SCHEDULERS AS ABSTRACTIONS--
AN OPERATING SYSTEM STRUCTURING CONCEPT

by

Robert P. Cook

Computer Sciences Technical Report #393

July 1980

SCHEDULERS AS ABSTRACTIONS--
AN OPERATING SYSTEM STRUCTURING CONCEPT

Abstract

In recent years, the emphasis in programming language design has been in the area of data abstraction. This paper explores the uses of scheduler modules and region statements as user abstractions to construct operating system programs. Also, generalized schedulers are presented as alternatives to the monolithic choices for operating system primitives available in current languages such as Mesa, Modula, or Euclid.

Keywords and Phrases: monitors, managers, schedulers, regions, operating systems, system programming languages.

CR Categories: 4.22, 4.31, 4.35

SCHEDULERS AS ABSTRACTIONS-- AN OPERATING SYSTEM STRUCTURING CONCEPT

1. Introduction

A scheduler is an algorithm that determines the order in which competing processes are allowed to use resources such as processors, devices, procedures, data, etc. One of the simplest examples of a scheduler is Dijkstra's[10] semaphore abstraction which can be used both for mutual exclusion and signaling. Even though semaphores are universally available as hardware primitives, they have some disadvantages[4,12] for concurrent programming. However, semaphores are very useful for implementing higher level abstractions.

The most important such scheduling abstraction is the monitor concept developed by Brinch Hansen[2,3] and Hoare[11]. Hoare defines a monitor as a "scheduler consisting of a certain amount of local administrative data, together with some procedures and functions which are called by programs wishing to acquire and release resources". The notation is as follows:

```
[class] monitorname: monitor
   begin
       ...declarations of data local to the monitor;
        procedure procname(...formal parameters...);
                begin ...procedure body... end;
        ...declarations of other procedures local to the monitor;
        ...initialization of data local to the monitor...
   end;
```

The local data defines a resource and the local procedures provide the only mechanism to manipulate the encapsulated data. Thus, a monitor provides its users with an abstraction which automatically protects the resource from uncontrolled access and which hides implementation details as suggested by Parnas[21]. The "class" keyword denotes a monitor type similar to a SIMULA67 class[9] which can be used to declare several monitors with identical structure and behavior, for example, three disk schedulers. However, there are other situations where it is preferable to have a single manager [13,25] for a pool of resources. In fact, as is shown by Silberschatz, Kieburtz and Bernstein[25], a manager and a monitor are really two distinct subclasses of schedulers.

We suspect that many more such subclasses exist. A fundamental question, then, is whether each new abstraction should be a "builtin" high-level language primitive or whether each new abstraction should be defined using existing language extensibility features. In this paper, the latter choice will be explored in the context of the *MOD (Star MOD)[6] programming system, which is an extension of our Modula[7] system. After describing the framework of the language, an implementation of a buffer manager will be examined. Finally, the concepts of a "scheduler module" and a "region" will be defined and their use as mechanisms to build scheduler abstractions will be illustrated.

2. System Overview

Before proceeding further with a more detailed discussion of scheduler modules, the module concept of Modula[28] will be considered as the focal point for the design of the *MOD system. A module usually corresponds to a program abstraction and consists of an external interface specification, data structure definitions, procedures, processes, and an optional initialization part. In *MOD, a module can be used as a type definition or to delineate a lexical scope as in Modula. Therefore, both the information-hiding properties proposed by Parnas and the flexibility of the Simula "class" mechanism are maintained.

In Modula, if the prefix "device" or "interface" is used before the keyword "module", the semantics of the module change. For instance, "interface" denotes a module which semantically is similar to a monitor[11]. *MOD extends the Modula prefix notation to provide the user with the following module types:

```
MODULETYPEDECLARATION ::= type IDENTIFIER = [MODULETYPE] module;
                                           [MODULEBODY]
                                           [begin STATEMENTLIST]
                                           end IDENTIFIER

MODULEDECLARATION ::= [MODULETYPE] module IDENTIFIER;
                               [[MODULEBODY]
                                [begin STATEMENTLIST]
                                end IDENTIFIER]

MODULETYPE ::= MODULETYPEID | scheduler

MODULEBODY ::= EXTERNALINTERFACE
               BLOCKHEADING

EXTERNALINTERFACE ::= [define ELEMENT [,ELEMENT]...;]
                      [export ELEMENT [,ELEMENT]...;]
                      [pervasive ELEMENT [,ELEMENT]...;]
```

```
BLOCKHEADING ::= [import IDENTIFIER [,IDENTIFIER]...;]
                 [DECLARATIONLIST]

ELEMENT ::= IDENTIFIER[(readonly|protected)]
```

