UNIVERSAL SYNCHRONIZATION OBJECTS

Christian Maurer

Fachbereich Mathematik und Informatik
Freie Universität Berlin

24.4.2000

Abstract. Certain asymmetries in the solutions of several classical concurrency problems can be remedied by adding redundant code that does not affect the efficiency of the algorithms. On this way a common pattern emerges which allows for the development of an abstraction of different applications in form of a single class that completely encapsulates the essence of the orresponding synchronization paradigm.

We specify and implement two such universal synchronization objects (a general monitor and a general semaphor technique of passing the baton) and show their universality by applying these concepts to other classical examples.

There is strong evidence that central aspects of other synchronization mechanisms such as message passing could be condensed in similar objects; in case this conjecture turns out to be true, the client server paradigm will be examined in a subsequent paper.

Introduction

As a motivation for our generalization we consider the wellknown classical first problem of the readers and writers (readers preference), which uses the following entry- and exit-protocols (we use Modula-2 for our code examples, since blackbox reuse of abstract data objects and types is an appropriate tool to express our ideas):

```
DEFINITION MODULE RW1; (* first readers and writers problem *)

TYPE criticalSections;

PROCEDURE initialize (VAR cs: criticalSections);
(* pre:  cs is not initialized.
   post: cs is initialized. *)

PROCEDURE readerIn (cs: criticalSections);
(* pre:  cs is initialized. The calling process is neither an active reader
         nor an active writer in cs.
   post: The calling process is blocked as reader in cs, iff there is
         an active writer in cs; otherwise it is an active reader in cs. *)
```

1994 *CR Classification Scheme.* D.1.3, D.3.2., D.3.3., F.1.2, H.2.4.

Key words and phrases. process, concurrency, critical section, condition synchronization, monitor, condition variable, signal semantics, semaphore, passing the baton, readers and writers, bounded buffer.

```
PROCEDURE readerOut (cs: criticalSections);
(* pre:  cs is initialized. The calling process is an active reader in cs.
   post: The calling process is no more an active reader in cs. If there
         are any blocked writers in cs, exactly one of them is unblocked
         and is an active writer in cs. *)

PROCEDURE writerIn (cs: criticalSections);
(* pre:  cs is initialized. The calling process is neither an active reader
         nor an active writer in cs.
   post: The calling process is blocked as writer in cs, iff there is an
         active reader or another active writer in cs; otherwise it is an
         active writer in cs. *)

PROCEDURE writerOut (cs: criticalSections);
(* pre:  cs is initialized. The calling process is an active writer in cs.
   post: The calling process is no more an active writer in cs. If there
         are any blocked readers or writers in cs, exactly one of them
         (preferably a writer) is unblocked and is an active reader or
         writer resp. in cs. *)

END RW1.
```

The standard monitor solution of this problem requires the existence of an abstract data type `conditions` with with operations `delay` and `resume` to block respectively unblock the calling processes:

```
IMPLEMENTATION MODULE RW1 [ModulePriority];

FROM Storage IMPORT ALLOCATE;

TYPE criticalSections = POINTER TO csrec;
     csrec = RECORD
               activeReaders, activeWriters: CARDINAL;
               readerMayEnter, readerMayLeave,
               writerMayEnter, writerMayLeave: conditions
             END;

PROCEDURE initialize (VAR criticalSection: criticalSections);
BEGIN
  NEW (criticalSection);
  WITH criticalSection^ DO activeReaders:= 0; activeWriters:= 0 END
END initialize;

PROCEDURE readerIn (cs: criticalSections);
BEGIN
  WITH cs^ DO
    WHILE activeWriters > 0 DO delay (readerMayEnter) END;
    INC (activeReaders); resume (readerMayEnter)
  END
END readerIn;

PROCEDURE readerOut (cs: criticalSections);
BEGIN
  WITH cs^ DO
    DEC (activeReaders);
    IF activeReaders = 0 THEN resume (writerMayEnter) END
  END
END readerOut;
```

```
PROCEDURE writerIn (cs: criticalSections);
BEGIN
  WITH cs^ DO
    WHILE (activeReaders > 0)
       OR (activeWriters > 0) DO delay (writerMayEnter) END;
    activeWriters:= 1
  END
END writerIn;

PROCEDURE writerOut (cs: criticalSections);
BEGIN
  WITH cs^ DO
    activeWriters:= 0;
    resume (writerMayEnter); resume (readerMayEnter)
  END
END writerOut;

END RW1.
```

