Priority Semaphores

B. FREISLEBEN[1] AND J. L. KEEDY[2]*

[1] *Fachbereich Informatik, Technische Hochschule Darmstadt, Alexanderstr. 24, D-6100 Darmstadt, F.R.G.*

[2] *Department of Computer Science, University of Newcastle, NSW 2308, Australia*

Neither low-level mechanisms such as semaphores nor higher-level mechanisms such as path expressions provide a simple means of solving synchronisation problems involving the scheduling of processes or classes of processes according to different priorities. This paper presents a new set of primitives which are easy to use and simple to implement. Their use is described in terms of the familiar reader–writer problem and the general scheduling problem involving arbitrary levels of priority with support for pre-emption and shared access by certain process classes. An efficient implementation, which reduces to a minimum the number of calls required to the process scheduler, is then described.

Received May 1987

1. INTRODUCTION

Dijkstra's semaphores[1] with P and V operations have been demonstrated to be adequate and sufficient to solve a wide variety of synchronisation problems. However, solutions for some classes of problem can be rather cumbersome and difficult to discover. In consequence there has been a shift of opinion in favour of higher-level language constructs such as monitors[2,3] and path expressions[4] to facilitate solutions for complex synchronisation problems. It has become common practice in proposing such solutions to demonstrate that they can be implemented via semaphores, the implicit assumption being that because semaphores *can* be used to provide a possible implementation they probably *will* be. In consequence, an increasing number of operating systems support P and V operations in the system kernel as the only synchronisation mechanism.

This situation can be compared with the view that because *add integer* and *negate integer* operations are the only necessary primitives for implementing all real and integer arithmetic operations, these are the only primitives which should be supported at the machine instruction level. Such a view is of course unacceptable, at least on the grounds of efficiency, and in practice a much wider range of integer and floating-point operations is provided as machine instructions in all modern computers.

In this paper we likewise propose an extended set of synchronisation operations which can be used to complement P and V operations and thus simplify the programming of a particular class of synchronisation problems, namely those problems involving the scheduling of individual processes or classes of processes according to different priorities.

2. PREAMBLE ON SEMAPHORE IMPLEMENTATION AND CLASSES OF PROCESSES

In an earlier paper[5] we discussed how P and V operations can be efficiently implemented by combining indivisible machine instructions with commutative queuing routines. The P operation is formed by combining the *dect* instruction (which decrements a semaphore variable by one and sets a condition code to show whether the result is negative, zero or positive) with the *suspend* routine (Fig. 1). Likewise the V operation consists of a *tinc* instruction (which increments the semaphore variable by one and sets a condition code indicating its value *before* the execution of the instruction) together with the *activate* routine (Fig. 1).

```
P macro: if dect(sem) < 0 then suspend(sem-
  queue)
V macro: if tinc(sem) < 0 then activate(sem-
  queue)
```

Figure 1. P and V macros based on machine instructions.

The *suspend* and *activate* routines can be considered as part of the system kernel, and must be commutative in the sense that *activate* wakes up exactly one process and that the sequence *suspend*; *activate* has the same net effect as *activate*; *suspend*.

We also demonstrated in the same paper how these primitive constituents of semaphores can be combined in different ways to produce efficient solutions for a variety of more complex synchronisation problems. One such problem of particular relevance to the present paper concerns the claiming/releasing of resources by the first/last process of a particular class on behalf of other processes in the class (equivalent to the PP and VV procedures proposed by Campbell and Habermann[4] to implement simultaneous execution in path expressions). The solution is described in Fig. 2.

```
PP(sem): P(preliminary);
         if tinc(count) = 0 then P(sem);
         V(preliminary)
VV(sem): if dect(count) = 0 then V(sem)
```

Figure 2. PP and VV operations.

Preliminary and *count* are auxiliary semaphores shared by the class of processes. The VV operation is simpler and more efficient than the normal semaphore-based solution because mutual exclusion for the count variable is achieved by the tinc/dect instructions rather than through use of an extra semaphore.

