SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 729–740 (1979)

Synchronization under a Commercial Operating System

JØRN JENSEN
BBC Nordisk Brown Boveri A/S, Vester Farimagsgade 7, DK-1606 Copenhagen V, Denmark

SØREN LAUESEN AND A. P. RAVN
Institute of Datalogy, Sigurdsgade 41, DK-2200 Copenhagen N, Denmark

SUMMARY

Mutual exclusion and general synchronization of concurrent processes ('tasks') are well-known principles for constructing reliable real-time systems. This paper shows how to implement these principles under a typical commercial operating system which provides incomplete synchronization operations. The problem of synchronizing erroneous tasks is discussed briefly.

KEY WORDS Synchronization Semaphores Commercial operating system Task Real-time systems

INTRODUCTION

The principles of mutual exclusion and semaphore synchronization of concurrent processes ('tasks') have been widely known for years, and the use of these principles for constructing reliable real-time systems has been demonstrated several times.[1, 2, 4, 8, 9] Yet, most commercially available real-time operating systems do not provide tools to match the principles.

When designing a real-time system for a particular computer, you then have three choices:

1. To develop an operating system of your own.
2. To use a commercially available operating system and try to implement the well-known principles.
3. To replace the well-known principles by *ad hoc* programming.

The authors have had long experience with the first choice, which gave high-quality dedicated systems. However, the systems required by customers tended to become less and less 'dedicated': customers wanted to develop modules of their own on the 'dedicated' system. In order to follow choice (1), we would have to develop or maintain several compilers, editors, etc. for our own operating systems. And that is extremely more costly than developing an operating system for a dedicated system.

Consequently we had to consider the second choice more seriously. The computer in question was the PDP–11 and the real-time operating system supplied by the manufacturer was RSX–11M. With this given, we tried to implement mutual exclusion and semaphore synchronization among tasks (jobs) running under RSX. If it could be done efficiently, a major problem would be solved.

RSX provides a large set of 'synchronization' operations. Two operations provide simple stop and start of tasks:

suspend Stops the calling task temporarily until it is started by another task.
resume(Q) Starts the task Q, provided that it is already suspended.

0038–0644/79/0909–0729$01.00

Received 20 February 1979
Revised 6 April 1979

Another group of operations provides indivisible setting and testing of flags or indivisible testing and suspension (stopping). But no operation provides setting, testing and suspension as one indivisible operation (semaphores can be considered as providing just that).

Still other operations provide a kind of message communication between tasks, but the pool of messages is common for all tasks. This implies that an erroneous user task can monopolize the pool and prevent important tasks from communicating.

All operations are rather time consuming, so an implementation of mutual exclusion based on them tended to involve too high an overhead, with the result that programmers had to circumvent mutual exclusion in cases where it would be the natural tool.

This paper shows how to implement mutual exclusion and general synchronization efficiently by means of unconditional stop and start operations (suspend and resume). In addition, the tasks involved need access to a common area. Protection against independent user tasks is arranged by not allowing these tasks to address the common area—a facility provided by RSX and the hardware.

The solution is efficient because it only calls the operating system to stop and start tasks—and only when stopping and starting is the natural consequence of the synchronization. As the solution is based on the simple stop and start, it can be implemented under a wide range of operating systems.

The implementation outlined here is now being used by Brown Boveri in a new line of process control systems.

A note on terminology: throughout the paper we use the term 'task' where the term 'process' would seem more natural to scientifically oriented readers. Apart from following the typical terminology of manufacturers, this gives us freedom to talk of 'processes implemented inside a single task' at the end of this paper.

OVERCOMING THE RACE CONDITION

Our first attempt was to synchronize tasks straightforwardly by means of 'suspend' and 'resume'. To our surprise this is not possible directly, as will be shown below.

Consider a task Q which wants to suspend itself until task R wakes it up. Before suspending, Q sets the variable wakeQ to tell R that it expects such a wake-up. The algorithms of Q and R will look like this:

```
task Q: wakeQ:=true;
        {point D}
        suspend
task R: if wakeQ then
        resume(Q)
```

The variable 'wakeQ' must of course reside in an area common to Q and R.

This synchronization example is much simpler than 'general synchronization'. In the general case, the statement 'wakeQ: =true' is replaced by several statements which update and test common variables and decide to continue immediately or suspend. The problem is to perform update, test and suspend decision indivisibly.

