Priority and Queuing Specification in ‘Distributed Processes’

Abraham Silberschatz
Department of Computer Science, The University of Texas, Austin, Texas 78712, USA

In a recent paper, Brinch Hansen has proposed a new language concept for process coordination in a distributed environment. A major weakness of the proposal is the lack of an effective control mechanism for scheduling the various ready-to-run activities within a single process. This paper addresses this issue by proposing simple extensions to the language to resolve this difficulty.

INTRODUCTION

In a recent paper Brinch Hansen has proposed a new language concept for process coordination in a distributed environment. Central to this language is the concept of a distributed process, which is a program module that is executed asynchronously. A process consists of some shared data, common (external) procedures and an initial statement. A process can access its shared data as well as invoke external procedures defined in other processes.

A process switches between various activities. An activity is the execution of a procedure (on behalf of another process) or the execution of the initial statement of the process. At any point in time a process is executing at most one activity. It switches activities only when it either reaches a false guarded region (i.e. a guarded region whose associated boolean expressions are all false), or it reaches the last statement in either a procedure or the initial statement. In this case it may start (or resume) processing the next activity.

Although the Distributed Processes (DP) scheme is attractive, its principal weakness is the lack of an effective control mechanism for the scheduling of the various ready-to-run activities within a single process. It is our aim here to elaborate on this issue, by proposing simple extensions to the language to resolve this difficulty.

QUEUING SPECIFICATIONS

The non-deterministic control structure of DP is the guarded region construct. In DP, a guarded boolean expression may include some of the process’ own variables as well as formal parameters passed in a procedure call. The inclusion of formal parameters in a boolean expression is very powerful, but it is also very expensive. In general, such boolean guards must be re-evaluated each time control switches among the various activities. In such an environment, the overhead for a process increases in direct proportion to the number of activities ready to be resumed within that process.

In some synchronization schemes such an overhead is expected. There are however many common synchronization problems (e.g. FIFO, shortest-job-next, alarm clock, disk scheduler) where such an overhead can be avoided via the use of a static priority-queue. Unfortunately no such queuing facility is available in DP.

We propose a simple extension to the guarded region construct to allow queuing specification in DP. The when and cycle statements of DP are to be redefined as follows:

\[
\text{when } [(\text{priority-expression})] \\
B_1 : S_1 | B_2 : S_2 | \ldots | B_n : S_n \\
\text{end}
\]

\[
\text{cycle } [(\text{priority-expression})] \\
B_1 : S_1 | B_2 : S_2 | \ldots | B_n : S_n \\
\text{end}
\]

The \textit{priority expression} is optional. For reasons that will become apparent later, a priority expression can only appear in \textit{when(cycle)} statements embedded in a common procedure.

When a process encounters a \textit{when(cycle)} statement whose associated boolean expressions are false, then the priority-expression is evaluated to produce an integer priority value. This priority value is stored in the activation-record associated with the called procedure. At some later point in time when the \textit{when(cycle)} statement is re-evaluated and found to be true (for either one procedure, or several), then the ready to run procedure (activity) whose associated priority-value is the smallest is resumed next. If no priority-value is present, an arbitrary ready to run procedure (activity) is resumed.

Let us illustrate this concept by considering a simple resource scheduling scheme. This example will allow us to compare our scheme with Brinch Hansen’s original one. Suppose that one wishes to define a process whose function is to allocate a resource among a number of processes in the shortest job next order.

\begin{verbatim}
process SJN;
  Free: boolean;
  proc Request (Time: int);
    when (Time)
      Free: Free := false;
    end;
  end;
  proc Release;
    Free := true;
  end;
end;
\end{verbatim}

It would be interesting to compare our solution with Brinch Hansen’s (see Ref. 1, p. 937). Our solution is much more concise, easier to understand, and more efficient; this is because Brinch Hansen does not provide an effective mechanism for handling static priority queues.
Finally we note, that we have tried our scheme on such scheduling problems as FIFO, alarm clock, the disk-elevator algorithm, etc. and found our algorithms to be more concise and efficient than those coded in the original DP.

