CHAPTER

7 Process Synchronization

Learning Objectives

After reading this chapter, you should be able to:

- Explain process synchronization and the need for it.
- Describe some fundamental synchronization problems.
- Discuss some useful synchronization tools.
- Describe solutions to synchronization problems.
- Explain deadlocks and their solutions.

7.1 Introduction

Processes in a computer system execute programs to manipulate data. One process may or may not interact with another process to accomplish its task, and as explained in Section 6.3.3 on page 138 such interactions take place only through shared data. If two processes use two distinct sets of data items, then they are not considered interacting processes and their activities do not affect each other. Their behaviours are independent and do not influence each other. Whereas, if a process A modifies a data item that another process B reads, then the behaviour of B may depend on activities of A. Here we say the two processes are interacting with one another through shared data. The behaviours of these interacting processes then depend on two factors: the relative speeds of the processes, and what they do with shared data. If the activities of interacting processes are not controlled suitably, their behaviour may not be as "consistent" as expected from their specifications.

≫ The operating system or any application execution, may be considered to be a collection of processes in which some interact and some others do not.

The theme of this chapter is the coordination of interacting processes, that is, the orderly execution of their accesses to shared data. Unlike interprocess communication (discussed in Chapter 6), process interaction is somewhat indirect. In many situations, a process may complete its own execution even if other processes are absent. Processes in general are independent, but they

≫ Synchronization is the single most important topic in highly concurrent systems such as operating systems, databases, and networks.

synchronize their relative speeds and accesses to shared data to ensure consistency of the shared data and/or system states. In this chapter, we will study many synchronization problems and their solutions using various synchronization tools.

7.2 Process Synchronization

≫ Concurrency is the notion of more than one related event happening at the same real time.

Modern operating systems are multiprocess systems. They run many processes concurrently. Concurrency among processes is exploited to achieve better utilization of system resources. Two processes are said to be *concurrent* if their program executions overlap in real time. That is, the first operation execution of each process starts before the last operation execution of the other process is complete. Figure 7.1 shows two typical execution scenarios of three concurrent processes. In Fig. 7.1(a), three processes are being executed in parallel (i.e., simultaneously) by three different processors, and in Fig. 7.1(b), they are executed in an interleaved fashion by a single processor. The solid line fragments indicate executions of instructions by a processor. The broken lines represent process durations. The gap between two solid line fragments in a process indicates that the process is waiting for some condition (multiprocessor case) and/or is preempted (uniprocessor case).

≫ Although both concurrent- and parallel executions imply that executions overlap in real time, traditionally parallel executions refer to executions on distinct processors that overlap in real-time while concurrent executions refer to executions that have the potential to be executed in parallel. That is, they can be executed in parallel, but their executions actually may or may not be parallel. When concurrent executions are not executed in parallel (on different processors) they are called pseudo parallel or apparently simultaneous executions.

Depending on the hardware platform configuration, execution of individual instructions from processes may overlap in real time (simultaneity) or interleave. In multiprocessor systems, instruction executions of many processes can proceed simultaneously in real time. In uniprocessor systems, the only way concurrency is achieved is through interleavings of instruction executions. In either case, an instruction execution of one process may affect those of other processes if the instruction executions reference the same piece of shared data. Whether processes execute instructions simultaneously or in an interleaved manner is immaterial to us here. As long as their relative speeds are unknown, they may lead to the occurrences of the same kinds of problems, all due to concurrency.

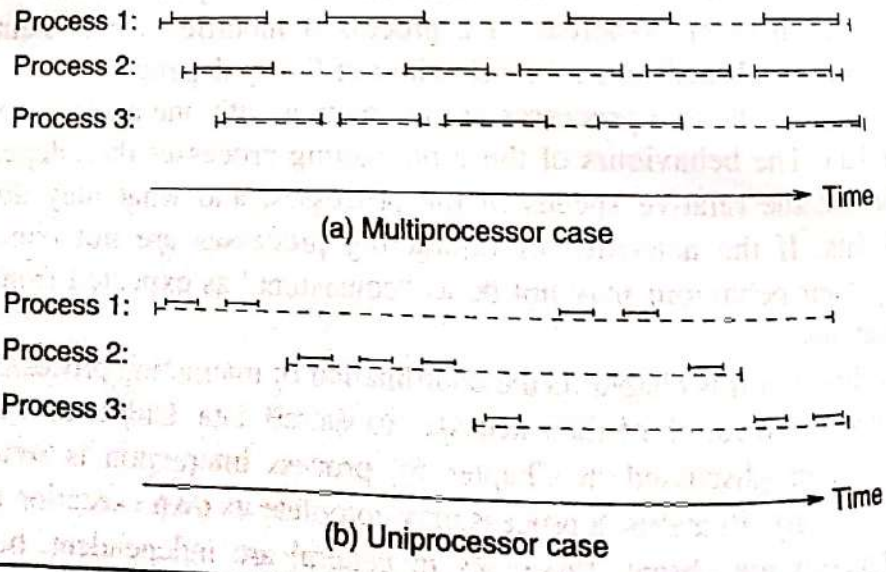

Figure 7.1: *Concurrent process executions.*

In multiprocess systems, different processes are at different states of execution. They normally execute programs and access data from their private address space. From time to time, they execute system call service routines from the kernel space. The system calls occur at unpredictable times, and they may access the same kernel data. They run at different speeds, and one process has no control over the speed of others. Consequently, the behaviour of one process may depend on what other processes do with the shared data and their relative speeds. The operating system must ensure that processes see consistent values of shared data. The data values must satisfy a priori specified integrity constraints.

≫ An integrity constraint is a set of invariants maintained by a system in order to ensure correct system behaviour and to fulfil its interface specification contract.

If executions of concurrent interacting processes are not controlled properly, they may store inconsistent values in shared data and their behaviour may be difficult to predict. Process coordination synchronization (aka) is a fundamental problem in multiprocess operating system design and development. In general, a synchronization problem is to achieve a specified coordination among a set of concurrent processes. Processes themselves must coordinate their activities to maintain a coherent behaviour. They, in general, are independent, and may not be aware of one another. One process becomes aware of another by waiting on a condition that is set by the other process. That is, processes need to communicate (directly or indirectly) information among themselves to coordinate their own activities. (This is different from interprocess communication.) Like interprocess communications, these communications are also done through reading- and writing shared variables.

Kernel space has many shared data structures. Accesses to these shared data are done by executing various kernel paths on behalf of the communicating processes. When a kernel path is modifying a shared variable, no other kernel paths should be allowed to access the shared variable. Otherwise, consistency of the shared variable and behaviours of kernel paths may not be guaranteed. These problems also arise in any shared memory-based communications where processes directly access shared variables in the shared memory. Therefore, whether a shared data is in the user space or in the kernel space, coordination of accesses to the data to ensure its consistency semantics is essential. In a process if there are multiple threads concurrently accessing common global data from the process address space, we have the same kind of synchronization problems as for threads. In multiprocessor systems, we have similar coordination problems for processor actions such as sharing the CPU queue.

In a nutshell, chaos ensues in the system when more than one agent (thread, process, kernel path, processor, etc.) accesses a piece of shared data concurrently, unless such accesses are properly controlled. Coordination or synchronization of the activities of agents is vital to maintain data consistency. In modern high performance computers, a large number of activities (of kernel paths) proceed concurrently in the kernel space. Synchronization problems are almost everywhere in operating systems. Designers and developers of operating systems must understand the intricacies and complexities of these problems before building highly concurrent operating systems.

7.3 A Typical Synchronization Problem

Let us first study a very simple two-process interaction, shown in Fig. 7.2, which illustrates a typical synchronization problem. There are two processes, namely P_1 and P_2. They share an integer variable v that is initialized to 0. It is read and written by executing *atomic* (i.e., indivisible) read- and write operations. In two separate higher-level operations, process P_1 reads and increments v by 1, and P_2 reads and decrements v by 1. The lower level operation executions interleave in the manner shown in the figure. The two reads on v are executed before the two writes on v. Both processes read 0 from v. The final value of v depends upon which write on v is executed last. Then, the final value is either -1 or $+1$, (in the figure, it is -1). If the two higher-level operations were executed sequentially, in either order, the final value of v remains 0. Consequently, for the scenario in the example, we say that the final value of v is inconsistent, that is, the value does not reflect any order in the two higher-level operation executions. The inconsistency has occurred because we allowed both processes to manipulate the shared variable v concurrently in an uncoordinated manner. (This is also the case in the message queue implementation in Fig. 6.6 on page 145 for the *count* shared variable.)

≫ Atomic means that the operation runs to completion or "as if" not run at all. Indivisible means that the operation cannot be stopped in the middle of the run and the shared state cannot be modified by another operation in the interim.

As mentioned previously, a shared data is accessed by a set of operations. When these operations are executed serially, each operation execution is guaranteed to transform the shared data from one consistent state into another. However, data consistency may not be guaranteed when many processes execute those operations concurrently on shared data. In general, when many processes read and write the same shared variable in an uncoordinated manner, the outcomes of these operation executions are unpredictable and would depend on the particular order of execution. A situation where the outcome is unpredictable, like in a race, is called a *race condition*. If the operations are carried out in a different order, the processes may behave differently. Unfortunately, not all outcomes of a race condition may need to be correct. Presence of race conditions in a system is regarded as bad design. Race conditions should be avoided at all costs to ensure predictable system behavior. Therefore, control of accesses to shared variables is necessary to eliminate race conditions occurring in the system. Orderly execution of operations on shared variables is called a coordination- or synchronization of operation executions.

≫ Anomalies due to race conditions are notorious in concurrent systems, and errors due to them are hard to debug and diagnose. As noted in Section 1.2.2 on page 6, when a race condition is present, a program execution may not complete and even if it does complete, it may not produce correct output. They are race hazards!

Shared data structure and initial value

```
int v = 0;
```

Process P_1	Process P_2
1. int v1 = read(v); /* v1 == 0 */	2. int v2 = read(v); /* v2 == 0 */
3. v1 = v1 + 1; /* v1 == +1 */	4. v2 = v2 − 1; /* v2 == −1 */
5. write(v, v1); /* v == +1 */	6. write(v, v2); /* v == −1 */

Figure 7.2: *A typical uncoordinated operation executions on a shared variable.*

If a shared variable is of a primitive data type, the machine architecture may provide the basic synchronization for accessing the variable, and it may be treated as an atomic variable. That is, operation executions on the variable happen in some total order respecting their original order. Unfortunately, most shared variables at a higher level are non-atomic data structures. Even for a single atomic variable, a process may execute a sequence of operations on the variable in a row as shown in Fig. 7.2 on page 154, and expect the entire operation sequence to be indivisible. The term *synchronization* is used to specify various constraints on the orderings of operation executions of different processes on shared variables. Until a process modifying a shared data has finished applying all its operations, no other process is allowed to read or write the same data.

Synchronization constraints mostly specify that some operations cannot be executed concurrently. For example, two store operation executions on the same memory address must exclude each other; otherwise some bits in the addressed memory cell may have values from one store operation and other bits from the other store operation. We will study some interesting synchronization constraints in this chapter. The term synchronization is also used to control activities of cooperating processes, for example, their relative speeds.

Shared variables are used to model system resources. Each resource is modelled by a set of related shared variables whose values collectively represent states of the resource; modifying these values reflects changes in the state of the resource. Processes operating on the resource make references to the shared variables. Control of operation executions on shared variables are necessary to keep the resource in consistent states.

Researchers in this field strive to support program designers and developers with tools they can use with relative ease for the synchronization required. As only the memory read and write operations are atomic, constructing a highly concurrent operating system without synchronization tools is a challenging task.

Many synchronization problems are reported in literature that may need different solutions. We will study some important synchronization problems and their solutions in this chapter. We start with defining some well-known synchronization problems first.

7.4 Classical Synchronization Problems

Concurrent processing by several asynchronous processes differs from sequential processing. In concurrent processing, the order of execution of the elementary steps (i.e., operations executions) of different processes is not predetermined. The order may depend on various parameters such as the relative speeds of processes. Interrupts from peripheral devices make things even more unpredictable. Concurrent processes interact with each other accessing shared variables. Synchronizing these accesses avoids race conditions and unwanted time-dependent erroneous behaviour. In the processes where they manipulate shared variables, certain sensitive sections of programs are designated as "critical sections".

A *critical section* is a segment of code where a process accesses a set of related shared variables. For example, a database application updates parts of a database file in a critical section. Different processes have their own critical sections where they manipulate the same set of shared data. For example, in Fig. 7.2 on page 154, the three operations of each process form a critical section. These sections are *critical* in the sense that the processes must not execute the sections without coordination among themselves. Shared data are always accessed inside (and, not outside) critical sections. These critical sections are called *similar* critical sections as they all manipulate the same set of shared variables. For the sake of this chapter, we assume that there is only one critical section shared by all processes, and which we will refer to as *the* critical section. The values of the shared variables determine the state of the critical section. When some process executes (or is in) the critical section, we say the critical section is *busy* or *occupied*; otherwise, the critical section is *free* or *empty*.

The critical section when executed in isolation is assumed to transform itself from one consistent state into another. However when it is executed concurrently by many processes, such a guarantee may not be possible. A specific *synchronization* or *critical section problem* is to order executions of the critical section satisfying some specific constraints on the shared data. So that these constraints are not violated, the processes follow some protocols in their execution. Solutions to synchronization problems entail designing these protocols for processes to be followed before entering and after exiting the critical section. The steps in the protocol are executed in the parts called *entry section* and *exit section*, respectively. The entry section is executed in preparation to enter the critical section, and in the exit section for leaving it.

