Stanford Artificial intelligence Laboratory
Memo AIM-261

June 1975

Computer Science Department
Report No. STAN-G- 7 5-50 1

Procedural Events as Software Interrupts

by

Odd Pettersen

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

PROCEDURAL EVENTS AS SOFTWARE INTERRUPTS

by

Odd Pet t ersen

Stanford University
Artificial Intelligence Laboratory *

June 1975

ABSTRACT and Introductory remarks.

The paper deals with procedural events, providing a basis for synchronization and scheduling, particularly applied on real-time program systems of multiple parallel *activities* ("multi-task").

There is a great need for convenient scheduling mechanisms for minicomputer systems as used in process control, but so far mechanisms somewhat similar to those proposed here are found only in PL/I among the generally known high-level languages. PL/I, however, is not very common on computers of this size. Also, the mechanisms in PL/I seem more restricted, as compared to those proposed here.

A new type of boolean program variable, the EVENTMARK, is proposed, Event marks represent events of any kind that may occur within a computational process and are believed to give very efficient and convenient activation and scheduling of program modules in. a real-t ime system. An eventmark is declared similar to a procedure, and the proposed feature could easily be amended as an extension to existing languages, as well as incorporated in future language designs.

Key Words and Phrases:
Scheduling, synchronization, language design, parallel programming, multiprogramming, concurrent processes, process communication, shared variables, events, software interrupts.

CR Categories:. 3.80, 3.82, 4.12, 4.20, 4.30, 4.32, 4.35

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract DAHC 15-73-C-0435.
The view and conclusions contained in this document are those of the author and should not be interpreted as necessarily . representing the official policies, either expressed or implied, of the Advanced Research Agency or the US Government.

:l: Present address:
Stanford Artificial Intelligence Laboratory
Computer Science Dept.
Stanford University
Stanford, CA. 94305

Address after August 1975:
The Technical University of Norway
Div. of Engineering Cybernetics,
7034 Trondheim - NTH
Norway

INTRODUCTION

Scheduling and mutual synchronization of individual parallel activities ("tasks") in a real-time multiprogramming, and possibly multiprocessing, system is governed by conditions of various kinds, as required by participating parallel activities. Some conditions are linked to time events, like

B ← (t > tl) (1)

where:

t is "wall clock time"
tl is some predetermined time
B is the condition, recognized as a binary boolean quantity.

Another type of condition, not linked to time, may be:

B ← (a= 1) (2)

where a, for example, can be representing an external boolean signal, of some finite duration,

A third example represents a common case:

B ← (x > y) (3)

where x and y may be computed quantities.

Conditions of this kind are either generated in some program modules or read in through the input system. They are generally used as activating conditions for program modules, different from where they are generated.

It is well recognized that the fastest response is obtained by use of interrupts, if the origin of events like those mentioned above is some hardware source outside the central processor, and if it is also required that a high degree of central processor utilization be maintained.

None of the languages in general use are known to have similar mechanisms for internally generated events, however, although we can simulate interrupts by different means: We can use trap instructions, special instructions activating the hardware interrupt system, or we can jump directly into some special place in the interrupt handling routine.

Both simulated interrupts and active restarting, like Example 1 below, depend, however, on *actions* in the program module generating the event (source module). We have no means to specify, in the *receiving* program module, what arbitrary conditions we want as trigger, if these conditions occur in a *different* module. The only way a receiving module can discover if some event has occurred in some other program module, is by repeated testing. This is very wasteful in respect of processor resource utilization, and moreover, this waste increases proportionally with required decreased response time. If a module could specify relations like eq. (1) to (3) as *events* which are wanted to subsequently trigger certain effects, and then suspend itself or do something useful, we would have a software feature resembling the hardware interrupt mechanism.

In Example 1, "receiving module" suspends itself by an appropriate monitor call. A subsequent monitor call WAKE in another program can restart a suspended program. In this case, the call **WAKE(TA)** will reactivate program TA to continue at the point immediately following where it was suspended (label LA1).

Repeated testing is illustrated in Example 2, where "receiving module" is Program TB in two alternative versions, **TB1** and **TB2**. A condition is changed in "source module" Program SB. In the case of **TB1**, this program is always active; repeated testing is the processortime-wasteful "busy waiting". **TB2** is potentially less wasteful, but involves a trade-off between response-time and processor time spendings, selectable by some choice of value for tdelay. This seems to be the method used in [2] for recognizing change in specified conditions.

Example 1: Active re-starting from source program.

