SYNCHRONIZING RESOURCES

by

Gregory R. Andrews

TR 78-360

Department of Computer Science
Cornell University
Ithaca, New York 14853

This work was supported in part by NSF grant MCS77-07554.

SYNCHRONIZING RESOURCES

Gregory R. Andrews

Department of Computer Science
Cornell University
Ithaca, NY 14853
February, 1979

ABSTRACT

A new proposal for synchronization and communication in parallel programs is presented. The proposal, called synchronizing resources, combines and extends aspects of procedures, coroutines, monitors, communicating sequential processes, and distributed processes. It provides a single notation for parallel programming with or without shared variables and is suited for either shared or distributed memory architectures. The essential new concepts are operations, input statements, multiple processes and resources. The proposal is illustrated by solving a variety of parallel programming problems.

Key Words and Phrases: parallel programming, processes, synchronization, process communication, monitors, distributed processing, programming languages, operating systems, data bases.

CR Categories: 4.20, 4.22, 4.32, 4.35.

This work was supported in part by NSF grant MCS 77-07554.

SYNCHRONIZING RESOURCES

Gregory R. Andrews

Department of Computer Science

Cornell University

Ithaca, NY 14853

February, 1979

1. Introduction

There is by now widespread agreement on a few basic concepts of sequential programming: assignment, composition (e.g. ;), alternation (e.g. *if*), and iteration (e.g. *do*). There is also widespread agreement that the process is a basic concept of parallel programming and that process communication and process synchronization are the two other fundamental issues. However, there is not yet agreement on the appropriate mechanisms for parallel programming. This results from the rapid changes in the field, the lack of a widely recognized set of selection criteria, the immense variety of applications and hardware architectures, and the diversity of philosophies about how systems should be structured.

Processes can communicate in two basic ways, directly by exchanging messages and indirectly by accessing shared

variables. Processes can also synchronize in two basic ways, directly by explicit signalling and indirectly by Boolean expressions on shared variables. Numerous communication and synchronization mechanisms have been proposed, each of which combines the above possibilities in different ways: semaphores [7], conditional critical regions [2,13], messages [1,2], event counts and sequencers [20], monitors [2,14], modules [24], path expressions [5,12], managers [16], communicating sequential processes [15], and distributed processes [4]. Underlying each mechanism is a different philosophy about the relationship between processes. The various mechanisms and philosophies can be illustrated by considering the differences between Concurrent Pascal [3], Modula [24], communicating sequential processes [15], and distributed processes [4].

A Concurrent Pascal program contains active processes, which execute statements, and passive monitors, which protect shared variables. Consequently, the language is best suited for hardware architectures where one or more processors share memory. Concurrent Pascal also prohibits simultaneous access to shared variables by requiring that all shared variables be protected by monitors. Synchronization is provided by special queue variables that are explicitly signalled.

Modula is similar to Concurrent Pascal in that processes interact via passive modules, synchronization is

explicit, and the language is intended for shared memory architectures. However, Modula permits simultaneous access to shared variables. It is an an example of a language that provides a tool for mutual exclusion (i.e., the interface module) but gives the programmer the freedom to allow concurrent access.

Hoare's recent communicating sequential processes (CSP) proposal illustrates a philosophy, hence a choice of mechanisms, quite different from that of Concurrent Pascal or Modula. In CSP, a system is a set of processes that interact only by means of input/output commands. As such, CSP is well suited to architectures in which processors do not share common memories (of course, it also can be implemented on shared memories). Communication is direct and synchronization is provided both explicitly by input/output commands and implicitly by Boolean expressions. A key attribute of CSP is that each process can explicitly control when ft will accept an input command.

Brinch Hansen's recent proposal, distributed processes (DP), is similar to CSP in that a system consists of a set of procesaes that interact only by direct communication. As such, DP is also well suited to diatributed architectures. One essential difference between the proposals is that in DP, processes communicate by procedural operations rather than the more primitive IO commands of CSP. Another difference is that in DP <u>any</u> operation in a process can be execut-

ed whenever the operation is called and the process is waiting. In short, DP combines processes and monitors whereas CSP combines processes and message passing.

Each of the above proposals is adequate for its intended purposes and each underlying philosophy has its devout adherents. However, each of the philosophies and proposals has a restricted domain of application. This paper presents a new proposal, called synchronizing resources, that generalizes and unifies the above approaches and, consequently, is equally suited to both shared and distributed memory implementations. It is the result of a search for a common denominator that is both primitive enough to be easily understood and implemented and powerful enough to provide a high-level, structured solution to a wide variety of parallel programming problems.

The remainder of the paper is organized as follows. Section 2 presents the philosophy, structure, and mechanisms of synchronizing resources. Sections 3-6 present solutions to numerous parallel programming problems; many are familiar communication and scheduling problems, but others are included to illustrate the range of application. Section 7 discusses a number of related language issues, including verification and implementation. Finally, Section 8 examines the relation between synchronizing resources and other programming concepts, including procedures and classes as well as other synchronization proposals.

2. Basic Concepts

The main contribution of synchronizing resources is to provide a single notation for parallel programming with or without shared variables. The essential ideas are:

(1) A system, hence a program, is a set of resources that communicate and synchronize via operations.

(2) A resource is a set of one or more processes together with the variables, if any, that they share.

(3) Processes in different resources interact only by operations; processes in the same resource interact by operations or shared variables.

(4) Operations are a generalization of procedures and message passing; they are defined by *in* statements, which are similar to Dijkstra's *if* statements [8], and activated by invocation statements, which are similar to calls. Operations are the mechanism for process communication, and *in* statements are the mechanism for synchronization.

Each of these ideas will now be described in detail.

2.1. Programs

A synchronizing resources program consists of one or more resources. It has the form:

resource {// resource}

where the braces denote zero or more repetitions of the enclosed unit. Many view parallel syatems as consisting of levels (or layers) of virtual machines. At each level, a programmer implements a set of resources by employing operations defined by lower levels. For example, the kernel level of an operating system is implemented by employing hardware operations and in turn provides a set of logical resources and operations for use by other portions of the operating system and user programs. Using synchronizing resources, one can program any of the levels. In short, the synchronizing resources philosophy is that parallel programs provide, contain, and are implemented in terms of resources.

2.2. Resources

A resource is a collection of processes together with the variables they share. It has the form:

```
resource name;

    [define operation names;]
    [var variable declarations;]
    [statement list;]

    process {// process}

    end name
```

where the brackets indicate that the enclosed unit is optional and a statement list consists of one or more statements separated by semi-colons. A resource contains at

least one process; it optionally contains permanent (ALGOL *own*) variables shared by the processes and statements to initialize the permanent variables. Its purpose is to encapsulate its permanent variables and processes. It does so by constructing a wall around them where operations provide the only gates through the wall. Operations are declared by processes (see Section 2.4) and those named in the *define* clause are exported from the resource. Processes external to the resource can invoke the exported operations; processes internal to the resource can invoke operations exported by other resources. (Processes in a resource can also invoke operations declared by other processes in the same resource as will be shown.)

