Chapter 9

Modelling III: Asynchronous Shared Memory Model

In this chapter, we give a formal model for asynchronous shared memory systems. This model is presented in terms of the general I/O automaton model for asynchronous systems that we defined in Chapter 8.

A shared memory system consists of a collection of communicating processes, as does a network system. But this time, instead of sending and receiving messages over communication channels, the processes perform instantaneous operations on shared variables.

9.1 Shared Memory Systems

Informally speaking, an *asynchronous shared memory system* consists of a finite collection of processes interacting with each other by means of a finite collection of shared variables. The variables are used only for communication among the processes in the system. However, so that the rest of the world can interact with the shared memory system, we also assume that each process has a *port*, on which it can interact with the outside world using input and output actions. The interactions are depicted in Figure 9.1.

We model a shared memory system using I/O automata, in fact, using just a single I/O automaton with its external interface consisting of the input and output actions on all the ports. It might seem more natural to use several automata, one per process and one per shared variable. However, that leads to some complications we would rather avoid in this book. For instance, if each process and each variable were an I/O automaton and we combined them using ordinary I/O automaton composition, then we would get a system in which an

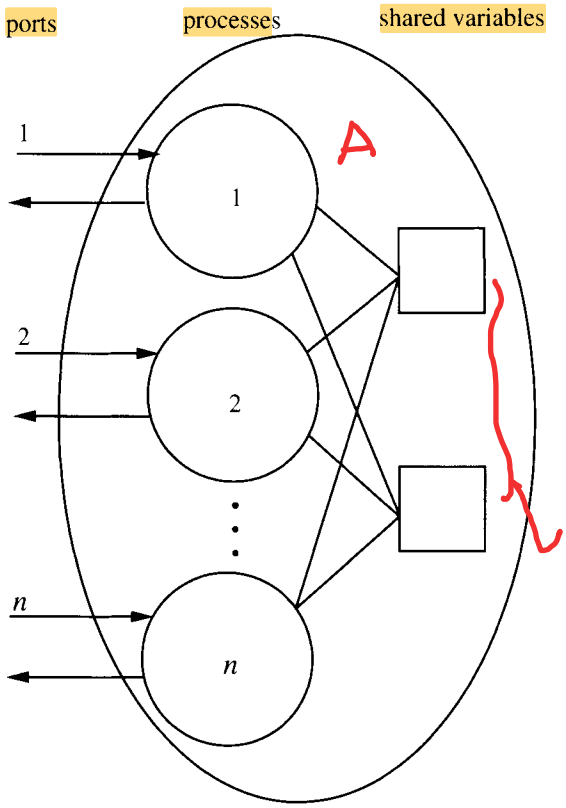


Figure 9.1: An asynchronous shared memory system.

operation by a process i on a shared variable x would be modelled by a pair of events—an invocation that is an output of process i and an input of variable x, followed by a response that is an output of variable x and an input of process i. But then the system would also have some executions in which these pairs of events are split. For instance, several operations could be invoked before the first of them returns. This kind of behavior does not occur in the shared memory systems that we are trying to model.

One way out of this difficulty would be to consider a restricted subset of all the possible executions—those in which invocations and corresponding responses occur consecutively. A second way out would be to model only the processes as I/O automata, but to model the shared variables as state machines of a different kind (with invocations and responses combined into single events); in this case, a new composition operation would have to be defined to allow combination of the process and variable automata into one I/O automaton. Since these approaches

introduce their own complexities—restricted subsets of the set of executions, pairs of events, a new kind of state machine, or a new operation—we sidestep all these issues by just modelling the entire system as one big I/O automaton A. We capture the process and variable structure within automaton A by means of some locality restrictions on the events.

As in the synchronous network model, we assume that the processes in the system are indexed by $1, \ldots, n$. Suppose that each process i has an associated set of *states*, $states_i$, among which some are designated as *start states*, $start_i$. Also suppose that each shared variable x in the system has an associated set of *values*, $values_x$, among which some are designated as the *initial values*, $initial_x$. Then each state in $states(A)$ (the set of states of the system automaton A) consists of a state in $states_i$ for each process i, plus a value in $values_x$ for each shared variable x. Each state in $start(A)$ consists of a state in $start_i$ for each process i, plus a value in $initial_x$ for each shared variable x.

We assume that each action in $acts(A)$ is associated with one of the processes. In addition, some of the internal actions in $int(A)$ may be associated with a shared variable. The input actions and output actions associated with process i are used for interaction between process i and the outside world; we say they occur on *port* i. The internal actions of process i that do not have an associated shared variable are used for local computation, while the internal actions of i that are associated with shared variable x are used for performing operations on x.

The set $trans(A)$ of transitions has some locality restrictions, which model the process and shared variable structure of the system. First, consider an action π that is associated with process i but with no variable; as we noted above, π is used for local computation. Then only the state of i can be involved in any π step. That is, the set of π transitions can be generated from some set of triples of the form (s, π, s'), where $s, s' \in states_i$, by attaching any combination of states for the other processes and values for the shared variables to both s and s' (the same combination to both).

On the other hand, consider an action π that is associated with both a process i and a variable x; as we noted above, π is used by i to perform an operation on x. Then only the state of i and the value of x can be involved in any π step. That is, the set of π transitions can be generated from some set of triples of the form $((s, v), \pi, (s', v'))$, where $s, s' \in states_i$ and $v, v' \in values_x$, by attaching any combination of states for the other processes and values for the other shared variables. There is a technicality: if π is associated with process i and variable x, then whether or not π is enabled should depend only on the state of process i, although the resulting changes may also depend on the value of x. That is, if π is enabled when the state of i is s and the value of x is v, then π is also enabled when the state of i is s and when x has any other value v'.

The task partition $tasks(A)$ must be consistent with the process structure: that is, each equivalence class (task) should include locally controlled actions of only one process. In many cases that we will consider, there will be exactly one task per process—this makes sense, for example, if each process is a sequential program. In this case, the standard definition of fairness for I/O automata, given in Section 8.3, says that each process gets infinitely many chances to take steps. In the more general case, where there can be several tasks per process, the fairness definition says that each task gets infinitely many chances to take steps.

Example 9.1.1 Shared memory system

Let V be a fixed value set. Consider a shared memory system A consisting of n processes, numbered $1, \ldots, n$, and a single shared variable x with values in $V \cup \{unknown\}$, initially $unknown$. The inputs are of the form $init(v)_i$, where $v \in V$ and i is a process index. The outputs are of the form $decide(v)_i$. The internal actions are of the form $access_i$. All the actions with subscript i are associated with process i, and in addition, the $access$ actions are associated with variable x.

After process i receives an $init(v)_i$ input, it accesses x. If it finds $x = unknown$, then it writes its value v into x and decides v. If it finds $x = w$, where $w \in V$, then it does not write anything into x, but decides w.

Formally, each set $states_i$ consists of local variables.

States of i:
$status \in \{idle, access, decide, done\}$, initially $idle$
$input \in V \cup \{unknown\}$, initially $unknown$
$output \in V \cup \{unknown\}$, initially $unknown$

The transitions are

Transitions of i:

$init(v)_i$
Effect:
$input := v$
if $status = idle$ then
$status := access$

$access_i$
Precondition:
$status = access$
Effect:
if $x = unknown$ then $x := input$
$output := x$
$status := decide$

$decide(v)_i$
Precondition:
$status = decide$
$output = v$
Effect:
$status := done$

There is one task per process, which contains all the *access* and *decide* actions for that process.

It is not hard to see that in every fair execution α of A, any process that receives an *init* input eventually performs a *decide* output. Moreover, every execution (fair or not, and with any number of *init* events occurring anywhere) satisfies the "agreement property" that no two processes decide on different values, and the "validity property" that every decision value is the initial value of some process.

We can formulate these correctness claims in terms of trace properties, according to the definition in Section 8.5.2. For example, let P be the trace property such that $sig(P) = extsig(A)$ and $traces(P)$ is the set of sequences β of actions in $acts(P)$ satisfying the following conditions:

1. For any i, if exactly one $init_i$ event appears in β, then exactly one $decide_i$ event appears in β.
2. For any i, if no $init_i$ event appears in β, then no $decide_i$ event appears in β.
3. (Agreement) If $decide(v)_i$ and $decide(w)_j$ both appear in β, then $v = w$.
4. (Validity) If a $decide(v)_i$ event appears in β, then some $init(v)_j$ event (for the same v) appears in β.

It is then possible to show that $fairtraces(A) \subseteq traces(P)$. The proof is left for an exercise.

9.2 Environment Model

Sometimes it is useful to model the environment of a system as an automaton also. This provides an easy way to describe assumptions about the environment's behavior. For instance, in Example 9.1.1, we might like to specify that the environment submits *exactly one* $init_i$ input for each i, or maybe *at least one* for each i. For shared memory systems that arise in practice, the environment can often be described as a collection of independent *user automata*, one per port.

Example 9.2.1 Environment model

We describe an environment for the shared memory system A described in Example 9.1.1. The environment is a single I/O automaton that is composed (using the composition operation for I/O automata defined in Section 8.2.1) of one *user automaton*, U_i, for each process index i. U_i's code is as follows.

U_i automaton:

Signature:

Input:
$decide(v)_i$, $v \in V$

Internal:
$dummy_i$

Output:
$init(v)_i$, $v \in V$

States:

$status \in \{request, wait, done\}$, initially $request$
$decision \in V \cup \{unknown\}$, initially $unknown$
$error$, a Boolean, initially $false$

Transitions:

$init(v)_i$
Precondition:
$status = request$ or $error = true$
Effect:
if $error = false$ then $status := wait$

$dummy_i$
Precondition:
$error = true$
Effect:
none

$decide(v)_i$
Effect:
if $error = false$ then
if $status = wait$ then
$decision := v$
$status := done$
else $error := true$

Tasks:
All locally controlled actions are in one class.

Thus, U_i initially performs an $init_i$ action, then waits for a decision. If the shared memory system produces a decision without a preceding $init_i$ or produces two decisions, then U_i sets an $error$ flag, which allows it to output any number of $init$s at any time. (The presence of the $dummy_i$ action allows it also to choose not to perform outputs.) Of course, the given shared memory system is not supposed to cause such errors.

The composition of the shared memory system A with all the U_i, $1 \leq i \leq n$, is depicted in Figure 9.2. This composition is quite well-behaved: in any fair execution of the composition, there is exactly one $init_i$ event and exactly one $decide_i$ event for each i. Moreover, the $decide$ events satisfy appropriate agreement and validity conditions.

More formally, let Q be the trace property such that $sig(Q)$ consists of outputs $init(v)_i$ and $decide(v)_i$ for all i and v, and such that

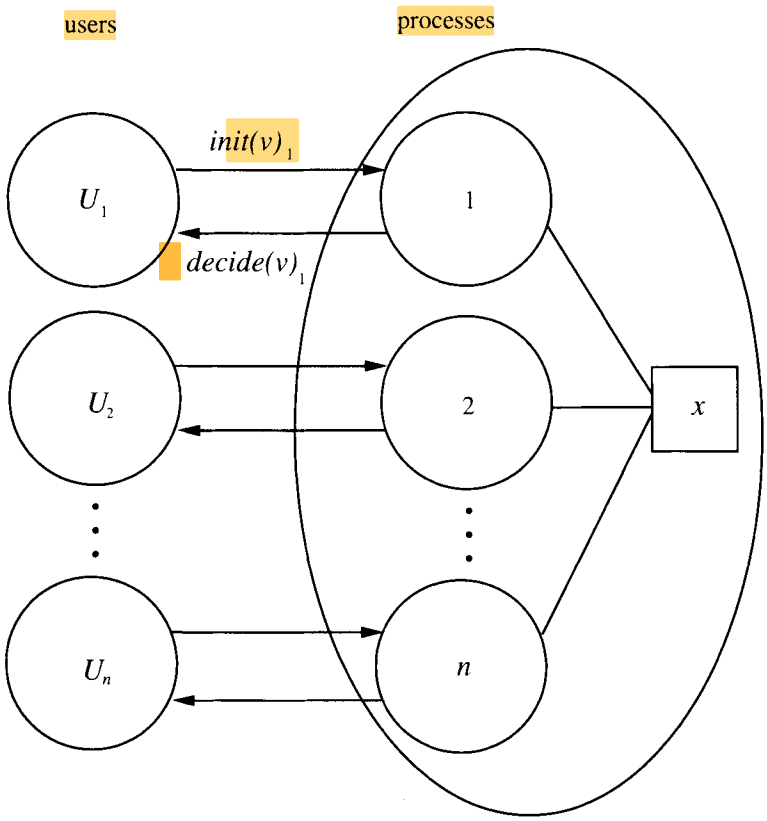


Figure 9.2: Users and shared memory system.

$traces(Q)$ is the set of sequences β of actions in $acts(Q)$ satisfying the following conditions:

1. For any i, β contains exactly one $init_i$ event followed by exactly one $decide_i$ event.
2. (Agreement) If $decide(v)_i$ and $decide(w)_j$ both appear in β, then $v = w$.
3. (Validity) If a $decide(v)_i$ event appears in β, then some $init(v)_j$ event (for the same v) appears in β.

Then it is possible to show that $fairtraces(A \times \prod_{1 \leq i \leq n} U_i) \subseteq traces(Q)$. The proof is left for an exercise.

9.3 Indistinguishable States

We define a notion of indistinguishability that will be useful in some impossibility proofs in Chapter 10.

Consider an n-process shared memory system A and a collection of users U_i, $1 \leq i \leq n$. Let s and s' be two states of the composed system $A \times \prod_{1 \leq i \leq n} U_i$. Then we say that s and s' are *indistinguishable* to process i if the state of process i, the state of U_i, and the values of all the shared variables are the same in s and s'. We write $s \overset{i}{\sim} s'$ to indicate that s and s' are indistinguishable to i.

9.4 Shared Variable Types

In the general definition we have given for shared memory systems, we have not restricted the types of operations a process may perform on a shared variable when it accesses the variable. That is, when a process i accesses a variable x, we have allowed arbitrary changes to the state of i and the value of x to occur, depending in arbitrary ways on the previous state of i and value of x. But in practice, shared variables normally support only a fixed set of operations, such as read and write operations, or a combined read-modify-write operation. In this subsection, we define the notion of a *variable type*, and say what it means for a shared memory system to observe type restrictions.[1]

A *variable type* consists of

- a set V of values
- an initial value $v_0 \in V$
- a set of *invocations*
- a set of *responses*
- a function f : *invocations* $\times V \rightarrow$ *responses* $\times V$

The function f says what happens when a given invocation arrives at the variable and the variable has a given value; f describes the new value the variable takes on and the response that is returned. Note that a variable type is *not* an I/O automaton, even though some of its components look similar to I/O automaton components. Most importantly, in a variable type, the invocations and responses are thought of as occurring together as part of one function application, whereas

[1] The definition we use here requires the variable to behave deterministically. This could be generalized to allow nondeterminism, but we would rather avoid the complication here, since it is not needed for the results in this book.

in the I/O automaton model, inputs and outputs are separate actions (and other actions may occur between them).

Suppose we have a shared memory system A. What does it mean to say that shared variable x in system A is of a given variable type? It means, first, that the set $values_x$ must be equal to the set V of values of the type, and that the set $initial_x$ of initial values for x consists of just one element, v_0. Moreover, all the transitions involving x must be describable in terms of the invocations and responses allowed by the type. Namely, each action involving x must be associated with some invocation a of the variable type. Moreover, for each process i and each invocation a, the set of transitions involving i and a must be describable in the following form, where p is some predicate on $states_i$ and g is some relation, $g \subseteq states_i \times responses \times states_i$. (In the code, we use the notation $state_i$ to denote the state of process i.)

Transitions involving i and a
  Precondition:
    $p(state_i)$
  Effect:
    $(b, x) := f(a, x)$
    $state_i :=$ any s such that $(state_i, b, s) \in g$

This code means that the determination that variable x is to be accessed by process i using invocation a is made according to predicate p (which just involves the state of i). If this access is to be performed, then the function f for the variable type is applied to the invocation a and the value of variable x to determine a response b and a new value for x. The response b is then used by process i to update its state, in some way allowed by relation g.

