Introduction to Concurrent Programming

1

1.1 Basic Concepts and Motivation for the Study of Concurrent Programming

A sequential program is a set of data definitions and instructions executed in a single sequence, linear sequence. In contrast, a concurrent program is written in a high-level programming language that specifies two or more independent execution units, which we will call processes. These units cooperate to perform useful work, which is essential for the progress of the program's computations, which is their main task.

A process should be understood as an abstract, dynamic and active software entity. It executes instructions and transitions through different states[1]. It is important to note that a set of instructions by itself cannot be considered a process. Therefore, the older definition of a concurrent process as "a sequential program in execution", found in some old operating system texts, is misleading.

For a computer processor to manage a process, it must have access to the process's state at any given time. A process is not only the independent execution of a sequence of instructions, it also includes its current state and its capacity to interact with the environment of the program. The state of a process integrates a set of values that define it at each point during execution. These values are stored in the processor's registers and in system memory.

To manage a process, the processor needs to know the values of certain processor's registers: the program counter (PC), the stack pointer (SP), and the heap memory (Heap). It also must handle data related to devices accessed by the process, such as files and any

[1] For now, we will understand the concept of program state as the set of values of visible and invisible variables: program counter (PC), stack pointer (SP) at each differentiated moment of our program execution.

M. I. Capel, *Concurrent and Real-time Programming*,
https://doi.org/10.1007/978-3-031-85233-6_1

other resources it owns. These values must be protected from competitive or uncontrolled access by other concurrent processes.

In a concurrent program, there may be many instruction execution sequences with, at least, one independent control flow or execution *thread* for each of the processes running in it.

From the point of view of the operating system, a process is characterised by a memory area divided into several zones (Fig. 1.1). These zones include the sequence of instructions to be executed, a fixed-size data area used by global or static variables, the stack (a variable-size area for local variables of procedure parameters) and a dynamic area called heap memory, which holds non-statically allocated variables (i.e., variables created, allocated and destroyed before the process terminates).

In programs containing many IO-bound or message-receiving instructions, exception handling, signal processing, …, the processor time utilisation of any concurrent program containing such instructions is much better than of an equivalent sequential program, i.e., one that would have been programmed to perform the same tasks or achieve the same results, but with a single execution sequence of instructions.

If the execution platform where the program runs has fewer cores or independent processing units than the number of processes in the program, the control flows of multiple processes are interleaved and form an execution sequence for each core. This interleaving maintains the logical parallelism of the program, regardless of how many processors are executing the code concurrently. As a result, concurrent programs are more efficient than sequential ones because they allow the advancement of multiple threads of control to progress simultaneously. This setup also prevents processes with frequent input and output operations from slowing down other processes that require more computation. In general, the existence of concurrent processes in programs avoids delays in processor execution by

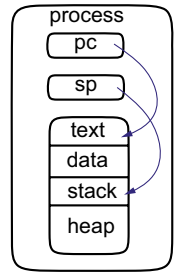

Fig. 1.1 Model of concurrent process

allowing computations to continue while other processes wait for data from the program's environment.

Furthermore, concurrent programs are fundamental for simulation applications. The concurrent programming paradigm represents real-world systems more accurately than sequential programming. Physical systems typically consist of multiple activities running in parallel, and each activity can be simulated more naturally with an independent concurrent process. In contrast, a sequential program, which follows a one-dimensional cycle of polling signals or messages sent by the program's environment, is less representative of real-world systems.

Hence, we use terms like "concurrent" and "concurrency" to describe the potential for parallel execution in any program written in modern programming languages. Concurrency refers to the potential for parallelism within certain code units, algorithms, application or systems—logically independent of the number of processors or cores provided by the system's hardware.

> We can define concurrent programming as the set of programming notations and techniques used to express the potential parallelism of programs and, consequently, to be able to solve the synchronisation and communication problems of the processes that make up such programs.

1.2 Concurrent Programming Abstract Model

Concurrent programming is not just a programming paradigm or model but primarily an abstract model of computation. It expresses the potential parallelism of computer programs at an appropriate level of abstraction[2], independent of how that parallelism is implemented at the architectural level of a computer system.

Good concurrent languages should provide useful primitives to address synchronisation and communication between concurrent processes, allowing the same code to run on different computer architectures. While this is an ideal goal, the abstract model of concurrency we will introduce represents significant progress towards the achievement of this maximal objective. It provides a set of tools that help conceptualise and design concurrent programs, allowing us to reason about solutions to synchronisation and communication problems. This model simplifies programming by enabling the use of high-level instructions for communication and synchronisation, without relying on system calls or machine-level code. In addition, this abstraction ensures that programs will be portable between

[2] A level of abstraction that allows solving problems that are of interest to us in the field of concurrency, without getting bogged down in the details of the implementation of parallelism at the microprocessor level.

P_1	P_2	possible sequences
I_{11}	I_{21}	I_{11} I_{11} I_{11} ...
I_{12}	I_{22}	I_{21} I_{12} I_{21} ...
I_{13}	I_{23}	I_{12} I_{21} I_{22} ...
I_{14}	I_{24}	I_{22} I_{13} I_{23} ...
...	...	I_{13} I_{22} I_{12} ...
...	...	... ...

Fig. 1.2 Interleaving sequences of two-process atomic instructions

architectures. The only requirement is that the appropriate compiler for the concurrent programming language exists for the platform being used.

1.2.1 The Abstract Model of Concurrency

The abstract model of concurrent programming is based on five key principles or axioms:

1. Atomicity and Instruction Interleaving
2. Ensuring Consistency of Program Data After Concurrent Access
3. Unrepeatability of Any Sequence of Atomic Instructions
4. Independence of Relative Speeds of Processes During Program Execution
5. Processes Finite Progress Hypothesis

1.2.1.1 Atomicity and Instruction Interleaving

From a concurrent program written in a compiled programming language, it is possible to derive a corresponding set of instructions at the assembly level, as shown in Fig. 1.2.

This instruction level typically corresponds to the machine or assembly language instruction set. Each instruction is executed atomically, meaning that once it starts, it runs to completion without being interrupted by a context switch or any system-level call. This indivisibility guarantees that concurrent execution, whether achieved through real parallelism[3] or logical parallelism (using time-sharing), does not affect the results of the program; only the performance or speed is impacted.

Figure 1.2 illustrates a possible sequence of instructions generated by a concurrent program P scheduling two processes. Each process's atomic instructions are represented as $\{I_{1x}\}$ and $\{I_{2x}\}$. The set of all possible interleaved sequences of these atomic instructions defines the observable behaviour of the program P. Each execution of P results in one of these sequences that define its behavior[4], but which specific sequence occurs cannot be

[3] True parallelism means simultaneous execution of instructions by a different processor or multiprocessor cores.

[4] The set of all interleaving sequences of atomic instructions that are generated from the program processes

predicted or influenced directly. This fundamental characteristic is often referred to as nondeterminism in concurrent programs.

This model of concurrent program execution describes parallelism at the logical level, independent of the specific hardware architecture on which the program code was ultimately executed.

1.2.1.2 Ensuring Consistency of Program Data After Concurrent Access

The concurrent execution of two single atomic instructions accessing the same memory address must produce the same results, whether both instructions are executed in real parallelism or logical parallelism, where they are executed sequentially, one after the other, but in an arbitrary and unpredictable order. Consistency in data access means that after both processes have completed their operations, the memory representation of the data must remain in a state consistent with the data type to which they belong. Figure 1.3 illustrates that the result could be 1 or 2 (without predicting which), but the value will remain consistent with the variable's definition and its allowed values, as specified in the program.

In Fig. 1.3, since instructions I_1 and I_2 access the location of the processor's accumulator registers sequentially, the final value of variable x is not predetermined. However, it will coincide with the last value assigned, as x is assigned twice by different processes during code execution in an unpredictable order. If the shared variable's memory location were accessed simultaneously without the contention managed by the memory bus arbiter, data could become corrupted, leading to incorrect values in x.

This hypothesis of the abstract model assumes that concurrent memory access will not corrupt data, which is supported by the memory access control hardware in real computer systems. This consistency in data values after simultaneous access to the same address is guaranteed by the memory controller's bus arbiter in computers.

1.2.1.3 Unrepeatability of Any Sequence of Atomic Instructions

The number of possible interleaved sequences of atomic instructions in a concurrent program is extremely large, making it quite unlikely that two successive executions of the program will follow the exact same sequence of atomic instructions. This unpredictability makes debugging and verifying the correctness of concurrent programs very difficult, leading to the emergence of transient errors (errors that occur in some execution sequences but not in others). These errors are difficult to detect and fix in the program code.

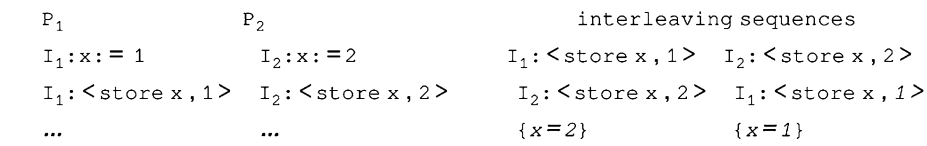

Fig. 1.3 Sequentialisation of the execution of concurrent processes in accessing shared variables by the order imposed by the hardware memory bus controller

To address the tendency of transient errors to arise in concurrent software, more robust methods are required. As we will discuss later, formal methods, with the rigour of mathematical logic, are essential for verifying the correctness of concurrent programs and for identifying and eliminating transient errors that may surreptitiously appear during the execution.

1.2.1.4 Independence of Relative Speeds of Processes During Program Execution

The correctness of concurrent programs must not depend on the relative execution speed of one process compared to others. If this requirement were not met, the following issues could arise when running a concurrent program on different computing platforms:

- Lack of portability, if assumptions about the process execution speed are made, the program may be not portable across platforms. The program might fail to function correctly on platforms with different processor characteristics.
- Race conditions occurrence, erroneous results may occur if processes concurrently access shared variables or code sections. If the correctness of a program depended on process speed, countless transient errors may occur even with slight changes in process context.

Figure 1.4 shows two processes, P_1 and P_2; both access and modify a shared variable `data` (initially 0). Depending on the order and speed of execution, data could end up as either value: `+1, 0, -1`, with no way to predict the final result. If one of these outcomes is incorrect according to the program's specification, we identify this as a race condition between the two processes.

An exception to this model exists in real-time applications, where timing and execution order are critical. In these cases, processes have different priority levels and privileges, which influence their execution speed. These programs, used in safety-critical systems, deviate from the typical abstract model of concurrency.

Lastly, the abstract model of concurrent programming assumes that all processes will complete in finite time. Without this assumption, correctness properties like process liveness cannot be guaranteed, as they would then depend on the execution platform.

1.2.1.5 Processes Finite Progress Hypothesis

To verify the correctness of concurrent programs, all processes must be able to execute and continuously make progress in their computations. This progress can be understood at two levels:

Fig. 1.4 Example of race condition between two processes

```
Process P1:     Process P2:
 a:= data;       b:= data;
 a:= a+1;        b:= b-1;
 data:= a;       data:= b;
```

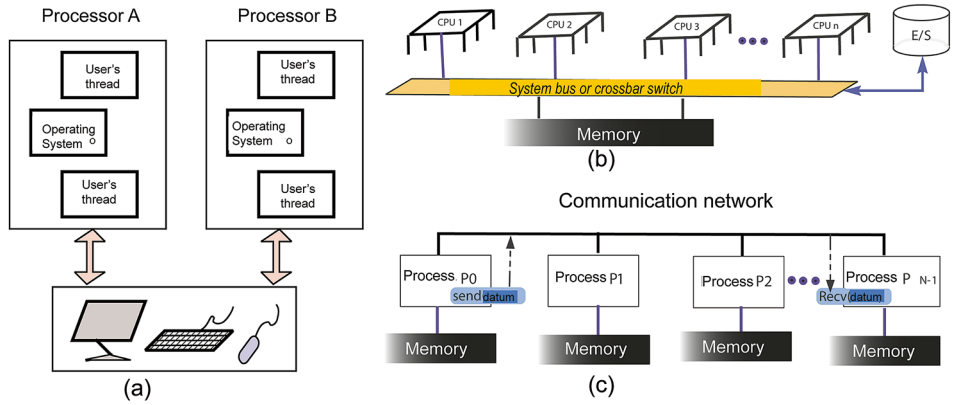

Fig. 1.5 System architectures with different degrees of parallelism: (**a**) multi-core processor (**b**) multiprocessor (**c**) multi-computer

1. Global progress: if at least one process is ready to execute, it must be allowed to run, provided the program has not entered a deadlock situation. In other words, eventually[5], some process will be allowed to execute in a correct concurrent program.
2. Local progress: any process that starts executing a section of code must eventually complete that section.

