STRUCTURED MULTIPROGRAMMING

PER BRINCH HANSEN

(1972)

This paper presents a proposal for structured representation of multiprogramming in a high level language. The notation used explicitly associates a data structure shared by concurrent processes with operations defined on it. This clarifies the meaning of programs and permits a large class of time-dependent errors to be caught at compile time. A combination of critical regions and event variables enables the programmer to control scheduling of resources among competing processes to any degree desired. These concepts are sufficiently safe to use not only within operating systems but also within user programs.

1 Introduction

The failure of operating systems to provide reliable long-term service can often be explained by excessive emphasis on functional capabilities at the expense of efficient resource utilization, and by inadequate methods of program construction.

In this paper, I examine the latter cause of failure and propose a language notation for structured multiprogramming. The basic idea is to associate data shared by concurrent processes explicitly with operations defined on them. This clarifies the meaning of programs and permits a large class of time-dependent errors to be caught at compile time.

The notation is presented as an extension to the sequential programming language Pascal (Wirth 1971). It will be used in a forthcoming textbook to explain operating system principles concisely by algorithms (Brinch Hansen

P. Brinch Hansen, Structured multiprogramming. *Communications of the ACM 15*, 7 (July 1972), 574–578.

1971). Similar ideas have been explored independently by Hoare. The conditional critical regions proposed in (Hoare 1971) are a special case of the ones introduced here.

2 Disjoint Processes

Our starting point is the *concurrent statement*

$$\textbf{cobegin } S_1;\ S_2;\ \ldots;\ S_n \textbf{ coend}$$

introduced by Dijkstra (1965). This notation indicates that statements S_1, S_2, ..., S_n can be executed concurrently; when all of them are terminated, the following statement in the program (not shown here) is executed.

This restricted form of concurrency simplifies the understanding and verification of programs considerably, compared to unstructured *fork* and *join* primitives (Conway 1963).

Algorithm 1 illustrates the use of the concurrent statement to copy records from one sequential file to another.

```
var f, g: file of T;
  s, t: T; eof: Boolean;
begin
  input(f, s, eof);
  while not eof do
    begin t := s;
      cobegin
        output(g, t);
        input(f, s, eof);
      coend
    end
end
```

Algorithm 1 Copying of a sequential file.

The variables here are two sequential files, f and g, with records of type T; two buffers, s and t, holding one record each; and a Boolean, *eof*, indicating whether or not the end of the input file has been reached.

Input and output of single records are handled by two standard procedures. The algorithm inputs a record, copies it from one buffer to another, outputs it, and at the same time, inputs the next record. The copying, output, and input are repeated until the input file is empty.

Now suppose the programmer by mistake expresses the repetition as follows:

```
while not eof do
  cobegin
    t := s;
    output(g, t);
    input(f, s, eof);
  coend
```

The copying, output, and input of a record can now be executed concurrently. To simplify the argument, we will only consider cases in which these processes are arbitrarily *interleaved* but *not overlapped* in time. The erroneous concurrent statement can then be executed in six different ways with three possible results: (1) if copying is completed before input and output are initiated, the *correct* record will be output; (2) if output is completed before copying is initiated, the *previous* record will be output again; and (3) if input is completed before copying is initiated, and this in turn completed before output is initiated, the *next* record will be output instead.

This is just for a single record of the output file. If we copy a file of 10,000 records, the program can give of the order of $3^{10,000}$ different results!

The actual sequence of operations in time will depend on the presence of other (unrelated) computations and the (possibly time-dependent) scheduling policy of the installation. It is therefore very unlikely that the programmer will ever observe the same result twice. The only hope of locating the error is to study the program text. This can be very frustrating (if not impossible) when it consists of thousands of lines and one has no clues about where to look.

Multiprogramming is an order of magnitude more hazardous than sequential programming unless we ensure that the results of our computations are *reproducible in spite of errors*. In the previous example, this can easily be checked at compile time.

In the correct version of Algorithm 1, the output and input processes operate on disjoint sets of variables (g, t) and $(f,\ s,\ eof)$. They are called *disjoint* or *noninteracting processes*.

In the erroneous version of the algorithm, the processes are not disjoint: the output process refers to a variable t changed by the copying process; and the latter refers to a variable s changed by the input process.

This can be detected at compile time if the following rule is adopted: a concurrent statement defines disjoint processes $S_1, S_2, \ldots, S_n$ which can be executed concurrently. This means that a variable v_i changed by statement S_i cannot be referenced by another statement S_j (where $j \neq i$). In other words, we insist that a variable subject to change by a process must be strictly *private* to that process; but disjoint processes can refer to *shared* variables not changed by any of them.

Throughout this paper, I tacitly assume that sequential statements and assertions made about them only refer to variables which are *accessible* to the statements according to the rules of disjointness and mutual exclusion. The latter rule will be defined in Section 3.

Violations of these rules must be detected at compile time and prevent execution. To enable a compiler to check the disjointness of processes the language must have the following property: it must be possible by simple inspection of a statement to distinguish between its constant and variable parameters. I will not discuss the influence of this requirement on language design beyond mentioning that it makes unrestricted use of *pointers* and *side-effects* unacceptable.

