# Extended Semaphore Operations

S. Hodgson, N. Dunstan and I. Fris
School of Mathematical and Computer Science
University of New England

**Abstract**

Extended semaphores systems such as in UNIX System V are a powerful extension of Dijkstra's semaphores. They allow efficient solutions to a number of classic synchronisation problems. UNIX semaphore operations appear to be insufficiently well defined, in particular when a semaphore is repeated in an operator. This results in several solutions relying on assumed properties. A new semaphore operator, *isem*, based on extended semaphore operators is introduced. The operator *isem* is clearly defined, can be implemented efficiently, and yields simple solutions to many classic synchronisation problems.

## 1 Introduction

The semaphore was proposed as a simple if not the simplest and elegant synchronising mechanism for concurrent processes [7]. Despite the development of more and more elaborate synchronisation mechanisms and concurrent programming language constructs, semaphores continue to exist in many operating systems (such as OS2, UNIX and POSIX). Also, the semaphore properties can be defined very precisely making semaphores a useful primitive to define more complicated constructs.

The practical concerns of concurrent programming led many researchers [16, 6, 4] to propose various extensions of semaphore operations. Extended semaphore operations are intended to provide more efficient and transparent solutions to synchronisation problems and greater convenience for programmers, albeit at the expense of the simplicity and elegance of the original concept. The peak in the development of extended semaphore operations must surely be those of UNIX System V which allow for a number of generalised operations to be applied to an array of semaphores simultaneously. Although criticised for being too complex [11] the UNIX System V semaphore system has been shown to be a powerful mechanism for the scheduling of concurrent processes [13].

This paper presents further extension to the semaphore operator and continues the long history of research stemming from Dijkstra's original concept. This research is principally motivated by programming problems that arose using various versions of UNIX System V semaphore system calls [8]. In fact, the new semaphore operator should be considered as an improved version of the UNIX System V semaphore system. The new operator has greater functionality, a more precise definition and enables more efficient and clearer solutions to some concurrent process synchronisation problems. After a brief discussion of extended semaphore operations, the new semaphore operator is defined. The power of the new operator is illustrated by several highly efficient algorithms for solving some common concurrent processing synchronisation problems.

## 2 Semaphore Systems

### 2.1 Dijkstra's Semaphores

The concept of (general) semaphore was introduced by Dijkstra in 1968 in [7]. Semaphore $s$ keeps a non-negative integer variable $s.val$ and allows two atomic operations: $V(s)$ increases $s.val$ by 1, while $P(s)$ decreases it by 1, however $P$ can only terminate if $s.val > 0$. Operations $P$ and $V$ are atomic on the same semaphore, which means that they do not interleave. Each operation must terminate[1] before another one may start. When an operation — such as $P(s)$ on a semaphore whose value is 0 — cannot terminate, it is, at least conceptually, not even started. The effect of this is that the process executing such an operation is suspended. In this paper we are not interested with the details of how such a suspension is implemented or how it is lifted. All these details are hidden in the word "when" in the description in Figure 1.

```
P(s):
    when $s > 0$
    $s \leftarrow s - 1$

V(s):
    $s \leftarrow s + 1$
```

Figure 1: Dijkstra Semaphores Operations

It is generally accepted that semaphores are low level primitives and thus not very suitable for programming [19]. Nevertheless semaphores are useful for implementation of higher level constructs, they can be used in programming languages (such as C) that do not have any parallel constructs built in, and are also useful for multi-language programming. When fully defined, semaphores can be useful in defining other more complex constructs for parallel programming. Finally, semaphores are of considerable theoretical interest.

