Concurrent Programming Concepts

PER BRINCH HANSEN

Information Science, California Institute of Technology, Pasadena, California 91109

This paper describes the evolution of language features for multiprogramming from event queues and semaphores to critical regions and monitors. It suggests that the choice of language concepts should be guided by two simple principles: First, it should be possible to understand a concurrent program in time-independent terms by an effort proportional to its size; secondly, it should be possible to state assumptions about invariant relationships among program components and have these assumptions checked automatically. The central problems of multiprogramming are illustrated by annotated algorithms written in a well-structured programming language.

Key words and phrases: structured multiprogramming, programming languages, operating systems, programming errors, resource protection, compile-time checking, correctness proofs, sequential and concurrent processes, synchronizing events, semaphores, shared data, mutual exclusion, critical regions, monitors.

CR categories: 4.22, 4.31, 4.32, 5.24

> Somewhere there are decisions made that are not rational in any sense, that are subject to nothing more than the personal bias of the decision maker. Logical methods, at best, rearrange the way in which personal bias is to be introduced into a problem Of course, this "at best" is rather important. Present intuitive methods unhappily introduce personal bias in such a way that it makes problems impossible to solve correctly. Our purpose must be to repattern the bias, so that it no longer interferes in this destructive way with the process of design, and no longer inhibits clarity of form
>
> CHISTOPHER ALEXANDER, *Notes on the Synthesis of Form*

1. INTRODUCTION

This paper describes an attitude toward multiprogramming—the programming techniques used to control concurrent activities by computers.* It tries to identify abstract properties of programming languages that facilitate the design of large, reliable multiprogramming systems. The techniques used to implement these language concepts, which apply both to single-processor and multiprocessor systems, are described elsewhere. [1]

When a new programming technique, such as multiprogramming, is invented, programmers will initially use it in a completely unrestricted manner to discover its potential. At a later stage, when experience with its practical limitations has been gained, designers begin to recognize the benefits of a more restrictive language notation which clarifies their own understanding of programs and enables compilers to detect serious errors.

This survey describes the evolution of language features for multiprogramming during the past decade. I view this development mainly as a gradual shift from concepts that have a strong resemblance to assembly language features toward a notation that encourages hierarchical structuring of programs. Several useful proposals are not mentioned here, and those that are included

* In this paper, the term *multiprogramming* is used consistently to denote all programming techniques used to control concurrent processes on single-processor as well as multiprocessor systems

CONTENTS

are only meant to illustrate a general attitude toward the subject.

The central problems of multiprogramming are illustrated by algorithms written in a high-level language. I have included informal assertions in the algorithms and outlined arguments of their correctness to show that multiprogramming concepts can be defined as concisely as sequential programming concepts. In many ways, this survey can be regarded as an expanded version of an earlier paper [2]. We will begin with a brief characterization of sequential programs.

2. SEQUENTIAL PROCESSES

Although most programmers have an intuitive understanding of what a *sequential process* is, it may be useful to illustrate this concept by an example. Algorithm 1 defines a sequential process that inputs a fixed number of records *R1*, *R2*, ..., *Rn* from one sequential file, updates these records, and outputs them on another sequential file.

The language notation is borrowed from Pascal. [3] The files are declared as two variables, *reader* and *printer*, each consisting of a *sequence* of records of some type *T* (declared elsewhere). A record of type *T* can be appended to or removed from a sequence by means of two standard procedures, *put* and *get*. The current record is kept in a variable, *this*, of type *T*. The variable *i* counts the number of records processed; it can assume integer values from 0 to *n*.

Algorithm 1 Sequential file updating

```
    var reader, printer: sequence of T;
        this: T; i: 0 .. n;
    begin
(1) i := 0;
(2) while i < n do
(3)   get (this, reader);
(4)   update (this);
(5)   put (this, printer);
(6)   i := i + 1;
(7) end
(8) end
```

The program defines a sequential process in terms of *data* (represented by constants

and variables) and *operations* on data (represented by statements). During the execution of the program, the operations are carried out strictly one at a time, and each operation is completed within a finite time. The result of an operation can be defined by assertions about the relationships of data before and after its execution.

In Algorithm 1, we can make the following assertions about the state of the variables before and after the execution of the input statement in line 3:

comment *3 i out of n records processed (where* $0 \leqslant i < n$*);*
get (this, reader);
comment *4 i out of n records processed (where* $0 \leqslant i < n$*), and this = input record Ri* + 1;

Sequential programs have two vital properties:

1) *The effect of a sequential program is independent of its speed of execution.* All that matters is that operations are carried out one at a time with positive speed, and that certain relationships hold before and after their execution. The time-independent behavior of sequential programs enables the user to ignore details over which he has no control (such as the scheduling policy of the operating system and the precise timing of operations carried out by processors and peripherals).

2) *A sequential program delivers the same result each time it is executed with a given set of input data.* The reproducible behavior of sequential programs under any circumstances is particularly important for program validation. It enables one to isolate and correct program errors by systematic testing.

These properties of sequential programs are so well known that we tend to take them for granted But as we shall see, these properties can only be maintained for multiprogramming systems by a careful selection of language concepts.

Algorithm 1 is a *well-structured program* that can be analyzed in a step-wise manner as a set of *nested operations.* At the most detailed level, the program can be described in terms of assertions about each statement. The following assertions will hold *before* the execution of lines 1–8, respectively.

comment *1* 0 *out of n records processed (where* $0 \leqslant n$*);*
comment *2 i out of n records processed (where* $0 \leqslant i \leqslant n$*);*
comment *3 i out of n records processed (where* $0 \leqslant i < n$*);*
comment *4 i out of n records processed (where* $0 \leqslant i < n$*), and this = input record Ri* + 1;
comment *5 i out of n records processed (where* $0 \leqslant i < n$*), and this = updated record Ri* + 1;
comment *6 i* + 1 *out of n records processed (where* $0 \leqslant i < n$*);*
comment *7 i out of n records processed (where* $0 < i \leqslant n$*);*
comment *8 n out of n records processed (where* $0 \leqslant n$*);*

Once the individual statements are understood, one can define the effect of a sequence of statements by a single pair of assertions. As an example, the body of the *while* statement in Algorithm 1 can be defined as a single operation by means of the following assertions:

comment *3 i out of n records processed (where* $0 \leqslant i < n$*);*
process next record and increment i by one;
comment *7 i out of n records processed (where* $0 < i \leqslant n$*);*

The next step of simplification is to reduce the *while* statement to a single operation:

comment *2 i out of n records processed (where* $0 \leqslant i \leqslant n$*);*
process the remaining n-i records;
comment *8 n out of n records processed (where* $0 \leqslant n$*);*

Finally, the whole program can be defined as a single operation:

comment *1* 0 *out of n records processed (where* $0 \leqslant n$*);*
process n records;
comment *8 n out of n records processed (where* $0 \leqslant n$*);*

At this point, only *what* the program does as a whole, is relevant, but the details of *how* it is done are irrelevant. The possibility of replacing complex descriptions by partial

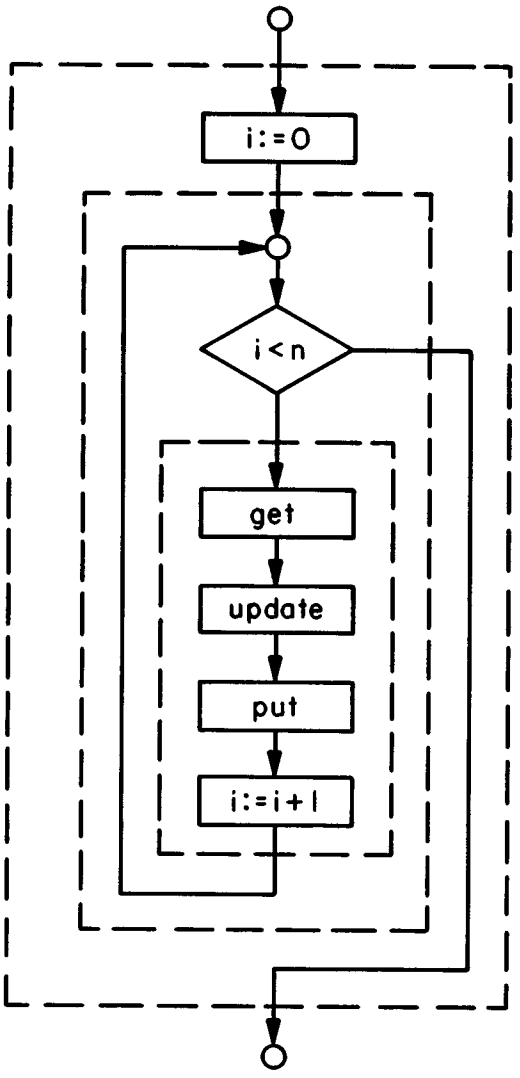

FIG. 1 Flowchart of Algorithm 1.

descriptions, called *abstractions*, is essential to enable us to understand large programs.