In the descriptions of shared memory algorithms in this book, transitions involving accesses to shared variables of particular types will not be written explicitly in terms of predicates p and relations g as above. However, theoretically, they could all be expressed in this style.

Example 9.4.1 Read/write shared variables (registers)

The most frequently used variable type in multiprocessors is one supporting only read and write operations. A variable of this type is known as a *read/write variable*, or a *read/write register*, or just a *register*.

A read/write register comes equipped with an arbitrary set V of values and an arbitrary initial value $v_0 \in V$. Its invocations are *read*

and $write(v), v \in V$. Its responses are $v \in V$ and ack.[2] Its function f is defined by: $f(read, v) = (v, v)$ and $f(write(v), w) = (ack, v)$.

Note that variable x in the system of ~~Example 9.1.1~~ cannot be described as a read/write register, because there is no way that the given accesses could be rewritten in the form given above. It is possible to rewrite the algorithm so that x is a register, for example, by separating each access into a read and a write step. The resulting process code might look as follows. The *status* value *access* is replaced by two new *status* values, *read* and *write*.

Transitions:

$init(v)_i$
  Effect:
    $input := v$
    if $status = idle$ then $status := read$

$read_i$
  Precondition:
    $status = read$
  Effect:
    if $x = unknown$ then
      $output := input$
      $status := write$
    else
      $output := x$
      $status := decide$

$write(v)_i$
  Precondition:
    $status = write$
    $v = input$
  Effect:
    $x := v$
    $status := decide$

$decide(v)_i$
  Precondition:
    $status = decide$
    $output = v$
  Effect:
    $status := done$

The task partition again groups together all locally controlled actions of process i. Although this code is not explicitly written in terms of a predicate p and a relation g, note that it could easily be rewritten in this way. For instance, for the $read_i$ action, the predicate p is simply "$status = read$," and the relation g is just the set of triples $(s, b, s') \in states_i \times (V \cup \{unknown\}) \times states_i$ such that s' is obtained from s by the code:

```
if b = unknown then
  output := input
  status := write
else
  output := b
  status := decide
```

For the $write(v)_i$ action, the predicate p is simply "$status = write$ and $v = input$," and the relation g is just the set of triples $(s, b, s') \in$

[2] The invocations and responses will sometimes also include additional information such as the name of the register. We mostly ignore such complications here.

$states_i \times (V \cup \{unknown\}) \times states_i$ such that s' is obtained from s by the code:

$status := decide$

So x is a read/write shared variable.

Notice that when we rewrite the algorithm in this way, the agreement condition mentioned in Example 9.1.1 is no longer guaranteed.

Example 9.4.2 Read-modify-write shared variables

Another important variable type allows the powerful *read-modify-write* operation. In one instantaneous read-modify-write operation on a shared variable x, a process i can do all of the following:

1. Read x.
2. Carry out some computation, possibly using the value of x, that modifies the state of i and determines a new value for x.
3. Write the new value to x.

It is not easy to implement a general read-modify-write operation using the usual primitives provided by multiprocessors. The shared memory model requires not only that each access to the variable be *indivisible*, but also that all the processes should get *fair* turns to perform such accesses. Implementing this fairness requires some sort of low-level arbitration mechanism.

As we have described it, it is not obvious that read-modify-write variables can be modelled in terms of variable types: the read-modify-write operation appears to involve two accesses to the variable rather than just one as required. One way to do this is to have a process that wishes to access the variable determine, based on its state, a function h to use as an invocation of the variable. The function h provides the information from the process's state that is needed to determine the transition, expressed in the form of a function to apply to the variable. The effect of the function h on the variable when it has value v is to change the variable's value to $h(v)$ and return the previous value v to the process. The process can then change its state, based on its old state and v.

Formally, a read-modify-write variable can have any set V of values and any $v_0 \in V$ as an initial value. Its invocations are all the functions h, where $h : V \to V$. Its responses are $v \in V$. Its function f is defined by $f(h, v) = (v, h(v))$. That is, it responds with the prior value and updates its value based on the submitted function.

For instance, in Example 9.1.1, the function submitted by a process to the variable is of the form h_v, where

$$h_v(x) = \begin{cases} v, & \text{if } x = \textit{unknown} \\ x, & \text{otherwise} \end{cases}$$

The particular h_v submitted by a process uses the process's *input* as the value of v. A return value of *unknown* causes *output* to be set to the value of *input*, while a return value of $v \in V$ causes *output* to be set to v. In either case, *status* is appropriately modified.

Example 9.4.3 Other variable types

Many of the variable types used in shared memory multiprocessors include restricted forms of read-modify-write, plus basic operations such as read and write. Some popular restricted form of read-modify-write include *compare-and-swap*, *swap*, *test-and-set*, and *fetch-and-add* operations. These operations are defined as follows. Fix a set V and initial value v_0.

The invocations for *compare-and-swap* operations are of the form *compare-and-swap*(u, v), $u, v \in V$, and the responses are elements of V. The function f is defined for compare-and-swap invocations by

$$f(\textit{compare-and-swap}(u, v), w) = \begin{cases} (w, v), & \text{if } u = w \\ (w, w), & \text{otherwise} \end{cases}$$

That is, if the variable's value is equal to the first argument, u, then the operation resets it to the second argument, v; otherwise, the operation does not change the value of the variable. In either case, the original value of the variable is returned.

The invocations for *swap* operations are of the form *swap*(u), $u \in V$, and the responses are elements of V. The function f is defined for swap invocations by

$$f(\textit{swap}(u), v) = (v, u).$$

That is, the operation writes the input value u into the variable and returns the original variable value v.

The invocations for *test-and-set* operations are of the form *test-and-set*, and the responses are elements of V. The function f is defined for test-and-set by

$$f(\textit{test-and-set}, v) = (v, 1).$$

That is, the operation writes 1 into the variable and returns the original variable value v. (We assume that $1 \in V$.)

Finally, the invocations for *fetch-and-add* operations are of the form *fetch-and-add*(u), $u \in V$, and the responses are elements of V. The function f is defined for fetch-and-add by

$$f(\textit{fetch-and-add}(u), v) = (v, v + u).$$

That is, the operation adds the input value u to the variable value v and returns the original value v. (This operation requires that the set V support a notion of addition.)

We can define the *executions* of a variable type in a natural way, as finite sequences $v_0, a_1, b_1, v_1, a_2, b_2, v_2, \ldots, v_r$ or infinite sequences $v_0, a_1, b_1, v_1, a_2, b_2, v_2, \ldots$. Here, the v's are values in V, v_0 is the initial value of the variable type, the a's are invocations, the b's are responses, and the quadruples $v_k, a_{k+1}, b_{k+1}, v_{k+1}$ satisfy the function of the type. (That is, $(b_{k+1}, v_{k+1}) = f(a_{k+1}, v_k)$.) Also, the *traces* of a type are the sequences of a's and b's that are derived from executions of the type.

Example 9.4.4 Trace of a read/write variable type

The following is a trace of a read/write variable type with $V = \mathbb{N}$ and $v_0 = 0$:

$$\textit{read}, 0, \textit{write}(8), \textit{ack}, \textit{read}, 8$$

We finish this section by defining a simple composition operation for variable types. This lets us regard a collection of separate variable types, each with its own operations, as a single variable type with several components, and with operations acting on the individual components.

We define a countable collection $\{\mathcal{T}_i\}_{i \in I}$ of variable types to be *compatible* if all their sets of invocations are disjoint, and likewise for all their sets of responses. Then the *composition* $\mathcal{T} = \prod_{i \in I} \mathcal{T}_i$ of a countable compatible collection of variable types is defined as follows:

- The set V is the Cartesian product of the value sets of the $\mathcal{T}_i$.
- The initial value v_0 consists of the initial values of the $\mathcal{T}_i$.
- The set of invocations is the union of the sets of invocations of the $\mathcal{T}_i$.
- The set of responses is the union of the sets of responses of the $\mathcal{T}_i$.

- The function f operates "componentwise." That is, consider $f(a, w)$, where a is an invocation of $\mathcal{T}_i$. Function f applies a to the ith component of w, using the function of $\mathcal{T}_i$, to obtain (b, v). It returns b and sets the ith component of w to v.

When I is a finite set, we sometimes use the infix operation symbol $\times$ to denote composition.

Example 9.4.5 Composition of variable types

We describe the composition of two read/write variable types $\mathcal{T}_x$ and $\mathcal{T}_y$. (You should think of x and y as the names of two registers.) Suppose the value sets are V_x and V_y, respectively, and the initial values are $v_{0,x}$ and $v_{0,y}$.

We can only compose these two types if they are compatible. So we disambiguate the invocations and responses of the two types by attaching the (literal) subscript x or y. Then the composed type $\mathcal{T}_x \times \mathcal{T}_y$ has $V_x \times V_y$ as its value set and the pair $(v_{0,x}, v_{0,y})$ as its initial value. Its invocations are $read_x$, $read_y$, $write(v)_x$, $v \in V_x$, and $write(v)_y$, $v \in V_y$. Its responses are v_x, $v \in V_x$, plus v_y, $v \in V_y$, plus ack_x and ack_y.

Now we consider the function f. Let $w = (v, v')$ be an arbitrary element of $V_x \times V_y$. Then f is defined for w by $f(read_x, w) = (v_x, w)$, $f(read_y, w) = (v'_y, w)$, $f(write(v'')_x, w) = (ack_x, (v'', v'))$, and $f(write(v'')_y, w) = (ack_y, (v, v''))$. Thus, a *read* returns the indicated component of the vector, while a *write* updates the indicated component.

9.5 Complexity Measures

In order to measure time complexity in asynchronous shared memory systems, we assume an upper bound of ℓ on process step time. Such an upper bound allows us to prove upper bounds on the time required for events of interest to occur (e.g., for a process that has received an $init_i$ input to produce a $decide_i$ output).

More precisely, we establish a *time complexity measure* for shared memory systems as a special case of the time complexity measure defined for general I/O automata in Section 8.6. That is, we define an upper bound of ℓ for each task C of each process; this imposes an upper bound of ℓ on the time between successive chances by task C to perform a step. We measure the *time* until some

designated event π by the supremum of the times that can be assigned to π by time assignments that respect the upper bounds. Likewise, we measure the *time* between two events of interest by the supremum of the differences between the times that can be assigned to those two events.

Note that our time measure does not take into account any overhead due to contention among processes for accessing a common variable. In multiprocessor settings where such contention is an issue, the time measure must be modified accordingly.

Other interesting measures of complexity for shared memory systems include some static measures such as the number of shared variables and the size of their value sets.

9.6 Failures

The stopping failure of a process i in a shared memory system is modelled using an input action $stop_i$, which causes the stopping failure of all tasks of process i but does not affect any other processes. More precisely, a $stop_i$ event can change only the state of process i, although we do not constrain these state changes except for requiring that they permanently disable all the tasks of process i. We leave open the issue of whether later inputs to process i are ignored, or cause the same changes to the state of process i that they would if no $stop_i$ had occurred, or cause some other state changes. These distinctions do not matter, because the effects of such state changes could never be communicated to any other processes.

Figure 9.3 depicts the architecture for an asynchronous shared memory system with stopping failures.

9.7 Randomization

A probabilistic shared memory system is defined by specializing the general definition of a probabilistic I/O automaton in Section 8.8 to the case where the I/O automaton is a shared memory system.

9.8 Bibliographic Notes

There are no special references for the basic model described in this chapter. It is a garden-variety shared memory model, formulated within the I/O automaton framework. Another model for shared memory systems was defined by Lynch and Fischer [216]; in that model, processes communicate by means of instantaneous accesses to shared variables, but not by means of external events. Kruskal,

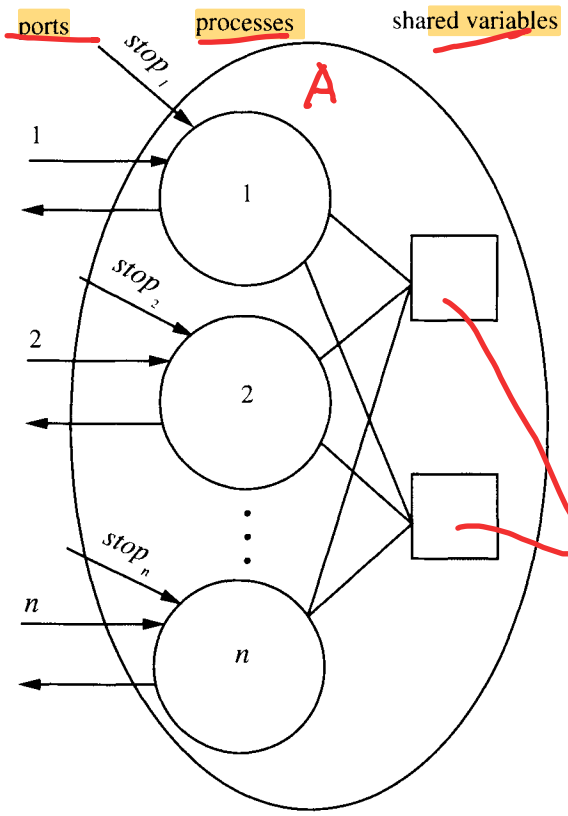


Figure 9.3: Architecture for asynchronous shared memory system with stopping failures.

Rudolph, and Snir [171] defined the various types of variables used in shared memory multiprocesssors.

Dwork, Herlihy, and Waarts [103] have suggested a time complexity measure that takes into account contention for shared memory access. The formal modelling of probabilistic shared memory systems is derived from work by Lynch, Saias, and Segala [208].

9.9 Exercises

9.1. Let A be the shared memory system described in Example 9.1.1.

(a) Prove that $fairtraces(A) \subseteq traces(P)$, where P is the trace property described in Example 9.1.1.

(b) Define an interesting trace safety property Q and show that the (not

necessarily fair) traces of A satisfy it. That is, show $traces(A) \subseteq traces(Q)$. Your property should include mention of what can happen where there is more than one $init_i$ action for the same process i.

9.2. Prove that $fairtraces(A \times \prod_{1 \leq i \leq n} U_i) \subseteq traces(Q)$, where A is the shared memory system described in Example 9.1.1 and Q is the trace property described in Example 9.2.1.

One way to do this is to reformulate Q as the intersection of a safety property S and a liveness property L. S can include the agreement and validity condition, plus part of the first condition—that for each i, the subsequence of actions of i is some prefix of a sequence of the form $init_i$, $decide_i$. L can just say that at least one $init_i$ event and at least one $decide_i$ event occur, for each i. Show that each system component preserves S and use Theorem 8.11 to show that $traces(A \times \prod_{1 \leq i \leq n} U_i) \subseteq traces(S)$. (The fact that A preserves S could be argued from the fact that $traces(A) \subseteq traces(P)$.) Then use the fairness assumptions to show liveness.

9.3. Prove that the following is an invariant of the system $A \times \prod_{1 \leq i \leq n} U_i$ of Example 9.2.1: If $decision_{U_i} \neq unknown$ and $decision_{U_j} \neq unknown$, then $decision_{U_i} = decision_{U_j}$.[3] Do this in two alternative ways:

(a) Based on the fact that $traces(A \times \prod_{1 \leq i \leq n} U_i) \subseteq traces(S)$, proved in Exercise 9.2.

(b) Using the usual method for proving invariants—an induction on the length of an execution leading to a given system state.

9.4. Does the system described in Example 9.4.1, based on a read/write register, satisfy the same trace property P as the system in Example 9.1.1? If so, prove this. If not, then give a counterexample and then state and prove the strongest claims you can for the system's behavior.

9.5. *Research Question:* Define an alternative model for shared memory systems by using I/O automata to model processes only, and by defining a new type of state machine (similar to the model for variable types) for shared variables. Define an appropriate composition operation to combine "compatible" process and shared variable automata into a single I/O automaton to model the entire system. What modifications are needed to the results in subsequent chapters to fit them to your new definitions?

[3] We use the subscript notation to designate the variables belonging to particular automata.