Now, in this solution of this problem there are certain obvious asymmetries: no `resume`-operation in the writers entry protocol `writerIn`, no `delay`-operations in the exit protocols `readerOut` and `writerOut` and an explicit boolean condition for the `resume`-operation only in the readers entry protocol `readerIn`.
By adding two superfluous condition variables (`...MayLeave`) in the exit protocols and several redundant signal operations—making use of the fact that, by the very definition of a monitor due to Hoare [H]—a signal operation on a condition variable does not have any effect if there are no processes delayed on the condition—these asymmetries can be avoided and one finds the common pattern in the four protocols:

```
IMPLEMENTATION MODULE RW1 [MonitorPriority];

FROM SYSTEM IMPORT ADDRESS;
FROM Storage IMPORT ALLOCATE;

TYPE criticalSections = POINTER TO csrec;
     csrec = RECORD
               activeReaders, activeWriters: CARDINAL;
               readerMayEnter, readerMayLeave,
               writerMayEnter, writerMayLeave: conditions
             END;

PROCEDURE initialize (VAR criticalSection: criticalSections);
BEGIN
  NEW (criticalSection);
  WITH criticalSection^ DO activeReaders:= 0; activeWriters:= 0 END
END initialize;

  PROCEDURE resumeAll (cs: criticalSections);
  BEGIN
    WITH cs^ DO
      IF activeWriters = 0 THEN resume (readerMayEnter) END;
      IF TRUE THEN resume (readerMayLeave) END;
      IF (activeReaders = 0)
       & (activeWriters = 0) THEN resume (writerMayEnter) END;
      IF TRUE THEN resume (writerMayLeave) END
    END
  END resumeAll;
```

```
PROCEDURE readerIn (cs: criticalSections);
BEGIN
  WITH cs^ DO
    WHILE NOT (activeWriters = 0) DO delay (readerMayEnter) END;
    INC (activeReaders)
  END;
  resumeAll (cs)
END readerIn;

PROCEDURE readerOut (cs: criticalSections);
BEGIN
  WITH cs^ DO
    WHILE NOT (TRUE) DO delay (readerMayLeave) END;
    DEC (activeReaders)
  END;
  resumeAll (cs)
END readerOut;

PROCEDURE writerIn (cs: criticalSections);
BEGIN
  WITH cs^ DO
    WHILE NOT ((activeReaders = 0) & (activeWriters = 0)) DO
      delay (writerMayEnter)
    END;
    activeWriters:= 1
  END;
  resumeAll (cs)
END writerIn;

PROCEDURE writerOut (cs: criticalSections);
BEGIN
  WITH cs^ DO
    WHILE NOT (TRUE) DO delay (writerMayLeave) END;
    activeWriters:= 0
  END;
  resumeAll (cs)
END writerOut;

END RW1.
```

In fact, the synchronization can be completely formulated in the following table:

operation	condition	action
`readerIn`	`activeWriters = 0`	`INC (activeReaders)`
`readerOut`	`TRUE`	`DEC (activeReaders)`
`writerIn`	`(activeReaders = 0)`	
	`& (activeWriters = 0)`	`activeWriters:= 1`
`writerOut`	`TRUE`	`activeWriters:= 0`

Our orthogonal redundant implementation is nothing but the realization of this table with the initial values `activeReaders = 0` and `activeWriters:= 0`.

A Universal Monitor

From all said above follows a universal specification of this concept.
We furthermore introduce the four standard signal semantics [A0, A] "signal and exit", "signal and continue", "signal and wait" and "signal and urgent wait" for the `resume`-operations in monitors and consider the following abstract data type:

```
DEFINITION MODULE Monitors;

FROM SYSTEM IMPORT ADDRESS;

CONST max = 10;
TYPE conditions = PROCEDURE (ADDRESS, CARDINAL): BOOLEAN;
     actions = PROCEDURE (ADDRESS, CARDINAL, ADDRESS);
     semantics = (exit, continue, wait, urgentWait);
     monitors;

PROCEDURE initialize (VAR m: monitors; s: semantics; x: ADDRESS;
                      n: CARDINAL; c: conditions; a: actions);
(* pre:  m is not initialized. n < max.
   post: m is initialized. m has the object x, the semantics s, the
         number of operations n, the condition c and the action a. *)

PROCEDURE operate (m: monitors; op: CARDINAL; b: ADDRESS);
(* pre:  m is initialized. op < number of operations of m. b is a
         reserved buffer address to hold data streams that are moved
         by the operation or NIL in case no data are to be transported.
   post: The calling process is delayed, until the condition of m for
         operation op has become true. Then the action of m for op is
         (under the semantics of m) executed, if necessary by moving
         a data stream starting at b into/out of the buffer. *)

END Monitors.
```