Figure 1 is a flowchart of Algorithm 1. It shows that the program consists of nested statements, each with a single starting point and a single completion point. Algorithm 1 can be analyzed by associating an assertion with each statement, and gradually replacing these assertions by fewer and fewer assertions. It is most important that *a program with a hierarchical structure can be built and understood in detail by an intellectual effort proportional to its size*, and that certain functional aspects of it can be described in even more economical terms. This is the whole purpose of structured programming!

It is also essential to use hierarchical structuring of programs that control concurrent processes, and choose language constructs for multiprogramming that can be understood in time-independent abstract terms; otherwise, we shall find ourselves unable to apply systematic methods of program analysis and verification.

3. UNRESTRICTED CONCURRENCY

Processes are called *concurrent* if their execution overlap in time. More precisely, two processes are concurrent if the *first* operation of one process starts before the *last* operation of the other process ends. The programming techniques used to control concurrent processes are called *multiprogramming*.

No assumptions will be made about the speed at which concurrent processes are executed (except that it be positive). We will benefit from this weak assumption in two ways: 1) It will encourage us to try to understand multiprogramming systems in time-independent terms; and 2) It will permit the underlying implementation to use a dynamic scheduling algorithm to achieve efficient resource sharing.

One of the earliest ideas of multiprogramming was to specify the start and completion of processes by two operations which I will call *start* and *complete* (In the literature, they are often called *fork* and *join*). The meaning of these operations can best be illustrated by an example:

```
var task: response;
begin
  start task do S1;
  S2;
  complete task;
  S3;
end
```

Initially, this program will be executed as a single, sequential process. The statement

start *task* **do** *S*1

will start the execution of statement *S*1 as a new process. As soon as the new process starts to run, the old process continues to execute the next statement *S*2. When statement *S*1 has been executed, the new process indicates its completion by means of a variable, *task*, of type *response*. Following this, the new process ceases to exist. After the execution of statement *S*2, the old process waits until the response variable indicates that statement *S*1 has terminated. The program then continues to execute statement *S*3 as a single, sequential process. (We will not be concerned with how the process and response concepts can be implemented.) Figure 2 shows a flowchart of this program.

Algorithm 2 is a concurrent version of the file updating program (Algorithm 1) that uses the *start* and *complete* statements. The purpose of introducing concurrent processes is to permit input/output operations to proceed simultaneously with updating operations. Algorithm 2 is deliberately written in an unstructured manner that makes it difficult to understand. A well-structured algorithm will be presented later. (I recommend that the reader not spend too much time trying to understand Algorithm 2 at this point.)

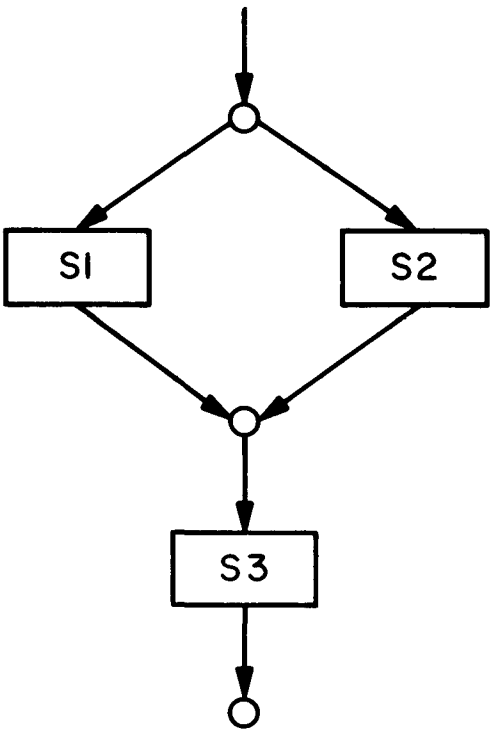

FIG 2 Flowchart of program to illustrate *start* and *complete* operations.

Algorithm 2 Concurrent file updating (unstructured version)

```
var reader, printer: sequence of T;
    reading, printing: response; next,
    this, last: T; i: 2 .. n;
begin
  if n ⩾ 1 then
    get (last, reader);
    if n ⩾ 2 then
      update (last);
      start printing do put (last,
        printer);
      get (this, reader);
      i:= 2;
      while i < n do
        start reading do get (next,
          reader);
        update (this);
        complete printing;
        last:= this;
        start printing do put (last,
          printer);
        complete reading;
        this:= next;
        i:= i + 1;
      end
      complete printing;
      last:= this;
    end
    update (last);
    put (last, printer);
  end
end
```

The program inputs and updates the first record *R*1. It then starts the output of the first record and inputs the second record *R*2 at the same time. The program now repeats the execution of a cycle that, in general, inputs the *next* record *Ri* + 1, updates *this* record *Ri*, and outputs the *last* record *Ri*-1 simultaneously. When all records have been input ($i = n$), the program completes the output of the second last record *Rn*-1. Finally, it updates and outputs the last record *Rn*.

In spite of this explanation, the details of Algorithm 2 are still hard to follow. The cause of this obscurity becomes clear when we examine the flowchart of the body of the *while* statement (Fig. 3). This flowchart cannot be decomposed into a set of nested operations. The program is highly confusing because a process started outside the *while* statement (*output*) may be completed either *inside* or *outside* the *while* statement (depending on whether or not $n > 2$).

It is certainly possible to write a simpler program using the *start* and *complete* statements (just as it is possible to structure programs written in assembly language). But the concepts of a high-level language should be chosen such that they naturally encourage the programmer to write well-structured programs. The *start* and *complete* statements clearly have the flavor of *go to* statements They make it very difficult to determine from the program text where concurrent processes begin and end, and which variables they operate on.

4. STRUCTURED CONCURRENCY

A well-structured representation of concurrent processes in a high-level programming language was suggested by Dijkstra in 1965. [4] The *concurrent statement*

cobegin $S1$; $S2$; . . . ; Sn **coend**

indicates that the statements $S1, S2, \ldots, Sn$ can be executed concurrently; when all of them are completed, the following statement in the program (not shown here) is executed.

Algorithm 3 defines file updating by means of concurrent statements. We will analyze this program, statement by statement, in terms of assertions that hold *before* the execution of lines 1–12

The initial assertion is the following:

comment *1* 0 *out of n records processed* (*where* $0 \leqslant n$);

If the input file is non-empty ($n \geqslant 1$), the program inputs the first record $R1$:

comment *2* 0 *out of n records processed* (*where* $1 \leqslant n$), *and this = input record* $R1$;

If the input contains more than one record ($n \geqslant 2$), the program inputs the second record $R2$ and updates the first record $R1$ simultaneously:

comment *3* 0 *out of n records processed* (*where* $2 \leqslant n$), *next = input record* $R2$, *and this = updated record* $R1$;

Algorithm 3 Concurrent file updating (structured version)

```
    var reader, printer: sequence of T;
        next, this, last: T; i: 2 . . n;
    begin
(1)   if n ⩾ 1 then
        get (this, reader);
(2)     if n ⩾ 2 then
          cobegin
            get (next, reader);
            update (this);
          coend
(3)       i:= 2;
(4)       while i < n do
            last := this; this := next;
(5)         cobegin
              get (next, reader);
              update (this);
              put (last, printer);
            coend
(6)         i:= i + 1;
(7)       end
(8)       put (this, printer);
          this := next;
(9)     end
(10)    update (this); put (this, printer);
(11)  end
(12) end
```