This Page Intentionally Left Blank

Chapter 10

Mutual Exclusion

In this chapter, we begin the study of *asynchronous algorithms*. Asynchronous algorithms are generally quite different from synchronous algorithms, since they must cope with the uncertainty imposed by asynchrony as well as the uncertainty caused by distribution. In asynchronous networks, for example, process steps and message deliveries do not necessarily take place in lock-step synchrony; rather, they may happen in an arbitrary order.

Instead of moving immediately to the study of asynchronous network algorithms, we first study algorithms in the asynchronous shared memory setting. The main reason we do this is that the setting is somewhat simpler. But also, as you will see in Chapter 17, there are close connections between the asynchronous shared memory model and the asynchronous network model. For instance, it is possible to translate algorithms written for the asynchronous shared memory model into versions that can run in asynchronous networks. In this chapter and Chapter 11, we will not consider failures very much; asynchrony alone introduces enough interesting complications for now.

The problem we study here is the *mutual exclusion* problem, a problem of managing access to a single indivisible resource (e.g., a printer) that can only support one user at a time. Alternatively, it can be viewed as the problem of ensuring that certain portions of program code are executed within *critical regions*, where no two programs are permitted to be in critical regions at the same time. It is not known which users are going to request the resource nor when they will do so. This problem arises in both centralized and distributed operating systems.

We present several mutual exclusion algorithms for the read/write shared memory model, starting with an early algorithm by Dijkstra. Subsequent algorithms improve on Dijkstra's by guaranteeing fairness to the different users and by weakening the type of shared memory that is used. We then give a fundamen-

tal lower bound for the number of read/write shared variables that are needed to solve the problem. Finally, we give a collection of upper and lower bound results for the case where the shared memory consists of stronger, read-modify-write shared variables.

This chapter is quite long. The main reason for its length is that we are using it not just to present a collection of algorithms and impossibility results, but also to introduce many ideas that will be used in the rest of the book. These include techniques for modelling shared memory systems and their environments, statements of correctness conditions for asynchronous algorithms (including safety, progress, and fairness conditions), proof techniques for asynchronous algorithms (including operational, invariant assertion, and simulation relation proofs), ways of defining and analyzing time complexity for asynchronous algorithms, and techniques for proving lower bounds.

10.1 Asynchronous Shared Memory Model

Before we begin describing any algorithms, we describe the computation model we will use in this and the next three chapters. Here, we describe the model briefly and informally; a more complete, formal description appears in Chapter 9.

The system is modelled as a collection of processes and shared variables, with interactions as depicted in Figure 10.1. Each process i is a kind of state machine, with a set $states_i$ of states and a subset $start_i$ of $states_i$ indicating the start states, just as in the synchronous setting. However, now process i also has labelled *actions*, describing the activities in which it participates. These are classified as either *input*, *output*, or *internal* actions. In Figure 10.1, the arrows entering and leaving the process circles represent the input and output actions of the various processes. We further distinguish between two different kinds of internal actions: those that involve the shared memory and those that involve strictly local computation. If an action involves the shared memory, we assume that it only involves one shared variable.

Unlike in the synchronous setting, there is no message-generation function, since there are no messages in this model. All communication between the processes is via the shared memory.

There is a transition relation *trans* for the entire system, which is a set of (s, π, s') triples, where s and s' are *automaton states*, that is, combinations of states for all the processes and values for all the shared variables, and where π is the label of an input, output, or internal action. We call these combinations of process states and variable values "automaton states" because, in the formal model of Chapter 9, the entire system is modelled as a single automaton. The statement that $(s, \pi, s') \in trans$ says that from automaton state s it is possible to

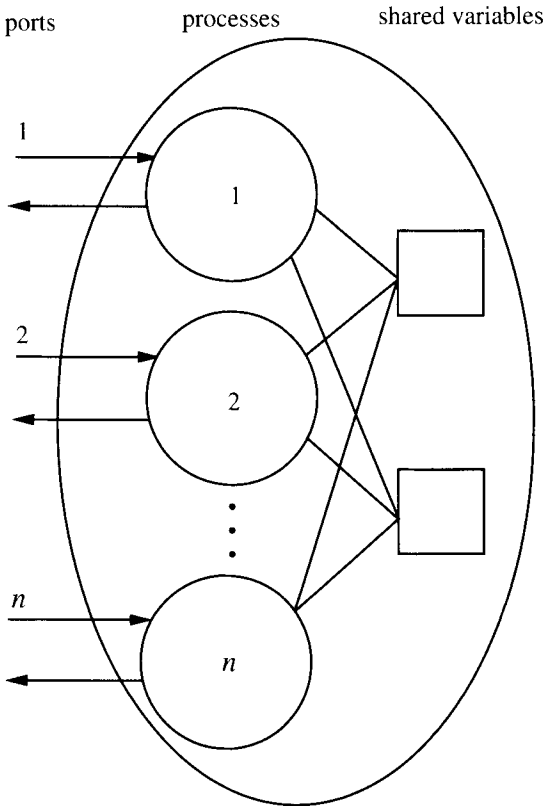


Figure 10.1: An asynchronous shared memory system.

go to automaton state s' as a result of performing action π. Note that *trans* is a *relation* rather than a function—for convenience, we allow our model to include nondeterminism.

We assume that input actions can always happen, that is, that the system is *input-enabled*. Formally, this means that for every automaton state s and input action π, there exists s' such that $(s, \pi, s') \in trans$. In contrast, output and internal steps might be enabled only in a subset of the states. The intuition behind the input-enabling property is that the input actions are controlled by an arbitrary external user, while the internal and output actions are controlled by the system itself.

The set of transitions has some "locality" restrictions. First, for any transition that does not involve the shared memory, only the state of the process that performs the action can be involved. On the other hand, for a transition that involves a process i and a shared variable x, only the state of process i and the

value of variable x can be involved. We assume that the *enabling* of a shared memory action depends only on the process state and not on the value of the shared variable accessed. However, the resulting changes to the process state and the variable value may depend also on the variable value.

The shared variable steps are usually constrained further, to be executions of operations of particular types, such as *read* and *write*. A *read* step for variable x involves changing the process state, based on its previous state and the value in x; however, the value of variable x does not change. A *write* step involves writing a designated value to a shared variable, overwriting whatever was there before; it may also change the process state. We will mostly consider the model in which the variables are accessed using *read* and *write* operations, but we will also consider some more powerful operations such as read-modify-write.

The execution of an asynchronous shared memory system is very different from that of a synchronous system. This time, processes are assumed to take steps one at a time, in an *arbitrary* order rather than in synchronized rounds. This arbitrary order is the essence of the asynchronous model. An execution is formalized as an alternating sequence, $s_0, \pi_1, s_1, \ldots$, consisting of automaton states alternated with actions (each action belonging to a particular process), where successive (state, action, state) triples satisfy the transition relation. An execution may be a finite or an infinite sequence.

There is one important exception to the arbitrariness in the order of process steps. We do not want to allow a process to stop taking steps when it is supposed to be taking steps, that is, when the process is in a state in which some *locally controlled* action (i.e., a non-input action) is enabled. (Although input actions are always enabled, we do not assume that they ever occur.) This condition is a little tricky to state precisely.

For example, we might try to express it by saying: "If a process takes only finitely many steps, then its final state is one in which no locally controlled action is enabled." But this is not quite sufficient—we might want also to rule out some situations in which a process takes infinitely many steps, but after some point, all the remaining steps are input steps. We need to make sure that the process itself also gets turns to perform locally controlled actions.

So, we might try to express the needed condition by saying: "If a process takes only finitely many steps, then its final state is one in which no locally controlled action is enabled, and if a process takes infinitely many steps, then infinitely many of these steps are locally controlled steps." But again this is not quite right—consider the situation in which the process receives infinitely many inputs and performs no locally controlled actions, but in fact no locally controlled actions are enabled. That situation seems fine, since we could say that

the process had "turns" to perform locally controlled steps, but simply had none that it "wanted" to perform.

We account for all these possibilities in the following definition. For each process i, we assume that one of the following holds:

1. The entire execution is finite, and in the final state no locally controlled action of process i is enabled.
2. The execution is infinite, and there are either infinitely many occurrences of locally controlled actions of i, or else infinitely many places where no such action is enabled.

We call this condition the *fairness condition* for this shared memory system. (In terms of the I/O automaton definitions in Chapter 8, this amounts to grouping all the locally controlled actions of one process into one task.)

10.2 The Problem

The mutual exclusion problem involves the allocation of a single, indivisible, nonshareable resource among n *users*, $U_1, \ldots, U_n$. The users can be thought of as application programs. The resource could be, for example, a printer or other output device that requires exclusive access in order to ensure that the output is sensible. Or it could be a database or other data structure that requires exclusive access in order to avoid interference among the operations of different users.

A user with access to the resource is modelled as being in a *critical region*, which is simply a designated subset of its states. When a user is not involved in any way with the resource, it is said to be in the *remainder region*. In order to gain admittance to its critical region, a user executes a *trying protocol*, and after it is done with the resource, it executes an (often trivial) *exit protocol*. This procedure can be repeated, so that each user follows a cycle, moving from its *remainder region* (R) to its *trying region* (T), then to its *critical region* (C), then to its *exit region* (E), and then back again to its remainder region. This cycle is shown in Figure 10.2.

We consider mutual exclusion algorithms within the shared memory model described above—see Figure 10.1 for the architecture. The shared memory system contains n processes, numbered $1, \ldots, n$, each corresponding to one user U_i. The inputs to process i are the try_i action, which models a request by user U_i for access to the resource, and the $exit_i$ action, which models an announcement by user U_i that it is done with the resource. The outputs of process i are $crit_i$, which models the granting of the resource to U_i, and rem_i, which tells U_i that it can continue with the rest of its work. The *try*, *crit*, *exit*, and *rem* actions

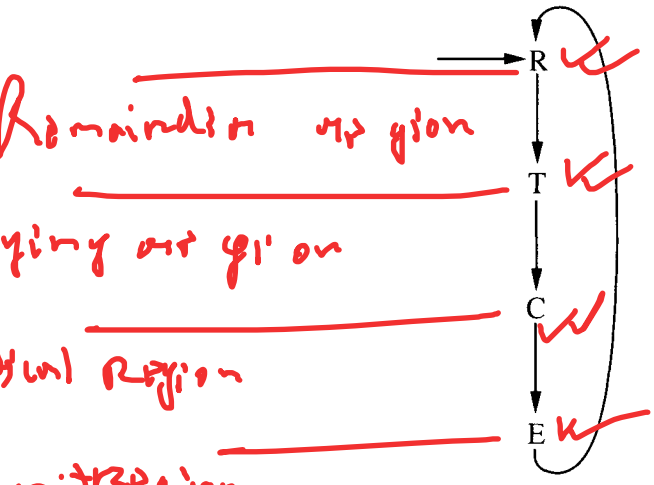


Figure 10.2: The cycle of regions of a single user

are the only external actions of the shared memory system. The processes are responsible for performing the trying and exit protocols. Each process i acts as an "agent" on behalf of user U_i.

Each of the users U_i, $1 \leq i \leq n$, is modelled as a state machine (formally, an *I/O automaton*) that communicates with its agent process using the try_i, $crit_i$, $exit_i$, and rem_i actions. The external interface (formally, the *external signature*) of U_i is depicted in Figure 10.3.

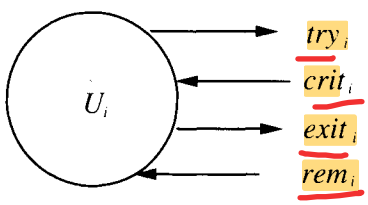


Figure 10.3: External interface of user U_i

We think of each user U_i as executing some application program. The only thing that we assume about U_i is that it obeys the cyclic region protocol, that is, that U_i is not the first to violate the cyclic order of actions, $try_i, crit_i, exit_i, \ldots$ (starting with try_i), between itself and its agent process. Formally, we define a sequence of try_i, $crit_i$, $exit_i$ and rem_i actions to be *well-formed* for user i if it is a prefix of the cyclically ordered sequence $try_i, crit_i, exit_i, rem_i, try_i, \ldots$. Then we require that U_i *preserve* the *trace property* defined by the set of sequences

that are well-formed for user i. (We use the definitions of *trace property* and *preserves* from Section 8.5.4.)

In executions of U_i that do observe the cyclic order of actions, we say that U_i is

- in its *remainder region* initially and in between any rem_i event and the following try_i event.
- in its *trying region* in between any try_i event and the following $crit_i$ event.
- in its *critical region* in between any $crit_i$ event and the following $exit_i$ event. During this time, U_i should be thought of as being free to use the resource (although we do not model the resource explicitly).
- in its *exit region* in between any $exit_i$ event and the following rem_i event.

Figure 10.4 depicts all the interactions in the system.

Now we can state what it means for a shared memory system A to *solve the mutual exclusion problem* for a given collection of users. Namely, the combination (formally, the composition) of A and the users must satisfy the following conditions:

System obeys cyclic discipline

Well-formedness: In any execution, and for any i, the subsequence describing the interaction between U_i and A is well-formed for i.

Mutual exclusion: There is no reachable system state (that is, a combination of an automaton state for A and states for all the U_i) in which more than one user is in the critical region C.

Progress: At any point in a *fair execution*

1. (Progress for the trying region) If at least one user is in T and no user is in C, then at some later point some user enters C.
2. (Progress for the exit region) If at least one user is in E, then at some later point some user enters R.

We say that a shared memory system A *solves the mutual exclusion problem* provided that it solves it for every collection of users.

Note that we have stated the correctness conditions in terms of the users' regions. Normally, the process states will also be classified according to their regions, and these regions will correspond exactly to the user regions. So we can equivalently state the correctness conditions in terms of process regions. We will talk interchangeably about user regions and process regions in the rest of this chapter.

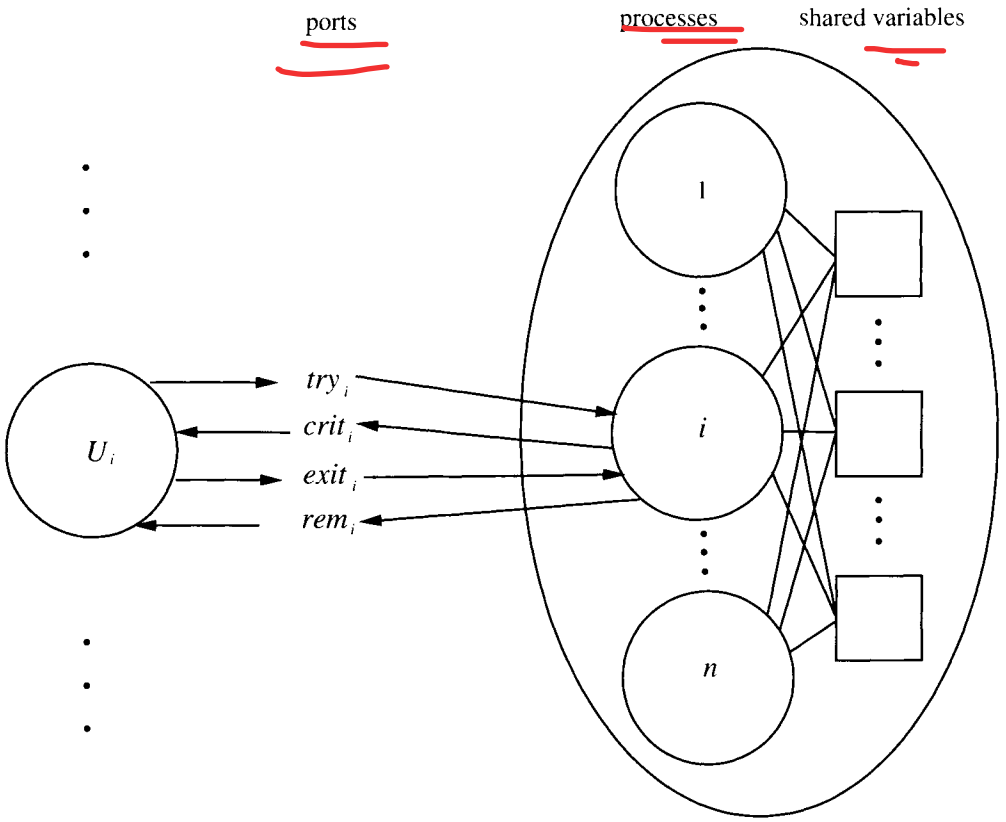


Figure 10.4: Interactions between components for the mutual exclusion problem.