[1]Instead of "terminate" we equivalently say "succeed"

## 2.2 Specialised Semaphores

Synchronisation problems other than critical regions have typically very complex solutions which are difficult to understand. Consequently, a number of specialised forms of semaphores have been proposed. The semaphore solution by Courtois et al to the readers and writers problem prompted `PP` and `VV` operations [5] which are essentially the entry and exit protocols for reader processes. The idea was extended to *Reader/Writer* semaphores [3] with operations `readP`, `readV`, `writeP` and `writeV`. Which in turn led to *Priority* semaphores [4] which generalise the readers and writers problem to one of any number of classes of processes with different priority levels. Also preemption by higher priority classes may take place. A further specialised form manages sets of identical resources. *Resource Set* semaphores [10] are initialised with the number of resources in an identical set. The `rsetP` operation causes the calling process to be suspended until a resource becomes available. The process returns the unique identification of the resource allocated to it. The `rsetV` operation releases a resource.

## 2.3 Notation for Extended Semaphore Operations

There have been many extensions to Dijkstra's P/V primitives e.g. [16, 18]. We are concerned with three extensions, essentially those which comprise the UNIX System V semaphore operations. To describe those extensions we will use the operator (function) $\Phi(s, m)$. In the operator $\Phi$, $s$ is a (generalised) semaphore and $m$ is an integer which extends the standard $P/V$ operators as follows :–

- Incrementing or decrementing a semaphore by values other than 1.
  When $m \neq 0$, $\Phi(s, m)$ alters the value of $s$ by adding $m$. The operation succeeds if and only if the new value of $s$ is $\geq 0$.

  Thus $\Phi$(s, 1) is V(s), while $\Phi(s, -1)$ is $P(s)$

- Blocking until the semaphore value equals 0.
  This operation is denoted as $\Phi$(s, 0). When the value of $s$ equals 0, the operation $\Phi(s, 0)$ succeeds and does nothing, the process issuing the operation continues normally, otherwise the executing process is suspended until the value of $s$ becomes 0. Thus we can describe it in Figure 2.

```
$\Phi(s, 0)$ :
  when $s = 0$
    do nothing
```

Figure 2:

- Operating on $s_1, \ldots, s_k$ semaphores simultaneously (in parallel) as an atomic operation.

$$\Phi(s_1, m_1||...||s_k, m_k) \qquad (1)$$

The semaphore operator in (1) succeeds iff all the component operations $\Phi(s_i, m_i)$ succeed. The process calling the function suspends until such time when all the components succeed. By atomicity we mean that either all or none of the updates of semaphore values are done.

## 2.4 UNIX Semaphore

UNIX Semaphores form a powerful and versatile synchronisation system but as space is limited only the system call which performs semaphore operations will be discussed. For a fuller description of UNIX semaphores see UNIX system manuals. The semaphore operations are performed using *semop(sid, ops, nops)*. The parameter *sid* specifies an array of semaphores, *ops* refers to an array of size *nops* which specifies the component operations to be executed. The component operations are specified by the data structure shown in Figure 3.

```
struct sembuf{
    ushort sem_num;
    short sem_op;
    short sem_flg;
    }
```

Figure 3:

Here *sem_num* indicates the index of the semaphore within the array, *sem_op* is the operation to be performed and *sem_flg* is used to modify the operation in ways we are not dealing with here. The operations that may be performed are described in Table 1.

| Semaphore Operations | |
|---|---|
| $sem\_op < 0$ | Calling process waits until $semval \geq \|sem\_op\|$ <br> $semval \leftarrow semval + sem\_op$ |
| $sem\_op = 0$ | Calling process waits until $semval = 0$ |
| $sem\_op > 0$ | $sem\_val \leftarrow semval + sem\_op$ |

Table 1:

When using UNIX semaphores in this paper we will use function $semop(s_1, o_1|| \cdots ||s_n, o_n)$ in place of the UNIX *semop* system call [8, 12]. In this notation $s_i$ are semaphore numbers and $o_i$ are integers which specify the operation to be performed on the semaphore $s_i$. For example, Dijkstra's P(s) and V(s) could be written $semop(s, -1)$ and $semop(s, 1)$ respectively.