The program now enters a main loop that repeatedly inputs a record $Ri + 1$, updates a record Ri, and outputs a record $Ri - 1$ simultaneously. The loop is initialized by setting a counter i equal to 2. This leads to the following program state:

comment *4* $i - 2$ *out of n records processed* (*where* $2 \leqslant i \leqslant n$), *next = input record Ri, and this = updated record* $Ri - 1$;

Within the loop, the program goes through the following sequence of states:

comment *5* $i - 2$ *out of n records processed* (*where* $2 \leqslant i < n$), *this = input*

record Ri, and last = updated record Ri − 1;

comment *6 i − 1 out of n record processed (where 2 ⩽ i < n), next = input record Ri + 1, and this = updated record Ri;*

comment *7 i − 2 out of n records processed (where 2 < i ⩽ n), next = input record Ri, and this = updated record Ri − 1;*

A comparison of assertions 4 and 7 shows that assertion 4 is a relation that remains *invariant* after each execution of the loop. When the loop terminates, assertion 4 still holds and i is equal to n:

comment *8 n − 2 out of n records processed (where 2 ⩽ n), next = input record Rn, and this = updated record Rn − 1;*

The program now outputs the second last record $Rn - 1$:

comment *9 n − 1 out of n records processed (where 2 ⩽ n), and this = input record Rn;*

The effect of the innermost *if* statement can be summarized as follows:

comment *2 0 out of n records processed (where 1 ⩽ n), and this = input record R1;*

process n − 1 records;

comment *10 n − 1 out of n records processed (where 1 ⩽ n), and this = input record Rn;*

If $n = 1$ then assertions 2 and 10 are equivalent, and if $n \geqslant 2$ then assertions 9 and 10 are equivalent. This covers the two cases in which the *if* statement is either skipped or executed.

Finally, the last record Rn is updated and output:

comment *11 n out of n records processed (where 1 ⩽ n);*

The whole program can be understood as a single operation defined as follows:

comment *1 0 out of n records processed (where 0 ⩽ n);*

process n records;

comment *12 n out of n records processed (where 0 ⩽ n);*

This concurrent program has precisely the same characteristics as a well-structured sequential program. It can be analyzed in terms of time-independent assertions by an effort proportional to its size, and assertions about simple statements can gradually be replaced by assertions about structured statements until the whole program has been reduced to a single statement defined by two assertions. This abstract description of the program even enables the programmer to ignore the concurrent nature of the solution. To see this, it is sufficient to observe that the initial and final assertions 0 and 12 for Algorithm 3 are identical to the assertions made at the beginning and end of the sequential version (Algorithm 1).

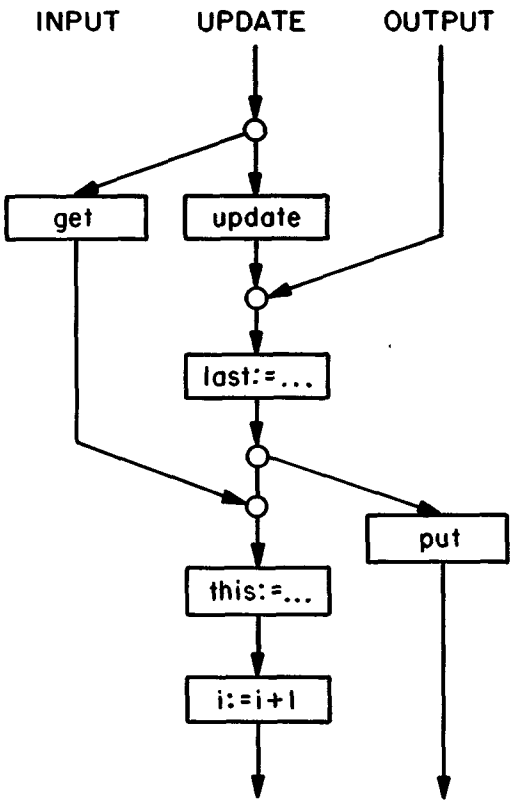

FIG. 3 Flowchart of the body of the *while* statement in Algorithm 2

In the previous discussion, the concurrent statement was defined and used informally to give you an intuitive understanding of its meaning. We will now explicitly discuss the assumptions that led to the simplicity of Algorithm 3.

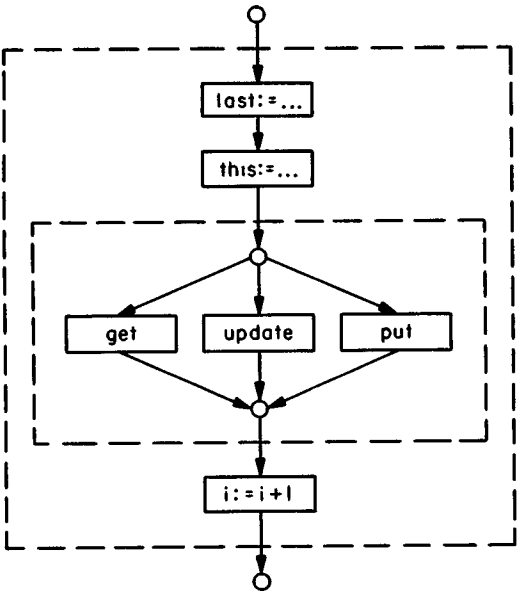

FIG. 4 A flowchart of the body of the *while* statement in Algorithm 3.

5. DISJOINT PROCESSES

There is one striking difference between Algorithms 2 and 3: *When concurrent statements are used, the program text directly shows where a given process begins and ends.* This simplification is achieved by restricting the freedom of scheduling. The *start* and *complete* statements enable the programmer to initiate and terminate processes in any order he pleases. A concurrent statement defines processes that are started and completed *at the same time.* Complete generality of programming implies complete absence of structure. As a compromise, one has to look for programming tools that are intellectually manageable and at the same time practical for most (but not necessarily all) applications. The concurrent statement is one such compromise.

Since a concurrent statement has a single starting point and a single completion point, it is well-suited to structured programming. Figure 4 shows a flowchart of the body of the *while* statement in Algorithm 3. In contrast to Fig. 3, the present flowchart can be decomposed into nested operations. This explains why we were able to reduce assertions about simple statements to assertions about structured statements, and then reduce assertions about structured statements to assertions about the whole algorithm.

It may seem surprising that the use of concurrent statements in Algorithm 3 in no way complicated the formulation of assertions. The informal arguments made in favor of its correctness are quite similar to the arguments on Algorithm 1 in Section 2. It turns out that we were able to apply sequential methods of program analysis to a concurrent program simply because *the processes defined by the concurrent statements are completely independent of one another.* These processes operate on disjoint sets of variables:

process	*variables*
get	next, reader
update	this
put	last, printer

They are called *disjoint* or *non-interacting processes.* The input, update, and output processes are carried out simultaneously only to utilize the computer more efficiently. But conceptually, these processes could just as well be carried out strictly sequentially as defined by Algorithm 1.

Since the processes in Algorithm 3 have no variables in common, they can be analyzed one at a time as unrelated sequential processes. Consider, for example, the concurrent statement inside the loop. It can be analyzed as three independent operations defined by the following assertions:

comment *5a* $i - 2$ *out of* n *records processed* (*where* $2 \leqslant i < n$);
get (*next, reader*);
comment *6a next = input record* $Ri + 1$ (*where* $2 \leqslant i < n$);

comment *5b* $i - 2$ *out of* n *records processed* (*where* $2 \leqslant i < n$), *and this = input record* Ri;
update (*this*);
comment *6b this = updated record* Ri (*where* $2 \leqslant i < n$);

comment *5c* $i - 2$ *out of* n *records processed* (*where* $2 \leqslant i < n$), *and last = updated record* $Ri - 1$;
put (*last printer*);

comment *6c* $i - 1$ *out of n records processed (where* $2 \leqslant i < n$*)*;

Here assertions 5 and 6 have been split into three pairs of assertions, one for each process statement. Each pair of assertions defines our assumptions about a variable i that remains *constant* during the execution of the concurrent statement and our assumptions about the variables that are *private* to the corresponding process and may be changed by it.

Since the processes are disjoint, the conjunction of assertions 6a, 6b, and 6c holds after the execution of the concurrent statement.