Note that the progress condition assumes that the execution of the system is fair, that is, it assumes that all the processes (and users) continue taking steps. If we did not assume this, then it would not be reasonable to require that *crit* or *rem* outputs eventually be performed. On the other hand, we do not need to assume fairness in order to require that the system guarantee well-formedness or mutual exclusion. The difference is that the well-formedness and mutual exclusion conditions are *safety properties* (properties that say that particular "bad" things never happen), while the progress condition is a *liveness property* (a property that says that some "good" thing eventually happens).

Trace properties. Still another equivalent way of presenting these correctness conditions is in terms of a *trace property*, as defined in Section 8.5.2. For example, we can define a trace property P, where $sig(P)$ has all the *try*, *crit*, *exit*, and

rem actions as outputs, and $traces(P)$ is the set of sequences β of these actions that satisfy the following three conditions:

1. β is well-formed for each i.
2. β does not contain two *crit* events without an intervening *exit* event.
3. At any point in β,
   (a) If some process's last event is *try* and no process's last event is *crit*, then there is a later *crit* event.
   (b) If some process's last event is *exit*, then there is a later *rem* event.

Then an equivalent restatement of the mutual exclusion problem is the requirement that, for all combinations B of A with users, $fairtraces(B) \subseteq traces(P)$. (Recall that the external actions of B are just the *try*, *crit*, *exit*, and *rem* actions.) Trace property P could also be split into two parts, a safety property encompassing the well-formedness and mutual exclusion conditions, and a liveness property for the progress condition.

Well-formedness and mutual exclusion are safety properties, don't depend on fairness

Shared responsibility for progress. According to the correctness conditions we have given, responsibility for the continuing progress of the entire system rests not only with the protocol, but with the users as well. If a user U_i gets the resource (by means of a $crit_i$ event) but never returns it (by means of an $exit_i$ event), then the entire system grinds to a halt. But if each user eventually returns the resource every time it receives it, then the progress condition implies that the entire system continues to make progress, repeatedly moving processes to new regions (unless all users remain in their remainder regions from some point on).

Lockout. The progress condition we have stated does not imply that any particular requesting user ever succeeds in reaching its critical region. Rather, it is a "global" notion of progress, saying only that *some* user reaches its critical region. For instance, the following scenario does not violate the progress condition: Starting from an initial state, user U_1 enters T. Then user U_2 cycles through its four regions infinitely many times, while U_1 remains in T and the rest of the processes remain in R. Our progress condition does not guarantee that U_1 ever reaches C.

Restricting process activity. There is one other constraint—a technical one—that we assume in this chapter: that a process within the shared memory system can have a locally controlled action enabled only when its user is in

the trying or exit region. This says that a process can be actively engaged in executing the protocol only while it has active requests. This assumption is consistent with the view that each process is simply an agent for its corresponding user.

In practical settings, this assumption might or might not be reasonable. The mutual exclusion problem was first studied in the setting of a time-shared uniprocessor, where the users are logically independent processes sharing a single processor. In this setting, allowing a permanent process to manage access to the resource would cause extra context-switching, between the manager process and the user processes. In a true multiprocessor environment, it is possible to avoid the context-switching by using a dedicated processor to manage the resource. However, there will generally be many resources to be managed, and all the processors dedicated to managing resources would be unavailable for participation in other computational tasks.

Read/write shared variables. For most of the chapter (except for Section 10.9), we assume that the shared variables are read/write variables, also known as *registers*. In one step, a process can either read or write a single shared variable, but not both. Thus, the two actions involving process i and register x are

1. (*read*) Process i reads register x and uses the value read to modify the state of process i.
2. (*write*) Process i writes a value determined from process i's state to register x.

We finish this section with a simple lemma saying that processes cannot stop taking steps while they are in their trying or exit regions.

Lemma 10.1 *Let A be an algorithm that solves the mutual exclusion problem (for all collections of users). Let $U_1, \ldots, U_n$ be any particular collection of users, and let B be the combination of A and the given collection of users. Let s be a reachable state of B.*

If process i is in its trying or exit region in state s, then some locally controlled action of process i is enabled in s.

Proof. Without loss of generality, we may assume that each of the users, $U_1, \ldots, U_n$, always returns the resource.

Let α be a finite execution of B ending in s, and suppose for the sake of contradiction that process i is in either its trying or exit region in state s, and

no locally controlled action of process i is enabled in s. Then we claim that no events involving i occur in any execution of B that extends α, after the prefix α. This follows from the fact that enabling of locally controlled actions is determined only by the local process state, plus the fact that well-formedness prevents inputs to process i while process i is in T or E.

Now let α' be a fair execution of B that extends α, in which no *try* events occur after the prefix α. Repeated use of the progress assumption, plus the fact that the users always return the resource, imply that process i must eventually perform either a $crit_i$ or a rem_i action. But this contradicts the fact that α' contains no further actions of i. □

10.3 Dijkstra's Mutual Exclusion Algorithm

The first mutual exclusion algorithm for the asynchronous read/write shared memory model was developed in 1965 by Edsger Dijkstra, based on a prior two-process solution by Dekker. This algorithm is not the most elegant or efficient algorithm now available, nor does it satisfy the strongest conditions. However, we present it anyway, for several reasons. First, it is the earliest example we can find of an algorithm that we would categorize as "distributed." Second, it contains several interesting algorithmic ideas. And third, it is a good example to use for illustrating some of the basic reasoning techniques for asynchronous shared memory algorithms.

10.3.1 The Algorithm

We begin by presenting code for the algorithm in a traditional "pseudocode" style, similar to that used in the original paper by Dijkstra. Although this code should make sense informally, it is probably not completely clear how it should be translated into an instance of our model. We call the algorithm *DijkstraME*.

***DijkstraME* algorithm:**

Shared variables:
$turn \in \{1, \ldots, n\}$, initially arbitrary, writable and readable by all processes
for every i, $1 \leq i \leq n$:
$flag(i) \in \{0, 1, 2\}$, initially 0, writable by process i and readable by all processes

Process i:

```
        ** Remainder region **

        try_i
L:      flag(i) := 1
        while turn ≠ i do
            if flag(turn) = 0 then turn := i
        flag(i) := 2
        for j ≠ i do
            if flag(j) = 2 then goto L
        crit_i

        ** Critical region **

        exit_i
        flag(i) := 0
        rem_i
```

The shared variables are *turn*, an integer in $\{1, \ldots, n\}$, and $flag(i), 1 \leq i \leq n$, one per process, each taking on values from $\{0, 1, 2\}$, initially 0. The *turn* variable is a *multi-writer/multi-reader* register, writable and readable by all processes. Each $flag(i)$ is a *single-writer/multi-reader* register, writable only by process i but readable by all processes.

In process i's first stage, it starts by setting its *flag* to 1 and then repeatedly checks the *turn* variable to see if $turn = i$. If not, and if the current owner of *turn* is seen not to be currently active, process i sets $turn := i$. Once having seen $turn = i$, process i moves on to the second stage.

In the second stage, process i again sets its *flag*, this time to 2, and then checks to see that no other process has its $flag = 2$. This check of other processes' *flags* can be done in any order. If the check completes successfully, process i goes to its critical region; otherwise, it returns to the first stage. Upon leaving the critical region, process i lowers its *flag* back to 0.

Before we can prove anything about *DijkstraME*, we need to understand it as an instance of our formal state machine model. It is not completely obvious how to translate the code into an automaton.

First, the state of each process should consist of the values of its local variables, as you would expect, plus some other information that is not represented explicitly in the code, including

- temporary variables needed to remember values just read from shared variables

- a program counter, to say where the process is in its code
- temporary variables introduced by the flow of control of the program (e.g., the for loop can introduce a set variable to keep track of the indices of processes that have already been checked successfully)
- a region designation, R, T, C, or E (R indicates the remainder region, T indicates the portion of the code from a try_i event until the next $crit_i$ event, C indicates the critical region, and E indicates the portion of the code from an $exit_i$ event until the next rem_i event)

The unique start state of each process should consist of specified initial values for local variables, arbitrary values for temporary variables, and the program counter and the region designation indicating the remainder region. The initial value for each shared variable is as specified.

The steps of the automaton should follow the code; however, there are some ambiguities in the code that need to be resolved in the automaton. Although the code describes the changes to the local and shared variables, it does not say explicitly what happens to the implicit variables (the temporaries, program counter, and region designation). For example, when a try_i action occurs, i's program counter should move to statement L in the code and i's region designation should become T. These changes must be described explicitly in the automaton.

The code also does not specify exactly which portions of the code comprise indivisible steps. However, it is essential to know this in order to reason carefully about the algorithm. For *DijkstraME*, the indivisible steps are the *try*, *crit*, *exit*, and *rem* steps at the user interface, plus individual writes to and reads from the shared variables, plus some local computation steps. There is at least one minor subtlety: the test for whether $flag(turn) = 0$ does not require two separate reads—since *turn* was just read in the previous line, a local copy of *turn* can be used.

We resolve all of these ambiguities by rewriting the *DijkstraME* code by hand, in the *precondition-effect* style used in Chapter 8. Rewriting in this way makes the code a good deal longer, but all the transitions are now described explicitly. For readability, we arrange the pieces of code for the different actions in approximately the order in which they are supposed to be executed; however, note that this order has no significance in the formal model—any action is allowed to occur at any time when it is enabled. The region designations R, T, C, and E are encoded into program counter values: R corresponds to *rem*; T corresponds to *set-flag-1*, *test-turn*, *test-flag*, *set-turn*, *set-flag-2*, *check*, and *leave-try*; C corresponds to *crit*; and E corresponds to *reset* and *leave-exit*. Note that each code fragment is performed indivisibly.

DijkstraME algorithm (rewritten):

Shared variables:
$turn \in \{1, \ldots, n\}$, initially arbitrary
for every i, $1 \leq i \leq n$:
$flag(i) \in \{0, 1, 2\}$, initially 0

Actions of i:

Input:	Internal:
try_i	$set\text{-}flag\text{-}1_i$
$exit_i$	$test\text{-}turn_i$
Output:	$test\text{-}flag(j)_i$, $1 \leq j \leq n$, $j \neq i$
$crit_i$	$set\text{-}turn_i$
rem_i	$set\text{-}flag\text{-}2_i$
	$check(j)_i$, $1 \leq j \leq n$, $j \neq i$
	$reset_i$

States of i:
$pc \in \{rem, set\text{-}flag\text{-}1, test\text{-}turn, test\text{-}flag(j), set\text{-}turn, set\text{-}flag\text{-}2, check, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially rem
S, a set of process indices, initially $\emptyset$

Transitions of i:

try_i
Effect:
$pc := set\text{-}flag\text{-}1$

$set\text{-}flag\text{-}1_i$
Precondition:
$pc = set\text{-}flag\text{-}1$
Effect:
$flag(i) := 1$
$pc := test\text{-}turn$

$test\text{-}turn_i$
Precondition:
$pc = test\text{-}turn$
Effect:
if $turn = i$ then $pc := set\text{-}flag\text{-}2$
else $pc := test\text{-}flag(turn)$

$test\text{-}flag(j)_i$
Precondition:
$pc = test\text{-}flag(j)$
Effect:
if $flag(j) = 0$ then $pc := set\text{-}turn$
else $pc := test\text{-}turn$

$set\text{-}turn_i$
Precondition:
$pc = set\text{-}turn$
Effect:
$turn := i$
$pc := set\text{-}flag\text{-}2$

$set\text{-}flag\text{-}2_i$
Precondition:
$pc = set\text{-}flag\text{-}2$
Effect:
$flag(i) := 2$
$S := \{i\}$
$pc := check$

$check(j)_i$
Precondition:
$pc = check$
$j \notin S$
Effect:
if $flag(j) = 2$ then
$S := \emptyset$
$pc := set\text{-}flag\text{-}1$
else
$S := S \cup \{j\}$
if $|S| = n$ then $pc := leave\text{-}try$

$crit_i$
Precondition:
$pc = leave\text{-}try$
Effect:
$pc := crit$

$exit_i$
Effect:
$pc := reset$

$reset_i$
Precondition:
$pc = reset$
Effect:
$flag(i) := 0$
$S := \emptyset$
$pc := leave\text{-}exit$

rem_i
Precondition:
$pc = leave\text{-}exit$
Effect:
$pc := rem$

The translation should be mainly self-explanatory. Note that the new style makes it easy to express slight improvements; for instance, the $set\text{-}turn_i$ action can allow process i to go directly to the second stage, without retesting $turn$.

10.3.2 A Correctness Argument

In this section we sketch a correctness proof for *DijkstraME*. This will be a somewhat brute-force *operational* proof, that is, one that consists of ad hoc arguments about executions. In the following section, Section 10.3.3, we give an alternative, more stylized proof of the mutual exclusion condition using invariant assertions.

We give three lemmas showing that *DijkstraME* satisfies its requirements.

Lemma 10.2 *DijkstraME guarantees well-formedness for each user.*

More precisely, we mean that in any execution of the combination (composition) of *DijkstraME* and any collection of users, the subsequence describing the interaction between any U_i and *DijkstraME* is well-formed for user i.

Proof. By inspection of the code, it is easy to check that *DijkstraME* preserves well-formedness for each user. Since, by assumption, the users also preserve well-formedness, Theorem 8.11 implies that the system produces only well-formed sequences. □

Lemma 10.3 *DijkstraME satisfies mutual exclusion.*

More precisely, we mean that in the combination of *DijkstraME* and any collection of users, there is no reachable state in which more than one user is in the critical region C.

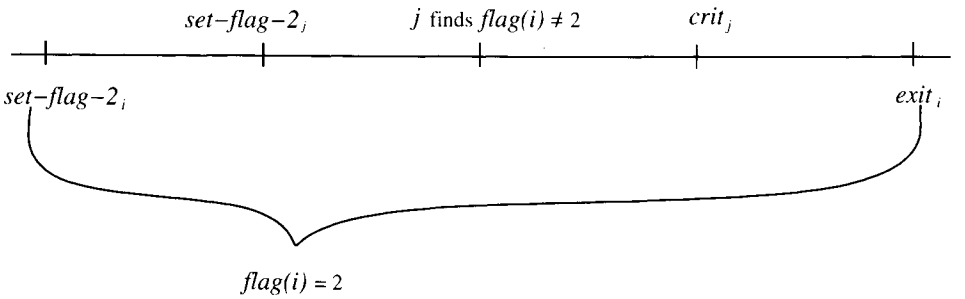


Figure 10.5: Order of events in the proof of Lemma 10.3.

Proof. By contradiction. Assume that U_i and U_j, $i \neq j$, are simultaneously in region C in some reachable state. Consider an execution that leads to this state. By the code, both process i and process j perform *set-flag-2* steps before entering their critical regions. Consider the last such step for each process and assume, without loss of generality, that *set-flag-2*$_i$ comes first. Then $flag(i)$ remains equal to 2 from that point until process i leaves C, which must be after process j enters C, by the assumption that they both end up in C simultaneously. So, $flag(i)$ has the value 2 throughout the interval from the *set-flag-2*$_j$ event until process j enters C. See Figure 10.5. But, during this time, process j must test $flag(i)$ and find it unequal to 2, a contradiction. □

Lemma 10.4 *DijkstraME guarantees progress.*

Proof. The argument for the exit region is easy: If at any point in a fair execution, U_i is in the exit region, then process i keeps taking steps. After at most two more of these steps, process i will perform a *rem*$_i$ action, sending U_i to its remainder region.

We consider the progress condition for the trying region. Suppose for the sake of contradiction that α is a fair execution that reaches a point where there is at least one user in T and no user in C, and suppose that after this point, no user ever enters C.

We begin by removing some complications. First, any process in E keeps taking steps, so after at most two steps, it must reach R. So, after some point in α, every process must be in T or R. Second, since there are only finitely many processes in the system, after some point in α, no new processes enter T. Thus, after some point in α, every process is in T or R, and no process ever again changes region. This implies that α has a suffix α_1 in which there is a fixed nonempty set of processes in T, continuing to take steps forever, and no region changes occur. Call these processes *contenders*.