But even the simple case above causes problems: assume that Q has reached point D, that is Q has declared 'wake me up' but has not yet suspended. At this moment it may happen that R gets the CPU and runs. R may then try to resume Q, but in vain, because Q has not yet suspended. A little later, Q will suspend itself although R has already responded to 'wake me up'. This is the *race condition* described by Lampson.[7]

If 'suspend' and 'resume' are the only primitives for synchronization, there seem to be

two ways out of the problem. One way is to arrange for R to repeat the 'resume(Q)'. This clearly is a form of busy waiting. Another way is to let R force Q to bypass the suspend. This cannot be expressed in a high-level language, but it is often possible on the level of machine language.

We will first describe these two repairs of the race condition. Later we will show how each of the repairs can form the basis for a lock controlling mutual exclusion. Finally, we will use the lock to complete the general synchronization problem, making sure that common variables are tested and updated indivisibly.

Repair 1: extended resume

Many practitioners overcome the race condition by letting all tasks run periodically. For instance, R will periodically repeat the algorithm above, and sooner or later Q will be resumed. This, however, is an example of *ad hoc* programming, because the algorithm relies on non-local scheduling of the task. Such principles would make it difficult to use the same program piece in different tasks.

We will not depend on periodical tasks or other non-local scheduling, we can do a local form of busy waiting. Let us assume a little help from the operating system:

resume(Q) gives a result showing whether Q could be resumed (i.e. whether Q was suspended at the time of 'resume(Q)').
delay(t) suspends the calling task for t time units.

We can now use busy waiting to program an *extended resume*:

```
extended resume(Q):
    while ¬ resume(Q) do delay (a short time)
```

If R uses 'extended resume' instead of 'resume', the race condition is overcome because R keeps trying until Q has completed 'suspend'. Note that 'delay' will not normally be executed—it only guards against Q being at point D.

The call of 'delay' helps to free the CPU for a while so that Q (and other tasks) can run. This way of programming an extended resume is typical, but variations are possible depending on the actual operating system.

Repair 2: forced bypass

If Q is just before the suspend-instruction (point D), it seems to be too late for R to prevent Q from suspending. However, R may change the very suspend-instruction in Q to a dummy instruction. Under most operating systems, this will require some part of Q's algorithm to be in a common area.

The principle in such a solution can be outlined as follows, with SD denoting the variable holding the suspend/dummy-instruction.

```
task Q: SD:="suspend"; wakeQ:=true;
                execute(SD)
task R: if wakeQ then
        begin SD:="dummy"; resume(Q) end
```

In this solution R is sure that Q continues, because either Q is suspended or Q is forced to bypass the suspend-instruction. Variations of this principle are possible depending on the operating system. For instance, some operating systems may allow R to influence the instruction counter of Q, thereby bypassing the suspend-instruction.

MUTUAL EXCLUSION BY EXTENDED RESUME

A set of tasks must mutually exclude each other from critical regions. A critical region is normally implemented by means of a lock associated with that region. A task must proceed as follows to access the region controlled by the lock CR:

```
lock(CR);
critical region;
unlock(CR);
```

If a task is in the critical region, another task trying to enter will be suspended at lock(CR). When a task leaves the critical region, it must resume one of the tasks possibly suspended at lock(CR).

We will now show an implementation of the lock based on 'extended resume'. The data structure CR consists of two variables in an area common for the tasks:

CR.count: An integer which is 1 when no task is in CR, 0 when one task is in CR and < 0 when one task is in CR and other tasks try to enter.

CR.wake: The set of tasks attempting access to CR. (This variable may, for instance, be represented as a bit pattern with one bit for each task. Tasks attempting access have their bit equal 1.)

We will assume the existence of certain machine instructions for indivisible updating of these variables, as will be explained below.

The operations lock and unlock can now be programmed in this way:

```
lock(CR):   CR.wake:=CR.wake+this task;
            if (CR.count:=CR.count-1) <0 then
              suspend;
            CR.wake:=CR.wake-this task;
unlock(CR): if (CR.count:=CR.count+1) ≤0 then
            begin find a task Q in CR.wake;
              extended resume(Q)
            end;
```

The updating of 'CR.wake' is assumed to be done indivisibly, because many tasks may try to update it simultaneously. This can be accomplished by a bit pattern with one bit per task, or by a common list of one-word task descriptions showing the CR to which the task attempts access.