Since a resource can contain more than one process, its permanent variables can be accessed simultaneously. It has become fashionable to preclude all concurrent access to variables but doing so precludes efficiently solving numerous parallel programming problems such as readers/writers [6], in place buffer access [14], and on-the-fly garbage collection [9,10]. Although concurrent access should occur only when it is both beneficial and safe to do so, it seems overly restrictive and unnecessary to prohibit it. Synchronizing resources allows concurrent access but [redacted] within a resource and *only* with respect to the permanent variables. The extent of sharing is therefore localized and identifiable.

2.3. Processes

A process contains a set of variables and a sequence of statements. It has the form:

```
process name;
    [var variable declarations ;]
    statement list
    end name
```

A process executes one statement at a time and terminates when its statement list terminates. The variables declared within a process are local to it and hence can be accessed only by the statements within the process. Processes can also declare operations by means of input statements (see next Section). Operations are automatically exported from the process: they can be invoked by processes in the same resource and, if exported from the resource, they can be invoked by processes in other resources. Together, processes and input statements allow mutually exclusive operations to be programmed. In fact, processes are the sole mechanism needed to provide for mutual exclusion.

A special kind of process is a multiprocess, which has the form:

```
multiprocess name;
    [var variable declarations;]
    statement list
    end name
```

Whereas a regular process has exactly one instance, a multiprocess declares a family of identical processes, each

with its own copy of the local variables. There are as many separate instances as there are distinct processes that invoke the operations the multiproceas declares; each instance is permanently associated with the process that uses it. (If a multiproceas declares no operationa, its family has no members; if it is used recursively, it has a potentially unbounded number of members). A multiprocess combines aspects of re-entrant (pure) procedures, coroutines, Concurrent Pascal classes [3], and Modula modules [24].

2.4. Statements

Six kinds of statements are used in synchronizing resources programs, four for sequential programming and two for communication and synchronization. The sequential statements are:

(1) null — skip

(2) assignment: variable:= expression

(3) alternation: if Boolean command { □ Boolean command} fi

(4) repetition: do Boolean command { □ Boolean command} od

The null statement does not change the value of any variable and always terminates. The assignment statement computes the value of the expression and terminates after assigning it to the indicated variable.

The alternative and repetitive statements are Dijkstra's [8]. Each contains one or more Boolean (guarded) commands of the form:

Boolean expression => statement list

A Boolean command can be executed if its guard (Boolean expression) is true.

An alternative statement specifies that exactly one of the Boolean commands is to be executed. If at least one of the guards is true, an arbitrary one of the corresponding statement lists is executed and the alternative statement terminates; if all guards are false the statement aborts and the executing process terminates abnormally.

A repetitive statement specifies that its constituent commands are to be executed as many times as possible. On each iteration, if at least one of the guards is true, an arbitrary one of the corresponding statement lists is executed; if all guards are false, the statement terminates.

Processes communicate and synchronize by means of two statements:

(1) invocation: operation_name ([actual parameters])

(2) input: in operation command { □ operation command} ni

Invocation is like a procedure call; it requests execution of the named operation. The input statement on the other hand declares, synchronizes, and executes operations.

Each operation command in an input statement has the form:

operation_name ([formal parameters]) [∧ Boolean expression]
[by arithmetic expression] => statement list

An operation command always has an operation_name and statement list; optionally, it has a Boolean expression used for synchronization and an arithmetic expression used for scheduling. The Boolean expression, arithmetic expression, and statements can refer to the formal parameters of the operation, the local variables of the process declaring the operation, or the permanent variables of the resource containing the process.

An operation guard consists of the operation name and Boolean expression. It is true if there is at least one pending invocation of the named operation and the corresponding Boolean expression is true (if omitted, the Boolean expression is implicitly true). If there are two or more pending invocations of the named operation for which the corresponding Boolean expression is true, the invocations are ordered by the values of the arithmetic expression with the minimum value first and the maximum value last (if the by phrase is omitted, the order is undefined). The guard determines whether or not any invocation of the operation can be executed, the arithmetic expression determines which invocation is to be executed first. Therefore, the guard specifies a correctness constraint, and the arithmetic expression specifies a performance constraint that applies once the correctness constraint is satisfied.

Execution of an input statement proceeds as follows.

If at least one of the operation guards is true, an arbitrary one is chosen, the first of the pending invocations is selected (as determined by the arithmetic expression), and the corresponding statement list is executed. Otherwise, the input statement is delayed (hence the executing process waits) until at least one of the operation guards becomes true. The input statement terminates when one of the operation commands has been executed. Whereas an alternative statement chooses one Boolean command if at least one guard is true, an input statment chooses one operation command when at least one guard becomes true.

An invocation names an operation and specifies a set of actual parameters. It is executed in three stages as follows. First, the named operation is activated. Second, at a time determined by an input statement that declares the named operation, it is selected for execution. Third, the body of the operation, as defined by the input command that selects the operation, is executed. When all the above stages have been completed, the invocation terminates.

Operation parameter passing is assumed to have value/result semantics. In particular, the value of each formal parameter becomes that of the corresponding actual parameter when an invoked operation is selected, and the value of eech actual result parameter becomes that of the corresponding formal parameter when the invocation terminates. Formal parameters are assumed to be declared in the

sama manner as in Pascal procedures [23].

Input and invocation are closely coupled. They correspond to singular points in the execution of processes in the sense that the body of an operation is executed by one process on behalf of another. Generally, the processes execute independently but when one requires an input operation and the other invokes the operation, the two processes synchronize and "merge" their execution.

Operations are declared within executable input statements rather than as procedures because this allows one to implement and enforce orders on the execution of operations. (They can, in fact, be declared in more than one input statement.) Consequently, input statements allow one to implement path expressions [5,12] as will be shown in Section 8. Synchronization conditions are specified by Boolean expressions rather than explicit signalling since this leads to more comprehensible programs; explicit synchronization by signalling, if desired, can be built using input and invocation. Scheduling constraints are specified by arithmetic expressions associated with operations in order to provide a powerful tool for selecting among a set of operations that vary only in the value of their parameters. Since efficient scheduling is a major concern of parallel systems, it is important to have adequate scheduling mechanisms in the language rather than to have to build them using the language.

Examples:

(1) in P( ) ∧ sem > 0 => sem:= sem - 1
□ V( ) => sem:= sem + 1
ni

Waits until P is invoked and sem > 0 or until V is invoked. Then the value of sem is either decremented or incremented by one, depending on which operation command was selected. This statement defines and implements the semaphore P and V operations [7].

