Chapter 9

Processes & Concurrency

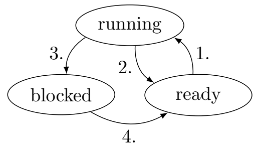
Each core in the computer can only execute a single program. How can a computer concurrently execute more than one program?

9.1 Process Scheduling

Definition 9.1 (Process)**.** *A program is a sequence of instructions. A* ***process*** *is a program that is being executed.*

Definition 9.2 (Process States)**.** *A process can be in one of three* ***states*** *at any time:*

- *it can be* ***running****: some processor core is executing the process;*
- *it can be* ***ready*** *(or* ***waiting****): the process could be executed, but is not;*
- *it can be* ***blocked****: the process is not currently being executed and some external event (e.g. an I/O operation) has to happen before the process becomes ready again; we say the process* ***blocks on*** *or* ***is blocked on*** *that event.*



1. Process is selected for execution
2. Process execution is suspended to run another process
3. Process blocks on some external event; its execution is suspended and cannot be resumed before the event happened
4. External event that blocked process happens

Figure 9.3: Transitions between the states of a process.

Remarks:

- The *scheduler* makes the decision which processes are running at any point in time, see Figure 9.3.

Definition 9.4 (Scheduler)**.** *The* ***scheduler*** *is the part of the OS that decides which processes are to be executed at which time. It can* ***suspend*** *(or* ***preempt****) the execution of a running process by saving the process' current state, and pick a ready process for execution whenever a core is free.*

Remarks:

- Process states are an abstraction that describe how the scheduler interacts with processes. Different OS implement the specifics differently, using different states and transitions.
- Numerous scheduling strategies exist. A simple one is *round robin*, where processes take equal-length turns.
- A more elaborate strategy is *priority scheduling* where a more important process takes priority over a less important one. The scheduler has to make tradeoffs that lead to these different priorities:
  - all processes should get to run at some point;
  - some processes are more vital than others, e.g. OS services;
  - some processes have deadlines, e.g. rendering a video frame;
  - the user can decide that a process should take priority;
- Priorities can be *fixed* or *dynamic* (recalculated regularly).
- Instead of keeping on running until the scheduler decides to suspend its execution, a process can also *yield* the core it is being executed on, which also sets the process to the ready state. On some embedded systems, the scheduler never preempts any process and waits until a process yields its core before scheduling the next process.
- Some processes, for example games, do multiple things at the same time. How can that be the case when they are a single process?

9.2 Threads

Definition 9.5 (Multithreaded Processes)**.** *A* ***multithreaded*** *process is a process that contains more than one* ***threads (of execution)****. All threads within a process share the address space managed by the process and the files opened by any thread in the process.*

Remarks:

- In modern OS, the scheduler schedules threads. This allows multiple threads within the same process to run concurrently.
- Since the scheduler schedules threads, the process states from Definition 9.2 would therefore more aptly be called thread states; we use "process states" since this is the term used in the literature.
- When we switch execution from one thread to another thread, we have to store the state information of the currently running thread and load the state information of the newly selected thread. This is called a *context switch*. A context switch between different processes is more expensive than a context switch between threads of the same process.
- The thread state information (that is saved and loaded during a thread context switch) is stored in a data structure called the *thread control block (TCB)* of the thread. A TCB stores the state of the thread (in the sense of Definition 9.2), the unique thread ID, the *instruction pointer* (also called the *program counter*; this is the number of the last instruction of the process' code that was executed), the stack of function calls this thread made, and other accounting information.
- The process state information (that is saved and loaded during a process context switch) is stored in an OS data structure called the *process control block (PCB)* of the process. A PCB stores the code of the program, the unique process ID, a list of open file handles, the address space available to the process, and other accounting information.
- If multiple processes/threads want to access shared resources while the scheduler can interleave their execution arbitrarily, how do we make sure everything works as intended?

9.3 Interprocess Communication

There are two general categories for sharing data between processes: message passing, and shared memory. In Definition 4.21, we already saw the concept of a *socket* as a message passing mechanism. There are other important mechanisms.