## 2.5 Issues Regarding Extended Semaphore Operations

In order to maximise the power and usefulness of extended semaphore operations two aspects must be clarified. First, when several semaphores are operated on simultaneously and all the component semaphores are different then the order in which each component operation is done is irrelevant. E.g. $\Phi(s, a||t, b)$ is the same as $\Phi(t, b||s, a)$. At the same time natural and economical solutions to many synchronisation problems may be produced by repeating a semaphore within a semaphore operator [Figures 4,12,19]. However, when one or more semaphores are repeated within a semaphore operator the order the component operations are executed **must** be defined. As an example consider the UNIX semaphore implementation of an *eventcounters* given in Figure 4. The eventcounter [17] has two operations

- *await(s,a)* suspend until the semaphore $s$ is $\geq a$
- *advance(s)* increments the semaphore $s$ by 1

```
await(s, a):
    semop(s, -a || s, +a)

advance(s):
    semop(s, +1)
```

Figure 4: The Event Counter Implemented with UNIX Semaphores

Quite clearly *await* defined as in Figure 4 works only if the component operations are attempted in the order $s, -a$ and $s, +a$.

The second aspect is "fairness". Ideally all operations whatsoever, including those on semaphores should be *strongly fair* . By definition $\Phi$ is *strongly fair* if $\Phi$ terminates when ever the conditions for termination are true infinitely often[14]. To be useful operations on semaphores must be at least *weakly fair*. Again, by definition, $\Phi$ is *weakly fair* if $\Phi$ terminates as long as the conditions for termination are true for sufficiently long time[14].

As an example consider process $A$ executing $\Phi(s, -1||s, 0)$. Clearly, this succeeds if the value of $s$ is 1. If the value is 2 (or more) the second component operation would suspend, so the whole operation suspends, and $s$ stays 2. If now another process executes $\Phi(s, -1)$, it succeeds changing $s$ to 1, and the weak fairness requires that now the process $A$ resumes and its $\Phi$ terminates too. On the other hand the value of $s$ was never 0, so if the description saying that $A$ should re-try $\Phi$ when $s = 0$ is taken literally[2], $A$ stays suspended. Such an implementation of $\Phi$ is thus not even weakly fair.

[2]A common description [ [1],[2]] says that when the first component $s_i, m_i$ that cannot be done is found then the executing process suspends until the operation can be done.

## 2.6 Problems with UNIX System V Semaphores

In [12] N. Dunstan and I. Fris identify some poorly defined areas of UNIX Semaphores. The following is a brief summary of their findings.

- The order component semaphore operations are executed within a *semop* call. It can be assumed, but is not clearly documented, that the component operations are executed in the order they occur in the *semop* call. Figures 4,12,19 rely on this assumption.
- The degree of Atomicity of the UNIX semaphore system is unclear.
- It is assumed that UNIX semaphore are at least *weakly fair*, but examples have been found showing that in some implementations semaphore operations are not always weakly fair (see Figure 12).

The combination of the above means that it may not be possible to produce efficient and portable solution when using repeated occurrences of semaphores with UNIX semaphores.

# 3 Interval Semaphores

In this section an alternative system, Interval Semaphores, is introduced. First the Interval Semaphore operator *isem* is defined. Then we show that *isem* overcomes the issues discussed in Section 2.6.

## 3.1 Interval Semaphore Operations

The Interval Semaphore atomic operator *isem* for a single semaphore $s$ is defined in Figure 4. The semaphore's value $s.val$ may take any integer value including negative. The arguments $a$, $b$, $i$ are also arbitrary integers with $a \leq b$, $a$ and $b$ can also be + or - $\infty$, to indicate "no condition".

```
isem(a : s: b, i) :
   when $a \leq s \leq b$
      $s \leftarrow s + i$
```