3. PREVIOUS SOLUTIONS FOR ORDERING PROCESS EXECUTION

In terms of problems involving the ordering of process execution, the P and V operations on semaphores are

* To whom correspondence should be addressed.

well suited to ensuring that only one process at a time may execute a critical section. Assuming an initial value of one for a semaphore, the P operation in effect inhibits all processes except the first from continuing and the V operation releases a single process (if one is waiting) from its inhibited condition. Likewise PP and VV operations, themselves composed of semaphore operations as discussed above, are well suited to ensuring that a class of processes may concurrently execute a critical section while excluding processes of other classes. Both kinds of operation can be clearly observed in the initial solution for the 'reader–writer' problem given by Courtois, Heymans and Parnas.[6] In that example the various semaphore operations ensure that readers exclude writers and that a writer excludes both readers and other writers. However, if both readers and writers are waiting when a writer executes the V operation, then it is arbitrary which proceeds first (if we accept, as is usual, that no assumptions about priority are built into the V operation).

Should we wish to control the order of further execution without changing the assumptions about semaphores, it becomes necessary to introduce further semaphores. If readers are to be given priority over writers, this can be arranged by nesting the writer code within a further P and V pair on an extra semaphore, thus ensuring that writers only wait at the P(w) operation if no other writers are present (Fig. 3).[8] In effect writers inhibit other writers before contending with readers for use of the database, by means of the P(extra) operation.

```
integer readcount; (initial value = 0)
semaphore mutex, w, extra; (initial value = 1)
  READERS                          WRITERS
  P(mutex);
  readcount:= readcount+1;         P(extra);
  if readcount = 1 then P(w);      P(w);
  V(mutex);
  {read database}                  {update data-
                                   base}
  P(mutex);                        V(w);
  readcount: = readcount-1;        V(extra);
  if readcount = 0 then V(w);
  V(mutex);
```

Figure 3. The reader priority solution using semaphores.

While this solution is straightforward, giving writers priority turns out to be much more complex, as is shown in the solution proposed by Courtois, Heymans and Parnas.[6] The reason for this complexity is that while semaphores are well suited to inhibiting other processes, they cannot directly be used by one class of processes to inhibit other classes of processes. (This is the reason why the PP and VV operations are in effect used in both the reader and writer code, although in the problem statement only the readers form a class which can share the database.) It appears to be solutions such as this which have led to the view that semaphores are too primitive for normal use and should be hidden by higher-level language constructs.

However, when we examine higher-level solutions for the same problem we find that they too are non-trivial. As an example consider the path expression solution proposed by Campbell and Habermann,[4] which appears in Fig. 4. This complexity can probably be attributed to the assumption that higher-level constructs will be implemented via a kernel which provides P and V operations as the only synchronisation tools.

```
        path readattempt end
        path requestread, {requestwrite} end
        path {openread; read}, write end
where
        readattempt = begin requestread end
        requestread = begin openread end
        requestwrite = begin write end
        READ = begin readattempt; read end
        WRITE = begin requestwrite end
```

Figure 4. The writer priority solution using path expressions.

In earlier papers[7,8] we have questioned this assumption and have proposed that additional low-level synchronisation tools can greatly simplify problem solutions at a higher level. One example of this was a 'reader–writer semaphore' which directly solves the problem under discussion.[8] It could be argued, however, that reader–writer semaphores provide a very specific solution for a particular problem. For this reason we now propose an alternative, which can be applied to a wider class of problems involving the scheduling of processes and classes of processes such that different classes of processes have different levels of priority and may need to use resources either in mutually exclusive or in shared mode.

4. THE HIGH-LEVEL SOLUTION

The aim of our proposal is to provide a simple and easy-to-use method of ensuring that individual processes or classes of processes can access a critical resource according to priorities assigned to the various classes of processes. To achieve this we propose two macros.

```
request(x,p): used by a process or
    class of processes to request
    resource x with priority p.
release(x): used by a process or class
    of processes to release resource x.
```

Using these macros we can very simply formulate a solution for all the priority scheduling problems described above (Fig. 5). The critical resource is represented by a new, special kind of semaphore, called a *priority semaphore*. It is assumed that there are k levels of priority, with 1 being the highest and k the lowest level.

```
resource: priority_semaphore;
PROCESS IN CLASS i, 1 <= i <= k
request (resource, i);
use resource
release (resource);
```

Figure 5. The general priority scheduling solution using priority semaphores.