(2) in P(val:integer) ∧ val>0 ∧ sem-val>0 => sem:= sem-val
□ V(val: integer) ∧ val > 0 => sem:= sem + val
ni

This extends the above example by allowing the invoker of P or V to specify a value by which to decrement or increment the semaphore. The guard for the P operation is true when P is invoked, the actual parameter is positive, and sem - val is positive. This example illustrates the use of formal parameters (hence the values of actual parameters) in guards for operation commands. If two processes have both invoked P(2) and the value of sem is initially 3, then one but not both of the P operations will be selected before an intervening V operation. (With the above statement, an invocation P(x) could be indefinitely delayed if sem were always less than x).

(3) in request (amount:integer)^ free by amount =>

free:= false

[] release( ) => free:= true

ni

Waits until request is invoked and free is true, or until release is invoked. Pending requests are ordered by the values of their actual parameters; when request is selected, the invocation with the minimum value for amount is executed. This statement implements a shortest-job-next scheduler (or a first-come, first-served scheduler if amount is a clock value).

2.5. Variables and Expressions

Variables are assumed to be declared in a manner analogous to Pascal [23]. The examples in subsequent sections will use basic types, such as Boolean and integer, and structured types, such as array and record.

Expressions in statements can refer to any variable local to the process containing the expression, the permanent variables of the enclosing resource, or in the case of input statements, any formal parameter in the operation command containing the expression. Expressions can also reference two special variables, which are implicitly associated with each operation:

(1) ?operation_name - the number of activated and not yet terminated invocations of the named operation;

(2) #operation_name - the number of times the named operation has been terminated (i.e. completely executed).

These attributes of operations are useful for synchronization and scheduling, as will be shown. (They are also easily implemented).

2.6. Initialization and Termination

A synchronizing resources program consists of a number of resources each of which consists of a number of processes. Execution of a program begins by concurrently executing the initialization statements in each resource. Once this has completed, each process is executed concurrently.

A program terminates when every process has either terminated or is delayed at an invocation or input statement. A program terminates normally if no process is delayed at an invocation statement; otherwise it terminates in deadlock. (It is reasonable and common to write processes that repeatedly wait for and execute input statements; this is fine as long as all operations that are invoked are eventually selected and executed).

3. Communication

Two processes can communicate directly if one defines an operation that the other invokes. Direct communication is synchronous and unbuffered: the invoking process waits for the operation to execute before proceeding and the operation's parameters provide the communication storage. In order to allow producer and consumer processes to execute at their own rates as much as possible, a resource that buffers communication can be constructed. The structure of such a communication system (or subsystem) is:

```
resource Producer;
   ...
   process P; var output: message;
        ... send(output)...end P
   end Producer
 //
resource Communication;
   define send,receive;
   ...
   end Communication
 //
resource Consumer;
   ...
   process C; var input: message;
        ... receive(input)...end C
   end Consumer
```

The following examples illustrate three possible implementations of the Communication resource: a one-slot mailbox, a bounded buffer, and a parallel bounded buffer. Each solution will work when used by one producer and one consumer (as shown above) or when used by many producers and/or many consumers.

3.1. Single-slot Mailbox [11]

A single-slot mailbox has one box for the storage of mail. The box is either empty or full. It is initially empty and can then be alternately filled by the send operation or emptied by the receive operation.

Solution:

```
resource Communication;
   define send,receive;
   var box: message;

   process Mailbox;
      do true =>
         in send (m1: message) => box:= m1 ni;
         in receive(var m2: message) => m2:= box ni
      od
      end Mailbox

   end Communication
```

Send and receive alternate since Mailbox first waits for and executes a send and then waits for and executes a receive. The Mailbox process never terminates but if the Producer and Consumer resources terminate, the system of resources will terminate normally. Note that no special variable, such as an empty flag, is required in this solution; the ordering of send and receive is controlled by the ordering of the two input statements that define them. If the Consumer invokes receive before the Mailbox executes the input statement that defines receive, the Consumer waits until receive is accepted; the same comment applies to the Producer.

3.2. Bounded Buffer [14]

A bounded buffer is a mailbox with several slots. It smooths the communication between the Producer and the Consumer when they execute at varying rates. To ensure that the buffer is accessed correctly, assume that send and receive operations are to be mutually exclusive.

Solution:

```
resource Communication;
   define send, receive;
   var buffer: array 1..10 of message;
       front, rear: integer;
   front:= 1; rear:= 1;

   process BoundedBuffer;
      do true =>
         in send (m1: message) ∧ (#send-#receive) < 10 =>
               buffer[rear]:= m1; rear:= rear mod 10 + 1
          □ receive(var m2: message) ∧ (#send-#receive) > 0 =>
               m2:= buffer[front]; front:= front mod 10 + 1
         ni
      od
      end BoundedBuffer

   end Communication
```

Send and receive are mutually exclusive since each is defined by the same process. Send can be executed when the buffer is not full; receive can be executed when the buffer is not empty. The synchronization expressions in the guards for send and receive employ the counter variables implicitly associated with operations: #send is the number of sends that have been executed by BoundedBuffer, #receive is the number of receives that have been executed. A valid invariant for the iterative statement above is: $0 \leq (\#send - \#receive) \leq 10$. When $0 < (\#send - \#receive) < 10$, either send

or receive can be selected for execution.

3.3. Parallel Bounded Buffer [11,19]

The bounded buffer implementation in the previous example is more restrictive than necessary. In order to ensure the integrity of the buffer, it is only necessary to ensure that send and receive do not concurrently access the same buffer slot. When there is at least one empty slot (#send - #receive < 10), and at least one available message (#send - #receive > 0), both send and receive can execute.

Solution:

```
resource Communication;
   define send, receive;
   var buffer: array 1..10 of message;
       front, rear: integer;
   front:= 1; rear:= 1;

   process deposit;
      do true =>
         in send (m1: message) ∧ (#send - #receive) <10 =>
            buffer[rear]:= m1; rear:= rear mod 10 + 1
         ni
      od
      end deposit
       //
   process fetch;
      do true =>
         in receive(var m2: message) ∧ (#send - #receive) > 0 =>
            m2:= buffer[front]; front:= front mod 10 + 1
         ni
      od
      end fetch

end Communication
```

This solution illustrates a resource having two processes that share common variables (buffer, front and rear). It differs from the previous solution only in that

send and receive are defined by separate processes. Consequently, they can execute concurrently, subject to the synchronization constraints, but send excludes other sends and receive excludes other receives. The solution works correctly because neither process invalidates the synchronization expression of the other.

Although one must take care when processes are allowed to access shared variables simultaneously, many problems, such as this one, can be correctly, efficiently, and cleerly solved by using shared variables. Although little is gained by programming a parallel bounded buffer for a system implemented on a single processor, since the logical parallelism cannot actually occur, it would be very efficient relative to the previous solution on a multiprocessor system with shared memory. A basic tenet of synchronizing resources is to allow the solution that is most appropriate for a given application or hardware architecture to be programmed.

