SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(10), 1141–1153 (OCTOBER 1995)

Process Scheduling and UNIX Semaphores

NEIL DUNSTAN AND IVAN FRIS

Department of Mathematics, Statistics and Computing Science University of New England, Armidale, NSW, 2351, Australia
(email: neil@neumann.une.edu.au ivan@neumann.une.edu.au)

SUMMARY

Semaphores have been used extensively in programming concurrent tasks. Various extensions have been proposed for problems in which traditional semaphores turned out to be difficult to use. The extended semaphore primitives investigated here are based on the version of semaphores implemented in UNIX System V. Implementation issues are discussed and practical illustrations of their use are provided. In particular, algorithms for a variety of common process scheduling schemes are shown. These solutions are evaluated and the strengths and weaknesses of UNIX semaphores are discussed.

KEY WORDS: UNIX; semaphores; synchronization; concurrent programming

1. INTRODUCTION

The semaphore is a synchronization primitive which can be used directly to solve many tasks in process synchronization and coordination, and in concurrent programming in general. It is also used to implement various higher level constructs used in parallel programming. Following the original description,[1] the semaphore can be described as a programming structure whose data component is a non-negative integer variable $semval$. Two operations only are defined on semaphores, namely the operations P and V. The operation V increases the value of $semval$ by 1. The operation P decreases the value of $semval$ by 1 at some time when the resulting value would be non-negative. The *completion* of the P-operation – i.e. the decision that this is an appropriate moment to put the decrease into effect and the subsequent decrease itself is an indivisible operation. In fact, both these operations, V and the completion of P, must be *atomic* in the sense that when several processes operate on the same semaphore, each operation is completed before another starts – instructions comprising these operations are never interleaved. More than one process may have initiated a P operation on the same semaphore whose value is 0. The above definition of semaphores deliberately left unspecified the decision which process will be allowed to complete its initiated P operation when another process does V on this semaphore. That is, the result is non-deterministic.

Many other synchronization primitives have been proposed,[2,3,4] nevertheless semaphores continue to be a popular choice and have been included in a number of operating systems, including[5,6] and UNIX System V.[7] Extensions to the basic semaphore model have been suggested in order to provide more convenient primitives for solving some specific problems in the coordination of concurrent processes.[8,9,10,11] UNIX semaphores have significant

CCC 0038–0644/95/101141–13

Received 1 December 1994
Revised 21 March 1995

additional properties which make them a quite different and more powerful synchronization mechanism than the original semaphores. While some of these additional properties are based on earlier theoretical work on semaphore extensions, and have already received attention in the literature, a full appraisal of the practical capabilities of UNIX semaphores has not appeared. The ideas behind the UNIX semaphores are interesting, fruitful and worth consideration outside UNIX circles. They have received too little attention in the literature unjustly, being even occasionally described as just a baroque version of semaphores.

In this paper the power of UNIX-like semaphores is illustrated by using them to implement a variety of common process scheduling schemes. A general description of UNIX semaphores is given, concentrating mainly on properties used later in the paper. It gives, in fact, a more precise definition of some properties than can normally be found in the UNIX literature. As the UNIX system is not really the focus of the paper, a simplified notation for operations on these semaphores is introduced, rather than using a general standard notation which is restrained to conform to the UNIX system call syntax. Features which are in addition to those available in the original semaphores are identified and examples of their use are given. In particular, two features are identified which are used extensively in further examples. When it is necessary to distinguish between the two kinds of semaphores used, the original Dijkstra version is referred to as D-semaphores, while the version based on UNIX is referred to as U-semaphores.

2. UNIX SEMAPHORES

Here is a brief overview of semaphores as they are implemented in UNIX System V. The focus is on those features of UNIX semaphores that are relevant to the discussion which follows. For more complete descriptions see UNIX system manuals and tutorial descriptions.[7,12,13,14]

UNIX System V semaphores are provided by three system calls which are summarized in Table I.