In summary, we can say that the finite progress hypothesis is defined as follows: there should be no situation during a program's execution where a process arbitrarily stops for an indefinite amount of time. In other words, no hidden condition (such as counters, registers, etc.) should cause a process to halt its progress indefinitely along any sequence of states in the program's execution.

1.2.2 Hardware Considerations

Depending on their computer architecture, computer systems currently implement concurrency according to three general models, which can also be hybridised, as shown in Fig. 1.5.

General multiprocessors capable of parallel computing can be classified according to the number of processors and their distribution: (a) multi-core processors represent a type of architecture in which there are usually many more processes than cores (independent physical execution units) and parallelism has to be modelled by interleaving multiple threads of execution; this model can be understood as a category of systems in which concurrency in the execution of processes can be achieved with or without real parallelism; (b) multiprocessors will always provide real parallelism, and this type of multiprocessor is usually designed to be programmed with languages or libraries that have specific

[5] That is, at some point in the future that cannot be postponed for a long time

instructions that efficiently translate into parallel instructions for a particular multiprocessor architecture, i.e., the program code will execute several elementary instructions simultaneously in different processors, and there will be a common memory through which the processes can communicate which each other at high speed. That is, the program code will simultaneously execute several elementary instructions in different processors. (c) Multicomputers are distributed multiprocessors in which each processor or physical execution unit has an independent memory that is not shared with the others, so that to communicate processes located in different processors, it is necessary to use a network or high-speed interconnection that allows a high level of scalability[6], and they usually have a more complicated programming notation than any of the previously introduced systems.

1.3 Mutual Exclusion and Synchronisation

Mutual exclusion ensures that a set of statements in a process's code, shared with other processes in the program, is executed in an indivisible or atomic manner. In other words, no more than one process in the program is allowed to execute a block of these statements at the same time[7]. A block (or section) of instructions that can only be executed by a single process is called a *critical section*.

Using the appropriate language programming primitives, if two or more processes attempt to execute a critical section, only one will succeed at a time, while the other must wait until that process finishes before trying again.

An example of mutual exclusion access is resource allocation such as file backups, plotters, printers, etc. managed by an operating system for various client processes. Mutual exclusive access to these resources is necessary because if the jobs from different user processes were mixed, the output produced by the shared device would become unusable.

The acquisition and restitution protocols are instruction blocks designed to ensure that the mutual exclusion condition is met whenever a process accesses a critical section. The acquisition protocol decides, in case of contention between multiple processes, which one enters the critical section first; the others must wait until the process that entered first has finished executing CS instructions. The restitution protocol allows a new process, which may have been waiting, to enter the critical section once it becomes free again. The typical structure of a process containing a critical section is as follows.

```
Rest of instructions
Acquisition protocol
<Sentences in Critical Section (CS)>
Restitution protocol.
```

[6] Integration of many new cores/processors depending on the computational power required to execute an algorithm, without loss of performance.

[7] It may not match a program block, as understood in most programming languages.

1.3.1 Synchronisation

In concurrent programs, processes need to communicate during execution to work collaborating on a common task. As mentioned earlier, communication between processes can occur via shared variables or messages, depending on the type of parallelism supported by the platform running the concurrent program. This inter-process communication creates the need for synchronisation between processes during execution.

There are two basic scenarios of synchronisation between processes: (a) Conditional synchronisation, which involves pausing the execution of a process until a specific condition is met. At a lower system level, this means the process cannot be allowed to change its state—i.e., it cannot execute the next instruction—until the condition, which is usually evaluated from the program's variable values, is satisfied. (b) Mutual exclusion, a special synchronisation condition, is crucial when a process must wait to enter a critical section until it becomes available. This prevents simultaneous access to a shared resource, avoiding potential conflicts.

The bounded circular buffer model shown in Fig. 1.6 is a classic example illustrating the difference between the two forms of synchronisation commonly used in concurrent programs. In this model, producer processes insert data into the circular buffer, and consumer processes remove data from it. The circular buffer is designed to decouple the execution of producer and consumer processes, allowing them to run independently without worrying about buffer overflows or trying to retrieve data from an empty buffer.

For example, if several producer processes run first without any consumer process ready to retrieve the data, the produced data is temporarily stored in the buffer, waiting to be consumed when a consumer process executes. In this bounded circular buffer implementation, mutual exclusion synchronisation ensures that no producer and consumer

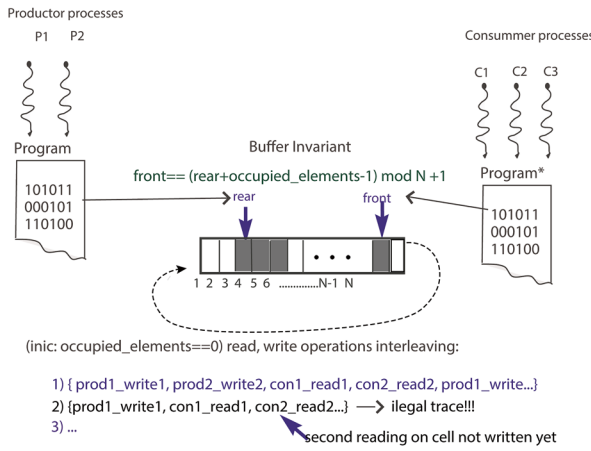

Fig. 1.6 Circular buffer of *N* elements in the concurrent producer-consumer program example

access the buffer at the same time, preventing race conditions that could occur if multiple processes try to access the same buffer element simultaneously[8].

Conditional synchronisation in this scenario ensures that a message inserted into the buffer is not overwritten before being extracted and that data are not inserted into a full buffer or extracted from an empty one. These are examples of transient errors that could occur, such as a consumer process attempting to retrieve data twice after a producer has inserted only one element into an empty buffer, as shown in the sequence labelled "illegal" in Fig. 1.6.

From the point of view of the abstract model of concurrent programming, the role of synchronisation is to limit the possible execution sequences to only those that are considered correct, ensuring the fulfilment of the concurrency properties required by the program.

1.3.2 Process Creation

Early notations in imperative programming languages for expressing the execution of concurrent processes in a program did not include independent process creation and synchronisation primitives. Instead, process creation operation itself implicitly synchronised the program with the termination of processes being created. Modern languages and systems with concurrent programming facilities separate the two concepts and impose a structure to the program and their processes.

The first step in separating process creation was the introduction of a process declaration primitive as a distinct program block within a program. This allowed developers to explicitly declare routines to be executed concurrently, while synchronisation between processes could later be handled explicitly with specific primitives in the rest of the program code.

Process declaration in concurrent programming can be categorised based on their duration during program execution: (a) static process declaration, when a fixed number of processes are declared at the beginning of the program and are activated when the program starts; (b) dynamic process declaration, when a variable number of processes can be declared and activated at any point during program's execution. In this case, it is also possible to eliminate processes that are no longer needed, to save memory and other resources.

The following are examples of programming languages that use one of the above mechanisms, MPI (Message Passing Interface) [42], which defines groups of connected processes within each session[9]. Occam [28] for transputer programming and Concurrent Pascal [8] and Modula [47] use static process creation. The most widely used language that supports dynamic process creation is Ada [5].

[8] If such a race condition were to occur with a buffer containing complex elements, a partially written message could be read from the buffer.

[9] Each communicator in MPI provides each process in the declared group with an independent identifier that identifies it within an ordered connection topology.

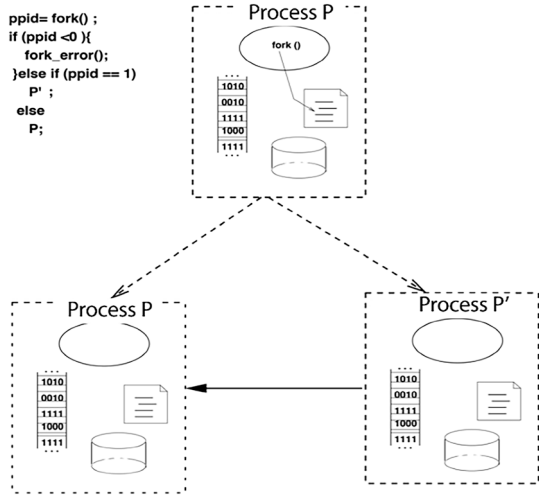

Fig. 1.7 Result of executing one `fork()` operation in UNIX

1.3.2.1 Process Creation by Branching

UNIX operating systems and their variants use an unstructured from of process creation that relies on two basic instructions: `fork()` and `join()`.

The `fork()` command creates a new process by duplicating the calling process's control flow. This results in the execution of a new, concurrent process, running independently of the original program. As shown in Fig. 1.7, when `fork()` is called, a second copy of the program (P') is created, which is completely independent of the original program (P) that initiated the execution. Both processes (P and P') continue to run concurrently, with their instructions interleaved.

The `join()` command, on the other hand, allows a process P to wait for another process (P') to complete its execution before proceeding. In this case, P is the process calling the `join()` operation, and P' is the target process. Once the `join()` call finishes, we can be certain that the target process has safely terminated.

The semantics of the `join()` operation establishes that it is a null operation if the target process (P') has already completed when the `join()` is called. This means no unnecessary processor cycles are consumed and no context switches occur in such cases. The advantage of the fork/join mechanism, which branches off a control flow and later waits for it to finish, is that the process calling `join()` is suspended and does not consume CPU cycles until the target process (P') finishes. This is in contrast to a busy wait loop, where the process would consume cycles unnecessarily while waiting for P' to finish.

The main advantages of process creation by branching over other mechanisms are that this form is practical and powerful, allowing dynamic, unstructured and very flexible creation of concurrent processes. However, the main drawback is that this unstructured

approach can make programs harder to understand. Since `fork()` and `join()` can appear in loops, conditionals or recursive functions, the resulting code can become difficult to debug and verify.

1.3.2.2 Structured Process Creation

The pair of process creation instructions, `cobegin`, `coend`, found in some concurrent languages is considered a structured statement because it clearly defines the start and end of concurrent execution. The `cobegin` instruction initiates the concurrent execution of the following instructions, functions, processes, etc. The block concludes with the `coend` instruction, which ensures that all concurrent operations within the block have completed before proceeding to the next statement in the program.

The statement following the `coend` statement in the previous program fragment will not be executed until all the component statements (S_1;S_2;…S_n), which interleave their instructions in an indeterminate command, have completed execution. Thus, the block of code initiated by `cobegin` has a single termination point, where all the concurrent control flows are merged back into one, resuming the control flow that existed before the `cobegin` instruction.

$$\texttt{cobegin}\ S_1;S_2;\ldots S_n\ \texttt{coend}$$

The block of code defined between `cobegin` and `coend` is part of the overall program execution and depends on the completion of all concurrent statements within it. As a result, this construct ensures that the code block behaves like a traditional structured block, with a single entry and exit point, thereby maintaining the structure and clarity of the program.

1.3.2.3 POSIX Threads Creation

The independence between the memory areas of processes in UNIX provides implicit protection in the access to program variables, but it is not very flexible when we need to use it to define synchronisation mechanisms in cooperative interactions due to the following drawbacks: (a) the time cost and resource consumption required to perform context switching between multiple UNIX processes is high, (b) the system scheduler can only efficiently manage up to a certain number of processes, (c) the system operations associated with this type of process creation are too slow to use shared synchronisation variables (locks, semaphores, etc.) (d) in addition, the `fork()` statement is an unstructured process creation mechanism, which usually leads to programming errors, as it is sometimes not clear which instance or realisation of the original process is being executed at any given time[10].

[10] To distinguish between the instances of the same process, the `fork()` function returns the value 0 if the code being executed at that moment is that of the parent process (the one started the "branching").

