Yugoslav Journal of Operations Research
8 (1998), Number 2, 323-329

UNINTERRUPTABLE AND OTHER REGIONS

Miroslav HAJDUKOVIĆ, Danilo OBRADOVIĆ, Branko PERIŠIĆ

Faculty of Engineering, University of Novi Sad
Fruškogorska 11, 21000 Novi Sad
Yugoslavia

Abstract: This paper examines the correspondence between exclusive variables, accessed by different processes, and atomic variables, accessed by interrupt handler routines and by different processes. The differences between these two kind of variables are described. Atomic variables handling primitives are introduced, and the implementation of these primitives is discussed. Generalization of the critical regions idea is proposed, and the usefulness of different accessing policies to shared variables is explained.

Keywords: Concurrent programming, shared data, critical regions, signaling regions, monitors, interrupts, interrupt handling.

1. INTRODUCTION

Multiprocessing in a shared memory monoprocessor with preemptive scheduling leads to race conditions and is troublesome when different processes access the same shared variables (in this paper such variables are called exclusive variables). For example, if a process producer writes data in an exclusive variable to be read by a process consumer, it is possible for the consumer to read an erroneous combination of new and old data. To prevent this, it is necessary that accesses to the exclusive variable be mutually exclusive. Even with mutually exclusive accesses to the exclusive variable, unexpected behavior of the producer and the consumer is still possible. For example, if the producer successively writes the exclusive variable, then some data are lost. On the other hand, if the consumer successively reads the exclusive variable, then the same data are read several times. To prevent this, it is necessary that accesses to the exclusive variable be conditionally synchronized.

Program sections in which only one process at a time is allowed to be active are called critical sections [6]. The mutual exclusion of critical sections is provided by variables of a special kind, called semaphores, and by two special operations P and V, defined for such variables [7], [8]. The first operation should be used at the beginning of each critical section, and the second should be used at the end of each critical section. The *P* operation stops the process activity at the beginning of a critical section if there is another process already active inside some other critical section. The *V* operation continues the stopped activity of one of the processes waiting to enter one of the critical sections. It is possible to achieve conditional synchronization in a similar manner. The main drawback of the *P* and *V* operations is their potential for misuse, which cannot be discovered by the compiler. So, an idea emerged [1], [2], [11] not to protect critical sections of programs, but to protect exclusive variables. To achieve this, variables defined as exclusive must be used inside of a special statement, called a (conditional) critical region. So, the compiler can detect any misuse of exclusive variables, and it is responsible for enforcing that all critical regions (applied to the same exclusive variable) are mutually exclusive. The drawbacks of critical sections are poor readability of the programs and inefficient implementation [5]. Poor readability of the programs is caused by permission to use exclusive variables (inside critical regions) through out an entire program. Inefficient implementation was the consequence of the implicit signaling of conditions needed for conditional synchronization.

The troubles with critical regions were fixed by the invention of monitors, and by using explicit signaling inside monitors [3], [12]. A monitor is a module which encapsulates exclusive variables with the operations that act on them. All such operations are implicitly mutually exclusive (that is enforced by the compiler). For explicit signaling it is necessary to introduce variables of a new type, called condition variables. Besides that, two operations, wait and signal, are defined for such variables. The first operation stops the activity of a process until some condition is fulfilled, and the second operation makes possible the continuation of the stopped activity when the condition is fulfilled. Explicit signaling raised the dilemma whether (after the signaling) to continue the activity of the signaling-process (signal-and-continue semantics) or to continue the activity of the signaled process (signal-and-return semantics). Such a dilemma is a result of the (inappropriate) connection between conditional synchronization and scheduling [13]. Besides that, the monitor concept unnecessarily connects data encapsulation with mutual exclusion. That causes serious negative consequences [13]. To avoid such consequences it is suggested to achieve mutual exclusion by using special regions inside data encapsulation modules. Such regions are called signaling regions [13] because conditional synchronization is based on conditional variables and wait and signal operations. Signaling regions have signal-and-continue-but-return semantics [13] (the signaling process leaves its signaling region and continues its activity if it has the highest priority, while the signaled process exclusively gets permission to enter its signaling region).

Approaches used for process synchronization accessing the same shared variables are not suitable for the synchronization of interrupt handling routines and processes accessing the same shared variables. The shared variables, accessed by interrupt handling routines as well as by processes, constitute a special subclass of shared variables. In this paper such shared variables are called atomic variables. The synchronization of accesses to atomic variables has practical importance. For example, when the *x* and *y* coordinates of an object are stored in an atomic variable by an interrupt handling routine, to be read by a background process in charge of displaying the object's position, it is important that all accesses to such atomic variable be mutually exclusive. This kind of mutual exclusion must be achieved by interrupt disabling, to prevent changing of the atomic variable by the interrupt handling routine while the background process is reading it (such mutual exclusion prevents displaying the object in the wrong position). Besides that, it is important that the background process can stop its activity in order to expect the event of changing the object's position. Also, it is important to enable continuation of the background process activity. In the meantime, other processes, not involved in displaying the object's position, are active.