The rule of disjointness is due to Hoare (1971). It makes the *axiomatic property* of a concurrent statement S very simple: if each component statement S_i terminates with a result R_i provided a predicate P_i holds before its execution then the combined effect of S is the following:

$$\text{“}P\text{”}\ S\ \text{“}R\text{”}$$

where

$$\begin{aligned} P &\equiv P_1 \ \&\ P_2 \ \&\ \cdots \ \&\ P_n \\ R &\equiv R_1 \ \&\ R_2 \ \&\ \cdots \ \&\ R_n \end{aligned}$$

As Hoare puts it: “Each S_i makes its contribution to the common goal.”

3 Mutual Exclusion

The usefulness of disjoint processes has its limits. We will now consider *interacting processes*—concurrent processes which access shared variables.

A *shared variable* v of type T is declared as follows:

$$\textbf{var}\ v\text{: }\textbf{shared}\ T$$

Concurrent processes can only refer to and change a shared variable inside a structured statement called a *critical region*

region v **do** S

This notation associates a statement S with a shared variable v.

Critical regions referring to the same variable exclude each other in time. They can be arbitrarily interleaved in time. The idea of progressing towards a final result (as in a concurrent statement) is therefore meaningless. All one can expect is that each critical region leaves certain relationships among the components of a shared variable v unchanged. These relationships can be defined by an assertion I about v which must be true after initialization of v and before and after each subsequent critical region associated with v. Such an assertion is called an *invariant*.

When a process enters a critical region to execute a statement S, a predicate P holds for the variables accessible to the process outside the critical region and an invariant I holds for the shared variable v accessible inside the critical region. After the completion of S, a result R holds for the former variables and invariant I has been maintained. So a critical region has the following axiomatic property:

"P"
region v **do** "$P\&I$" S "$R\&I$";
"R"

4 Process Communication

Mutual exclusion of operations on shared variables makes it possible to make meaningful statements about the effect of concurrent computations. But when processes cooperate on a common task they must also be able to wait until certain conditions have been satisfied by other processes.

For this purpose I introduce a synchronizing primitive, **await**, which delays a process until the components of a shared variable v satisfy a condition B:

region v **do**
begin ... **await** B; ... **end**

The await primitive must be textually enclosed by a critical region. If critical regions are nested, the synchronizing condition B is associated with the innermost enclosing region.

The await primitive can be used to define *conditional critical regions* of the type proposed in (Hoare 1971):

"Consumer"	"Producer"
region v **do**	**region** v **do** S_2
begin await B; S_1 **end**	

The implementation of critical regions and await primitives is illustrated in Fig. 1. When a process, such as the consumer above, wishes to enter a critical region, it enters a *main queue* Q_v associated with a shared variable v. After entering its critical region, the consumer inspects the shared variable to determine whether it satisfies a condition B. In that case, the consumer completes its critical region by executing a statement S_1; otherwise, the process leaves its critical region *temporarily* and joins an *event queue* Q_e associated with the shared variable.

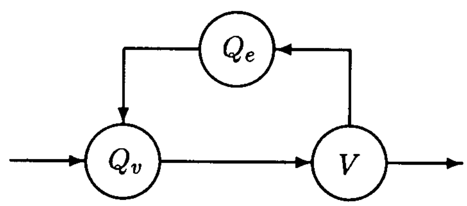

Figure 1 Scheduling of conditional critical regions V by means of process queues Q_v and Q_e.

All processes waiting for one condition or another on variable v enter the same event queue. When another process (here called the producer) changes v by a statement S_2 inside a critical region, it is possible that one or more of the conditions expected by processes in the event queue will be satisfied. So, after completion of a critical region, all processes in the event queue Q_e are transferred to the main queue Q_v to enable them to reenter their critical regions and inspect the shared variable v again.

It is possible that a *consumer* will be transferred in vain between Q_v and Q_e several times before its condition B holds. But this can only occur as frequently as *producers* change the shared variable. This controlled amount of *busy waiting* is the price we pay for the conceptual simplicity achieved by using arbitrary Boolean expressions as synchronizing conditions.

The desired *invariant* I for the shared variable v must be satisfied before an *await* primitive is executed. When the waiting cycle terminates, the assertion B & I holds.

As an example, consider the following resource allocation problem: two kinds of concurrent processes, called readers and writers, share a single resource. The readers can use the resource simultaneously, but the writers must have exclusive access to it. When a writer is ready to use the resource, it should be enabled to do so as soon as possible.

This problem is solved by Algorithm 2. Here variable v is a record consisting of two integer components defining the number of *readers* currently using the resource and the number of *writers* currently waiting for or using the resource. Both *readers* and *writers* are initialized to zero.

```
var v: shared record readers, writers: integer end
    w: shared Boolean;

"Reader"                        "Writer"
region v do                     region v do
begin                           begin
  await writers = 0;              writers := writers + 1;
  readers := readers + 1;         await readers = 0;
end                             end
read;                           region w do write;
region v do                     region v do
readers := readers − 1;         writers := writers − 1;
```

Algorithm 2 Resource sharing by readers and writers.