4. Scheduling and Allocation

Efficient scheduling and allocation of shared resources is one of the most important concerns of parallel systems. In this section, several familiar scheduling and allocation problems are considered: an alarm clock, readers/writers, data base transactions, and a disk head scheduler. They illustrate different uses of multiprocesses and input statements and different kinds of scheduling disciplines.

4.1. Alarm Clock [14]

Processes often need to be able to delay their execution for a period of time; for example, in polling or real-time applications. The problem is to program an alarm clock resource to facilitate this. The alarm clock defines two operations: wakeme, which delays its caller for a specified interval, and tick, which is periodically invoked by a clock interrupt handler to record the passage of time.

Solution:

```
[redacted] AlarmClock;
   define wakeme, tick;
   var time: integer;
   time:= 0;

   multiprocess userclock;
      var setting: integer;
      do true =>
         in wakeme(interval: integer) =>
               setting:= time + interval;
               wakeup(setting)
         ni
      od
      end userclock
      //
   process systemclock;
      do true =>
         in wakeup(t: integer) ∧ t ≤ time => skip
         □ tick( ) => time:= time + 1
         ni
      od
      end systemclock

   end AlarmClock
```

Users of the alarm clock call the wakeme operation. Since it is defined by a multiprocess, each user process communicates with his own private instance of the userclock process. Each wakeme operation computes the appropriate clock setting and then invokes wakeup. The systemclock waits for invocations of tick and wakeup; tick can always be selected but wakeup can be selected only when its actual parameter is less than or equal to time. When an invocation of wakeup terminates, the invoking instance of wakeme terminates and the corresponding user can proceed. In this example, the userclock multiprocess is similar to a pure procedure.

The above version of the alarm clock may, in practice,

have two undesirable properties. First, the systemclock could possibly delay accepting an invocation of tick if it is busy servicing wakeup operations; this may cause the value of time to become inaccurate. This problem can be solved by defining and executing the tick operation in another process in the alarmclock resource; tick and wakeup operations could then be executed in parallel. The second potentially undesirable property is that there is no guarantee that pending wakeup operations serviced in the right order. This problem can be solved by adding the scheduling requirement by t to the operation command defining wakeup. This guarantees that if more than one invocation of wakeup could be selected (i.e. more than one satisfies t $\leq$ time), one which minimizes t will be selected.

4.2. Readers/Writers [6]

Two groups of processes, readers and writers, share a data base. To protect the integrity of the data base, at most one writer at a time can alter it and no reader can access it while a writer is writing. However, readers can examine the data base concurrently.

Solution:

```
resource ReadersWriters;
   define read,write;
   var database: array 1..10 of item;

   multiprocess RW;
      do true =>
         in read (var value: item; index: (1..10)) =>
                  startread( );
                  value:= database[index];
                  endread( );
            □ write(value: item; index: (1..10)) =>
                  startwrite( );
                  database[index]:= value;
                  endwrite( )
         ni
      od
      end RW
      //
   process allocator;
      var state: integer; state:= 0
      do true =>
         in startread( ) ∧ state > 0 => state:= state + 1
          □ endread( ) => state:= state - 1
          □ startwrite( ) ∧ state = 0 => state:= -1
          □ endwrite( ) => state:= 0
         ni
      od
      end allocator

   end ReadersWriters
```

To access the data base, a process invokes either read or write; many processes can be executing or waiting within the read end write operations concurrently since read and write are declared within a multiprocess. Each read operation invokes startread before and endread after accessing the data base (similarly for write). The allocator process implements the required data base exclusion by recording the state of the data base (-1 = one writer, 0 = no readers or writers, state > 1 = state readers) and synchronizing the four allocator operations appropriately. The solution gives

readers preference in the sense that a reader will not be delayed if the data base is already being read, even if writers are waiting.

An important attribute of this solution, which is not present in numerous other proposed solutions [4,6,14], is that the ReadersWriters resource encapsulates the use of the data base. A reader or writer can only access the data base by invoking read or write, which in turn provides the desired access and guarantees that access is correctly scheduled.

An alternative solution that gives writer preference over readers is achieved by replacing the allocator of the above solution by:

```
process allocator;
   var state: integer; state:= 0;
   do true =>
      in startread( ) ∧ state > 0 ∧ ?startwrite = 0 =>
                              state:= state + 1
       □ endread( ) => state:= state - 1
       □ startwrite( ) ∧ state = 0 => state:= -1
       □ endwrite( ) => state:= 0
      ni
   od
end allocator
```

The only change is that startread can be selected for execution only if there are no active or waiting writers (recall that ?startwrite is the number of invoked but not yet completed executions of startwrite).

Each of the above solutions potentially starves one group of processes; the first could starve writers, the

second would starve readers. To preclude starvation, the allocator can be changed so that it alternates between readers and writers when both are waiting, allowing concurrent reads as long as no writers are delayed, then allowing one write, and so on. This can be achieved as follows:

```
process allocator;
   var state: integer; writerlast: Boolean;
   state:= 0; writerlast:= false;

   do true =>
      in startread( ) ∧ state ≥ 0 ∧ (writerlast ∨ ?startwrite = 0) =>
            state:= state + 1; writerlast:= false
       □ endread( ) => stste:= state - 1
       □ startwrite( ) ∧ state = 0 ∧ (¬ writerlast ∨ ?startread = 0)
            state:= -1; writerlast:= true
       □ endwrite( ) => state:= 0
      ni
   od
   end allocator
```

Boolean variable writerlast indicates whether a writer was the last process to start. When both readers and writers are waiting to start, they alternate turns, depending on the value of writerlast (waiting if necessary for the correct state).

4.3. Data Base Transactions

The previous example considered ways to synchronize the reading and writing of single data base records. However, in data base applications, processing a user transaction may require reading or writing several records. In order to ensure the integrity and consistency of the data base, it is necessary to synchronize entire transactions, not just individual reads and writes. Asaume that processing a transac-

tion consists of first requesting read or write access to the data base, then invoking possibly several read or write operations, and finally releasing control of the data base. The problem is to modify the ReadersWriters resource of the previous example to meet these constraints and to ensure that the data base is requested before it is accessed.