Definition 9.6 (Remote Procedure Call)**.** *A **remote procedure call (RPC)** is a message passing mechanism that allows a process on one machine (the client) to run a procedure on another machine (the server). The calling process packs the arguments for the procedure in a message, sends it to the server, and blocks until the call returns. The server unpacks the message, runs the procedure, and sends the result back to the client.*

Definition 9.7 (Pipe)**.** *A **pipe** is a unidirectional channel of data between processes that is handled within the OS. One process only writes to the pipe, and the other process only reads from the pipe. The OS buffers the written data until it is read.*

Remarks:

- In this chapter, we focus on shared memory. Memory can be shared between processes or between threads, and the solutions presented here can be used in both contexts. We will be speaking of processes throughout.

Definition 9.8 (Shared Memory)**.** *A **shared memory system** consists of asynchronous processes that access a common (shared) memory. Apart from this shared memory, processes can also have some local (private) memory.*

Remarks:

- Memory is usually shared among threads within the same process unless it is explicitly made thread-local.
- Memory is usually not shared between processes unless the programmer explicitly uses interprocess communication constructs.

Definition 9.9 (Atomicity)**.** *A process can **atomically** access a word in the shared memory through a set of predefined operations. An **atomic** modification appears instantaneously to the rest of the system.*

Remarks:

- Various shared memory systems exist. A main difference is how they allow processes to access the shared memory. All systems can atomically read or write a shared memory word W. Most systems do allow for advanced *atomic* read-modify-write (RMW) operations, for example:
  - test-and-set(W): atomically $\{t := W;\ W := 1;$ return $t;\}$
  - fetch-and-add(W, x): atomically $\{t := W;\ W := W + x;$ return $t;\}$
  - compare-and-swap(W, x, y): atomically {if $W = x$ then $\{W := y;$ return **true**;} else return **false**; endif;}

9.4 Mutual Exclusion

A classic problem in shared memory systems is mutual exclusion. We are given a number of processes which occasionally need to access the same resource. The resource may be a shared variable, or a more general object such as a data structure or a shared printer. The catch is that only one process at the time is allowed to access the resource. If that is not guaranteed, problems like the following can happen:

Example 9.10. *Consider two processes p_1 and p_2 that can concurrently update the balance of an account* `A` *in a banking system using Algorithm 9.11 on* `A`*:*

Now assume A has an initial balance of 0 before p_1 executes A.deposit(1) and p_2 executes A.deposit(2). If the execution of p_1 and p_2 can be interleaved

Algorithm 9.11 Deposit Money to Account

```
Account internals: balance
Input: deposit amount
1: balance = balance + deposit;
```

arbitrarily due to scheduling, we could get this schedule:

(1) p_1 : *evaluate* balance + deposit; // $0+1$
(2) p_2 : *evaluate* balance + deposit; // $0+2$
(3) p_1 : *assign* balance = 1;
(4) p_2 : *assign* balance = 2;

Considering all possible schedules, instead of always arriving at the expected balance of 3*, we can get a balance of* 1, 2*, or* 3 *depending on scheduling decisions.*

Remarks:

- If a result depends on scheduling decisions, we have a *race condition.*
- Race conditions make the behavior of programs unpredictable; they are notoriously difficult to understand.
- We formalize the prevention of race conditions as follows:

Definition 9.12 (Mutual Exclusion)**.** *We are given a number of processes, each executing the following code sections:*
<Entry> → <Critical Section> → <Exit> → <Remaining Code>
A mutual exclusion algorithm consists of code for entry and exit sections, such that the following holds:

- *Mutual Exclusion: At all times* ***at most one*** *process is in the critical section.*
- *No deadlock: If some process manages to get to the entry section, later* ***some*** *(possibly different) process will get to the critical section.*

Sometimes we additionally ask for

- *No starvation (or no lockout): If some process manages to get to the entry section, later* ***the same*** *process will get to the critical section.*
- *Unobstructed exit: No process can get stuck in the exit section.*

Remarks:

- Using RMW primitives one can build mutual exclusion algorithms quite easily. Algorithm 9.14 shows an example with the test-and-set primitive.

