Brinch Hansen’s paper “Distributed Processes: A Concurrent Programming Concept” [1] proposes language constructs for synchronization and communication among concurrent processes in real-time computer systems. The scheme is attractive, but several aspects of the proposal seem to have inherent weaknesses. In the hope of directing further research toward the correction of these deficiencies, the following comments are offered.

1. INTRODUCTION
In the distributed processes environment each process executes on a separate processor. The initial statement of a process executes until it either finishes or waits for one of the guards (Boolean expressions) of a guarded region [2] to become true. At that point an external request from another process may be honored. External requests are invocations of procedure entry points in the called process. The external request executes until the called procedure finishes or waits. (The Boolean expressions which constitute guards may include parameters of the external procedure entry points.) The switching among operations (external requests or the initial statement) occurs only at completions or waits. (The Boolean expressions which constitute guards may include parameters of the external procedure entry points.) The calling process performs no operations until the external request is completed. External requests are the only means of communication and synchronization.

Brinch Hansen’s proposal is attractive in its simplicity. However, certain assumptions about the real-time environment (infinite life span of processes and the desirability of dedicating one processor to each process) are unrealistic. The indeterminacy of the real-time environment is too strongly reflected in an unnecessary lack of control over the order of interleaving of operations. Completely unpredictable interleaving makes it impossible to implement the priority policies which are necessary in a real-time environment. In addition, the use of parameters in guards causes reevaluation overhead proportional to the number of waiting operations.

These problems are not necessarily inherent in the basic approach of distributed processes; it may be that further development of the concept can include modifications correcting them.

2. PRIORITIES AND NONDETERMINISM
Dijkstra [2] has given some persuasive arguments favoring nondeterministic selection among the “true” guards of an alternative statement construct for sequential programs. These arguments do not necessarily transfer directly to the concurrent programming environment. Furthermore, a programmer who wishes to do so can easily circumvent the nondeterminism inherent in Dijkstra’s sequential guarded commands by introducing Boolean-valued variables and simulating the familiar if ... then ... else ... construct. In distributed processes, the guards within a given guarded region may be assigned relative priorities by similar techniques.

No such artifice is available to establish programmer-defined priorities among the eligible external requests of a process in [1]. Until the implicit (nondeterministic) scheduler has actually activated a new external request, it can...
neither bind any parameter value nor set the value of any process variable that might be used by a process to determine priority. The only circumstance under which the distributed processes scheme guarantees that a given external request will be scheduled is when no other task (initial statement or external request) is ready to execute.

The lack of priority mechanisms is a pervasive problem. Although the order of requests in a real-time system is not predictable, the order in which pending requests must be serviced may be quite critical.

The point is readily illustrated if we consider a variant of the shortest-job-next scheduler given as an example in [1]. Let us suppose that the resource whose use is to be scheduled is encapsulated within the same process module that performs the scheduling:

```plaintext
process Sjn
  Queue: set[n] Int; Rank: array[n] Int;
  User, Next, Min: Int;
  ... resource declarations ...

proc Service(Who, Time: Int);
  begin
    Queue.Include(Who);
    Rank[Who] := Time; Next := nil;
    when User = Who:
      ... use of resource ...
      Next := nil; User := nil;
    end
  end
begin
  Queue := []; User := nil; Next := nil;
  ... initialize resource ...
  cycle
    (not Queue.Empty) & (Next = nil):
      Min := Maxinteger;
      for i in Queue:
        if Rank[i] > Min: skip
          Rank[i] <= Min:
            Next := i;
            Min := Rank[i];
          end
        end
      (User = nil) & (Next <> nil):
        User := Next; Queue.exclude(User);
    end
end
```

The significant difference between this version of the example and that given in [1] is that in this version we can no longer justify the assumption that the processor is almost always available to serve a new request. It is possible that external requests may accumulate at the entry point of the procedure `Service` while the common resource is being used by a prior request.

If this should happen, then, at the next time the processor completes the service of a request, a choice must be made among actions including the following:

1. initiating a new external request, which would allow one of the waiting external processes to register its request for service;
2. resuming the initial statement, either to schedule the next request for service or to enable that service and terminate further scheduling; or
3. serving an enabled request by resuming execution of an invocation of the `Service` procedure in its `when` statement.

Clearly, a nondeterministic selection among these possibilities is not appropriate. Every arriving service request should be given precedence over execution of the scheduling algorithm or initiation of service so that the new request may provide the service time estimate that is to establish its relative priority. Otherwise, the shortest-job-next algorithm for scheduling actual use of the common resource will not be effective.

We can draw the following conclusions concerning the importance of a language notation used to express relative priorities of the several tasks of a process:

1. Without an expression of priority (i.e., if scheduling is nondeterministic) and without preemption, a programmer cannot be sure that a scheduling algorithm written in the language will be effective.
(2) If static priorities can be attached to the various tasks of a process, then effective scheduling algorithms can be programmed.

3. INTERRUPTS AND PREEMPTIVE SCHEDULING

Interrupts are a special instance of the priority problem. An example from computer graphics illustrates the difficulties.

Example. The cursor on a CRT is being continuously redrawn by the process Display. At irregular intervals the process Update generates new coordinates and passes them to Display so that the cursor may be moved to a new position. Let us say Update is of the form

```
process Update
  X, Y: Int;
  cycle
    true:
      begin
        generate coordinates X, Y;
        Display.Send(X, Y);
      end
  end
```

How can Display be coded within the framework of distributed processes? One nonsolution is

```
process Display
  Xd, Yd: Int; New: Bool;
proc Send(X, Y: Int);
  begin
    Xd := X;
    Yd := Y;
    New := true;
  end
begin
  New := false;
  cycle
    true:
      paint cursor at Xd, Yd
    end
end
```

This formulation of Display fails because the initial cycle statement never waits. The external request to the procedure Send never executes. But what can Display safely wait upon? Display cannot wait for the Boolean New; the cursor must be continuously refreshed even if another Update never occurs. Display needs to wait for New only when New is set, but New can never be set until Display waits. The only way another process can affect Display's variables is by an external request, but external requests are not allowed to execute so long as the initial statement continues.

Since external information is obtainable only via external requests, the only solution is to have Display make the external request. Display must call some external procedure (either a modified Update or some buffer process) to obtain current coordinates each time it paints the cursor.

This solution replaces interrupts with a polling scheme requiring interprocess communication—a costly alternative. Any process doing continuous, useful work which involves no waiting must frequently poll for changes in the environment. Furthermore, if an initial data-gathering process such as Update provides information to many customer processes such as Display, care must be taken that the processor which executes Update remains lightly loaded. Otherwise, in the presence of the nondeterministic scheduling of external requests, one or more of the polling requests from a Display process may be subjected to arbitrary and intolerable delay.

4. CONCLUSIONS AND SPECULATION

The stipulation of FIFO behavior for ready-to-continue operations (or at least a prohibition of indefinite overtaking) would at least make it possible to program priority schemes. The ability to associate static priorities with the entry procedures would be of even greater utility. A priority parameter at waits (giving a priority-ordered queue for ready-to-continue operations) would greatly sim-
plify the programming of many priority schemes. The overhead of evaluating guards for many waiting operations might be reduced by prohibiting direct inclusion of procedure parameters in guards (a scheme analogous to that advocated by Kessels for monitors [3]). These and other possibilities should be explored.

R. L. EARLE
Department of Mathematical Sciences
The University of Texas at Dallas
Richardson, TX 75080

R. B. KIEBURTZ
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794

A. SILBERSCHATZ
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

REFERENCES