We prove that this concept can be implied by constructing an implementation on the basis of general semaphores as defined by Dijkstra [D], where `I` is the initialization of a semaphore with a given non negative integer value and `P` and `V` are the blocking resp. unblocking semaphore operations.

```
IMPLEMENTATION MODULE Monitors;

FROM SYSTEM IMPORT ADDRESS; FROM Storage IMPORT ALLOCATE;
FROM Semaphores IMPORT semaphores, I, P, V;

TYPE
  operations = [0..max-1];
  monitors = POINTER TO RECORD
               semantic: semantics;
               object: ADDRESS;
               numberops: CARDINAL;
               monitorEntry: semaphores;
               conditionSemaphor: ARRAY operations OF semaphores;
               waitingForCondition: ARRAY operations OF CARDINAL;
               urgentSemaphor: semaphores;
               urgentWaiting: CARDINAL;
               isTrue: conditions;
               execute: actions
             END;
```

```
PROCEDURE initialize (VAR monitor: monitors; s: semantics; x: ADDRESS;
                      n: CARDINAL; c: conditions; a: actions);
VAR op: operations;
BEGIN
  IF n >= max THEN RETURN END;
  NEW (monitor);
  WITH monitor^ DO
    semantic:= s;
    object:= x;
    numberops:= n;
    I (monitorEntry, 1);
    FOR op:= 0 TO numberops - 1 DO
      waitingForCondition [op]:= 0;
      I (conditionSemaphor [op], 0)
    END;
    IF semantic = urgentWait
      THEN I (urgentSemaphor, 0);
           urgentWaiting:= 0
    END;
    isTrue:= c;
    execute:= a
  END
END initialize;

PROCEDURE operate (monitor: monitors; operation: CARDINAL; buffer: ADDRESS);
VAR op: operations;
BEGIN
  WITH monitor^ DO
    IF operation >= numberops THEN RETURN END;
    P (monitorEntry);
    WHILE NOT isTrue (object, operation) DO
      CASE semantic OF
        exit:
          INC (waitingForCondition [operation]);
          V (monitorEntry);
          P (conditionSemaphor [operation]);
          DEC (waitingForCondition [operation]) |
        continue:
          INC (waitingForCondition [operation]);
          V (monitorEntry);
          P (conditionSemaphor [operation]);
          P (monitorEntry);
          DEC (waitingForCondition [operation]) |
        wait:
          INC (waitingForCondition [operation]);
          V (monitorEntry);
          P (conditionSemaphor [operation]);
          DEC (waitingForCondition [operation]) |
        urgentWait:
          INC (waitingForCondition [operation]);
          IF urgentWaiting > 0
            THEN V (urgentSemaphor)
            ELSE V (monitorEntry)
          END;
          P (conditionSemaphor [operation]);
          DEC (waitingForCondition [operation])
      END
    END;
```

```
    execute (object, operation, buffer);
    FOR op:= 0 TO numberops - 1 DO
      IF isTrue (object, op)
        THEN CASE semantic OF
               exit:
                  IF waitingForCondition [op] > 0
                    THEN V (conditionSemaphor [op]);
                         RETURN
                  END |
               continue:
                 IF waitingForCondition [op] > 0
                   THEN V (conditionSemaphor [op])
                  END |
               wait:
                 IF waitingForCondition [op] > 0
                   THEN V (conditionSemaphor [op]);
                        P (monitorEntry)
                 END |
               urgentWait:
                 IF waitingForCondition [op] > 0
                   THEN INC (urgentWaiting);
                        V (conditionSemaphor [op]);
                        P (urgentSemaphor);
                        DEC (urgentWaiting)
                 END
             END
      END
    END;
    CASE semantic OF
      exit, continue, wait:
         V (monitorEntry) |
      urgentWait:
         IF urgentWaiting > 0
           THEN V (urgentSemaphor)
           ELSE V (monitorEntry)
         END
    END
  END
END operate;

END Monitors.
```

The details of the implementation—particularly with respect to the different signal semantics—are described in [A] or [M].