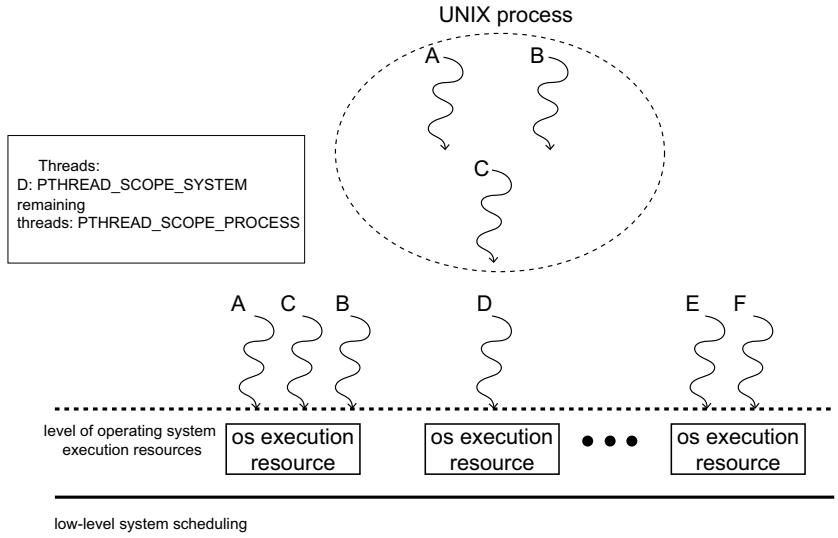

Fig. 1.8 Representation of POSIX 1003.1c thread scheduling (part of POSIX thread package)

One possible solution would be to abandon the protection afforded by the separate address spaces assumed when programming directly with UNIX processes. This leads to the concept of a separate control thread within a process as a scheduling unit. POSIX 1003.1c compliant operating systems—such as Linux—schedule processes and threads (see Fig. 1.8). A process can now contain one or more threads. We can therefore think of a thread as a control flow that shares the text of the process to which it belongs with other threads. Each thread has its own stack, which is initially empty, where it stores its local variables, and shares with other threads the data area, where the global variables and the process heap are located.

1.3.2.4 POSIX Threads Creation Function

Each process has a text or program associated with it, which initially exists as a single thread executing the `main()` function in C/C++ programming languages. As the program progresses, the thread that originally executed `main()` can create one or more POSIX threads associated with the same process. A thread that creates another thread specifies a function `f()` for the latter to execute and continues its own execution after the new thread is started. The latter will run concurrently with the rest of the program threads until the program terminates itself or one of the thread termination conditions included in the POSIX specification is met. Using the POSIX 1003.1c interface, the function that creates and initiates a new thread is as follows.

```
int pthr_create(pthread_t *new_thread, const pthread_attr_t *atr,
                void *(*function_name)(void *), void *args);
```

The parameters of the above function, in order of appearance from left to right, are as follows:

(a) `pthread_t` is an opaque type that acts as a handle to the new thread, i.e., this handle will be used as a reference when the standard operations of the newly created thread need to be called.
(b) `pthread_attr_t` is a creation attribute that is associated with specific functions that modify the internal data structure that stores the initial creation characteristics, including the size of the stack to store thread variables, the policy with which it is scheduled (real time, shared time), whether another thread can wait for it to finish by calling `pthread_join()` or whether it is a system (independent) thread (see Table 1.1).
(c) `void*(*function_name)(void*)` must be replaced with the name of the function to be executed by the thread.
(d) `void *args` is a pointer to the beginning of the list of arguments of the function to be executed by the thread and can be replaced with NULL if it has no arguments.

The causes of termination of a created thread include (a) in the case of threads other than the initial one, when the function specified as the third argument to `pthread_create()` finishes its execution; (b) the created thread calls `pthread_exit()` in its code; (c) its execution is terminated by another thread of the program, which calls `pthread_cancel()`, specifying its identifier; (d) the same UNIX process in whose scope the thread was created terminates by calling `exit()`; and (e) the thread associated with the `main()` function terminates its execution without calling `pthread_exit()`, which would cause it to terminate as another thread of the program.

Example 1.1 As an example of POSIX thread creation, we will implement a simple program that functions as an alarm scheduler, which users of the system can use as a basic agenda. The user interface in this example is minimal, supporting only textual input. The functionality of the program can be summarised as follows:

1. User requests are continuously accepted in a loop.
2. A full line of text is entered for each such request, until either an error is detected or the end of the file is reached in `stdin`.
3. Each input line starts with a number that specifies the number of seconds to wait before displaying the warning message.
4. The rest of the line (up to 64 characters) contains the warning message.

The program is structured into several sections or phases. In the first phase, the following declarations are made:

- Importing necessary libraries
- Declaring a global data type (`control_package`) to store the suspension time (in seconds) and the warning message
- Declaring resources, such as the file pointer (`fp`) where warnings will be logged

```
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

typedef struct control_package{
  int seconds;
  char message[64];
  } control_package_t;

FILE *fp;
```

The `warning()` function contains the code common to all the threads in the program. Each thread calling this function independently suspends itself for the number of seconds specified in its control packet. Once it resumes, the thread prints the string with the user's warning message. The `warning()` function, executed by each thread to handle a warning request, will be programmed as follows:

```
void *warning(void *arg){
        control_package *pct= (control_package_t *) arg;
        int status;
        status= pthread_detach(pthread_self());
        if (status != 0)
            fprintf(stderr, "warning thread\n"), exit(0);
        sleep(pct->seconds);
        fprintf(fp, "(%d)%s\n", pct->seconds, pct->message);
        free(pct);
        return NULL;
}
```

Each thread passes its argument as a pointer to a `control_package_t` structure. By calling `pthread_detach()`, the thread becomes independent of the main thread—meaning, it can use resources independently and continue running even after the main thread terminates. However, the UNIX process containing the thread cannot exit or abort while the thread is active. The `sleep()` function suspends the thread for the number of seconds specified by the user. Once the thread resumes, it logs the suspension time and the warning message to the `fp` file. Afterwards, it releases the dynamically allocated memory for the `control_package` structure.

```
while (1) {
  printf("Warning demand> ");
  if (fgets(line, sizeof(line), stdin) == NULL)
    exit(0);
  if (strlen(line) <= 1) continue;
  int seconds;
  char message[65];
  if (sscanf(line, "%d%64[^\n]", &seconds, message) < 2) {
    if (seconds == -1)
      break;
    else {
      fprintf(stderr, "Bogus input\n");
      continue;
    }
  }
  control_package_t* pcontrol = malloc(sizeof(control_package_t));
  if (pcontrol == NULL) {
    fprintf(stderr, "assign memory to pcontrol\n");
    exit(0);
  }
  pcontrol->seconds = seconds;
  strcpy(pcontrol->message, message);
  status = pthread_create(&thread, NULL, warning, pcontrol);
  if (status != 0) {
    fprintf(stderr, "thread creation warning\n");
    exit(0);
  }
}
```

All the threads created in this program share the same memory address space as the UNIX process running them. The `malloc()` function call creates a dynamic structure of type `control_package_t` containing the values of the timeout and warning message associated with each new user request. A pointer to such a structure is passed as the fourth parameter in the call to `pthread_create()` within the `main()` function. Since the program does not need the main thread to wait for all the warning threads to finish, the threads are created with the detached (non-joinable) thread option. This ensures that the resources used by each thread are automatically returned to the system once the program terminates.

The main thread serves as the execution backbone of the program and runs the `main()` function, which includes the `pthread_create()` call to start the execution of an independent thread in each iteration of an infinite loop. The loop continues until the user enters a blank line or specifies '−1' as the suspension time, which is interpreted as the command to terminate the program. An empty line causes the program to exit by calling `exit(0)`, without waiting for the termination of a warning thread.

1.4 Low-Level Synchronisation Mechanisms in Shared Memory

However, to implement synchronisation mechanisms, some high-level concurrent programming languages use very-low-level instructions that exploit the synchronous nature of the hardware. These mechanisms basically consist of making a concurrent process non-interruptible by using an assembler instruction or by exploiting the fact that the memory hardware can only serve one request at a time, thus sequencing access to certain program

variables, which could lead to race conditions between concurrent processes, which are generally undesirable.

1.4.1 Disable Interruptions

In this case, a machine instruction will be used to disable system interrupts[11]. New interrupts are delayed until the active process executes an instruction that re-enables them. If a process succeeds in disabling interrupts before executing a critical section, then it is safe to execute its instructions in a fully exclusive manner. However, other processes that may be critical to the proper functioning of the system would stop executing until interrupts are re-enabled.

The only advantage of this method is that it is very fast, since only one machine-level instruction is needed to turn it on. However, it has the following disadvantages, some of which are unacceptable in system programming:

(a) Once interrupts are disallowed, real-time events cannot be handled, e.g., hardware devices that require a lot of service and update their state with high temporal frequency; long critical sections make it very difficult to achieve good performance in concurrent scheduling if we use this method.
(b) It excludes non-conflicting activities from interleaving their instructions.
(c) When critical sections are executed, interrupts are prohibited.
(d) If critical sections are executed without interrupts, then no clock-dependent instructions or tasks can be scheduled within them.
(e) It has problems of violating the concurrent safety property if nesting of such critical sections were allowed.
(f) Deadlocks can occur if the instruction to re-enable interrupts on exiting a critical section is not scheduled.

1.4.2 Locks

They are based on an explicit memory synchronisation instruction called `test_and_set(TST)`. This function performs two operations as a single atomic operation, i.e., it cannot be interrupted by interleaving instructions until it is finished: First,

[11] Obviously, we are mainly referring to the interruption caused by the rescheduling of concurrent processes, for example, by round-robin, since hardware interrupts cannot be inhibited without stopping the computer from running.

reads the value of the synchronisation variable. Then, it sets a value for this variable. Then it gives you the value that it read in the first operation it did on the synchronisation variable.

Therefore, since it is not allowed, during the execution of the atomic function `test_and_set()`, to execute any instruction of another process in the middle of the execution of the two previous operations, a process can read and modify the value of a synchronisation variable which would force the rest of the processes to perform busy waiting[12] until it returns to its initial value.

The use of locks in concurrent programs prevents the occurrence of race conditions between processes accessing shared data, although they can cause an indefinite busy wait for threads waiting to read a particular value of the synchronisation variable, which is favourable for them to continue executing the rest of their instructions.

In order to use the locks in synchronisation efficiently and safely, the `test_and_set()` operation must be an operation belonging to the repertoire of low-level instructions of the platform on which the program is executed. However, for didactic purposes, to solve synchronisation problems, without going into unnecessary detail, we could write it as a function of any algorithmic programming language:

```
atomic function test_and_set(var c:boolean): boolean;
 begin
  test_and_set:= c;
  c:= true;
 end;
```

The variables of type Lock take only two values, so in the simulation that we are doing with a high-level function, it is practical to declare them of type Boolean. The use of the mechanism that provides us with the synchronisation variables associated with the locks to solve the problem of access in mutual exclusion of a group of concurrent processes to a critical part of the program requires the programming of two protocols of acquisition and restitution, respectively:

[12] Iterating in an empty loop until the value of the variable changes. It is called busy waiting because it consumes cycles by iterating until the condition of the loop changes.

```
type sync_variable= boolean;
var c: sync_variable:= false;
...
-- Adquisition
procedure SectionBegins(var c: sync_variable);
 begin
  while test_and_set(c) do null;
 end;
-- Critical section of the program to be protected
-- Restitution
procedure SectionEnds(var c: sync_variable);
 begin
  c:= false;
 end;
```

Since each critical section contains a set of shared variables that need to be protected, an independent synchronisation variable is associated with each of these groups. In this way, groups of processes acquiring different locks can interleave their instructions without having to wait for each other.

1.4.2.1 POSIX Locks 1003

If the POSIX interface is used, locks are usually allocated memory either statically as in C programming language:

```
pthread_mutex_t lock;
...
```

Either dynamically by calling the function `malloc()`:

```
pthread_mutex_t *mp;
...
mp=(pthread_mutex_t*)malloc(sizeof(pthread_mutex_t));
...
```

Two methods are also used for its initialisation. The first one consists of using an initialiser, which in old POSIX versions only guarantees its operation with statically assigned variables. This restriction has been removed in the latest version of the POSIX standard and can now also be used when the variable is an automatic variable defined in the body of a function:

```
pthread_mutex_t lock= PTHREAD_MUTEX_INITIALIZER;
```

The NULL value will initialise the lock with the default values of its attributes, and the initialisation of this type of variable must be unique for each execution of a program. The second method uses an `init()` function from the `pthread` library:

```
pthread_mutex_init(&lock, &atr);
```

and is used when we need special features of the mutex construction, such as being able to use it recursively or being able to share it between different processes, not just between threads.

