HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports ISSN 0783–5396
No. **53**; August 1998 ISBN 951-22-4251-6

SEMANTICS OF SEMAPHORES

Stefan Rönn

Swedish School of Economics
Department of Statistics and Computer Science
PB 479, 00101 Helsinki, FINLAND

Helsinki University of Technology
Department of Computer Science and Engineering
Digital Systems Laboratory
P.O.Box 1100, FIN–02015 HUT, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports ISSN 0783–5396
No. **53**; August 1998 ISBN 951-22-4251-6

Semantics of Semaphores

STEFAN RÖNN

Abstract: We present a formal definition of semaphores using weakest preconditions. With the definition we prove the mutual exclusion algorithm and show that the semaphore axioms given by Martin and de Snepscheut are implied by the invariant relation of the proof. We further demonstrate how a general k-semaphore can be realised with the help of a split binary semaphore. To this end, we introduce a program construct that facilitates the usage of the split binary semaphore.

Keywords: semaphores, process synchronization, mutual exclusion, weakest preconditions

Printing: Picaset Oy; Helsinki 1998

Helsinki University of Technology
Department of Computer Science and Engineering
Digital Systems Laboratory
P.O.Box 1100
FIN–02015 HUT, FINLAND

Phone: $\frac{(09)}{+358\ 9}$ 4511
Telex: 125 161 htkk fi
Telefax: +358 9 451 3369
E-mail: lab@saturn.tcs.hut.fi

1 Introduction

Semaphores are well-known synchronisation primitives for parallel programs where communication between processes takes place via shared variables [1]. Martin and van de Snepscheut have defined the semantics of semaphores by four axioms, expressed as equalities and inequalities that the variables characterising the semaphore state have to satisfy [3]. The axioms are applicable to the constructive development of many classical synchronisation problems. In this treatise we take a different approach and rely on weakest preconditions to define the semantics. Our definition represents a refinement of Martin's and van de Snepscheut's semantics, since it enables us to derive their axioms as consequences of the solution of the mutual exclusion problem. It also provides some guidelines for how to improve the heuristics of writing synchronisation algorithms with the split binary semaphore.

We consider a concurrent program consisting of an unspecified number of processes, each of the form:

process i :

do $true \rightarrow$
  $NCS.i$;
  X; (1)
  $CS.i$;
  Y
od

Process i communicates with the other processes via shared variables, which it accesses in the critical section $CS.i$. The non-critical section $NCS.i$ contains no such accesses. X and Y are synchronisation primitives, which have the same form in all processes. Their joint task is to ensure that at most k, $k \geq 1$, prosesses are engaged in their critical sections at any time. With the notation $\#S$ standing for the number of times that a statement S has been performed, the above synchronisation requirement is expressed by $\#X - \#Y \leq k$. ¿From the order of execution of X and Y in each process ("the topology of the program") one deduces the validity of $\#X - \#Y \geq 0$, too. The total synchronisation requirement is therefore

(2) $$0 \leq \#X - \#Y \leq k$$

This inequality suggests that one implements X and Y as operations on an integer variable, termed *semaphore*, whose values are in the interval $[0, k]$. We designate the semaphore by s, which thus should be an argument of X and Y. For historical reasons we write $P.s$ and $V.s$ instead of $X.s$ and $Y.s$.

The execution of $P.s$ decreases s by one, the execution of $V.s$ increases s by one. If the initial value of s is k, the topology of program text (1) by itself maintains the validity of $s \leq k$. To quarantee the validity of $s \geq 0$, too, a process starting to execute $P.s$ must be suspended when $s = 0$. Several processes can be suspended this way; they may then be thought of as forming a queue before $P.s$. We let $Q.s$ stand for the length of the queue.

Summing up, we have the following variables, which we shall use to characterise the state of a semaphore:

s denotes the semaphore, i.e. an integer variable that satisfies $0 \leq s \leq k$, where $k \geq 1$.

$\#P.s$ is the number of times $P.s$ has been executed by the processes.

$\#V.s$ is the number of times $V.s$ has been executed by the processes.

$Q.s$ is the number of processes that are suspended at a $P.s$ – operation when $s = 0$.