In general, we can define the properties of a concurrent statement by the following *rule of disjointness:* The statement

cobegin *S*1; *S*2; ... *Sn* **coend**

defines statements *S*1, S2, ... , *Sn* that can be executed concurrently as disjoint processes. *The disjointness implies that a variable vi changed by a statement Si cannot be referenced by another statement Sj* (where $j \neq$ i). In other words, a variable subject to change by a process must be strictly private to that process; but disjoint processes can refer to common variables not changed by any of them.

The general rule that we intuitively followed in analyzing Algorithm 3 can be stated as follows: Suppose that we know the following about statements *S*1, *S*2, ... , *Sn* that operate on disjoint variables:

Statement *S*1 will terminate with a result *R*1 if a precondition *P*1 holds before its execution.

Statement *S*2 will terminate with a results *R*2 if a precondition *P*2 holds before its execution.

.....

Statement *Sn* will terminate with a result *Rn* if a precondition *Pn* holds before its execution.

Then we can conclude that a concurrent execution of *S*1, *S*2, ... , *Sn* will terminate with the result *R*1 & *R*2 ... & *Rn* if the precondition *P*1 & *P*2 & ... & *Pn* holds before its execution. (It should be added that the assertions *Pi* and *Ri* made about statement *Si* must only refer to variables that are accessible to *Si* according to the rule of disjointness.)

6. TIME-DEPENDENT ERRORS

An error in a sequential program can be located by repeating the execution of the program several times with the data that revealed the error. In each of these experiments, the values of selected variables are recorded to determine whether or not a given program component works. This process of elimination continues until the error has been located.

When a given program component has been found to behave correctly in one test, we can ignore that component in subsequent tests because it will continue to behave in exactly the same manner each time the program is executed with the given data. In other words, *our ability to test a large sequential program in a step-wise manner depends fundamentally on the reproducible behavior of the program.*

A careful programmer who writes a well-structured concurrent program, such as Algorithm 3, and outlines an informal proof of its correctness can still make mistakes when he types the final program. And he may not find all these errors during a proofreading of the program text. One possible mistake would be to type the concurrent statement within the loop of Algorithm 3 as follows:

cobegin
get (next, reader);
update (this);
put (this, printer);
coend

In this case, the concurrent statement will input the next record *Ri* + 1 correctly, but will update and output the current record *Ri* simultaneously. So the output record will normally be only partially updated. In a multiprocessor system with a common store, a record occupying, say 256 machine words, may be output with x words updated and 256-x words unchanged (where $0 \leqslant x \leqslant 256$). The processing of a single record can therefore produce 257 different results. If

we update a given file of 10,000 records, the program can give of the order of $257^{10,000}$ different results.

In a multiprogramming system, the execution of concurrent statements can be interleaved and overlapped in arbitrary order. The system uses this freedom to multiplex fast computer resources (processors and stores) among several user computations to achieve short response times at a reasonable cost.

The result of the erroneous version of Algorithm 3 depends on the relative rates at which updating and output of the same record take place. The rates will be influenced by the presence of other (unrelated) computations, and by the absolute speeds of peripheral devices and operators interacting with the computations. It is therefore very unlikely that the erroneous program will ever deliver the same result twice for a given input file. The error will be particularly hard to find if the updated file is not inspected by humans, but just retained for later processing by other programs.

Such unpredictable program behavior makes it impossible to locate an error by systematic testing. It can only be found by studying the program text in detail. This can be very frustrating (if not impossible) when it consists of thousands of lines and one has no clues about where to look. *If we wish to succeed in designing large, reliable multiprogramming systems, we must use programming tools that are so well-structured that most time-dependent errors can be caught at compile time.*

A closer look at the incorrect version of Algorithm 3 reveals a clear violation of the rule of disjointness: Within the erroneous concurrent statement the output process refers to a variable, *this,* which is changed by the updating process. It is therefore inevitable that the result of the output depends on the *time* at which updating takes place.

To make it simple for a compiler to check the disjointness it should be possible by scanning the program text to recognize concurrent statements and variables accessed by them. The compiler must be able to distinguish between variables that can be changed by a statement and variables that can be referenced by a statement but not changed by it. These two kinds of variables are called the *variable parameters* and *constant parameters* of a statement.

When *start* and *complete* statements are used, one cannot, in general, recognize concurrent statements from the syntax alone. This recognition is, however, trivial when concurrent statements are enclosed in brackets, *cobegin* and *coend.* To make the checking of disjointness manageable, it is necessary to restrict the use of pointer variables and procedure parameters far more than present programming languages do.

Although the problem has not yet been analyzed completely, it seems certain that the necessity of compile-time checking of disjointness will have a profound influence on language design. An example will make this clear. In sequential programming languages a pointer variable may be bound to other variables of a given type, for example:

var *p:* **pointer to** *integer;*

This declaration indicates that variable p is a pointer to *any integer* variable. The notation enables a compiler and its run-time system to check that p always points to a variable of a well-defined type (in this case, an integer) This kind of pointer variable is far better than one that can point to an arbitrary store location containing data of unknown type (or even code). But the declaration *pointer to integer* still does not enable a compiler to recognize *which integer* the variable p will point to during program execution So, unless the programmer is willing to make *all* integers private to a single process, this pointer concept is inadequate for multiprogramming. We need to bind a pointer p to a particular set of integers, for example:

var *i, j, k: integer; p:* **pointer to** *i* **or** *j;*

Some of the language requirements needed to make compile-time checking of disjointness practical are discussed in more detail in [1].

The concurrent statement has not yet been implemented and used for practical

programming, but I would expect it to be a convenient tool for the design of small, dedicated multiprogramming systems involving a predictable number of processes. I have some reservations about its usefulness in larger multiprogramming systems (for example, operating systems) as discussed later in this paper.

The rule of disjointness is introduced to help the programmer; it enables him to state explicitly that certain processes should be independent of one another and to depend on automatic detection of violations of this assumption. To make multiprogramming intellectually manageable and reasonably efficient, disjoint processes should be used wherever possible. But, as we shall see, all multiprogramming systems must occasionally permit concurrent processes to exchange data in a well-defined manner. The components of a concurrent statement must, for example, be able to indicate their termination in a common (anonymous) variable; otherwise, it would be impossible to determine when a concurrent statement is terminated as a whole. The *cobegin coend* notation hides this communication problem from the user, but it has to be solved at some other level of programming (in this case by the code generated by a compiler). The following sections describe language features used to control interactions among processes

7. TIMING SIGNALS

Concurrent processes that access common variables are called *interacting* or *communicating processes*. When processes *compete* for the use of shared resources, common variables are necessary to keep track of the requests for service. And when processes *cooperate* on common tasks, common variables are necessary to enable processes to ask one another to carry out subtasks and report on their results.

We will study systems in which one process *produces* and *sends* a sequence of data items to another process that *receives* and *consumes* them. It is an obvious constraint that these *data items cannot be received faster than they are sent.* To satisfy this requirement it is sometimes necessary to delay further execution of the receiving process until the sending process produces another data item. *Synchronization* is a general term for timing constraints of this type imposed on interactions between concurrent processes.

The simplest form of interaction is an exchange of *timing signals* between two processes. A well-known example is the use of *interrupts* to signal the completion of asynchronous peripheral operations to a central processor. Another kind of timing signals, called *events*, was used in early multiprogramming systems to synchronize concurrent processes. When a process decides to *wait* for an event, the execution of its next operation is delayed until another process *causes* the event. An event occurring at a time when no processes are waiting for one has no effect.

The following program illustrates the transmission of timing signals from one process to another by means of a variable *e* of type *event*. Both processes are assumed to be cyclical:

```
var e: event;
cobegin
  cycle "sender"
    ... cause event(e); ...
  end
  cycle "receiver"
    ... await event(e); ...
  end
coend
```

A relationship of this type exists in a real-time system in which one process schedules concurrent tasks regularly by sending timing signals to other processes that carry out these tasks.

In Section 6 we recognized that simultaneous operations on the same variable can lead to a large number of different results. In that context, the problem was caused by a program error. Now we have the same problem again: The concurrent operations, *await* and *cause*, both access the same variable *e*. But we can no longer regard this as a violation of the rule of disjointness since our in-

tention is that the processes should exchange data. So *we must relax the rule of disjointness and permit concurrent processes to access shared variables by means of well-defined synchronizing operations.* Our next task is to determine under which conditions synchronizing operations are "well-defined".