Figure 5: The Interval Semaphore Operator for single semaphore

The operator defined in Figure 5 may be extended to operate atomically on several semaphores simultaneously:

$$isem(a_1 : s_1 : b_1, i_1 || \cdots || a_n : s_n : b_n, i_n)$$

The whole operation succeeds if and only if all its components operations succeed. Dijkstra's $V(s)$ can be expressed as $isem(0 : s : \infty, 1)$, while $P(s)$ is $isem(0 : s : \infty, -1)$ The operator *isem* overcomes the first difficulty of extended semaphore operations discussed in 2.5. Consider the following *isem* operation.

$$isem(a : s : b, d \mid\mid a' : s : b', d') \tag{2}$$

This operation terminates successfully when $a \leq s \leq b$ and $a' \leq s + d \leq b'$. Thus (2) succeeds if and only if

$$\max(a, a' - d) \leq s \leq \min(b, b' - d)$$

In other words (2) is equivalent to $isem(mx : s : mi, d + d')$ where $mx = \max(a, a' - d)$ and $mi = \min(b, b' - d)$. Consequently, there is no need to allow repeated use of the same semaphore in *isem*[3]. The example in Figure 6 shows an implementation of *eventcounters* using Interval Semaphores. Unlike the UNIX semaphore implementation of the eventcounter which decreases the value of the semaphore only to increase it immediately the Interval Semaphore solution simply suspends until s.val $\geq t$.

```
await(s,t):
   isem(t : s : ∞, 0)

advance(s):
   isem(0 : s : ∞, 1)
```

Figure 6: The Event Counter Implemented with Interval Semaphores

To overcome the second problem discussed in 2.5 we explicitly decree that *isem* is strongly fair.

## 4 Algorithm Comparisons

In this section a collection of synchronisation problems are solved using Dijkstra's semaphores. Then it will be shown that the extended semaphore operations of UNIX semaphores enable simpler and more economical solution to be produced. Finally, the Interval Semaphores operator, *isem*, will be used to solve the same problem.

### 4.1 Bounded Buffer

#### 4.1.1 Dijkstra's Solution

The solution in Figure 7 is a standard solution found in many books [9, 19].

[3]Moreover as $isem(-\infty : s : \infty, 0)$ is a no-operation which always completes we may, if we wish, require that every semaphore is listed exactly once.

There are three semaphores: *empty* initialised to the size of the buffer, *full* initialised to 0, *mutex* initialised 1

```
producer:                          consumer:
   P(empty)                           P(full)
   P(mutex)                           P(mutex)
     enter item                         take item
   V(mutex)                           V(mutex)
   V(full)                            V(empty)
```

Figure 7: A Solution to the Bounded Buffer using Dijkstra's semaphores

#### 4.1.2 A UNIX Solution

In the solution in Figure 7 it can be seen that all semaphore operations are in pairs and each pair performs the same operation but on different semaphores. UNIX semaphores enable the number of system calls to be reduced by operating on the pairs of semaphores simultaneously. The solution is outlined in Figure 8.

There are three semaphores: *empty* initialised to the size of the buffer, *full* initialised to 0, *mutex* initialised 1

```
producer:                             consumer:
   semop(empty, -1 || mutex, -1)         semop(full, -1 || mutex, -1)
         enter item                             take item
   semop(mutex, 1 || full, 1)            semop(mutex, 1 || empty, 1)
```

Figure 8: A UNIX Solution to the Bounded Buffer

The number of system calls can not be further reduced but UNIX semaphores allow a reduction in the number of semaphores. A solution to the bounded buffer that uses just two semaphores is given in Figure 9.

Two semaphores: *empty* and *full* both initialised to *n* the size of the buffer

```
producer:                             consumer:
   semop(empty, -1 || full, -n)          semop(full, -(n+1))
         enter item                             take item
   semop(full, n+1)                      semop(empty, 1 || full, n)
```

Figure 9: A UNIX Solution to the Bounded Buffer using Two Semaphores

This solution is not as clear or as natural as the first. The added complexity is due to the fact that one of the semaphores, *full* in this case, is used to synchronise two conditions. Synchronisation is achieved in the following way. The producer is able to pass the entry condition iff $empty > 0$ **and** $full \geq n$. The semaphore *full* will be less than $n$ when and only when a process is in its critical section. When a producer completes the exit protocol *full* is incremented by $n + 1$ so that the semaphore value is equal to $n$ plus the number of items in the buffer. The consumer may only pass the entry protocol when there is at least one item to take and the critical section is free. In other words $full \geq n + 1$. The exit protocol of the consumer releases mutual exclusion by increasing *full* by $n$ and increases *empty* by 1.