A process wanting to execute the critical section must first make known its intention to the other processes. In other words, it must acquire permission to enter the critical section. If its immediate entry is not allowed, it must wait until it may enter. A process evinces its interest in the critical section when it starts executing the entry section. Again, at the end of the execution, it must inform other processes by executing the exit section that it has indeed finished.

≫ We may view the critical section as a shared resource. The entry section implements the arbitration logic to "acquire" the critical section and the exit section implements the "release" of the critical section. The entry section implements some non-trivial arbitration logic, but the exit section is normally trivial.

Note that executions of the entry- and exit sections involve communications among the processes, and that the two sections employ a different set of shared variables called *synchronization, coordination*, or *control variables*. These control variables differ from the shared variables proper used within the critical section. (Control variables are solely used to construct the entry- and exit sections.) The processes may, however, concurrently access the control variables. In other words, we allow race conditions to occur for the control variables. These control variables are normally simple shared variables, and the outcomes of the race conditions are somewhat predictable and manageable.

For the sake of this chapter, we logically partition a process address space into four distinct sections, as shown in Fig. 7.3. A process cycles through its remainder, entry, critical, and exit sections until its execution is complete. In the remainder section, a process does something else and does not access the critical section-related shared variables or the synchronization variables. A process can terminate its execution only in the remainder section.

```
while (process is not complete)
{
    <remainder section>;
    <entry section>;
    <critical section>;
    <exit section>;
}
```

Figure 7.3: *Partitions of process address space.*

and not in the other three sections. When a process executes a particular section, we say that the process is "in" that section. A solution to a critical section problem involves designing the codes for entry and exit sections.

We assume that processes are asynchronous, and they execute at a non-zero speed. A process is not stopped when it is supposed to be taking basic steps. That is, we assume that all processes interested in the critical section continue taking steps. However, they proceed at different relative speeds that are not a priori known. Further, in the remainder section, a process may halt its execution or pause for an arbitrary length of time. It is not known which processes are going to request the critical section next or when they will do so. Moreover, we assume that all processes take a finite amount of time to execute the critical section. A process may not stop in the critical, entry, or exit section; otherwise the system as a whole may not be able to make any progress.

A few well-known classical synchronization- or critical section problems are described in the following subsections. They are representatives (or abstractions) of a large number of practical synchronization problems. They differ from one another in the way critical section executions of different processes are ordered. Solutions to these problems are presented in Section 7.5.

7.4.1 The Mutual Exclusion Problem

Mutual exclusion is the most fundamental problem of synchronization—and the most studied—in which the synchronization constraints specify mutually exclusive executions of the critical section. That is, no two processes can execute the critical section concurrently. A process that begins execution of the critical section must finish the execution before another process starts its execution of the critical section.

≫ Mutual exclusion models the problem of managing accesses to an indivisible, non-sharable resource that can only be allocated to one process at a time. For example, a graphics plotter is a physical resource that cannot be used simultaneously by two processes. They must use the plotter mutually exclusively, one after another.

7.4.2 The Producer–Consumer Problem

Producer–consumer is another fundamental synchronization problem. There are two sets of processes—one called *producers* and the other called *consumers*. Producers produce data items that are consumed by consumers. Processes are asynchronous, and they proceed at different non-zero speeds that are not known a priori. Consequently, when a producer produces an item, sometimes no consumer may be ready to receive the item. The producer then has two alternatives to follow:

- **Synchronous communication:** The producer waits until a consumer is ready to receive the produced item. When both are ready, the data exchange takes place.
- **Asynchronous communication:** The producer deposits the item in a known storage space from where a consumer retrieves the item later (see Fig. 7.4).

We consider only asynchronous communications in this book. The storage space used to hold data items is called a *buffer*. The buffer provides a finite storage space to hold unconsumed items. Two primitive operations, namely **put** and **get**, are implemented to access the buffer contents. A producer executes the **put** operation to store an item in the buffer, and a consumer executes the **get** operation to retrieve an item from the buffer. The operation executions by the processes are subject to the following constraints:

- Both the **put** and **get** operations are (or appear to be) atomic.
- Items are received in the order they are put in the buffer.
- When the buffer is full, a **put** execution is blocked until consumers receive some items.
- When the buffer is empty, a **get** execution is blocked until producers put some items.

7.4.3 The Readers–Writers Problem

Readers–writers is another fundamental synchronization problem. It is a special case of the producer–consumer problem described in the previous subsection. The senders are called the *writers*, and the consumers the *readers*. They access a single shared variable. A writer overwrites the previous value of the variable with a new value. A reader returns the current value from the variable without altering the content of the variable. Each writer has an exclusive access right to the variable, but several readers may access the variable simultaneously. That is, a write operation execution is exclusive to other operation executions, but many read operation executions may be concurrent. Unlike the mutual exclusion problem, we allow many readers to concurrently execute the critical section, as they do not change values of the shared variable.

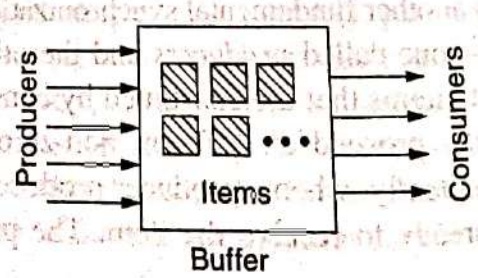

Figure 7.4: *Producer-consumer interactions.*

7.4.4 The Dining-philosophers Problem

The dining-philosophers problem is one of the best-known synchronization problems. Some philosophers are perpetually sitting around a circular table (see Fig. 7.5). They spend their time thinking and eating. A philosopher either thinks or eats, but does not do both at the same time. A philosopher usually thinks, and when she is thinking she does not need any physical resources (chopsticks, in this case). From time to time, each philosopher becomes hungry, and she needs two chopsticks to eat. Eating is a finite time action. As shown in the figure, there is a single chopstick between every adjacent pair of philosophers. A philosopher can pick up the two chopsticks on the table adjacent to her, and not the others. She can however pick up one chopstick at a time. She can eat only when she acquires two chopsticks. If a fellow philosopher holds any of the adjacent chopsticks, she needs to wait until it is available. When she has finished eating her meal, she releases both the chopsticks one by one, and then resumes thinking again. The restriction is that no two neighbouring philosophers can eat at the same time.

7.4.5 The Sleeping-barber Problem

A barbershop has two rooms: a barber's room and a waiting room. The barber's room has a single chair. When there are no customers to serve, the barber goes to sleep on the chair. There are a fixed number of chairs in the waiting room. When the barber is busy serving a customer, new customers sit

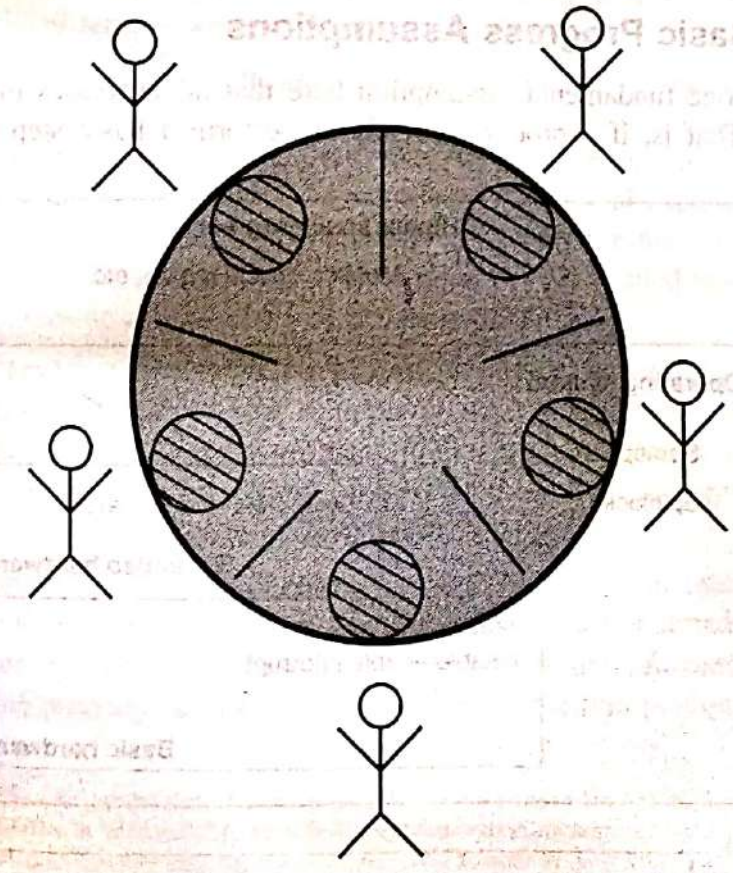

Figure 7.5: *The dining philosophers' round table conference.*

on the chairs in the waiting room, one on each chair, waiting for their turn for the barber. If there are no free chairs, new customers leave the barbershop. If the barber is asleep, the new customer wakes her up. When there is no customer the barber should sleep and when there are customers, the barber should serve them one at a time.

7.5 Synchronization Solutions

In the previous section, we specified several synchronization problems, each requires a different flavour of coordination for executions of critical sections. Different synchronization problems need different types of solutions. Different solutions employ different synchronization primitives. Synchronization primitives are supported at various levels such as basic hardware, extended hardware, operating system, and higher-level programming languages as shown in Fig. 7.6. As operating system experts, we should look at primitives at all levels.

Given a set of synchronization primitives, a solution to a synchronization problem involves designing entry and exit protocols using those primitives, which ensures the desirable properties of the problem. In the following subsections, we study some synchronization solutions based on the read-write, test-and-set, semaphore, spinlock, region, and condition synchronization variables. Before doing so we state a fundamental assumption in Section 7.5.1, and describe various desirable properties of synchronization solutions in Section 7.5.2.

7.5.1 Basic Progress Assumptions

We make one fundamental assumption here that all processes make finite progress. That is, if a process is ready to perform a basic step (aka, one

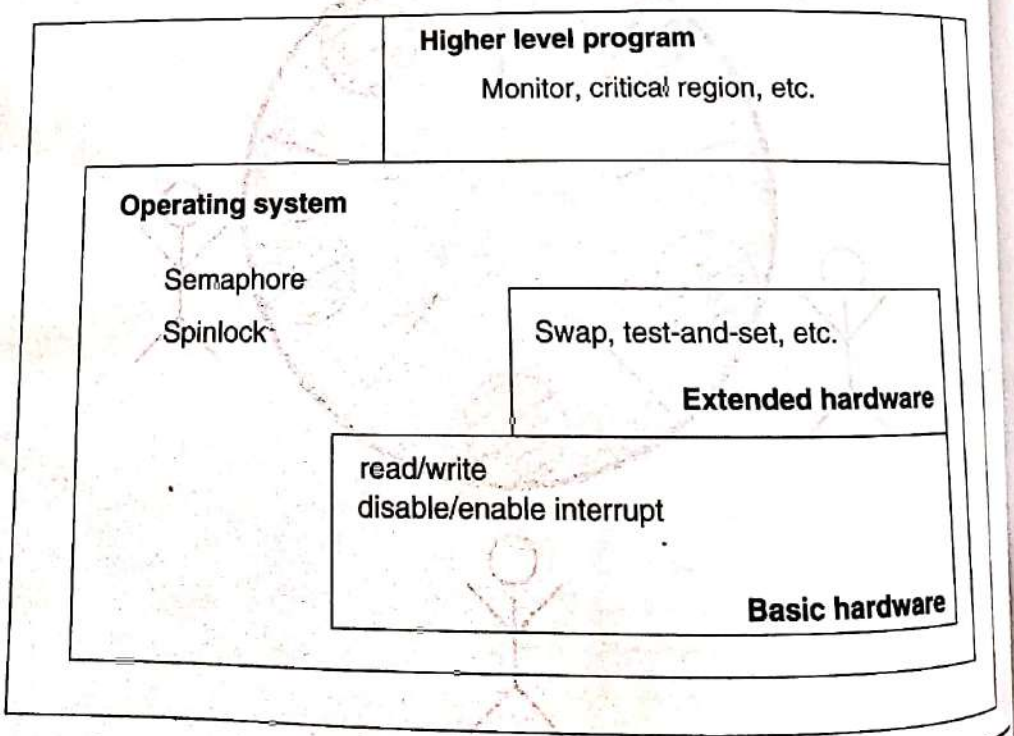

Figure 7.6: *Hierarchy of synchronization primitives.*

machine instruction execution), it does so in a finite time. If a process is infinitely delayed in performing a basic step, then no progress may be ensured for the process. An execution of a set of processes is said to be *fair* if the following two conditions hold for each process P in that set.

- If the execution is finite, then P does not take any basic step at the end of the execution.
- If the execution is infinite, then P performs infinitely many basic steps or P is not interested in taking basic steps infinitely many times.

Without the basic progress assumption, some synchronization properties may not be possible to achieve. We will explicitly point out the use of this basic progress assumption in presenting synchronization solutions.

7.5.2 Solution Characteristics

First, any synchronization solution should correctly satisfy the intent of the problem it is solving. That is, nothing untoward should happen during its execution. This is called a *safety* or *correctness property*. Secondly, the solution should assure some other useful qualities such as making progress, maintaining fairness, etc. We list these properties here.