To analyze the effect of an interaction between the receiver and the sender in the previous program, we must consider all the possible ways in which the execution of *await* and *cause* operations can be interleaved and overlapped in time. In a continuous time scale there are infinitely many possibilities to consider. A drastic simplification is clearly needed to reduce this infinity to a finite (small) number of cases. The only practical solution is to assume that *synchronizing operations on a given variable cannot be executed at the same time.* In other words, *await* and *cause* operations on a given event variable can be arbitrarily interleaved (but not overlapped) in time. If a process tries to operate on an event variable while another process is operating on it, the system must delay the former process until the latter process has completed its operation on the event variable.

If this requirement is satisfied, there are only two cases to consider in the previous program: either an *await* operation is executed *before* a *cause* operation or *after* it. If the receiver waits before the sender causes the next event, an interaction between the two processes is defined by the following sequence of operations:

comment *receiver not waiting;*
await event(*e*);
comment *receiver waiting;*
cause event (*e*);
comment *receiver not waiting;*

But, if the sender causes the event before the receiver waits for it, the receiver will remain delayed until the next event is caused:

comment *receiver not waiting;*
cause event(*e*);
comment *receiver not waiting;*
await event (*e*);
comment *receiver waiting;*

The most important result of this analysis is the general observation that *mutual exclusion of all operations on a shared variable enables the programmer to analyze the possible effects of a process interaction in finite, sequential terms.* For the special case of event variables, we have also discovered that the net effect of *await* and *cause* operations depends on the order in which these operations are carried out. Or, to put it more strongly: *event operations force the programmer to be aware of the relative speeds of the sending and receiving processes.*

The programmer does not control the order in which concurrent statements are executed; he is therefore unable to predict the effect of a process interaction involving events. In a real-time system, the programmer has essentially lost control of process scheduling—he cannot define a process that will schedule other processes in a predictable manner. Event variables are only meaningful to use when one can assume that a process never is asked to carry out another task until it has completed its previous task. Although this assumption may be satisfied in some applications, it will still complicate programming tremendously if one depends on it. The programmer must then be aware of the relative speeds of processes under all circumstances. If a multiprogramming system is so large that no single person understands its dynamic behavior in detail, the individual programmer cannot make reliable estimates of the relative speeds of processes under all circumstances. In particular, it will be an intolerable burden to verify that the speed assumptions are uninfluenced by modifications or extensions of a large system.

We must therefore conclude that event variables of the previous type are impractical for system design. *The effect of an interaction between two processes must be independent of the speed at which it is carried out.*

A far more attractive synchronizing tool, the *semaphore,* was invented by Dijkstra in 1965. [4, 5] A semaphore is a variable used to exchange timing signals among concurrent processes by means of two operations, *wait* and *signal* (originally called *P* and *V*). All

operations on a semaphore exclude one another in time. Associated with a semaphore v are two integers defining the number of signals *sent* and *received* through v and a queue in which receiving processes can await the sending of further timing signals by other processes. Initially, the number of signals sent and received are zero and the queue is empty.

Signals cannot be received faster than they are sent. This *semaphore invariant:*

$$0 \leqslant received \leqslant sent$$

is satisfied by using the following synchonization rules:

1) If a *wait* operation on a semaphore v is executed at a time when $received < sent$ then *received* is increased by one and the receiving process continues; but if $received = sent$, the receiver is delayed in the queue associated with v.

2) A *signal* operation on a semaphore v increases *sent* by one; if one or more processes are waiting in the queue associated with v, one of these processes is enabled to continue its execution and *received* is increased by one.

We assume that all processes waiting to receive signals eventually will be able to continue their execution (provided a sufficient number of signals are sent by other processes). The scheduling algorithm used for a semaphore queue must not delay any process indefinitely in favor of more urgent processes. But, apart from this requirement of *fair scheduling,* no assumptions are made about the specific order in which waiting processes are allowed to continue. The weak assumption of *finite progress* (rather than absolute speed) for any process is a recurrent theme of programming. We have made this assumption for sequential and disjoint processes, and now we make it again for interacting processes to achieve simplicity of program analysis and flexibility of implementation.

Algorithm 4 defines a transmission of timing signals from one process to another by means of a semaphore v.

Algorithm 4 Exchange of timing signals by means of a semaphore

```
var v: semaphore;
cobegin
  cycle "sender"
    ... signal (v); ...
  end
  cycle "receiver"
    ... wait (v); ...
  end
coend
```

Since *wait* and *signal* operations on the semaphore v exclude each other in time, a signal can be sent either *before* or *after* the receiver decides to wait for it. In the first case, we can make the following assertions about the sequence in which an interaction takes place:

```
comment receiver not waiting and 0 ⩽
  received ⩽ sent;
signal (v);
comment receiver not waiting and 0 ⩽
  received < sent;
wait (v);
comment receiver not waiting and 0 <
  received ⩽ sent;
```

In the second case, the *wait* operation may or may not delay the receiver (depending on whether $received = sent$ or $received < sent$). But, in any case, the subsequent *signal* operation will ensure that the receiver continues its execution:

```
comment receiver not waiting and 0 ⩽
  received ⩽ sent;
  wait (v);
comment receiver waiting and 0 ⩽ re-
  ceived = sent, or receiver not waiting
  and 0 < received ⩽ sent;
  signal (v);
comment receiver not waiting and 0 <
  received ⩽ sent;
```

The effect of an interaction is independent of the order in which the *wait* and *signal* operations are carried out. *The commutative property of semaphore operations enables the programmer to ignore the precise moment at which a timing signal is produced.* This is certainly the most important contribution of semaphores to program clarity.

Other uses of semaphores will be described later when we have clarified the fundamental role of mutual exclusion in multiprogramming.

8. CRITICAL REGIONS

We will now consider concurrent processes that exchange data of arbitrary type (and not just timing signals). As an example we choose a multiprogramming system in which job statistics are collected by a process P and printed by another process Q. When a user job has been completed in this system, process P increases an integer v by one. At regular intervals process Q prints the value of v and resets it to zero. To discover the problems of data sharing we will begin with a naive "solution":

```
var v: integer;
begin
  v:= 0;
  cobegin
    cycle "P"
      ...v:= v + 1;...
    end
    cycle "Q"
      ...print (v); v:= 0;...
    end
  coend
end
```

This program violates the rule of disjointness since processes P and Q both refer to and change the same variable v. Although this violation suggests a weakness of the present approach to the problem we will ignore it for the time being.

The state of the system accumulating job statistics can be defined by two integers (initially equal to zero):

x the number of jobs executed
r the number of jobs reported

The variable v should represent the number of jobs executed but not yet reported; that is, the relationship $v = x - r$ should remain *invariant* after each sequence of operations on the shared variable v.

The example illustrates two general characteristics of multiprogramming systems:

1) *The correctness criterion for concurrent operations on a shared variable is defined by an invariant*—a relationship that must be true after initialization of the variable and continue to hold before and after subsequent operations on the variable.

2) *The invariant property of a shared variable is defined in terms of actual and implicit variables.* An *actual* variable (such as v) is declared in the program and represented by a store location during its execution. An *implicit* variable (such as x or r) refers to a property of the system that is not represented in the computer during program execution.

The statements executed by processes P and Q can be arbitrarily overlapped and interleaved in time. We will, however, only analyze the effects of an arbitrary interleaving of the concurrent statements that operate on the shared variable v. The increase of v by process P can occur either *before, after,* or in the *middle* of the printing and resetting of v by process Q. So we have three cases to consider:

$v.= v + 1;$	*print*(v);	*print*(v);
print(v);	$v:=0;$	$v:= v + 1;$
$v.=0;$	$v:=v + 1;$	$v:= 0,$

The correctness criterion of the possible interactions between processes P and Q is defined by an invariant $(v = x - r)$ that relates an actual variable v to two implicit variables x and r. To analyze the results of the three possible cases, we must extend the program with implicit statements referring to x and r (even though these statements will never be executed). Conceptually, x is increased by one when v is increased, and r is increased by the value of v when the latter is printed. Assuming that the desired invariant holds before an interaction takes place, we can make the following assertions about the first case:

```
comment x - r = v;
v:= v + 1; [x:= x + 1]
comment x - r = v;
print (v); [r:= r + v]
comment x - r = 0;
v:= 0;
comment x - r = v;
```

The implicit statements are enclosed in square brackets to distinguish them from actual statements. In this case, the invariant still holds after the process interaction.