Table I. Semaphore system calls

$semget(key, nsems, flags)$	creates an array of $nsems$ semaphores and returns its identification sid;
$semop(sid, ops, nops)$	performs atomically the list of operations on sid as specified by the array ops of length $nops$
$semctl(sid, snum, cmd, arg)$	changes or returns information about the current state of semaphores as contained in $struct\ sem$.

A data structure maintained for each semaphore is shown in Figure 1.

```
struct sem{
    ushort semval;
    short sempid;
    ushort semncnt;
    ushort semzcnt;
 }
```

Figure 1

Here $semval$ is the current value of the semaphore; $sempid$ is the process identification number of the process which performed the last operation; $semncnt$ is the number of processes waiting for $semval$ to reach a suitable positive value and $semzcnt$ is the number of processes waiting for $semval$ to be zero.

The parameters in $semop$ refer to array ops of size $nops$ and of the type *struct sembuf* as shown in Figure 2.

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

Figure 2

Here sem_num indicates the index of the semaphore within the array; sem_op is the operation to be performed and sem_flg can be IPC_NOWAIT – so that the calling process will not be suspended; or SEM_UNDO – to automatically adjust $semval$ if the process exits.

The operations that may be performed are described in Table II.

Table II. Semaphore operations

$sem\_op < 0$	Calling process waits until $semval \geq \lvert sem\_op \rvert$ $semval \leftarrow semval + sem\_op$
$sem\_op = 0$	Calling process waits until $semval = 0$
$sem\_op > 0$	$semval \leftarrow semval + sem\_op$

Note that when $sem\_op = -1$ the operation is equivalent to a conventional P. When $sem\_op = +1$ it is equivalent to a conventional V. Hence the capabilities of UNIX semaphores are a superset of those of Dijkstra's semaphores.

Block-set semaphores

There are many definitions of semaphores in the literature and the correctness of a program will depend on the exact definition used.[15]

Consider a process p attempting to execute $P(s)$ when s has the value zero. This operation cannot be completed. Consider two possible interpretations of what happens.

In the 'weaker' interpretation, P is attempted repeatedly, and is only completed when the value of the semaphore has risen.

In the other possible interpretation the semaphore data structure consists not only of the already mentioned $semval$, but has an additional component $semset$. When the P operation cannot complete, the process p is suspended, and this fact is remembered by storing the process in $semset$. Later, when $V(s)$ is executed, a process is removed from $semset$ and restarted. Only when $semset$ is empty is the $semval$ increased.

The first interpretation of semaphores corresponds to *busy-wait semaphore*,[15] the second to *blocked–set semaphore*.[15] The busy-wait semaphore has been called a *weak semaphore*.[16] It is easy to see that these two descriptions are functionally equivalent for D-semaphores because there are no operations on semaphores that will distinguish between them. This is because it cannot be determined whether a process actually has begun a P operation with

$semval = 0$, and become suspended, or has not yet reached that operation. The situation is fundamentally different for U-semaphores, yet the standard descriptions tend to ignore the problem. The UNIX system call $semctl$ can return $semncnt$ and $semzcnt$ – the number of processes actually suspended. There are many tasks programmed with semaphores, in particular those concerning priorities, where this distinction between different kinds of semaphores is relevant.

A further complication with U-semaphores concerns the fact that $semop$ may increase $semval$ by more than one in a single call, thus several processes may be released on one call. Here the requirement of atomicity of semaphore operations is interpreted by explicitly requiring that all processes which are released by a single call of $semop$ are released simultaneously rather than, for example, one-by-one. What this really means is that no intermediate values of the semaphore data structure may be detected. A final requirement is that only processes which are already suspended partake in this release. This is the meaning of the requirement that no further operations on the semaphore may start, until the current one is finished. This is considered more formally in the following.

Consider a semaphore s with current values $semval = v$, and $semncnt = 0$, i.e. in particular, no process is blocked. And consider n processes p_i, $(1 \leq i \leq n)$ which initiate an operation with $sem\_op = v_i$, $(v_i < 0)$ on s. Assume $v < \min|v_i|$, so that all these processes get blocked. After that, i.e. when $semncnt = n$, a process p_0 which may release some of those processes initiates an operation where $sem\_op = v_0$ where $v_0 > 0$.