Mutual exclusion of readers and writers is achieved by letting readers wait until the number of writers is zero, and vice versa. Mutual exclusion of individual writers is ensured by the critical region on the Boolean w.

The priority rule is obeyed by increasing the number of writers as soon as one of them wishes to use the resource. This will delay subsequent reader requests until all pending writer requests are satisfied.

A correctness proof of Algorithm 2 is outlined in (Brinch Hansen 1972). In this paper I also point out the superiority of conditional critical regions over *semaphores* (Dijkstra 1965). Compared to the original solution to the problem (Courtois 1971) Algorithm 2 demonstrates the conceptual advantage of a structured notation.[1]

[1]The original solution includes the following refinement: when a writer decides to make a request at most one more reader can complete a request ahead of it. This can be ensured by surrounding the reader request in Algorithm 2 with an additional critical region associated with a shared Boolean r.

The conceptual simplicity of critical regions is achieved by ignoring details of scheduling: the programmer is unaware of the sequence in which waiting processes enter critical regions and access shared resources. This assumption is justified for processes which are so *loosely connected* that simultaneous requests for the same resource rarely occur.

But in most computer installations *resources* are *heavily used* by a large group of users. In this situation, an operating system must be able to *control the scheduling of resources explicitly* among competing processes.

To do this a programmer must be able to associate an arbitrary number of event queues with a shared variable and control the transfers of processes to and from them. In general, I would therefore replace the previous proposal for conditional delays with the following one:

The declaration

var e: **event** v;

associates an event queue e with a shared variable v.

A process can leave a critical region associated with v and join the event queue e by executing the standard procedure

await(e)

Another process can enable all processes in the event queue e to reenter their critical regions by executing the standard procedure

cause(e)

A consumer/producer relationship must now be expressed as follows:

"Consumer"	"Producer"
region v **do**	**region** v **do**
begin	**begin**
while not B **do** await(e);	S_2;
S_1;	cause(e);
end	**end**

Although less elegant than the previous notation, the present one still clearly shows that the consumer is waiting for condition B to hold. And we can now control process scheduling to any degree desired.

To simplify explicit scheduling, I suggest that processes reentering their critical regions from event queues take priority over processes entering critical

```
var v: shared record
          available: set of R;
          requests: set of P;
          grant: array P of event v;
        end

procedure reserve(process: P; var resource: R);
region v do
begin
  while empty(available) do
  begin enter(process, requests);
    await(grant[process]);
  end
  remove(resource, available);
end

procedure release(resource: R);
var process: P;
region v do
begin enter(resource, available);
  if not empty(requests) then
  begin remove(process, requests);
    cause(grant[process]);
  end
end
```

Algorithm 3 Scheduling of heavily used resources.

regions directly through a main queue (see Fig. 1). If the scheduling rule is completely unknown to the programmer as before, additional variables are required to ensure that resources granted to waiting processes remain available to them until they reenter their critical regions.

Algorithm 3 is a simple example of completely controlled resource allocation. A number of processes share a pool of equivalent resources. Processes and resources are identified by indices of type P and R respectively. When resources are *available*, a process can *acquire* one immediately; otherwise, it must enter a request in a data structure of type *set of P* and wait until a resource is *granted* to it. It is assumed that the program controls the entry and removal of set elements completely.

5 Conclusion

I have presented structured multiprogramming concepts which have simple axiomatic properties and permit extensive compile time checking and generation of efficient machine code.

The essential properties of these concepts are:

1. A distinction between disjoint and interacting processes;
2. An association of shared data with operations defined on them;
3. Mutual exclusion of these operations in time;
4. Synchronizing primitives which permit partial or complete control of process scheduling.

These are precisely the concepts needed to implement *monitor procedures* such as the ones described in (Brinch Hansen 1970). They appear to be sufficiently safe to use not only within operating systems but also within user programs to control local resources.

References

Brinch Hansen, P. 1970. The nucleus of a multiprogramming system. *Communications of the ACM 13*, 4 (April), 238–250.

Brinch Hansen, P. 1971. An outline of a course on operating system principles. *International Seminar on Operating System Techniques*, Belfast, Northern Ireland, (August–September).

Brinch Hansen, P. 1972. A comparison of two synchronizing concepts. *Acta Informatica 1*, 190–199.

Conway, M.E. 1963. A multiprocessor system design. *Proc. AFIPS FJCC 24*, Spartan Books, New York, 139–146.

Courtois, P.J, Heymans, F., and Parnas, D.L. 1971. Concurrent control with "readers" and "writers." *Communications of the ACM 14*, 10 (October), 667–668.

Dijkstra, E.W. 1965. Cooperating sequential processes. Technological University, Eindhoven. Also in *Programming Languages*, F. Genyus, Ed. Academic Press, New York, 1968.

Hoare, C.A.R. 1971. Towards a theory of parallel programming. *International Seminar on Operating System Techniques*, Belfast, Northern Ireland, (August–September).

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