In the second case, the invariant is also maintained by the interaction:

comment $x - r = v$;
print (v); $[r := r + v]$
comment $x - r = 0$;
$v := 0$;
comment $x - r = v$;
$v := v + 1$; $[x := x + 1]$
comment $x - r = v$;

But in the third case, the "invariant" no longer holds after the process interaction:

comment $x - r = v$;
print (v); $[r := r + v]$
comment $x - r = 0$;
$v := v + 1$; $[x := x + 1]$
comment $x - r = 1$;
$v := 0$;
comment $x - r = v + 1$;

Whether or not the invariant is maintained depends on the ordering of the concurrent statements in time. This is, of course, unacceptable to the programmer who has no control over the scheduling of concurrent processes.

Notice that in the first two cases, in which the invariant continues to be satisfied, the two processes have exclusive access to the variable v while they are operating on it:

P	$v := v + 1;$		Q	*print* (v); $v := 0;$	
Q	*print* (v); $v := 0;$		P	$v := v + 1;$	

The conflict in the third case is caused by mixing the sequence of operations of process Q on variable v with an operation of process P on the same variable:

Q	*print* (v);
P	$v := v + 1;$
Q	$v := 0;$

Process Q performs a sequence of operations on the shared variable v. It is assumed that the invariant holds before and after this sequence of operations. But while the operations are carried out, the variable v may be in various intermediate states in which the invariant is not satisfied. If other concurrent statements are carried out on the same variable in one of its intermediate states, the result will normally be incorrect. In the previous example, process P starts to operate on variable v at a time when $x - r = 0$ (instead of $x - r = v$); this leads to a final state in which $x - r = v + 1$ (instead of $x - r = v$).

In Section 7 we found that the effect of synchronizing operations on event variables and semaphores can be predicted only if they exclude one another in time. The present example shows that one must also be able to achieve mutual exclusion of arbitrary concurrent statements referring to a shared variable of an arbitrary type.

Concurrent statements that refer to the same shared variable are called *critical regions*. The previous program contains two critical regions:

$$v := v + 1$$

and

$$\begin{array}{l} \mathit{print}(v) \\ v := 0 \end{array}$$

Critical regions referring to the same variable exclude one another in time. They can, however, be arbitrarily interleaved in time. We make three assumptions about critical regions that operate on the same shared variable:

Mutual exclusion: At most, one process at a time can be inside a critical region.
Termination: A process will always complete a critical region within a finite time.
Fair scheduling: A process can always enter a critical region within a finite time.

Dekker has shown that mutual exclusion of arbitrary statements can be implemented by means of *load* and *store* operations only (provided that load and store operations on a given variable exclude one another in time). His solution is far too complicated and inefficient to be of practical value [4], but it does illustrate the subtlety of the mutual exclusion problem.

The inadequacy of the load-and-store approach to mutual exclusion inspired Dijkstra and Scholten to invent semaphores. Algorithm 5 defines an implementation of critical regions by means of a semaphore *mutex*.

Algorithm 5 Critical regions implemented by means of a semaphore

```
var v: integer; mutex: semaphore(1);
begin
  v:= 0;
  cobegin
    cycle "P"
      ... wait (mutex); v:= v + 1;
          signal (mutex); ...
    end
    cycle "Q"
      ... wait (mutex); print (v);
          v:= 0; signal (mutex); ...
    end
  coend
end
```

The declaration of a semaphore *mutex* has been extended with an integer constant defining the *initial* number of available signals:

var *mutex: semaphore (initial)*

A semaphore is now characterized by three integer components:

initial the number of signals initially available
sent the number of signal operations completed
received the number of wait operations completed

The *semaphore* invariant must therefore be revised slightly:

$$0 \leqslant received \leqslant sent + initial$$

For the semaphores used in Section 7, *initial* is zero. In Algorithm 5, *initial* is equal to one.

To make a sequence of statements $S1, S2, \ldots, Sn$ a critical region, we enclose it by a pair of *wait* and *signal* operations on a semaphore *mutex* initialized to one:

```
var mutex: semaphore(1);
... wait (mutex); S1; S2; ... Sn;
        signal (mutex); ...
```

The initial signal allows precisely one process to enter its critical region. Once a process has consumed the available signal and entered its critical region, no other process can enter a critical region associated with the same semaphore until the former process leaves its critical region and produces another signal.

A more rigorous proof of the mutual exclusion depends on the following observations:

1) Independent of how *wait* and *signal* operations are used they maintain the *semaphore invariant:*

$$0 \leqslant received \leqslant sent + 1$$

2) When a semaphore is used to achieve mutual exclusion, a process always executes a *wait* operation followed by a *signal* operation. At any given time, some processes may have executed a *wait* operation, but not yet the corresponding *signal* operation. So the structure of the program shows that the invariant:

$$0 \leqslant sent \leqslant received$$

is also maintained.

3) Finally, it is clear that the number of processes that are *inside* their critical regions at any given time are those processes that have completed a *wait* operation but not yet the following *signal* operation, that is:

$$inside = received - sent$$

By combining these three invariants, we find that the first assumption about critical regions is satisfied:

$$0 \leqslant inside \leqslant 1$$

At most, one process at a time can be inside a critical region.

Assuming that processes are scheduled fairly when they are inside their critical regions, we can also conclude that the statements executed within the critical regions of Algorithm 5 will terminate within a finite time. And if the scheduling of processes waiting for timing signals in the semaphore queue is also fair, then a process can only be delayed a finite number of times while other processes enter critical regions ahead of it; so a process will always be able eventually to enter its critical region. The implementation of critical regions in Algorithm 5 is therefore correct. Notice that the analysis of this concurrent program is stated in terms of the implicit variables, *received, sent,* and *inside.*

A semaphore is an elegant synchronizing tool for an ideal programmer who never makes mistakes. But unfortunately the consequences of using semaphores incorrectly can be quite serious. A programmer might by mistake write the *wait* and *signal* operations in reverse order for a particular critical region:

signal(*mutex*); ... *wait*(*mutex*);

In that case, the system will sometimes permit three processes to be inside their "critical regions" at the same time. This is a time-dependent error that only reveals itself if other processes enter critical regions while the erroneous critical region is being executed.

Another serious error would be the following:

wait(*mutex*); ... *wait*(*mutex*);

This one causes the process executing the critical region to wait forever at the end of it for a timing signal that will never be produced. Since the incorrect process is unable to leave its critical region, other processes trying to enter their critical regions will also be delayed forever. Notice, that the behavior of the other processes is only influenced by the erroneous process after the latter has entered its incorrect region. So the error is clearly time-dependent. Such a situation in which two or more processes are waiting indefinitely for synchronizing conditions that will never be satisfied is called a *deadlock*.

These examples show how easy it is to cause a time-dependent error by means of a semaphore. Even if a semaphore is used correctly it still does not provide us with a satisfactory notation to indicate that the violation of the rule of disjointness with respect to the variable v in Algorithm 5 is deliberate.

A semaphore is a general programming tool that can be used to solve arbitrary synchronizing problems Hence a compiler cannot always assume that a pair of *wait* and *signal* operations on a semaphore initialized to one delimits a critical region. In particular, a compiler cannot recognize the following errors: if a pair of *wait* and *signal* operations are exchanged, if one or both of them are missing, or if a semaphore is initialized incorrectly. A compiler is also unaware of the correspondence between a shared variable v and the semaphore *mutex* used to gain exclusive access to v. Consequently, the compiler cannot protest if a critical region implemented by a semaphore *mutex* by mistake refers to another shared variable w (instead of v). Indeed, *a compiler cannot give the programmer any assistance whatsoever in establishing critical regions correctly by means of semaphores.*

Since semaphores alone do not enable a programmer to indicate whether a variable v should be *private* to a single process or *shared* by several processes, a compiler must either forbid or permit any process interaction involving that variable. To forbid process interaction is unrealistic (since it prevents us from building interactive multiprogramming systems consisting of cooperating processes); to permit arbitrary process interaction is disastrous (because of the danger of irreproducible programming errors). We must therefore conclude that semaphores do not enable a compiler to give a programmer the effective assistance in error detection that he should expect from an implementation of a high-level language.

