

Barbed bisimulation is extended to a congruence by the following definition:

Definition 5.2.25. Two processes P, Q are barbed congruent (written $P \sim_{c, b, \text{barbed}} Q$) if for all contexts C : $C[P] \sim_{b, \text{barbed}} C[Q]$.

One can also define barbed congruence without using the labeled transition system, but only using reduction, which follows from the following lemma and the Harmony Lemma:

Lemma 5.2.26. For all processes of the synchronous π -calculus (with sums), the following applies:

- $P \Downarrow^x$ if and only if $P \equiv \nu v_1 \dots \nu v_n. (x(y).P' + M \mid Q)$ where $x \neq v_i$.
- $P \Downarrow^{\bar{x}}$ if and only if $P \equiv \nu v_1 \dots \nu v_n. (\bar{x}y.P' + M \mid Q)$ where $x \neq v_i$.

The barbed congruence contains full bisimilarity:

Theorem 5.2.27. For all processes P, Q , the following holds:

$$P \sim_{b, \text{full}} Q \implies P \sim_{c, b, \text{barbed}} Q$$

It is unknown whether the converse of this implication holds.

5.2.5.2 May- and Must-Testing

We now present another definition of equality that does not use bisimulation, but is based on the observation of barb behavior, replacing the termination test.

Definition 5.2.28. The may-testing preorder $\leq_{\text{may}}$ on processes is defined as:

$P \leq_{\text{may}} Q$ if and only if for all contexts C , for all names x : $C[P] \downarrow_x \implies C[Q] \downarrow_x$ and $C[P] \downarrow_{\bar{x}} \implies C[Q] \downarrow_{\bar{x}}$.

The may-testing equivalence $\sim_{\text{may}}$ is the symmetric closure of the preorder, i.e., $P \sim_{\text{may}} Q$ if and only if $P \leq_{\text{may}} Q$ and $Q \leq_{\text{may}} P$.

The may-testing equivalence is a congruence. It is very coarse-grained, so, for example,

Proposition 5.2.29. For all processes P, Q , the following applies: $P \sim_{c,b,\text{barbed}} Q \implies P \sim_{\text{may}} Q$.

The converse does not hold, since, for example, $a(x).0 \oplus (b(x).0 \oplus c(x).0)$ and $(a(x).0 \oplus b(x).0) \oplus c(x).0$ are may-testing equivalent but not barbed congruent.

In fact, may-testing equivalence is too coarse-grained, since, for example, the processes $x(y).0 \oplus 0$ and $x(y).0$ are may-testing equivalent, even though their behavior is very different: $x(y).0 \oplus 0$ can become a process that no longer has any communication options, while $x(y).0$ can always receive on channel x .

Therefore, the *should-testing* equivalence is also considered. For $\mu \in \{x, \bar{x}\}$, we write $P \Downarrow_\mu$ (P must have a barb μ) if

$$\forall Q : P \xrightarrow{*} Q \implies Q \downarrow_\mu .$$

That is, every reduction successor of P has a barb at μ .

Definition 5.2.30. The should-testing preorder $\leq_{\text{should}}$ on processes is defined as:

$P \leq_{\text{should}} Q$ if and only if for all contexts C , for all names x : $C[P] \Downarrow_x \implies C[Q] \Downarrow_x$ and $C[P] \Downarrow_{\bar{x}} \implies C[Q] \Downarrow_{\bar{x}}$.

The should-testing equivalence $\sim_{\text{should}}$ is the symmetric closure of the preorder, i.e., $P \sim_{\text{should}} Q$ if and only if $P \leq_{\text{should}} Q$ and $Q \leq_{\text{should}} P$.

The should-testing equivalence is finer than the may-testing equivalence, i.e., $\sim_{\text{should}} \subset \sim_{\text{may}}$, but coarser than barbed congruence:

Proposition 5.2.31. For all processes P, Q , the following applies: $P \sim_{c,b,\text{barbed}} Q \implies P \sim_{\text{should}} Q$.