The static variant is usually preferable if the conditions of the program allow it; it allows us to write sections of code that are executed at the start of our program much more easily. It is also true that if we enter program's code during its execution that uses `mutex_init()`, we can be sure that the mutex has always been initialised; this is a very useful safety mechanism in a multi-threaded context.

1.4.2.2 Properties of Programs That Use Locks

Locks are an easy mechanism to implement and allow easy verification of programs written with them. It is a synchronisation mechanism that can be used on multiprocessors with shared memory. Unlike disabling the interrupt mechanism, lock operations do not prevent processes from accessing critical unrelated sections at the same time. This means that using this synchronisation primitive in the case of single processors, processes accessing different critical sections can freely interleave their instructions unless other synchronisation instructions, such as semaphore operations, are explicitly programmed in their code. If there are several critical sections in a program, a lock is declared for each of them, in an attempt to encourage as many interleaving as possible and thus optimise the concurrency of the processes' execution.

The most important drawback of locks is that some of the concurrent processes of the program may be marginalised or *starved*, which fail to execute useful instructions because they never manage to read a value of the synchronisation variable that is favourable to them in order to advance in the execution of their calculations and have to do busy wait for an indefinite time, as they might be continuously waiting to acquire the lock. If there are several processes waiting for the synchronisation variable of the lock to change value, it is not possible to know which of them gets access and acquires the lock first.

In summary, scheduling concurrent programs with multiple processes using locks will not be very efficient due to the waiting time wasted by processes that do not yet have the lock that will allow them to progress in their computations. In addition, if multiple locks are needed in a program, they would have to be acquired and released in reverse hierarchical order; otherwise, deadlocks may occur.

Example 1.2 As an example of programming with POSIX locks, we will introduce an improvement of the program in the previous Example 1.1, which created a prompt thread for each user request, containing a timeout and the label of the prompt to write to a file. In this new, more efficient version of the program, a single server thread is used to extract the first item from the list of requests. The main thread, whose operation is obtained from the `main()` function of the program, inserts new requests into the above list, but sorted by the shortest time remaining until the prompt is printed. The request list must be protected by a `list_mutex` lock, and the server thread will pause for at least 1 s in each iteration to ensure that the main thread has a chance to acquire the lock and insert a new request from the client into the request list. Unlike the first declaration section in Example 1.1, the `control_package` structure now contains `time_t`, whose value is the absolute time in seconds since the start of the UNIX epoch at which each request occurs. This allows requests to be ordered globally according to a single timescale. The `seconds` member of the `control_package` structure only stores the warning suspension time from the time the user's request occurred.

```
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <time.h>

Typedef struct control_package{
   struct control_package* link;
   time_t time;
   int seconds;
   char message[64];
   } control_package_t;

pthread_mutex_t list_mutex= PTHREAD_MUTEX_INITIALIZER;
control_package_t *list_pet = NULL;

FILE *fp;
```

It would not be sufficient to obtain a complete ordering of the warning requests to store only the seconds until the warning expires, since the server thread would not be able to determine how long each request has remained in the list, and therefore the warnings would not occur in a timely manner according to the stored request.

The behaviour of the specialised server thread is determined by the code of the following function:

```
void *server(void *arg){
  control_package_t *pcontrol;
  int sleep_time;
  int status;
  time_t now;
  while (1) {
    status= pthread_mutex_lock(&list_mutex);
    if (status !=0)
      fprintf(stderr,"Error when locking list_mutex\n"),exit(0);
    pcontrol = (control_package__t*)list_pet;
    if (pcontrol==NULL) sleep_time = 1;
      else{
        list_pet= pcontrol->link;
        now= time(NULL);
        if (pcontrol->time<= now) sleep_time=0;
           else sleep_time = pcontrol->time-now;
        }
    status= pthread_mutex_unlock(&list_mutex);
    if (status !=0)
      fprintf(stderr, "error at unlocking mutex"),exit(0);
    if (sleep_time >0) sleep (sleep_time);
      else sched_yield();
    if (pcontrol != NULL){
       fprintf(fp,"(%d)%s{%ld}\n",pcontrol->seconds, pcontrol->message,
       pcontrol->time);
       free(pcontrol);
     }
  }
}
```

If the list of requests accessed by the `pcontrol` reference is not empty, the server thread extracts its first element and determines the time remaining for the warning to expire. If the warning time has expired, i.e., the expression `pcontrol->time<= now` evaluates to true, then `sleep_time` is set to 0; otherwise, the number of seconds the current warning must wait before being printed must be calculated, and this value is also set to `sleep_time`.

The purpose of calling the `sched_yield()` function of the POSIX thread interface when `sleep_time` is 0 is to give the main thread an opportunity to run and check if it has a pending user entry to add to the list. On the other hand, if `sleep_time` is positive, i.e., the warning time has not yet expired, then the server thread is suspended for the time remaining until that moment. Finally, note that the server thread releases the `list_mutex` lock by calling the function `pthread_mutex_unlock(&list_mutex)`, before suspending.

In order for the main thread to correctly add a new user request to the list, it must have mutually exclusive access to the list of requests, which is guaranteed by having previously acquired the lock by executing

```
pthread_mutex_lock (&list_mutex)
```

The main code of the alarm program with the order list by shorter times is the following:

```
int main (int argc, char *argv[]){
  int status; char line[128];
  control_package_t*pcontrol,**ult,*sig;
  pthread_t thread;
  if ((fp= fopen("outputs", "w"))==NULL)
    fprintf(stderr,"Error in the outputfile\n"),exit(0);
  status=pthread_create(&thread,NULL,server,NULL);
  if (status != 0)
    fprintf(stderr,"Error at thread creation"),exit(0);
  while (1){
    printf("Warning petition> ");
    if(fgets(line,sizeof(line),stdin)==NULL) exit (0);
    if (strlen(line) <= 1) continue;
    pcontrol=(control_package_t*)malloc(sizeof(control_package_t));
    if(pcontrol== NULL)
      fprintf(stderr,"by assigning memory to control packet"), exit(0);
    if(sscanf(line,"%d%64[^\n]", &pcontrol->seconds, pcontrol->message)<2){
      if (pcontrol->seconds==-1) break;
         else{printf(stderr,"Bogus input\n");
      free(pcontrol);}
      }
      else{
        pthread_mutex_lock(&list_mutex);
        if (status !=0)
          fprintf(stderr,"Error at lockin mutex"),exit(0);
          pcontrol->time=time(NULL)+pcontrol->seconds;
          ult=  &list_pet;
          sig= *ult;
          while (sig != NULL){
            if (sig->time >= pcontrol->time){
              pcontrol->link = sig; *ult = pcontrol;
            }
          ult = &sig->link;
          sig= sig->link;
      }
      if (sig == NULL) {
        *ult= pcontrol; pcontrol->link= NULL;}
        status= pthread_mutex_unlock(&list_mutex);
        if (status != 0)
          fprintf(stderr,"Error unlocking mutex"),exit(0);
         }
      }
  }
}
```

The above code corresponds to the main thread whose task, in addition to creating the server thread by calling the function `pthread_create(…)`, is to calculate the absolute time at which the warning will be printed by obtaining the current UNIX epoch time with the `time(NULL)` function and updating the value stored in the time member of the structure control package and then to insert a new `pcontrol` structure in the request list with the user's request data before the first element of `list_pet` that has a warning time not shorter than that of this new request. If the entire request list is traversed and no element with a warning time that satisfies the above condition is found, then it is inserted at the last position in the list. Finally, the main thread releases the lock providing mutually exclusive access to the request list.

1.4.3 Semaphores

The semaphore is an abstract data type proposed by Edsger Dijkstra in 1968 for the structured development of multi-user operating systems. Since then, it has been used in concurrent programming as a powerful process synchronisation primitive. Semaphores are usually system primitives designed to synchronise processes through shared memory and are usable in certain high-level programming languages; consequently, if several semaphores are needed in a program, an equivalent number of variables of type semaphore must be declared.

Since mutual exclusion access to a critical section is a special case of synchronisation between processes, semaphore variables can be used both to define critical sections in the programs and to synchronise the concurrent processes that we may have defined. In addition, semaphores introduce only the synchronisation strictly necessary to achieve the maximum interleaving[13] of atomic instructions of a concurrent program; for example, processes using different semaphores can run in parallel, since there would be no synchronisation constraint between these processes at all. Critical sections can therefore be easily nested by simply using different semaphores to access them one after the other. There should also be no possibility of blocking if care is taken not to create permutations when programming the calls to the semaphore operations of the critical sections nested in this code.

1.4.3.1 Definition and Operations on a Semaphore

From an algorithmic perspective, semaphores can also be understood as an abstract data type (ADT), which has only non-negative values defined and for which only three operations are defined:

initialisation:
To be executed only once at the beginning of the program; values must be non-negative.
`wait(s)` or *P(*`s`*)*:
If `s > 0`, then `s := s - 1`
Otherwise, exclusively block the process in a queue associated to the semaphore variable `s`.
`signal(s)` or *V(*`s`*)*:
Check if there are blocked processes.
If there are, then resume one (not necessarily the one that has been blocked the longest) otherwise assign the semaphore variable: `s := s + 1`

[13] That is, they exclude the smallest possible number of interleaving sequences of atomic instructions from the processes; only those that violate the synchronisation condition expressed by the semaphore invariant, which must always be met, are excluded.

There are no further operations defined on semaphore variables, for example, it is not legal to check the value of the semaphore's protected variable. Semaphore operations are usually implemented in the operating system kernel and are provided as other low-level system calls. Unnecessary execution of semaphore operations should be avoided, e.g., executing the operation `signal(s)` when the queue of the semaphore variable `s` is empty, since like any other operating system call, it will cause a processor context switch, which has a cost in terms of performance and final program execution time.

```
void* p1(void *arg){
  while (1){
   wait(s1);
   s= s+1;
   if (s mod 5==0)
     signal(s2)
  else signal(s1)
  }
}
```

```
void* p2(void *arg){
  int s0;
  while (1){
   wait(N);
    wait(s2);
     sum= sum + s;
     s0= s;
    signal(s1);
   write(s0);
  }
  write(sum);
}
```

```
int main (int argc, char *argv[]){
  Semaphore s1, s2, N;
  int s, sum, status;
  pthread_t h1,h2;
  s=0;
  sum= 0;
  init(s1, 1); init(s2, 0); init(N,10);
  pthread_create(&h1,NULL,p1,NULL);
  pthread_create(&h2,NULL,p2,NULL);
}
```

In the earlier example, we want to calculate the series of the cumulative sums of the first ten multiples of 5, for which two operations are used: the first calculates the sequence of the natural numbers, and the second writes the multiples of 5 and writes the sum of the multiples found so far. It stops at the tenth multiple of 5.

1.4.3.2 Atomicity of Operations and Types of Semaphores

The operations on the synchronisation variable are performed atomically, because if the instructions of several operations on the same semaphore variable were allowed to interleave, the final value of the protected variable `s` would be unpredictable, and neither the safety properties nor the liveliness properties of the program using them would be guaranteed.

The semaphore variables used in concurrent programs are classified according to the set of values that the non-negative[14] variable of synchronisation `s` can take, thus obtaining two types of semaphore, depending on the data they can take. Each type of semaphore has different characteristics and is chosen according to what it is used for in the program code. Therefore, according to the range of values that the synchronisation variable of the semaphore can take, we will have (a) binary semaphores, where the synchronisation variable will only take the values 0 and 1, and (b) general semaphores: in this case, the synchronisation variable can take any non-negative integer value.