To improve this situation, I have suggested a structured notation for shared variables and critical regions [2]. A *shared variable* v of type T is declared as follows:

var v: **shared** T

Concurrent processes can only refer to and change a shared variable v inside a structured statement of the form:

region v **do** $S1$; $S2$ ... Sn **end**

This notation indicates that the sequence of statements $S1, S2, \ldots, Sn$ should have exclusive access to the shared variable v. By explicitly associating a critical region with the shared variable on which it operates the programmer tells the compiler that the sharing of this variable among concurrent processes is a deliberate exception to the rule of disjointness; at the same time, the compiler

can check that a shared variable is used only inside critical regions and can generate code that implements mutual exclusion correctly. It is perfectly reasonably to use semaphores in the underlying implementation of this language feature, but at higher levels of programming the explicit use of semaphores to achieve mutual exclusion is debatable. A similar notation for critical regions was developed independently by Hoare [6].

Algorithm 6 shows the use of a shared integer v and two critical regions to solve the previous problem (See also Algorithm 5).

Algorithm 6 Critical regions represented by a structured notation

```
var v: shared integer;
begin
  v:= 0;
  cobegin
    cycle "P"
      ... region v do v:= v + 1 end ...
    end
    cycle "Q"
      ... region v do print(v); v:= 0
          end ...
    end
  coend
end
```

It has been our persistent goal to look for multiprogramming features that can be understood in time-independent terms. Since the precise ordering of critical regions in time is unknown, a time-independent assertion of their net effect can only be an assertion about a propertly of the associated shared variable that remains constant—in short, an *invariant I* that must be true after initialization of the variable and before and after each critical region operating it.

A relationship I that remains true at all times must in some way reflect the entire history of the processes referring to the shared variable. So we find that the invariant for Algorithms 5 and 6 is expressed in terms of the total number of jobs executed and reported throughout the existence of the multiprogramming system. However, since the range of such implicit variables is unbounded, they cannot be represented in a computer with a finite word length. On the other hand, actual variables being bound to a finite range can only represent the most recent past of a system's history. We must therefore find a function $f(x, r)$ of the implicit variables x and r with a finite range that can be represented by an actual variable v. In Algorithms 5 and 6 the function is

$$f(x, r) = x - r = v$$

For a semaphore, the invariant is a relationship among the implicit variables representing all signals *sent* and *received* throughout the lifetime of a semaphore:

$$0 \leqslant received \leqslant sent + initial$$

In a computer, these implicit variables can be represented by a single integer (equal to *sent* + *initial* − *received*) that must remain non-negative.

People with a strong interest in correctness proofs may well find it helpful to declare implicit variables and implicit statements referring to them explicity in the program text Implicit quantities have no effect on the execution of a program; their sole function is to facilitate program verification by making assumptions about the system's history explicit.

Implicit variables and statements should be subject to the following restrictions:

1) Assertions about a program may refer to actual as well as implicit variables.

2) Expressions involving actual and implicit variables may formally be "evaluated" and "assigned" to implicit variables.

3) Expressions evaluated and assigned to actual variables may only refer to actual variables.

Critical regions referring to different shared variables can be executed simultaneously. The use of *nested critical regions* can, however, lead to a *deadlock* unless precautions are taken. Consider, for example, the following program with two shared variables v and w of types T and T':

```
var v: shared T; w: shared T';
cobegin
  "P" region v do region w do ... end
      end
```

"Q" **region** w **do region** v **do** ... **end**
end
coend

Process P can enter its region v at the same time that process Q enters its region w. When process P tries to enter its region w, it will be delayed because Q is already inside its region w. And process Q will be delayed trying to enter its region v because P is already inside its region v.

The deadlock occurs because the two processes enter their critical regions in opposite order and create a situation in which each process is waiting indefinitely for the completion of a region within the other process. It can be proved that a deadlock cannot occur if all processes enter nested regions in the same (hierarchical) order [1] A compiler might prevent such deadlocks simply by checking that nested critical regions refer to shared variables in the (linear) order in which these variables are declared in the program.

It is an amusing paradox of critical regions that to implement one, we must appeal to the existence of simpler critical regions (called *wait* and *signal*). The implementation of *wait* and *signal* operations, in turn, requires the use of an *arbiter*—a hardware implementation of still simpler critical regions that guarantee exclusive access to a semaphore by a single processor in a multiprocessor system. This use of nested critical regions continues at all levels of machine design until we reach the atomic level, at which nuclear states are known to be discrete and mutually exclusive.

The main conclusion of this section must be that *it is impossible to make useful assertions about the effect of concurrent statements unless operations on shared variables exclude one another in time. Mutual exclusion* is necessary to reduce a virtual infinity of possible time-dependent results to a small number of possibilities. The use of *invariant relationships* simplifies the program analysis further. Together, these mental tools enable us to study concurrent programs in time-independent terms by an effort proportional to the number of critical regions used. So in the end, our understanding of concurrent processes is based on our ability to execute their interactions strictly sequentially; *Only disjoint processes can proceed truly simultaneously.*

9. CONDITIONAL CRITICAL REGIONS

We will now consider multiprogramming systems in which processes can wait until certain conditions are satisfied by other processes. The classic example is the exchange of *messages* between two processes by means of a *buffer* of *finite capacity* as defined by Algorithm 7. Here the sender must be able to wait while the buffer is full, and the receiver must able to wait while the buffer is empty.

The message buffer v is declared as a shared record consisting of two components: a sequence s of messages of some type T, and an integer *full* defining the number of messages currently stored in the sequence.

Initially, the buffer is empty ($full = 0$). Since the buffer has a finite *capacity*, the operations used to *send* and *receive* a message must maintain the following *buffer invariant:*

$$0 \leqslant full \leqslant capacity$$

Algorithm 7 Message buffer

```
var v: shared record
          s: sequence of T;
          full: integer;
       end
  m, n: T;
comment send message m;
region v when full < capacity do
  put(m, s);
  full:= full + 1;
end

comment receive message n;
region v when full > 0 do
  get(n, s);
  full:= full − 1;
end
```

To indicate that the sending of a message must be postponed until $full < capacity$, we will use the *conditional critical region* proposed by Hoare [6]:

region *v* **when** *full* < *capacity* **do** . . . **end**

When the sender enters this conditional critical region, the Boolean expression *full* < *capacity* is evaluated. If the expression is true, the sender completes the execution of the critical region by putting a message *m* of type *T* into the sequence *s* and increasing *full* by one. But if the expression is false, the sender leaves the critical region temporarily and enters an anonymous queue associated with the shared variable *v*. The sender will be allowed to reenter and complete the critical region as soon as the receiver has removed a message from the buffer (thus making *full* < *capacity*).

Another conditional critical region is used to postpone the receiving of a message until *full* > 0. If *full* = 0, the receiver leaves the critical region temporarily and joins the anonymous queue. In that case, the critical region will be continued when the sender has put another message into the buffer and made *full* > 0. At this point, the receiver will take a message *n* of type *T* from the sequence *s* and decrease *full* by one.

In general, a conditional critical region

region *v* **when** *B* **do** *S*1; *S*2; . . . ; *S*n **end**

is used to delay the completion of a critical region until a shared variable *v* satisfies a a specific condition *B* (in addition to an invariant *I*).

When a process enters a conditional critical region, a Boolean expression *B* is evaluated. If *B* is true, the critical region is completed by executing the statements *S*1, *S*2, . . . , *Sn;* otherwise, the process leaves its critical region temporarily and enters a queue associated with the shared variable *v*.

All processes waiting for one condition or another on the variable *v* enter the same queue. When a process completes a critical region on *v*, the synchronizing conditions of the waiting processes are reevaluated. If one of these conditions is satisfied, the corresponding process is allowed to reenter and complete its critical region.

The scheduling of waiting processes must be fair in the following sense: If a process is waiting for a condition *B* that is repeatedly made true by one or more "producers" and false by one or more "consumers," the completion of the given critical region can only be delayed a finite number of times by other critical regions.