The above code represents a solution for all problems involving the scheduling of processes or classes of processes according to different priorities. The priority

semaphore can be initialised appropriately to specify whether the processes in a particular class can share access or must have mutually exclusive access to the resource. Since the solution consists entirely of a request/release pair, it offers a maximal degree of simplicity. It now remains for us to consider whether the proposed macros can be implemented efficiently.

5. IMPLEMENTATION

The proposed implementation, like the semaphore implementation discussed in Section 2, is based on the principle that the kernel's scheduling routines should only be called if a queuing operation is actually necessary, in order to reduce the calling and process scheduling overheads to a minimum.

To achieve this we propose that a priority semaphore is a data structure embedded in the user program, and has several fields (Fig. 6).

```
type priority_semaphore =
  record
        preempt:    boolean
        preempted:  boolean
        active:     integer
        class:      array [1..k] of
                    record
                          owned:      boolean
                          shared:     boolean
                          proccount:  integer
                    end
  end
```

Figure 6. The data structure for a priority semaphore.

The first two fields (*preempt* and *preempted*) have been included in the data structure to ensure (if appropriate) that newly arrived processes in a low priority class with shared access are not permitted to enter the critical region while there are waiting processes of higher priority.

When the boolean field *preempt* is initialised to 'true', newly arriving processes in the currently active class, if it is a sharing class, may only proceed if there are no higher priority processes waiting. The boolean field *preempted* is set to 'true' to indicate that higher-priority processes have been delayed for this reason. If *preempt* is initialised to 'false', newly arrived processes in a currently active (sharing) class are permitted to continue regardless of waiting processes of higher priority.

The field *active* counts the number of processes currently using the resource. Each process class is represented by the three fields *owned*, *shared* and *proccount*. The field *owned* is set dynamically when the corresponding class has been granted access to the resource, and *shared* is initialised to specify whether the corresponding class has shared access or not. The *proccount* field holds a count of the number of waiting processes in the corresponding class. Two indivisible machine instructions (described in a PASCAL-like notation) operate on this structure (Fig. 7).

These instructions are used in combination with the process scheduler's normal *suspend* and *activate* routines to build the desired macros. However, the *activate* routine may be extended to allow the caller to specify how many processes should be activated. This reduces the calling overheads to the kernel if several processes of a shared class have to be activated. The macros can then be programmed as shown in Fig. 8.

The *request* macro starts with the execution of the *priority-P* instruction. A calling process can only gain access to the critical resource either if no other process is present or if it belongs to the currently active (shared) class and no higher-priority process is waiting (if *preempt* is set). Otherwise, a condition code is set to indicate that the process has to wait, which is effected by the subsequent call of the *suspend* routine in the *request* macro. It is assumed that the process scheduler maintains a separate queue for each priority class of the priority semaphore, and that a process is suspended by inserting it into the appropriate queue.

The *release* macro starts with the execution of the *priority-V* instruction. A calling process will only cause further actions if it is the last active process of a class. In this case the waiting process or process class with the highest priority is determined by sequentially scanning the *proccount* fields starting with *class*[1]. The first non-zero *proccount* entry determines the waiting process or class with the highest priority. A successful search is indicated by setting a condition code which is passed back to the *release* macro, and this results in calling the *activate* routine of the process scheduler. If the waiting class with the highest priority is allowed to have shared access to the resource all processes in the queue are activated together.

From the process scheduler's viewpoint there is no reason to distinguish between normal semaphore queues and priority semaphore queues (except in so far as *activate* may specify how many processes should be woken up). We assume that these routines are commutative, as discussed in Section 2. This is necessary because a process may be interrupted in the *request* macro between the *priority-P* instruction and the *suspend* call with an intervening execution of *release*.