Note that after at most a single step in α_1, each contender i ensures that $flag(i) \geq 1$, and it remains ≥ 1 for the rest of α_1. So we can assume, without loss of generality, that $flag(i) \geq 1$ for all contenders i throughout α_1.

Clearly, if *turn* is modified during α_1, it is changed to a contender's index. Moreover, we have the following claim.

Claim 10.5 *In α_1, turn eventually acquires a contender's index.*

Proof. Suppose not, that is, suppose the value of *turn* remains equal to the index of a non-contender throughout α_1. Consider any contender i.

If pc_i ever reaches *test-turn* (i.e., the beginning of the while loop in the original code), then we claim that i will set *turn* to i. This is because i first performs a *test-turn*$_i$ and finds that *turn* equal to some $j \neq i$. Then it performs a *test-flag*$(j)_i$ and finds $flag(j) = 0$, since j is not a contender. Process i therefore performs *set-turn*$_i$, setting *turn* to i.

Now we show that i reaches *test-turn*. The only way it might not is if i succeeds in its *check*s of all the other processes' *flag*s (in the second stage of the original code) and proceeds to *leave-try*. But by assumption about α_1, we know that i does not reach C. So it must be that some *check* must fail, taking i back to *set-flag-1*, from which it proceeds to *test-turn*.

So, i reaches *test-turn* and thereafter sets $turn := i$. Since i is a contender, this is the needed contradiction. □

Once *turn* is set to a contender's index, it is always thereafter equal to *some* contender's index, although the value of *turn* may change to the index of different contenders. (This is because it is possible for several processes to be simultaneously at *set-turn*.) Then any later *test-turn* and subsequent *test-flag* yield $flag(turn) \geq 1$, since for all contenders i, $flag(i) \geq 1$. Thus, *turn* will not be changed as a result of these tests. Therefore, eventually *turn* stabilizes to a final (contender's) index. Let α_2 be a suffix of α_1 in which the value of *turn* is stabilized at some contender's index, say i.

Next we claim that in α_2, any contender $j \neq i$ eventually ends up with its program counter looping forever between *test-turn* and *test-flag*. (That is, it winds up looping forever in the while loop.) This is because if it ever reaches *check* (in the second stage), then, since it doesn't reach C, it must eventually return to *set-flag-1*. But then it is stuck looping forever, because $turn = i \neq j$ and $flag(i) \neq 0$ throughout α_2. So let α_3 be a suffix of α_2 in which all contenders other than i loop forever between *test-turn* and *test-flag*. Note that this means that all contenders other than i have their *flag* variables equal to 1 throughout α_3.

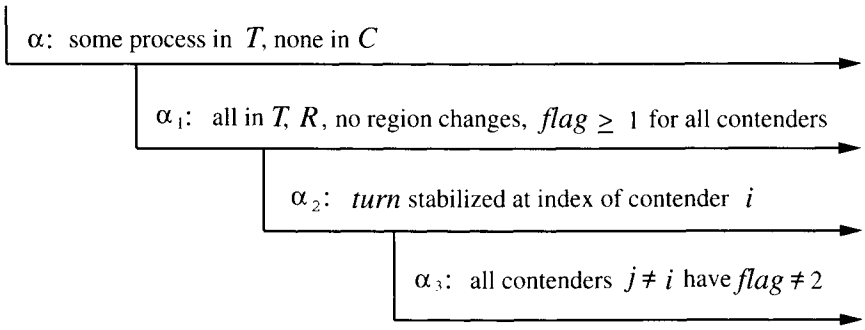


Figure 10.6: Successive suffixes in proof of Lemma 10.4.

We conclude the argument by claiming that in α_3, process i (the one whose index is in *turn*) has nothing to stand in the way of its reaching C. For example, if i performs *test-turn*, then i finds $turn = i$ and so proceeds to *set-flag-2*. Then, since no other process has $flag = 2$, process i succeeds in all its *check*s and enters C.

See Figure 10.6 for a depiction of the successive suffixes that appear in this proof.

Theorem 10.6 *DijkstraME solves the mutual exclusion problem.*

Although the arguments above are correct, they are rather intricate and ad hoc. It would be nice to have some more systematic ways of carrying out such proofs. In the following section, we give an alternative proof of the mutual exclusion condition, using invariant assertions. The progress condition could also be proved somewhat more systematically using temporal logic, but we do not do that in this book.

10.3.3 An Assertional Proof of the Mutual Exclusion Condition

In the synchronous network model, many of the neatest and most systematic proofs are based on invariant assertions about the state of the system after some number of rounds. In the asynchronous setting, there is no notion of round, but invariants can still be used. The method just has to be applied at a finer granularity, to verify claims about the system state after any number of individual process steps. Of course, it is usually harder to devise statements about the state of an asynchronous system after any number of steps than it is to devise statements about the state of a synchronous system after any number of rounds. And proving such statements is also usually more difficult. But the effort is

generally worthwhile because of the insights that the invariants provide. Invariant assertions are the single most important formal tool for reasoning about the correctness of asynchronous algorithms.

We now give an assertional proof of the mutual exclusion condition for the *DijkstraME* algorithm.

Proof (of Lemma 10.3). To prove mutual exclusion, we must show

Assertion 10.3.1 *In any reachable system state,*[1] $|\{i : pc_i = crit\}| \leq 1$.

We would like to prove this assertion by induction on the number of steps in an execution. But, as usual, the given statement is not strong enough to prove alone in this way—we need some auxiliary invariants. We prove Assertion 10.3.1 as a consequence of the next two assertions.

Assertion 10.3.2 *In any reachable system state, if* $pc_i \in \{leave\text{-}try, crit, reset\}$, *then* $|S_i| = n$.

Assertion 10.3.3 *In any reachable system state, there do not exist* i *and* j, $i \neq j$, *such that* $i \in S_j$ *and* $j \in S_i$.

If both Assertions 10.3.2 and 10.3.3 are true, then Assertion 10.3.1 follows immediately: Assume, for contradiction, that in some reachable system state, there are two distinct processes, i and j, such that $pc_i = pc_j = crit$. Then by Assertion 10.3.2, $|S_i| = |S_j| = n$. But then $j \in S_i$ and $i \in S_j$, contradicting Assertion 10.3.3.

Assertion 10.3.2 can be proved easily by induction on the length of an execution. The basis is true vacuously, since all the processes are in R in the initial system state. The inductive step is a case analysis, considering all the types of actions one at a time. In this case, the only steps that could cause a violation are those that cause pc_i to enter the set of listed values and those that reset S_i to $\emptyset$, namely, $check_i$ and $reset_i$. In the case of a $check_i$, the only way the condition $pc_i \in \{leave\text{-}try, crit, reset\}$ could be true after the step is if $|S_i| = n$, which is just what we need. In the case of a $reset_i$, the process leaves the indicated set of values after the step, so the statement is true vacuously.

So it remains to prove Assertion 10.3.3. This uses two simple facts. The first one constrains where process i can be in its code when $S_i \neq \emptyset$.

Assertion 10.3.4 *In any reachable system state, if* $S_i \neq \emptyset$, *then* $pc_i \in \{check, leave\text{-}try, crit, reset\}$.

[1] Recall that a system state is a combination of states of the users and processes plus values of the shared variables.

This is also proved by a simple induction on the length of an execution. The basis is easy, since $S_i = \emptyset$ in the initial system state. For the inductive step, the only events that could cause a violation of this statement are events that cause S_i to become unequal to $\emptyset$ and events that cause pc_i to leave the set of listed values, that is, *set-flag-2*$_i$, *check*$_i$, and *reset*$_i$. But *set-flag-2*$_i$ sets $pc_i :=$ *check*. Also, when *check*$_i$ causes pc_i to leave the set of listed values, it also sets $S_i := \emptyset$. Finally, *reset* sets $S_i := \emptyset$. Thus, all these events preserve the condition.

The second fact says that $flag(i) = 2$ when process i is at certain points in its code.

Assertion 10.3.5 *In any reachable system state, if* $pc_i \in \{$*check, leave-try, crit, reset*$\}$, *then* $flag(i) = 2$.

This is also proved by an easy induction on the length of an execution. Putting these two facts together, we see the following:

Assertion 10.3.6 *In any reachable system state, if* $S_i \neq \emptyset$, *then* $flag(i) = 2$.

Now we can prove Assertion 10.3.3, again by induction on the length of an execution. The basis is easy because in the initial state, all sets S_i are empty. For the inductive step, the only event that could cause a violation is one that adds an element j to S_i for some i and j, $i \neq j$, that is, a *check*$(j)_i$ for some i and j, $i \neq j$. So consider the case where j gets added to S_i as a result of a *check*$(j)_i$ event. Then it must be that $flag(j) \neq 2$ when this event occurs. But then Assertion 10.3.6 implies that $S_j = \emptyset$, so $i \notin S_j$. Thus, this step cannot cause a violation. □

10.3.4 Running Time

In this section, we prove an upper bound on the time from any point in an execution when some process is in T and no one is in C, until someone enters C.

The first difficulty we face in proving such a bound is that it is not clear what this "time" should mean—unlike in the synchronous setting, there are no rounds to count. Instead, we just assume that each step occurs at some point in real time and that the execution begins at real time 0. We impose an upper bound of ℓ on the time between successive steps of each process (when these steps are enabled); recall that all the precondition-effect code for one action is assumed to comprise a single step. We also assume an upper bound of c on the maximum time that any user spends in the critical region. In terms of these assumed bounds, we can deduce upper bounds for the time required for interesting activity to occur.

Theorem 10.7 *In DijkstraME, suppose that at a particular time some user is in T and no user is in C. Then within time $O(\ell n)$, some user enters C.*

The constant involved in the big-O is independent of ℓ, c, and n. This proof is ad hoc and a little tricky, using ideas from the proof of the progress condition.

Proof. Suppose the lemma is false and consider an execution in which, at some point, process i is in T and no process is in C, and in which no process enters C for time at least $k\ell n$, for some particular large constant k. Constant k is chosen to be considerably bigger than the constants in the big-O terms in the following analysis.

First, it is easy to see that the time elapsed from the starting point of the analysis until there is no process either in C or E is at most $O(\ell)$.

Second, we claim that the additional time until process i performs a *test-turn*$_i$ is at most $O(\ell n)$. This is because i can at worst spend this much time checking flags in the second stage before returning to *set-flag-1*. We know that it must return to *set-flag-1*, because otherwise it would go to C, which we have assumed does not happen this quickly.

Third, we claim that the additional time from when process i does *test-turn*$_i$ until the value of *turn* is a contender index is at most $O(\ell)$. To see this, we need a rather annoying case analysis. If at the time i does *test-turn*$_i$, *turn* already holds a contender index, then we are done, so suppose that this is not the case; specifically, suppose that $turn = j$, where j is not a contender. Then within time $O(\ell)$ after this test, i performs a *test-flag*$(j)_i$. If process i finds $flag(j) = 0$, then i sets *turn* to i, which is the index of a contender, and we are again done. But if it finds $flag(j) \neq 0$, then it must be that in between the *test-turn*$_i$ and the *test-flag*$(j)_i$, process j entered the trying region and became a contender. If *turn* has not changed in the interim, then *turn* is equal to the index of a contender (j) and we are done. But if *turn* has changed in the interim, then it must have been set to the index of a contender. So again, we are done.

Fourth, after an additional time $O(\ell)$, a point is reached at which the value of *turn* has stabilized to the index of some particular contender, say j, and furthermore no process advances again to *set-turn* or *set-flag-2* (at least until time $k\ell n$ after the starting point of the analysis).

Fifth, we claim that by an additional time $O(\ell n)$, all contenders other than j will have their program counters in {*test-turn*, *test-flag*}. This is because otherwise they would reach C, which we have assumed does not happen this quickly.

Sixth and finally, within an additional time $O(\ell n)$, j must succeed in entering C. This contradicts the assumption that no process enters C within this amount of time.

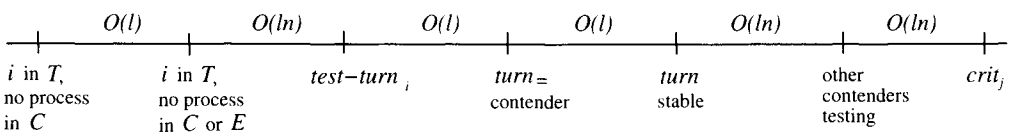


Figure 10.7: Order of events and time bounds in proof of Theorem 10.7.

The order of events in this proof, and the time bounds between them, are depicted in Figure 10.7. □

10.4 Stronger Conditions for Mutual Exclusion Algorithms

Although the *DijkstraME* algorithm guarantees mutual exclusion and progress, there are other desirable conditions that it does not guarantee. It does not guarantee that the critical region is granted *fairly* to different users; for example, it allows one user to be repeatedly granted access to its critical region while other users trying to gain access are forever prevented from doing so. This situation is sometimes called *lockout* or *starvation.*

Note that the kind of fairness we are talking about here is different from that discussed up to this point. So far, we have been talking about fair execution of process steps (and user automata steps), whereas now we are talking about fair granting of the resource. In order to distinguish these two types of fairness, we will call the fair execution of process steps and user automata steps *low-level fairness*, and the fair granting of the resource *high-level fairness.* In practice, high-level fairness might not be critical; in many practical situations in which mutual exclusion is used, contention between users is sufficiently infrequent that a user can afford to wait until all conflicting users get their turns. The importance of high-level-fairness considerations depends on the amount of contention for the resource, as well as the criticality of individual user programs.

Another not-so-attractive property of Dijkstra's algorithm is that it uses a shared *multi-writer/multi-reader* register (*turn*). Such a variable is difficult and expensive to implement in many kinds of multiprocessor systems (as well as in nearly all message-passing systems). It would be better to design algorithms that use only *single-writer/multi-reader* registers, or even better, *single-writer/single-reader* registers.

Many mutual exclusion algorithms that improve upon *DijkstraME* in various ways have been designed. In the rest of this chapter, we shall look at a representative collection of these algorithms.

Before proceeding to the algorithms, we define carefully what it means for a mutual exclusion algorithm to guarantee high-level fairness. Depending upon the context in which the algorithm is used, different notions of high-level fairness may be appropriate; we define three notions. Each of these properties is stated for a particular mutual exclusion algorithm A composed with a particular collection $U_1, \ldots, U_n$ of users.

Lockout-freedom: In any low-level-fair execution, the following hold:

1. (Lockout-freedom for the trying region) If all users always return the resource, then any user that reaches T eventually enters C.
2. (Lockout-freedom for the exit region) Any user that reaches E eventually enters R.

Note that the lockout-freedom condition, like the basic well-formedness, mutual exclusion, and progress conditions, can be expressed as a trace property.

Time bound b: In any low-level-fair execution with associated times, the following hold:

1. (Time bound b for the trying region) If each user always returns the resource within time c of when it is granted, and the time between successive steps of each process in T or E is at most ℓ, then any user that reaches T enters C within time b.
2. (Time bound b for the exit region) If the time between successive steps of each process in T or E is at most ℓ, then any user that reaches E enters R within time b.

(Note that the value of b will typically be a function of ℓ and c.)

Number of bypasses a: Consider any interval of an execution starting when a process i has performed a locally controlled step in T, and throughout which it remains in T. During this interval, any other user j, $j \neq i$, can only enter C at most a times.

In the first two cases above, we have stated high-level-fairness conditions for the exit region that are similar to those for the trying region. However, in most algorithms, the exit regions are actually trivial.

We say that algorithm A is *lockout-free* provided that it guarantees lockout-freedom for all collections of users. We extend the other high-level-fairness definitions similarly. There are some simple implications among these fairness conditions:

Theorem 10.8 *Let A be a mutual exclusion algorithm, let $U_1, \ldots, U_n$ be a collection of users, and let B be the composition of A with $U_1, \ldots, U_n$. If B has any finite bypass bound and is lockout-free for the exit region, then B is lockout-free.*

Proof. Consider a low-level-fair execution of B in which all users always return the resource, and suppose that at some point in the execution, i is in T. Assume for the sake of contradiction that i never enters C.