Program SA:

```
----
x:- ----;
WAKE(TA);
----
----
```

Program T A:

```
      ----
      SUSPEND;
LA 1: y:= ----
      comment reactivating at LA1, performed by call in program SA. Evaluation of some function y:=f(x).;
      ----
```

Example 2:' Testing and "busy waiting",

Program S B:

```
      ----
      x:= ----;
      comment some function evaluation changing x;
      ----
      ----
```

Program T Bl:

```
      ----
LB1:  if x < x1 then go to LB1;
      y:= ----; comment some function y=f(x).;
      ----
```

Program TB2:

```
      ----
LB2 : if x < x1 then begin
            SUSPEND(tdelay);
            comment suspend for duration tdelay, then test again.;
            go to LB2 end;
      y:= ----; comment some function y=f(x).;
      WI--
```

It is my intention, with the present paper, to present *procedural events* as a more efficient method to perceive expected conditions. I want to show that such procedural events can be implemented relatively easily, and that they can give substantial benefits in ease of programming and scheduling of independent but interacting parallel activities in a **multiprogramming/multiprocessor** system, particularly real-time (RT) systems. Such events constitute a real software counterpart to hardware 'interrupts, and I call them *procedural* because they resemble procedures in the way they are declared and evaluated.

Short reaction and processing time is a major requirement, if software "interrupts" are to resemble hardware interrupts; this requirement is considered seriously and it makes internal testing and "busy waiting" quite superfluous and obsolete.

To gain speed during run-time, some conditions are prepared during compilation time. Thus, it requires some minor modifications and additions to the compiler. If not realizable for existing systems, it should be very easy to include in new systems, however.

Program elements for synchronization and scheduling, applying procedural events, are covered only summarily in a following section, since the author has **considered** this problem in a separate paper **[1].**

DEFINITION OF "EVENT"

I find the following definition of the term *event* applicable:

> An *event* is a significant discrete occurrence or incident which is intended to **affect** some program execution in a planned manner. The source of an event can belong logically to an entity distinctively apart from the affected program unit or units. An event itself occurs instantaneously and is of infinitesimal duration. The fact that an event has occurred is indicated by an *eventmark* which is a binary-valued program variable of type **Boolean.**

Additional explanation to the definition:

The occurrence may be external or internal: The effect of an external event will be transmitted through the processor's input interface system as an interrupt request signal or a signal actively read by some program statements. The source of an external event will generally be of some physical nature.

An internal event is characterized by some condition change within a program as a consequence of some program action. With relation to the effect Of an event, it is not distinguished between external or internal nature of its origin.

Evidently, relations like (1) to (3) are covered by this definition, B being an *eventmark*.

Time oventmarks are usually defined, and tested, according to eq. (1). Such an eventmark is "turned off" again, i.e. made false, by giving tl an updated value corresponding to the next future occurrence.

I have, hitherto, linked eventmarks to only a few simple examples of defining equations, like (1) to (3). The **beauty** of the concept is, however, that any syntactical correct boolean expression may be used, **defining** a particular eventmark. A component of a defining boolean expression could also, for example, be representing an external event, input as hardware interrupt. There is virtually no limitation, although unnecessary complicated expressions will take some more run-time to evaluate, though usually not excessively more time.

DECLARATION OF EVENTMARKS AND BLOCK STRUCTURE OF THE LANGUAGE

Eventmarks are global variables, common for at least **two** rest-time programs, the one(s) where the event occur(s) and the one(s) that is (are) triggered by the event. Eventmarks must be ***declared,*** similar to variables and procedures. A declaration of three eventmarks may look like:

eventmark **B:=t>t** 1; **C:=avb∧e;** D:=qvx>y *end;* **(4)**

A block structure which may be incorporated in the language, should be available for use to its **full** advantage, also in combination with procedural events. Like any other type of declared variable, an **eventmark** should have a scope confined to the block where it is declared. If all variables in the defining declaration-equations are treated as global, their scope will be the entire system of programs, and no problems would result.

It could be desirable to put the declaration together with the source module(s), however, such that **the** free variables, or some of them, be local to a block comprising this (these) module(s). The purpose could, for example, be to screen these variables from the remaining program including the receiving module(s). **The** eventmark must be common to both source and receiving module, however; thus the. **eventmark(s)** would belong to a comprising block, i.e. at a higher level in the block hierarchy.

This consequence contradicts the rules of possible block levels in usual procedure declarations, whore the free variables may belong to a higher block level, whereas the dependent variable name, i.e. the procedure name itself, always attains the level of the block heading containing the declaration.