### 4.1.3 Interval Semaphore Solution

One of the differences between *semop* and *isem* is that *isem* may suspend a process if a semaphore value is not between two values. This diversity of Interval semaphores enables the bounded buffer to be solved with just one semaphore as given in Figure 10.

One semaphore : $s$ initialised to the twice the size of the buffer

```
producer:                              consumer:
   isem(n+1 : s : 2n, -(n+1))              isem(n : s : 2n-1, -n)
         enter item                              take item
   isem(0 : s :∞, n)                       isem(0 : s: ∞, n+1)
```

Figure 10: A Interval Semaphore Solution to the Bounded Buffer

This solution works as follows. The entry protocol to the producer subtracts $(n + 1)$ from $s$ if $n + 1 \leq s \leq 2n$. After a producer has accessed the buffer $s \leq n - 1$ which prevents other producers or consumers gaining access to the buffer. The exit protocol adds $n$ to $s$. When the buffer is full $s = n$, which prevents producers entering the buffer. The entry protocol of the consumer prevents entry if the buffer is empty. i.e $n \leq s \leq 2n - 1$. When this condition is meet mutual exclusion is guaranteed by subtracting $n$ from $s$. The exit protocol releases mutual exclusion and increase the number of spaces in the buffer.

## 4.2 Concurrent Readers and Writers

The Concurrent Readers and Writers (CRW) Problem is a classic synchronisation problem [15] which models access to a database. It is acceptable to have multiple processes reading the database, but if one process is writing to the database, no other process may have access to it. The problem is how to synchronise the reader and writer processes.

### 4.2.1 Dijkstra Semaphore Solution

A solution using Dijkstra's semaphores [15] is illustrated in Figure 11. A detailed evaluation of Courtois et.al's solution and the solution produced using UNIX semaphores is given in [8].

```
There are three semaphores: mutex, r and w all initialised to 1
and one integer shared variable 'readers' initialised to 0

reader:                                  writer:
  P(mutex)                                 P(w)
    readers ← readers+1                      write
    if reader = 1 then P(w)                V(w)
  V(mutex)
    read
  P(mutex)
    readers ← readers-1
    if readers = 0 then V(w)
  V(mutex)
```

Figure 11: A solution to CRW using Dijkstra's Semaphores

### 4.2.2 A UNIX Semaphore Solution

The solution by Fris and Dunstan [8] in Figure 12 uses one semaphore and one system call at entry and exit for both readers and writers. A natural implementation to the readers

```
One semaphore rw initialised to 1

readers:                                 writers:
  semop(rw, -1 || rw, 2)                   semop(rw, -1 || rw, 0)
        read                                     write
  semop(rw, -2 || rw, 1)                   semop(rw, 1)
```

Figure 12: One UNIX semaphore solution to CRW

exit protocol would be $semop(wr, -1)$. In some, but not all, implementations of UNIX this simpler *semop* call may not release processes suspended on the entry protocol of the Writer. Such an implementation of semaphores is not even weakly fair!

### 4.2.3 Interval Semaphore Solution

The algorithm using Interval Semaphores in Figure 13 uses the same logic as the solution using UNIX semaphores. The difference is Interval Semaphores uses one semaphore operation in the entry and exit protocols of both reader and writer. In addition as Interval

Semaphore are *strongly fair* a natural exit protocol to the reader is possible.