An efficient implementation is clearly possible. If the *active* and *proccount* fields are represented by 6 bits each and all remaining fields by one bit each, then a priority semaphore to support 3 priority classes with a maximum of 63 waiting processes in each queue and 63 concurrently active processes in a shared class can be implemented in a single word of a 32-bit computer. An extension to support more than 3 priority classes requires that the fields *owned*, *shared* and *proccount* for each additional class are contained in additional machine words. However, a solution involving several words of memory would complicate the microcode to handle the case where a semaphore crosses a page boundary. An alternative solution can be achieved by redefining the *request* and *release* macros in such a way that several individual priority semaphores (with three classes) are manipulated. The solution is described in detail in Ref. 9.

The machine instructions can easily be implemented in microcode. The main advantage is that the kernel's scheduling routines are called only when a queuing operation is actually necessary. It would, however, be possible to implement the operations entirely as kernel primitives if microcoding facilities are not available. A microcoded implementation carried out for an ICL PERQ revealed execution times of about 8 microseconds for *priority-P* and 5 microseconds for the *priority-V* instruction.

```
const maxclass = maximum number of process classes
      maxproc = maximum number of processes in a class
instruction priority-P (prsem: priority_semaphore;
                        claimant: 1..maxclass;
                        var condition_code: boolean);
begin with prsem do
begin
if active = 0 or
  (class[claimant].owned and class
  [claimant].shared and not preempted)
  then begin
          class[claimant].owned: = true;
          active: = active+1;
          condition_code: = false;
          end
    else begin
          condition_code: = true;
          class[claimant].proccount: = class[claimant].proccount+1;
          if preeempt and not preempted
            then begin
                 i: = maxclass;
                 while i > claimant and not preempted do
                   begin
                   if class[i].owned then preempted: = true;
                   i: = i-1;
                   end;
                 end;
          end;
end;
end.

instruction priority-V (prsem: priority_semaphore;
                        var    restart: 1..maxclass;
                               pc: integer;
                               condition_code: boolean);
var i: integer;
begin with prsem do
begin
active: = active-1;
condition_code: = false;
if active = 0
  then begin
       for i: = 1 to maxclass do class[i].owned: = false;
       preempted: = false;
       i: = 0;
       repeat        i: = i+1;
                     if class[i].proccount > 0
                       then begin
                            class[i].owned: = true
                            restart: = i;
                            if class[i].shared
                              then pc: = class[i].proccount
                              else pc: = 1;
                            class[i].proccount: = class[i].proccount-pc;
                              active: = pc;
                              condition_code: = true;
                              end;
              until i = maxclass or condition_code;
                              end;
end;
end.
```

Figure 7. Machine instructions to support priority semaphores.

```
macro request(x: priority_semaphore; p: 1..max-
      class);
var   cc: boolean;
begin priority-P(x, p, cc);
      if cc then suspend(p-queue);
end.

macro release(x: priority_semaphore);
var   cc: boolean;
      p: integer;
      n: integer;
begin priority-V(x,p,n,cc);
      if cc then activate(p-queue,n);
end.
```

Figure 8. The priority semaphore macros.

6. CONCLUSION

Not only are some common synchronisation problems involving the ordering of processes with different priorities difficult to solve with semaphores, but they may also lead to rather complex solutions using higher-level constructs such as path expressions. We have presented an alternative approach based on priority semaphores which is easy to apply to a variety of problems and can be very efficiently implemented. The implementation can be generalised for an arbitrary number of process classes with shared or mutually exclusive access and is capable of dealing with both pre-emption and non-pre-emption cases.