Applications of the Universal Monitor

We now give some examples as applications for the universal monitor in order to demonstrate the universality of this concept.
First of all, there is the simple solution of our introductory example, the first readers and writers problem. The implementor just has to write code to realize the table and to determine the semantics and then can leave all synchronizing details to the universal monitor:

```
IMPLEMENTATION MODULE RW1;

  FROM SYSTEM IMPORT ADDRESS; FROM Storage IMPORT ALLOCATE;
IMPORT Monitors;
```

```
TYPE
  operations = (RIN, ROUT, WIN, WOUT);
  criticalSections = POINTER TO RECORD
                       activeReaders, activeWriters: CARDINAL;
                       monitor: Monitors.monitors
                     END;

  PROCEDURE c (x: ADDRESS; op: CARDINAL): BOOLEAN;
  VAR cs: criticalSections;
  BEGIN
    cs:= x;
    WITH cs^ DO
      CASE VAL (operations, op) OF
        RIN:  RETURN activeWriters = 0 |
        WIN:  RETURN (activeReaders = 0) & (activeWriters = 0) |
        ROUT, WOUT: RETURN TRUE
      END
    END
  END c;

  PROCEDURE a (x: ADDRESS; op: CARDINAL; adr: ADDRESS);
  VAR cs: criticalSections;
  BEGIN
    cs:= x;
    WITH cs^ DO
      CASE VAL (operations, op) OF
        RIN:  INC (activeReaders) |
        ROUT: DEC (activeReaders) |
        WIN:  activeWriters:= 1 |
        WOUT: activeWriters:= 0
      END
    END
  END a;

PROCEDURE initialize (VAR criticalSection: criticalSections);
BEGIN
  NEW (criticalSection);
  WITH criticalSection^ DO
    activeReaders:= 0; activeWriters:= 0;
    Monitors.initialize (monitor, Monitors.urgentWait,
                                  criticalSection, 4, c, a)
  END
END initialize;

PROCEDURE readerIn (cs: criticalSections);
BEGIN
  Monitors.operate (cs^.monitor, ORD (RIN), NIL)
END readerIn;

PROCEDURE readerOut (cs: criticalSections);
BEGIN
  Monitors.operate (cs^.monitor, ORD (ROUT), NIL)
END readerOut;

PROCEDURE writerIn (cs: criticalSections);
BEGIN
  Monitors.operate (cs^.monitor, ORD (WIN), NIL)
END writerIn;
```

```
PROCEDURE writerOut (cs: criticalSections);
BEGIN
  Monitors.operate (cs^.monitor, ORD (WOUT), NIL)
END writerOut;

END RW1.
```

Second, we apply the universal monitor to another classical example, the concurrent bounded buffer problem. This example is conceptually different from our introductory problem, since there are data into and out of the buffer. Hence we make use of the third parameter in the procedure `Monitors.operate`. Bounded buffers are an abstract data type, generically specified by

```
DEFINITION MODULE Buffers;

FROM SYSTEM IMPORT ADDRESS;

CONST max = ...;

TYPE buffers; (* bounded multiprocessible buffers
                 of byte streams of length <= max. *)

PROCEDURE initialize (VAR b: buffers; k, n: CARDINAL);
(* pre:  b is not initialized. k <= max.
   post: b is initialized. b has capacity k and stream length n. *)

PROCEDURE insert (b: buffers; a: ADDRESS);
(* pre:  b is initialized.
   post: The stream starting at a of the stream length of b is inserted
         into b as last stream in b. *)

PROCEDURE remove (b: buffers; a: ADDRESS);
(* pre:  b is initialized.
   post: Starting at a there is the first stream of b of the stream length
         of b. This stream is removed from b. *)

END Buffers.
```