Lemma 10.1 implies that eventually i must perform a locally controlled action in that trying region, if it has not already done so. Repeated use of the progress condition and of the assumption that users always return the resource together imply that infinitely many total region changes occur. But then some process other than i enters C an infinite number of times while i remains in T, which violates the bypass bound. □

Theorem 10.9 *Let A be a mutual exclusion algorithm, let $U_1, \ldots, U_n$ be a collection of users, and let B be the composition of A with $U_1, \ldots, U_n$. If B has any time bound b (for both the trying and the exit region), then B is lockout-free.*

Proof. Consider a low-level-fair execution of B in which all users always return the resource, and suppose that at some point in the execution, i is in T.

Associate times with the events in the execution in any monotone nondecreasing, unbounded way, so that the times for the steps of each process are at most ℓ and the times for all the critical regions are all at most c.

Since the algorithm satisfies the time bound b, i enters C in at most time b, so in particular, i eventually enters C, as needed for lockout-freedom. □

In the following sections, we will look at some protocols that satisfy some of these stronger high-level-fairness conditions.

10.5 Lockout-Free Mutual Exclusion Algorithms

The first improvements that we present are a trio of algorithms developed by Peterson, all of which guarantee lockout-freedom. The first algorithm is for two processes only, but it demonstrates most of the basic ideas. This algorithm is then extended to $n > 2$ processes in two ways: first, by using a version of the two-process algorithm in a series of $n - 1$ competitions, and second, by using a version of the two-process algorithm in a tournament to select a single winner.

10.5.1 A Two-Process Algorithm

We start with the two-process solution, which we call *Peterson2P*. Usually, we name the two processes in a two-process system processes 1 and 2. This time,

for convenience, we count mod 2 and identify 2 with 0, that is, we call the two processes 0 and 1. If $i \in \{0,1\}$, then we write $\bar{i}$ to indicate $1-i$, the index of the other process. The code, in a traditional style, is given below.

***Peterson2P* algorithm:**

Shared variables:
turn $\in \{0,1\}$, initially arbitrary, writable and readable by all processes
for every $i \in \{0,1\}$:
  flag$(i) \in \{0,1\}$, initially 0, writable by i and readable by $\bar{i}$

Process *i*:

```
** Remainder region **

try_i
flag(i) := 1
turn := i
waitfor flag(ī) = 0 or turn ≠ i
crit_i

** Critical region **

exit_i
flag(i) := 0
rem_i
```

In the *Peterson2P* algorithm, process i starts by setting its *flag* to 1, which is the same as the processes do in *DijkstraME*. But this time, process i immediately proceeds to set *turn* $:= i$. It then waits to discover either that the other process's *flag* is 0, or else that *turn* $\neq i$. That is, either the other process is not currently involved in the competition at all, or else the *turn* variable has been reset by the other process since the most recent time when i set it. Thus (and slightly strangely), having the *turn* variable set to the index of the *other* process gives permission for i to enter its critical region.

How can this program be translated into a state machine in the formal model? As before, we need to introduce a program counter, temporary variables, and a region designation. An ambiguity in the code that needs to be resolved is the order in which process i checks the *flag* and the *turn* variables, in the waitfor statement. For correctness, it is necessary that both checks be done repeatedly; for simplicity, we assume that the checks are done alternately, though looser assumptions would also work.

We rewrite the algorithm in precondition-effect notation, in order to make it easier to carry out a proof. Here, the region designation R corresponds to *rem*; T corresponds to *set-flag*, *set-turn*, *check-flag*, *check-turn*, and *leave-try*; C corresponds to *crit*; and E corresponds to *reset* and *leave-exit*.

Peterson2P algorithm (rewritten):

Shared variables:
$turn \in \{0,1\}$, initially arbitrary
for every $i \in \{0,1\}$:
  $flag(i) \in \{0,1\}$, initially 0

Actions of i:

Input:	Internal:
try_i	$set\text{-}flag_i$
$exit_i$	$set\text{-}turn_i$
Output:	$check\text{-}flag_i$
$crit_i$	$check\text{-}turn_i$
rem_i	$reset_i$

States of i:
$pc \in \{rem, set\text{-}flag, set\text{-}turn, check\text{-}flag, check\text{-}turn, leave\text{-}try, crit, reset, leave\text{-}exit\}$,
  initially *rem*

Transitions of i:

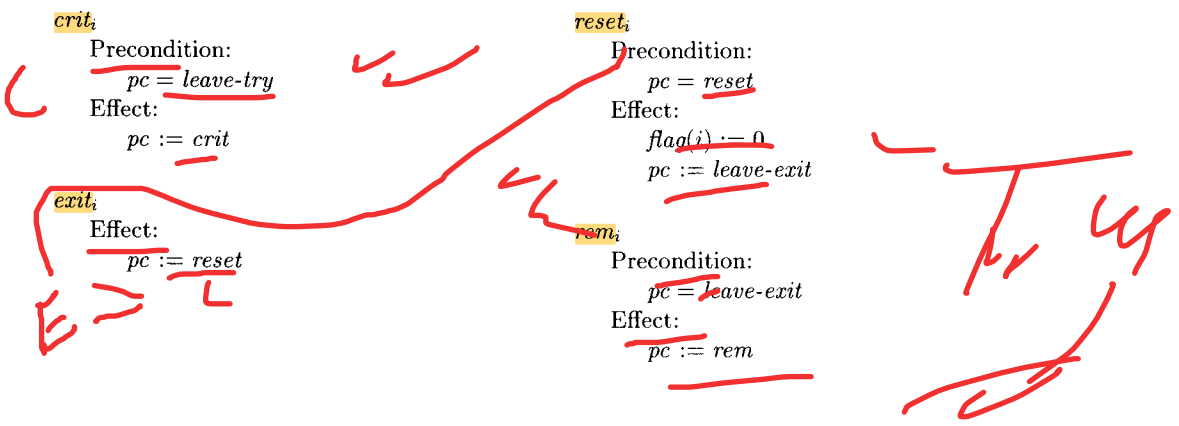
try_i
  Effect:
    $pc := set\text{-}flag$

$set\text{-}flag_i$
  Precondition:
    $pc = set\text{-}flag$
  Effect:
    $flag(i) := 1$
    $pc := set\text{-}turn$

$set\text{-}turn_i$
  Precondition:
    $pc = set\text{-}turn$
  Effect:
    $turn := i$
    $pc := check\text{-}flag$

$check\text{-}flag_i$
  Precondition:
    $pc = check\text{-}flag$
  Effect:
    if $flag(\bar{i}) = 0$ then
      $pc := leave\text{-}try$
    else
      $pc := check\text{-}turn$

$check\text{-}turn_i$
  Precondition:
    $pc = check\text{-}turn$
  Effect:
    if $turn \neq i$ then
      $pc := leave\text{-}try$
    else
      $pc := check\text{-}flag$



We now argue that the *Peterson2P* algorithm is correct. Well-formedness is easy to check.

Lemma 10.10 *Peterson2P satisfies mutual exclusion.*

Proof. We use an argument based on invariant assertions. It is easy to show by induction that

Assertion 10.5.1 *In any reachable system state, if* $flag(i) = 0$*, then* $pc_i \in \{leave\text{-}exit, rem, set\text{-}flag\}$.

Using Assertion 10.5.1, we can show by induction that

Assertion 10.5.2 *In any reachable system state, if* $pc_i \in \{leave\text{-}try, crit, reset\}$ *and* $pc_{\bar{i}} \in \{check\text{-}flag, check\text{-}turn, leave\text{-}try, crit, reset\}$*, then* $turn \neq i$.

That is, if i has won the competition, and if $\bar{i}$ is a competitor, then the *turn* variable is set favorably for i, that is, set to the value $\bar{i}$. In the inductive step of the proof of Assertion 10.5.2, the key events to check are

1. "Successful" $check\text{-}flag_i$ events, that is, those that cause pc_i to reach *leave-try*
2. "Successful" $check\text{-}turn_i$ events
3. $set\text{-}turn_{\bar{i}}$ events, which cause $pc_{\bar{i}}$ to take on the value *check-flag*
4. $set\text{-}turn_i$ events, which falsify the conclusion $turn \neq i$

When i does a successful *check-flag*$_i$, it must be that *flag*$(\bar{i}) = 0$, which implies by Assertion 10.5.1 that $pc_{\bar{i}} \notin \{$*check-flag*, *check-turn*, *leave-try*, *crit*, *reset*$\}$, which makes the statement true vacuously. When i does a successful *check-turn*$_i$, it must be that $turn \neq i$, which suffices. When $\bar{i}$ does *set-turn*$_{\bar{i}}$, it explicitly sets $turn \neq i$, which suffices. Finally, when i does *set-turn*$_i$, then the resulting $pc_i =$ *check-flag*, which makes the statement true vacuously.

This proves Assertion 10.5.2. Now mutual exclusion follows easily: Suppose that both i and $\bar{i}$ are in C, in some reachable state. Then Assertion 10.5.2, applied twice—for i and $\bar{i}$—implies that both $turn \neq i$ and $turn \neq \bar{i}$. This is a contradiction. □

Lemma 10.11 *Peterson2P guarantees progress.*

Proof. Suppose for the sake of contradiction that α is a low-level-fair execution that reaches a point where at least one of the processes, say i, is in T and neither process is in C, and suppose that after this point, neither process ever enters C. We consider two cases. First, if $\bar{i}$ is in T sometime after the given point in α, then both processes must get stuck permanently in their *check* loops, since neither ever enters C. But this cannot happen, since *turn* must stabilize to a value that is favorable to one of them.

On the other hand, suppose that $\bar{i}$ is never in T after the given point in α. In this case, we can show that *flag*$(\bar{i})$ eventually becomes and stays equal to 0, contradicting the assumption that i is stuck in its *check* loop. □

Lemma 10.12 *Peterson2P is lockout-free.*

// proof in copy

Proof. The argument for the exit region is trivial; we consider the trying region. We show the stronger condition of two-bounded bypass and invoke Theorem 10.8.

Suppose the contrary, that is, that at some point in execution α, process i is in T after having performed *set-flag*$_i$, and thereafter, while i remains in T, process $\bar{i}$ enters C three times. Note that in each of the second and third times, it must be that $\bar{i}$ first sets $turn := \bar{i}$ and then sees $turn = i$; it cannot see *flag*$(i) = 0$, because *flag*(i) remains at 1. This means that there are at least two occurrences of *set-turn*$_i$ after the given point in α, because only i can set *turn* to i. But *set-turn*$_i$ is only performed once during one of i's trying regions. This is a contradiction. □

So we have Theorem 10.13.

Theorem 10.13 *Peterson2P solves the mutual exclusion problem and guarantees lockout-freedom.*

Complexity analysis. As for the analysis of *DijkstraME*, let ℓ and c be upper bounds on process step time and critical section time, respectively. You might want to reread our discussion at the beginning of Section 10.3.4 to be sure you understand exactly what these bounds mean.

Theorem 10.14 *In Peterson2P, the time from when a particular process i enters T until it enters C is at most $c + O(\ell)$.*

Proof Sketch. Suppose the bound does not hold and consider an execution in which process i is in T at some point, but does not enter C for time at least $c + k\ell$ after that point, for some particular large constant k. The constant k is chosen to be considerably bigger than the constants in the big-O terms in the following analysis.

First, within time at most 3ℓ, process i performs *check-flag*$_i$. This can be seen by a case analysis, based on the various places where i might be in its trying region. Note that i cannot succeed in any of its *check*s during this time, because if it did, it would go to C within time $O(\ell)$, which we have assumed does not happen this quickly. Then when process i performs this *check-flag*$_i$, it must find *flag*$(\bar{i}) = 1$, because otherwise i would reach C within time $O(\ell)$. So by Assertion 10.5.1, it must be that $pc_{\bar{i}} \in \{$*set-turn*, *check-flag*, *check-turn*, *leave-try*, *crit*, *reset*$\}$ at that point.

Then we claim that either *crit*$_{\bar{i}}$ occurs within additional time $O(\ell)$ or *reset*$_{\bar{i}}$ occurs within additional time $c + O(\ell)$. This is argued by a case analysis, based on the value of *turn* and where the processes are in their code; the key point is that the *turn* variable, once stabilized, will be set favorably to one of the processes. But the former case would again mean that i would reach C too soon, so the latter must hold, that is, *reset*$_{\bar{i}}$ occurs within additional time $c + O(\ell)$.

Now i performs *check-flag*$_i$ again, within additional time $O(\ell)$. Once again, it must find *flag*$(\bar{i}) = 1$. This means that $\bar{i}$ has entered T again, after the *reset*$_{\bar{i}}$. Then either *turn* already has taken on the value $\bar{i}$, or will do so within additional time ℓ. Then within at most another time $O(\ell)$, process i finds conditions favorable for it to enter C. This contradicts the assumption that i does not enter C within this amount of time. Figure 10.8 shows the order of events in this proof and the time bounds between them. □

10.5.2 An n-Process Algorithm

For n processes, we can use the idea of the *Peterson2P* algorithm iteratively, in a series of $n - 1$ competitions at levels $1, 2, \ldots, n - 1$. At each successive competition, the algorithm ensures that there is at least one *loser*. Thus, all n

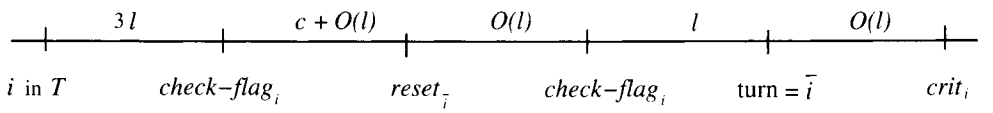


Figure 10.8: Order of events and time bounds in the proof of Theorem 10.14.

processes may compete in the level 1 competition, but at most $n-1$ processes can win. In general, at most $n-k$ processes can win at level k. So at most one process can win at level $n-1$, which yields the mutual exclusion condition.

The code is given below. Here we have reverted to our usual convention of numbering the processes $1, \ldots, n$. We call the algorithm *PetersonNP*.

***PetersonNP* algorithm:**

Shared variables:
for every $k \in \{1, \ldots, n-1\}$:
    $turn(k) \in \{1, \ldots, n\}$, initially arbitrary, writable and readable by all processes
for every i, $1 \leq i \leq n$:
    $flag(i) \in \{0, \ldots, n-1\}$, initially 0, writable by i and readable by all $j \neq i$

Process i:

```
** Remainder region **

try_i
for k = 1 to n − 1 do
    flag(i) := k
    turn(k) := i
    waitfor [∀j ≠ i : flag(j) < k] or [turn(k) ≠ i]
crit_i

** Critical region **

exit_i
flag(i) := 0
rem_i
```

Process i engages in one competition for each level, $1 \leq k \leq n-1$. Now each level k has its own *turn* variable, $turn(k)$. At each level k, process i behaves similarly to the way a process behaves in the *Peterson2P* algorithm: it sets $turn(k) := i$, then waits to discover either that all the other processes' *flag*

variables are strictly less than k, or else that $turn(k) \neq i$. That is, either none of the other processes is currently involved in the level k competition, or else the $turn(k)$ variable has been reset by some other process since i most recently set it.

As before, there are some ambiguities in the code that need to be resolved. First, one of the conditions in the waitfor statement involves the *flag* variables for all the other processes. In our model, these variables cannot all be checked simultaneously. Rather, we intend that the variables be checked one at a time, and we regard the condition as satisfied if all the values seen during these checks are less than k. Second, we need to specify some conditions on the order in which process i checks the various *flag* variables and the $turn(k)$ variable, in the waitfor statement. For simplicity, we assume that process i cycles through the checks, in each cycle first checking all the *flag* variables in arbitrary order and then checking the $turn(k)$ variable.

The details appear below. The code is quite similar to that of *Peterson2P*. Note the use of the local variable *level* to keep track of which competition the process is engaged in (or is ready to engage in) and the use of S to keep track of processes that have been observed to have *flag* values smaller than k.