There is one semaphore *wr* initialised to 1

```
readers:                          writers:
    isem(1 : wr : ∞, 1)               isem(1 : wr : 1, -1)
        read                              write
    isem(2 : wr : ∞, -1)              isem(0 : wr : ∞, 1)
```

Figure 13: A Interval Semaphore solution to CRW

The solution in Figure 12 works in the following way: To write to the database the value of *wr* must equal 1. If this condition is met the value of *wr* is decremented preventing any readers or writes accessing the database. The exit protocol of the writer returns the value of *wr* to 1. A process may read if a process is not writing to the database, i.e $wr \geq 1$. When this condition is met *wr* is incremented preventing writers accessing the database but allowing more readers access. The exit protocol of the reader decrements *wr* by 1.

## 4.3 The One Lane Bridge Problem

There is traffic travelling from opposite directions at irregular intervals towards a single lane bridge. When a car reaches the bridge it may cross the bridge if one of the two conditions hold:-

- The bridge is empty.
- The traffic on the bridge is travelling in the same direction.

If neither of these conditions is met the car must wait until the first condition is met.

### 4.3.1 A Dijkstra Semaphore Solution

A solution using Dijkstra semaphores is not obvious and complicated in the sense it uses three semaphores and a shared variable with at least two fields. The solution is given in Figure 14.

### 4.3.2 A UNIX Solution to the One Lane Bridge

A natural solution to this problem requires only two UNIX semaphores and is given in Figure 15. Apart from being more efficient than the Dijkstra semaphore solution it is so simple that an explanation is almost unnecessary. The two semaphores keep count of the traffic travelling on the bridge in each direction. To enter the bridge a vehicle must complete its entry protocol by testing to see if the bridge is free from traffic travelling in the opposite direction by executing the first semaphore operation as written in the entry

Three semaphore *block*, *mutex_left*, *mutex_right* initialised to 1
and two integer shared variable *left_count* , *right_count*

```
left traffic:                              right traffic:
   P(mutex_left);                             P(mutex_right)
   if ( left_count = 0)                        if( right_count = 0)
      left_count ← left_count + 1                  right_count ←right_count + 1
      P(block);                                    P(block)
   V(mutex_left);                             V(mutex_right)
        on the bridge                              on the bridge
   P(mutex_left);                             P(mutex_right)
   if( left_count = 1)                        if( right_count = 1)
      left_count ← left_count - 1                right_count ← right_count - 1
      V(block);                                  V(block)
   V(mutex_left);                            V(mutex_right)
```

Figure 14: A Dijkstra's Semaphore Solution to the One Lane Bridge

Two semaphore left and right initialised to 0

```
left traffic:                              right traffic:
  semop(right, 0 || left, 1)                  semop(left, 0 || right,1)
        travelling on bridge                          travelling on bridge
  semop(left, -1)                            semop(right, -1)
```

Figure 15: A UNIX Solution to the One Lane Bridge

protocol. If this condition is met the process increments the number of vehicles travelling in its direction by executing the second semaphore operation in the entry protocol. Note as the semaphores are different the order the component operations are executed is relevant. The exit protocol decreases the number of vehicles travelling on the bridge.

### 4.3.3 An Interval Solution to the One Lane Bridge

The Interval Semaphores solution to the One lane bridge problem is even more economical that the UNIX solution as it may solved with one semaphore. For left traffic to gain access

One semaphore: s initialised to 0

```
left traffic:                              right traffic:
  isem($-\infty$ : s : 0, -1)                      isem(0 : s : $\infty$, 1)
       travelling on bridge                         travelling on bridge
  isem($-\infty$ : s : 0, 1)                       isem(0 : s : $\infty$, -1)
```

Figure 16: Interval Semaphore Solution to the One Lane Bridge

to the bridge when $s \leq 0$ . When this is true the value of $s$ is decremented. Traffic travelling from the right may only gain access to the bridge when $s \geq 0$. When this condition is met the value of $s$ is incremented. Both exit protocols perform the opposite operation to their respective entry protocol.