Solution:

```
resource ReadersWriters;
   define request, read, write, release;
   var database: array 1..1000 of item;

   multiprocess Transaction;
      var mode: (r,w,inactive);
      mode:= inactive;
      do true =>
         in request(operation: (r,w)) =>
            if operation = r => startread( )
            [] operation = w => startwrite( )
            fi;
            mode:= operation
         ni;
         do mode = r =>
            in read(var result: item; index: integer) =>
               result:= database[index]
            [] release( ) => endread( ); mode:= inactive
            ni
         [] mode = w =>
            in write(val: item; index: integer) =>
                  database[index]:= val
            [] release( ) => endWrite( ); mode:= inactive
            ni
         od
      od
      end Transaction
   //
   process allocator;
      (* any of the allocators of Example 4.2 *)
      end allocator

   end ReadersWriters
```

In this solution, the variable mode local to each instance of Transaction indicates whether the transaction is

reading, writing, or inactive. The solution illustrates the utility of being able to accept input operations at different places in the body of a process, in this case to enforce the ordering of request, read or write, then release. The reader is encouraged to use this technique to program a solution to the buffer allocation problem described in [14, pp. 554-555].

4.4. Disk Scheduling

A different kind of scheduling problem, which involves ordering the operations invoked by different processes, occurs with a moving head disk. In order to efficiently utilize a disk, it is important to reduce excessive head movement. Of the several possible scheduling algorithms, one of the most attractive is the circular scan (CSCAN) algorithm, which decreases the variance in expected waiting time relative to the more widely known elevator (SCAN) algorithm [22]. In the CSCAN algorithm, the disk head moves across the disk in one direction, servicing requests as it goes, then jumps back to the starting point to begin its next pass. (CSCAN exploits the fact that a disk head can be moved across all cylinders in only about twice the time that it takes to move from one cylinder to an adjacent one.) Assuming that processes wishing to access the disk invoke operation doIO and pass the index (c) of the desired cylinder and other data as parameters, the problem is to program a disk resource that utilizes the CSCAN algorithm.

Solution:

```
resource disk;
   define doIO;
   var buffers, etc.

   process driver;
      var position: (1,maxcylinder); position:= 1;
      do true =>
         in doIO(c: (1.maxcylinder);...)
            by (c-position) mod maxcylinder =>
            position:= c;
            start IO on disk;
            wait for interrupt
         ni
      od
      end driver

   end disk
```

In the above solution, the driver process repeately waits for, selects, and executes doIO operations. If more than one doIO has been activated, driver, if possible, selects the one that requests access to the cylinder closeat to the current disk head position in the forward direction (c-position $\geq$ 0); otherwise driver selects the one that is closest to the initial position of the disk head. (The scheduling expressions assumes that mod can take a negative argument, e.g. that -1 mod 3 = 2). The reader is encouraged to write a scheduling expression that implements the elevator algorithm. (Hint: use the function sign(n), which returns 1 if n $\geq$ 0 and -1 if n < 0.)

Note that in the above solution, the disk driver schedules itself. No separate scheduling "monitor" is required since the by phrase allows the driver to select the appropriate invocation of doIO. However, if disk traffic is

heavy, it may be desirable to schedule future disk accesses in parallel with performing one access. In this case, a scheduler process is added to the disk resource and the user and driver interfaces are changed appropriately.

5. [redacted] and Device Control

As argued in Section 2.1, any parallel system consists of one or more layers of virtual (software) machine built on top of a physical (hardware) machine. In fact, what appears to be hardware at one level may actually be implemented at least partially by software (e.g. virtual memory). Hardware and software systems have many common characteristics: they each contain processes (processors), they may share memory, and they are connected by communication channels. If a language is intended for programming systems that control hardware, it should also be possible to use the language to write programs that describe the behavior of the hardware itself since this unifies the concepts of physical and virtual machines and leads to cleaner, more efficient hardware/software interfaces. This section contains three examples that illustrate the utility of synchronizing resource for hardware description and device control.

5.1. Computer System

Consider a small computer that has a central processor, one device and a memory that is shared by the device and central processor. Assume the device is an independent pro-

cessor that accepts a startIO operation from the CPU, transfers data to or from memory, and signals completion by causing an interrupt. The CPU fetches and executes instructions from memory and accepts interrupts when they are invoked and not inhibited. The computer halts when the halt instruction is executed.

```
resource Computer;

   var memory: array 1..size of byte;

   (* initialization - executed when "Load" button is pushed *)
   load bootstrap program;

   process CPU;
      var IC: integer; inhibit,halt: Boolean; ...
      IC:= load address; inhibit:= false; halt:= false;
      do ¬ halt =>
         if ?interrupt > 0 ∧ ¬ inhibit =>
               in interrupt (code: integer) =>
                  memory[int.location]:= IC;
                  IC:= interrupt handler address
               ni
          □ ?interrupt = 0 ∨ inhibit =>
            IC:= IC + 1;
            fetch and execute instruction in memory[IC]
            (* the instruction may set inhibit or invoke
               startIO or set halt:= true *)
         fi
      od
      end CPU
    //
   process device;
      var storage for parameters and data transfer;
      do true =>
         in startIO(p: params) => save p
         ni;
         transmit data to or from memory;
         interrupt(code)
      od
      end device

   end Computer
```

5.2. Device Drivers and Interrupt Handlers

For increased efficiency and reliability, it has become common to associate a device driver process with each device (or group of identical devices) controlled by a system. A driver process repeatedly accepts an operation requesting access to the device it controls, performs the operation, and (optionally) returns a result. Processing a request involves starting the device and waiting for the device to complete, which is usually signalled by an interrupt. Although the actual implementation of device control and interrupt handling is device and machine dependent, a program that shows the interaction of a device driver and an interrupt handler can be outlined.

Assume that the driver has the form:

```
process driver;
   do true =>
      in doIO(operations) =>
         start device d;
         waitinterrupt(d)
      ni
   od
   end driver
```

where waitinterrupt invokes an operstion defined by the interrupt handler. If actual IO interrupts implicitly cause an interrupt operation to be invoked (by a run-time kernel), the interrupt handler can be programmed as follows:

```
resource InterruptHandler;
   define interrupt, waitinterrupt;

   var done: array device_ids of Boolean;
   done:= false; (* for all device ids *)

   process IH;
      do true =>
         in interrupt(d: deviceid) => done[d]:= true
          [] waitinterrupt(d: deviceid) ∧ done[d] =>
               done[d]:= false
         ni
      od
      end IH

   end InterruptHandler
```

The InterruptHandler uses Boolean flags to synchronize interrupt and waitinterrupt operations. With this solution, process IH cannot be delayed if an interrupt occurs before the corresponding driver is waiting for it. This solution can be changed to give the interrupt operation priority over waitinterrupt by adding (?interrupt = 0) to the guard for waitinterrupt; it is also easy to add interrupt codes to the parameter lists.

5.3. CRT Display

As a final example of a hardware resource, consider a CRT display that continuously refreshes a screen. The points to be displayed on the screen are stored in memory in a grid array. A refresh process within the CRT transfers points from the grid to the screen; in parallel, an update process accepts write operations and alters the grid. The refresh and update processes can logically execute in parallel (and do normally physically execute in parallel) since a

new update merely overwrites a previous part of the screen.