The best solution to this dilemma seems to be that the eventmark declaration and the procedure declaration are considered slightly different in one respect, proposed below:

All variables involved in an eventmark declaration should be declared separately, as variables, either at the **same level** or at a higher (encompassing) level related to the point where the **eventmark** declaration is placed. Thus, the eventmark name itself must be declared elsewhere, as a boolean variable, either in the **same block-head** or at a higher level. Then, the blocklevel-structure of the participating variables may be **chosen** according to the specific need, and the eventmark declaration equation merely serves to define the **arithmetic** relation, besides it defines the previously declared boolean variable as an eventmark and introduces it, as well as the free variables, to the compiler mechanism handling the suggested TEV-table.

The typical block-structure would, for example, look like:

```
begin boolean B, C, D; boolean array F[1:10];
real t, ---;
    begin PROGRAM 1 real t1, x, y; boolean a, b, c, q;
    eventmark B := t > t1; D:=qvx>y end;
    (1---
    x := --- ;

    end;
    begin PROGRAM 2 real p, s, --- ;
    ----
    await D;
    ----
    end;

end main program;
```

Basically, the main reason for the necessity of deviating from identity with procedure declaration is that the latter serves a double function: The introduction and declaration of the procedure name itself, and the definition of the procedure with formal parameters, body, and its relation to the procedure name. For procedures, these two functions of the declaration can conveniently be combined, whereas they need to be separated for eventmarks, in order to satisfy desirable requirements for the scope of the variables.

A NOTE ABOUT IMPLEMENTATION IN COMPILER

The declaration like (4) will tell the compiler that the "free" variables on right hand side of the defining equations are those variables, scattered around in the program, that may cause an eventmark to become true, and these variables are the only variables that can affect an eventmark directly. Thus, the compiler should, immediately upon recognizing the eventmark declaration, generate a temporary table (here called the TEV-table), where the free variables are entries, providing reference to the *eventmark,* according to defining declarations. Generally, one variable may be an argument for more than one eventmark, so that these eventmarks should be linked in the TEV-table.

The eventmark declaration resembles a procedure declaration, since it defines an algebraic expression which is to be used for evaluation of eventmark value. Just like any other algebraic expression, each eventmark will be represented by a parsed tree.

GENERATION OF EVENTS

During the compilation, every time an assignment is encountered, the TEV-table is checked against the assigned variable. If the assigned variable is an entry in the TEV-table, a reference to the eventmark evaluation routine in the operating system is inserted in the generated code stream immediately following the assignment. The run-time consequence is, that every time a new value is computed for a variable that is a free variable, in a definition expression for an eventmark, this eventmark expression is also evaluated, and the eventmark is given a possibly new value.

At *run* time, the operating system (OS) will maintain some dynamic scheduling table, possibly tables, of individual program modules that await some event. Let us here distinguish between internal scheduling tables which the OS uses when RT programs are preempted due to limited processor or memory resources, and the scheduling table(s) that the programmer is concerned with. The former table should be completely transparent to the user, but not the latter. Only the latter table is of concern here and is here called the OS-table (Dynamic Scheduling).

As soon as an eventmark is evaluated, the scheduler is notified accordingly. Thus, the concerned program(s) may be activated very fast, shortly after the activating eventmark became true.

AVOIDANCE OF EXCESSIVE SLOW-DOWN OF EXECUTION

It may be objected, that this eventmark evaluation, automatically inserted at several places in the compiled program, may slow down the execution of the program. This would be a very serious effect which frequently can not be tolerated. Inclusion Of eventmark evaluation is under complete control of the programmer, however, who has a direct and very simple option Of eliminating eventmark evaluation from critical parts of the program (particularly innermost loops and other "bottlenecks"): Instead of using the variable name that represents a free variable (x say) in an eventmark expression, he should, in critical parts of the program, use another variable name (xl say). Then, as soon as desirable and Outside the critical loop, the variable of the eventmark expression is assigned the value of the other:

X := xl; (5)

Obviously, the eventmark evaluation will not take place until eq. (5) is encountered. With relation to computing time, there is no reason to consider implications from eventmark evaluation outside "bottlenecks"; thus the burden upon the programmer, of considerations due to eventmark evaluations, is negligible and coincides with other optimization efforts within "bottlenecks". If computing time and "bottlenecks" are of no concern, the programmer may neglect tricks like (5) completely.

SPECIFICATION OF TRIGGERING EVENTS

When an eventmark has been declared, as'shown in a preceding section, it may freely be specified as the condition for the transition of a program into an executing state. The following examples should serve as typical cases:

* A designated program, PROG, may be started when an eventmark D becomes true:

start(PROG, D); (6)

Obviously, the program containing this statement is a different one from PROG.