The first three of Martin's and van de Snepscheut's axioms now look like:

A1. $s \geq 0$

A2. $s = s_0 + \#V.s - \#P.s$

A3. $s = 0 \vee Q.s = 0$

In A2, s_0 is the initial value of s. A1 is one half of the inequality $0 \leq s \leq k$. The other half, $s \leq k$, is implied by A2 and the topology of program text (1), provided $s_0 = k$. Axiom A3, which could also be written $Q.s > 0 \Rightarrow s = 0$, states that if there are processes suspended at a $P.s$ – operation, $s = 0$ holds. The fourth axiom of the aforementioned authors states that there are never any processes suspended at a $V.s$ – operation. Here, we take this for granted without formulating a separate axiom.

2 Wp–definition of semaphores

We continue to study the concurrent program that consists of processes of the form (1). With the notation just introduced we can further refine the program text:

Initially: $s = k \wedge \#P.s = 0 \wedge \#V.s = 0 \wedge Q.s = 0$

process i:

do $true \rightarrow$
    $NCS.i$;
    $P.s$;   (1')
    $CS.i$;
    $V.s$
od

A process is only allowed to pass $P.s$ if $s{>}0$, in which case it decreases s and increases $\#P.s$ by one. If $s = 0$, on the other hand, the process should be suspended at $P.s$, and $Q.s$ should be increased by one. Hence, the $P.s$–operation corresponds to the following program fragment:

$P.s$:

if $s > 0 \ \rightarrow \ s, \#P.s := s - 1, \#P.s + 1$ (3)
$\square\ s = 0 \ \rightarrow \ Q.s := Q.s + 1$
fi

A process is always allowed to pass the $V.s$–operation. If $s{\geq}0$ and there are no processes suspended at $P.s$, s and $\#V.s$ are both increased by one. If $s = 0$ and there are suspended processes, one of these is allowed to pass $P.s$, consequently $\#V.s$ as well as $\#P.s$ should be increased by one and $Q.s$ should be decreased by one. For $V.s$ we therefore obtain the code:

$V.s$:

if $s{\geq}0 \wedge Q.s = 0 \ \rightarrow \ s, \#V.s := s + 1, \#V.s + 1$ (4)
$\square\ s = 0 \wedge Q.s > 0 \ \rightarrow \ \#V.s, Q.s, \#P.s := \#V.s + 1, Q.s - 1, \#P.s + 1$
fi

Both $P.s$ and $V.s$ are alternative constructs of the form:

IF:

if $C1 \ \rightarrow \ x1 := E1$
$\square\ C2 \ \rightarrow \ x2 := E2$
fi

The notation $x := E$ stands for a simple assignment, such as $Q.s := Q.s + 1$, or a multiple assignment, such as $s, \#P.s := s - 1, \#P.s + 1$. The weakest precondition of IF with respect to an arbitrary predicate R is:

(5) $$wp(IF; R) \equiv (C1 \vee C2) \wedge (C1 \Rightarrow R(x1 := E1)) \wedge (C2 \Rightarrow R(x2 := E2))$$

The notation $R(x := E)$ designates the predicate one gets when replacing in R all occurrences of each variable in the variable list x by the corresponding expression in the expression list E.

By evaluating (5) for program fragment (3) we obtain for $P.s$ the wp-definition:

(6) $$wp(P.s; R) \equiv F1 \wedge F2 \wedge F3$$

where

$$F1 \equiv s \geq 0$$

$$F2 \equiv s > 0 \Rightarrow R(s, \#P.s := s - 1, \#P.s + 1)$$

$$F3 \equiv s = 0 \Rightarrow R(Q.s := Q.s + 1)$$

Evaluation of (5) for program fragment (4), in turn, yields the wp-definition of $V.s$:

(7) $$wp(V.s; R) \equiv G1 \wedge G2 \wedge G3$$

where

$$G1 \equiv (s \geq 0 \wedge Q.s = 0) \vee (s = 0 \wedge Q.s > 0)$$

$$G2 \equiv (s \geq 0 \wedge Q.s = 0) \Rightarrow R(s, \#V.s := s + 1, \#V.s + 1)$$

$$G3 \equiv (s = 0 \wedge Q.s > 0) \Rightarrow R(\#V.s, Q.s, \#P.s := \#V.s + 1, Q.s - 1, \#P.s + 1)$$