***PetersonNP* algorithm (rewritten):**

Shared variables:
for every $k \in \{1, \ldots, n-1\}$:
  $turn(k) \in \{1, \ldots, n\}$, initially arbitrary
for every i, $1 \leq i \leq n$:
  $flag(i) \in \{0, \ldots, n-1\}$, initially 0

Actions of i:

Input:	Internal:
try_i	$set\text{-}flag_i$
$exit_i$	$set\text{-}turn_i$
Output:	$check\text{-}flag(j)_i$, $1 \leq j \leq n$, $j \neq i$
$crit_i$	$check\text{-}turn_i$
rem_i	$reset_i$

States of i:
$pc \in \{rem, set\text{-}flag, set\text{-}turn, check\text{-}flag, check\text{-}turn, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially rem
$level \in \{1, \ldots, n-1\}$, initially 1
S, a set of process indices, initially $\emptyset$

Transitions of i:

try_i
Effect:
$pc := set\text{-}flag$

$set\text{-}flag_i$
Precondition:
$pc = set\text{-}flag$
Effect:
$flag(i) := level$
$pc := set\text{-}turn$

$set\text{-}turn_i$
Precondition:
$pc = set\text{-}turn$
Effect:
$turn(level) := i$
$S := \{i\}$
$pc := check\text{-}flag$

$check\text{-}flag(j)_i$
Precondition:
$pc = check\text{-}flag$
$j \notin S$
Effect:
if $flag(j) < level$ then
$S := S \cup \{j\}$
if $|S| = n$ then
$S := \emptyset$
if $level < n - 1$ then
$level := level + 1$
$pc := set\text{-}flag$
else
$pc := leave\text{-}try$
else
$S := \emptyset$
$pc := check\text{-}turn$

$check\text{-}turn_i$
Precondition:
$pc = check\text{-}turn$
Effect:
if $turn(level) \neq i$ then
if $level < n - 1$ then
$level := level + 1$
$pc := set\text{-}flag$
else
$pc := leave\text{-}try$
else
$S := \{i\}$
$pc := check\text{-}flag$

$crit_i$
Precondition:
$pc = leave\text{-}try$
Effect:
$pc := crit$

$exit_i$
Effect:
$pc := reset$

$reset_i$
Precondition:
$pc = reset$
Effect:
$flag(i) := 0$
$level := 1$
$pc := leave\text{-}exit$

rem_i
Precondition:
$pc = leave\text{-}exit$
Effect:
$pc := rem$

We now argue that *PetersonNP* is correct. Well-formedness is clear. For mutual exclusion, the key idea is that the level k competition only permits $n - k$ winners.

In any system state of *PetersonNP*, we say that a process i is a *winner* at level k provided that either $level_i > k$ or else $level_i = k$ and $pc_i \in \{leave\text{-}try, crit, reset\}$. (This latter condition will only arise for $k = n - 1$.) We also say that process i

is a *competitor* at level k, provided that it is either a winner at level k or else $level_i = k$ and $pc_i \in \{check\text{-}flag, check\text{-}turn\}$.

Lemma 10.15 *PetersonNP satisfies mutual exclusion.*

Proof. In order to prove mutual exclusion, we prove the following assertion, which is analogous to Assertion 10.5.2 for *Peterson2P*. An important difference is that now the assertion must deal with intermediate stages in the process of checking flags.

> **Assertion 10.5.3** *In any reachable system state of PetersonNP, the following are true:*
>
> 1. *If process i is a competitor at level k, if $pc_i = check\text{-}flag$, and if any process $j \neq i$ in S_i is a competitor at level k, then $turn(k) \neq i$.*
> 2. *If process i is a winner at level k and if any other process is a competitor at level k, then $turn(k) \neq i$.*

The proof, by induction as usual, is left as an exercise. Using Assertion 10.5.3, we prove

> **Assertion 10.5.4** *In any reachable system state of PetersonNP, if there is a competitor at level k, then the value of $turn(k)$ is the index of some competitor at level k.*

Again, the inductive proof is left as an exercise. Finally, we show the following, which directly implies the mutual exclusion condition.

> **Assertion 10.5.5** *In any reachable system state of PetersonNP, and for any k, $1 \leq k \leq n - 1$, there are at most $n - k$ winners at level k.*

The proof of Assertion 10.5.5 is also an induction, but not on the length of an execution. Rather, we use induction on the value of k.

Basis: $k = 1$. If the statement is false for $k = 1$, it means that all n processes are winners at level 1. Then Assertion 10.5.3 implies that the value of $turn(1)$ cannot be the index of any of the processes, a contradiction.

Inductive step: We assume the statement for k, $1 \leq k \leq n - 2$, and show it for $k + 1$. Suppose for the sake of contradiction that the statement is false for $k+1$, that is, that there are strictly more than $n - (k+1)$ winners at level $k+1$; let W be the set of such winners. Every winner at level $k + 1$ is also a winner

at level k, and by the inductive hypothesis, the number of winners at level k is at most $n-k$. It follows that W is also the set of winners at level k, and that $|W| = n-k \geq 2$.

Then Assertion 10.5.3 implies that the value of $turn(k+1)$ cannot be the index of any of the processes in W. And Assertion 10.5.4 implies that the value of $turn(k+1)$ is the index of some competitor at level $k+1$. But every competitor at level $k+1$ is a winner at level k, and so is in W. This is a contradiction. □

In order to prove progress, it is enough to prove lockout-freedom (see Exercise 10.6). And Theorem 10.9 implies that lockout-freedom is in turn implied by a time bound. A time bound for the exit region is trivial; the following theorem gives a time bound for the trying region. Warning: We do not claim that this bound is tight—we leave it as an exercise to try to tighten it—but any bound is enough to prove lockout-freedom.

Theorem 10.16 *In PetersonNP, the time from when a particular process i enters T until it enters C is at most $2^{n-1}c + O(2^n n\ell)$.*

Proof. We prove the bound using a recurrence. Define $T(0)$ to be the maximum time from when a process enters T until it enters C. For k, $1 \leq k \leq n-1$, define $T(k)$ to be the maximum time from when a process becomes a winner at level k until it enters C. We want to bound $T(0)$.

By the code, we know that $T(n-1) \leq \ell$, since only one step is needed to enter C after winning the final competition. In order to bound $T(0)$, we set up a recurrence for $T(k)$ in terms of $T(k+1)$, where $0 \leq k \leq n-2$.

Suppose process i has just won at level k if $k \geq 1$, or has just entered T if $k = 0$. Then within time 2ℓ, process i performs *set-turn*$_i$, setting $turn(k+1) := i$. Let π denote this *set-turn*$_i$ event. We consider two cases.

First, if $turn(k+1)$ gets set to some value other than i within time $T(k+1) + c + (2n+2)\ell$ after π, then i wins at level $k+1$ within an additional time $n\ell$. Then within additional time $T(k+1)$, i enters C. In this case, the total time from π until i's entrance to C is at most $2T(k+1) + c + (3n+2)\ell$.

On the other hand, assume that $turn(k+1)$ does not get set to any value other than i within time $T(k+1) + c + (2n+2)\ell$ after π. Then no process can set its *flag* to $k+1$ within time $T(k+1) + c + (2n+1)\ell$ after π. Let I be the set of processes $j \neq i$ for which $flag(j) \geq k+1$ when π occurs. Then each process in I wins at level $k+1$ within time at most $n\ell$ after π (since it finds $turn(k+1)$ unequal to its index), then enters C within an additional time $T(k+1)$, then leaves C within additional time c and performs *reset* within additional time ℓ. That is, within time $n\ell + T(k+1) + c + \ell = T(k+1) + c + (n+1)\ell$ after π, all processes in I set their *flag*s to 0.

Thus, within time $T(k+1)+c+(n+1)\ell$ after π, all processes $j \neq i$ for which $\mathit{flag}(j) \geq k+1$ when π occurs, set their *flag*s to 0. As we assumed above, for an additional time $n\ell$ after that, no process sets its *flag* to $k+1$. That is sufficient time for process i to detect that all the *flag* variables are less than $k+1$ and so to win at level $k+1$. That is, in this case, process i wins at level $k+1$ within time $T(k+1)+c+(2n+1)\ell$ after π. Again, within another $T(k+1)$, i enters C. In this case, the total time from π until i's entrance to C is at most $2T(k+1)+c+(2n+1)\ell$.

The worst-case time is thus at most 2ℓ plus the maximum of the times in the two cases above, that is, $2T(k+1)+c+(3n+4)\ell$. Thus, we need to solve the following recurrence for $T(0)$:

$$T(k) \leq 2T(k+1)+c+(3n+4)\ell, \text{ for } 0 \leq k \leq n-2$$
$$T(n-1) \leq \ell$$

Solving this recurrence yields the claimed time bound. (See the following subsection, Section 10.5.3, for a more detailed solution for a similar recurrence.) □

We have

Theorem 10.17 *PetersonNP solves the mutual exclusion problem and is lockout-free.*

10.5.3 Tournament Algorithm

Another way to extend the basic *Peterson2P* algorithm to more processes is to use a version of the basic two-process algorithm as a building block in a *tournament*. For simplicity, we assume that n, the number of processes, is a power of 2. Once again, we number the processes starting with 0, as $0, \ldots, n-1$ rather than $1, \ldots, n$. Each process engages in a series of $\log n$ competitions in order to obtain the resource. You should think of these competitions as being arranged in a complete n-leaf binary *tournament tree*; the n leaves correspond left-to-right to the n processes $0, \ldots, n-1$.

We need some notation to name the various competitions, the roles played by the processes in all of the competitions, and the set of potential opponents that all the processes can have in all the competitions. For $0 \leq i \leq n-1$ and $1 \leq k \leq \log n$, we define the following notions.

- $\mathit{comp}(i,k)$, the *level* k *competition* of process i, is the string consisting of the high-order $\log n - k$ bits of the binary representation of i. In terms

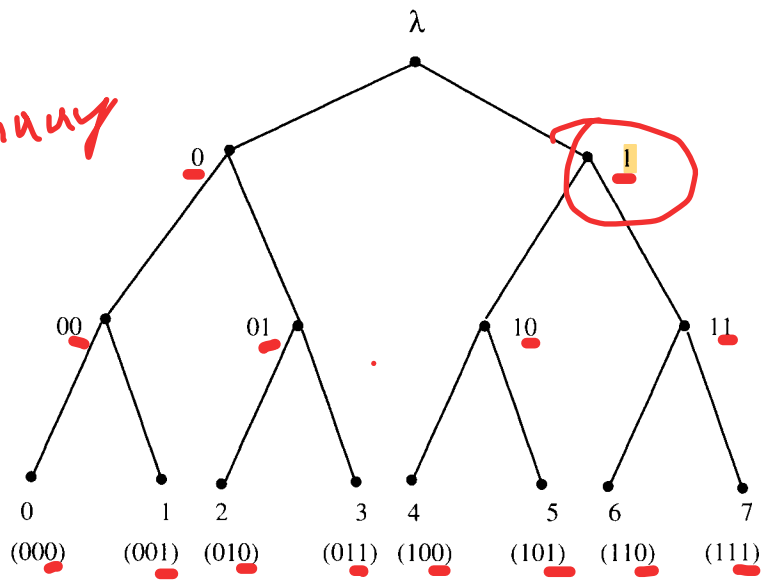


Figure 10.9: Names of competitions in the *Tournament* algorithm.

of the tournament tree, $comp(i, k)$ can be used as a name for the internal node that is the level k ancestor of i's leaf. In particular, the root is named by λ, the empty string.

- $role(i, k)$, the *role* of process i in the level k competition of process i, is the $(\log n - k + 1)$st bit of the binary representation of i. In terms of the tournament tree, $role(i, k)$ indicates whether i's leaf is a descendant of the left or right child of the node for competition $comp(i, k)$.

- $opponents(i, k)$, the *opponents* of process i in the level k competition of process i, is the set of process indices with the same high-order $\log n - k$ bits as i and the opposite $(\log n - k + 1)$st bit. In terms of the tournament tree, the processes in $opponents(i, k)$ are those whose leaves are descendants of the opposite child of node $comp(i, k)$, that is, of the child that is not an ancestor of i's leaf.

Example 10.5.1 Tournament tree

Figure 10.9 shows the tournament tree for $n = 8$. For example, note that $comp(5, 2) = 1$, $role(5, 2) = 0$ and $opponents(5, 2) = \{6, 7\}$.

We call the algorithm the *Tournament algorithm*.

***Tournament* algorithm:**

Shared variables:
for every binary string x of length at most $\log n - 1$:
    $turn(x) \in \{0, 1\}$, initially arbitrary, writable and readable by exactly those processes i for which x is a prefix of the binary representation of i
for every i, $0 \leq i \leq n - 1$:
    $flag(i) \in \{0, \ldots, \log n\}$, initially 0, writable by i and readable by all $j \neq i$

Process i:

    ** Remainder region **

    try_i
    for $k = 1$ to $\log n$ do
        $flag(i) := k$
        $turn(comp(i, k)) := role(i, k)$
        waitfor $[\forall j \in opponents(i, k) : flag(j) < k]$ or $[turn(comp(i, k)) \neq role(i, k)]$
    $crit_i$

    ** Critical region **

    $exit_i$
    $flag(i) := 0$
    rem_i

This code is very much like that of the *PetersonNP* algorithm. The main difference is that in each competition, the process only checks the flags of its opponents in that competition. As in *PetersonNP*, we assume that a process checks its opponents in any order, one at a time. The testing must alternate in some systematic way; for example, it can be done in a cycle where all the *flags* are first tested, and then *turn*. We only sketch the correctness arguments for the *Tournament* algorithm briefly, since the ideas are so similar to those for the *PetersonNP* and *Peterson2P* algorithms.

First, the algorithm should be rewritten in precondition-effect style, making explicit the program counters and the variables that accumulate the sets of processes whose flags have already been checked. Then notions such as "winner at level k" and "competitor at level k" must be defined for the *Tournament* algorithm, analogously to the way they were defined for *PetersonNP*.

Lemma 10.18 *The Tournament algorithm satisfies mutual exclusion.*

Proof Sketch. The proof uses the same ideas as the invariant assertion proofs of the *Peterson2P* and *PetersonNP* algorithms. This time, the key invariant is

> **Assertion 10.5.6** *In any reachable system state of the Tournament algorithm, and for any k, $1 \leq k \leq \log n$, at most one process from any subtree rooted at level k is a winner at level k.*

This follows immediately from an invariant analogous to the second part of Assertion 10.5.3.

> **Assertion 10.5.7** *If process i is a winner at level k and if any level-k opponent of i is a competitor at level k, then $turn(comp(i,k)) \neq role(i,k)$.*

As for the second part of Assertion 10.5.3, we cannot prove Assertion 10.5.7 directly by induction. We must strengthen it as before to include some information about what happens inside the waitfor loop, after the process has discovered that some of its opponents have *flag* variables with values that are strictly less than k. We leave this strengthening and the inductive proof as an exercise for the reader. □

In order to show progress and lockout-freedom, we prove a time bound.

Theorem 10.19 *In the Tournament algorithm, the time from when a particular process i enters T until it enters C is at most $(n-1)c + O(n^2\ell)$.*

Proof. The proof is like the proof of Theorem 10.16. Define $T(0)$ to be the maximum time from when a process enters T until it enters C. For k, $1 \leq k \leq \log n$, define $T(k)$ to be the maximum time from when a process wins at level k until it enters C. We want to bound $T(0)$. By the code, we know that $T(\log n) \leq \ell$, since only one step is needed to enter C after winning the final competition. We bound $T(k)$ in terms of $T(k+1)$, where $0 \leq k \leq \log n - 1$.

Suppose process i has just won at level k if $k \geq 1$, or has just entered T if $k = 0$. Let x denote $comp(i, k+1)$. Then, within time 2ℓ, process i sets the $turn(x)$ variable to $role(i, k+1)$. Let π denote this event; we consider two cases.

First, if $turn(x)$ gets changed within time $T(k+1) + c + (2^{k+1} + 4)\ell$ after π, then i wins at level $k+1$ within an additional time $(2^k + 1)\ell$. Then, within additional time $T(k+1)$, i enters C. In this case, the total time from π until i's entrance to C is at most $2T(k+1) + c + (2^{k+1} + 2^k + 5)\ell$.