1. **Safety:** The intended condition will never be violated. For example, a solution to the mutual exclusion problem should never violate the mutual exclusion property. For another example, a solution to the readers-writers problem should never allow a reader to enter the critical section while a writer is engaged in it.
2. **Liveness:** The system should not be blocked forever due to contention for the critical section. That is, contention due to synchronization must be resolved in a finite time so that the system can progress eventually. For example, a solution to the mutual exclusion problem should assure that some process competing for the critical section enters it in a finite time. This property is also called the *finite arbitration-* or *non-blocking* or *progress* property.
3. **Starvation Freedom:** No process should be denied progress forever due to contention. That is, not only some processes, but all processes must eventually make progress. For example, a solution to the mutual exclusion problem should assure that every interested process enters the critical section in a finite time.
4. **k-bounded Bypass:** No process should be allowed to enter the critical section more than k times when another process is already waiting to enter the critical section. This is also called the *k-fairness property*.[1] This property is stronger than starvation freedom property.

≫ *Starvation* is a situation in which one or more processes wait indefinitely in the entry section, and other processes overtake the waiting processes in entering the critical section.

[1] We need to be a bit careful here. We are talking now of some higher-level fairness in critical section executions. Earlier, in Section 7.5.1, we talked about lower-level fairness in basic execution steps. In general, lower-level fairness may not ensure higher-level fairness in critical section executions.

5. **FIFO Fairness:** No process will ever overtake another waiting process. This is the 0-fairness property.
6. **LRU Fairness:** The process that received the service least recently gets the service next.

Among these properties, safety, liveness, and starvation freedom are considered essential properties of any synchronization solution and the remaining are considered desirable properties. Further, an efficient solution needs to assure that a process does not consume CPU time unnecessarily waiting at the entry section.

7.5.3 Solutions to the Mutual Exclusion Problem Using Atomic Variables

We view each memory cell as an ordinary read/write variable. In a single step, a process can either read or write a single cell, but cannot do both; neither can it do so on many cells. Reading and writing each memory cell is viewed as a special case of the readers-writers synchronization problem. The memory hardware arbitration circuit solves this special problem. The arbitration circuit minimally ensures that a read and a write on the same cell exclude each other in time, and also multiple *writes* on the same cell exclude one another in time. When simultaneous conflicting requests are made to the same memory cell, the hardware allows only one of them to proceed and the remaining ones have to wait until the one granted is complete. That is, when two or more conflicting memory operations are executed concurrently, the outcome of the execution is equivalent to their sequential executions in some arbitrary order. One operation execution proceeds while others wait, and these other calls are said to have stalled. That is, this hardware-based solution has a kind of busy waiting property, because the caller remains stalled or blocked until its memory operation (read or write) is complete. The memory hardware though may allow concurrent executions of operations on different cells.

Without loss of generality, we assume, in the rest of this subsection that read and write operation executions on memory cells (primitive variables) are atomic and terminating. We would like to solve various synchronization problems using these two primitives. The race condition, displayed in Fig. 7.2 on page 154, could be avoided if processor architecture supports more powerful atomic instructions than plain read and write instructions. That is, with powerful atomic instructions such as read-increment-write and read-decrement-write, we would be able to solve the synchronization problem stated in Fig. 7.2 trivially: one atomic operation on the shared integer variable replaces the three operations.

All processors support atomic read and write instructions, and some, in addition, support atomic read-modify-write, test-and-set, swap, compare-and-swap, fetch-and-add, and other instructions. The latter operations are made up of many simpler actions, but their atomicity is ensured by the underlying

hardware. For example, the read-modify-write operation on a shared variable x is made up of three actions, in this sequence: (1) read x, (2) evaluate some function using the value of x, and (3) write a new value in x. These three actions must be performed in that sequence in one indivisible execution of the read-modify-write operation, and the underlying hardware ensures this.

A shared variable accessed only by executing atomic operations is called an *atomic variable*. If read and write are the only permitted operations on a variable, we call it an *atomic read-write variable*. If in addition test-and-set operation is permitted, we call it an *atomic test-and-set variable*.

Operating systems normally contain many non-atomic data structures that cannot be manipulated indivisibly by applying atomic operations. Compound operations are needed to manipulate these data structures. It is often not possible to implement atomic compound operations using atomic read-write variables. Consequently, data structures are manipulated in critical sections. We present below a number of solutions to the mutual exclusion problem. Each solution employs atomic variables to implement entry and exit section protocols. Subsections A Naive Two-process Solution to The Lamport Solution present solutions using atomic read-write control variables, and subsections A Solution Using test-and-set Operation to A Solution Using swap Operation using stronger control variables.

If the code of a synchronization solution for the interacting processes is identical except the reference to the process id, which is the typical case for most synchronization algorithms, it is customary to write the code for a generic process with id i. In those subsections, we present solutions for a typical process P_i, $0 \leq i < n$, where n, greater than 1, is the number of concurrent processes.

≫ Intel 80386 processor has atomic inc/dec instructions. They work fine on single CPUs, but are ineffective for multiple CPUs. The processor also supports lock-prefixed instructions, for example, Lock dec or Lock inc. A lock-prefixed instruction blocks all other CPUs from accessing the system bus until the instruction execution is complete.

A Naive Two-process Solution

A very simple solution to the mutual exclusion problem for two processes, say P_0 and P_1, is presented in Fig. 7.7. The solution uses a single atomic boolean control variable *turn*, whose value is initialized to 0.[2] (The variable could also be initialized to 1.) Process P_i uses two local constants *me* and *other*, and *me* is set to i and *other* to $1 - i$. The idea is that a process enters the critical section only when it is its turn. The entry section contains a single while-loop statement, where process P_i repeatedly reads *turn* and tests whether the value indicates its own turn to enter the critical section. If the *turn* value is i, it enters. The exit section contains a single assignment statement where *turn* is set to *other*, indicating that the other process can enter the critical section.

What properties does the solution possess? It ensures the mutual exclusion (aka, the safety) property and 1-bounded bypass property; however, it may not ensure the liveness property. The solution forces processes to take turns alternately to the critical section. If one process completes its execution

[2]The volatile specifier is an indication to compiler not to optimize or reorder accesses to the variable.

```
Data structures and initial values
    shared volatile boolean turn = 0; /* shared by P0 and P1 */
    const local boolean me = i;       /* for process Pi */
    const local boolean other = 1 - i;

Solution
    while turn  me ≠ do; // entry section
    ⟨ Critical section; ⟩
    turn = other;        // exit section
```

Figure 7.7: *A naive solution to the 2-process mutual exclusion problem.*

or becomes uninterested in the critical section, the other process will be blocked forever in the entry section. For example, assume *turn* is 0, P_0 is not interested in the critical section, but P_1 is. P_1, however, cannot enter the critical section until the *turn* becomes 1. The turn value can become 1 only if P_0 becomes interested in entering the critical section (that is not known), enters the critical section, and changes the value of *turn* at the exit section. So we cannot assure that whether P_1 will eventually enter the critical section or not.

The main problem with this solution is that one process does not have sufficient information about the other process for meaningful arbitration. The sufficient information in this case is the status of the other process. Since the boolean variable *turn* can store only information about who can enter the critical section next, if both become interested at the same time, we need more control variables to store the status of the processes.

The Dekker Solution

The previous solution to the two-process mutual exclusion is correct but not satisfactory. The first correct and satisfactory solution to the 2-process mutual exclusion problem is due to Dekker, a Dutch mathematician. The Dekker solution is presented in Fig. 7.8. It is an extension of the naive solution of Section A Naive Two-process Solution. As in the naive solution, we denote two processes by P_0 and P_1. The Dekker solution uses three shared boolean variables: *status[0]*, *status[1]*, and *turn*. The *status* variables are initialized to *false*, and the *turn* to 0. Each *status* variable is written by one process and read by the other process, and they can do so at the same time. However, the *turn* variable is read and written by both the processes. The protocol structure ensures that *turn* is never written by both processes at the same time.

When a process P_i becomes interested in the critical section, it sets *status[i]* to *true* first to indicate its interest in the critical section. P_i then executes an arbitration algorithm to resolve any potential conflict of interests. If the other process is not interested in the critical section, that is, the *status[other]* value is *false*, then P_i enters the critical section directly. Otherwise, there is a conflict, and the tie is broken by allowing the process that did not enter the

```
Data structures and initial values
    shared volatile boolean status[2] = false; /* shared by P0 and P1
    shared volatile boolean turn = 0;          /* shared by P0 and P1 */
    const local boolean me = i;                /* for process Pi */
    const local boolean other = 1 - i;

Solution
    status[me] = true; /* show interest in the critical section */
    while (status[other]) do /* busy wait arbitration */
    {
        if (turn == other) { /* turn of the other process */
            status[me] = false;/* temporary withdrawal */
            while (turn == other) do; /* busy wait until my turn */
            status[me] = true;
        }
        /* else it is my turn, wait until status [other] becomes false */
    }
    ⟨ Critical section; ⟩
    turn = other;
    status[me] = false; /* no more interested in the critical section */
```

Figure 7.8: *The Dekker solution to the 2-process mutual exclusion problem.*

critical section for the longest time (the LRU fairness). This is indicated by the value of *turn*, and whoever has the *turn* wins the tiebreak and enters the critical section.

Consider the case when *turn* value is equal to *0* and both processes have set their *status* value as *true*. Now there are three possible scenarios: (1) both are in the entry section; (2) P_0 has already crossed (i.e., finished) the entry section before P_1 set its *status* as *true*; (3) P_1 has already crossed the entry section before P_0 set its *status* as *true*. There is no way that a process can know in which scenario it is. Although P_0 has the current turn, it cannot simply cross the entry section because safety is violated if it is in scenario (3). Therefore, P_0 must confirm that P_1 has not already crossed the entry section. This information can come only from P_1. For that, P_1 temporarily withdraws from the competition by setting its *status* to *false* so that the process P_0 that has the current turn can go ahead and enter the critical section. After the temporary withdrawal from the competition, P_1 simply waits for its turn by repeatedly checking for it. When P_0 has finished executing the critical section, it sets the value of *turn* to *other* (i.e., to 1), and resets *status[0]* to *false*. When the *turn* value is equal to 1, P_1 restarts its competition by setting its *status* to *true* and continues.

This solution ensures the properties of mutual exclusion, liveness, and starvation freedom. The solution assures the liveness and starvation-freedom properties under the assumption of finite (non-zero) progress in taking basic steps by both processes. However, it does not have the *k*-bounded bypass property; during a temporary withdrawal from the competition, theoretically, there is no bound on how many times the other process can cross the entry code and enter the critical section.

The Peterson Solution

Figure 7.9 presents a new solution to the 2-process mutual exclusion problem. It is substantially simple compared to the Dekker solution, and uses the same number and types of shared variables. It uses three shared boolean variables: *status[0]*, *status[1]*, and *turn*. The *status* variables are initialized to *false*, and the *turn* to 0. Each *status* variable is written by one process and read by the other process, and they can do so at the same time. However, the *turn* variable is read and written by both processes. Unlike the Dekker solution, here both processes can write the *turn* variable at the same time. The outcome of simultaneous accesses is determined by the atomicity property of *turn*. The idea is simple: when both processes are competing, the later process (which writes on *turn* latest) "pushes" the former process (which writes on *turn* first) out of the entry code (the while-loop) by changing the *turn* value. The later process could exit the waiting while-loop as soon as the former process completes and sets its *status* to *false*.

When a process P_i becomes interested in the critical section, it sets *status[i]* to *true* first to indicate the other process its interest in the critical section. P_i then gives the other process (referred to as P_j, $j = 1 - i$) a preferential treatment to enter the critical section if the other process is also interested in executing the critical section. (If both processes attempt to enter the critical section at the same time, it is possible that they both write *turn* simultaneously. The atomicity property of *turn* will determine its final value.) P_i then executes an arbitration algorithm to resolve any potential conflicts. The arbitration logic here is a very simple while-loop statement. P_i repeatedly reads *status[j]* and *turn* values (in arbitrary order), and checks their values. P_i waits on the while-loop until either P_j is not currently involved in the competition to enter the critical section or *turn* $= i$. When P_i has finished executing the critical section, it resets *status[i]* to *false*.

This solution ensures the properties of mutual exclusion, liveness, starvation freedom, and bounded bypass (1-fairness). The solution assures the liveness property under the assumption of finite progress in taking basic steps by the two processes.

```
Data structures and initial values
    shared volatile boolean status[2] = false; /* shared by P0 and P1 */
    shared volatile boolean turn = 0;          /* shared by P0 and P1 */

    const local boolean me = i;                /* for process Pi; */
    const local boolean other = 1 – i;

Solution
    status[me] = true; /* show interest in the critical section */
    turn = other;
    while (status[other] and turn == other) do; /* busy wait arbitration */

    ⟨ Critical section; ⟩

    status[me] = false; /* no more interested in the critical section */
```

re 7.9: The Peterson
ion to the 2-process
l exclusion problem.

The Lamport Solution

Dekker was the first to solve the mutual exclusion problem satisfactorily. However, his solution is for only two processes. Dijkstra was the first to propose a solution to the general n-process, $n > 1$, mutual exclusion problem. The Dijkstra solution is inefficient, and it does not satisfy the stronger properties such as starvation freedom. Here, we present a simpler solution to the general n-process mutual exclusion problem. The solution, due to Lamport, is known as the *Bakery algorithm*. His solution is presented in Fig. 7.10. It uses n boolean variables *choosing*, all initialized to *false*, and n integer variables *number*, all initialized to 0. Each variable is written by one process and read by others. The values of integer variables are used to form token numbers, the kinds that are used in bakery shops (in USA)! A customer on entering the bakery chooses a token number, and within the bakery, customers are served in the order of their token numbers.