REFERENCES

1. E. W. Dijkstra, Cooperating sequential processes. In *Programming Languages and Systems*, edited F. Genuys. Academic Press, London (1968).
2. P. Hansen Brinch, The programming language concurrent Pascal. *IEEE Transactions on Software Engineering* **SE-1** (2), 199–207 (1975).
3. C. A. R. Hoare, Monitors: an operating system structuring concept. *Comm. ACM* **17** (10), 549–557 (1974).
4. R. H. Campbell and A. N. Habermann, *The Specification of Process Synchronisation by Path Expressions*. Lecture Notes in Computer Science, vol. 16, Springer, Heidelberg (1974), pp. 89–102.
5. J. L. Keedy and B. Freisleben, On the efficient use of semaphore primitives. *Information Processing Letters* **21** (4), 199–205 (1985).
6. P. J. Courtois, F. Heymans and D. L. Parnas, Concurrent control with readers and writers. *Comm. ACM* **14** (10), 667–668 (1971).
7. J. L. Keedy, K. Ramamohanarao and J. Rosenberg, On implementing semaphores with sets. *The Computer Journal* **22** (2), 146–150 (1979).
8. J. L. Keedy, J. Rosenberg and J. Ramamohanarao, On synchronising readers and writers with semaphores. *The Computer Journal* **25** (1), 121–125 (1982).
9. B. Freisleben, *Mechanismen zur Synchronisation paralleler Prozesse*. Informatik Fachberichte 133. Springer, Heidelberg (1987).

Announcement

12–13 OCTOBER 1989

International Workshop on Raster Imaging and Digital Typography, Ecole Polytechnique Fédérale, Lausanne, Switzerland (sponsored by the Eurographics Association)

Raster image processors for non-impact printers and plotters require highly sophisticated algorithms and performant hardware. Outline character acquisition, design, manipulation and rasterisation, as well as graphic and image rendering are of major concern to scientists and engineers involved in the development of raster imaging devices.

Contributions include:
- Shape acquisition (curve fitting)
- Shape manipulation
- Character design
- Character representation and transformation
- Measuring type quality
- Character structures (generation/recognition)
- Page description languages
- Rasterisation algorithms
- Rasterisation accuracy
- Fast rasterisation hardware

For further information contact one of the following:

Debra Adams, XEROX PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA. Tel: (415) 4944417; telefax (415) 4944022; e-mail: adams.pa@Xerox.COM.

Dr Jacques André, INRIA/IRISA – Rennes Campus, Université de Beaulieu, F-35042 Rennes Cédex, France. Tel: (33) 99 36 20 00; telefax: (33) 99 38 38 32; telex: 950 473F UNIRISA; e-male: jandre@irisa.irisa.fr; jandre@irisa.uucp.

Professor Roger Hersch, LSP/EPFL, 37 Avenue de Cour, CH-1007 Lausanne, Switzerland. Tel: (4121) 47 43 57/693 43 57; telefax: (4121) 47 39 09/693 39 09; e-mail: hersch @elde.epfl.ch.

18–20 OCTOBER 1989

Communicating to the World, Garden City Hotel, International Professional Communication Conference

The theme for 1989 fits well with the New York location, and underscores the international character of communication. The health and growth of world culture in information-based societies truly depends on expert communicators creating comprehensible messages.

The three-day conference will focus on topics related to the conference theme, and on other issues of concern to technical professionals. We encourage your proposals for original papers, complete sessions devoted to an issue, demonstrations, panel discussions or workshops.

Topics
- International communications systems
- Managing communications
- Usability testing and communications research
- On-line database systems
- Computer bulletin boards
- Sharing information internationally by satellite
- Buying and managing communication technologies
- Partnerships: university and industry
- Networking: pros and cons
- Wide-area networks
- Impact of networks on communication patterns
- The editing process in a desktop environment
- Communicating in multi-cultural markets
- Computer-aided graphics
- Handling professional and public information in multi-national environments – banks, airlines
- Communicating technology to the public
- Translation of technical information
- Marketing and proposal development
- Graphics for multi-cultural applications
- Automating proposal preparation
- Video for proposals, reports or public information
- Managing for productivity in the contemporary communication environment

For general IPCC 89 information contact:

Mr Richard M. Robinson, Grumman Corporation, MS C39-05, Bethpage, NY 11714, USA. Tel: 516-575-5472.