The module "type" declaration can be used to construct extended data types as in Simula[9] except that a *MOD programmer has more control over the external interface and protection specifications. The builtin module type, "scheduler", can be used to build synchronization abstractions such as monitors[11], managers[13], or interface modules[28]. A module "type" can be used to replicate an existing module definition or to modify the properties of a module, as is the case with "scheduler" prefixes.

The module's IDENTIFIER names the module and must be matched by the IDENTIFIER following the "end". The DECLARATIONLIST may consist of declarations for constants, types, variables, modules, regions, procedures or processes. The STATEMENTLIST can be used to initialize the module. The module boundary delineates a closed lexical scope which can only be superseded by the explicit specification of "define", "export", "import", "pervasive" or "region". Regions are discussed in Section 7.

The protection attributes, "readonly" and "protected", specify read access or no access, respectively, for exported variables or types. The protection specification is relaxed in two cases. First, the protection is not applied within the exporting module. Therefore, protection forces the user to manipulate exported variables by calling procedures that have also been exported from the defining module. Secondly, a protected variable can be used as a subscript and a protected pointer can be used to qualify a reference to an unprotected type. The utility of these exceptions is discussed in Section 4.

An IDENTIFIER specified in an "import" list causes a declaration from a global scope to be made accessible within the module. The "export" attribute allows a local declaration to be visible at the enclosing lexical level, while "pervasive" makes the IDENTIFIER known at all enclosed lexical levels within the outer scope. The "define" statement is provided as an alternative to "export". It gives the user the ability to list those IDENTIFIERS which can be referenced externally, but only by qualifying the reference with the module name as with the SIMULA[9] class notation. "define" can be used to prevent naming conflicts and to shorten import lists. The ability to specify the external interface for each module is becoming a standard feature of modern programming languages as is demonstrated by its use in Mesa[20], Euclid[16], Alphard[23], Ada[12], etc. MODULETYPEs will be discussed in Section 6 and regions in Section 7.

3. *MOD Synchronization Primitives

*MOD provides the user with the simplest possible mechanisms necessary to implement a critical section and with a flexible set of process scheduling operations. The procedures are listed below:

sm:semaphore

_p_(sm)	the process is queued until sm is true. set sm to false.
_v_(sm)	sm is set to true.

q:queue

link(q [,r])	link the process on q with rank r. the default rank is zero. a process cannot perform two links in succession.
delay	delay the process on q. must be paired with a link.
join(q [,r])	the same as a link followed by a delay.
swap(q [,r])	if no process is queued, continue execution. if r is not specified, select the process with the highest delay rank. otherwise, select the first process with a matching rank. control of the processor is transferred to the selected process.
unlink(q [,r])	same as swap except that control of the processor is not transferred.
awaited(q [,r])	yields "true" if a process with a matching delay rank is waiting.

Each of these operations is justified on the basis of its efficiency and the difficulty that a programmer would have in its duplication. The rank selection option for queues allows the programmer to order process execution based on an examination of an arbitrary data structure. The data structure can be tested to determine the rank used to order a process on a "wait" queue. In a similar fashion, if a unique number is used as a rank, the process to wake up can be calculated before executing the "unlink" statement. The "link" and "delay" statements are provided to allow a process to hold a place in line, to perform some action, and then to delay. If an "unlink" or "swap" occurs between a "link" and a "delay", the process is removed from the queue and is marked as "waiting for delay". When the "delay" occurs, the process is marked "ready" and continues execution. The "swap" versus "unlink" option was included to give the programmer control over process switching. The following example implements a "test and set" operation on a Boolean variable using these primitives.

```
var s:semaphore :=true;
type tsboolean = module;
        define testnset, set;
        import s;
        var b:boolean :=true;
        procedure testnset:boolean;
                begin  p(s);  testnset:=b;
                if b then b:=false  end if;
                v(s)
                end testnset;
        procedure set;
                begin p(s);  b:=true;  v(s)
                end set;
        end tsboolean;
```

The semaphore, "s", is a global variable which is shared by all variables declared with the "tsboolean" type. Since the time spent within the two procedures is quite small, the degree of granularity provided by a single lock should be sufficient. Next, we will discuss how protected types can be used to build managers.