When a process becomes interested in the critical section, it chooses a token number greater than those held by other interested processes. The first three statements in the entry section comprise a "doorway" where the process chooses a token number. This is done by reading all *number* variables in arbitrary order. To make other processes aware that it is choosing a token number, a process P_i sets *choosing[i]* to true before reading *number* variables and sets *choosing[i]* to false after reading them and writing its own token number that is one greater than the max of others. After choosing its token number, process P_i enters the arbitration section by a way of executing the for-loop statement where it evaluates the status of other processes. If any other process P_j is choosing its token number, then P_i busy-waits until P_j has obtained its token number. The arbitration logic allows the process with the lowest token number to enter the critical section first. The original process then waits until all processes with lower token numbers are served by the arbitration logic. Because of the race condition at the doorway, it is possible that several processes pick up the same

```
Constants
    n = number of processes, where n > 1;

Data structures and initial values
    shared volatile boolean choosing[n]; /* initialized to false */
    shared volatile int number[n];       /* initialized to 0 */

Solution
    choosing[i] = true; /* show interest in the critical section */
    number[i] = 1 + max(number[0],···, number[n - 1]); /* get a token number */
    choosing[i] = false;
    for (int j = 0; j < n; j = j + 1) { /* busy wait arbitration */
      while (choosing[j])do; /* wait until Pj gets its token */
      while (number[j]≠0 and (number[j],j) < (number[i],i))do;/ token comparison
    }
    ⟨ Critical section; ⟩
    number[i] = 0;
```

Figure 7.10 Lampo
Bakery algorithm fo
process P_i.

token number. The tie is broken by process indexes; the ones with lower indexes are served first. In the solution, we use the relation $\langle a, b \rangle \prec \langle c, d \rangle$ to mean $(a < c)$ or $(a = c)$ and $(b < d)$. When a process has finished the critical section execution, it resets its *number* variable to 0.

This solution ensures the properties of mutual exclusion, liveness, and bounded bypass. Actually, the solution ensures the FIFO fairness property after a wait-free[3] doorway: if process P_i completes the doorway before process P_j starts the doorway, then P_j cannot enter the critical section before P_i does. The only shortcoming with this solution, at least theoretically, is that the *number* values may grow without any bound.

A Solution Using test-and-set Operation

In the previous four subsections we studied solutions to the mutual exclusion problem for processor architectures that support only atomic read and write instructions on primitive variables. Some processor architectures implement special machine instructions that allow the CPU to test and modify the content of a single primitive variable, or to swap the contents of two primitive variables, atomically. These operations are more powerful than ordinary read and write operations, and make programming tasks a little easier. In this subsection, we study a solution that uses test-and-set atomic instruction, and in the following subsection, a solution that uses swap instruction. The advantages of these machine instructions are their simplicity and they work for any number of CPUs. They help us developing simple, easy-to-understandable solutions.

There are many variations of test-and-set operation semantics. Figure 7.11 presents one variation in algorithmic form. The operation takes address of a boolean variable, and a boolean value. It writes the new value to the address and returns the old value stored at the address. The reading of the old value from a variable and setting of the new value to the same variable are done in a single *indivisible* operation, and not in two different atomic operations as shown in the figure. The underlying processor and memory hardwares ensure the indivisibility. If two or more CPUs concurrently execute the test-and-set operation on the same variable, the operation executions appear as if they were executed in a total (arbitrary) order.

A solution to the mutual exclusion problem is presented in Fig. 7.12. The solution uses a single shared boolean variable *lock* that is initialized to *false*. (The value *false* indicates that the lock is free, and *true* indicates that the lock is taken.) The entry section contains a single while-loop statement where a process sets the value of *lock* to *true*. If the original value of *lock* is *false*, the process breaks the while-loop and enters the critical section. Otherwise, it keeps trying to obtain the lock. In the exit section, it resets the value of *lock* back to *false*.

This solution is simple compared to those presented in previous subsections, and it scales for arbitrary number of processes, that is, the structure

[3]Wait-freedom means that a process that has started a computation is guaranteed to complete the computation if the process continues to take its own steps, regardless of what other processes do

```
boolean TestAndSet(boolean* lock, boolean newValue)
{
  boolean oldValue;
  oldValue = *lock;
  *lock = newValue;
  return oldValue;
}
```

Figure 7.11: *Semantics of test and set instruction.*

of the solution is independent of the number of processes. The solution ensures the properties of mutual exclusion and liveness, but does not ensure the starvation freedom property, and hence, does not ensure the bounded bypass property.

≫ The more power primitive operations have, the easier it becomes to construct a synchronization solution.

A Solution Using Swap Operation

The semantics of swap operation is presented in Figure 7.13 in algorithmic form. The operation takes two memory addresses, and exchanges their contents in a single indivisible operation execution. The underlying processor and memory hardwares ensure the atomicity. Normally, one of the addresses is shared, and the other one is local.

A solution to the mutual exclusion problem is presented in Fig. 7.14. (It is similar to the one in Fig. 7.12.) The solution uses a single shared boolean variable *lock* that is initialized to *false*. Each process uses a private boolean variable *key* that is initialized to *true*. The entry section contains a single while-loop statement where a process exchanges the values of *lock* and *key*. If the original value of *lock* is *false*, the process breaks the while-loop and enters the critical section. Otherwise, it repeatedly tries to take the lock. In the exit section, the process sets the value of *lock* to *false*.

This solution is as simple as the one in the previous subsection. It ensures the properties of mutual exclusion and liveness, but does not ensure the starvation freedom property, and hence, does not ensure the bounded bypass property.

7.5.4 Interrupt Disabling

Interrupt disabling is a means to coordinate critical section executions in the kernel space and not in the user space, because application processes do not have

```
Data structure and initial value
    shared volatile boolean lock = false; /* shared by all processes */

Solution
    while(TestAndSet(&lock,true)) do;
    ⟨ Critical section; ⟩
    TestAndSet(&lock, false);
```

Figure 7.12: *Mutual exclusion using a TestAndSet variable.*

```
void Swap(boolean* locA, boolean* locB)
{
   boolean temp;
   temp = *locA;
   *locA = *locB;
   *locB = temp;
}
```

Figure 7.13: *Semantics of swap instruction.*

control over the interrupt system. The kernel controls executions of the critical section by manipulating the processor interrupt facilities. For this scheme, we need instructions to enable and disable the interrupt system. The entry section is a single interrupt disable instruction, and the exit section a single interrupt enable instruction (see Fig. 7.15). That is, the CPU disables the processor interrupt circuit before entering the critical section, and re-enables the circuit after exiting the critical section. As the interrupt circuit is disabled when a process executes the critical section, the CPU cannot be interrupted and preempted from the running process and allocated to another process. Consequently, the critical section is always executed mutually exclusively.

One potential weakness of this scheme is that if the critical section is long, then the interrupt circuit is disabled for a lengthy duration soon. Consequently, many peripheral devices may sit idle waiting to get service from the CPU, and thereby, performance of the system as a whole may degrade. Though this technique of handling critical section executions can be used in uniprocessor systems, it is ineffective in multiprocessor systems, because disabling interrupts by one CPU may not disable interrupts in all CPUs. Disabling and re-enabling interrupts on all CPUs could be time-consuming tasks if not impossible. Many practical operating systems use interrupt disabling on local CPUs in conjunction with other synchronization mechanisms for multiprocessor systems.

7.5.5 Non-preemptive Kernels

In a system with non-preemptive kernel, when a process executes the kernel, the system cannot arbitrarily stop the process and start executing another process. Unless the running process voluntarily releases the CPU, the process will have

```
Data structure and initial value
    shared volatile boolean lock = false; /* shared by all processes */

Solution
    local boolean key = true;
    while (key) Swap(&lock, &key);
    ⟨ Critical section; ⟩
    Swap(&lock, &key);
```

Figure 7.14: *Mutual exclusion using a swap variable.*

```
disable interrupt;  /*entry section */
( critical section; )
enable interrupt;  /* exit section */
```

Figure 7.15: *A solution to the mutual exclusion problem using disable/enable interrupt.*

the CPU until it leaves the kernel. Consequently, when a process executes a system call, it is assured that the CPU will not be taken away from it; but devices can interrupt the process. The data structures that are not modified by interrupt or exception handlers are guaranteed to be free from race conditions. Consequently, critical sections that lie outside the reach of interrupt and exception handlers are always executed mutually exclusively. If a process in the kernel voluntarily releases the CPU, it has to ensure that the data structures are in a consistent state. Critical sections that are executed by the interrupt and exception handlers can be controlled either by using synchronization primitives discussed in other subsections or by preventing the nested interrupt from occuring.

Though this technique is somewhat effective in some uniprocessor systems, it is in general ineffective in multiprocessor systems where many kernel paths (due to system calls) simultaneously execute kernel programs and access kernel data.

≫ There is a related tool called *mutex*. Though it is the same as a semaphore, it is used slightly differently. For mutex, only a token holder can execute an up operation, and not others.

7.5.6 Semaphores

We observed in Section 7.5.3 on page 162 that solutions to the mutual exclusion problem using atomic (read/write) variables are difficult to construct and verify. We did not even attempt to solve other synchronization problems using atomic variables. Atomic variables are not sophisticated enough to be used to solve complex synchronization problems efficiently. In addition, those solutions waste a lot of CPU time as they employ "busy waiting" or "continual retry" when processes cannot immediately enter the critical section: as long as a process is in the critical section, any other process that attempts to enter the critical section loops continuously in the entry section until it can enter the critical section. Busy waiting requires that the values of synchronization variables are repeatedly read and checked until they have the desired values: the CPU loops on synchronization variables. Busy waiting is a huge burden in practical systems where CPUs are shared by many active processes. In uniprocessor systems that prohibit process preemption, busy-waiting solutions become ineffective because when a process loops on a condition the whole system stalls. Even in multiprocessor systems, busy waiting causes a flood of memory requests that may choke the system bus or "processor to memory" network. Using a different synchronization tool called *semaphore* overcomes these shortcomings. Another motivation behind semaphore is to reduce complexity of the programming required to implement solutions to synchronization problems, especially the mutual exclusion problem. It is a very important synchronization tool, usually supported by operating systems. What, then, exactly is a semaphore?

A process needs coordination when its further progress does not guarantee safety. In such situations, tokens are often used as safety certificates. That is, progress of a process can guarantee safety when it holds a token.

```
struct semaphore {
    int       count;/* number of tokens in the semaphore */
    process* waitQ;/* processes that are waiting to receive tokens */
};
```

Figure 7.16: *Semaphore structure definition.*

Here, a semaphore is considered roughly as a token manager. A process requests a semaphore for a token, proceeds further after obtaining one, and returns it to the semaphore when no longer needed. When a process requests a token from a semaphore, it (the semaphore) issues the process a token if it has one. Otherwise, the requesting process is blocked until a token is available. The semaphore accepts the token as soon as a process returns one. Thus, the token request is a blocking operation, but not the token return. Actually, as all tokens are the same, the semaphore manager, instead of keeping them in semaphore storage space, keeps only a count on the number of available tokens, and manipulates their number.

≫ A semaphore is a special case of message-based interprocess communication, where senders send tokens to the semaphore, and receivers retrieve the tokens out of the semaphore. Unreceived tokens remain in the semaphore. Here, no two tokens are different.

Essentially, a *semaphore* is a shared data structure that has two member components (see Fig. 7.16). The first component is an integer variable, referred to as *count*, which takes value from a range of integers to indicate the number of tokens the semaphore has. If the *count* value is zero, the semaphore is said to be empty. The second component is a wait queue, referred to as *waitQ*, to hold the processes waiting to receive tokens from the (empty) semaphore. The initial value of *count* defines the initial number of tokens. Usually, a semaphore is created with a fixed number of tokens.

A semaphore structure, apart from initialization, is accessed only by invoking two "atomic" operations, namely **down** and **up**, respectively, for token request and token release. [Many names are used for these two operations. Some popular names are P and V (first letters of equivalent words for down and up in Dutch), acquire and release, wait and release, wait and signal, etc.]

When a token is available, a decision must be made about which waiting process to select for issuing that token. This decision in essence defines the service discipline of *waitQ*. Some service disciplines such as FIFO may be considered fair and others such as random selection unfair because they may lead to starvation of a process. Based on the *waitQ* service discipline, semaphores are classified into various types. The study of such classification is beyond the scope of this book.

The semantics of the two semaphore operations are sketched in Fig. 7.17 in algorithmic forms. A process takes a token out of a semaphore by invoking the **down** operation on the semaphore, and inserts a token into a semaphore by invoking the **up** operation on the semaphore. In a **down** operation execution, a process attempts to take a token out of the semaphore. If there are tokens in the semaphore (i.e., $count > 0$), the process removes one token by a way of decrementing the *count* component by one. Otherwise, (i.e., $count = 0$), the process waits on the semaphore queue until it receives a token. It is not a busy wait. The process blocks itself, and releases the CPU while it is waiting at the *waitQ*. In an **up** operation execution on a semaphore, a process first

```
/* Get a token from semaphore sem */
void down(semaphore* sem)
{
    if (sem-> count > 0) {
        sem-> count = sem -> count - 1;
    }
    else {
        put the calling process in the sem -> waitQ;
        block; /* invoke CPU scheduler to release the CPU */
        /* process returns here when rescheduled and it has got a token */
        remove the calling process from the sem -> waitQ;
    }
    return;
}

/* Return a token to semaphore sem */
void up(semaphore* sem)
{
    if (not empty sem-> waitQ) { /* allocate token to a waiting process */
        select a process from the sem-> waitQ;
        awake the selected process;
    }
    else { /* put the token in the semaphore */
        sem-> count = sem-> cont + 1;
    }
    return;
}
```

Figure 7.17: *A typical semaphore implementation.*

checks if there are any processes waiting at the *waitQ*. If there are some, one of them is awakened. (It is a positive wakeup, and the awakened process now has a token, and is ready to run again.) Otherwise, the semaphore member variable *count* is incremented by one.