Now, let $\mathcal{J} = \{j_1, \ldots, j_k\}$ be any set, possibly empty, of (distinct) indices $\in \{1, \ldots, n\}$ such that $v' = |v_{j_1} + \cdots + v_{j_k}| \leq v + v_0$ but $|v_{j_0} + v_{j_1} + \cdots + v_{j_k}| > v + v_0$ for any $j_0 \in \{1, \ldots n\} - \mathcal{J}$. Under these conditions processes $p_{j_1}, \ldots, p_{j_k}$ and process p_0 all complete their initiated operations on s as a part of a single atomic operation.

The resultant values are $semncnt = n - k$, and $semval = v + v_0 - v'$. Note, that the description was made for a single semaphore to avoid complex notation. Also, processes waiting for the $semval$ to become 0 were not considered. The idea extends straightforwardly to more complex cases, such as those involving several semaphores. The above described policy of process release may be called *strictly atomic* policy.

Also note that the convention is maintained that in general exactly which processes are released is non-deterministic.

The definitions of the relevant UNIX system calls are consistent with but do not necessarily imply these interpretations.

Notation

Rather than using UNIX system call syntax to describe semaphore operations a simple notation adjusted to our purpose is described as in the following.

Let sid be the value returned by the $semget$ function. It identifies an array of $nsem$ individual semaphores numbered 0 to $nsem - 1$. As it is of no significance what index number any particular semaphore has, it is preferable to use identifiers. Thus, for example in Figure 5, the semaphore array contains three semaphores, referred to as $a, delay, mutex$ rather than s_0, s_1, s_2. The name of the array is lost, but this, hopefully, causes no confusion.

Let $s_1, \ldots, s_k$ be a sequence of not necessarily distinct semaphores which are all part of the same semaphore array. Then $semop(s_1 : o_1, \ldots, s_k : o_k)$, where $o_1, \ldots, o_k$ are all integers, indicates the use of the system call $semop$ to perform the list of operations where $sem\_num = s_i$ and $sem\_op = o_i$. Note that the flags are not mentioned in the simplified notation – this is because they are not used in this paper (i.e. they are zero). Of course, the

simplified notation can be trivially extended for the use of non-zero flags.

Similarly, $val(s)$ and $ncount(s)$ will be used to obtain (using $semctl$) the values $semval$ and $semncnt$ of s respectively. Finally, $set(s, v)$ is used to set $semval$ of s to value v.

Differences between Dijkstra and UNIX semaphores

The features of UNIX semaphores which are not present in Dijkstra's original mechanism are:

— the inclusion of flags to qualify operations
— the execution of a list of operations indivisibly
— non-unitary operations.

The IPC_NOWAIT flag provides a means of decrementing the semaphore value only if that can be immediately achieved, otherwise the operation is aborted and an error indication is returned. This feature was also included in semaphore-based concurrency mechanisms for the MONADS-PC.[17] An obvious example of its use is when a process has alternative work to do. The process may attempt to secure access to some resource (via a semaphore operation including NO_WAIT) but rather than being blocked waiting for that resource it may do some other work if the resource is not immediately available.

The SEM_UNDO flag is useful where a process is liable to exit or be terminated in exceptional circumstances. If semaphores are used to guard access to resources and a process holds some of those resources its termination may leave them forever unavailable. Use of the SEM_UNDO flag in semaphore operations establishes an adjustment value which is used to restore a semaphore value automatically when a process is terminated. This effectively makes associated resources available again.

The $semop$ system call may be used to specify a list of operations on an array of semaphores. These operations may involve a number of different semaphores and/or a number of operations on the same semaphore. This feature is a generalization of *simultaneous* P operations,[9] it has also been called operations on *multiple event variables*,[11] and which we will refer to as *simultaneous operations*.

The original semaphore operations P and V are unitary operations in that they perform decrements or increments to the semaphore value by 1 only. UNIX semaphores permit increments or decrements by arbitrary integral amounts – a feature that is referred to as *non-unitary operations*. The non-unitary operation has been referred to as a *parameterized test–value extension*.[11]

Both simultaneous and non-unitary operations are powerful features and are used, sometimes in combination, in examples throughout this paper.