The rest of this paper explains the usage of atomic variables and uninterruptable and other kinds of regions, and compares atomic and exclusive variables.

2. ATOMIC VARIABLES AND UNINTERRUPTABLE REGIONS

Atomic variables and uninterruptable regions are described in the context of the programming language C used for their implementation.

The consistency of atomic variables is protected only if all accesses to them are made under disabled interrupts. This is, by definition, true inside interrupt handling routines. Such routines are defined as functions, designated by the keyword interrupt (interrupt functions do not return values). An interrupt function becomes the interrupt handling routine after its address is placed into the interrupt table. This is impossible without the special interrupt_set function.

Uninterruptable regions are introduced to disable interrupts while processes are accessing atomic variables. Such regions begin with the keyword uninterruptable, and end with the keyword back. These keywords bracket statements to be executed under disabled interrupts. The first keyword causes the disabling of interrupts, and the second keyword returns interrupts to the state before the execution of the uninterruptable region started. The disabling of interrupts inside of an uninterruptable region is in effect only for an active process (the context switch potentially causes interrupt enabling).

All uninterruptable regions are mutually exclusive and accesses to atomic variables are allowed only inside uninterruptable regions (and, of course, inside interrupt function bodies). To enforce this rule, all atomic variables are marked by the keyword atomic. Atomic variables are defined as structures with the keyword atomic preceding the description of all fields.

To stop process activity inside an uninterruptable region until some event happens, it is necessary to name such event. This is the purpose of event type variables. The process stops its activity inside the uninterruptable region by calling the expect function. The only argument of such a call is an address of the event variable that names the expected event. The events are connected to interrupts and they are notified from interrupt functions. Notification of event happening is fulfilled by a notify function call. The only argument of such a call is an address of the event variable, which names the event. Notification of event happening enables the process expecting the event to continue its activity. It is important to stress that interrupt handling routines are more urgent than the processes, and this causes notify function calls to have signal-and-continue semantics. This means that processes continue their activity only after the notifying interrupt handling routine is finished.

Delaying the activity of a process for some number of time units is a good example of atomic variables usage. The atomic variable tick:

```
static struct {
          atomic
          unsigned long countdown;
          event alarm;
} tick;
```

contains the countdown and alarm fields. The first field is intended to contain a number of time units, while the second field names the delay interval expiration event. The initial value of the countdown field is 0 (the tick variable initialization is not shown).

To delay its activity, processes call the delay function. The only argument of such a call contains a number of time units:

```
void delay (unsigned long duration)
{
          if (duration > 0)
                    uninterruptable
                              tick.countdown = duration;
                              expect(&tick.alarm);
                    back;
};
```

(the delay function enables only one process at a time to postpone its activity).

The positive countdown field is decremented whenever a time unit expires. When this field is decremented to 0, the delay interval expiration notification occurs. The clock function describes such interrupt handling:

```
static interrupt clock (void)
{
          if ((tick.countdown > 0) && ((--tick.countdown) ==0))
                    notify(&tick.alarm);
};
```

(placing the clock interrupt function address into the interrupt table is not shown).

The tick atomic variable, the delay function and the clock interrupt function constitute a module. The delay function (and the module initialization function, which is not shown) is the only visible part of that module.

3. IMPLEMENTATION OF ATOMIC VARIABLES AND UNINTERRUPTABLE REGIONS

Atomic variables and uninterruptable regions are implemented as part of the programming language conCert (concurrent C for embedded real time) [9], [10]. The programming language conCert is an extension of the programming language C. This extension is based on the C preprocessor.

Atomic variables are ordinary structures, with the exception that their usage is limited to uninterruptable regions and interrupt functions.

Interrupt functions are ordinary functions (before their execution the conCert executive, which is part of every conCert program, saves all processor registers and restores them after interrupt function execution).

Behind the keyword uninterruptable there is a hidden function call. Its role is to save the status register and to disable interrupts. Behind the keyword back there is a hidden functional call, too. Its role is to restore the status register.

The event type corresponds to the process descriptor's list head. Each expect function call inserts the active process descriptor into this list, and causes a context switch as well. The descriptor is pulled out of this list and inserted into the ready list when the notify function is called by an interrupt handling routine (the context switch eventually happens only after the interrupt handling routine finishes).

The solution to the process activity delaying problem, presented as an example of the usage of atomic variables and uninterruptable regions, is a very simplified part of the conCert executive code in charge of postponing process activities.