Here is the implementation using a universal monitor for each buffer:

```
IMPLEMENTATION MODULE Buffers;

  FROM SYSTEM IMPORT ADDRESS;
  FROM Storage IMPORT ALLOCATE;
  FROM Streams IMPORT memcpy;
IMPORT Monitors;

CONST
  INSERT = 0;
  REMOVE = 1;
TYPE
  places = [0..max-1];
  buffers = POINTER TO RECORD
              capacity, streamlength: CARDINAL;
              array: ARRAY places OF ADDRESS;
              in, out: places;
              buffered: [0..max];
              monitor: Monitors.monitors
            END;
```

```
  PROCEDURE c (x: ADDRESS; op: CARDINAL): BOOLEAN;
  VAR buffer: buffers;
  BEGIN
    buffer:= x;
    WITH buffer^ DO
      CASE op OF
        INSERT: RETURN buffered < capacity |
        REMOVE: RETURN buffered > 0
      END
    END
  END c;

  PROCEDURE a (x: ADDRESS; op: CARDINAL; address: ADDRESS);
  VAR buffer: buffers;
  BEGIN
    buffer:= x;
    WITH buffer^ DO
      CASE op OF
        INSERT: memcpy (address, array [in], streamlength);
                in:= (in + 1) MOD capacity;
                INC (buffered) |
        REMOVE: memcpy (array [out], address, streamlength);
                out:= (out + 1) MOD capacity;
                DEC (buffered)
      END
    END
  END a;

PROCEDURE initialize (VAR buffer: buffers; k, n: CARDINAL);
VAR p: places;
BEGIN
  NEW (buffer);
  WITH buffer^ DO
    IF k > max THEN HALT END;
    capacity:= k;
    streamlength:= n;
    FOR p:= 0 TO capacity - 1 DO ALLOCATE (array [p], streamlength) END;
    in:= 0; out:= 0; buffered:= 0;
    Monitors.initialize (monitor, Monitors.continue, buffer, 2, c, a)
  END
END initialize;

PROCEDURE insert (buffer: buffers; address: ADDRESS);
BEGIN
  Monitors.operate (buffer^.monitor, INSERT, address)
END insert;

PROCEDURE remove (buffer: buffers; address: ADDRESS);
BEGIN
  Monitors.operate (buffer^.monitor, REMOVE, address)
END remove;

END Buffers.
```

The problem of the sleeping barber can be solved by this method as well. One simply has to implement the table

operation	condition	action
`getNextCustomer`	`numberClients > 0`	`DEC (numberClients);` `barberAvailable:= FALSE`
`finishedCut`	`TRUE`	`barberAvailable:= TRUE`
`getHaircut`	`barberAvailable`	`INC (numberClients)`

with the initial values `numberClients = 0` and `barberAvailable = TRUE`.
Of course we are also able to reconstruct semaphores along this line, although that might be only of academic interest:

```
IMPLEMENTATION MODULE Semaphores;

  FROM SYSTEM IMPORT ADDRESS; FROM Storage IMPORT ALLOCATE;
IMPORT Monitors;

TYPE semaphores = POINTER TO RECORD
                    value: CARDINAL;
                    monitor: Monitors.monitors
                  END;

  PROCEDURE c (x: ADDRESS; op: CARDINAL): BOOLEAN;
  VAR semaphor: semaphores;
  BEGIN
    semaphor:= x;
    WITH semaphor^ DO
      IF op = 0 THEN RETURN value > 0 ELSE RETURN TRUE END
    END
  END c;

  PROCEDURE a (x: ADDRESS; op: CARDINAL; adr: ADDRESS);
  VAR semaphor: semaphores;
  BEGIN
    semaphor:= x;
    WITH semaphor^ DO
      IF op = 0 THEN DEC (value) ELSE INC (value) END
    END
  END a;

PROCEDURE I (VAR semaphor: semaphores; n: CARDINAL);
BEGIN
  NEW (semaphor);
  WITH semaphor^ DO
    value:= n;
    Monitors.initialize (monitor, Monitors.urgentWait, semaphor, 2, c, a)
  END
END I;

PROCEDURE P (semaphor: semaphores);
BEGIN Monitors.operate (semaphor^.monitor, 0, NIL) END P;

PROCEDURE V (semaphor: semaphores);
BEGIN Monitors.operate (semaphor^.monitor, 1, NIL) END V;

END Semaphores.
```