3. COMMON SCHEDULING SCHEMES

Weak and strong fairness for D-semaphores have been defined.[15] As U-semaphores offer more comprehensive functionality, these definitions must be reformulated. The problems with the usual definitions are that with U-semaphores processes may be blocked not only because the value of a semaphore is not positive, but also because it is not high enough or even because it is positive. A process may block another process without 'competing' for a semaphore. A final complication is that a single operation may refer to several semaphores. Fairness can be defined as follows.

Let us first call an operation executed by a process *blocked* if the process cannot complete the operation, and *unblocked* otherwise. Examples of operations causing processes to be

blocked are `semop(s:-5)` on a semaphore s the value of which is < 5, or `semop(s:0)` on a semaphore the value of which is > 0. A semaphore system is *weakly fair* if a process can complete any operation which stays unblocked sufficiently long, and is called *strongly fair* if a process can complete any operation which stays unblocked sufficiently often. In other words, a system in which a process can be permanently delayed on an unblocked operation is not weakly fair. If a process can be permanently delayed on an operation which is being blocked and unblocked repeatedly, then although such a system may be weakly fair, it is not strongly fair. This can happen, for example, in a busy-waiting implementation of semaphores, where a process may be permanently delayed because it only checks the semaphore when it happens to be blocked. Although not explicitly specified in UNIX, it is assumed that U-semaphores are at least *weakly fair*.* In this paper it is not required that U-semaphores be *strongly fair*. Strongly fair discipline on semaphores guarantees *starvation free* operations, however many applications need some specific *scheduling scheme*, in other applications strong fairness may be too constraining.

A scheduling scheme describes the order in which processes are released after being suspended. Several schemes may be considered.

(i) First-in-first-out (FIFO) or queue scheduling.
(ii) Linear delay – scheduling in which a process is not scheduled (released) a second time, while another process is still suspended. This is a more suitable discipline in many instances than FIFO as FIFO assumes that time is common to different processes even before they reach the scheduling point (e.g. a semaphore operation).
(iii) Limited delay – is a scheduling protocol giving a bound on how often a process can be overtaken by other processes, depending, for example, on the number of processes suspended or other possible parameters.
(iv) Finite delay – scheduling as in (iii) but not necessarily bounded.

Schemes (i) to (iv) are successively weaker.[18] Under protocols (i) to (iv) no starvation occurs, consequently these *protocols* are often called fair.

Then,

(v) Priority scheduling – in which each process is assigned a priority and released in the order of these priorities. Again, scheduling may involve FIFO with priority and linear delay with priority.
(vi) Ticketing systems – processes are released in the order of some associated number series. This scheme is analogous to a ticketing system for customer service. Each new customer receives the next numbered ticket from a monotonically increasing series. Customers receive service in that order. Note that if the ticket is issued at the time of entry to the 'service counter' then this is the same as FIFO scheduling. Eventcounts and sequencers[4] constitute a synchronization mechanism which embodies this scheduling scheme. A sequencer is a shared object which dispenses the next number from a monotonically increasing series. An eventcount is a non-decreasing integer variable on which some functional primitives are defined. The operation $Advance(eventcount)$ adds one to the eventcount value. The operation $Await(eventcount, ticket)$ causes the calling process to be delayed at least until the eventcount value equals the value of $ticket$.
(vii) Non-deterministic scheduling – implies that the order is not specified.

* But see the footnote in the section 'Readers and writers'.

(viii) Arbitrary scheduling – enabling releasing processes (or in a simpler case, one 'scheduler' process) to identify and release any single individual process.
(ix) Readers and writers scheduling[19] – where processes attempting read access to some database may do so together but a writing process must have exclusive access.

4. SCHEDULING WITH U-SEMAPHORES

The underlying scheduling scheme for semaphores, by which processes are released after waiting on the semaphore, is not a factor in every semaphore application. In the case of U-semaphores, this applies to the scheduling of the release of processes which become suspended due to invoking the *semop* operation with the same negative value on the same semaphore. Solutions presented here are based on non-deterministic scheduling of the built-in operations and are thus independent of the underlying scheduling scheme. Simultaneous and non-unitary operations are used to achieve specific scheduling policies. The solutions in this paper are for the general case in which processes are anonymous (except in the case of arbitrary scheduling) and (except in the case of linear delay scheduling) the number of processes is not fixed and processes may be created dynamically.