Another way of obtaining the same classification of types of scheduling semaphores would have been to classify them according to their use:

(a) Mutual exclusion: they are so-called because they are generally used to ensure that a section of program code is executed as a critical section; this type is equivalent to binary semaphores, the synchronisation variable is initialised to the value 1 and the second process that tries to execute the `wait(s)` operation will be blocked, because it will find the protected variable s with the value 0.
(b) General synchronisation: the semaphore variable s is usually initialised to the value 0, the first process attempting to execute the `wait(s)` operation is blocked and the value of the variable s could reach arbitrarily large non-negative values if the `signal(s)` operation is executed many times.

1.4.3.3 Properties of Programs That Use Semaphores as a Synchronisation Primitives for Concurrent Processes

Semaphores prevent processes from performing busy waiting by locking processes in a queue separate from the queue of processes subject to scheduling by the processor and therefore do not consume processor cycles while they are suspended. If multiple processes are waiting for a particular condition to occur during program execution, they are blocked until the program state changes and the condition is satisfied. The processes check for the satisfaction of such a synchronisation condition, and if it is not satisfied, they would execute the `wait(s)` operation of the synchronisation semaphore "`s`", declared as a global variable, to block and enter the queue of such a semaphore.

With the above informal semantics of the semaphores, it can be understood that nothing prevents the processes of a concurrent program from simultaneously executing operations of different semaphores. However, if several semaphores are declared in a concurrent program, they must be used hierarchically, i.e., following the same order of resource acquisition (`wait(s)` operation) and resource release (`signal(s)` operation); otherwise, deadlocks may occur, as in the following example.

[14] "Non-negative" means that the protected variable s of a semaphore is of type integer and can never take negative values.

```
void* p1(void *arg){
wait(s1);
 wait(s2);
-- rest of instructions
}

void* p2(void *arg){
wait(s2);
wait(s1);
-- rest of instructions
}

void* p3(void *arg){
wait(s);
-- rest of instructions
}

void* p4(void *arg){
wait(s);
wait(s1);
-- rest of instructions
}

Main program
int main (int argc, char *argv[]){
  Semaphore s1, s2, s;
  pthread_t h1, h2; srand (time(NULL));
  if (rand()%2){
    init(s1, 1); init(s2, 1);
    pthread_create(&h1,NULL,p1,NULL);
    pthread_create(&h2,NULL,p2,NULL);
    --program in deadlock if processes p1 and p2 are executed as corrutines
   }
   else{
     init(s1, 0);init(s, 1);
     pthread_create(&h1,NULL,p3,NULL);
     pthread_create(&h2,NULL,p4,NULL);
     --program in deadlock if p4 starts before p3
   }
}
```

The main disadvantage of using semaphores as synchronisation primitives in concurrent programming languages is that they are difficult to program correctly because, being global objects of the programs, the execution of operations can affect the value of the semaphore protected variable `s` in other blocks of the program, thus causing unforeseen locks in concurrent processes.

Programming with semaphores generally does not follow the modern principles of structured programming. Moreover, their indiscriminate use can lead to process starvation, since the `signal(s)` operation does not ensure which process in the `s` queue will be released first, since the queue associated with each semaphore need not have a FIFO queue structure.

1.4.3.4 Semaphores POSIX 1003

Semaphores are an optional synchronisation facility in the POSIX thread programming interface. Not all implementations of the thread interface for concurrent programming will offer the ability to program with semaphores. To verify this, just check whether the `_POSIX_SEMAPHORES` constant is defined in the `<unistd.h>` file of your Linux distribution. The semaphores belong to the old P1003.b standard, rather than the new P1003.1c which defines pthreads, so their programming interface has a slightly different style than the other POSIX synchronisation variables.

The semaphores defined in the POSIX standard can be of two types: (a) unnamed, which are like other thread-like synchronisation variables, and (b) named, which are associated with a globally known string in the system that is similar to a UNIX path name. Unnamed semaphores are stored directly in memory, while named semaphores are accessed using string identifiers and, are allocated memory within the address space of a process and must be initialised with a specific operation.

They could be used by more than one process, but this would have to be indicated by assigning certain values to the arguments of the initialisation operation. Named semaphores are always shared by multiple processes, have an identifier known to the applications using them, a group identifier, and UNIX permission-based protection, just like regular system files. To be portable between POSIX-compliant systems, the name of a semaphore must begin with a slash (/), contain no other slashes, and may include letters, digits, and underscores. Note that since the namespace is shared by all processes in the system, if a number of processes use the same name, then we will get the same semaphore. Both types of semaphore (unnamed and named) are represented by the type `sem_t`, and all operations and other semaphore entities are defined in the header file `<semaphore.h>`.

1.4.3.5 Initialisation Operations of POSIX Semaphores

Unnamed semaphores are initialised by the operation:

```
int sem_init(sem_t*semaphore,int pshared,
             unsigned int counter)
```

The initial value of the semaphore is assigned to `counter`. If `pshared` is non-zero, then the semaphore can be used by threads residing in different processes; otherwise, it can only be used by threads within the address space of a single process. An unnamed semaphore can be destroyed by the operation:

```
int sem_destroy(sem_t* semaphore)
```

To be properly destroyed, the semaphore must have been explicitly initialised with the `sem_init(...)` operation before being used in the code. The operation `sem_destroy(...)` should not be used with named semaphores. This operation returns an error if the implementation detects that the semaphore is being used by other locked threads that have called the operation `sem_wait(...)` on the semaphore variable in question.

With regard to the initialisation operation defined on named semaphores, the first thing to do would be to establish a connection between the semaphore and the calling process in order to be able to perform operations on that semaphore.

```
sem_t* sem_open(const char* name,
       int oflag [, unsigned long mode, unsigned int value])
```

The semaphore remains usable until it is closed. The `sem_open()` operation returns the address of the semaphore to the calling process. The `name` parameter points to a string naming the semaphore object. If there is an error condition, it returns "−1" and assigns `errno` to indicate the error condition. The operation `sem_open()` is idempotent, i.e., if a process makes several calls to the operation with the same name value, the same semaphore address will always be returned. The `oflag` flag determines whether the semaphore is created or accessed by the `sem_open()` operation call. Valid values of `oflag` are: `'0', O_CREAT, O_EXCL`. If '0', it means to open an existing semaphore; if `O_CREAT,` a semaphore will be created if it does not already exist; or `O_CREAT|O_EXCL` can be specified which will fail if the semaphore already exists. After a semaphore has been created with the `O_CREAT` flag and given a name, other processes can connect to this semaphore by calling `sem_open()` and using the value of the semaphore name, but without setting bits in `oflag`. Moreover, if the `oflag` flag is used, two more arguments must be given: the mode (third argument), which sets the permissions of the semaphore after clearing all the bits assigned in the process file creation mask, and the value (fourth argument) to create the semaphore with an initial value that must be less than or equal to `SEM_VALUE_MAX`. On the other hand, the operation `int sem_close(sem_t* semaphore)` is used to close the connection to a named semaphore and must not be used on unnamed semaphores (Fig. 1.9).

1.4.3.6 POSIX Semaphore Synchronisation Operations

For each type of semaphore, application threads will call the function `int sem_wait(sem_t* s)`, passing it a semaphore identifier initialised with the value '0', to use it as a general synchronisation semaphore that immediately blocks the thread, or if the value is other than '0', then s is decremented by one and does not block the thread.

The opposite operation to the previous one is `int sem_post(sem_t* s)` and serves to signal blocked threads in a semaphore queue and make one of them ready to execute. If there are no blocked threads in this semaphore, then the execution of this operation simply increments the value of the protected variable `s`. It should be noted, to use this operation correctly, that there is no unlock order defined if there are multiple threads waiting in the semaphore queue, since the system-level implementation of the `sem_post(...)` operation assumes that the scheduler can choose to unlock any of the

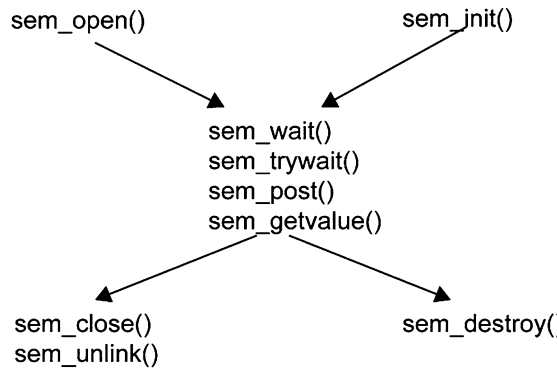

Fig. 1.9 Representation of the correct order of use of library functions Semaphore

suspended threads without such a choice affecting the safety property of the program. In particular, the following scenario could occur, another running thread could decrement the value of the semaphore pointed to by the `sem_post(...)` operation before any thread is unlocked from the semaphore queue and the awakened thread would then be re-blocked.

The two functions explained above may return errors if the semaphore is not correctly initialised before either of them is executed. The internal implementation of the previous operations is asynchronously safe, which means that if a signal fired by a program thread interrupts an operation that is being executed in another thread, while the latter has exclusive access to the semaphore variable, the semaphore will not remain in an inaccessible state for other operations on it that may be performed later by that thread or by others, i.e., there is atomicity of the semaphore operations that are executed concurrently by the threads of an application against asynchronous signals external to the thread.

On the other part, sometimes, it is convenient to avoid the blocking caused by calling the `sem_wait(...)` operation when the value of `s` is '0'. As an alternative to the latter, we can use another POSIX semaphore function: `int sem_trywait(sem_t* s)`, which atomically decrements the value of `s` only if it is positive; otherwise, it returns the error value '−1'. In addition, the following function `int sem_getvalue(sem_t* s,int *valuep)` is used to get the current value of the semaphore variable s. After it has been fully executed, it will store this value in `valuep`.

Example 1.3 The following example shows two threads, one writing to the global variable `protected_data` and the other reading from this variable. Each thread performs 10,000 iterations of a loop in which the threads repeatedly write or read the value of the shared variable. Two unnamed semaphores, called `write_ok` and `read_ok`, are declared and initialised in the program to write and read the value of the variable, respectively. Initially, only the write thread can act, so the value '1' is passed in the third argument of `sem_init(...)` for the `write_ok` semaphore, and the same is done for the mutex semaphore. Similarly, if the shared variable cannot be read initially because nothing has been written to it, the value '0' is passed in the third argument of the `sem_init(...)` function for the semaphore `read_ok`. The purpose of the mutex semaphore is to ensure that no errors occur in the screen output, since access to this resource could cause a race condition between the two program threads.

```
#include <pthread.h>
#include <iostream>
#include <semaphore.h>
  sem_t write_ok, read_ok, mutex;
  unsigned long protected_data;
  const unsigned long num_iter = 10000;
  sem_t write_ok, -- must be initialized to 1
  read_ok, -- must be initialized to 0
  mutex ; -- must be initialized to 1
```

Normally, we need to specify the compilation directive '-l' to link with the pthread library and, optionally, 'rt' (if we need to use the librt real-time library in our program). Note that the include <pthread.h> allows us to use the pthread functions in the program, but unlike the functions declared in the other written includes, such as studio.h, the actual code of the functions in pthread.h is not linked by default when we compile our program with gcc. Consequently, if we use functions from the POSIX thread library and do not specify the -lpthread option when compiling, the linking phase with the pthread library will fail because it will not be able to find the functions it needs, such as pthread_create(...).

```
--Code of the writer thread
void* write( void* p ){
  unsigned long counter=0;
  for( unsigned long i= 0; i< num_iter; i++ )
  {
    counter= counter + 1;
--generates one new value
    sem_wait( &write_ok );
    protected_data = counter;
-- write the value
    sem_post( &read_ok );
    sem_wait( &mutex );
    cout << "written data == " << counter << endl << flush;
    sem_post( &mutex );
   }
  return NULL;
}

--Code of the reader thread
void* read( void* p ){
  unsigned long read_value;
  for( unsigned long i= 0; i< num_iter; i++ )
  {
    sem_wait( &read_ok );
    read_value= protected_data;
-- read the protected value
    sem_post( &write_ok);
    sem_wait( &mutex );
    cout << " read value == " << read_value << endl;
    sem_post( &mutex );
  }
  return NULL ;
}
```

Finally, we just need to initialise the unnamed semaphores that we are going to use in the main() function, create the threads that will support the concurrent processes of our program and compile with the correct directives, and that's it!

```
Main Program
 int main(){
   pthread_t writer, reader;
   sem_init( &mutex, 0, 1 ) ;
   sem_init( &write_ok, 0, 1);
   sem_init( &read_ok, 0, 0);
   pthread_create( &writer, NULL, write, NULL );
   pthread_create( &reader, NULL, read, NULL );
   pthread_join( writer, NULL ) ;
   pthread_join( reader, NULL );
   sem_destroy(&write_ok); sem_destroy(&read_ok ); sem_destroy(&mutex);
 }
```

1.5 Properties of Concurrent Systems

In concurrent systems[15], a property is understood as a correctness attribute that must be satisfied in every execution of the system. The set of all possible executions, whose correctness we need to verify, define the concept of system's behaviour. Any property of a concurrent system, however complex it may be, can be formulated as a combination of two fundamental types of properties: (1) safety[16], or the certainty that the behaviour of the system will never enter a situation where all its processes are deadlocked, and (2) the property[17] ensuring that the system will eventually[18] reach a desired state,for example, that all its processes will eventually reach the critical section. To verify this property, it must be shown that no execution of the system indefinitely delays reaching this desired state, even if it is not possible to set a specific time limit for when it will be achieved.