A Universal Semaphore Baton

In a similar way, the technique of passing the baton with semaphores can be constructed and universally implemented. The details of this concept are explained in [A]. Here is the specification:

```
DEFINITION MODULE Batons;

FROM SYSTEM IMPORT ADDRESS;

CONST max = ...;
TYPE conditions = PROCEDURE (ADDRESS, CARDINAL, ARRAY OF CARDINAL): BOOLEAN;
     actions = PROCEDURE (ADDRESS, CARDINAL);
     batons;

PROCEDURE initialize (VAR b: batons; x: ADDRESS; n: CARDINAL;
                      c: conditions; e, l: actions);
(* pre:  b is not initialized.
   post: b is initialized. b has the object x, the number of operations n,
         the condition c and the actions e on enter and l on leave. *)

PROCEDURE enter (b: batons; op: CARDINAL);
(* pre:  b is initialized. op < number of operations of b.
         The calling process is not active in operation op in cs.
   post: The calling process is blocked on operation op in cs, iff its con-
         dition is not true, otherwise it is active in operation op in cs. *)

PROCEDURE leave (b: batons; op: CARDINAL);
(* pre:  b is initialized. op < number of operations of b.
         The calling process is active in operation op in cs.
   post: The calling process is no more active in operation op in cs.
         If there are blocked processes on any operation in cs, whose
         condition is true, one of them is unblocked and is active
         in operation op in cs. *)

END Batons.
```

And here comes the implementation:

```
IMPLEMENTATION MODULE Batons;

FROM SYSTEM IMPORT ADDRESS; FROM Storage IMPORT ALLOCATE;
FROM Semaphores IMPORT semaphores, I, P, V;

TYPE
  operations = [0..max-1];
  batons = POINTER TO RECORD
             object: ADDRESS;
             numberOps: CARDINAL;
             mutex: semaphores;
             semaphor: ARRAY operations OF semaphores;
             waiting: ARRAY operations OF CARDINAL;
             isTrue: conditions;
             executeOnEntry, executeOnLeave: actions
           END;

PROCEDURE initialize (VAR baton: batons; x: ADDRESS; n: CARDINAL;
                      c: conditions; e, l: actions);
VAR op: operations;
```

```
BEGIN
  IF n >= max THEN RETURN END;
  NEW (baton);
  WITH baton^ DO
    object:= x;
    numberOps:= n;
    I (mutex, 1);
    FOR op:= 0 TO n - 1 DO
      I (semaphor [op], 0);
      waiting [op]:= 0
    END;
    isTrue:= c;
    executeOnEntry:= e;
    executeOnLeave:= l
  END
END initialize;

  PROCEDURE Vall (baton: batons);
  VAR op: operations;
  BEGIN
    WITH baton^ DO
      FOR op:= 0 TO numberOps - 1 DO
        IF isTrue (object, op, waiting) & (waiting [op] > 0)
          THEN DEC (waiting [op]);
               V (semaphor [op]);
               RETURN
        END
      END;
      V (mutex)
    END
  END Vall;

PROCEDURE enter (baton: batons; op: CARDINAL);
BEGIN
  WITH baton^ DO
    IF op >= numberOps THEN RETURN END;
    P (mutex);
    IF NOT isTrue (object, op, waiting)
      THEN INC (waiting [op]);
           V (mutex);
           P (semaphor [op])
    END;
    executeOnEntry (object, op)
  END;
  Vall (baton)
END enter;

PROCEDURE leave (baton: batons; op: CARDINAL);
BEGIN
  WITH baton^ DO
    IF op >= numberOps THEN RETURN END;
    P (mutex);
    executeOnLeave (object, op)
  END;
  Vall (baton)
END leave;

END Batons.
```