A typical problem with some semaphore implementation is starvation. Starvation is a situation in which one or more processes wait indefinitely at the semaphore queue and new **down** operations overtake the waiting process(es). Fairness of a semaphore implementation is determined by the scheduling policy followed in the management of waiting processes. Most semaphore implementations are assumed to exhibit the fairness property, that is, no process while executing the **down** remains delayed forever if **up** operations are performed frequently by other processes. The need for fairness arises when many processes are simultaneously delayed, all attempting to execute the **down** operation on the same semaphore. Clearly, the implementation must choose which process is allowed to proceed when an **up** operation on that semaphore is ultimately performed.

≫ The core semantics of a semaphore is independent of its queue service discipline.

It is of utmost importance that the two semaphore operations, **up** and **down**, are atomic. That is, the operation executions on the same semaphore variable must exclude one another. This situation itself is a mutual exclusion problem; however, the critical section (containing **up** and **down** implementations) is very small. We can solve the problem using a suitable solution discussed in Sections

7.5.3 to 7.5.4; they do involve busy waiting albeit for a shorter duration. As the critical section is very short, busy waiting occurs rarely in practice.

≫ There is a related tool called *mutex*. Though it is the same as a semaphore, it is used slightly differently. For mutex, only a token holder can execute an up operation, and not others.

Based on the range of values that *count* can take, semaphores are classified into binary-, general-, and counting semaphores. The binary semaphore takes the count value as either 0 or 1; the general semaphore takes any non-negative value; and the counting semaphore takes any integer value. In the counting semaphore, the positive value of *count* indicates the number of tokens available and the negative value of *count* indicates the number of processes waiting for tokens. These semaphores offer flexibility and convenience, but they are all equivalent when the *waitQ* is the same. That is, one can be implemented using the other. In the case of the counting semaphore, *count* is always updated (decremented for **down** operation and incremented for **up** operation). If the resulting value of *count* is negative, then the process is put into *waitQ* for **down** operation, and a process from *waitQ* is chosen and awakened for **up** operation.

≫ A binary semaphore acts like a switch; the semaphore is either open or close. It can hold, at most, one token. Counting and general semaphores are useful when there is a need to manage multiple identical copies of a shared resource.

Semaphores are normally used in synchronizing unconditional ordering of critical section executions. Whenever two or more processes need to synchronize their relative speeds, we can also use a semaphore to block fast processes. A few semaphore-based solutions to critical section problems are discussed in the following subsections.

A Solution to the Mutual Exclusion Problem

Semaphores are widely used to control entries to critical sections. The semaphore counter is used as a locking mechanism. A semaphore-based solution to the mutual exclusion problem is presented in Fig. 7.18. This solution is very simple compared to those constructed using atomic variables in Section 7.5.3. It uses a single semaphore variable whose *count* component is initialized to 1 and waitQ is initialized to *NULL*. By initializing the counter value to 1, it is possible to prevent more than one process from entering the critical section at a time. The entry section consists of a single **down** operation on the semaphore, and the exit section a single **up** operation on the semaphore. The solution ensures the properties of mutual exclusion and liveness. The fairness property depends on the implementation of the *waitQ* management.

A Solution to the Producer–Consumer Problem

The producer–consumer problem introduced in Section 7.4.2 on page 157 is solved easily by using semaphores. We have a buffer with n, $n > 0$, slots

```
Data structure and initial value
semaphore sem = {1, NULL};

down(&sem);  /* entry section */
⟨ critical section; ⟩
up(&sem);    /* exit section */
```

Figure 7.18: *A solution to the mutual exclusion problem using a semaphore.*

```
Data structures and initial values
    semaphore empty = (n, NULL); /* n = number of slots in the buffer */
    semaphore full = (0, NULL);
    semaphore mutex = (1, NULL);

void put(Item* m)
{
    down(&empty);
    down(&mutex);
    ⟨add item m in the buffer;⟩
    up(&mutex);
    up(&full);
}
void get(Item* m)
{
    down(&full);
    down(&mutex);
    ⟨remove an item from the buffer;⟩
    up(&mutex);
    up(&empty);
}
```

Figure 7.19: *Implementation of a bounded buffer using semaphores.*

to hold unconsumed data items. The buffer is accessed by two operations, namely **get** and **put**. A general schematic of **get** and **put** operations is given in Fig. 7.19. The solution uses three semaphores: *empty*, *full*, and *mutex*. The *empty* and *full* semaphores count the number of empty and full slots, respectively, available in the buffer. Their initial values are n and 0, respectively, indicating that initially there is no data item in the buffer and all slots are empty. These two semaphores control the number of simultaneous executions of **put** and **get** operations, respectively. The *mutex* semaphore is used to ensure mutually exclusive access to the buffer; the semaphore is imitalized to 1.

The put operation first removes a token from the *empty* semaphore by way of reserving a free slot in the buffer. Then, it adds an item in the buffer, which is guarded by the *mutex* semaphore. It then inserts a token into the *full* semaphore to inform receivers that there is a new item in the buffer. The implementation for the get operation is symmetric.

A Solution to the Readers–Writers Problem

Solutions to the readers–writers problem of Section 7.4.3 on page 158 are generally asymmetric. To enter the critical section, either readers get priority over writers or vice versa. Therefore, there are a few varieties of solutions to this problem: (1) If a writer is ready to execute the critical section, no new readers may enter the critical section before the writer completes its critical section execution; (2) No reader waits to enter the critical section until a writer has obtained permission to do so. We study a solution for the latter variety in this subsection.

```
Data structures and initial values
    semaphore writeSem = {1, NULL};
    semaphore readSem = {1, NULL};
    int readCount = 0;

void write(Value val)
{
    down(&writeSem); /* exclude other writers */
    ⟨write val in the shared variable proper;⟩
    up(&writeSem);
}
Value read()
{
    Value val;
    down(&readSem); /* exclude other readers */
    readCount = readCount + 1;
    if (readCount == 1) down(&writeSem); /* special reader: exclude writers */
    up(&readSem);
    ⟨read val from the shared variable proper;⟩
    down(&readSem);
    readCount = readCount – 1;
    if(readCount == 0) up(&writeSem); /* special reader */
    up(&readSem);
    return val;
```

Figure 7.20: *A solution to the readers–writers problem using semaphores.*

Figure 7.20 presents a typical solution to the readers-writers problem. The solution uses two semaphores, namely *writeSem* and *readSem*, and an integer variable *readCount*. The semaphores are initialized to 1, and *readCount* to 0. The solution uses the *writeSem* semaphore to exclude writers one another in the **write** operation. The number of *active* readers (that are in the entry section or the critical section) is stored in the *readCount* variable. The *readSem* semaphore controls accesses to this synchronization variable. The read operation is simple: it increments the *readCount* under the *readSem* semaphore, performs the actual **read** proper, and finally decrements the *readCount* again under the *readSem* semaphore. A reader that sees *readCount* = 0 at the beginning of the entry section or at the end of the exit section, however, is special and has a special task to perform. In the entry section, a special reader obtains the *writeSem* semaphore to block new writers in entering the critical section, and in the exit section a special reader releases the *writeSem* semaphore. Readers that enter or leave the critical section while other readers are present in the critical section ignore the *writeSem* semaphore. The solution is not free from starvation even with FCFS semaphores as readers keep on entering the critical section may perpetually overtake the writers. See the Literature section at the end of this chapter for starvation-free solutions to the readers–writers problem.

A Solution to the Dining-philosophers Problem

The dining philosophers problem was introduced in Section 7.4.4 on page 159. Here chopsticks are the resources allocated to hungry philosophers. A simple solution is to represent each chopstick by a semaphore initialized to 1. A hungry philosopher first takes the left chopstick, and then the right chopstick. Having finished eating, she releases the left chopstick first, and then the right chopstick. This solution is simple and ensures the safety property that no two neighbouring philosophers can eat simultaneously; it is, however, not free from deadlock. A deadlock occurs when all the philosophers become hungry at the same time, and each picks up her left chopstick and waits indefinitely to get the right chopstick. (The definition of deadlock and a discussion on its issues appear in Section 7.6.)

It has been shown in the synchronization literature that there is no symmetric[4] solution to the dining philosophers problem. Then, to solve the problem we need to break symmetry, and there are several ways to do this: different processes may use different algorithms or randomization. A simple solution is presented in the following paragraph.

Processes are partitioned into two categories, and they execute slightly different algorithms. We assume that the number of philosophers is even, and are numbered consecutively starting from zero, say clockwise. All even numbered processes form one category and odd numbered processes the other. Odd numbered processes pick up the right chopstick first, and the even numbered processes the left. This solution ensures the safety and deadlock freedom properties. If semaphores are FIFO, it also ensures the starvation freedom property.

A Solution to the Sleeping-barber Problem

In Fig. 7.21 we present a solution to the sleeping barber problem described in Section 7.4.5 on page 159. The solution uses three semaphores: (1) **customer** to coordinate customers in the waiting room, (2) **barber** to coordinate activities of the barber, and (3) **mutex** for their general mutual exclusion. The **customer** semaphore keeps track of total number of waiting customers. When a customer arrives, she attempts to acquire the **mutex** first. Then she checks whether she can have a free chair (either in the waiting room or the barber's room). If there is no free chair, she departs the barbershop releasing the **mutex**. Otherwise *she* occupies a chair (statement in the figure is: nFreeChairs = nFreeChairs − 1). She then notifies her presence to the barber, releases the **mutex**, and waits for the barber. The barber algorithm is an infinite-loop type. In each iteration, she waits for some customer to come. When there are customers, she takes the **mutex**, gets up from her chair, and tells customers that she is ready to cut their hair. Finally she releases the **mutex**, and serves one customer.

What properties does the solution provide? It ensures the safety property that the barber cuts the hair of one customer at a time, and that there is no

[4] A solution is said to be symmetric if all processes are identical and may only refer to chopsticks by their names like chopstick(left) and chopstick(right), and if all shared variables have the smae initial values.

```
Data structures and initial values
    semaphore customer = {0, NULL};/* initially no customer */
    semaphore barber = {0, NULL};          /* initially barber is not ready */
    semaphore mutex = {1, NULL};  /* controlling access to nFreeChairs variable */
    int nFreeChairs = init-value;  /* greater than 1 */ /* for controlled
                                      accesses to the nFreeChairs variable */

Customer Algorithm
{
    down(&mutex); /* begin of mutual exclusion */
    if (nFreeChairs > 0) {
      nFreeChairs = nFreeChairs – 1; /* grab a free chair */
      up(&customer); /* tell barber that a new customer is in the shop */
      up(&mutex); /* end of mutual exclusion */
      down(&barber); /* wait for her turn to cut hair */
    } else {
      up(&mutex); /* end of mutual exclusion; she leaves the barber shop dejected */
    }
}
Barber Algorithm
repeat-forever {
    down(&customer); /* if there are customers, choose one; otherwise sleep */
    down(&mutex); /* mutual exclusion with customers to manipulate chairs */
    nFreeChairs = nFreeChairs + 1; /* frees up one chair */
    up(&barber); /* tell customers that she is ready to cut hair */
    up(&mutex); /* end of mutual exclusion */
    ⟨Cut hair of the chosen customer;⟩
}
```

Figure 7.21: *A solution to the sleeping-barber problem using semaphores.*

deadlock in the system. It also satisfies the liveness property. It, however, fails to ensure the starvation freedom property.

7.5.7 Spinlock

In the semaphore implementation (see Fig. 7.17 on page 173), adding (or removing) processes to (or from) the semaphore waiting queue is a costly operation compared to incrementing and decrementing the counter variable. In addition, when there is no token available in the semaphore, a down operation execution on the semaphore blocks the executing process, and calls upon the CPU manager to release the CPU from the process, thereby causing a context switch. We know from Chapter 5 that context switches incur additional overhead. If the execution time of the critical section is short compared to the summation of enqueue, dequeue, and context switch times, semaphore would be an inefficient tool in multiprocessor systems for "processor" synchronization. We use a different flavour of semaphore called spinlock in multiprocessor systems to synchronize the CPU activities.

A *spinlock* is like a semaphore, but there is no waiting queue associated with the spinlock. The spinlock is a special kind of shared non-negative integer variable that is solely manipulated by atomic **up** and **down** operations. A typical implementation of spinlock operations is shown in Fig. 7.22.

```
/*Get a token from spinlock lock*/
void down(spinlock* lock)
{
    while (*lock == 0) do;
    *lock = *lock - 1;
}
/*Return a token to spinlock lock*/
void up(spinlock* lock)
{
    *lock = *lock + 1;
}
```

Figure 7.22: *A typical spinlock implementation.*

Reading and incrementing the lock value in an **up** operation execution (and reading, testing, and decrementing the lock value in a **down** operation execution) must be done indivisibly. When a CPU, in a **down** operation execution, finds the spinlock value is zero, it repeatedly reads and checks the spinlock instead of blocking the running process. That is, it "spins" on the lock until the lock value becomes greater than zero, thereby causing no context switch. Many multiprocessor architectures provide support for spinlock via special machine instructions. Note that spinlocks cannot be used in uniprocessor systems. Spinlocks are primarily used to synchronize CPU activities. For example, it was mentioned in Chapter 5 that in multiprocessor systems many CPUs can simultaneously access the ready queue. Such accesses to the ready queue can be synchronized using spinlocks.