1.5.1 Safety Properties

Safety properties define conditions[19] that must be upheld throughout the execution of a program. These often include mutual exclusion conditions, precedence relations between instructions or processes and the absence of deadlock. A practical rule for distinguishing safety properties from other properties is to check whether the property would always be

[15] We will understand the term "system" as a set of active components that interact, through communication and synchronisation. It would include concurrent and real-time programs and applications; but not only that, hardware devices controlled by the software are also considered part of it.

[16] The property states that no execution included in the system behaviour can enter a forbidden state.

[17] This concurrent property is called "liveness".

[18] The term "eventually" indicates that the system must reach a desired state within a time frame that is in principle indefinite, but not arbitrarily long.

[19] They usually coincide with requirements that must always be met by the system, which are also called static system specifications, i.e., they are always met regardless of the concrete execution that the system may dynamically follow.

satisfied if the software were implemented as a sequential program. For example, the impossibility of reaching a deadlock—where processes cannot proceed—is a critical safety property that must be verified. In a sequential version of the software, deadlocks will not occur because processes would sequentially obtain all the resources needed to complete their tasks. As a result, the second of Coffman's conditions[20] for deadlock would not be met.

There are several practical examples of safety properties in concurrent systems, which are common in solving well-known concurrent programming problems:

- Mutual exclusion problem: the condition that two program processes can never simultaneously execute the instructions belonging to a critical section is a safety property.
- Producer-consumer problem: similarly, a safety property ensures that the consumer process cannot extract data from an empty buffer and the producer process cannot insert data into a full buffer.
- Deadlock situation: this represents the most serious violation of safety properties. A deadlock occurs when none of the processes in a system release their resources while all are trying to access memory, causing memory to fill up and preventing any process continuing.

1.5.2 Liveness Properties

Liveness properties express that a system will reach certain desired states within a finite amount of time. This means there must be executions in the system's behaviour that reach these "good states", identifying correctness attributes, and that this happens often. Liveness properties also ensure that the system's execution conditions do not lead to certain processes being starved[21], i.e., processes are not indefinitely delayed or prevented from making progress. There are several practical examples that help illustrate liveness properties, such as the following scenarios that occur in the paradigmatic examples mentioned earlier:

- Mutual exclusion: if a process wants to enter a critical section, the protocol must guarantee that no process is locked in a busy waiting state indefinitely, leading to starvation. To satisfy the liveness property, it must be proven that every process will eventually be able to enter the critical section.
- Producer-consumer: a process that wants to insert or remove data from the buffer must be able to do so within a finite amount of time. This result must be provable based only on the values of the shared variables involved in the buffer access protocol.

[20] In operating systems, it is studied that four conditions must be met for a mutual deadlock or interlock to occur between processes in a concurrent program, known as the "Coffman conditions".

[21] Processes suffer starvation when they are systematically overtaken by others and fail to make progress in the execution of useful instructions.

The most serious violation of the liveness property results in the starvation of one or more processes in the concurrent system under test. While this is less severe than a complete system deadlock, since other processes can still perform some useful work, allowing any process to starve is unacceptable. Starvation indicates that the system contains inoperative code, meaning it does not adhere to the principles of concurrent programming and would be considered incorrect.

1.5.3 Fairness Properties

In addition to the liveness property in a concurrent system, it is essential to ensure that when a process is ready to run, it can do so with justice compared to other concurrent processes. This property requires the system to meet more stringent conditions than those needed to demonstrate the liveness. Whether or not fairness can be guaranteed often depends on the implementation of the process scheduler on the specific execution platform.

A classic example where the fairness property is not typically satisfied is in real-time systems, where different priorities are assigned to application processes. In such cases, the "unfairness" is intentional and necessary because not all the processes have the same criticality in terms of reliability or timeliness. Ensuring that higher-priority processes receive more CPU time is essential for the system to function properly in these scenarios.

Example 1.4 Five philosophers dedicate their lives to two activities: thinking and eating. Both activities take an unknown but limited amount of time. These philosophers share a round table with five chairs, each one belonging to one philosopher. On the table, there are five plates of slippery spaghetti and five chopsticks (instead of forks, to better fit the analogy). As shown in Fig. 1.10, philosophers require two chopsticks to eat their spaghetti.

When a philosopher is not trying to sit at the table, he is assumed to be thinking, a task to which the philosophers devote most of their time. From time to time, a philosopher feels

Fig. 1.10 The five philosophers at the table with *chopsticks* between them

hungry, approaches the table and attempts to grab the two chopsticks closest to him. Once in possession of the chopsticks, the philosopher can eat from the plate and will not release the chopsticks until he has finished eating and is ready to return to thinking.

However, philosophers are very stubborn: if one manages to grab one chopstick, he will never release it, even if he is unable to acquire the second chopstick. Similarly, no philosopher will allow another to take the chopstick he already holds while he waits for the second one. To solve this problem, a protocol must be invented that ensures each philosopher can eventually eat while maintaining the safety and liveness properties of the concurrent philosopher processes.

The most serious violation of the safety property would occur if each philosopher indefinitely held onto one chopstick without being able to acquire the second, leading to a situation where no philosopher can eat. In the above scenario, none of the philosophers would release their chopstick, and all would starve.

A less severe, but still problematic, violation scenario of the liveness property could occur if two philosophers conspired to starve the philosopher sitting between them. In this scenario, when one of the two conspiring philosophers finished eating and releases the chopsticks, the other immediately takes them, leaving the philosopher between them perpetually short of either the left or right chopstick. This would result in the starvation of the middle philosopher, demonstrating that the liveness property is not satisfied, as at least one process-philosopher would be prevented from making progress.

According to Coffman's second condition for deadlock (where processes hold onto resources while waiting indefinitely for additional resources), if each philosopher attempts to acquire one chopstick at a time, the system inevitably reaches a deadlock state where no philosopher can proceed, leading to starvation for all.

One possible solution to this problem is to introduce an additional butler process. If philosophers cannot acquire all the resources at once, the butler could limit the number of philosophers allowed to sit at the table to a maximum of four at any given time. This ensures that at least one philosopher will always be able to acquire two chopsticks and eat. Starvation of the philosophers can also be prevented by the butler by ensuring that no philosopher is permanently excluded for sitting at the table.

The most general definition of software correctness that we can give, and that can be applied to both sequential and concurrent systems, is the following: a system is correct if it always satisfies its predefined properties, during any of its executions.

1.6 Hoare's Program Logic and Concurrent Programs Verification

A program is said to be partially correct if, given that the program terminates, the results obtained after its execution are as expected. In addition, a program is said to be completely correct if it is partially correct and its termination can always be proved. However,

the above concepts are tied to proving the correctness of sequential programs and, therefore, cannot be considered adequate to define the correctness of concurrent systems or applications, since we will not normally be able to assume a termination state in our proofs, as concurrent systems are implemented to be in perpetual execution. In fact, their termination is usually associated with the occurrence of an error condition, followed by an abort, or the occurrence of an exception or a forced system restart. Operating systems, real-time control systems, ATM software, flight control and reservation software, etc. are classic examples of systems that are designed to run continuously and are concurrent systems by nature.

1.6.1 Options Available for Testing Concurrent Software

In order to demonstrate that a code written in any programming language is correct, i.e., to carry out the activity called "software verification", we could consider that different software testing methods (not necessarily concurrent) can be used, such as the following:

- *Code debugging*[22], which consists of exploring some of the possible executions of a code generated by a concurrent program and verifying that these executions are acceptable because they satisfy the initially specified properties. The problem is that it is not useful for verifying concurrent systems, because it will never be possible to prove the absence of transient errors that may occur in the execution of a concurrent program due to the occurrence of race conditions in unexplored execution sequences.
- *Operational point of view*—this method of software testing could be understood as performing an exhaustive case analysis, i.e., exploring all possible execution sequences of a given code, considering all possible interleavings of the atomic instructions that its processes will generate. However, this is not feasible for concurrent systems due to the astronomical number of interleaved instruction sequences that even a short piece of software with a few threads of execution can produce.
- *Assertive reasoning* is a formal analysis based on predicate logic that allows an abstract representation of the concrete states that a program reaches during its execution. Recall that a state of a program is defined by the values that the program variables have at a given moment of its execution. Therefore, if the initial state is an assertion that satisfies the input data or initial conditions and the final state is satisfied by the expected results of the program, we will have a demonstration in which the number of states to be checked is equal to the number of instructions in the code.

[22] Debugging or testing is the name for this activity in the texts.

1.6.2 Program Verification Based on the Axiomatic Semantics of a Programming Language

It is based on the use of assertive reasoning, i.e., based on assertions or propositions that are evaluated as true or false, thanks to a formal logic system (FLS) that facilitates the elaboration of certain assertions, with a precise logical-mathematical basis. This semantics interprets the constructs of a programming language based on the states of a program that uses them and on the evolution of these states during execution. More specifically, the mathematical definition of an FLS is as follows:

$$\text{FLS} = \{\text{symbols, formulas, axioms, inference rules}\}$$

1.6.2.1 Symbols: {Programming Language Sentences, Propositional Variables, Operators of Logic, etc.}

The *Formulas* are well-formed sequences of symbols *Rules of inference* indicate how to derive true formulas from axioms (formulas that we know to be true) and from other formulas that we have earlier proved to be true.

The inference rules have the following meaning: if all the hypotheses are true, then the conclusion C is also true: $(\textit{rule name})\dfrac{H_1, H_2, \ldots, H_n}{C}$

Both the hypotheses and the conclusion of the inference rules must be syntactically well-formed formulas or a schematic representation of them.

Additional concepts for verifying code using this method are the definition of theorems and the way in which they can be proved. A logic theorem, or proposition, is a formula that makes a certain statement about facts that belong to the domain of discourse. In our case, facts can be understood as the values of the states reached by the execution of a program and their relation to the fulfilment of the previously specified properties that the program must satisfy. From a technical point of view, the asserts of our FLS, which we will define precisely, correspond to the logical lines or sentences in which the proof of a program is structured. A proof of the correctness of a program is a sequence of asserts such that each one can be derived from the previous ones by applying an inference rule of the FLS.

For an FLS to be formally acceptable, and for the proofs we make using it to be reliable, such an FLS must satisfy the abstract properties called *soundness* and *completeness*, for whose definition we will need to introduce the concept of interpreting formulas.

- *Interpretation*: In order to know whether a well-constructed FLS assertion is true, it is necessary to provide an evaluation of the certainty of the formulas which is mathematically well defined, such as the following correspondence: *Interpretation* → *Logic Constants*: {*F*, *V*}. This means that every formula constructed with our logic must be evaluated to determine whether it is true (*V*) or false (*F*).

- *Soundness*: the FLS we define is certain with respect to a given interpretation if all the asserts that can be derived with this system are true facts of the domain of discourse. Therefore, if we define the set facts = {certainties expressed as formulas} and the set asserts = {demonstrable formulas}, then the FLS has the property named soundness if the following inclusion relation is satisfied: *asserts* ⊆ *facts.*
- *Completeness*: this property is satisfied if every true assertion of the FLS is provable, i.e., the following inclusion relation must be satisfied: *facts* ⊆ *asserts*. We will interpret that every fact in the domain of discourse is provable with the FLS we propose, using for this the axioms and the derivation rules of the FLS.