On the other hand, assume that $turn(x)$ does not get changed within time $T(k+1) + c + (2^{k+1} + 4)\ell$ after π. Then no level $k+1$ opponent of i can set its *flag* to $k+1$ within time $T(k+1) + c + (2^{k+1} + 3)\ell$ after π. If j is a level

$k+1$ opponent of i for which $flag(j) \geq k+1$ when π occurs, then within time $(2^k+1)\ell + T(k+1) + c + \ell = T(k+1) + c + (2^k+2)\ell$ after π, process j sets its *flag* to 0.

Thus, within time $T(k+1)+c+(2^k+2)\ell$ after π, all level $k+1$ opponents j of i for which $flag(j) \geq k+1$ when π occurs, set their *flag*s to 0. As we assumed above, for an additional time $(2^k+1)\ell$ after that, no process sets its *flag* to $k+1$. That is sufficient time for process i to detect that all its level $k+1$ opponents' *flag* variables are less than $k+1$, and so to win at level $k+1$. That is, in this case, process i wins at level $k+1$ within time $T(k+1)+c+(2^{k+1}+3)\ell$ after π. Within another $T(k+1)$, i enters C. In this case, the total time from π until i's entrance to C is at most $2T(k+1)+c+(2^{k+1}+3)\ell$.

The worst-case time is thus at most 2ℓ plus the maximum of the times in the two cases above, that is, $2T(k+1)+c+(2^{k+1}+2^k+7)\ell$. Thus, we need to solve the following recurrence for $T(0)$:

$$T(k) \leq 2T(k+1) + c + (2^{k+1}+2^k+7)\ell, \text{ for } 0 \leq k \leq \log n - 1$$
$$T(\log n) \leq \ell.$$

Choose some constant a such that $(2^{k+1}+2^k+7) \leq a \cdot 2^k$. Then we have

$$\begin{aligned}
T(0) &\leq 2T(1) + 2^0 c + a2^0\ell \\
&\leq 2^2T(2) + (2^0+2^1)c + a(2^0+2^2)\ell \\
&\leq 2^3T(3) + (2^0+2^1+2^2)c + a(2^0+2^2+2^4)\ell \\
&\vdots \\
&\leq 2^kT(k) + (2^0+2^1+\ldots+2^{k-1})c + a(2^0+2^2+\ldots+2^{2k-2})\ell \\
&\vdots \\
&\leq 2^{\log n}T(\log n) + (2^0+2^1+\ldots+2^{\log n-1})c + a(2^0+2^2+\ldots 2^{2(\log n-1)})\ell \\
&\leq (n-1)c + n\ell + O\left(n^2\ell\right) \\
&= (n-1)c + O\left(n^2\ell\right).
\end{aligned}$$

□

Theorem 10.20 *The Tournament algorithm solves the mutual exclusion problem and is lockout-free.*

Bounded bypass. The *Tournament* algorithm does *not* guarantee any bound on the number of bypasses. To see this, consider an execution in which process 0 enters the tournament at its leaf and takes steps with intervening times exactly

equal to the assumed upper bound ℓ. Meanwhile, process $n-1$ enters the tournament at its leaf, going much faster. Process $n-1$ can reach the top and win, and in fact it can repeat this arbitrarily many times, before process 0 even wins at level 1. This is possible because we have not assumed any lower bound on process step times.

Note that there is no contradiction between unbounded bypass and a time upper bound. No process is locked out for very long—the unbounded bypasses only occur because some processes operate very fast.

10.6 An Algorithm Using Single-Writer Shared Registers

The mutual exclusion algorithms we have studied so far use multi-writer shared registers (the *turn* variables) as well as single-writer shared registers (the *flag* variables). Because multi-writer registers are often difficult to implement, it is worth investigating algorithms that use only single-writer shared registers. In this section and the next, we present two such algorithms.

The algorithm in this section solves the mutual exclusion problem (including the progress condition, as usual), but does not guarantee any high-level-fairness condition. Its shared registers are all binary. The algorithm in Section 10.7 is also lockout-free, but it has the disadvantage of using unbounded size variables.

We call the first algorithm *BurnsME*, after Burns, its inventor.

BurnsME algorithm:

Shared variables:
for every i, $1 \le i \le n$:
  $flag(i) \in \{0,1\}$, initially 0, writable by i and readable by all $j \neq i$

Process i:

```
        ** Remainder region **

        try_i
L:      flag(i) := 0
        for j, 1 ≤ j ≤ i − 1 do
            if flag(j) = 1 then goto L
        flag(i) := 1
        for j, 1 ≤ j ≤ i − 1 do
            if flag(j) = 1 then goto L
M:      for j, i + 1 ≤ j ≤ n do
            if flag(j) = 1 then goto M
```

$crit_i$

** Critical region **

$exit_i$
$flag(i) := 0$
rem_i

The *flag* values used in *BurnsME* are 0 and 1 instead of 0, 1, and 2 as in *DijkstraME*. Each process executes three for loops. The first two loops involve checking the *flag*s of all processes with smaller indices, while the third loop involves checking the *flag*s of all processes with larger indices. If process i passes all the tests in all three loops, it proceeds to its critical region.

Lemma 10.21 *The BurnsME algorithm satisfies mutual exclusion.*

Proof. The proof is similar to the first (operational) proof that *DijkstraME* satisfies mutual exclusion (see Lemma 10.3). The main difference is that now the *flag* variables are set to 1, whereas in *DijkstraME* they are set to 2.

Thus, if processes i and j are simultaneously in C, then assume that i sets its *flag* to 1 first. Then $flag(i)$ keeps the value 1 until process i leaves C. But after j sets $flag(j)$ to 1, j must check that $flag(i) = 0$ before j can enter C. (If $i < j$, then this is done in the second for loop, while if $i > j$, then it is done in the third for loop.) This check must occur during the interval when the value of $flag(i) = 1$, which yields a contradiction. □

Note that the first for loop in the code is not needed for the mutual exclusion condition.

Lemma 10.22 *BurnsME guarantees progress.*

Proof. The argument for the exit region is easy. For the trying region, we assume for the sake of contradiction that α is a low-level-fair execution that reaches a point where there is at least one process in T and no process in C, and that after this point, no process ever enters C. Arguing similarly to the way we did in the proof of Lemma 10.4, we can assume without loss of generality that every process is in T or R and that no process changes region, in α. Let the *contenders* be the processes in T.

Now we partition the contenders into two sets: those that ever reach label M and those that never do. Call the first set P and the second set Q. There must

be some point in α by which all the processes in P have already reached label M; note that they never thereafter drop back to any point in the code prior to label M. Let α_1 be a suffix of α in which all processes in P are in the final for loop, after label M.

We claim that there is at least one process in P. Specifically, the process with the smallest index among all the contenders is not blocked from reaching label M.

Let i be the largest index of a process in P. We claim that eventually in α_1, any process $j \in Q$ such that $j > i$ has $flag(j)$ set permanently to 0. This is because each time j executes one of the first two for loops, it discovers the presence of a smaller index contender and returns to L. Whenever it does this, it sets $flag(j) := 0$, and once it has done this, it can never progress far enough to set $flag(j) := 1$. So let α_2 be a suffix of α_1 in which all processes in Q with indices $> i$ always have their *flag*s equal to 0.

Now in α_2, there is nothing to stop process i from reaching C: every larger-index process j has $flag(j) = 0$, so i will complete the third for loop successfully. Thus, i enters C, which is a contradiction. □

Theorem 10.23 *BurnsME solves the mutual exclusion problem.*

10.7 The Bakery Algorithm

In this section we present the *Bakery algorithm* for mutual exclusion. It works somewhat the way a bakery does, where customers draw tickets when they enter and are served in the order of their ticket numbers.

The *Bakery* algorithm only uses single-writer/multi-reader shared registers. In fact, it also works using a weaker form of register known as a *safe register*, in which the registers are allowed to provide arbitrary responses to reads that are performed concurrently with writes.

The *Bakery* algorithm guarantees lockout-freedom and a good time bound. It guarantees bounded bypass and also a related condition—it is "FIFO after a wait-free doorway" (to be defined below). An unattractive property of the *Bakery* algorithm is that it uses unbounded size registers.

The code follows. We remark that the code given here can be simplified if we are only interested in the usual sort of registers (and not weaker types of registers such as safe registers). We leave this simplification for an exercise.

Bakery algorithm:

Shared variables:
for every i, $1 \leq i \leq n$:
  $choosing(i) \in \{0, 1\}$, initially 0, writable by i and readable by all $j \neq i$
  $number(i) \in \mathbb{N}$, initially 0, writable by i and readable by all $j \neq i$

Process i:

```
** Remainder region **

try_i
choosing(i) := 1
number(i) := 1 + max_{j≠i} number(j)
choosing(i) := 0
for j ≠ i do
    waitfor choosing(j) = 0
    waitfor number(j) = 0 or (number(i), i) < (number(j), j)
crit_i

** Critical region **

exit_i
number(i) := 0
rem_i
```

In the *Bakery* algorithm, the first part of the trying region, until the point where process i sets $choosing(i) := 0$, is designated as the *doorway*. While in the doorway, process i chooses a *number* that is greater than all the numbers that it reads for the other processes. It reads the other processes' *number*s one at a time, in any order, then writes its own *number*. While it is reading and choosing numbers, i makes sure that $choosing(i) = 1$, as a signal to the other processes.

Note that it is possible for two processes to be in the doorway at the same time, which can cause them to choose the same number. To break such ties, processes compare not just their *number*s, but their $(number, index)$ pairs. This comparison is done lexicographically, thus breaking ties in favor of the process with the smaller index.

In the rest of the trying region, the process waits for the other processes to finish choosing and also waits for its $(number, index)$ pair to become the lowest.

To prove correctness, let D denote the doorway (i.e., the set of process states in which the process is in the doorway), and let $T - D$ denote the rest of the trying region. Well-formedness is easy to see. To show the mutual exclusion condition, we use a lemma.

Lemma 10.24 *In any reachable system state of the Bakery algorithm, and for any processes i and j, $i \neq j$, the following is true. If i is in C and j is in $(T - D) \cup C$, then $(number(i), i) < (number(j), j)$.*

We give an operational proof, since it can be extended more easily to the safe register case.

Proof. Fix some point s in an execution in which i is in C and j is in $(T - D) \cup C$. (Formally, s is an occurrence of a system state.) Call the values of $number(i)$ and $number(j)$ at point s the *correct* values of these variables.

Process i must read $choosing(j) = 0$ in its first waitfor loop, prior to entering C. Let π denote this reading event; thus, π precedes s. When π occurs, j is not in the "choosing region" (i.e., the portion of the doorway after setting $choosing(j) := 1$). But since j is in $(T - D) \cup C$ at point s, j must pass through the choosing region at some point. There are two cases to consider.

1. j enters the choosing region after π. Then the correct $number(i)$ is chosen before j starts choosing, ensuring that j sees the correct $number(i)$ when it chooses. Therefore, at point s, we have $number(j) > number(i)$, which suffices.

2. j leaves the choosing region before π. Then whenever i reads j's number in its second waitfor loop, it gets the correct $number(j)$ But since i decides to enter C anyhow, it must be that $(number(i), i) < (number(j), j)$. This again suffices.

□

Lemma 10.25 *The Bakery algorithm satisfies mutual exclusion*

Proof. Suppose that, in some reachable state, two processes, i and j, are both in C. Then by Lemma 10.24 applied twice, we must have both $(number(i), i) < (number(j), j)$ and $(number(j), j) < (number(i), j)$. This is a contradiction. □

Lemma 10.26 *The Bakery algorithm guarantees progress.*

Proof. The exit region is easy, as usual. For the trying region, we again argue by contradiction. Suppose that progress is not guaranteed. Then eventually a point is reached after which all processes are in T or R, and no new region changes occur. By the code, all of the processes in T eventually complete the doorway and reach $T - D$. Then the process with the lowest $(number, index)$ pair is not blocked from reaching C. □

Lemma 10.27 *The Bakery algorithm guarantees lockout-freedom.*

Proof. Consider a particular process i in T and suppose it never reaches C. Process i eventually completes the doorway and reaches $T - D$. Thereafter, any new process that enters the doorway sees i's latest *number* and so chooses a higher number. Thus, since i doesn't reach C, none of these new processes reach C either, since each is blocked by the test of $number(i)$ in its second wait loop.

But repeated use of Lemma 10.26 implies that there must be continuing progress, including infinitely many *crit* events, which contradicts the fact that all new entrants to the trying region are blocked. □

Theorem 10.28 *The Bakery algorithm solves the mutual exclusion problem and is lockout-free.*

Complexity analysis. An upper bound for the time from when a process i enters the trying region until it enters the critical region is $(n-1)c + O(n^2\ell)$. This is not so easy to show; we just give a brief sketch and leave the details for an exercise.

First, it only takes time $O(n\ell)$ for process i to complete the doorway; we must bound the length of the time interval I that i spends in $T - D$. Let P be the set of other processes already in T at the moment i enters $T - D$. Then only processes in P can enter C before i does, and each of these can only do so once. It follows that the total time within interval I during which some process is in C is at most $(n-1)c$, and that the total time within interval I during which some process is in the doorway is at most $O(n^2\ell)$.

It remains to bound the *residual time* within interval I, that is, the total time within I during which no process is either in C or in the doorway. We bound the residual time by considering the progress of processes in $P \cup \{i\}$. During the residual time, note that none of these processes is ever blocked in its first waitfor loop, since all the *choosing* variables are 0. Moreover, some process in $P \cup \{i\}$ will not be blocked at any step of its second waitfor loop either, and so, within residual time $O(n\ell)$, will enter C. After it finishes, some other process in $P \cup \{i\}$ will not be blocked, and so, within an additional residual time $O(n\ell)$, will enter C, and so on. This continues until i enters C, for a total residual time of $O(n^2\ell)$.

FIFO after a wait-free doorway. The *Bakery* algorithm guarantees a high-level-fairness condition that is somewhat stronger than lockout-freedom. Namely, if process i completes the doorway before j enters T, then j cannot enter C before i does. Note that the algorithm is not actually FIFO based on the time of entry

to T, or even the first locally controlled step in T. For example, process 1 could enter and set $choosing(1) := 1$; then process 2 could enter, choose a number, and complete the doorway; then process 1 could choose its number. In this case, process 1 would choose a larger number than process 2's, allowing 2 to precede it into C.

It would not be useful just to claim that an algorithm was "FIFO after a doorway," because there are no constraints on where the doorway might end. (If the doorway ended right at the entrance to C, then this claim is completely trivial.) However, the doorway in the *Bakery* algorithm has an interesting property: it is *wait-free*, which means that a process is guaranteed eventually to complete it, if that process continues to take steps, regardless of whether any other processes continue to do so.

Thus, the property of being "FIFO after a wait-free doorway," which is a nontrivial and interesting high-level-fairness condition, is satisfied by the *Bakery* algorithm.

10.8 Lower Bound on the Number of Registers

We have presented several mutual exclusion algorithms that use read/write shared memory. All guarantee the basic conditions of mutual exclusion and progress, and most also guarantee some sort of high-level-fairness condition: lockout-freedom, a time bound, or a bypass bound. One thing that all the algorithms have in common, though, is that they all use at least n shared variables.

In this section, we show that this is not an accident: it turns out that the mutual exclusion problem cannot be solved at all with fewer than n read/write shared variables! This is so even if we only require the basic conditions—mutual exclusion and progress; no high-level-fairness requirements are needed for proving this lower bound. Also, the impossibility result holds regardless of the size of the shared variables (as measured by the number of values they can take on)—they can be as small as a single bit or even unbounded in size. This result represents a fundamental limitation on the power of shared memory systems.

We need two definitions. First, as in Section 9.3, we say that two system states, s and s', are *indistinguishable* to process i, written as $s \overset{i}{\sim} s'$, if the state of process i, the state of U_i, and the values of all the shared variables are the same in s and s'. Second, we define a system state s to be *idle* if all processes are in their remainder regions in s.

In the proof, we consider a fixed collection of user automata. Namely, we assume that each user U_i is the most nondeterministic possible—that it is able to perform its *try* and *exit* outputs at any time, subject only to the well-formedness