Similarly, the update and test on 'CR.count' must be done indivisibly. This is done in PDP–11 by the instruction 'dec' and 'inc'.

In order to prove that the algorithm controls the critical region properly, we have to prove the following points (adopted from Dijkstra[3]):

1. If a task is in the critical region, other tasks cannot enter.
2. If a task is stopped outside the critical region and lock/unlock, other tasks can enter the critical region.
3. If more tasks try to enter the critical region at the same time and no task is in the critical region, one of the tasks will be allowed to enter.
4. If one task leaves the critical region while other tasks try to enter, one of them will be allowed to enter.

Let us first prove point (1): if a task is inside the critical region, it will have decreased 'count' to 0. Other tasks trying to enter will make count < 0 and will thus decide to suspend themselves. Count will only be increased when a task leaves the critical region.

Point (2) is easy to prove: being outside the critical region and lock/unlock, the task has decreased and increased 'count' the same number of times.

Point (3) is fulfilled because 'count' is updated and tested indivisibly. This means that one task will complete its handling of 'count' before the others. This task will then enter, while the others will decide to suspend themselves.

Point (4) causes all the troubles. We have to distinguish between two cases: first, assume that the leaving task makes count > 0. This means that no task has yet decided to suspend itself (although some tasks may have included themselves in 'wake'). As a result there is no task to resume, and one task will later be allowed to enter.

Second, assume that the leaving task makes count $\leqslant 0$. This means that at least one task has decided to suspend itself, and it has already included itself in 'wake'. So there is at least one task in 'wake', and a 'Q' can be found in unlock. Tasks that do not try to enter at all have removed themselves from 'wake' before they left the critical region.

However, some tasks may have included themselves in 'wake' but have not yet decreased 'count'. But if they are allowed to continue they will suspend themselves as the leaving task made count $\leqslant 0$.

The Q selected at unlock will be the task allowed to enter, whether it has decreased 'count' or not. The extended resume will thus allow Q to decrease 'count' and suspend as needed. All other tasks trying to enter will be suspended or will suspend themselves when allowed to continue. This completes the proof.

It should be noted that the construct is a lock, and it cannot be used as a semaphore (Dijkstra[3, 4]).The difference is that only one task at a time attempts 'unlock', while two tasks simultaneously might attempt a 'signal' on a semaphore. Trying to use the lock as a semaphore would give wrong results in this case, as the two tasks might select the same task for 'extended resume'.

Whether the lock uses 'fair scheduling' depends on the way 'unlock' finds its Q. 'Fair scheduling' means that a task trying to enter will be allowed to enter sooner or later—no matter how many other tasks try to get access. Fair scheduling can be introduced by means of additional variables in the lock or by letting the leaving task select the first task after itself in CR.wake (cyclically). This does not cause additional problems because only one task at a time tries to leave the critical region.

MUTUAL EXCLUSION BY FORCED BYPASS

The lock of the previous section can be implemented using a forced bypass rather than extended resume. Each task must then have a 'suspend variable' (SD) holding either a suspend instruction or a dummy instruction. SD must reside in an area common for all the tasks. Although the lock and unlock procedures could be shared by all tasks, the suspend variable must be separate for each task (for instance, a two-word subroutine for each task).

The lock and unlock procedures will now look like this:

```
lock(CR):   this task.SD:="suspend";
            CR.wake:=CR.wake+this task;
            if (CR.count:=CR.count-1) <0 then
              execute (this task.SD); {suspend or dummy}
            CR.wake:=CR.wake-this task;
```

```
unlock(CR): if (CR.count:=CR.count+1) ⩽0 then
            begin find a task Q in CR.wake;
              Q.SD:="dummy"; resume(Q)
            end;
```

The proof of this algorithm is similar to the proof above, and we will not show it.

MUTUAL EXCLUSION BY INSTRUCTION SWAP

The principle of forced bypass was originally suggested by A. P. Ravn as a lock combining the suspend variable and the count variable. This lock is the one actually implemented on PDP–11, and it is explained below because it is shorter and has a simpler data structure. However, it may cause problems to implement on certain other computers.