Overall, this results in the following hierarchy of process equivalences:

Theorem 5.2.32. $\sim_{b,\text{strong},\text{full}} \subseteq \sim_{b,\text{full}} \subseteq \sim_{b,c,\text{barbed}} \subseteq \sim_{\text{should}} \subseteq \sim_{\text{may}}$

5.3 References

The standard references for the π -calculus are (Milner, 1999) and (Sangiorgi & Walker, 2001). The encoding of the lambda calculus in the π -calculus is taken from (Milner, 1992). The encoding of the synchronous π -calculus in the asynchronous π -calculus comes from (Boudol, 1992). The equality definitions can essentially be found in (Sangiorgi & Walker, 2001).

Bibliography

- Ben-Ari, M. (2006).** *Principles of concurrent and distributed programming*. Addison-Wesley, Harlow, UK.
- Boudol, G. (1992).** Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA.
- Buhr, P. A., Fortier, M., & Coffin, M. H. (1995).** Monitor classification. *ACM Comput. Surv.*, 27(1):63–107.
- Carriero, N. & Gelernter, D. (1992).** *How to Write Parallel Programs*. MIT Press, Cambridge, MA. <http://www.lindaspaces.com/book/>.
- Church, A. (1941).** *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey.
- Coffman Jr., E. G., Elphick, M. J., & Shoshani, A. (1971).** System deadlocks. *ACM Comput. Surv.*, 3(2):67–78.
- Dice, D., Shalev, O., & Shavit, N. (2006).** Transactional locking ii. In S. Dolev, editor, *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer.
- Dziuma, D., Fatourou, P., & Kanellou, E. (2015).** Consistency for transactional memory computing. In R. Guerraoui & P. Romano, editors, *Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TMIC1001*, volume 8913 of *Lecture Notes in Computer Science*, pages 3–31. Springer.
- Engelfriet, J. & Gelsema, T. (2007).** An exercise in structural congruence. *Inf. Process. Lett.*, 101(1):1–5.
- Fischer, M. J., Lynch, N. A., & Paterson, M. S. (1985).** Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382.
- Gelernter, D. (1985).** Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112.
- Guerraoui, R. & Kapalka, M. (2008).** On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '08*, pages 175–184. ACM, New York, NY, USA.
- Guerraoui, R. & Kapalka, M. (2010).** *Principles of Transactional Memory*. Morgan and Claypool Publishers, 1st edition.
- Herlihy, M. (1991).** Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149.
- Herlihy, M. & Shavit, N. (2008).** *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

- Hoare, C. A. R. (1969).** An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.
- Larus, J. R. & Rajwar, R. (2006).** *Transactional Memory*. Morgan & Claypool.
- Larus, J. R. & Rajwar, R. (2010).** *Transactional Memory, 2nd edition*. Morgan & Claypool.
- Milner, R. (1992).** Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141.
- Milner, R. (1999).** *Communicating and mobile systems: the π -calculus*. Cambridge University Press, New York, NY, USA.
- Palamidessi, C. (1997).** Comparing the expressive power of the synchronous and the asynchronous π -calculus. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–265. ACM, New York, NY, USA.
- Palamidessi, C. (2003).** Comparing the expressive power of the synchronous and asynchronous π -calculi. *Mathematical Structures in Comp. Sci.*, 13(5):685–719.
- Peyton-Jones, S. (2007).** Beautiful concurrency. In A. Oram & G. Wilson, editors, *Beautiful code*. O'Reilly.
- Raynal, M. (2013).** *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer.
- Reppy, J. H. (2007).** *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England.
- Sangiorgi, D. & Walker, D. (2001).** *The π -Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA.
- Schmidt-Schauß, M., Rau, C., & Sabel, D. (2013).** Algorithms for Extended Alpha-Equivalence and Complexity. In F. van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 255–270. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.
- Taubenfeld, G. (2006).** *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- Varacca, D. & Völzer, H. (2006).** New perspectives on fairness. *Bulletin of the EATCS*, 90:90–108.