1.6.2.2 Propositional Logic

This is a clear example of an FLS that can be used to formalise what we normally call common sense reasoning. The formulas of this logic are called propositions, and their symbols are the following:

- *Propositional constants*: {V, F}
- *Propositional variables*: {p, q, r, …}
- *Logical operators or connectors*: {¬, ∧, ∨, →, ←…}
- *Expressions using constants, variables, and operators*

As for the interpretation of a propositional formula, we can understand this concept in the context of the propositional logic by the evaluation we obtain from the formulas after substituting the values of the variables in each state of the program, for example, {$X = a$} evaluates as true if indeed in the current state of the program it is satisfied that the value of the variable is identically equal to 'a'. We can formally define the concept as follows: given a state s of a program, described by the formula P, in which we replace each propositional variable by its value in that state and then use the truth table of logical connectors {¬, ∧, ∨, →, ←…} to obtain the result, the propositional formula interpretation coincides with its overall truth value of P.

Moreover, the certainty of a formula P will depend on the state at a given point of the program's execution, according to the following definitions:

- One formula is satisfied in one state s if and only if it has a true interpretation in that state.
- One formula will be satisfiable in a program p if and only if there exists some state of p in which the formula can be satisfied.
- One formula is valid if and only if it can be satisfied in any state of p.

Valid propositions are called tautologies in the Logic. In a propositional logic, the soundness property of an FLS is always fulfilled. This is because it can be shown that all axioms are tautologies, since every axiom must always be valid.

1.6.2.3 Tautologies or Propositional Equivalence Laws

These laws can be understood as equivalences that allow the substitution of a proposition by its equivalent, thus simplifying complex formulas in demonstrations:

1. Law of negation: $P = \neg(\neg P)$
2. Excluded media law: $P \vee \neg P \rightarrow V$
3. Contradiction law: $P \wedge \neg P = F$
4. Law of implication: $P \Rightarrow Q \equiv \neg P \vee Q$
5. Equality Law: $(P \rightarrow Q) \wedge (Q \rightarrow P) \equiv P \Leftrightarrow Q$
6. Or-Simplification laws:

$$\begin{aligned} P \vee P &= P \\ P \vee V &= V \\ P \vee (P \wedge Q) &= P \\ P \vee F &= P \end{aligned}$$

7. And-Simplification laws:

$$\begin{aligned} P \wedge V &= P \\ P \wedge P &= P \\ P \wedge F &= F \\ P \wedge (P \vee Q) &= P \end{aligned}$$

8. Commutative laws:

$$\begin{aligned} (P \wedge Q) &= (Q \wedge P) \\ (P \vee Q) &= (Q \vee P) \\ (P = Q) &= (Q = P) \end{aligned}$$

9. Associative laws:

$$\begin{aligned} P \wedge (Q \wedge R) &= (P \wedge Q) \wedge R \\ P \vee (Q \vee R) &= (P \vee Q) \vee R \end{aligned}$$

10. Distributive laws:

$$\begin{aligned} P \wedge (Q \vee R) &= (P \wedge Q) \vee (P \wedge R) \\ P \vee (Q \wedge R) &= (P \vee Q) \wedge (P \vee R) \end{aligned}$$

11. Morgan's laws:

$$\begin{aligned} \neg(P \wedge Q) &= \neg P \vee \neg Q \\ \neg(P \vee Q) &= \neg P \wedge \neg Q \end{aligned}$$

12. And-deletion: $(P \wedge Q) \rightarrow P$
13. Or-deletion: $P \rightarrow (P \vee Q)$

Rule 12 can be explained by saying that the set of states satisfying P trivially includes the set of states satisfying $(P \wedge Q)$; proposition P is said to be weaker and is therefore implied by the stricter expression $(P \wedge Q)$. According to the above, the constant F (or false) is the strongest proposition, since it implies every other proposition of the Logic, and the constant V (or true) is the weakest proposition since it would be implied by every proposition of the logic.

1.6.2.4 Program Logic

It is an FLS that allows precise statements to be made about the execution of a program. Program Logic (PL) symbols include the statements of a programming language and the logical formulae, which are called triples, are of the form $\{P\}\ S\ \{Q\}$, where P and Q are asserts and S is a simple or structured statement of a programming language. The free variables of P and Q belong to the program or are logical variables. The latter act as containers for the values of the common variables of the program and cannot be assigned more than once, i.e., they always keep the value to which they were assigned the first time. They only appear in asserts, not in statements, and are usually represented in uppercase to distinguish them from program variables, which appear in lowercase.

Triples $\{P\}\ S\ \{Q\}$ are interpreted theorems of PL, i.e., we say that a triple is true or the logical formula it represents evaluates to true if the execution of the instruction S starts in a state of the program that satisfies the assert P (or precondition) and that the final state must satisfy the assert Q (or postcondition), after any interleaving of atomic instructions as a result of the execution of S. Note that an assert characterises an acceptable state of the system, i.e., a state that can be reached by the program if its variables take certain values. Each state of the program must satisfy its associated assert for the interpretation of the triple to have the value V (true).

The assert $\{V\}$, the logical constant V, will represent all the possible states of the program, since this assert is satisfied in every state of the program independently of the values taken by the variables at any state, i.e., $\{P\} \rightarrow V$ is a valid formula. On the other hand, an assertion equivalent to the logical constant F is not true in any state of the program, i.e., $F \rightarrow \{P\}$ is also a valid formula.

The axioms and rules of inference of PL are defined in the following list:

1. *Axiom of the null instruction*: $\{P\}null\{P\}$: if the assertion is true before the execution of the null sentence, it remains true after the execution.
2. *Textual substitution*: $\left\{P_e^x\right\}$ is the result of substituting the expression e in any free occurrence of the variable x in P. The names of the free variables of the expression e must not conflict with bound variables that exist in P; and, therefore, any relation of the program state that has to do with the variable x and that is true after the assignment must also have been true before the assignment.

3. *Axiom of assignment*: $\left\{P_e^x\right\} x := e \left\{P\right\}$: this sentence of the programming language assigns a value e to a variable x and thus, in general, modifies the state of the program. An assignment changes only the value of the target variable; all other variables keep the same value as before the execution of the assignment sentence. For example, the triple $\{V\}x := 5\{x = 5\}$ is a true assert, because the textual substitution of 5 in the variable x always is tautologic $\{x = 5\}_5^x \equiv V$. In PL, there are inference rules for each of the statements that affect the flow of control in a structured sequential program, plus three additional inference rules for connecting triples in program demonstrations.
4. *Rule of consequence* (1): $\dfrac{\{P\}\, S\, \{Q\}, \{Q\} \to \{R\}}{\{P\}\, S\, \{R\}}$ The interpretation of this rule is that it is always possible to make the postcondition weaker, i.e., to replace the postcondition of a triple by a weaker assert and that its interpretation is preserved, in which case the triple with postcondition $\{R\}$ remains true.
5. *Rule of consequence* (2): $\dfrac{\{R\} \to \{P\}, \{P\}\, S\, \{Q\}}{\{R\}\, S\, \{Q\}}$ The meaning of this rule is that the precondition of a triple can always be made stricter and that its interpretation is preserved, i.e., the triple with precondition $\{R\}$ remains true, given that the original triple does.
6. *Composition rule*: $\dfrac{\{P\}\, S_1\, \{Q\}, \{Q\}\, S_2\, \{R\}}{\{P\}\, S_1; S_2\, \{R\}}$ allows to obtain the composition of postcondition and precondition of two sentences together, from the precondition of the first one and the postcondition of the second one, if the postcondition of the first one coincides with the precondition of the second one.
7. If rule:

$$\frac{\{P\} \wedge \{B\}\, S_1\, \{Q\}, \{P\} \wedge \{\neg B\}\, S_2\, \{Q\}}{\{P\}\, \mathit{if}\ B\ \mathit{then}\ S_1\ \mathit{else}\ S_2\ \mathit{fi}\, \{Q\}}$$

   Suppose that the precondition of the if statement to be proved is $\{P\}$ and the postcondition to be reached is $\{Q\}$; then, to prove the certainty of $\{Q\}$, we need only prove that the two branches of the if statement make the same postcondition $\{Q\}$ true.
8. *Iteration rule*: $\dfrac{\{I \wedge B\}\, S\, \{I\}}{\{I\}\, \mathit{while}\ B\ \mathit{do}\ S\ \mathit{enddo}\, \{I \wedge \neg B\}}$ A while statement can be iterated an arbitrary number of times, including 0, without affecting the certainty of the invariant I. For this reason, the rule of iterative inference is based on this loop invariant, i.e., an assert $\{I\}$ that is satisfied before and after each iteration of the loop. The final postcondition of the while complete statement is identically equal to the conjunction of the invariant and the negation of the loop iteration condition.

1.6.3 Verification of Concurrent and Synchronisation Statements

The verification of concurrent programs is affected by a problem known as interference between assertions and statements running on another thread. When such interference occurs, the individual process assertions made with the PL are invalidated, i.e., the inference rules are no longer sound. This is because another concurrent process can execute an atomic instruction that makes the precondition (or postcondition) of a statement within the first demonstration false. If this were to happen, the soundness property would no longer be satisfied in the PL system, and PL would not be useful for verifying concurrent programs. The following program is an example of interference between processes of a concurrent program. Depending on which register is loaded (load y) and incremented (add z)[23] before, after or at the same time as the two right assignment operations ($y := 1$ and $z := 2$), the execution of this program results in a value of $x \in \{0, 1, 2, 3\}$, but we cannot determine which one.

$$y := 0;\ z := 0;$$
$$\texttt{cobegin}\ x := y + z \parallel y := 1;\ z := 2\ \texttt{coend};$$

A peculiarity of the above program is that it can produce the final value of $x = 2$ although $y + z = 2$ does not correspond to any valid state of this program.

We must bear in mind that not all nested sequences of process instructions are acceptable, and race conditions can occur, leaving variables shared by the processes with incorrect values. Therefore, if we make good use of synchronisation statements in our programs, such interference between concurrent processes can be avoided.

Variants of the synchronisation constructs are often used in different programming languages to avoid inter-process race conditions and transient errors in execution sequences:

- Critical sections: programmed as instruction blocks in the process code, which must be executed respecting the property of mutual exclusion when accessing the instructions they contain. In this way, simple atomic instructions are combined into structured atomic actions that are executed indivisibly by the program processes.
- Conditional synchronisation: delays the execution of a process until an assert is satisfied, indicating that the program has reached a certain desired state, and then allows the delayed process to continue without unwanted interference with other running processes. An example of this type of instruction would be the use of semaphore operations `sem_post()` and `sem_wait()` in simple concurrent programs (without the use of monitors, actors, etc.) or programming with condition variables used in monitor procedures.

[23] $x := y + z$, interpreted as load y; add z; store x

1.6.3.1 Elemental Atomic Action and Concurrent Composition Inference Rule

The abstract statement marked "<Sentence>" is used to express the atomicity of the code between brackets and has different realisations in concurrent programming languages, such as synchronised blocks, critical regions, monitor procedures, etc. In sequential programs, assignments are always atomic actions, since there is no intermediate state visible to the rest of the processes; however, this situation does not occur in concurrent programs, since an instruction containing the assignment statement is often equivalent to a sequence of elementary atomic operations:

> An elementary atomic action, no matter how many instructions it contains, performs an indivisible transformation of the program state.

$\{x=0\}$
$x := x+1;$
$\{x=1\}$
the assignment to variable x is
compiled into the next atomic
instructions:
$<load\ x,\ 0>$
$<add,\ 1>$
$<store,\ x>$
and finally the postcondition is $\{x=1\}$

The effect of declaring an elementary atomic action in the text of a process is that any intermediate state that might exist during the execution of that statement would not be visible to the rest of the processes in the program, and therefore, transient errors in the program execution sequences would be avoided.

The process P_1 "sees" only two possible states of this program, before and after the atomic action <x:= x+2>, i.e., the precondition { x = 0 } and the postcondition { x = 2 } of the atomic action contained in the process P_2. Something completely equivalent happens to the process with respect to the atomic action <x:= x+1> seen by the other process. As a consequence, the precondition and postcondition of both processes now become disjunctions if we assume that both processes are executed simultaneously, for the purpose of performing a test to prove the non-interference of both atomic propositions:

$$P_1 :: \{x = 0 \vee x = 2\} < x := x+1 >; \{x = 1 \vee x = 3\}$$
$$P_2 :: \{x = 0 \vee x = 1\} < x := x+2 >; \{x = 2 \vee x = 3\}$$

and now we can apply the inference rule of concurrent composition.