4. A "Manager" Example

Silberschatz, Kieburtz and Bernstein[25] suggest the following criteria for manager construction:

1) It should allow the definition of multiple, identical instances of an abstract data type, mutually disjoint in address space, to be managed as a resource pool.

2) Resource instances should be dynamically allocated from the pool to customer processes, but in such a way that a customer process can neither determine the identity of the particular instance that it has been allocated, nor alter its allocation.

3) It should ensure that at most one process can have access to the address space of a resource instance at any time.

4) Synchronization blocking should not take place when two processes simultaneously attempt to access distinct, previously allocated instances of a common type.

Figure 1 illustrates the use of protected types to implement a buffer pool manager which meets these criteria.

Restrictions 1 and 4 are trivially satisfied by the example.

Figure 1

```
module bufmanager;
        export bufindex(protected), acquire, release, buffer;
        const MAXBUF=1Ø;  BUFSIZ=2Ø;
        type bufindex= Ø..MAXBUF :=Ø;  (*zero is an invalid subscript*)
        subtype NUMBUFS(bufindex)=1..MAXBUF;
        var buffer:array NUMBUFS of
                        array 1:BUFSIZ of char;
            link:array NUMBUFS of bufindex; (*list of free buffers*)
            free:bufindex:=1;  (*head of free list*)
            i:NUMBUFS;         (*used to initialize free*)
            mutex,freex:semaphore:=true;
        procedure acquire(var pnt:bufindex);
                begin
                if pnt=Ø then  (*only empty pointers allowed*)
                        p(freex);
                        p(mutex);
                        pnt:=free;  free:=link[free];
                        if free <> Ø then v(freex) end if;
                        v(mutex);
                     end if;
                end acquire;
        procedure release(var pnt:bufindex);
                begin
                if pnt <> Ø then (*only allocated pointers allowed*)
                        p(mutex);
                        link[pnt]:=free;   free:=pnt;
                        v(mutex);    v(freex);
                        end if;
                end release;
        begin
        p(mutex);
        for i:=1 to NUMBUFS-1 do link[i]:=i+1 end for;
        link[NUMBUFS]:=Ø;
        v(mutex);
        end bufmanager;
```

Since the "bufindex" type is "protected", "bufindex" variables cannot be read, written or copied except within the defining module. The protected variables are similar to capabilities with the protection enforced at compile-time. As mentioned previously, however, "protected" variables can be used as subscripts and pointers (assuming that the referent type is not "protected"). Also, the subscript of "buffer" must match "bufindex" in type. As a result, "buffer" can only be referenced via subscripts that have had their values set in "bufmanager". All variables of type "bufindex" are initialized to zero which is an invalid subscript for "buffer". The "subtype" declaration defines a subrange type which matches "bufindex" and which refers only to valid "buffer" indices. Finally, "acquire" cannot allocate the same buffer twice and a single "bufindex" cannot be allocated multiple buffers simultaneously. Thus, Restrictions 2 and 3 are also satisfied.

5. The Problems with Monitor Implementations

Since the introduction of the monitor concept by Brinch Hansen[2,3] and Hoare[11], its structure and effectiveness for operating system design have come under considerable criticism[1,13,17,18,22,24,25,27]. More recently, Keedy[15] has summarized these objections as well as adding several of his own with the following conclusion:

"The significant point to note is not the individual advantages and disadvantages of these basically similar scheduling mechanisms, but the fact that a standard mechanism is provided at all. The implicit assumption is that all the schedulers in an operating system will be happy to adopt the same basic scheduling strategy. ... <u>We conclude that a simple scheduling mechanism as envisaged by monitors is too inflexible to serve the needs of all schedulers in real operating systems</u>."

We agree with the conclusion. If monitors are considered as a subclass of schedulers, it is obvious that they cannot be used to solve all of the synchronization and exclusion problems, but only those problems that fall into the monitor class. Other criticisms arose because people confused Hoare's sample implementation of a monitor with the class of all monitor algorithms. In fact, Hoare gives several different combinations of implementation details based on the usage situation.