Non-deterministic scheduling

It is assumed that U-semaphores are implemented in this way.

First in first out scheduling

FIFO scheduling does not seem to have a straightforward implementation unless the underlying scheduling scheme is FIFO. However, an approximate solution is outlined later.

Linear delay scheduling

It has been shown that linear delay scheduling for a fixed number of processes can be programmed using only weakly fair semaphores.[20] The solution uses three D-semaphores and three integer variables. This solution can be simplified by using U-semaphores. Only two of them and one integer variable are needed. The algorithm is illustrated in Figure 3.

```
/* initially */
set(a, 1);   set(m, 0);
nm = 0;

/* Linear Delay Scheduling */
semop(a: -1);
nm = nm+1;
if( ncount( a ) == 0 ) semop(m: +1);
   else semop(a: +1);
semop(m: -1);

Critical Region

nm = nm-1;
if( nm  == 0 ) semop(a: +1 );
   else semop(m: +1 );
```

Figure 3

The idea behind this is quite simple. A 'batch' of processes which come more or less at the same time pass through $semop(a : -1) \ldots semop(a : +1)$ and get blocked on $semop(m : -1)$. As $semval$ for a was initially 1, processes from such a batch pass that region one-by-one. The last one increases $semval$ for m rather than a which remains 0. So any process arriving later, including those which have already passed through the critical region, get blocked on a, until all processes from this batch get through. Then a is 'opened' and m is left 'closed' and the cycle is repeated. This works regardless of the order in which processes are released from semaphores a and m.

Ticketing systems

Scheduling using a ticketing system can be provided by the algorithms described in Figure 4 to implement eventcounts.

```
Await( s, t )
{
    semop(s: -t);
    semop(s: +t);
}

Advance( s )
{
    semop(s: +1 );
}
```

Figure 4

$Await$ causes the calling process to wait until $semval$ can be decremented by t and then increments it by the same amount, so that there is no net change.

The $Await$ operation can be improved using simultaneous operations. Although there is no need for $Await$ to be indivisible, simultaneous operations will save one system call. So the sole system call is $semop(s : -t, s : +t)$. However, note that this relies on an undocumented feature of UNIX semaphores although it has been used.[14] Namely, that simultaneous operations are performed in the order they appear in the list – i.e. in their order in the ops array. If the two operations were to be performed in the reverse order, $Await$ would never delay a process.

Priority scheduling

Two approaches are discussed. Both utilize sem_op to represent the priority level. D-semaphore operations P and V are commutative in the sense that the sequence of operations $P(s),\ V(s)$, executed by different processes, has the same net effect on the semaphore value as the sequence $V(s),\ P(s)$. The $PriorityP$ and $PriorityV$ operations described here are not commutative. $PriorityV$ has no effect if there is no waiting process. Such a priority scheduling scheme is sufficient for implementing monitor priority conditions.[21]

In the first approach, the procedure repeatedly increases the value of the semaphore a until the process with the smallest priority level is released. In the general case, where more than one process might release processes, this procedure must be indivisible, so an

associated semaphore, $mutex$ is used for this purpose. Another associated semaphore, $delay$ is also required, it guarantees that not more than one process is released on one call. The semaphore array therefore has $nsem = 3$ (In fact, one array of two semaphores and one array with a single semaphore are sufficient, and potentially simpler). Figure 5 describes the first algorithm.

```
/* initially */
set(a, 0); set(delay, 0); set(mutex, 1);

PriorityP(priority)
{
    semop(a:-priority, delay:-1);
}

PriorityV( )
{
    semop(mutex: -1);
    if( ncount(a) > 0 )
    {
        semop(delay: +1);
        while( val(delay) ==  1 )
            semop(a: +1);
        set(a:0);
    }
    semop(mutex: +1);
}
```