≫ Solaris supports a locking mechanism that is in between spinlock and semaphore. It is called *adaptive mutex*, and is very effective in multiprocessor systems for thread synchronization. The adaptive mutex is like a spinlock. A thread trying to acquire an adaptive mutex spins on the mutex if the thread holding the mutex is in the running state. Otherwise, the requesting thread blocks itself and releases the CPU.

7.5.8 Critical Region

The semaphore and spinlock are very elementary synchronization tools. They may be used to solve almost all synchronization problems. However, if used incorrectly, the system robustness may be compromised. For example, if in a program fragment one mistakenly performs a **down** operation instead of a required **up** operation, it may lead to permanent blocking (or self-locking) of the executing processes. Another example is that a program fragment misses a **down** or **up** operation. Such mistakes are natural and common in software development. However, these kinds of subtle development errors are sometimes difficult to detect, because in many situations time-dependent concurrency errors are irreproducible or extremely difficult to reproduce. To eliminate these kinds of silly, unintentional mistakes in programs, the concept of critical region was invented. Critical region is a programming language construct, and is used to eliminate simple silly programming errors that one may make using semaphores. Note, however, that the critical region construct does not eliminate programming errors of all kinds. It only helps the developers of synchronization solutions in eliminating silly errors and reducing the number of errors.

A *critical region* is defined by a higher-level language statement. The general construct is "region v do S", where v is called a *region variable*. A region variable is a shared data structure that is used only in statements like S under the control of the region variable. Statement S is actually the critical section, and v guards the executions of S. When a process executes S, it has

the exclusive right to the use of v. All statements under the same region variable are guaranteed execution in mutual exclusion. A compiler can check if a region variable is ever used outside critical regions, and signal a compilation error if it is found to be so. It is the duty of the compiler to ensure that all critical regions that use the same region variable v are executed mutually exclusively. The compiler translates the region statements into full proof implementation of critical regions (see Section 7.5.11 on page 185). It is up to that implementation what fairness condition it will follow to schedule waiting processes to execute their critical regions. Critical regions that refer to different region variables can, however, be executed concurrently.

For a solution to the mutual exclusion problem, we put all similar critical sections under the care of a single region variable. The mutual exclusion is guaranteed by the compiler's translation of the programs.

7.5.9 Conditional Critical Region

The synchronization tools discussed so far in the previous subsections help processes to synchronize their activities for special conditions. For example, a binary semaphore acts as a switch. These synchronization tools are very general, and are used to solve almost all synchronization problems. However, using them to synchronize processes for arbitrary conditions becomes a challenging task. For example, suppose we want to make some processes wait until the value of a shared integer variable becomes greater than a specific value. Constructing solutions for these kinds of conditional synchronization problems using the previously mentioned tools is an uphill task, if not wholly impossible. To alleviate the design burden for solutions to arbitrary conditional synchronization problems, the concept of conditional critical region was invented. This new tool enables design of synchronization solutions for arbitrary conditions with lesser effort.

Conditional critical region is another programming language construct and is a generalization of the unconditional critical region of Section 7.5.8. The general construct is "region v do $S1$ ..., **await**(B), ..., $S2$ done", where v is called a *condition variable* that is a shared data structure, and B is an arbitrary boolean expression that can refer to values of the variable v, and **await** is a new synchronization primitive. The entire sequence of statements "$S1$, ..., **await**(B), ..., $S2$" is a critical region. Like unconditional critical region, all statements under the care of a common condition variable are executed mutually exclusively.

Without condition variables, a process would need to have continual polling (possibly in a critical section) to check if the desired condition B is met. This can be very resource consuming since the process would be continuously busy in this activity, and quite unnecessarily occupy the CPU. A condition variable is a means to achieve the same goal without polling. While a region variable implements synchronization by controlling the accesses of processes to a region variable, a condition variable allows processes to synchronize based on the actual values of the condition variable. Like region variables, a condition variable v also ensures mutually exclusive execution of the regions guarded

the variable. However, if condition B evaluates to false (inside the critical region), the process temporarily stops executing the critical region and waits in some event queue after releasing the region for other processes. The process would wait in the event queue until the expression is satisfiable. When another process makes changes to the value of v, the condition B may become true. Therefore, when a process exits its critical region, it makes all waiting processes reevaluate their await conditions. (It is a non-positive wakeup.) If the boolean expression B is not satisfied at that point of reevaluation, the process temporarily suspends its execution of the statement, and resumes its execution at a later time.

A Solution to the Producer–Consumer Problem

We now solve the message buffer problem introduced in Section 6.4.3 on page 143. The solution presented there may not work if **put** and **get** operations are executed concurrently. Here we present a solution using the conditional critical region construct. The solution is presented in Fig. 7.23. It is easier to comprehend the solution and prove its correctness. Buffer is a shared data and it acts as a condition variable. The structure of **put** and **get** routines indicate that they are effectively critical regions, and the buffer variable ensures their mutual exclusion. The await statements in the two routines help processes to synchronize until certain conditions are met. For example, in the put routine, if all the message slots are occupied, the sender waits.

A Solution to the Readers–Writers Problem

A simple solution to the readers-writers problem using the conditional critical region construct is presented in Fig. 7.24. The solution uses one region variable *writer* and one condition variable v. Readers, under the condition variable v,

```
Constant
    int n = buffer-size;

Data structures and initial values
    struct T {
        message buff[n];
        int n_message = 0;
        int in = 0;
        int out = 0;
    } buffer;

void put(message* m)
{

region buffer do {
await(n_message < n);
buff[in] = *m;
in = (in + 1)% n;
n_message = n_message + 1;
}
}
```

```
void get(message* m)
{
{
region buffer do
await (n_message > 0);
*m = buff[out];
out = (out + 1)% n
n_message = n_message – 1;
}
}
```

Figure 7.23: *Implementation of bounded buffer using a condition variable.*

```
Data structures and initial values
    struct T {
        int waitingWriter;
        int readingReader;
    } v = {0, 0};

    shared boolean writer;

Reader:
    region v do await(v.waitingWriter == 0);
                v.readingReader = v.readingReader + 1;

    ⟨Read⟩;
    region v do v.readingReader = v.readingReader – 1;

Writer:
    region v do v.waitingWriter = v.waitingWriter + 1;
                await(v.readingReader == 0);
                done;
    region writer do ⟨Write⟩; /* mutual exclusion of writers */
    region v do v.waitingWriter = v.waitingWriter – 1;
```

Figure 7.24: *A solution to the readers–writers problem using condition and region variables.*

first wait until all writers leave the critical section and then increment the active reader counter *readingReader;* then, they read. Finally, they decrement the counter again under *v*. Writers, under *v*, first increment the active writer counter *waitingWriter* and then wait until the readers have left the critical section. Then, they write under the region variable *writer* to exclude one another. Finally, they decrement the counter *waitingWriter* under *v*.

7.5.10 Monitor

Monitor is another programming language construct for process synchronization. A monitor is an abstract data structure that consists of a set of variables whose values represent states of the monitor. It also has a set of functions that are executed to operate on the monitor to change its state. Resembling C++ or Java programming language, the variables are private to the monitor, and some monitor functions are made public. One cannot access private variables from outside the monitor. The public functions are the sole user interface or entry points to the monitor, and they may take formal parameters. The monitor functions can access private variables and parameters. Monitor is very similar to traditional (C++ or Java) object paradigm. However, process synchronization is a little more involved than just traditional objects. The condition is that at the most one process can be executing at any time inside a monitor. That is, processes access a monitor mutually exclusively. Monitor locks itself when a process begin an execution of a procedure and unlocks when the process completes its execution or blocks for some condition. If another process tries to invoke a procedure of a locked monitor, the process is suspended until the monitor is unlocked. The process is said to

be blocked at the entry to the monitor. The processes do not need to bother about how the synchronization is performed. Normally the compiler supplies these synchronization codes.

A monitor, in addition to ordinary variables, may have synchronization-specific variables called *condition variables*. (One should not confuse this term with conditional critical region variables.) A condition variable supports two interface operations, namely *wait* and *signal*. Inside a monitor, a process suspends its execution by executing the wait operation on a condition variable, and resumes it when another process executes a signal operation on the same condition variable. (Though blocked and not executing the suspended process is still considered to be active inside the monitor.) The signal operation resumes only one suspended process; if there is no suspended process, then the signal operation becomes a no-op. The signal operation does not activate any processes that are waiting at the entry to the monitor. However, only one of the two processes can continue its execution in the monitor while the other has to wait until the former completes its execution or suspend itself by executing another wait operation. In the original implementation of monitor by Hoare, a signalling process always waits until the resumed process leaves the monitor or executes another wait operation. In a variation by Hansen, the signalling process must leave the monitor by executing a return statement in order for the signalled process to continue.

≫ All processes inside a monitor are "active". But only one of them is executing the monitor. The rest are blocked, i.e., waiting on some condition variables.

We use the following syntax to declare a monitor.

```
Monitor name-of-the-monitor {
Variables (global and condition) declarations;
Procedure proc-1(parameters);
Procedure proc-2(parameters);
:
Procedure proc-n(parameters);
{ Initialization code, aka, constructor }
};
```

≫ A monitor should allow at the most one process to execute within it. Assume that a process A signals on a condition variable on which another process B is waiting; the question is, among A and B, which process will execute a monitor procedure and which process will exit the monitor or wait. There are three popular disciplines: (1) A exits the monitor (signal and exit); (2) A waits until B leaves the monitor (signal and wait); and (3) B waits until A leaves the monitor (signal and continue). Java implements the signal and continues discipline.

To solve the mutual exclusion problem using a monitor, put the critical section inside the monitor something like the following: The process can invoke mutex.Critical_Section() to execute the critical section.

```
Monitor mutex {
Critical_Section(void);
Procedures(parameters);
};
```

For a specimen example, we solve the dining philosophers problem using a monitor. (It was solved in Section "A Solution to the Dining Philosophers Problem" on page 177 using semaphores.) Assume that there are $n > 1$ philosophers. The chopsticks must be accessed in a mutually exclusive way. The routines that access the chopsticks are put inside the monitor for synchronized access. The philosopher waits on a condition variable when the chopstick

requested is in use. For brevity, in this section we use left and right to refer the ids of the left chopstick and right chopstick, respectively, of each philosopher. This simple solution is given below.

```
repeat-forever {
  think();
  DP.get-chopstick(left);
  DP.get-chopstick(right);
  eat();
  DP.put-chopsticks();
}
Monitor DP {
  const int n = init-number; /* greater than 1 */
  boolean cs[n] = 0; /* representing chop sticks */
  condition csc[n];
  get-chopstick(i)
  {
  if(cs[i] ≠ 0) wait(csc[i]);
  cs[i] = 1;
  }
  put-chopsticks()
  {
  cs[left] = 0;
  cs[right] = 0;
  signal(csc[left]);
  signal(csc[right]);
  }
};
```

This solution assures the safety, but not the liveness property. If every philosopher picks the left chopstick then they form a circular wait and then they are deadlocked. To avoid circular wait, a simpler solution would be just to prohibit all the n philosophers to request chopsticks simultaneously. That is, a philosopher can request a chopstick only if the number of other philosophers requesting chopsticks is less than n. The solution incorporating this idea is given below.

```
repeat-forever {
   think();
   DP.safe();
   DP.get-chopstick(left);
   DP.get-chopstick(right);
   eat();
   DP.put-chopsticks();
}
```

```
Monitor DP {
const int n = init-number; /* greater than 1 */
int count = 0;
boolean cs[n] = 0; /* representing chop sticks */
condition csc[n];
condition notsafe;

safe()
{
  count++;
  if (count == n) wait(notsafe);
}
get-chopstick(i)
{
  if(cs[i] ≠ 0) wait(csc[i]);
  cs[i]= 1;
}
put-chopsticks ()
{
  count–;
  if(count == n–1) signal(notsafe);
  cs[left] = 0;
  cs[right] = 0;
  signal(csc[left]);
  signal(css[right]);
}
};
```

7.5.11 Comparison of Synchronization Primitives

The synchronization tools introduced above are equivalent in the sense that any synchronization problem can be solved using any of the tools. However, for a given problem some tools may lead to complex, difficult to understand solutions, or inefficient solutions. A particular tool is ideal to solve particular kinds of synchronization problems, but leads to complex solutions for other kinds of synchronization problems. A tool is said to support a particular type of synchronization if it provides primitives that make it convenient for constructing solutions to that class of synchronization problems. Each tool solves certain kinds of problems elegantly and efficiently.

For example, we can implement critical region construct using semaphores. For each region variable *v*, which is used in statements such as "region *v* do S", a compiler allocates a new binary semaphore variable, say *v_mutex* whose counter is initialized to 1. The region statement is

transformed by the compiler as "**down**(*v_mutex*); *S*; **up**(*v_mutex*)". We can also implement conditional critical region- and monitor constructs using semaphores and other control variables. See the Literature section of this chapter for appropriate articles. These constructions imply that we can implement language-based synchronization tools using semaphores.

We have assumed that semaphore operations (**up** and **down**) are indivisible, and the indivisibility can be achieved through machine instruction-based constructs from Sections 7.5.3 and 7.5.4. We have assumed that read and write operations on memory cells are executed atomically by the memory hardware. Thus, reading/writing of memory cells is a fundamental synchronization problem solved by the memory hardware.