When proving the correctness of a concurrent algorithm we shall rely on the so-called *Gries-Owicki-theory* [4]. This implies that the proof is centred on an invariant relation I, which is formulated as a predicate on the total state space. The total state space is made up of all possible values of all shared and local variables of the algorithm. To validate I we have to show, first, that it holds in the initial state, and, second, that it is maintained by each atomic action of the algorithm. With regard to the synchronisation primitives $P.s$ and $V.s$, in particular, we must establish:

(8a) $$I \wedge D \Rightarrow wp(P.s; I \wedge E)$$

(8b) $$I \wedge F \Rightarrow wp(V.s; I \wedge G)$$

D, E, F, and G are additional predicates, of which E and G are mostly deduced from the topology of the program, while D and F are to be chosen so that the formulas become valid. Once the correctness of the invariant has been established, one can use it to verify safety and liveness properties of the algorithm.

3 Proof of the mutual exclusion algorithm

If we let the initial value k $(= s_0)$ of s in (1') be 1, i.e. we choose a *binary* semaphore, we obtain the standard algorithm for implementing mutual exclusion between concurrent processes. As an example of how to use our definition of semaphores and due to the importance of the algorithm we shall prove its correctness in accordance with the Gries-Owicki theory.

As invariant we choose

(9) $$I \;\equiv\; I1 \wedge I2$$

where

$$I1 \;\equiv\; s = 1 + \#V.s - \#P.s$$

$$I2 \;\equiv\; (0 \leq s \leq 1 \wedge Q.s = 0) \vee (s = 0 \wedge Q.s > 0)$$

The invariant holds for the initial values of the variables in (1'). We further observe that Martin's and van de Snepscheut's axioms A1 and A3 are implied by I2, while A2 for $s_0 = 1$ is implied by I1. Hence, for a binary semaphore we have confirmed our previous claim that axioms A1–A3 are consequences of the proof of the mutual exclusion algorithm.

To be able to apply definitions (6) and (7) to the invariant $I1 \wedge I2$ we need to evaluate $I1(x := E)$ and $I2(x := E)$ for the four assignments of the definitions. We obtain:

(10a) $$I1(s, \#P.s := s-1, \#P.s+1) \;\equiv\; I1$$

(10b) $$I2(s, \#P.s := s-1, \#P.s+1) \;\equiv\; (1 \leq s \leq 2 \wedge Q.s = 0) \vee (s = 1 \wedge Q.s > 0)$$

(10c) $$I1(Q.s := Q.s+1) \;\equiv\; I1$$

(10d) $$I2(Q.s := Q.s+1) \;\equiv\; (0 \leq s \leq 1 \wedge Q.s = -1) \vee (s = 0 \wedge Q.s \geq 0)$$

(10e) $$I1(s, \#V.s := s+1, \#V.s+1) \;\equiv\; I1$$

(10f) $I2(s,\#V.s := s+1, \#V.s+1) \equiv (-1{\le}s{\le}0 \wedge Q.s = 0) \vee$
$(s = -1 \wedge Q.s > 0)$

(10g) $I1(\#V.s, Q.s, \#P.s := \#V.s+1, Q.s-1, \#P.s+1) \equiv I1$

(10h) $I2(\#V.s, Q.s, \#P.s := \#V.s+1, Q.s-1, \#P.s+1) \equiv$
$(0{\le}s{\le}1 \wedge Q.s = 1) \vee (s = 0 \wedge Q.s > 1)$

When process i has just executed $P.s$ in (1') we can due to the topology of the program and assuming that mutual exclusion is realised assert $\#P.s = \#V.s+1$. In order to prove that $I1 \wedge I2$ is an invariant we thus have to show the validity of:

(11) $I1 \wedge I2 \wedge D \;\Rightarrow\; wp(P.s; I1 \wedge I2 \wedge \#P.s = \#V.s+1)$

where D is to be chosen appropriately. We calculate:

$$
\begin{array}{ll}
 & wp(P.s; I1 \;\wedge\; I2 \;\wedge\; \#P.s = \#V.s+1) \\
\equiv & \quad \{(6)\} \\
 & s{\ge}0 \;\wedge \\
 & (s>0 \;\Rightarrow\; I1(s,\#P.s := s-1, \#P.s+1) \\
 & \qquad\qquad \wedge\; I2(s,\#P.s := s-1, \#P.s+1) \wedge \\
 & \qquad\qquad \#P.s+1 = \#V.s+1) \;\wedge \\
 & (s=0 \;\Rightarrow\; I1(Q.s := Q.s+1) \;\wedge\; I2(Q.s := Q.s+1) \wedge \\
 & \qquad\qquad \#P.s = \#V.s+1) \\
\equiv & \quad \{(10a)\text{–}(10d), \text{arithmetic}\} \\
 & s{\ge}0 \;\wedge \\
 & (s>0 \;\Rightarrow\; I1 \;\wedge\; ((1{\le}s{\le}2 \;\wedge\; Q.s=0) \vee\; (s=1 \;\wedge\; Q.s>0)) \wedge \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \#P.s = \#V.s) \;\wedge \\
 & (s=0 \;\Rightarrow\; I1 \;\wedge\; ((0{\le}s{\le}1 \wedge Q.s=-1) \;\vee\; (s=0 \wedge Q.s{\ge}0)) \wedge \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \#P.s = \#V.s+1) \\
\equiv & \quad \{I1 \;\wedge\; \#P.s = \#V.s \;\Rightarrow\; s=1\;; \\
 & \quad\;\; I1 \;\wedge\; \#P.s = \#V.s+1 \;\Rightarrow\; s=0\} \\
 & s{\ge}0 \;\wedge \\
 & (s>0 \;\Rightarrow\; I1 \wedge ((s=1 \wedge Q.s=0) \vee\; (s=1 \wedge Q.s>0)) \wedge \\
 & \qquad\qquad\qquad\qquad\qquad \#P.s = \#V.s) \;\wedge \\
 & (s=0 \;\Rightarrow\; I1 \wedge ((s=0 \wedge Q.s=-1) \;\vee\; (s=0 \wedge Q.s{\ge}0)) \wedge \\
 & \qquad\qquad\qquad\qquad\qquad \#P.s = \#V.s+1) \\
\equiv & \quad \{\text{predicate calculus}\} \\
(12) & s{\ge}0 \;\wedge \\
 & (s>0 \;\Rightarrow\; I1 \;\wedge\; s=1 \;\wedge\; Q.s{\ge}0 \;\wedge\; \#P.s = \#V.s) \;\wedge \\
 & (s=0 \;\Rightarrow\; I1 \;\wedge\; s=0 \;\wedge\; Q.s{\ge}-1 \;\wedge\; \#P.s = \#V.s+1)
\end{array}
$$

Using (12) we are now able to validate (11) with two choices for D, viz.

(13a) $\quad I1 \wedge I2 \wedge \#P.s = \#V.s \Rightarrow wp(P.s; I1 \wedge I2 \wedge P.s = \#V.s + 1)$

(13b) $\quad I1 \wedge I2 \wedge \#P.s = \#V.s + 1 \Rightarrow$

$$wp(P.s; I1 \wedge I2 \wedge P.s = \#V.s + 1)$$

Taking into account that $I1 \wedge \#P.s = \#V.s \Rightarrow s = 1$, we see that (13a) is valid because of the first implication of (12); taking into account that $I1 \wedge \#P.s = \#V.s + 1 \Rightarrow s = 0$, we see that (13b) is valid because of the second implication of (12). The former choice for D models the situation when process i executes $P.s$ and is allowed to enter its critical section. It presupposes that no other process is engaged in its critical section. The latter choice is relevant when process i becomes suspended at the $P.s$–operation, in which case another process is already engaged in its critical section.