Figure 5

$PriorityV$ requires potentially many individual calls of $semop$ in order to progressively increase $semval$ of a until a process is released. This is not a very efficient solution where the possible range of priorities is large. This problem could be avoided by the addition of a new flag the qualification of $semop$ of which is to increase $semval$ to the minimum value required to release a process. It is not clear if the semantics of such a flag can be sensibly extended to the case when processes are simultaneously suspended on several semaphores. Note, that here there is a case where the strictly atomic policy is essential. When $semval$ becomes large enough for the process with highest priority to be released, the algorithm assumes this occurs atomically, along with necessary adjustments to the values of $semncnt$ and $semval$. There is a problem concerning the UNIX implementation of the $ncount$ function. The $PriorityV$ must check that there is a process waiting. On our reading of UNIX system manuals, this check could be done by either $ncount(a)$ or $ncount(delay)$, but in practice, the latter always returns 0, implying that $semncnt$ is only raised for the first semaphore in an array of operations which results in a process being suspended. This is consistent with the description of operations on *multiple event variables*.[11]

The second approach avoids the incremental method of releasing the process with the lowest priority level. Instead, a shared list is maintained which holds each sem_op value (i.e. the priority) specified for processes currently suspended in order of priority. If the list is non-empty the $PriorityV$ operation removes the lowest value in the list and applies this value in the operation to release a process with this priority level. This implies the programmer must duplicate data already stored by the operating system. Again, the operations must be indivisible, necessitating the use of an associated semaphore. The algorithms are described

in Figure 6. Note that the reliance on strictly atomic policy can be avoided by including $semop(a : 0)$ after $semop(a : p)$ in $PriorityV$.

```
/* initially */
set(a , 0 ); set(mutex, 1 );

PriorityP( priority )
{
    semop(mutex:-1);
    put( priority, list);
    semop(mutex: 1);
    semop(a: -priority);
}

PriorityV( )
{
    semop(mutex: -1);
    if( list is non-empty )
    {
         p = take( list );
         semop(a: p );
    }
    semop(mutex: +1 );
}
```

Figure 6

The algorithms could be easily modified to allow pending releases and hence commutativity. To include pending releases, it would be necessary to maintain an associated $sempcnt$ variable and amend $PriorityP$ so that if $sempcnt > 0$ it is decremented and the call to $semop(a : -priority)$ is skipped. $PriorityV$ increments $sempcnt$ if the count of waiting processes is 0. Hence if there are two releases pending, then the next two processes to call $PriorityP$ will be immediately released regardless of their priority.

Arbitrary scheduling

Assume, for simplicity, that there is one process p_0, which, using some built in criteria, wants to schedule n processes $p_1, \ldots, p_n$. This can be easily achieved using D-semaphores. When a process p_i wants to be suspended, it does so on a 'private' semaphore s_i, possibly, when necessary, after setting a shared variable $v_i \leftarrow true$. By doing $V(s_i)$ the process p_0 can release p_i as required. Note that v_i need not be protected, as it is only set by p_i and unset by p_0 when p_i is blocked (although concurrent-read rather than the simpler exclusive-read model of shared memory is required). Also note that although there is a time window between $v_i \leftarrow true$ and $V(s_i)$, this does not effect the algorithm.

By using U-semaphores $ncount(s_i)$ may be used instead of variables v_i. More importantly, only $\lceil \lg(n+1)$ rather than n semaphores are needed. This may be important as the number of semaphores is very limited in many UNIX implementations. The idea is to use the binary representation of i as follows. Let $k = \lceil \lg(n+1)$ and $(i_{k-1}, i_{k-2}, \ldots, i_0)$ be the binary representation of i, that is $i = \sum_{j=0}^{k-1} 2^j \times i_j$. The process p_i is always suspended using $semop(s_0 : -i_0, \ldots, s_{k-1} : -i_{k-1})$. When this process is to be released p_0 does $semop(s_0 : +i_0, \ldots, s_{k-1} : +i_{k-1})$ (in this call the pairs $s_j : i_j$ where $i_j = 0$ can be omitted) possibly followed by $semop(s_0 : 0, \ldots, s_{k-1} : 0)$. The second call

$semop(s_0 : 0, \ldots, s_{k-1} : 0)$ is not needed if it is known that p_i is actually already blocked, and that raising a semaphore value atomically releases a process whenever possible, that is, strict atomicity. Note that it may be possible to use the process identification number pid, which on many systems is 15 or 16 bits, rather than some private numbering of processes. This algorithm also provides an interesting example of the use of $semop$ with $sem\_op = 0$.