The irony of the software solutions (presented in previous subsections) to various synchronization problems is that we transfer problems from one level to another: from macro level to micro, from coarse granules to fine, from complex to simpler problems. True synchronization, by atomic executions of read and write operations on memory cells, is provided by the memory hardware. Software solutions elevate the hardware capability to higher levels so that it becomes convenient for users to use synchronization tools. What happens when the memory hardware does not guarantee atomicity of read and write operation executions? Can we develop good synchronization primitives in such systems? These are the fundamental questions in synchronization theory. We will address this question in Section 7.7. Incidentally, the Lamport solution (see Section "The Lamport Solution" on page 167) to the general mutual exclusion problem does not assume the atomicity of read and write operations on primitive shared variables. His solution, however, requires shared variables of unbounded size.

7.6 Deadlock

A computer system has many resources that processes use. One of the major advantages provided by operating systems is the ability to share (i.e., time-multiplex) resources among processes to improve resource utilization and system performance. However, unless done carefully, such sharing may lead to unwanted situations where processes are unable to make progress in their computations. Operating system designers and developers should be well aware of the problems due to resource sharing, and also well acquainted with solutions to those problems.

An example of identical ource units is memory ations.

In this section, we use the term resource in a very generic sense to mean anything that may conditionally block processes. A resource may be a "reusable resource" such as a printer, or a "consumable resource" such as a message. A resource may have many identical units; the units are considered equivalent, and any unit will satisfy a request on the resource. There is a priori known fixed number of units of each reusable resource. Processes compete for reusable resources. A unit of a reusable resource can be assigned at the most to one process at a time. The total number of units of a consumable resource is

not fixed. Producers produce consumable resources. Once a consumable resource unit is allocated to a consumer, the unit is considered destroyed.

A process acquires a resource, uses it, and finally releases it. At the time of acquisition, the process might be blocked if the resource is not immediately available. A process requests a resource, and waits until the resource is granted to it. When granted, the process comes out of the waiting state and uses the resource. Finally, when the resource is no more required, the process releases the resource (reusable case) or destroys it (consumable case). A resource, once allocated to a process, cannot be preempted from the process by force. A process may request as many resources as it needs to accomplish its task, but, not for more reusable resources than the system has.

As mentioned above when a process requests a resource, the process is blocked until a unit of the resource is allocated to it. A blocked process cannot come out of the waiting state on its own and make progress in its execution (i.e., change its state) until it is unblocked by someone else. In such an environment, processes may enter into deadlocks or deadly embrace situations. In any computer system where processes share exclusive resources or exchange messages, we have to handle deadlock related issues.

A *deadlock* is a situation in which there is a group of agents such that each one in the group is waiting for some (resource allocation and release) events to occur, but these events never happen because the events are supposed to occur at other agents in the group. In our case, the agents are processes. The waiting processes never change their states as each has requested a resource that is held by another waiting process. The processes are in a deadly embracing situation, and they cannot come out of the situation on their own. They are blocked forever, one on another, unless an external agent takes some action to unblock them.

In the simplest case a deadlock occurs when two processes, say P_1 and P_2, try to use two exclusive resources r_1 and r_2. Suppose P_1 has r_1, and P_2 has r_2. After a while, P_1 requests for r_2, and P_2 for r_1. See Fig. 7.25; it is called a *resource-allocation graph*, and represents resource holding and unsatisfied requests. The solid arrows represent holding information, and the broken arrows requesting information. Now, P_1 is waiting for P_2 to release r_2, and P_2 is waiting for P_1 to release r_1. Thereby, they form a circular waiting list: P_1 is waiting for P_2 (to release r_2) that in turn is waiting for P_1 (to release r_1). Consequently, none of the two processes can make progress in their respective computations, leading to a deadlock situation.

≫ That a process is blocked does not necessarily imply that it is involved in a deadlock. Also, due to bad system designs some process(es) may be *blocked forever* without involving in deadlocks. The processes are frozen individually and not in an embracing situation. For example, we have a semaphore initialized to 0. A process, however, instead of performing an **up** operation on the semaphore does a **down** and perpetually blocks or freezes or self-locks itself on the semaphore. In this book, we do not study these issues arising out of wrong code development.

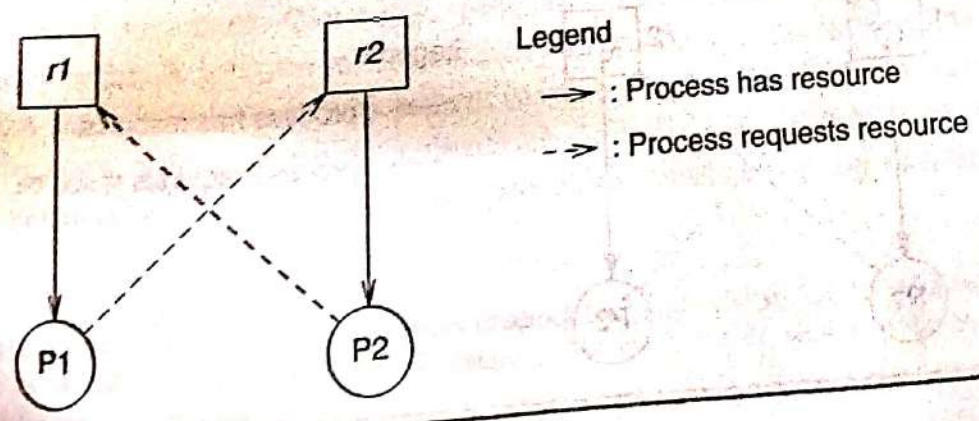

Figure 7.25: *A typical deadlock scenario.*

As mentioned above, a process waiting for a resource does not necessarily mean that it is in a deadlock. There are criteria to declare the existence of deadlocks in a system of processes. The following are the four necessary conditions for deadlock formation.

- **Mutual exclusion:** There are resources that cannot be used by more than one process concurrently. If one process holds an exclusive resource, another requesting process has to wait until the first process releases the resource.
- **Non-preemptive assignment:** Resources allocated to processes cannot be preempted. Only the process that has acquired the resource can release it.
- **Partial allocation of resources:** Resource allocation is incremental. Each process holds some resources and requests additional resources as its computation evolves.
- **Circular waiting:** There exists a circular chain of waiting processes. Each process in the chain is waiting for a resource held by the next process in the chain.

Note that the above four are necessary conditions, but may not be sufficient for some systems. (The first two signify the usage of exclusive resources and the semantics of resource allocation, respectively.) If there is no circular chain of processes, then there is definitely no deadlock. If there is a circular chain, there is however a *possibility* of a deadlock. If a circular chain involves those resources that have one unit each, then there is definitely a deadlock. Otherwise, there may or may not be a deadlock, and we need to derive a correct answer by using additional information. For example, Fig. 7.26 depicts a scenario with a circular chain of processes not involving a deadlock. In the figure, small circles in boxes represent identical resource units. For example, resource r_1 has two identical units that are allocated to processes P_1 and P_3. When P_3 releases the r_1 unit, the unit can be allocated to P_2 to break the cycle.

≫ The presence of a knot is a sufficient condition for a deadlock. A knot is a non-empty set K of nodes such that reachable set of each node in K is the set K ifself.

Deadlock is undesirable. In fact, it is a serious problem as it leads to underutilization of resources and wastage of processing time. Processes

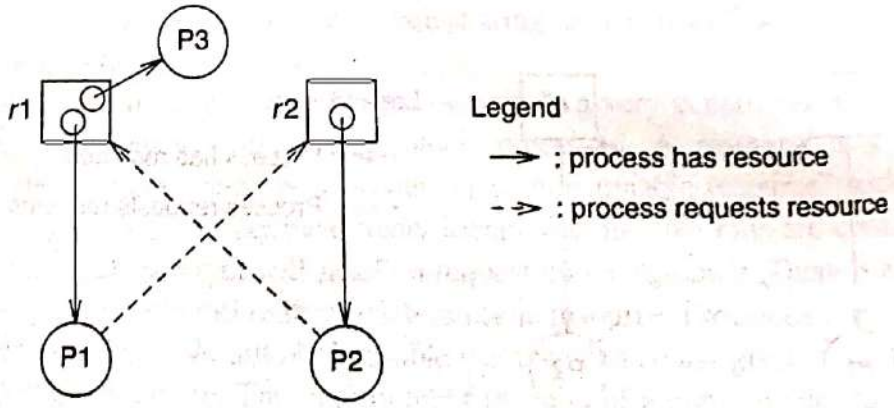

Figure 7.26: *A typical circular chain without a deadlock.*

involved in a deadlock cannot make any progress in their computations but unnecessarily occupy valuable resources. We need to solve this problem.

Deadlock can be dealt with at many levels. We can prevent deadlocks ahead of time (usually by rigid static rules), make careful moves (or aka transitions) to avoid deadlocks by looking at the current state (usually by dynamic or adaptive rules), detect and resolve once deadlocks occurred (by breaking one of the conditions), or simply ignore (popularly called Ostrich algorithm—stick your head in the sand and pretend there is no problem at all). Many solutions to the deadlock problem have been reported in literature. The solutions are classified into three broad categories: (1) prevention, (2) avoidance, and (3) detection and recovery.[5] Solutions in the former two categories ensure that at least one of the four necessary conditions (noted above) never holds, directly or indirectly. Solutions in the last category do allow deadlock formation; they run deadlock detection tests at regular intervals. If a deadlock is found, some processes are forcefully terminated to break the circular list of waiting processes. We discuss a typical solution in each of these three categories in the following three subsections, where we only deal with reusable resources.

≫ Many current operating systems, although consider deadlock issues, appear to restrict concurrency of kernel executions. For example, Linux 2.4 and its earlier releases use a single global kernel lock to synchronize concurrent kernel paths on multiprocessor systems. Before reduced level of kernel concurrency becomes a severe bottleneck, deadlock-related features will have to be incorporated in operating systems. The operating systems we are aware of stipulate manual deadlock prevention.

7.6.1 Deadlock Prevention

A deadlock prevention scheme specifies a set of rules that processes need to follow in acquiring resources. Their strict observance guarantees that no deadlock ever occurs in the system. Most prevention schemes violate at least one necessary condition for formation of deadlocks. In the following paragraph, we discuss a very simple scheme to prevent circular waiting by processes.

All resources are totally ordered. That is, each resource has an ordinal position in the total order. Processes always request resources in this order. If a process holds a resource at ordinal position k, it cannot request resources at ordinal positions k', when $k' \leq k$. If a process P waits for a resource r at ordinal position k and r is held by another process Q, then Q will not further request any resource at ordinal positions less than or equal to k. Hence, Q will never wait (directly or indirectly) for P. Consequently, there will be no circular list of waiting processes, and thereby, no deadlock forms in the system.

7.6.2 Deadlock Avoidance

Deadlock avoidance is a kind of deadlock prevention without any static ordering of resources. Normally, processes can request resources in an arbitrary order. A solution in this category keeps the system always in a safe state. A *safe* state is one in which there exists a way to complete executions of all processes without forming a deadlock. Initially, the system is in a safe state because we can always execute processes sequentially. By definition, a

[5] Levine lately has strenthened the four necessary deadlock conditions, and proposed that prevention and avoidance should be classified into a single category. See the Literature section at the end of this chapter for more information.

sequential execution does not throw the system into a deadlock because there is no competition for resources. A safe state is normally identified by a set of safety conditions. When a process makes a new request on a resource, an avoidance algorithm re-evaluates safety conditions to check whether the system remains in a safe state after the request is granted. If the state becomes unsafe, the request cannot be granted and hence the process needs to wait.

Unlike deadlock prevention schemes, a deadlock avoidance scheme does not force processes to acquire resources in a particular order, but it does require advance knowledge about what resources processes need to accomplish their tasks. Without such advance knowledge, it may not be possible to predict possible future states of the system. With the advance knowledge and that of the current allocation status of all resources, the scheme can always determine whether the current state is safe. Deadlock avoidance solutions differ from one another for information they need about the resource requirements of each process.

Let us study a simple example here. Let R be a resource with $2c + 1$, $c > 0$, identical units, and P and Q be two processes. They may need 2c and $c +$ 1 units, respectively, to complete their executions. Suppose P already has c units, and Q one unit. They may each request c more units in future. The system is still in a safe state as we can complete the execution of both processes, at least sequentially in arbitrary order from now onwards. If we allocate one more unit to both P and Q, then P will have $c + 1$ and Q two units, and $c - 2$ units remain free. Now, the processes each may need $c - 1$ units to complete their executions, which is impossible to allocate as there are fewer free units. Therefore, the system is not in a safe state: although the processes are not in a deadlock now, they could however be in a deadlock in future. We need to avoid such deadlock formations in the future by carefully allocating resource units to processes. One simple solution is to execute processes mutually exclusively at different times. This, however, would lead to poor utilization of resources and an increase in process response time. A solution that would allow as much concurrency as possible without any possibility of forming deadlocks in the system is the need of the hour.

Initially, the system is in a safe state: no resource is allocated to any process. A resource allocation algorithm must keep the system in a safe state and, at the same time, ensure as much resource utilization as possible. What we need is an algorithm that can determine whether a given system state is safe or not. When a new request comes, we may first pretend to grant the request, and then evaluate the algorithm to determine whether the new state is safe. The request is granted only if it is safe in the new state as well. Otherwise, we shelve the request for future consideration. In the rest of this subsection, we present a deadlock avoidance algorithm that is popularly known as the *banker* algorithm.

Banker Algorithm

Suppose that there are m, $m > 0$, resources $r_1, \ldots, r_m$ that are used by n, $n > 1$, processes $P_1 \ldots, P_n$. Resource r_i has c_i, $c_i \geq 1$, identical units. These units are

equivalent in the sense that when a process requests for a resource r_i, any of the c_i units will satisfy the request.