To show that $I1 \wedge I2$ remains invariant under the $V.s$–operation we have to prove

(14) $\quad I1 \wedge I2 \wedge E \Rightarrow wp(V.s; I1 \wedge I2)$

for an appropriate choice of E. We calculate:

$$\begin{array}{ll}
 & wp(P.s; I1 \wedge I2) \\
\equiv & \quad \{\text{definition (7)}\} \\
 & ((s{\ge}0 \wedge Q.s = 0) \vee (s = 0 \wedge Q.s > 0)) \wedge \\
 & (s{\ge}0 \wedge Q.s = 0 \Rightarrow I1(s, \#V.s := s+1, \#V.s+1) \\
 & \qquad\qquad \wedge\ I2(s, \#V.s := s+1, \#V.s+1)) \wedge \\
 & (s = 0 \wedge Q.s > 0 \Rightarrow \\
 & \quad I1(\#V.s, Q.s, \#P.s := \#V.s+1, Q.s-1, \#P.s+1) \\
 & \quad \wedge\ I2(\#V.s, Q.s, \#P.s := \#V.s+1, Q.s-1, \#P.s+1)) \\
\equiv & \quad \{(10e)\text{–}(10h)\} \\
 & ((s{\ge}0 \wedge Q.s = 0) \vee (s = 0 \wedge Q.s > 0)) \wedge \\
 & (s{\ge}0 \wedge Q.s = 0 \Rightarrow \\
 & \quad I1 \wedge ((-1{\le}s{\le}0 \wedge Q.s = 0) \vee (s = -1 \wedge Q.s > 0))) \wedge \\
 & (s = 0 \wedge Q.s > 0 \Rightarrow \\
 & \quad I1 \wedge ((0{\le}s{\le}1 \wedge Q.s = 1) \vee (s = 0 \wedge Q.s > 1))) \\
\equiv & \quad \{\text{predicate calculus, the Law of Absorption in particular}\} \\
(15) & ((s{\ge}0 \wedge Q.s = 0) \vee (s = 0 \wedge Q.s > 0)) \wedge \\
 & (s{\ge}0 \wedge Q.s = 0 \Rightarrow I1 \wedge -1{\le}s{\le}0 \wedge Q.s = 0) \wedge \\
 & (s = 0 \wedge Q.s > 0 \Rightarrow \\
 & \quad I1 \wedge ((0{\le}s{\le}1 \wedge Q.s = 1) \vee (s = 0 \wedge Q.s > 1)))
\end{array}$$

The choice $\#P.s = \#V.s + 1$ for E validates (14) on account of $I1 \wedge \#P.s = \#V.s + 1 \Rightarrow s = 0$. The first implication of (15) corresponds to the situation when there are no processes suspended at a $P.s$–operation, the second to the situation when there are such processes and the first in the queue is allowed to enter a critical engagement.

4 The usage of the split binary semaphore

The technique of the so-called split binary semaphore is important for many synchronisation problems [2]. It consists in using two binary semaphores, a and b say, with the property that the sum $a+b$ of the semaphore values is at any time at most one. A sequence of statements frequently occurring in this context is a V–operation performed on the one semaphore, immediately followed by a P–operation performed on the other. The program text of synchronisation problems can sometimes be simplified if one may assume that such a sequence is executed as one atomic action. We therefore introduce the *atomicity operator* & with the following informal semantics: for two statements X and Y, a process executes $X\&Y$ as a concatenation $X;Y$, and no other process interferes with the execution. To be able to use the &–operator to our advantage we assume that the queue length $Q.c$ for any semaphore c is directly accessible by the processes. (Nothing would prevent us from making the same assumption about $\#P.c$ and $\#V.c$, but we do not need it here.)

We now address the problem of implementing a general k–semaphore, i.e. a semaphore that may take the values from 0 to $k-1$, with the help of a split binary semaphore. More specifically, our task is to employ the aforementioned technique to implement the semaphore operations $\mathcal{P}.s$ and $\mathcal{V}.s$ such that $0 \le s \le k-1$. The solution is essentially to surround program fragments (3) and (4) with the operations $P.a$ and $V.a$, where a is a binary semaphore. s is regarded as a global variable, access to which is protected by a. The initial value of s is $k-1$. Processes executing the second alternative of (3) are suspended at $P.b$, where b is another binary semaphore. We thus obtain:

Initially: $s = k-1 \;\wedge\; a = 1 \;\wedge\; b = 0 \;\wedge\; Q.a = 0 \;\wedge\; Q.b = 0$

$\mathcal{P}.s$:

$P.a;$
$\textbf{if}\ s > 0 \;\rightarrow\; s := s-1; V.a$ (16)
$\Box\ s = 0 \;\rightarrow\; V.a \,\&\, P.b; V.a$
$\textbf{fi}$

$\mathcal{V}.s$:

$P.a$;
if $s{\geq}0 \wedge Q.b = 0 \;\rightarrow\; s := s + 1; V.a$ (17)
$\Box\; s = 0 \wedge Q.b > 0 \;\rightarrow\; V.b$
fi

The atomicity operator is indispensable in $V.a \,\&\, P.b$, because it ensures that a process is added to the queue of processes suspended at $P.b$ before another process is permitted to enter a critical section protected by semaphore a. This process then has access to the variable $Q.b$, which has taken over the role played by $Q.s$. Assignments to the ghost variables $\#P.s$ and $\#V.s$ are for the time being omitted.

The purpose of the $V.b$–operation in the second alternative of $\mathcal{V}.s$ is to dequeue one of the processes suspended at $P.b$ in the second alternative of $\mathcal{P}.s$. The action is considered finished only when this process has passed $P.b$, and therefore the operation $V.a$ has been added to the second alternative of $\mathcal{P}.s$ and not to the second alternative of $\mathcal{V}.s$.

If we replace $P.s$ and $V.s$ in (1') by $\mathcal{P}.s$ and $\mathcal{V}.s$, we want the resulting program to maintain, in analogy with (9):

(18) $$J \;\equiv\; J1 \wedge J2$$

where

$$J1 \;\equiv\; s = k - 1 + D - C$$

$$J2 \;\equiv\; (0{\leq}s{\leq}k - 1 \wedge Q.b = 0) \;\vee\; (s = 0 \wedge Q.b > 0)$$

Assignments to the ghost variables C and D are still to be inserted into $\mathcal{P}.s$ and $\mathcal{V}.s$. C corresponds to $\#P.s$ and D to $\#V.s$ in (3) and (4). They should therefore be increased by one when the first alternatives of (16) and (17) are executed. When a process executes the second alternative of (17), and another process is dequeued and allowed to pass $P.b$ in the second alternative of (16), both C and D should be increased by one. We prefer to do this in the second alternative of (17).

¿From the invariant $J2$ we also deduce the following formulas:

(19a) $$Q.b > 0 \;\Rightarrow\; s = 0$$

(19b) $$Q.b = 0 \;\Rightarrow\; s{\geq}0$$

They enable us to simplify the guards of $\mathcal{V}.s$. In fact, the above observations together yield the following code:

Initially: $s = k - 1 \wedge a = 1 \wedge b = 0 \wedge Q.a = 0 \wedge Q.b = 0 \wedge C = 0 \wedge D = 0$

$\mathcal{P}.s$:

$P.a$;
if $s > 0 \;\rightarrow\; s := s - 1 \,\&\, C := C + 1;\; V.a$ (16')
$\Box\; s = 0 \;\rightarrow\; V.a \,\&\, P.b;\; V.a$
fi

$\mathcal{V}.s$:

$P.a$;
if $Q.b = 0 \;\rightarrow\; s := s + 1 \,\&\, D := D + 1; V.a$ (17')
$\Box\; Q.b > 0 \;\rightarrow\; D := D + 1 \,\&\, C := C + 1; V.b$
fi

The appearance of the &-operator in connection with the updating of C and D guarantees that $J1$ is maintained as a consequence of the execution of atomic actions. The atomicity operator is particularly helpful in such a context, where ghost variables are updated.

Acknowledgments

I thank professors Leo Ojala and Tomi Janhunen for helpful comments.

References

[1] E.W. Dijkstra, Co-operating sequential processes, in: F. Genuys, ed., Programming Languages (Academic Press, London New York, 1968) 43–112.

[2] C.A.R. Hoare, Monitors: an operating system structuring concept, Comm. ACM 17, 10 (1974) 549–557.

[3] A.J. Martin and J.L.A. van de Snepscheut, Design of synchronisation algorithms, in: M. Broy ed., Constructive Methods in Computing Science (Springer, Berlin Heidelberg, 1989) 447–478.

[4] S. Owicki and D.Gries, An axiomatic proof technique for parallel programs, Acta Informatica 6 (1976) 319–340.