PRIORITY SPECIFICATION

At any given instant there may be a number of activities ready to execute within a process. These include: initial calls to the various common procedures, older activities ready to be dequeued from guarded regions, and the initial statement. Different scheduling constructs allow different possibilities, but in all cases there can be many ready activities. Although nondeterminacy is both desirable and inevitable, the language designer must be careful to allow the programmer enough control to implement his own long-term scheduling policies. A computer system may be nondeterministic, but there are likely to be very definite priorities performed within its processes.

Priorities among various activities can be partitioned into two groups: static priorities—those that can be determined at compile time, and dynamic priorities—that those need to be evaluated at run time. As an example of the first type of priority consider the SJN process defined in the previous section. In order to maximize resource utilization the Release procedure should be always executed before the Request procedure; this, however, is beyond the control of the programmer.

As an example of the second type of priority, consider a background process which almost always has useful work to do but whose operation is occasionally modified by an external request (i.e. interruption). This situation was illustrated in Ref. 3 with a bounded buffer scheme that is to be implemented on a distributed processing architecture. In such an environment (i.e. where no shared memory is assumed), the bounded buffer scheme should be implemented (for efficiency reason) by encapsulating the N buffers in the Producer process rather than in a separate buffering process (as has been traditionally done). Unfortunately, such a scheme could not be easily implemented in DP.

Since DP does not provide an effective mechanism to handle these two kinds of priorities, a mechanism is presented below to allow a programmer to explicitly specify the order in which the various activities within a process should be executed.

Static priorities

Static priorities can be easily handled by allowing the programmer to define a partial order among the various procedures (and the initial statement). A possible syntax is

order \( A_1 < A_2 < A_3 < A_4 < \ldots < A_{n-1} < A_n \)

Where each \( A_i \) is either a procedure name or the name of the process (corresponding to the initial statement).

The order list specifies an order in which activities should be processed. For example, consider the following statement:

order \( A(B; A(C) \);
the Buffers are full or when the Consumer has invoked
the Get procedure. Such a scheme could not be imple-
mented easily (if at all) in DP.

CONCLUSION

We have pointed out two problems with Brinch Hansen’s
Distributed Processes concept. The first one concerns the
repeated evaluation of the boolean wait expressions in
the guarded regions when formal parameters are in-
cluded. This causes overhead proportional to the number
of waiting activities within that process. Since there are
many common synchronization problems where such an
overhead can be avoided (via the use of static priority
queues) we have proposed a simple extension to the
guarded region construct to allow queuing specification
to be explicitly defined by the programmer.

The second problem concerns the insufficient control
over the scheduling of the ready to run activities. We
have identified two types of priorities among the various
activities. We have shown that static priorities can be
handled via the use of the order list concept, while
dynamic priorities can be handled via the use of the ?
function. We have demonstrated how these two features
can be utilized in the scheduling of the various activities
within a single process.

Acknowledgment

This research was supported in part by the National Science Foundation
under Grant MCS 7702463, and in part by the Office of Naval Research
under Contract N00014-80-K-0987.

REFERENCES

1. P. Brinch Hansen, Distributed Processes: a concurrent pro-
gramming concept. Communications of the ACM 21 (No. 11), 934–
940 (November 1978).
2. E. W. Dijkstra, Guarded Commands, Nondeterminacy, and
Formal Derivation of Programs, Communications of the ACM 18
(No. 8), 453–457 (August 1975).
3. A. Silberschatz, On the decomposition of distributed systems
into modules. Proceeding of COMPCON Fall 80 (September
1980).
5. J. D. Ichbiah, et al., Rationale for the design of the ADA
programming language, ACM Special Interest Group for Pro-
gramming Languages (SIGPLAN) Notices 14 (No. 6), part B
(June 1979).

Received August 1980
© Heyden & Son Ltd, 1982