Theorem 9.13. *Algorithm 9.14 solves the mutual exclusion problem as in Definition 9.12.*

Algorithm 9.14 Mutual Exclusion: Test-and-Set

```
Init: Shared memory word $W := 0$
<Entry>
1: lock($W$)                                  // Algorithm 9.15
<Critical Section>
2: ...
<Exit>
3: unlock($W$)                                // Algorithm 9.16
<Remainder Code>
4: ...
```

Algorithm 9.15 lock()

```
Init: Shared memory word $W := 0$
1: repeat
2:     $r$ := test-and-set($W$)
3: until $r = 0$
```

Algorithm 9.16 unlock()

```
Init: Shared memory word $W := 0$
1: $W := 0$
```

Proof. Mutual exclusion follows directly from the test-and-set definition: Initially W is 0. Let p_i be the i^{th} process to successfully execute the test-and-set, where successfully means that the result of the test-and-set is 0. This happens at time t_i. At time t'_i process p_i resets the shared memory word W to 0. Between t_i and t'_i no other process can successfully test-and-set, hence no other process can enter the critical section concurrently.

Proving no deadlock works similar: One of the processes loitering in the entry section will successfully test-and-set as soon as the process in the critical section exited.

Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit. □

Remarks:

- No starvation, on the other hand, is not given by this algorithm. Even with only two processes, there are asynchronous executions where always the same process wins the test-and-set.
- Algorithm 9.14 can be adapted to guarantee fairness (no starvation), essentially by ordering the processes in the entry section in a queue.
- The problem of providing mutual exclusion is called *synchronization*, and the algorithms and data structures used to provide mutual exclusion are called *synchronization mechanisms.*

- A natural question is whether one can achieve mutual exclusion with only reads and writes, that is without advanced RMW operations. The answer is yes!
- The general idea of Algorithm 9.17 is that process p_i (for $i \in \{0,1\}$) has to mark its desire to enter the critical section in a "want" memory word W_i by setting $W_i := 1$. Only if the other process is not interested ($W_{1-i} = 0$) access is granted. This however is too simple since we may run into a deadlock. This deadlock (and at the same time also starvation) is resolved by adding a priority variable Π.
- Note that Line 3 in Algorithm 9.17 represents a "spinlock" or "busy-wait", similar to Algorithm 9.15.

Algorithm 9.17 Mutual Exclusion: Peterson's Algorithm

```
Initialization: Shared words $W_0, W_1, \Pi$, all initially 0.
Code for process $p_i$ , $i = \{0,1\}$
<Entry>
1: $W_i := 1$
2: $\Pi := 1 - i$
3: repeat until $\Pi = i$ or $W_{1-i} = 0$ end repeat
<Critical Section>
4: ...
<Exit>
5: $W_i := 0$
<Remainder Code>
6: ...
```

Theorem 9.18. *Algorithm 9.17 solves the mutual exclusion problem as in Definition 9.12.*

Proof. The shared variable Π elegantly grants priority to the process that passes Line 2 first. If both processes are competing, only process p_Π can access the critical section because of Π. The other process $p_{1-\Pi}$ cannot access the critical section because $W_\Pi = 1$ (and $\Pi \neq 1 - \Pi$). The only other reason to access the critical section is because the other process is in the remainder code (that is, not interested). This proves mutual exclusion!

No deadlock comes directly with Π: Process p_Π gets direct access to the critical section, no matter what the other process does.

Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit.

Thanks to the shared variable Π also no starvation (fairness) is achieved: If a process p_i loses against its competitor p_{1-i} in Line 2, it will have to wait until the competitor resets $W_{1-i} := 0$ in the exit section. If process p_i is unlucky it will not check $W_{1-i} = 0$ early enough before process p_{1-i} sets $W_{1-i} := 1$ again in Line 1. However, as soon as p_{1-i} hits Line 2, process p_i gets the priority due to Π, and can enter the critical section. □

Remarks:

- Extending Peterson's Algorithm to more than 2 processes can be done by a tournament tree, like in tennis. With n processes every process needs to win $\log n$ matches before it can enter the critical section. More precisely, each process starts at the bottom level of a binary tree, and proceeds to the parent level if winning. Once winning the root of the tree it can enter the critical section. Thanks to the priority variables Π at each node of the binary tree, we inherit all the properties of Definition 9.12.
- Note that Line 3 in Peterson's Algorithm does a lot of busy waiting where process A just keeps checking whether it is its turn while process B is in its critical section, thereby wasting computation time for the whole time where B is in its critical section.
- Implementing Peterson's Algorithm on modern computers can easily fail because of optimizations like instruction reordering that modern compilers offer, or memory access reordering of processing cores. The problems posed by these optimizations can be prevented with some care, but the solutions to these problems are more expensive than the synchronization mechanisms that we discuss in the next Section.

9.5 Semaphores

Definition 9.19 (Semaphore)**.** *A **semaphore** is a non-negative integer variable that can only be modified via atomic operations **wait()** and **signal()**.*

- *wait() checks if the semaphore is strictly positive, and if so, decrements it; if it is* 0*, the calling process is blocked until the semaphore becomes positive.*
- *signal() unblocks a process that is blocked on the semaphore if one exists, and otherwise increments the semaphore.*

Remarks:

- Internally, a semaphore contains an integer S initialized to a non-negative value, a list L of blocked processes, and a memory word W.

Remarks:

- OS and programming libraries offer semaphores as a synchronization mechanism.
- What value a semaphore is initialized depends on how it is used.

Definition 9.22 (Mutex, Counting Semaphore)**.** *A semaphore that only takes the values* 0 *and* 1 *is called a **binary semaphore** or a **mutex** or a **lock**. A semaphore that takes more than two values is called a **counting semaphore**.*

Algorithm 9.20 Semaphore: wait()

```
Input: process $P$ that called wait()
<Entry>
lock($W$)                                        // Algorithm 9.15
<Critical Section>
if $S == 0$ then
    $L$.addAsLast($P$);
  <Exit>
    unlock($W$)                                  // Algorithm 9.16
    $P$.block();   // changes state of $P$ to blocked, deschedules it
else
    $S$--;
  <Exit>
    unlock($W$)                                  // Algorithm 9.16
end if
```

Algorithm 9.21 Semaphore: signal()

```
<Entry>
lock($W$)                                        // Algorithm 9.15
<Critical Section>
if $L$ is not empty then
    $P = L$.removeFirst();
    $P$.unblock();                    // changes state of $P$ to ready
else
    $S$++;
end if
<Exit>
unlock($W$)                                      // Algorithm 9.16
```

Remarks:

- Decrementing a mutex from 1 to 0 is also called *locking* it, while incrementing a mutex from 0 to 1 is *unlocking* it.
- Mutexes provide mutual exclusion. Some OS offer data structures called mutex that behave like a binary semaphore.
- Counting semaphores are useful when there is a limited amount of a given resource that concurrent processes can use.
- Notice that signal() (see Algorithm 9.21) has no way to check whether the semaphore is incremented above an intended maximum value. Thus a semaphore that is used incorrectly can take values outside of the intended range without providing feedback to the user.

9.6 Classic Problems in Synchronization

To illustrate what problems can occur in a concurrent environment, we show three classical problems along with solutions using semaphores.

Definition 9.23 (Dining Philosophers Problem)**.** *In the* ***dining philosophers problem****, k philosophers sit around a round table. Between every two philsophers, there is exactly one chopstick. In the middle of the table there is a big bowl of food. A philosopher needs two chopsticks to eat and can only pick up the two chopsticks to her left and right. Each philosopher keeps thinking until she gets hungry, then she eats (this is the critical section), then thinks again, and so on. Is there an algorithm that ensures neither deadlock nor starvation occur?*

Remarks:

- We could proceed as follows: every hungry philosopher tries to get both chopsticks to either side of her, then eats, and only then releases the chopsticks.
- In this solution, a deadlock occurs if each philosopher grabbed a chopstick since then none of them can eat and none of them will release their chopstick.
- We can prevent this by numbering philosophers and chopsticks in a clockwise fashion from 0 to $k-1$, with chopstick i to the right-hand side of philosopher i. Each philosopher tries to grab the even-numbered chopstick she can reach first; if k is odd, then philosopher $k-1$ will have two even-numbered chopsticks (0 and $k-1$), and she tries to grab 0 first.
- Algorithm 9.24 is one way to solve the dining philosophers problem.