More formally, the atomic assignment action a does not interfere with the critical assertion $\{C\}$ if the triple $\{C \wedge pre(a)\}\, a\{C\}$ can be proven as a theorem of PL. This triple is interpreted to mean that the execution of a does not change the truth value of $\{C\}$. In other words, the certainty of the assertion $\{C\}$ remains invariant with respect to the atomic action a executed by another process, which must start in one state that satisfies the precondition $\{pre(a)\}$.

To correctly demonstrate non-interference between an atomic action and a critical program assertion, it may be necessary to rename the local variables of $\{C\}$. This avoids potential conflicts between the local variables in $\{C\}$, the instruction a and the precondition $\{pre(a)\}$.

A set of processes is said to be free of interference if there is no atomic action in any of these processes that interferes with any critical assertion contained in the demonstration of any other process.

This statement is the antecedent of the following new PL inference rule:

9. *Rule of safe composition of concurrent processes*:

$$\frac{\{P_i\}\, S_i\, \{Q_i\} \text{ are non interferring triples}, 1 \le i \le n}{\{P_1 \wedge P_2 \wedge \ldots \wedge P_n\}\ \texttt{cobegin}\ S_1 \,\|S_2\, \|\ldots\|S_n\ \texttt{coend}\{Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n\}}$$

The interpretation of the above rule is as follows: if we can prove that the set of concurrent processes S_i is interference free, then their concurrent composition transforms the conjunction of the preconditions of the processes $\{P_1 \wedge P_2 \ldots \wedge P_n\}$ of the state before the `cobegin` statement into the conjunction $\{Q_1 \wedge Q_2 \ldots \wedge Q_n\}$ of their postconditions satisfied in the state after the `coend` statement, i.e., the demonstrations of the individual processes S_i as sequential programs are valid even though these processes run concurrently, and no additional demonstration is required to ensure that such code satisfies the safe concurrent composition of these processes.

$$\{\, x=0 \,\}$$
$$\textit{cobegin}$$
$$\{x=0\}\ P_1::<x:=\ x+1;>\{\, x=1 \,\}\ \|\ \{x=0\}\ P_2::\ <x:=x+2>\ \{x=2\}$$
$$\textit{coend}$$
$$\{\, x=3 \,\}$$

$$\{x=0 \,\}$$
$$\textit{cobegin}$$
$$\{x=0\ \vee\ x=2\} \quad \{x=0 \vee\ x=1\}$$
$$x:=x+1; \quad \| \quad x:=x+2$$
$$\{x=1\ \vee\ x=3\} \quad \{x=2 \vee\ x=3\}$$
$$\textit{coend}$$
$$\{\, x=3 \,\}$$

Fig. 1.11 Two processes with elementary atomic actions and the correctness demostration of the program with the two processes

Thus, applying the rule of safe concurrent process composition, to demonstrate the correctness of the example shown in Fig. 1.11, and since it can be shown that the processes P_1 and P_2 do not interfere with each other, the demonstration written schematically as the second block of code in the figure is valid.

1.6.4 Verifying Concurrent Programs Using Global Invariants

Global invariants (GIs) are predicates in programming logic (PL) used to prove the safety properties of concurrent programs in a straightforward way. This avoid the need to apply the concurrency rule, which can be tedious in complex systems with many processes $(S_i{:}1 \ldots N\}$. In such systems, proving noninterference between N atomic instructions and N critical assertions would require N^2 non-interference demonstrations.

The GIs are expressions defined from the global variables of a program and are typically formulated as predicates in PL that capture the relationships between the variables shared by the processes. They are preferred over non-interference proofs because GIs eliminate the need to prove all non-interference theorems between the critical assertions $\{C\}$ and atomic actions a in individual process demonstrations.

If any critical assertion $\{C\}$ in the individual demonstrations of a concurrent process $\{P_i\}S\{Q_i\}$ can be written as a conjunction of the form $\{GI \wedge L\}$, where GI is a global invariant of the program and $\{L\}$ is a predicate involving only local process variables or function parameters, then it becomes unnecessary to perform the non-interference demonstrations required by the rule 9 of PL, named of safe composition of concurrent processes. This GI-based rule can be applied directly to the set of processes to prove the desired result: $\left\{\wedge_{i=1}^{n} P_i\right\}\left\{\left\|_{i=1}^{n} S_i\right.\right\}\left\{\wedge_{i=1}^{n} Q_i\right\}$.

However, an additional check is required for a predicate $\{I\}$, defined from the shared variables among the processes of a concurrent program, to be considered a valid global invariant. The following conditions must be satisfied:

1. The proposed predicate $\{I\}$ must be valid for the initial values of the variables involved.
2. $\{I\}$ must remain true after the execution of each atomic action a, concurrently executed by other processes. In other words, the following triple[24] must be proven for every action a included in the system:

$$\{I \wedge pre(a)\}a\{I\}$$

Put differently, it must demonstrate that there is no interference between the predicate $\{I\}$ and any elementary atomic process action a performed by the concurrent processes of the program.

Example 1.5a As an example of verifying the safety property of a concurrent program using *GI*, we will present a concurrent program scheme to perform secure transfers between two accounts of the same bank as a transaction[25], as this operation is understood in DBMS. We will therefore have to program the minimum critical sections necessary for a customer to make a safe withdrawal from one account and deposit into another, i.e., without at any time missing money from the total sum of the balances of all the bank's accounts. In addition, the banking system's software will iteratively execute code in parallel with other transfers to verify that the sum of the bank's account balances always remains constant. The necessary synchronisation operations must be included in the solution, trying to optimise the overall concurrency of the program. The elementary atomic action represents the transaction consisting of two operations, (1) withdrawal and (2) deposit, from the ordering account to the beneficiary account.

Solution Let us consider two processes: the first process P_1, performs a transaction consisting of two operations: decreasing the balance of account x by K units and increasing the balance of account y by the same amount, K. It is important that the intermediate state (where the balance of account x has been decreased but the balance of account y has not yet been increased) remains invisible to the second process P_2. This ensures that, from the perspective P_2, the total sum of the balances across all accounts always appears constant.

P_2 periodically checks that the sum of the all the account balances, represented by the array $c[i]$, remains unchanged. The global invariant $\{GI\}$ that must be satisfied throughout the execution of the program is expressed by the assertion: $Total = c[1] + \ldots c[n] = constant$.

To verify the safety property, the proof for process P_1 can be written by proving the following triple:

[24] The triple demonstration is the proof of non-interference between the atomic action a and the global invariant *GI*.

[25] A unit of work performed within a system against a database or repository and handled in a consistent and reliable manner so that it does not interfere with other transactions. A transaction is generally used to make any change to a database while maintaining data consistency.

$var\ c{:}array[1..n]\ of\ int;$

$P_1{::}$

$\{A_1{::}c[x] = X\ \wedge\ c[y] = Y\} \wedge \{GI\}$

$S_1{:} < c[x]{:=}\ c[x] - K; c[y]{:=}\ c[y] + K >$

$\{A_2{::}c[x] = X\ - K \wedge\ c[y] = Y + K\} \wedge \{GI\}$

on the other hand, for P_2we will have to carry out the proof given by

$P_2{::}$

$var\ Sum{:=}\ 0;$

$i{:=}\ 1;$

$\{Sum\ =\ \sum_{x=1}^{i-1} c[x]\}$

$while(i \le n)\ do$

$begin$

$\{B_1\} = \{Sum = \sum_{x=1}^{i-1} c[x] \wedge i \le n\}$

$S_2{:}Sum{:=}\ Sum + c[i];$

$\{B_2\} = \{Sum = \left(\sum_{x=1}^{i-1} c[x] + c[i]\right) \wedge i \le n\} \rightarrow \{Sum = \sum_{x=1}^{i} c[x] \wedge i \le n\}$

$S_3{:}i{:=} i + 1;$

$\{B_3\} = \{Sum = \sum_{x=1}^{i-1} c[x] \wedge i \le n + 1\}$

end

$enddo$

$\{B_4\} = \{Sum =\ \sum_{i=1}^{n} c[i]\} \rightarrow \{GI\}$

For process P_2 we must perform the demonstration using the sequence of triples described earlier. It is important to note that the P_1's assertions $\{A_1\}$, $\{A_2\}$ are not critical because the process instructions of P_2 only read the values of the elements in the array $c[i]$ without modifying them. Therefore, evaluating the logical value of these assertions cannot interfere with the execution of the program fragment in the demonstration of P_2.

However, the assertions B_1, B_2 are critical because their interpretation can be interfered by the assignment of array elements, which is programmed as an elementary atomic action S_1 in process P_1. This action will be executed concurrently with the atomic actions of P_2 in an order that is not predetermined. For instance, the values of the elements $c[x]$ or $c[y]$ in the array c could be modified while the process P_2 is summing these values in the execution of S_2.

To ensure the correctness of the global concurrent program, the instructions in P_2 must be preceded or *guarded* by a synchronisation mechanism. This synchronisation would prevent P_2 from accessing the same array elements at the same time as P_1. An example of such a synchronisation instruction could be:

$$S_1 : \langle wait((x < i \wedge y < i) \vee (x > i \wedge y > i)) \rightarrow c[x] = c[x] - K; c[y] = c[y] + K \rangle.$$

To prove safety properties of concurrent systems, it is often convenient to formulate them as the negation of predicates that characterise certain system states that cannot be reached by more than one process simultaneously. For instance, in the mutual exclusion problem, the safety property can be expressed by stating that the preconditions for access to the critical sections of processes P_1 and P_2 can never be true at the same time.

Let's define *NOTSAFE* as a predicate that characterises a program state where conflicting conditions can be true simultaneously. In other words, {*NOTSAFE* = P_1 *at CS* $\wedge$ P_2 *at CS*}. To prove that program *P* satisfies the mutual exclusion property, it is sufficient to show that the system can never reach this unsafe state. This can be done by proving the validity of the formula P_1 *at CS* $\wedge$ P_2 *at CS* = *False* throughout the entire execution of the program *P*.

More formally, let *NOTSAFE* be a predicate that characterises an undesirable state of the program. Suppose that we can prove the triple {*P*} *S* {*Q*}, where the assertion {*P*} characterises the initial state of the program. If we define {*GI*}, a global invariant of the proof, then S satisfies the safety property specified by ¬ (*NOTSAFE*) for any execution if we can prove the validity of the formula: {*GI*} →¬ (*NOTSAFE*). This can be considered an alternative definition of the safety property for the concurrent program under consideration. Therefore, in the example above, *NOTSAFE* ≡ *(x* ≥ *i* ∨ *y* ≥ *i)* ∧ *(x* ≤ *i* ∨ *y* ≤ *i).*

Example 1.5b In the case of bank transfers between accounts within the same bank, it can be shown that S_1 (transfer between accounts) and S_2 (constant balance calculation) are executed concurrently but always under mutual exclusion.

To demonstrate this, we use the invariant {*GI*} ≡ {*Total* = *c*[1] + *c*[2] ... *c*[*n*] = *Sum* = *constant*}, which states that the sum of the elements in the array *c* remains constant throughout the execution. The predicate

$$NOTSAFE = pre(S_1) \wedge pre(S_2) \equiv \{A_1 : c[x] = X + \in \wedge c[y] = Y + \delta\}$$
$$\wedge \left\{B_1 : Sum = \sum_{x=1}^{i-1} c[x] \wedge i \leq n\right\}$$

represents a program state where the sum of the array values is no longer constant, and such a state could be visible in the total balance calculation.

In order to verify the safety property of the program, we must show that it is impossible for the preconditions of both S_1 and S_2, represented by their respective predicates A_1 and B_1, to be true simultaneously. It can be trivially verified that {*GI*} →¬ (*NOTSAFE*) is a valid formula. By the very definition of {*GI*} and *NOTSAFE*, this formula can be interpreted as "the maintenance of {*GI*] throughout any execution guarantees that the program will never reach the unsafe state represented by *NOTSAFE*".