### 4.3.4 A Starvation Free Dijkstra Solution

The solutions to the one lane bridge given above may lead to starvation. This occurs when traffic from one direction is waiting to enter the bridge and traffic from the other direction is entering the bridge at a faster rate than is leaving it. A simple way to prevent starvation is to stop all cars from reaching the bridge if a car is waiting to access the bridge. A Dijkstra semaphore solution achieves this by using a new semaphore, *start* initialised to 1. The semaphore *start* is decremented by a process before access the bridge is tested. It is incremented back after the process has access to the bridge. Hence if a process is waiting to access the bridge no other process may access the entry protocol of the bridge preventing starvation. The solution is given in Figure 17.

Three semaphore *block=mutex_left=mutex_right=start=1*
Two integer shared variable *left_count, right_count*

```
left traffic:                               right traffic:
  P(start)                                    P(start)
  P(mutex_left)                               P(mutex_right)
  if ( left_count = 0)                        if ( right_count = 0)
      left_count ← left_count + 1                 right_count ←right_count + 1
      P(block)                                    P(block)
  V(start)                                    V(start)
  V(mutex_left)                               V(mutex_right)
        on the bridge                               on the bridge
  P(mutex_left)                               P(mutex_right)
  if( left_count = 1)                         if( right_count = 1)
      left_count ← left_count - 1                 right_count ←right_count - 1
      V(block);                                   V(block)
  V(mutex_left)                               V(mutex_right)
```

Figure 17: Starvation Free Solution using Dijkstra Semaphores

### 4.3.5 A Starvation Free UNIX Solution

A starvation free UNIX solution uses logic similar to the Dijkstra semaphore solution. Starvation is prevented in the solution shown in Figure 18 in the following way. A process entering the bridge changes *start* from 1 to 0 suspending all other processes from entering the bridge. This operation is followed by a single operation which checks that there is no

Three semaphores left, right initialised to 0, start initialised to 1

```
left traffic:                                right traffic :
   semop(start, -1)                             semop(start, -1)
   semop(right, 0 || left, 1 || start, 1)       semop(left, 0 || right, 1 ||start, 1)
         travelling on bridge                         travelling on bridge
   semop(left, -1)                              semop(right, -1)
```

Figure 18: A UNIX Starvation Free Solution

opposite traffic, increases the count of vehicles in its direction, and incrementing *start* back to 1. Starvation is prevented (assuming fairness) as the execution of $semop(start, -1)$ by, say, a right traffic prevents further left (and right) traffic from entry. Thus the bridge will eventually clear, allowing the right traffic to proceed.

Three semaphores left and right both initialises to 1

```
left traffic:                                right traffic:
   semop(right, -1 || left, -1)                 semop(left, -1 || right, -1)
   semop(right, 0 || right, 1 || left, 2)       semop(left, 0 || left, 1 || right, 2)
         travelling on bridge                         travelling on bridge
   semop(left, -1)                              semop(right, -1);
```

Figure 19: A UNIX Starvation Free Solution to the One Lane Bridge Problem using Two Semaphores

Actually, the extra semaphore *start* is not necessary in the case of UNIX semaphores. Instead, we let *left* and *right* semaphores to have values 1+*number of vehicles* in their particular directions. So, 1 means "no vehicles", and we use the value 0 to indicate that no further traffic in this direction should enter the bridge. See Figure 19. If both semaphores have positive values further traffic can successfully execute *semop(left, -1 || right, -1)*, leaving one of the semaphores with a value of 0. If the bridge is free from traffic travelling in opposite direction the value of both semaphores are incremented by 1. Otherwise neither of the semaphores are incremented. This causes all new processes to suspend on $semop(right, -1 || left, -1)$. Hence starvation is prevented.

### 4.3.6 A Interval Semaphore Solution

The Interval Semaphore solution in Figure 20 is a direct transposition of the UNIX solution given in Figure 19. It is a more economical and simpler solution as semaphores are not repeated in any of the *isem* operators.