Finally, language designers have dropped the ball by providing users with arbitrary choices for monitor implementations. For example, Modula[28], Mesa[2Ø] and Concurrent Pascal[5] all provide different monitor implementations. It is no more reasonable to assume that a single choice in this area would be any more acceptable than the choice of the "array" as the only data structure in ALGOL. In the next sections, we have combined ideas from the <u>Schemes</u> [19] paper by Mitchell and Wegbreit and from the <u>Alphard</u> [23] paper by Shaw, Wulf and London to construct a simple mechanism for defining arbitrary scheduler abstractions.

6. <u>Schedulers as Abstractions</u>

In Modula[28,7], if the MODULETYPE "interface" precedes the module definition, the module automatically becomes a Modula style monitor. We have extended this idea so that a programmer can use the keyword "scheduler" to define new module types as prefixes. The following examples will illustrate most of the key points:

```
type simple_monitor= scheduler module;
        var s:semaphore :=true;
        procedure entry;
                begin p(s) end entry;
        procedure exit;
                begin v(s) end exit;
        end simple_monitor;
```

Each "scheduler" module must contain an "entry" and an "exit" procedure. The "entry" procedure is invoked each time a procedure within a scheduled module is called. A scheduled module is declared by preceding "module" with a SCHEDULERTYPENAME such as "simple_monitor". When a procedure within a scheduled module terminates, the "exit" procedure in the corresponding scheduler module is automatically invoked. The Figure 2 implementation of a Hoare/Brinch Hansen monitor (HBH_monitor) is an example of a more complicated scheduler which also defines a set of "builtin" operators.

The "entry" semaphore enforces exclusion on the "wait" and "signal" procedures within each monitor instantiation. The use of "join" and "delay" in the "wait" procedure allows a process to hold its place in the queue while it releases the monitor. In the sample implementation, a specific processor switch occurs for each "signal" operation. The switch to the "wait"ing process is necessary to maintain Hoare's proof rules for monitors. An instance of a HBH_monitor can be declared as in this improved version of the alarmclock example from Hoare[11]. We will assume that the "condition" operators have been extended to include a

Figure 2

```
type HBH_monitor=scheduler module;
        pervasive wait, signal, condition(protected);
        type condition=record
                        q:queue
                        end record;
        var mutex:semaphore:=true;
            urgent:queue;
        procedure entry;
                begin p(mutex) end entry;
        procedure exit;
                begin
                if awaited(urgent) then unlink(urgent)
                                   else v(mutex)
                                   end if;
                end exit;
        procedure wait(var c:condition);
                begin
                join(c.q);  exit;  delay;
                end wait;
        procedure signal(var c:condition);
                begin
                if awaited(c.q) then
                        join(urgent);  swap(c.q);  delay;
                        end if;
                end signal;
        end HBH_monitor;
```

"rank" attribute. The "condition" type, "signal" and "wait" are automatically imported into "alarmclock" by the use of the module prefix. A negative rank is used to put events at the smallest future time at the head of the "wait" queue. If no process is waiting for a "tick", the signal is ignored.

```
type alarmclock= extended_HBH_monitor module;
        define tick, wakeme;
        var now:integer :=Ø;
            wakeup:condition;
        procedure tick;
                begin
                inc(now);  signal(wakeup,-now);
                end tick;
        procedure wakeme(n:integer);
                begin
                wait(wakeup,-(now+max(n,1)));
                signal(wakeup,-now);
                end wakeme;
        end alarmclock;
```

The scheduler abstraction can be used to construct a static instance of a module which can act like a global manager or to construct "classes" of scheduler types by using a type declaration. Thus, each user is free to choose an environment-dependent solution to the problems listed in Keedy[15].

If the user decides not to use abstractions like the monitor concept, then new proof rules and programming methodologies must be adopted. For instance, the Mesa[2Ø] "notify" statement (similar to "signal") does not guarantee a context switch to the waiting process. Without a context switch, the process which receives the "notify" has no way of knowing whether an intervening process has destroyed the "notify" invariant. Therefore, the recommended programming style is to surround each "wait" with a "while" loop which tests that the "notify" invariant has been maintained. It is the user's responsibility to maintain the invariants which are required to ensure the valid use of an abstraction. In the next section, we will discuss another aspect of scheduler modules which is designed to improve system integrity in the presence of less than conscientious programmers.