Applications of the Universal Baton

Let us show how to solve the first readers and writers problem with the help of the universal baton:

```
IMPLEMENTATION MODULE RW1;
  FROM SYSTEM IMPORT ADDRESS; FROM Storage IMPORT ALLOCATE;
IMPORT Batons;

TYPE operations = (READ, WRITE);
     criticalSections = POINTER TO RECORD
                          active: ARRAY operations OF CARDINAL;
                          baton: Batons.batons
                        END;

  PROCEDURE c (x: ADDRESS; op: CARDINAL; dummy: ARRAY OF CARDINAL): BOOLEAN;
  VAR cs: criticalSections;
  BEGIN
    cs:= x;
    WITH cs^ DO
      CASE VAL (operations, op) OF
        READ:  RETURN (active [WRITE] = 0) |
        WRITE: RETURN (active [READ] = 0) & (active [WRITE] = 0)
      END
    END
  END c;

  PROCEDURE e (x: ADDRESS; op: CARDINAL);
  VAR cs: criticalSections;
  BEGIN
    cs:= x;
    WITH cs^ DO
      CASE VAL (operations, op) OF
        READ:  INC (active [READ]) |
        WRITE: active [WRITE]:= 1
      END
    END
  END e;

  PROCEDURE l (x: ADDRESS; op: CARDINAL);
  VAR cs: criticalSections;
  BEGIN
    cs:= x;
    WITH cs^ DO
      CASE VAL (operations, op) OF
        READ:  DEC (active [READ]) |
        WRITE: active [WRITE]:= 0
      END
    END
  END l;

PROCEDURE initialize (VAR criticalSection: criticalSections);
BEGIN
  NEW (criticalSection);
  WITH criticalSection^ DO
    active [READ]:= 0; active [WRITE]:= 0;
    Batons.initialize (baton, criticalSection, 2, c, e, l)
  END
END initialize;
```

```
PROCEDURE readerIn (cs: criticalSections);
BEGIN Batons.enter (cs^.baton, ORD (READ)) END readerIn;

PROCEDURE readerOut (cs: criticalSections);
BEGIN Batons.leave (cs^.baton, ORD (READ)) END readerOut;

PROCEDURE writerIn (cs: criticalSections);
BEGIN Batons.enter (cs^.baton, ORD (WRITE)) END writerIn;

PROCEDURE writerOut (cs: criticalSections);
BEGIN Batons.leave (cs^.baton, ORD (WRITE)) END writerOut;

END RW1.
```

This can easily be modified to a solution of the second readers and writers problem (writers preference), which shows the idea behind the introduction of the third parameter in the type `conditions`; one simply has to replace the procedure `c` by

```
  PROCEDURE c (x: ADDRESS; op: CARDINAL;
               waiting: ARRAY OF CARDINAL): BOOLEAN;
  VAR cs: criticalSections;
  BEGIN
    cs:= x;
    WITH cs^ DO
      CASE VAL (operations, op) OF
        READ:  RETURN (active [WRITE] = 0) & (waiting [ORD (WRITE)] = 0)
        WRITE: RETURN (active [READ] = 0) & (active [WRITE] = 0)
      END
    END
  END c;
```

The comparison of this very clearly understandable solution with the original solution of Courtois, Heymans and Parnas [CHP] certainly shows the strength of the concept.
It is simple to generalize this to even more fair variants of the readers and writers problem. As a more general example we show the solution of the left-right-problem (e.g. traffic over a one track road or use of a single sex bathroom by females and males), where either members of one of the two involved classes or of the other may use the resource. Members of either class are granted access to the resource, because members of one class can use the resource only a certain amount of times. This maximum is defined by the client:

```
IMPLEMENTATION MODULE LR;

  FROM SYSTEM IMPORT ADDRESS; FROM Storage IMPORT ALLOCATE;
IMPORT Batons;

TYPE
  operations = (LEFT, RIGHT);
  criticalSections = POINTER TO RECORD
                       maximum, active, done: ARRAY operations OF CARDINAL;
                       baton: Batons.batons
                     END;

  PROCEDURE c (x: ADDRESS; op: CARDINAL;
               waiting: ARRAY OF CARDINAL): BOOLEAN;
  VAR cs: criticalSections;
```

```
  BEGIN
    cs:= x;
    WITH cs^ DO
      CASE VAL (operations, op) OF
        LEFT:  RETURN (active [RIGHT] = 0)
                    & ((waiting [ORD (RIGHT)] = 0) OR
                       (done [LEFT] < maximum [LEFT])) |
        RIGHT: RETURN (active [LEFT] = 0)
                    & ((waiting [ORD (LEFT)] = 0) OR
                       (done [RIGHT] < maximum [RIGHT]))
      END
    END
  END c;

  PROCEDURE e (x: ADDRESS; op: CARDINAL);
  VAR cs: criticalSections;
  BEGIN
    cs:= x;
    WITH cs^ DO
      CASE VAL (operations, op) OF
        LEFT:  INC (active [LEFT]);
               INC (done [LEFT]);
               done [RIGHT]:= 0 |
        RIGHT: INC (active [RIGHT]);
               INC (done [RIGHT]);
               done [LEFT]:= 0
      END
    END
  END e;

  PROCEDURE l (x: ADDRESS; op: CARDINAL);
  VAR cs: criticalSections;
  BEGIN
    cs:= x;
    WITH cs^ DO
      CASE VAL (operations, op) OF
        LEFT:  DEC (active [LEFT]) |
        RIGHT: DEC (active [RIGHT])
      END
    END
  END l;

PROCEDURE initialize (VAR criticalSection: criticalSections;
                      max: ARRAY OF CARDINAL);
BEGIN
  NEW (criticalSection);
  WITH criticalSection^ DO
    maximum [LEFT]:= max [ORD (LEFT)]; maximum [RIGHT]:= max [ORD (RIGHT)];
    active [LEFT]:= 0; active [RIGHT]:= 0;
    done [LEFT]:= 0; done [RIGHT]:= 0;
    Batons.initialize (baton, criticalSection, 2, c, e, l)
  END
END initialize;

PROCEDURE leftIn (cs: criticalSections);
BEGIN
  Batons.enter (cs^.baton, ORD (LEFT))
END leftIn;
```

```
PROCEDURE leftOut (cs: criticalSections);
BEGIN
  Batons.leave (cs^.baton, ORD (LEFT))
END leftOut;

PROCEDURE rightIn (cs: criticalSections);
BEGIN
  Batons.enter (cs^.baton, ORD (RIGHT))
END rightIn;

PROCEDURE rightOut (cs: criticalSections);
BEGIN
  Batons.leave (cs^.baton, ORD (RIGHT))
END rightOut;

END LR.
```