Algorithm 9.24 Dining Philosophers Algorithm for Process i

```
Input: process ID $i \in \{0, \dots, k-1\}$
Shared data structures: semaphore array chopsticks[k], all initially 1
if $i$ is even then
    even = $i$ mod $k$;
    odd = $i + 1$ mod $k$;
else
    even = $i + 1$ mod $k$;
    odd = $i$ mod $k$;
end if
while true do
    thinkUntilHungry();
    chopSticks[even].wait($i$);
        chopSticks[odd].wait($i$);
            eat();
        chopSticks[odd].signal();
    chopSticks[even].signal();
end while
```

Definition 9.25 (Producer-Consumer Problem)**.** *The* ***producer-consumer problem*** *(or* ***bounded-buffer problem****) consists of two processes that have access to a shared buffer of a fixed size that acts like a queue. The* ***producer***

process wants to put units of data into the buffer if it is not full, the ***consumer*** *process wants to take units of data out of the buffer if it is not empty.*

Remarks:

- The producer-consumer problem occurs in situations where one process needs the output of another process to continue working, e.g. an assembler that needs the output of a compiler.
- We can solve the producer-consumer problem using three semaphores: one counts how much empty **space** there is in the buffer, one counts how much **data** there is in the buffer, and a **mutex** for modifying the buffer, see Algorithms 9.26 and 9.27. The solution also works for multiple producers and multiple consumers.

Algorithm 9.26 Producer

```
Input: process ID $i$
Shared data structures: buffer B, semaphore space initialized to size of B,
  semaphore data initially 0, semaphore mutex initially 1
while true do
    space.wait($i$);
    mutex.wait($i$);
        B.put(generateData());          // put new data into buffer
        data.signal();
    mutex.signal();
end while
```

Algorithm 9.27 Consumer

```
Input: process ID $i$
Shared data structures: buffer B, semaphore space initialized to size of B,
  semaphore data initially 0, semaphore mutex initially 1
while true do
    data.wait($i$);
    mutex.wait($i$);
        newData = B.pop();              // remove data from buffer
        space.signal();
    mutex.signal();
    process(newData);
end while
```

Definition 9.28 (Readers-Writers Problem)**.** *In the* ***readers-writers problem*** *there is shared data that some processes (the* ***readers****) want to read from time to time, while other processes (the* ***writers****) want to modify. Multiple readers should be able to read the data at the same time, but while a writer is modifying the data, no other process can be allowed to read or modify the data. Is there an algorithm to make sure no deadlock and no starvation occur?*

Remarks:

- Notice that the readers-writers problem is a little underspecified: what if readers want to read and writers want to write at the same time, in what order should they be allowed to do so?
  - One variant of the problem, the *Readers-writers problem*, prioritizes readers: if at least one process is already reading, no other process that wants to start reading can be kept waiting. Here, the possibility of writer starvation is built into the problem, and not just into solutions to the problem!
  - The *readers-Writers problem* prioritizes writers, where a process can only start reading when no writer is active. The possibility of reader starvation is built in here.
- For the Readers-writers problem, we use a variable **readCount** to count how many processes are currently reading the data. Writers are only allowed to modify the data when **readCount** is zero, while readers increment it whenever they start reading and decrement it when they stop. We protect modification of **readCount** with a mutex **readCountMutex**.
- To make sure that no other process accesses the data while a writer is modifying it, we use a separate mutex **accessMutex**. The first process to start reading the data locks **accessMutex** to show that reading is currently ongoing, and the last process to stop reading unlocks it again. Every writer has to lock **accessMutex** before writing.
- Algorithms 9.29 and 9.30 solve the Readers-writers problem.

Remarks:

- As we have seen, semaphores are powerful in solving synchronization problems. However, they have problems: the use of semaphores is scattered throughout the code of several processes, making the code hard to understand, and a single misuse of a single semaphore can already produce bugs.
- Such bugs can lead to undefined behavior. Since they often only manifest under certain race conditions, they are hard to debug.
- The fundamental problem with semaphores is that the synchronization mechanism is decoupled from the data it protects.