Two semaphores *left* and *right* both initialises to 1

```
left traffic:                                  right traffic:
  isem(1:left :1,-1 || 1:right:1, -1)            isem(1:left:1,-1|| 1:right :1,-1)
  isem(0:right:0, 1 || 0:left:∞, 2)              isem(0:left:0, 1 || 0:right:∞,2)
        travelling on bridge                           travelling on bridge
  isem(0:left:∞, -1)                             isem(0:right :∞, -1)
```

Figure 20: A Interval Semaphore Solution to the One Lane Bridge

## 5 Conclusions

In this paper several classic synchronisation problems where solved using various semaphore systems. It is clear that extended semaphore operations allow us to find more economical solutions than those produced using Dijkstra's semaphores. It is also apparent that by allowing a semaphore to be repeated in a UNIX Semaphore operation the economy and clarity is increased. In addition a more natural solution may be produced. When a semaphore is repeated in a semaphore operation the order component operations are executed must be defined. As this is a poorly defined area of UNIX semaphores a better defined semaphore operator was designed. The new semaphore operator *isem* has been defined as atomic and strongly fair. Using *isem* eliminates the need to repeat a semaphore in any semaphore operation. It should be noted that a user level prototype of the Interval Semaphore System has been implemented and used to test our algorithms. We propose that a kernel level implementation of Interval Semaphores replaces the current implementation of UNIX System V semaphores.

## References

[1] *AT and T Unix System and user's Manual.* Prentice-Hall, Englewood Cliffs, N.J, 1986.

[2] M.J. Bach. *Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, N.J,, 1986.

[3] B.Freisleben and J.L. Keedy. On synchronizing readers and writers with semaphores. *The Computer Journal*, 25(1), 1982.

[4] B.Freisleben and J.L. Keedy. Priority semaphores. *The Australian Computer Journal*, pages 24–28, 1989.

[5] R.H. Campbell and A.N Habermann. *The specification of synchronisation by path expressions, Springer Lecture Notes in Computing Science, 16.* Springer-Verlag, 1974.

[6] R. Conradi. Some comments on concurrent readers and writers. *Acta Information*, 10(8):335–340, 1977.

[7] E.W. Dijkstra. Cooperating sequential processes. *Programming Languages*, pages 43–112, 1968.

[8] N. Dunstan and I.Fris. Concurrent readers and writers revisited. Technical report, University of New England, 1992.

[9] G.J.Nutt. *Centralised and Distributed Operating Systems.* Prentice-Hall, 1993.

[10] K.Ramamohanrao J.L.Keedy, J.Rosenberg. On implementing semaphopres with sets. *The Computer Journal*, 22(2), 1979.

[11] M.J.Rochkind. *Advanced UNIX Programming.* Prentice-Hall, Englewood Cliffs, N.J,, 1985.

[12] N.Dunstan and I.Fris. A tighter defintion of unix semaphores. In *Proceedings of the Australian UNIX Users Group Conference*, Melbourne, 1992.

[13] N.Dunstan and I.Fris. Process scheduling and unix semaphores. *Software Practice and Experience*, 25(10):1141–1153, 1995.

[14] N.Francez. *Fairness.* Springer-Verlag, 1986.

[15] F.Heymans P.J. Courtious and D.L. Parnas. Concurrent control with 'readers' and 'writers'. *Communications of the ACM*, 14(10), 1977.

[16] L. Presser. Multiprogramming coordination. *Computing Surveys*, pages 21–44, 1975.

[17] D.P Reed and R.K Kanodia. Synchronisation with eventcounters and sequencers. *Communications of the ACM*, 22(2):115–123, 1979.

[18] T.Agerwala. Some extended semaphore primitives. *Acta Information*, 10(8):171–176, 1977.

[19] A.S. Tannenbaum and A.S. Woodhull. *Operating Systems Design and Implementation.* Prentice-Hall, Englewood Cliffs, N.J, 1997.