Arbitrary scheduling may be used to implement most other scheduling, including priority, and, in a way, even FIFO. The latter is achieved by a function, not so far mentioned, which returns the time when a semaphore was operated on. In this case, a separate semaphore for each process is needed, rather than a $\lg(n + 1)$ array so that the time the semaphore is operated on is the time the process was blocked. Unfortunately, this solution is not entirely realistic as the time resolution is probably not fine enough. Using arbitrary scheduling also means having a special process to orchestrate the scheduling scheme. This is contrary to our preferred usage of semaphores, in which synchronizing code is executed in-process, that is by the processes themselves, without the need (and added overhead) of a special 'scheduling' process.

Readers and writers

Figure 7 shows a possible U-semaphore implementation for concurrent readers and writers.

```
initially: set(rc,0), set(wc,0), set(m,1)

Readers:                          Writers:
  semop( rc:1, wc:0)               semop( rc:0, wc:1; m:-1)

    read                                    write

  semop( rc:-1)                    semop( wc:-1, m:1)
```

Figure 7

Basically, rc counts the number of readers in, wc counts the writers. $wc : 0$ checks that there are no writers writing, while $rc : 0$ does the same for readers. Finally m is used conventionally to let at most one writer in.

This is a very natural solution. It is simpler than the D-semaphore solution[19] in that although it still needs three semaphores, it does not need any shared variables. Each entry and exit protocol consists of a single call. Another advantage of this solution is that it is easier to see that this is a non-priority solution. Readers do have priority over writers when other readers are in – and that is probably why the original D-semaphore solution was called 'readers priority', however when no process or only a writer is in, the entry of further readers or writers is non-deterministic.

Figure 8 shows a solution which needs a single semaphore only. Initially, rw has the value 1, which is the 'neutral' state meaning that neither readers nor writers are in. Value v of rw when > 1 means that there are $v - 1$ readers reading, while $v = 0$ indicates one writer writing. In both entry protocols the initial $rw : -1$ disallow the operation unless $v \geq 1$. The readers increment it by their $rw : 2$, (thus the total increment is one) the writer decrements it by its $rw : -1$, but this can be only done when the new value is equal to zero. This is checked by $rw : 0$. Each process restores the original value of the semaphore on exit. Note that the reader's exit protocol is what can be called 'dual' to its entry protocol.

```
initially: set(rw,1)

Readers:                              Writers:
  semop( rw:-1, rw:2)                   semop( rw:-1, rw:0)

       read                                  write

  semop( rw:-2, rw:1)                   semop( rw:1)
```

Figure 8.

Logically, it should be enough to use $semop(rw : -1)$ in the reader's exit part, and that is our preferred semantics, but for better or worse this does not work in at least some real UNIX implementations.*. It is not clear to us whether this feature is intentional. Also note that this solution depends on the order in which the operations in a call to $semop$ are attempted.

Further discussion of the concurrent readers and writers problem, including U-semaphore solutions to reader and writer priority variations, can be found in another study.[22]

5. CONCLUSIONS

The features of UNIX semaphores extend the range of applications that may be conveniently supported by semaphores. To provide these features, UNIX maintains substantial data structures for each semaphore as well as code to manipulate them. Non-trivial examples of using features of UNIX semaphores which extend traditional Dijkstra semaphores have been shown. These examples include

— algorithms for priority scheduling
— efficient implementation of arbitrary scheduling
— efficient implementation of eventcounts and their operations and
— efficient implementation of concurrent readers and writers.