9.7 Monitors

Definition 9.31 (Monitor)**.** *A* ***monitor*** *is an abstract data type that encapsulates* ***shared data*** *with* ***methods*** *to operate on the data. At most one process can execute any method of the monitor at any given time (****mutual exclusion****).*

Algorithm 9.29 Readers-writers problem: Reader

Input: process ID i
Shared data structures: semaphores `readCountMutex`, `accessMutex` both initially 1, integer `readCount:=0`

```
while true do
    readCountMutex.wait($i$);
        readCount++;
        if readCount == 1 then // first reader waits until last writer finishes
            accessMutex.wait($i$);
        end if
    readCountMutex.signal();
    read();
    readCountMutex.wait($i$);
        readCount--;
        if readCount == 0 then // last reader lets writers start writing
            accessMutex.signal();
        end if
    readCountMutex.signal();
end while
```

Algorithm 9.30 Readers-writers problem: Writer

Input: process ID i
Shared data structures: semaphore `accessMutex` initially 1

```
while true do
    accessMutex.wait($i$);
        write();
    accessMutex.signal();
end while
```

Remarks:

- The shared data encapsulated by the monitor is only accessible via the monitor's methods.
- In the Java Virtual Machine, the keyword `synchronized` is implemented as a monitor.
- When a process executes a method of a monitor, we say the process *is active in* the monitor. If process Q tries to enter monitor M while process P is currently active in M, then Q will block on M.
- While process P is active in monitor M, it can happen that some condition has to be satisfied for the method to complete. For example, let P be a producer in the producer-consumer-problem, and let M be the monitor for the buffer. While P waits until the buffer has empty space, consumers have to be allowed to consume data from the buffer. As the buffer is only accessible via M, P has to block on the condition "buffer not full" and exit M so consumers can consume data. Once the condition is satisfied, P can unblock and attempt to re-enter M.

- Abstractly, it can happen that P needs to block while in M's critical section. When it does, it has to exit the monitor and try to re-enter once the condition it blocked on is satisfied. Monitors coordinate such issues with *condition variables.*

Definition 9.32 (Condition Variable)**.** *A* ***condition variable*** *is a monitor-private queue of processes that represents some condition C. A condition variable offers the following interface:*

- `conditionWait(Semaphore monitorMutex, Process` P`)`*, which performs the following:*
  1. *P is added to the condition variable associated with C.*
  2. *P signals the monitorMutex and then blocks.*
  3. *Once it is unblocked, P waits on the monitorMutex so it can safely re-enter the monitor.*
- `conditionSignal()` *unblocks a process that blocked on C.*

Algorithm 9.33 Condition Variable: conditionWait()

Condition Variable internals: list of processes blocked on this condition variable L, semaphore `conditionMutex` initialized to 1
Input: semaphore `monitorMutex`, process P that called conditionWait()

```
conditionMutex.wait($P$);
    $L$.addAsLast($P$);
conditionMutex.signal();
monitorMutex.signal();      // leave the monitor's critical section
$P$.block();
monitorMutex.wait($P$);         // re-enter monitor once unblocked
```

Algorithm 9.34 Condition Variable: conditionSignal()

Condition Variable internals: list of processes blocked on this condition variable L, semaphore `conditionMutex` initialized to 1
Input: process P that called conditionSignal()

```
conditionMutex.wait($P$);
    if $L$ is not empty then
        $P_{blocked}$ = $L$.removeFirst();
        $P_{blocked}$.unblock();
    end if
conditionMutex.signal();
```

Remarks:

- Algorithms 9.33 and 9.34 implement a condition variable.
- A condition variable does not internally represent the condition it is used to control, instead that condition will be checked for in the monitor. That way, condition variables are very flexible.

- In the example of producer-consumer, we would have the conditions "buffer not full" and "buffer not empty". A producer-consumer-monitor would offer a method `produce(data)` that checks whether "buffer not full" before letting a producer add data, and a method `consume()` that checks whether "buffer not empty" before letting a consumer take data.