[redacted]:

```
[redacted] CRT;
   define update;

   var grid: array 1..M,1..N of intensity;
       i,j: integer;

   grid:= "blank";  (* for all elements of grid *)

   process update;
      do true =>
         in update(x,y,i: integer) =>
            grid[x,y]:= i
         ni
      od
      end update
    //
   process refresh;
      do true =>
         i:= 1; j:= 1;
         do i ≤ M ∧ j ≤ N => output(grid[i,j]); j:=j + 1
         □ i ≤ M ∧ j > N => i:= i + 1; j:= 1
         od
      od
      end refresh

   end CRT
```

If many processes need to display information on the screen, update can be declared as a multiprocess without affecting the behavior of CRT.

6. Miscellaneous

Two final examples, the dining philosophers and a parallel version of the sieve of Erathosenes, will respectively illustrate the use of formal parameters in input guards and a recursive use of the multiprocess construct.

6.1. Dining Philosophers [7]

Five philosophers alternately think and eat. When a philosopher wants to eat, he enters the dining room and sits in his own chair at a circular table set with five plates of spaghetti and five forks, one between each plate. To eat, a philosopher must pick up two forks, the one on his left and the one on his right. Two philosophers can eat at the same time only if they are not sitting in adjacent chairs. Once a philosopher has finished eating, he puts both forks down and leaves the room.

[redacted]:

```
    [redacted] P0;
       process philosopher0;
          var i: integer; i:= 1
          ... enter(i);
          end philosopher1
       end P0
       //
       (* resources P1 to P4 containing philosophers 1-4 are similar *)
       //
    [redacted] dining room;
       define enter:

       var plate: array 0..4 of spaghetti;
       plate: fresh spaghetti;

       multiprocess chair;
          do true =>
             in enter(i: integer) =>
                get forks(i);
                eat from plate[i];
                releaseforks(i)
             ni
          od
          end chair
          //
       process forks;
          var eating: array 0..4 of Boolean;
          eating:= false;  (* for eating [0]..eating[4] *)
          do true =>
             in getforks(i: integer)∧ ¬ eating[i ⊕ 1]∧
               -¬ eating[i ⊖ 1] => eating[i]:= true
             □ releaseforks(i: integer) => eating[i]:= false
             ni
          od
          end forks

       end dining room
```

In the above program, ⊕ and ⊖ denote module 5 addition and substraction. The program terminates when all philosophers have terminated (they are either full or out of thoughts).

6.2. Sieve of Erathosenes [15]

The sieve of Erathosenes is a method for generating prime numbers. A parallel version of the method is programmed below; it is a direct translation from communicating sequential processes to synchronizing resources of the program given in [15]. It contains a starter process, a Sieve multiprocess, and a limit (say 10000). The starter prints the first prime, 2, and then passes 3,5,7,...,limit on to the first instance of Sieve. Each instance of Sieve accepts a stream of integers, the first of which is prime and is therefore printed, and passes on integers that are not multiples of the first input value.

Solution:

```
resource Primes;

   process starter;
      var n: integer;
          limit: integer; limit:= 10000;
      print(2); n:= 3;
      do n < limit => candidate(n); n:= n + 2
      od
      end starter
   //
   multiprocess Sieve;
      var p,m,mp: integer;
      (* p is a prime; mp is a multiple of p *)
      in candidate(c: integer) => p:= c ni;
      print(p); mp:= p;
      do true =>
         in candidate(c: integer) => m:= c
         ni;
         do m > mp => mp:= mp + p
         od;
         if m = mp => skip
          [] m < mp => candidate(m)
         fi
      od
      end Sieve
```

and Primes

This example is more controversial than the previous one since it uses the multiprocess construct recursively. The first instance of Sieve inputs candidate operations from starter and invokes the candidate operation of a second instance of Sieve; each subsequent instance of Sieve receives numbers from its successor and potentially passes them on to its successor. There are as many instances of Sieve as there are primes less than limit. The program terminates when the starter process has terminated and each activated instance of Sieve is waiting for further candidates (of which there are none).

Relative to the communicating sequential processes solution given in [15], this solution has the advantage that there is no fixed bound on the number of instances of Sieve that are created. Merely by changing the initial value of limit, a different set of primes is generated in parallel. Unfortunately, this solution also has the disadvantage that the exact number of instances of Sieve that are required cannot be determined at compile time; this means that an implementation of recursive multiprocesses must either support dynamic process creation or impose a fixed limit on the depth of recursion. Although this implementation problem is a serious concern in practice, the example does illustrate another aspect of the expressive power of synchronizing resources.

7. Related Language Issues

The main purposes of this paper are to present a new approach to process communication and synchronization and to illustrate its expressive power. However, several related language issues must also be addressed before synchronizing resources can be generally useful as a language for systems programming. This section identifies and discusses the most important of these issues.

7.1. Verification

A rule of thumb in language design is that if a language construct has a complex (or worse yet no) proof rule, then it is undoubtedly difficult to understand and use. Although formal proof rules for synchronizing resources have not yet been developed, the desire to be able to formulate such rules has influenced the language. Since the basic statements (null, assignment, alternation and iteration) are the same as those used by Dijkstra, they have the same proof rules [8]. The new constructs, for which proof rules do not yet exist, are invocation and input statements, processes, and resources.

To the invoking process, an invocation statement has the effect of a procedure call: parameters are passed from the process invoking an operation to the process declaring the operation, the invoking process waits for the invoked operation to terminate, and results are returned. A proof

rule for invocation should therefore be similar to a proof rule for procedure call with value/result parameter passing semantics.

The effect of an input statement is a combination of the effects of procedures and alternative statements. Suppose that an input statement has the form

$$\underline{\text{in}}\ op_1(formals_1) \wedge B_1 \Rightarrow S_1$$
$$\cdots$$
$$\square\ op_N(formals_N) \wedge B_N \Rightarrow S_N\ \underline{\text{ni}}$$

where B_i and S_i are, respectively, the Boolean expression and statement list for operation op_i (<u>by</u> phrases are ignored since they do not affect correctness). Since input statements are non-deterministic, any of the operations that has a true guard can be executed. Therefore the combined effect of several operations must be independent of which operation is executed. If one could prove

$$\{P \wedge formals_i \wedge B_i\}\ S_i\ \{Q \wedge formals_i'\}$$

for each operation i (where P and Q are predicates about variables accessible to the statement and $formals_i$ and $formals_i'$ are predicates about the initial and final parameter values) then a proof rule for input statements would allow one to assert that if P is true before execution of the statement, then Q is true after (provided the statement terminates).