Appendix

Semaphores are fundamental for our implementations. They can be implemented using POSIX 1003.1b semaphores as follows (`man 3 sem_init` for details):

```
IMPLEMENTATION MODULE Semaphores;

FROM Storage IMPORT ALLOCATE;
FROM libpthreads IMPORT sem_init, sem_wait, sem_post;

TYPE semaphores = POINTER TO CHAR;

PROCEDURE I (VAR semaphor: semaphores; n: CARDINAL);
BEGIN
  ALLOCATE (semaphor, 16);
  sem_init (semaphor, 0, n)
END I;

PROCEDURE P (semaphor: semaphores);
BEGIN
  sem_wait (semaphor)
END P;

PROCEDURE V (semaphor: semaphores);
BEGIN
  sem_post (semaphor)
END V;

END Semaphores.
```

With this module, all given examples can be tested using the Modula-2-Compiler of the GMD, Karlsruhe (URL: `i44www.info.uni-karlsruhe.de/~modula/`).
Please note, that you have to inform that compiler about the stack organization for calls of C-routines and to use the `pthread`-library in the link process; for that you have to add the undocumented option `-CcallsMocka` in the shell script `mocka` and the option `-lpthreads` in the shell script `link` of that compiler (both in the subdirectory `sys`).

References

[A0] Andrews, G. R., *Synchronizing Resources*, reprinted in [GM], ACM Trans. Prog. Lang. Syst. **3** (1981), 405–430.

[A] Andrews, G. R., *Concurrent Programming*, Principles and Practice, Addison-Wesley Publishing Company, 1991.

[CHP] Courtois, P. J., Heymans, F., Parnas, D. L., *Concurrent Control with "Readers" and "Writers"*, Commun. ACM **14** (1971), 667–668.

[D] Dijkstra, E. W., *Co-operating Sequential Processes*, reprinted in [G], 1965, pp. 43–112.

[G] Genuys, F. (ed.), *Programming Languages*, Academic Press, 1968.

[GM] Gehani, N., McGattrick, A. D. (eds.), *Concurrent Programming*, Addison-Wesley Publishing Company, 1988.

[H] Hoare, C. A. R., *Monitors: An Operating Systems Structuring Concept*, reprinted in [GM], Commun. ACM **17** (1974), 549–557.

[M] Maurer, Ch., *Grundzüge der Nichtsequentiellen Programmierung*, Springer, 1999.

Institut für Informatik, Freie Universität Berlin, Takustr. 9, D-14195 Berlin

E-mail address: `maurer@inf.fu-berlin.de`

URL: `www.inf.fu-berlin.de/~maurer/`