- Algorithms 9.35, 9.36 and 9.37 solve the producer-consumer problem with a monitor. The significant difference to the solution with semaphores is that all synchronization mechanisms are now encapsulated in the monitor. This makes them easier to design, understand and debug, and any process can use the methods of the monitor to safely take part without having to implement any synchronization.

Algorithm 9.35 Producer-Consumer-Monitor

```
Internals: semaphore monitorMutex initially 1;
    condition variables bufferNotFull, bufferNotEmpty, initially empty;
    bufferSize:= size of buffer; full:=0
procedure produce(Data data, Process P){
    monitorMutex.wait($P$);
        while full == bufferSize do
            bufferNotFull.conditionWait(monitorMutex, $P$);
        end while
        full++;
        bufferNotEmpty.conditionSignal();
    monitorMutex.signal();
}
procedure consume(Process P){
    monitorMutex.wait($P$);
        while full == 0 do
            bufferNotEmpty.conditionWait(monitorMutex, $P$);
        end while
        full--;
        bufferNotFull.conditionSignal();
    monitorMutex.signal();
}
```

Algorithm 9.36 Producer for Monitor from Algorithm 9.35

```
Internals: Producer-Consumer-Monitor $M$
Input: Producer process $P$
while true do
    $M$.produce($P$.generateData(), $P$);
end while
```

Algorithm 9.37 Consumer for Monitor from Algorithm 9.35

```
Internals: Producer-Consumer-Monitor $M$
Input: Consumer process $P$
while true do
    $M$.consume($P$);
end while
```

Chapter Notes

The first solution to the mutual exclusion problem for two processes was given by the Dutch mathematician T.J. Dekker in 1959 according to an unpublished paper by Edsger W. Dijkstra [3]. Dijkstra himself is widely credited with publishing the founding paper in the field of concurrent programming [2] that solved the mutual exclusion problem for n processes.

Semaphores were invented by Dijkstra around 1962/1963 [4]. He called the operations P() and V() (from the Dutch words "probeer" for "try" and "verhoog" for "increase") instead of wait() and signal().

Monitors were the result of a number of refinements to the basic idea of encapsulating data with synchronization mechanisms by C.A.R. Hoare and P. Brinch Hansen [6, 7, 8]. The ideas by Hoare and Hansen have slightly different semantics which was too much detail for this script; most concurrency textbooks expound on Hoare monitor semantics and Hansen monitor semantics.

The Dining Philosophers Problem, the Producer Consumer Problem, and the Readers Writers Problem (both versions) are classic examples often used to evaluate newly suggested synchronization mechanisms. Dijkstra posed the Dining Philosophers Problem as an exam problem in 1965 in terms of access to tape drives; Hoare came up with the formulation in terms of hungry philosophers soon after. Dijkstra also introduced the Producer Consumer Problem [5] in 1972 to illustrate the usefulness of Hoare's/Hansen's developing ideas about conditional critical sections that would evolve and be incorporated into monitors.

The two original variants of the Readers Writers Problem were posed and solved with the use of semaphores by Courtois, Heymans, and Parnas [1]. The Readers-writers problem is called the *first readers-writers problem* in the literature, and the readers-Writers problem is the *second readers-writers problem* in the literature; using "first" and "second" derives from Courtois et al. naming the variants "Problem 1" and "Problem 2" in their paper.

This chapter was written in collaboration with Georg Bachmeier.

Bibliography

[1] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[2] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[3] E. W. Dijkstra. Cooperating sequential processes. In Per Brinch Hansen, editor, *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 3–61. Springer New York, 2002.

[4] E. W. Dijkstra. Over de sequentialiteit van procesbeschrijvingen, undated. circulated privately; original at `https://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html` English translation at `https://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html`.

[5] Edsger W Dijkstra. Information streams sharing a finite buffer. *Information Processing Letters*, 1(5):179–180, 1972.

[6] Per Brinch Hansen. *Operating system principles.* Prentice-Hall, Inc., 1973.

[7] C. A. R. Hoare. A structured paging system. *The Computer Journal*, 16(3):209–215, 1973.

[8] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.