Since a process (or multiprocess) consists of a set of

distinct variables and statements, its effect is the composition of the effects of its statements. Therefore, to develop a correctness proof for a process, one merely composes the proofs for individual statements. One of the virtues of the input statement as a means for synchronization and communication is that each process consists of a sequentially ordered set of statements. Therefore, it is possible to follow the logic of a process by sequentially examining its statements. This is in direct contrast to monitors [14] or distributed processes [4] where execution of statements can be interleaved in time due to waiting and signalling and one is always forced to find invariant relations unaffected by possible interleaving.

The most difficult verification problem is to develop a proof for each resource. Since a resource can contain several processes that share variables, one must (usually) establish an invariant that captures the state of the shared variables and also verify that the proofs of the processes are interference-free, as defined by Owicki and Gries [19] (a multiprocess must not interfere with other instances of itself). If the processes are disjoint, or at least no process references variables altered by another, the interference-free property is trivial to demonstrate. However, if processes share variables, as in the parallel bounded buffer example, great care must be taken both in programming and proving a solution. One must recognize the inherent dangers that shared variables introduce, but if one

is willing to use them carefully when the situation warrants, it should be allowed.

7.2. Implementation

Implementing the four basic statements is straightforward using standard techniques. The challenge is to implement invocation and input statements. Assume for simplicity that a program is implemented on a single processor system. One way to proceed is outlined below; the technique presented can be readily generalized to allow a program to be implemented on a multiprocessor system with or without shared memory.

The basic data structures used to support processes and operations are process descriptors, a ready list, work queues, and operation queues. Associated with each process is a descriptor containing the process' status, execution state (when it is not executing), actual or result parameters (when it has invoked an operation) and linkage to one of the other data structures. The ready list contains the descriptors of all processes that are ready to execute; initially all process descriptors are on the ready list. A work queue is associated with each process that declares operations; an operation queue is associated with each distinct operation. These queues are manipulated as described below.

Operations are implemented via three kernel procedures:

invoke, wait, and reply. Invoke takes an operation name and set of values as parameters; it stores the values in the calling process' descriptor and inserts the descriptor onto the work queue of the process that declares the named operation. Wait delays the calling process until an entry is present on its work queue, then transfers the entry to the appropriate operation queue. Reply takes a process name and set of result values as parameters; it inserts the named process onto the ready list and saves the result values in the process' descriptor (they are returned to the process when it resumes execution).

Given the above three procedures, invocation and input statements can be implemented as follows. An invocation statement calls invoke, passing the appropriate operation name and actual parameter values; the invoking process is then blocked until it receives a reply. An input statement is implemented by searching the appropriate operation queue(s) for an available operation that satisfies the specified Boolean expression. If one is found, the scheduling expression is evaluated (if necessary) and one operation is selected. The appropriate statement list is then executed and a reply is sent to the process that invoked the selected operation. If no acceptable operation is found, the process attempting the input statement calls wait. Once awakened, the process examines the new operation. If acceptable, the operation is executed and a reply is sent; otherwise the process again calls wait and repeats the above algorithm.

If an operation guard refers to a variable that is changed by another process (as in Example 3.3), the process may need to wait for the variable to change This can be implemented by busy waiting, if the process executes on a separate processor, or by having any change to the variable result in an implicit invocation of a special operation that awakens the potentially waiting process. (Synchronizing with respect to shared variables is beneficial and efficient only when the synchronizing processes are executing on separate processors, as was discussed in Section 3.3).

The chief source of overhead in this implementation is the searching and testing involved in finding and scheduling an acceptable operation in an input statement. This overhead does not appear to be excessive although final judgement cannot be made until an actual implementation is completed and analyzed. In any case, this overhead occurs only when an input statement uses Boolean or scheduling expressions. Less expressive mechaniams are by themselves more efficient to implement, but they require that a programmer build and maintain queues in order to solve scheduling problems. Typically, such scheduling queues duplicate information already present in the kernel (such as process names and operation parameters) and are therefore unnecessarily costly.

7.3. Machine Interfaces

Any language for systems programming also needs ways to interface to hardware features. This includes mechanisms to bind variables to fixed locations, to associate operation entry points with hardware signals (e.g., interrupts), to access individual bits in storage locations, and to utilize available hardware priority levels such as for interrupts. Hardware interface mechanisms are by their nature machine dependent; consequently, the appropriate set of mechanisms may well vary from implementation to implementation. However, to enhance software reliability, machine interface mechanisms should be limited to a few, well-defined places in any system program. Modula introduced the device module for this purpose [24]. The related concept in synchronizing resources would be to declare some resources as real resources and to allow hardware attributes to be accessed within them only. The details of this approach have yet to be completed but it appears to be a viable approach.

7.4. Naming

A fourth language issue, which has only been cursorily addressed in this paper, is that of naming. For simplicity, all names have been assumed to be unique. However, this is far too restrictive in large programs, especially those that are occasionally altered. This problem could be alleviated, as long as resource names are distinct, by using the Pascal dot notation to name operations in invocation statements

(i.e. by using resource.operation). The name space of each resource could be further restricted if it were to name the external resources that it wishes to use (e.g. as in Modula [24]).

A related naming issue is how to declare families of resources or processes that are nearly identical. For example, if a system requires several communication resources, as illustrated in Section 3, it should be easy to declare multiple instances, perhaps varying only in the type of the transmitted information or the size of the buffering storage area. One solution to this problem would be to attach an optional range specification to the declaration of a resource (e.g. resource Communication[1..5]); individual instances would then be named by using subscript notation (e.g. Communication[1].send). Resources could also be parameterized by using an approach like that of Concurrent Pascal [23] or Euclid [17]. Although these are important language issues, no specific suggestion is made here.

7.5. Access Control

Another important language issue, especially in operating systems and data bases, is access control. Access control in synchronizing resources is static; namely the scope of each object is determinable at compile time. However, it is often necessary or desirable to allocate objects dynamically, for example to implement a file system. This requires some means to bind names and alter access rights to

objects at run-time. One possible approach is described in [18]: it is based on the concept of a capability and can be readily combined with the language presented in this paper.

8. Summary and Discussion

The novel aspects of synchronizing resources are operations, input statements, multiprocesses, and resources. These facilities generalize many previous parallel programming proposals including procedures, coroutines, classes, modules, monitors, path expressions, message passing, communicating sequential processes, and distributed processes.

Operations combine aspects of procedures and message passing. They are like procedures in that to the invoking process an operation looks and behaves like a procedure call; and to the declaring process, an operation has formal parameters and specifies a statement list. Operations are like message passing in that they cause information to be transferred from one process to another and they may result in execution delays.

Input statements allow processes to wait for several operations at a time, control which operations can be selected (using Boolean expressions), and control the order in which multiple invocations of the same operation are executed (using arithmetic scheduling expressions). Being able to express scheduling requirements using the by phrase appears to be an especially powerful programming tool, as illustrated by the disk head scheduler (Example 4.4). It is similar to the scheduling mechanisms proposed in [21] and has the same advantages. In particular, it alleviates the need for programming scheduling queues and extra scheduling

processes (or monitors).