Each process, at the start of its execution, declares the maximum requirements of all resources it needs to complete its execution. That is, the banker algorithm requires advance knowledge of maximum resource requirements for all processes. A process may not need all its required resources right at the start of its execution. It requests resources one by one in an unknown pattern. If a process's maximum need is satisfied, it is guaranteed that the process will release the resources in a finite time when it has finished using those allocated to it.

The *state* of the system is determined by the state of each resource unit (whether free or not), and the maximum resource requirements of all processes and the resource units they presently hold. A state is called *safe* if the system can enable all its processes complete their executions in a finite time without throwing any of them into deadlocks. In a safe state, it is always possible to satisfy new resource requests from each process in a way that all processes can complete their executions. Otherwise, the state is deemed *unsafe*. An unsafe state does not necessarily mean that there is a deadlock in the system. It only means that there is a possibility of deadlock forming in the future. Some unsafe states do have deadlocks. The banker algorithm takes a pessimistic position, and keeps the system always in a safe state.

The banker algorithm is presented in Fig. 7.27. The constants *units* identify the number of units of all resources. The variables *maxNeed* identify the maximum requirements for all processes, and the variables *allocated* identify the current resource allocation. The assertion that for all processes i, and all resources j, *unit[j] >= maxNeed[i][j] >= allocated[i][j]* is an invariant to the algorithm. The invariant says that a process cannot ask for more resource units than the system has, and the system never allocates more resources to a process than it needs. The algorithm is quite simple. It repeatedly looks for a process that it can complete with the then available free resources. If the algorithm ends with all processes completed, then the current state is safe.

Further, the system finds a dynamic order of processes such that if they are executed in that order deadlocks will not occur. However, the banker algorithm is quite expensive: each execution may take $O(n^2m)$ time, where n is the number of processes and m the number of resources.

Examples of The Banker Algorithm

Consider the following resource allocation state involving five processes P_0, P_1, P_2, P_3, and P_4, and five resources R_0, R_1, R_2, R_3, and R_4. *Max[i, j]* specifies the maximum number of instances that process P_i may request for resource R_j. *Alloc[i, j]* gives the number of instances of resource R_j currently allocated to process P_i. *Avail[j]* specifies the number of instances of resource R_j are currently available. We need to determine whether the system is in a safe state. Initially, Avail = [7, 7, 7, 7, 10].

```
Constants
   n = number of processes;
   m = number of resources;

Data structures and initial values
   const int unit[m] =          {c1,···,cm};     /* number of copies of m resources */
   const int maxNeed[n][m];                      /* maximum need of processes; initialized to proper values */
   int allocated[n][m];                          /* present allocation; initially all zero */

boolean safe(n,m,unit,maxNeed,allocated) /* check if state is safe */
{
   int available[m];
   boolean completed[n] = {false, false, ···, false};
   for(int j = 0; j < m; j = j + 1) available[j] = unit[j]−Σ_{i=0}^{n−1} allocated[i][j];
   repeat
      boolean change = false;
      for (int i = 0; i < n; i = i + 1) {
         if (! completed[i]) { /* Pi is not complete */
            if (completion_possible(maxNeed[i], allocated[i], available)) {
               for (int j = 0; j < m; j = j + 1) {
                  available[j] = available[j] + allocated[i][j];
               }
               completed[i] = true;
               change = true;
            }
         }
      }
   until (! change);
   return    (∀j[unit[j] == available[j]]);
}

boolean completion_possible(int maxNeed[m], int allocated[m], int available[m])
{
   for (int j = 0; j < m; j = j + 1) {
      if (maxNeed[j] − allocated[j] > available[j]) {
         return false; /* process needs more resources than currently available */
      }
   }
   return true;
}
```

Figure 7.27: *The banker algorithm.*

Max = [[4,2,3,1,1], [1,2,3,4,5], [3,2,3,1,1], [5,4,3,2,1], [2,0,0,2,2]]
Alloc = [[1,1,1,1,1], [1,0,1,1,1], [0,1,2,0,1], [3,0,2,1,1], [1,0,0,2,1]]
Avail = [1,5,1,2,5]

This is a safe state. We can schedule processes in this sequence to complete their executions: P_4, P_3, P_0, P_1, and P_2. After P_4 is complete Avail = ([1, 5, 1, 2, 5]+[1, 0, 0, 2, 1]) or [2, 5, 1, 4, 6]; after P_3 is complete Avail = ([2, 5, 1, 4, 6]+[3, 0, 2, 1, 1]) or [5, 5, 3, 5, 7]; after P_0 is complete Avail = ([5, 5, 3, 5, 7]+[1, 1, 1, 1, 1]) or [6, 6, 4, 6, 8]; after P_1 is complete Avail = ([6, 6, 4, 6, 8]+[1, 0, 1, 1, 1]) or [7, 6, 5, 7, 9]; and after P_2 is complete Avail = ([7, 6, 5, 7, 9]+[0, 1, 2, 0, 1]) or [7, 7, 7, 7, 10].

Now, suppose process P_0 requests a unit of R_0. If the system grants the request, then we have the following state of the system.

Max = [[4,2,3,1,1], [1,2,3,4,5], [3,2,3,1,1], [5,4,3,2,1], [2,0,0,2,2]]
Alloc = [[2,1,1,1,1], [1,0,1,1,1], [0,1,2,0,1], [3,0,2,1,1], [1,0,0,2,1]]
Avail = [0,5,1,2,5]

This is an unsafe state because requests on R_0 from processes P_0, P_2, P_3, and P_4 and on R_2 from P_1 cannot be satisfied.

7.6.3 Detection and Recovery

The prevention technique (presented in Section 7.6.1) forces one to organize all resources in a total order, and also compels processes to acquire them in that order. It may not be convenient for developers of applications and operating systems to do so. It also leads to poor utilization of resources, as processes are sometimes compelled to acquire resources that they may use only in the remote future or not at all.

The banker algorithm (given in Section "Banker Algorithm") though does not enforce processes acquiring resources in a predetermined order, but it adopts a pessimistic approach to avoid deadlocks. Each process has to declare its maximum requirements of resources when it starts even though it may not acquire them in this incarnation of program execution. The algorithm leads to poorer utilization of resources because the system may keep some resources idle on the assumption that some processes may need them soon. In addition, the banker algorithm is too costly to execute on each request for a resource.

In some practical systems, it may be worth allowing formation of deadlocks, and detecting and resolving deadlocks rather than preventing or avoiding their occurrences. This way of treating deadlocks is suitable in environments where deadlocks are rare and/or recovery is not very expensive. The system may not perform a "safeness" check right when it allocates resources to processes. Consequently, there is a possibility of deadlock formation in the system. The system runs some deadlock detection algorithm, at regular intervals. If a deadlock has indeed occurred, the system takes some corrective actions by forcefully terminating some processes and making their acquired resources available to other processes. The victim processes can be re-started later.

Here, we describe a simple deadlock detection scheme that works for systems having one unit of each resource. (Readers may consult the Literature section of this chapter to know about general solutions.) The system keeps track of which processes use which resources, and which processes wait for which resources. The scheme builds resource-allocation graphs, like the one shown in Fig. 7.25 on page 187. Given a resource-allocation graph, we merge resource nodes with respective processes that have the resources, and we obtain a new graph called *wait-for graph*, (see Fig. 7.28). The broken arcs in the wait-for graph indicate which processes are waiting on which processes. If P_i - - > P_j is an arc in the wait-for graph, then P_i is waiting for a resource that is held by P_j. A deadlock exists if and only if there is a cycle in the wait-for graph. (In Fig. 7.28, the two processes are deadlocked.) We can execute known graph algorithms to find cycles in wait-for graphs. Once a deadlock is found, the system forcefully terminates some

≫ A word of caution. Most real systems do not implement any sophisticated deadlock handling mechanisms. They go the ostrich way: pretend that deadlocks will not be frequent. In case system responsiveness becomes intolerable, reboot the system!

≫ The question is when or how frequently do we run the detection algorithm. A better approach is to associate a timeout with each pending request. On timeout, run the detection algorithm.

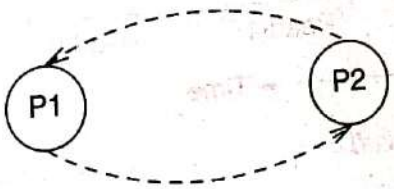

Figure 7.28: *The wait-for graph of Fig. 7.25.*

≫ A livelock is similar to starvation. The state of a process involved in a livelock does change, but it is effectively not progressing due to frequent aborts.

process(es) to break the deadlock. Care is necessary in terminating processes, as repeated termination of the same processes may induce livelock in the system.

7.7 The Real Challenge

In Section 7.5, we discussed many solutions to typical synchronization problems using well known tools. How fundamental are these solutions? What assumptions do we make to construct these solutions? Are these assumptions realistic? The most important assumption we made there is that the memory hardware ensures atomic executions of read and write operations on individual memory cells. Conflicting operation executions on the same memory cell exclude each other in real time. Consequently, the read and write operations are the true mutual exclusion problems and which the memory hardware solves. Software solutions elevate the hardware capability to higher levels so that it becomes convenient to use synchronization tools. What happens when multiple processors are connected to the memory through different ports of the memory? The read and write operation executions may truly overlap in real time. What happens if the memory hardware does not ensure atomicity of read and write operation executions? Can we develop good synchronization primitives in such systems? This is the fundamental question in synchronization theory.

Let us study a concrete example to understand this question better. Let v be an integer shared variable with initial value 0. A process P is writing 1 in v. To execute this write operation, a memory port takes some finite amount of time to overwrite the old value of v by the new value. What is guaranteed is that if some other process Q reads v before (respectively, after) the write operation execution starts (respectively, completes) will read 0 (respectively, 1). What value does Q read during the value transition from 0 to 1? This is in general unknown and may depend on the true state of the memory storage medium. What values does Q get if it repeatedly reads v in the writing period? The values read by Q are generally unpredictable if read during the transition period. Figure 7.29 displays a typical scenario of read and write executions on the shared variable v. In reality, the memory hardware arbitration unit solves this synchronization problem. The arbitration unit allows at the most one operation at a time to be performed on a given memory cell. Thus, the fundamental problem of synchronization is solved by the memory hardware. Their solution based on mutual exclusion (or critical region) is implemented in the hardware arbitration unit.

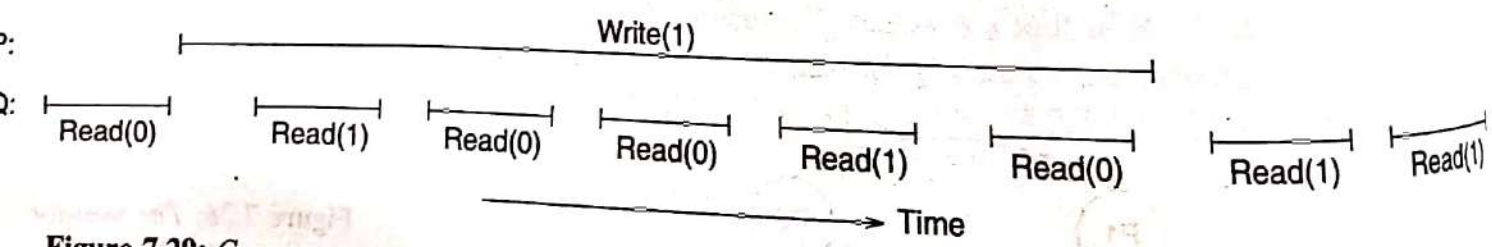

Figure 7.29: *Concurrent read and write on a memory cell.*

Incidentally, Lamport's Bakery algorithm (see Section "The Lamport Solution" on page 167) does not assume atomicity of read and write operations. Nevertheless, the solution needs unbounded size variables. Therefore, we have not really solved the fundamental synchronization problem in software if the memory arbitration unit does not effectively solve the problem at the level of the individual memory cell. The fundamental question is how we can ensure atomicity of (non-blocking, wait-free) read and write operation executions in the absence of the memory arbitration hardware. (Wait-freedom means operation executions of one process are not blocked by activities of other processes.)

Lamport defines the following three types of 1-writer multireader shared variables in which read and write operations can be executed in a *wait-free* manner.

- A *safe* variable is one in which a Read not overlapping any Write returns the most recently written value. A Read that overlaps a Write may return any value from the domain of the variable.
- A *regular* variable is a safe variable in which a Read that overlaps one or more Writes returns either the value of the most recent Write preceding the Read or of one of the overlapping Writes.
- An *atomic* variable is a regular variable in which the Reads and the Writes behave as if they occur in some total order that is an extension of their execution order.

Safe boolean variables come naturally in the form of flip-flops. Other variables are constructed from safe variables. Such constructions have been developed in recent years. However, the constructions are too complicated to present in a book such as this. Interested readers may consult articles referred to in the Literature section of this chapter.

Summary

Concurrency is the notion of doing multiple related activities simultaneously. Concurrency is ubiquitous in modern operating systems, and synchronization (of threads, processes, processors, kernel paths) is becoming increasingly important. This chapter introduces subtleties in process interactions. Race conditions arise in these systems because of the concurrent access of shared data by multiple agents. Not all outcomes of a race condition are correct. Unless race conditions are handled carefully, chaos ensues in these systems.

Synchronization is the coordination of concur-

the single most challenging aspect in the desi development of multiprocess operating sy especially for multiprocessor platforms. Th many kinds of synchronization problems. V to handle them tactfully to promote mo currency in the kernel without violating a rity constraints of the operating system.

This chapter introduces many type chronization problems such as the mut sion, the producer-consumer, the reade the dining philosophers, and the sleepi and solutions to these problems usi synchronization tools such as semaph