The lock is called *lock by instruction swap*. It consists of two variables:

CR.wake: The set of tasks attempting access to CR: (As before).
CR.SD: A short subroutine holding either the instruction
suspend
or the following combined instruction, called 'close'
CR.SD: ="suspend"; passed: =true

(In PDP–11 the subroutine is entered with the following register contents:
r4 ="emt 377", i.e. "suspend"
r3 =address of CR.SD
Ncondition =0, i.e. passed =false.

Upon return—possibly after a suspended period—Ncondition represents 'passed' and all other registers are unchanged.

The actual suspend instruction is
emt 377; call RSX

The actual close instruction is

mov r4, (r3); CR.SD:="suspend"; Ncondition:=1).

The operations lock and unlock can now be programmed in this way:

```
lock(CR):   CR.wake:=CR.wake+this task;
            repeat
              passed:=false; execute(CR.SD)
            until passed;
            CR.wake:=CR.wake-this task;
unlock(CR): CR.SD:="close";
            if CR.wake not empty then
              resume (a task in CR.wake);
```

The algorithm assumes that instruction fetch and execution are performed indivisibly in CR.SD.

Let us prove as before the four properties of the lock:

1. Point 1: if a task is in the critical region, it has left a suspend instruction in CR.SD. Thus nobody else can enter.

2. Point 2: if a task is stopped outside the critical region, it left a close instruction in CR.SD. Thus some other task may enter.
3. Point 3: if more tasks try to enter at the same time, one of them will be the first to execute 'close'. When this instruction is completed, the task has entered, and the other tasks will find a suspend instruction. (Here we use the assumption of indivisible fetch and execute.)
4. Point 4: if one task leaves and others try to enter, there will be at least one entering task in CR.wake. This task will be resumed at unlock or will be running already, but other entering tasks may be running as well. All running entering tasks will now compete as in point (3) because of the loop in lock(CR). Thus, exactly one task will enter.

As before, the lock cannot be used as a semaphore (not even as a binary semaphore). Implementation of fair scheduling is not obvious here as before, because the task scheduling in the operating system interferes in point (4).

GENERAL SYNCHRONIZATION

By means of locks we are able to solve the general synchronization problem, where we need indivisible update, test and suspend decision on common variables.

Consider a task Q which wants a certain relation to hold between variables shared with other tasks. If the relation does not hold, Q wants to suspend until some other task makes the relation hold. This model of general synchronization could be expressed as follows:

```
task Q: lock(CR);
        if ¬relation then
        begin wakeQ:=true;
          unlock(CR) and suspend
        end else unlock(CR);

task R: lock(CR);
        modify common variables;
        if relation and wakeQ then
        begin wakeQ:=false; resume(Q) end;
        unlock(CR);
```

We assume here that there are several tasks behaving like R. The only problem is to perform 'unlock(CR) and suspend' as an indivisible action.

The general algorithm above is nicely expressed in the language construct called a *monitor* (Hoare[6], Brinch Hansen[1]). Hoare[6] has shown how to implement the construct by means of semaphores. Actually, locks can be used instead, because only one task at a time attempts to 'signal' (unlock) a given semaphore.

Synchronization by extended resume

The lock in the algorithm can be implemented in any of the three ways above. But this still leaves a choice of how to implement the indivisible 'unlock(CR) and suspend'. By means of an extended resume, we can tolerate a delay between 'unlock(CR)' and 'suspend'. This leads to the following solution:

```
task Q: lock(CR);
        if ¬relation then
        begin wakeQ:=true;
          unlock(CR); suspend
        end else unlock(CR);

task R: lock(CR);
        modify common variables;
        if relation and wakeQ then
        begin wakeQ:=false; extended resume(Q) end;
        unlock(CR);
```

This is only a proper synchronization because wakeQ is made **false** in task R before other tasks get the chance to access the critical region.

Synchronization by forced bypass

Instead of extended resume, we can use a forced bypass to tolerate delays between 'unlock' and 'suspend'. The solution is straightforward:

```
task Q: lock(CR);
        if ¬relation then
        begin wakeQ:=true;
          this task.SD:="suspend";
          unlock(CR); execute (this task.SD)
        end else unlock(CR);

task R: lock(CR);
        modify common variables;
        if relation and wakeQ then
        begin wakeQ:=false; Q.SD:="dummy"; resume(Q) end;
        unlock(CR);
```

Note that the placement of 'this task.SD:="suspend" ' is not critical—as it was when constructing the lock by means of forced bypass. The reason is that here we are already inside a critical region guarded by CR.