This is in contrast to disparagement of these features.[7]

UNIX semaphores were used in a manner which provided most common forms of process scheduling. Priority scheduling is not well supported but could be with the addition of a new flag designed for that purpose. FIFO scheduling is also not well supported, but starvation free scheduling can be achieved by linear delay scheduling.

When these algorithms were tested on several UNIX machines, a number of ill-defined areas concerning the behaviour of the relevant system calls were discovered which are dealt with in another study.[23]

* When `semop(rw: -1)` is used as the exit protocol for the reader, the following happens in some implementations. When a writer arrives while a reader is in its critical section, i.e. when the value of *rw* is 2, the writer is correctly blocked. When, however, the reader leaves, adjusting *rw* to 1, the writer stays blocked. What apparently happens is that when a semaphore goes **down**, but stays positive, the assumption was made that this would not release any processes, and thus the list of blocked processes is not checked. This gives more 'efficient' implementation for the cost, that in some unusual cases the semaphore is not even weakly fair! The program works as expected, i.e. the writer is released, when `semop(rw:-2, rw:1)` is used as shown in Figure 8.

REFERENCES

1. E. W. Dijkstra, 'Cooperating sequential processes', in F. Genuys (ed.), *Programming Languages*, Academic Press, New York, 1968, pp. 43–112.
2. R. H. Campbell and A. N. Habermann, 'The specification of process synchronization by path expressions', *Springer Lecture Notes in Computer Science 16*, Springer-Verlag, 1974.
3. C. A. R. Hoare, 'Monitors: An operating system structuring concept', *Communications of the ACM*, **17**(10), 549–557 (1974).
4. D. P. Reed and R. K. Kanodia, 'Synchronization with eventcounts and sequencers', *Communications of the ACM*, **22**(2), 115–123 (1979).
5. E. W. Dijkstra, 'The structure of the THE multiprogramming system', *Communications of the ACM*, **11** (5), 341–346 (1968).
6. R. Duncan, 'A programmer's introduction to OS/2', *BYTE*, September, 101–108 (1987).
7. M. J. Rochkind, *Advanced UNIX Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1985.
8. T. Agerwala, 'Some extended semaphore primitives', *Acta Informatica* **8**, 171–176 (1977).
9. R. Conradi, 'Some comments on concurrent readers and writers', *Acta Informatica* **8**, 335–340 (1977).
10. B. Freisleben and J. L. Keedy, 'Priority semaphores', *The Australian Computer Journal*, 24–28 (1989).
11. L. Presser, 'Multiprogramming coordination', *Computing Surveys* **7** 21–44 (1975).
12. M. J. Bach, *Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
13. K. Haviland and B. Salama *UNIX System Programming*, Addison-Wesley, UK, 1987.
14. R. Stevens *UNIX Network Programming* , Prentice-Hall, Englewood Cliffs, NJ, 1990.
15. M. Ben-Ari *Principles of Concurrent and Distributed Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
16. E. W. Stark, 'Semaphore primitives and starvation-free mutual exclusion', *Journal of the ACM*, **29**(4), 1049–1072 (1982).
17. N. Dunstan, J. Rosenberg and J. L. Keedy, 'Support for concurrent programming on the MONADS-PC', *The Australian Computer Journal*, **25**(1), 1–6 (1993).
18. M. Raynal, *Algorithms for Mutual Exclusion*, North Oxford Academic, 1986.
19. P. J. Courtois, F. Heymans and D. L. Parnas, 'Concurrent control with "readers" and "writers" ' *Communications of the ACM*, **14** (10), 667–668 (1971).
20. J. M. Morris, 'A starvation free solution to the mutual exclusion problem', *Information Processing Letters*, **8**(2) 76–80 (1979).
21. N. Dunstan, 'Building monitors with UNIX and C', *ACM SIGCSE Bulletin*, **23**(3), 7–9 (1991).
22. I. Fris and N. Dunstan, 'Concurrent readers and writers revisited', University of New England, Dept. Maths, Stats and Comp. Sci., *Technical Report 92-57*, 1992.
23. N. Dunstan and I. Fris, 'A tighter definition of UNIX semaphores', *Proceedings of the Australian UNIX Users Group Conference*, Melbourne, 1992.