7. Region Specifications

One disadvantage of the modular approach to data abstraction is that all operators are available to all processes within a module name's scope of reference. Some algorithms, such as readers and writers[8], require mutually exclusive operation sets or may require a collection of operations in a certain sequence. The problem is how to be certain that the programmer follows the rules. The *MOD "region" declaration and statement are designed to address this problem. The syntax is as follows:

REGIONDECLARATION::= <u>region</u> IDENTIFIER=[PROCEDUREID],[PROCEDUREID]
[,ELEMENTLIST];

A REGIONDECLARATION is a named export list for a module. Arbitrarily many regions may be declared to form distinct or overlapping operation and variable sets. Note that the ELEMENTLIST has the same syntax as the "export" statement; thus, a region can be thought of as a restricted external interface specification which also dictates access control. The two optional PROCEDUREIDs specify the procedure to call when the "region" statement is entered and the procedure to call on exit. A region reference is written as follows:

```
REGIONSTATEMENT::= region [(ARGUMENTLIST)] VARIABLEREFERENCE;
                         STATEMENTLIST
                         end region [(ARGUMENTLIST)];
```

Since the region entry procedure is anonymous to the region user, the optional arguments to the entry procedure follow the keyword "region". When the region is executed, the ARGUMENTLIST is evaluated, and then the VARIABLEREFERENCE is used to select a particular region instance. At this point, the entry procedure, if present, is called. Next, the STATEMENTLIST is executed followed by the call to the exit procedure. The region IDENTIFIER which is known at compile-time opens a scope for the ELEMENTLIST variables and procedures which is closed when the "end region" is met. Therefore, the region concept provides both a finer degree of referencing and of control than the general module interface specification. The scope control provided by a region is similar to the "inner" feature of Simula[9] and Pascal-Plus[26], but is more general in that multiple regions with different access and scope characteristics can be declared. The exclusion control is an extension of Kammerer's "excluding region" concept[14]. An "excluding region" enforces exclusion at compile-time; whereas a *MOD region performs the checks at runtime. As an example, consider the following implementation of a critical region for a set of variables as proposed by Brinch Hansen[3].

```
module shared_variables;
        export access_a;
        region access_a=enter_a, leave_a, a;
        var a:integer;
            s_a:semaphore:=true;
        procedure enter_a;
                begin p(s_a)  end enter_a;
        procedure leave_a;
                begin v(s_a)  end leave_a;
        end shared_variables;
        .
        .
        .
region access_a;
        inc(a)
        end region;
```

The variable "a" can only be accessed within its corresponding region since "a" is not exported from the module. Also, all of the "a" regions are mutually exclusive by construction.

Next, consider the implementation of the readers and writers example from Hoare[11] in Figure 3. The first observation that we can make is that a programmer cannot access a "data_record" except within a "reader" or "writer" region. It is also impossible for a reader to change a "data_record". In addition, the integrity of the solution does not depend on the programmer to call entry and exit procedures as in Hoare's solution[11]. The only assumption that this abstraction makes is that each region will terminate in finite time.

Figure 3

```
type readers_and_writers=HBH_monitor module;
        export reader, writer;
        region reader=startread, endread, data_record(readonly);
        region writer=startwrite, endwrite, data_record;
        var readercount:integer;
            busy:boolean;
            OKtoread, OKtowrite:condition;
            data_record:record . . .  end record;
        procedure startread;
                begin
                if busy or awaited(OKtowrite) then wait(OKtoread)
                        end if;
                inc(readercount);   signal(OKtoread)
                end startread;
        procedure endread;
                begin
                dec(readercount);
                if readercount <= Ø then signal(OKtowrite)
                        end if;
                end endread;
        procedure startwrite;
                begin
                if (readercount <> Ø) or busy then wait(OKtowrite)
                        end if;
                busy:=true;
                end startwrite;
        procedure endwrite;
                begin busy:=false;
                if awaited(OKtoread) then signal(OKtoread)
                                     else signal(OKtowrite)
                                     end if;
                end endwrite;
        begin
        readercount:=Ø;   busy:=false
        end readers_and_writers;
```

8. Conclusions

This paper presents the concepts of protected types and scheduler modules as general mechanisms for the construction of operating system abstractions such as monitors and managers. The advantages are increased flexibility and control for the systems programmer as well as arbitrary extensibility to adapt to changing requirements. In addition, the region concept is introduced as a means of dividing a module's name space into subsets of secure and controlled referencing environments. Just as scheduler modules can be used to build monitor abstractions, regions can be used to define a variety of critical region abstractions as statements. We feel that these constructs provide the means to successfully address Keedy's[15] criticisms of high-level language primitives, such as monitors, for operating system development.