Note also that the use of 'this task.SD' does not conflict with the lock using the same variable. The reason is that only lock(CR) uses 'this task.SD', not unlock(CR).

Synchronization by instruction swap

In order to replace the extended resume by 'lock by instruction swap', we have to use Hoare's version of the synchronization (in this case the same as Dijkstra's 'private semaphore'[3,4]). We introduce a private lock for each task in addition to the common lock CR. The private lock of Q is denoted sleepQ. Initially sleepQ is in the locked state, and the algorithm becomes:

```
task Q: lock(CR);
        if ¬relation then
        begin wakeQ:=true;
          unlock(CR); lock(sleepQ)
        end else unlock(CR);
```

```
task R: lock(CR);
        modify common variables;
        if relation and wakeQ then
        begin wakeQ:=false; unlock(sleepQ) end;
        unlock(CR);
```

Note that the locks are unlocked by only one task at a time, because unlocking is only performed inside the critical region.

Use of general synchronization

The general synchronization can be utilized to implement semaphores, message communication, etc. For instance, to implement a semaphore we just have to choose common variables which represent the semaphore counter and the queue of tasks waiting on the semaphore. The algorithms can be found elsewhere,[1, 6] so we will not show them here. A more complicated example of general synchronization is shown in the next section.

PROCESSES INSIDE TASKS

It is often convenient to let a task simulate several concurrent processes. This makes sharing of code and data easy and is the natural way to express a re-entrant process.

It is possible to implement such processes and let them synchronize across task boundaries. What is needed is to describe all processes in an area common to the tasks. The scheduling inside each task has to be somewhat primitive, because a process must actively do something (for instance, wait on a semaphore) in order to let other processcs run in the same task.

As an example we will show a *kernel* which implements processes and allows them to synchronize by means of semaphores. Seen from the processes, the kernel consists of two procedures: one to wait on a semaphore and one to signal a semaphore. These procedures must be available from all the tasks—either in a common area or as separate copies.

Any of the tools explained in the previous section can be used. Here we show the algorithm with private locks, which is the version actually implemented.

The following data structures must be in the common area:

Q.actlist	Each task Q has a list (actlist) of active process descriptions, i.e. processes wanting to run or currently running.
Q.proc	The currently running process in task Q. It will run whenever the operating system lets Q run.
Q.sleep	A private lock of task Q. When Q has no process to run, Q will wait on Q.sleep.
P.task	Each process description P has a field specifying the task to which P belongs. The process description will also contain fields for saved registers of P and a field for chaining P to actlist or semaphores.
sem.count	Each semaphore has a counter.
sem.chain	A chain of process descriptions corresponding to processes waiting on the semaphore.
mutex	A lock used by all tasks to implement the indivisible operations on semaphores and actlists.

The kernel algorithm looks like this:

```
procedure wait(sem: semaphore);
begin lock(mutex);
      Q:=this task;
      sem.count:=sem.count-1;
      if sem.count <0 then
      begin P:=Q.proc; remove (P) from: (Q.actlist);
          insert (P) into: (sem.chain);
      end;

      if Q.actlist is empty then
      begin
          unlock(mutex); lock(sleepQ)
      end else

      begin Q.proc:=first in (Q.actlist);
          unlock(mutex)
      end;

      return to Q.proc
end;

procedure signal (sem: semaphore);
begin lock(mutex);
      sem.count:=sem.count+1;
      if sem.count≤0 then
      begin P:=first in(sem.chain);
          remove (P) from: (sem.chain);
          Q:=P.task; wake:=Q.actlist is empty;
          insert (P) into: (Q.actlist);
          if wake then
          begin Q.proc:=P; unlock(Q.sleep) end;
      end;

      unlock(mutex)
end;
```

INPUT/OUTPUT

A task performs input/output by calls to the operating system. From the point of view of the task, an input/output operation can be executed in two modes:

1. Indivisibly (start operation and wait for completion as one call).
2. Asynchronously (start operation and wait for completion as separate calls).

Both modes work well with synchronization between single process tasks. A task can either be running (active), or waiting for synchronization with another task, or waiting for completion of input/output.

However, if each task implements several processes, input/output causes problems. A process can synchronize to processes in the same task or other tasks, but if the process waits for completion of input/output, all other processes in the same task will be suspended as well.

