

THE PROBLEM OF NESTED MONITOR CALLS

Andrew Lister
Department of Computer Science,
University of Queensland,
St. Lucia,
Queensland, 4067
Australia

The concept of a monitor has been developed by Hoare [1] and Brinch-Hansen [2, 3, 4] into a powerful and useful tool for building well-structured and reliable operating systems. Some experience of the author in constructing monitor-based systems [5, 6] has highlighted a problem of implementation which does not seem to have been explicitly addressed in the literature. This article describes the problem and discusses some (inadequate) solutions; its aim is to solicit better solutions from workers in the field.

One of the fundamental attributes of a monitor is that executions of its procedures are mutually exclusive in time. This restriction is a necessary condition for ensuring the integrity of the data and resources administered by a monitor. Looked at from the point of view of correctness proofs, the mutual exclusion attribute allows one to demonstrate that certain invariants remain true under execution of monitor procedures [1, 7]. To enter a monitor, a process must therefore gain exclusion for that monitor; the exclusion is released on monitor exit.

Release of exclusion is also implied by executing a *wait* operation: if this were not the case there would be no way for another process to enter the monitor and perform the corresponding *signal* operation. (*Wait* and *signal* are the process synchronisation primitives suggested by Hoare; slightly different primitives are used by Brinch-Hansen). Strictly speaking, execution of a *signal* operation should also imply release of exclusion, since the process which is *signal*-led is immediately resumed. However, it seems to be common (universal?) practice to place all *signal* operations at the end of procedure bodies, so that exclusion is released in any case by exit from the monitor.

Acquisition and release of exclusion leads to a problem when monitor calls are nested. For example, suppose that a procedure *proc1* of a monitor *mon1* calls a procedure *proc2* of monitor *mon2*. If *proc2* contains a *wait* operation should exclusion be released on both *mon1* and *mon2*, or on *mon2* alone? The two options are discussed below.

First, consider the option of releasing exclusion on only the last monitor called (eg. on *mon2* alone). This has the alarming implication that all monitors back to the original call are inaccessible to other processes. This could have adverse effects on system performance; in particular it could lead to deadlock if the resumption of the waiting process is dependent

on some other process executing a similar sequence of monitor calls in order to effect a *signal* operation. Such a situation might be expected in any neatly hierarchical system in which the control paths leading to "matching" *wait* and *signal* operations pass through the same monitors.

The alternative option, that of releasing exclusion at all levels, requires a means of recording what exclusions a process possesses at the time it executes *wait*. A stack seems the obvious mechanism: execution of *wait* would cause the release of all exclusions currently stacked. However, the resumption of a process after a *signal* operation requires that all these exclusions be restored. It is difficult to see how this can be achieved, since some of the exclusions might at that time belong to other processes.

A further implication of this second option is that any monitor invariant associated with an outer level monitor must hold at the point at which a nested monitor call is made. (In the example above, the invariant for *mon1* must hold when *mon2* is called). This is necessary in order to ensure that any other process which enters the outer level monitor will find the local data in the state it expects. Again, it is difficult to see how this can be achieved in general.

The above analysis shows that both options have severe disadvantages. A rather crude way of avoiding them is to implement a single "global" exclusion mechanism for all monitors, rather than implement "local" exclusion for each monitor separately. If a global mechanism is used then only one level of exclusion need be maintained: the question of how many exclusions to release on a *wait* operation does not arise. Of course the gain in simplicity is not without cost, since the degree of potential parallelism in the system is artificially reduced. This technique has been used by the author in a small pilot operating system [6], where interrupt disablement is used as a global exclusion mechanism. This is successful in a small system on a single CPU machine, but it is doubtful whether it would be acceptable in a more realistic situation.

Another, even cruder, way around the problem is to forbid nested monitor calls completely. This is a severe restriction which is unacceptable in any system which is built hierarchically. It has however been adopted in at least one implementation [8].

The only implementation known to the author in which the nested call problem is tackled head-on, rather than being merely avoided, is that by Brinch-Hansen [3]. In this implementation a local exclusion mechanism is used for each monitor, and a *wait* operation causes release of exclusion on only the most recently called monitor. It is not clear what measures, if any, are taken to avoid the degradation of performance and potential for deadlock mentioned earlier.

The partial solutions described above are clearly far from satisfactory. The author would be pleased to learn of any better solutions: or is the nested monitor call problem intractable?

REFERENCES

- [1] HOARE, C.A.R. Monitors: an operating system structuring concept. *Comm. ACM.*, 17, p.549 (1974).
- [2] BRINCH-HANSEN, P. The programming language Concurrent Pascal. *IEEE Trans. on Software Engineering*, 1, p.199 (1975)
- [3] BRINCH-HANSEN, P. Concurrent Pascal machine. *Technical Report*, Dept. of Information Science, California Institute of Technology (1975).
- [4] BRINCH-HANSEN, P. The Solo operating system. *Software: Practice & Experience*, 6, p.139 (1976)
- [5] LISTER, A.M., & MAYNARD, K.J. An implementation of monitors. *Software: Practice & Experience*, 6, p.377 (1976).
- [6] LISTER, A.M., & SAYER, P.J. Hierarchical monitors. *Proc. 1976 International Conference on Parallel Processing*, pp 43-49. (IEEE Cat. No. 76CH1127-OC (1976).
- [7] HOWARD, J.H. Proving monitors. *Comm. ACM*, 19, p.273 (1976)
- [8] KAUBISCH, W.H., PERROTT, R.H., & HOARE, C.A.R. Quasi-parallel programming. *Software: Practice & Experience*, 6, p. 341, (1976)