We will use conditional critical regions to solve the following problem [8]: A stream of data elements of type *T* produced by a process *P*0 passes through a sequence of processes *P*1, *P*2, . . . , *Pn* that operate on the data elements in that order:

$$P0 \rightarrow P1 \rightarrow P2 \cdots \rightarrow Pn$$

Each pair of processes (*Pi*-1 and *Pi*, where $1 \leqslant i \leqslant$ n) is connected by a sequence $s(i)$ that can hold one or more data elements of type *T*. The sequences $s(1), \ldots, s(n)$ are kept in a common store with a finite *capacity*. Algorithm 8 gives an overview of this *pipeline* system.

The common store is declared as a variable *v* of type *pipeline* (to be defined later). A process *Pi* receives a message *ti* of type *T* from its predecessor and sends an updated message to its successor by means of two procedures

$$receive(ti, v, i) \quad send(ti, v, i+1)$$

Algorithm 9 defines the data type *pipeline* and the procedures *send* and *receive*. A pipeline is a shared record consisting of four components: an array of sequences *s;* an array of integers defining the number of *full* locations ("messages) in each sequence; an array of integers defining the minimum number of store locations *reserved* permanently for transmission of messages through each sequence; and an integer defining the number of store locations that are generally *available* for transmission of messages through all sequences (when they have used up their reserved locations).

Initially, all sequences are empty and the pipeline has been divided into reserved and generally available storage. A sequence $s(i)$ can always hold at least $reserved(i)$ messages. When this amount has been used, the sequence must compete with other sequences for use of the rest of the available store. So the condition for sending a message through sequence $s(i)$ is

$$full(i) < received(i) \textbf{ or } available > 0$$

Algorithm 8 Pipeline system

```
var v: pipeline; t0, t1, . . . , tn: T;
begin
  initialize(v);
  cobegin
    cycle "P0"
      produce(t0);
      send(t0, v, 1);
    end
    . . .
    cycle "Pi"
      receive (ti, v, i);
      update(ti);
      send(ti, v, i + 1);
    end
    . . .
    cycle "Pn"
      receive(tn, v, n);
      consume(tn);
    end
  coend
end
```

Algorithm 9 Pipeline system (cont.)

```
type pipeline = shared record
      s: array 1 . . n of
        sequence of T;
      full, reserved:
        array 1 . . n of integer;
      available: integer;
    end
procedure send(t: T; var v: pipeline; i:
  1 . . n);
region v
  when full(i) < reserved(i) or available
    > 0 do
  put(t, s(i));
  full(i) := full(i) + 1;
  if full (i) > reserved (i) then
    available := available − 1;
  end
end
procedure receive (var t: T; var v: pipe-
  line; i: 1 . . n);
region v
  when full(i) > 0 do
  get(t, s(i));
  if full(i) > reserved(i) then
    available := available + 1;
  end
  full(i) := full(i) − 1;
end
```

The condition for receiving a message through sequence $s(i)$ is

$$full(i) > 0$$

A sequence $s(i)$ may temporarily contain more than $reserved(i)$ messages. But the total number of messages stored in all sequences cannot exceed the capacity of the pipeline. So the system must maintain the following *invariant:*

$$full(i) \geqslant 0 \text{ for } 0 \leqslant i \leqslant n$$

$$available + \sum_i max(full(i), reserved(i)) = capacity$$

It can be shown formally that Algorithm 9 maintains this invariant, but the proof is tedious and adds nothing to one's informal understanding of the pipeline.

In Hoare's conditional critical regions, processes can only be delayed at the beginning of a critical region. In practice, one must be able to place synchronizing conditions anywhere within critical regions as shown in [2].

The conditional critical region is an important description tool for multiprogramming systems. It remains to be seen whether it also is a practical programming tool. The main difficulty of achieving an efficient implementation is the reevaluation of synchronizing conditions each time a critical region is completed.

In simple cases, the reevaluation can be reduced (but seldom eliminated) by the use of semaphores. To do this, one must associate a semaphore with each synchronizing condition When a process makes a condition B true, the process must check whether other processes are waiting for that condition, and, if so, produce a signal that will enable one (and only one!) of them to continue. As I have pointed out elsewhere, [7] the efficiency of this scheme is bought at the expense of increased program complexity.

Concurrent statements and critical regions seem well-suited to the design of small multiprogramming systems dedicated to user applications (but remember that they have not yet been implemented and used in practice). Although these concepts

are simple and well-structured, they do not seem to be adequate for the design of large multiprogramming systems (such as operating systems). The main problem is that the use of critical regions scattered throughout a program makes it difficult to keep track of how a shared variable is used by concurrent processes. It has therefore recently been suggested that one should combine a shared variable and the possible operations on it in a single, syntactic construct called a *monitor* [1, 10, 11]. It is, however, too early to speculate about what this approach may lead to.

10. CONCLUSION

I have tried to show that the design of reliable multiprogramming systems should be guided by two simple principles that are equally valid for sequential programming:

1) It should be possible to understand a program in time-independent terms by an effort proportional to its size.

2) It should be possible to state assumptions about invariant relationships among program components, and have them checked automatically.

I have also discussed some specific language features for multiprogramming. To avoid misunderstanding, I ask you to regard these not as definite proposals but merely as illustrations of a common theme. Better language concepts for multiprogramming will undoubtedly be proposed by others. But I would expect any realistic proposal to:

1) distinguish clearly between disjoint and interacting processes;

2) associate shared data explicitly with operations defined on them;

3) ensure mutual exclusion of these operations in time; and

4) include synchronizing primitives that permit partial or complete programmer control of process scheduling.

ACKNOWLEDGEMENTS

I am indebted to my students Ram Rao, David Smith, and Sankaran Srinivas for many helpful comments on this paper. I am also grateful for the constructive criticism of Coen Bron, Peter Denning, Brian Wichmann, Mike Woodger, and the referees

BIBLIOGRAPHY

1. Brinch Hansen, P. *Operating system principles.* Prentice-Hall, Englewood Cliffs, New Jersey (July 1973).

An introduction to operating systems covering sequential and concurrent processes, processor and store management, scheduling algorithms and resource protection. Describes the RC 4000 *multiprogramming system* in detail. Introduces a programming language notation for monitors based on the class concept of Simula 67.

2 Brinch Hansen, P. "Structured multiprogramming" *Comm ACM* **15,** 7 (July 1972), 574–578

A condensed presentation of the viewpoints described in the present paper It suggests the use of event queues within critical regions as a means of reducing the overhead of process scheduling.

3. Wirth, N. "The programming language Pascal." *Acta Informatica* **1,** 1 (1971), 35–63.

A highly readable definition of a sequential programming language that combines the algorithmic notation of Algol 60 with far more general data structures

4. Dijkstra, E. W. "Cooperating sequential processes" in *Programming Languages,* F. Genuys, (Ed.) Academic Press, New York, New York, 1968.

The classical monograph that introduced concurrent statements, semaphores, and critical regions It also contains Dekker's solution to the mutual exclusion problem.

5 Habermann, A. N. "Synchronization of communicating processes" *Comm ACM* **15,** 3 (March 1972), 171–176.

An axiomatic definition of the semaphore operations, *wait* and *signal*

6 Hoare, C. A R. "Towards a theory of parallel programming." in *Operating Systems Techniques,* C A. R Hoare and R. H. Perrott, (Eds), Academic Press, New York, New York, 1973

The original proposal for conditional critical regions which includes an axiomatic definition of disjoint processes and critical regions.

7. Brinch Hansen, P "A comparison of two synchronizing concepts." *Acta Informatica* **1,** 3 (1972), 190–199.

A comparison of the use of semaphores and conditional critical regions to solve a scheduling problem.

8 Dijkstra, E. W. "Information streams sharing a finite buffer." *Information Processing Letters* **1,** (1972), 179–180.

A solution to the "pipeline" problem by means of conditional critical regions

9. DIJKSTRA, E. W. "The structure of THE multiprogramming system." *Comm ACM* **11,** 5 (May 1968), 341–346.

A brief description of the hierarchical structure of THE multiprogramming system.

10. DIJKSTRA, E. W. "Hierarchical ordering of sequential processes." *Acta Informatica* **1,** 2 (1971), 115–138

A more extensive motivation of the basic design decisions made in THE multiprogramming system (See also 9). Recommends the use of monitors (called "secretaries") for future operating system design.

11. HOARE, C. A. R. "A structured paging system." *Computer Journal* **16,** 3 (August 1973), 209–214.

Illustrates the use of the monitor concept presented in (1) for the design of a demand paging system