* A program may, at any point in its execution, call suspension of itself, specifying an eventmark for subsequent reactivation:

```
----
suspend(C);                                                    (7)
---- comment The program resumes execution at this point when C becomes true;
```

* As shown in [1], synchronization and resource-sharing may be controlled effectively, by the use of conditional critical regions with priorities:

```
region v:=p when B do S                                        (8)
region v:=p do S                                               (9)
await B                                                        (10)
```

where:

- v designates the critical region
- p specifies the relative priority for entrance into the region
- B a declared eventmark specifying the condition for:
  in (8): entrance into competition for the region
  -in (10): continuation in the program, with the succeeding statement.
- S constitutes the statement(s) to be executed within the region.

Statement (9) is unconditional and is equivalent to (8) when B=true.

All statements (8) to (10) cause an activation of a monitor routine which suspends the calling program, The suspension will be in effect until all the following conditions are satisfied:

* Condition B is true (statement (8) and (10))
* p is highest compared to priorities for other programs competing for the same region
* The named region is not presently assigned to another program (statement (8) and (9)).

It follows, that it would be desirable that' the condition eventmark B = true be recognized as soon as possible. The use of "procedural eventmarks" as described here should provide an efficient method, besides it is also very convenient for the programmer.

COMPARISON WITH PL/I SOFTWARE INTERRUPTS

In PL/I, one can specify an arbitrary condition to cause the activation of a certain piece of code:

CHECK (namelist) (11)

where "namelist" is a number of variables and/or labels. (11) is a *condition* and can be used as a part of a statement, for example:

```
ON CHECK(namelist)
     BEGIN;
     ----                                   (12)
     ----
     END;
```

The action block (enclosed by BEGIN; - END;) will be activated when any variable in "namelist" is changed, or any label in "namelist" is passed. This resembles an interrupt mechanism, with software interrupt generated whenever a change is made in a variable, or a certain statement is executed, as specified in the namelist. Generally, however, we do not want to execute a certain piece of code whenever a certain variable is merely changed. Rather, the action is to be performed when, for example, the particular variable is within or outside a certain range, i.e. the action depends on a boolean *function* of the variable(s). This function must be included in the ON- statement, which instead of (12) then becomes:

```
ON CHECK(namelist) IF boolean expression THEN
     BEGIN;
     ----                                   (13)
     ----
     END;
```

The ON-statement is activated every time any change is made in the namelist element(s) and *then* the boolean expression is evaluated, upon which the BEGIN; -- END; block *may* be invoked.

The net effect is somewhat similar to the proposed use of declared eventmarks. There are some differences, though, which at least make the author feel in favor of his method:

A declared sventmark stands out from the rest of the program, so that it is easy for the compiler to generate a fast expression, which in run-time can be executed directly under supervision of the monitor, with monitor priority. The construct (13), however, is more integrated with the rest of the application program, which makes it more difficult to separate the evaluation of the boolean expression from the rest of the code, Unless the compiler analyzes the situation and makes an exception, the whole construct (13) will be compiled to a corresponding piece of object code application program. In relation to scheduling, every software interrupt, caused by the CHECK(namelist) condition will then cause the whole IF -- THEN -- BEGIN; -- END; code piece to be scheduled (queued) for execution, and activated when priorities and other scheduling conditions allow. Since the evaluation of the boolean expression generally takes place substantially more often than the evaluation yields the value *true* (usually the exception), a considerable waste of processor time results. In fact, the operation resembles somewhat "busy waiting" of Program TB2 of Example 2 in a previous section, although the PL/I testing is, in principle, less wasteful because the expression is evaluated only when a variable is changed.

If, on the other hand, the compiler makes an exception for an expression like (13), the resulting operation will be similar to the proposed procedural events. The only difference then is in program notation and its consequences for

i) the clarity and simplicity for the programmer

ii) the compiler construction; ease of implementation.

Regarding ii), it is usually better to avoid exceptions, Regarding i), personal preference may be different, but the author feels that the clarity is enhanced by the modular distinguishing between event evaluation and activated code as is the case for procedural events.

REFERENCES

[1] Pettersen, 0. SYNCHRONIZATION OF CONCURRENT PROCESSES.
Submitted February 1975 for consideration of publication in Comm. ACM.

[2] Dertouzos, M. L. CONTROL ROBOTICS; THE PROCEDURAL CONTROL OF PHYSICAL PROCESSES,
Information Processing 74. North-Holland Publishing Co. 1974 (IFIP 74 Proceedings).