4. ATOMIC VARIABLES VERSUS EXCLUSIVE VARIABLES

There are several similarities between atomic and exclusive variables. Both of them are shared variables, therefore special attention is necessary to protect their consistency. Consequently, both of them can appear only in a special kind of region. Besides that, primitives of a special kind (expect/wait) are necessary to stop the activity of processes (until some event has happened/until some condition becomes true). Also, primitives of a special kind (notify/signal) are necessary to continue the activity of processes (after some event has happened/after some condition becomes true). But, there are also very important differences between the implementation of the region required by atomic variables, and of the region required by exclusive variables, as well as between their primitives, controlling the activities of processes.

The atomic variable region relies on the disabling of interrupts. Such disabling is connected to the process activity. The exclusive variable region only partially relies on the disabling of interrupts, and gives the right of accessing an exclusive variable to only one process. Such right is connected to the process itself, not to its activity. This difference means that context switches are not generally acceptable inside the atomic variable region, while they are acceptable inside the exclusive variable region. In the case of the atomic variable region, context switches are acceptable only if the atomic variable is in a consistent state.

The expect primitive as well as the wait primitive causes a context switch, but the former potentially enables interrupts, while the latter takes the right of accessing exclusive variable from the active process.

The notify primitive notifies the happening of the event, while the signal primitive signals that the condition is true. The notify primitive does not cause a context switch (the event cannot be canceled, therefore there is no need to hurry with activation of the process, expecting the happening of the event). Several successive notify calls are possible. The signal primitive can cause a context switch (to give the process a chance, waiting for the condition to become true, to detect that the condition is true). Several successive signal calls could be troublesome (could cause a form of busy waiting [10]).

The similarities between atomic and exclusive variables are misleading because their differences rule their usage as well as their implementation.

5. OTHER REGIONS

Critical regions and signaling regions are invented to enable the compiler to check the consistency of shared variable usage. The uninterruptable regions are based on the same idea. All three kinds of regions offer mutually exclusive access to the shared variables, although they differ in implementations of the mutual exclusion.

Besides needs for exclusive regions, there are examples of shared variable usage with less restrictive access policies. For example, in the five philosophers problem [7] it is necessary only to prevent two philosophers from using the same fork at the same moment. There is no reason to prevent two philosophers from using different forks at the same time. Similarly, in the readers and writers problem [4] mutual exclusion is obligatory among readers and writers, as well as among writers, but not among readers. For expressing different access policies to shared variables it is possible to extend the regions idea by entry and exit protocols. The role of such protocols is to define specific access policies to shared variables. Mutual exclusion is necessary only for entry and exit protocols, but not for statements bracketed by these protocols. In this way it is possible to use shared variables in less restrictive ways, yet still under full compiler control.

6. CONCLUSION

Uninterruptable regions protect the consistency of atomic variables in a similar way as signaling regions protect the consistency of exclusive variables, but there are important differences between their usage and implementation. Atomic variables and uninterruptable regions simplify interrupt handling. They played a very important role in the conCert executive design and implementation.

REFERENCES

[1] Brinch-Hansen, P., "A comparison of two synchronizing concepts", *Acta Informatica*, 1 (1972) 190-199.

[2] Brinch-Hansen, P., "Structured multiprogramming", *Comunications of the ACM*, 15 (1972) 574-577.

[3] Brinch-Hansen, P., *Operating System Principles*, Prentice Hall, 1973.

[4] Courtois, P.J., Heymans, F., and Parnas, D.L., "Concurrent control with 'readers' and 'writers'", *Communication of the ACM*, 14 (1971) 667-668.

[5] Courtois, P.J., Heymans, F., and Parnas, D.L., "Comments on 'A comparison of two synchronizing concepts' by Brinch-Hansen", *Acta Informatica*, 1 (1972) 375-376.

[6] Dijkstra, E.W., "Solution of a problem in concurrent programming control", *Communications of the ACM*, 8 (1965) 569.

[7] Dijkstra, E.W., "Cooperating sequential processes", in: F. Gennyus (ed.), *Programming Languages*, Academic Press, 1968.

[8] Dijkstra, E.W., "Hierarchical ordering of sequential processes", *Acta Informatica*, 1 (1971) 115-138.

[9] Hajduković, M., *The Programming Language ConCert*, University textbook (in Serbian), 1995.

[10] Hajduković, M., *Concurrent Programming in the Programming Language ConCert*, University textbook (in Serbian), 1996.

[11] Hoare, C.A.R., "Towards a theory of parallel programming", in: C.A.R. Hoare, and R.H. Perrott (eds.), *Operating System Techniques*, Academic Press, 1972.

[12] Hoare, C.A.R., "Monitors: an operating systems concept", *Communications of the ACM*, 17 (1974) 549-557.

[13] Reynolds, C.W., "Signaling regions: multiprocessing in a shared memory reconsidered", *Software-Practice and Experience*, 20 (1990) 325-356.