A way out of this problem may be offered by the asynchronous mode. The principle is to delay waiting for input/output until no process in the task is active. Somehow then, the first completed operation must be served first. Means must also be provided for other tasks to resume a task which waits for input/output. These problems have been solved under RSX for PDP–11, but the solution depends heavily on specific facilities of RSX. For this reason we will not show it here.

SYNCHRONIZATION AND ERRONEOUS TASKS

In the preceding sections we have assumed that the only purpose of tasks is to provide parallel execution of programs. Our discussion has concentrated on synchronizing these parallel executions.

However, tasks serve other purposes: the operating system *protects* tasks from destroying each other. Further, it provides *dynamic exchangeability* of tasks, which means that the operator can remove a task and install new tasks. Both purposes conflict to some extent with the synchronization.

In order that tasks can synchronize, they need access to a common area where process descriptions, semaphores, etc. are stored. Thus an erroneous task can completely corrupt these data structures and crash not only itself, but the entire system. This means that tasks participating in the synchronization must be rather reliable, so that they only access the common data structures through the authorized subroutines (the kernel). In principle, this could be accomplished by incorporating the kernel in the manufacturer's operating system.

Further, the tasks must use the semaphores in such a way that they do not deadlock the other tasks, for instance by 'forgetting' to signal a semaphore. As far as we know, the operating system cannot protect against such errors.

The conclusion seems to be that tasks which synchronize cannot be protected against each other.

In practice, we restrict synchronization to tasks of the same reliability hoping that reliable tasks are so well debugged that they do not corrupt the common areas or deadlock each other. Unreliable tasks are not allowed to synchronize with the reliable tasks or access their common area. However, the unreliable tasks can communicate with the reliable tasks through other common areas or through the unruly message passing provided by the manufacturer.

The question still remains whether a reliable, synchronizing task could be exchanged dynamically. This would at least require some care. For instance, the operator may not remove a task while it is inside a critical region, because other tasks will never be allowed to enter. Note that a kind of time-out on critical regions only helps from a superficial view. Other tasks will be allowed to enter after some time, but the critical region was established to guarantee consistency of some data, and this consistency may be lost.

With careful planning of the reliable tasks, dynamic exchangeability may still be provided. For instance, the operator would have to remove one reliable task through another reliable task which locks all critical regions to make sure that no other task is in them, returns records sent to the removed task, etc.

The general impression of this discussion might be that synchronization is the wrong principle for this kind of application. But as far as we know, synchronization is the only available principle for reliable, multiprogrammed systems. And no principles seem available for combining synchronization with protection and dynamic exchangeability.

CONCLUSION

We have shown how well-known synchronization principles can be implemented efficiently under 'primitive' operating systems. The requirements are:

1. A simple stop and start operation (suspend and resume).
2. An area common to the tasks.
3. A few indivisible machine instructions found in most computers.

Within this framework three different implementations have been shown.

We have further shown how to implement several processes inside a single task, although the input/output problem in this case has just been outlined.

ACKNOWLEDGEMENTS

We are grateful to Joseph Muheim of Brown Boveri, Switzerland, for persuading us to search for a solution. Without his interest we might never have started.

REFERENCES

1. P. Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, New Jersey, 1977.
2. O. Caprani, S. Lauesen and U. Ougaard, 'Design principles for dedicated data collection programs', *Euromicro 1978*, 329–343, North-Holland.
3. E. W. Dijkstra, 'Cooperating sequential processes', in *Programming Languages* (Ed. F. Genuys), Academic Press, New York, 1968, pp. 43–112.
4. E. W. Dijkstra, 'The structure of the THE multiprogramming system', *Comm.ACM*, **11**, No. 5, 341–346 (1968).
5. A. N. Habermann, *Introduction to Operating System Design*, Science Research Associates, Chicago (1976).
6. C. A. R. Hoare, 'Monitors: an operating system structuring concept', *Comm.ACM*, **17**, No. 10, 549–557 (1974).
7. B. W. Lampson, 'A scheduling philosophy for multiprocessing systems', *Comm.ACM*, **11**, No. 5, 347–360 (1968).
8. S. Lauesen, 'A large semaphore based operating system', *Comm.ACM*, **18**, No. 7, 377–389 (1975).
9. B. H. Liskov, 'The design of the Venus operating system', *Comm.ACM*, **15**, No. 3, 144–149 (1972)