REFERENCES

[1] Arvind, K.P., Gostelow, W. and Plouffe, W. Indeterminacy, monitors and dataflow. Proceedings of the 6th Symp. on Operating Systems Principles, (Nov. 1977), 159-169.

[2] Brinch Hansen, P. The nucleus of a multiprogramming system. Comm. ACM 13, 4(April 197Ø), 238-25Ø.

[3] Brinch Hansen, P. Structured multiprogramming. Comm. ACM 15, 7(July 1972), 574-578.

[4] Brinch Hansen, P. Operating Systems Principles, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1973.

[5] Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Trans. Software Engineering 1, 3(June 1975), 199-2Ø7.

[6] Cook, R.P. *MOD--a language for distributed programming. Proceedings of the 1st Intl. Conf. on Distributed Computing Systems, Huntsville, Alabama, (Oct. 1979) 233-241.

[7] Cook, R.P. An introduction to modular programming for Pascal users. University of Wisconsin Technical Report 372, (Nov. 1979).

[8] Courtois, P.J., Heymans, F. and Parnas, D.L. Concurrent control with readers and writers. Comm. ACM 14, 1Ø(Oct. 1971), 667-668.

[9] Dahl, O.J. Hierarchical program structures. in Structured Programming, Academic Press, New York, New York, 1972.

[1Ø] Dijkstra, E.W. The structure of the "THE" multiprogramming system. Comm. ACM 11, 5(May 1968), 341-346.

[11] Hoare, C.A.R. Monitors: an operating system structuring concept. Comm. ACM 17, 1Ø(Oct. 1974), 549-557.

[12] Ichbiah, J.D. et al. Rationale for the design of the ADA programming language. SIGPLAN Notices 14, 6(June 1979), Part B, 11-3Ø.

[13] Jammel, A.J. and Stiegler, H.G. Managers versus monitors. IFIP Congress Proceedings, Toronto,1977, 827-83Ø.

[14] Kammerer, P. Excluding regions. The Computer Journal 2Ø, 2(1977), 128-131.

[15] Keedy, J.L. On structuring operating systems with monitors. Operating System Review 13, 1(Jan. 1979), 5-9.

[16] Lampson, B.W. et al. Report on the programming language Euclid. SIGPLAN Notices 12, 2(Feb. 1977).

[17] Lister, A.M. The problem of nested monitor calls. Operating System Review 11, 3(Sept. 1977), 5-7.

[18] Lister, A.M. and Maynard, K.J. An implementation of monitors. Software-Practice and Experience 6, 3(July 1976), 377-385.

[19] Mitchell, J.G. and Wegbreit B. Schemes: a high-level data structuring concept. Xerox PARC Technical Report CSL-77-1, (Jan. 1977).

[2Ø] Mitchell, J.G., Maybury, W. and Sweet, R. Mesa language manual. Xerox PARC Technical Report CSL-79-3, (April 1979).

[21] Parnas, D.L. A technique for software module specification with examples. Comm. ACM 15, 5(May 1972), 33Ø-336.

[22] Parnas, D.L. The influence of software structure on reliability. Proceedings of the International Conference on Reliable Software, Los Angeles, Calif., (April 1975), 358-362.

[23] Shaw, M., Wulf, W.A. and London, R.L. Abstraction and verification in Alphard: defining and specifying iteration and generators. Comm. ACM 2Ø, 8(Aug. 1977), 553-564.

[24] Shrivastava, S.K. Systematic programming of scheduling algorithms. Software-Practice and Experience 6, 3(July 1976), 357-37Ø.

[25] Silberschatz, A., Kieburtz, R.B. and Bernstein, A.J. Extending concurrent Pascal to allow dynamic resource management. IEEE Trans. on Software Engineering 3, 3(May 1977), 21Ø-217.

[26] Welsh, J. and Bustard, D.W. Pascal-Plus--another language for modular multiprogramming. Software-Practice and Experience 9, 11(Nov. 1979), 947-957.

[27] Wettstein, H. The problem of nested monitor calls revisited. Operating System Review 12, 1(Jan. 1978), 19-23.

[28] Wirth, N. Modula: a language for modular multiprogramming. Software-Practice and Experience 7, 1(Jan. 1977), 3-35.