Input statements also provide a means for implementing path expressions [5,12], which define meaningful sequences of operations. For example, the path expression

path send; receive end

specifies that send and receive operations alternate, with send executing first. The mailbox resource in Example 3.1 exactly meets this specification by sequentially ordering input statements declaring send and receive. Another example of a path expression is

path request; (read* + write*); release end

which specifies that request precedes either zero or more reads or zero or more writes, and that release precedes the next request. The data base transaction resource of Example 4.3 satisfies this specification. Path expressions can also contain Boolean expressions that conditionally enable operations, for example, to specify that in a bounded buffer, send can execute only when there are empty slots. Conditional path elements are implemented by using Boolean expressions to synchronize input operations.

The process construct combines the conventional concept of a process as an active program with the concept of a coroutine. Processes can execute independently, have a

master/slave relationship, or interact as coroutines. For example, the processes in the CRT resource (Example 5.3) execute independently; the bounded buffer process in the Communication resource (Example 3.2) is a slave of the producer and consumer processes; and the Transaction processes in the ReadersWriters resource (Example 4.3) are coroutines with respect to their users. Although not shown here, all of the coroutine examples of [15] have also been programmed in synchronizing resources.

Processes, together with operations, also provide a means to implement monitors, as illustrated by the mailbox (Example 3.1) and bounded buffer (Example 3.2) processes, among others. The mutual exclusion required by a monitor is automatically provided by a process and input statements provide the mechanism for declaring and synchronizing monitor entries. The chief differences between a synchronizing resources implementation of a monitor and an actual monitor are that synchronization is specified by Boolean expressions; that scheduling is specified by arithmetic expressions; and that an operation, once selected, executes to completion. This requires that distinct operations be declared for each set of statements that may be delayed (as opposed to waiting in the middle of a monitor procedure), but yields the benefit that each operation and input statement can be analyzed and verified as an uninterruptable unit.

The multiprocess construct provides a way for several processes to share the same set of operations. Consequently, it generalizes the procedure and module [24] concepts. A multiprocess can be used like a procedure, as in the alarm clock resource (Example 4.1), like a Modula module, as in the readers writers resource (Example 4.2), or like a recursive procedure, as in the primes resource (Example 6.2). Since a multiprocess can have local variables, it can also be used to implement a Concurrent Pascal class [3] as illustrated by the readers/writers (Example 4.3) and dining philosopher (Example 6.1) resources.

The resource construct is the means for encapsulating processes and the variables they share. It provides a way to define an abstract object while hiding details of its representation. This was illustrated by all the examples but in particular by the three versions of the communication resource programmed in Section 3. Since processes in a resource can share variables, it is also possible to implement resources such as a parallel bounded buffer (Example 3.3), data base (Examples 4.2 and 4.3), and on-the-fly garbage collector [9,10], and to ensure that the shared variables are accessed only by meaningful operations. The resource construct also provides an appropriate representation for hardware as discussed in Section 4.

The recent communicating sequential processes proposal of Hoare [15] combines processes and message passing, and

the recent distributed processes proposal of Brinch Hansen [4] combines processes and monitors. This paper has attempted to show that combining processes, message passing, and monitors results in an even more expressive language. The examples presented in this paper provide a measure of the breadth of its applicability and are only a small subset of the examples that have been programmed.

Acknowledgements

David Gries, Carl Hauser, Tom Murtagh, Richard Reitman, and especially Fred Schneider provided very helpful feedback on the ideas presented here and on an earlier draft of this paper. Their assistance is greatly appreciated.

BIBLIOGRAPHY

1. Atkinson, R. and Hewitt, C. Synchronization in actor systems. Proc. Third Symp. on Principles of Prog. Languages, January 1976, 267-280.

2. Brinch Hansen, P. Operating System Principles. Prentice Hall, Englewood Cliffs, NJ, 1973.

3. Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Trans. Software Eng. 1, 2 (June 1975), 199-207.

4. Brinch Hansen, P. Distributed processes: A concurrent programming concept. Comm. ACM 21, 11 (Nov. 1978), 934-941.

5. Campbell, R.H. and Habermann, A.N. The specification of process synchronizatin by path expresains. Lecture Notes in Computer Science 16, Springer-Verlag, 1974, 89-102.

6. Courtois, P.J., Heymans, F. and Parnas, D.L. Concurrent control with readers and writers. [illegible]. ACM 14, 10 (Oct. 1971), 667-668.

7. Dijkstra, E.W. Cooperating sequential processes. In Programming [illegible]uages, F. Genuys, Ed., Academic Press, New York, 1968, 43-112.

8. Dijkstra, E.W. Guarded commands, nondeterminacy, and formal derivation of programs. Comm. ACM 18, 8 (Aug. 1975). 453-457.

9. Dijkstra, E.W. et. al. On-the-fly garbage collection: An exercise in cooperation. Comm. ACM 21, 11 (Nov. 1978). 966-975.

10. Gries, D. An exercise in proving parallel programs correct. Comm. ACM 20, 12 (Dec. 1977), 921-930.

11. Habermann, A.N. Synchronization of communicating processes. Comm. ACM 15, 3 (Mar. 1972), 171-176.

12. Habermann, A.N. Path expressions. Technical Report, Carnegie-Mellon University, June 1975.

13. Hoare, C.A.R. Towards a theory of parallel programming. In Operating Systems Techniques, Academic Press, New York 1972, 61-71.

14. Hoare, C.A.R. Monitors: an operating system structuring concept. Comm. ACM 17, 10 (Oct. 1974), 549-557.

15. Hoare, C.A.R. Communicating sequential processes. Comm. ACM 21, 8 (Aug. 1978), 666-667.

16. Jammel, A.J. and Stiegler, H.G. Managers versus monitors. Information Processing 77, B. Gilchrist, Ed., North-Holland, 1977, 827-830.

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

18. McGraw, J.R. and Andrews, G.R. Access control in parallel programs. IEEE Trans. Software Eng. 5, 1 (Jan. 1979), 1-9.

19. Owicki, S. and Gries, D. An axiomatic proof technique for parallel programs. Acta Informatica 6 (1976), 319-340.

20. Reed, D.P. and Kanodia, R.K. Synchronization with eventcounts and sequencers. Comm. ACM, to appear.

21. Schneider, F.B. and Bernstein, A.J. Mechanisms for specifying scheduling policies. Tech. Report 79-365, Cornell University, January 1979.

22. Teorey, T.J. and Pinkerton, T.B. A comparative analysis of disk scheduling policies. Comm. ACM 15, 3 (Mar. 1972), 177-184.

23. Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.

24. Wirth, N. Modula: A programming language for modular multiprogramming. Software-Practice and Experience 7, 1 (Jan. 1977), 3-35.