Extending CSP to Allow Dynamic Resource Management

ABRAHAM SILBERSCHATZ

Abstract—In his paper “Communicating Sequential Processes,” Hoare suggested the use of the input/output construct and Dijkstra’s guarded commands for handling the task of communication and synchronization in distributed systems. Hoare’s proposal was intended for programming general parallel systems; as a result, little consideration was given by Hoare to the question of how his mechanisms could be utilized in the construction of reliable dynamic resource management schemes. In this paper, we examine this problem and propose several simple extensions to Hoare’s constructs that will make the extended Communicating Sequential Processes concept more suitable for the handling of such management schemes.

Index Terms—Communication ports, distributed systems, guarded commands, input/output commands, programming languages, resource management, synchronization.

I. INTRODUCTION

In his paper “Communicating Sequential Processes” [1] (CSP), Hoare introduced a language concept for concurrent processing which is suitable for microcomputer network environment with distributed storage. Hoare emphasized in his paper that his proposal is at best only a partial solution to the problem of finding an appropriate language for concurrent programming. In particular, Hoare noted that his proposal was mainly intended for programming general parallel systems; as a result, little consideration was given by Hoare to the question of how his mechanisms could be utilized in the construction of reliable and efficient resource management schemes.

Recently, we have presented some simple extensions to Hoare’s constructs, which made the extended communicating sequential process concept more suitable for the writing of resource management schemes. In particular, we have proposed the concept of a communication port [2] which provides the appropriate mechanisms for relaxing the explicit naming requirement and permitting the inclusion of both input and output commands in guards.

One issue that the port concept cannot effectively handle is priority scheduling. In this paper we address this issue by proposing a simple extension to the port construct. We then argue that the new proposed scheme could be effectively used in the construction of reliable and efficient resource management schemes.

II. PORT-DIRECTED COMMUNICATION

Before presenting our results we need to briefly survey the central concepts of the port scheme. It should be noted that the concept of ports as a communication mechanism is not new [3], [4]. What is new about the communication-port scheme is that it pertains specifically to Hoare’s work. Moreover, it has a different semantic interpretation.

Each CSP process can declare in his address space a set of port names. This declaration is accomplished via the statement:

\[
\text{<list of port names>}: \text{port};
\]

For example, a process P may declare:

\[
A, B, C: \text{port};
\]

We will say that process P is the owner of the ports A, B, and C. Each port can have one and only one owner. The notion of ownership of a port plays an important role in our discussion concerning synchronization and communication.

A process can use a port that it does not own, by declaring:

\[
\text{use <list of port names>};
\]

For example, a process Q may declare:

\[
\text{use A, C};
\]

This declaration specifies that process Q may communicate with process P through the ports A and C. We will denote this fact by saying that Q is a user of these two ports. Thus, in general, a port has one owner and several users.

Suppose that two processes wish to communicate. In order to accomplish this the processes need to employ a common port. Moreover, one of these processes needs to be the owner of that port. As with Hoare’s proposal, communication through ports is accomplished through the input and output commands. The only difference is that port names are used instead of process names.

There is one main semantic difference between Hoare’s proposal and ours. In the original CSP an I/O command specifies exactly one communicating partner while in our proposal an I/O command may involve several processes; however, only one of them will be selected for communication. More precisely, let A be a port with owner P and users U_i, i = 1, 2, \ldots, 10. Suppose that processes U_1, U_5, and U_7 are ready to do I/O (e.g., they have each invoked an I/O command involving port A). Further suppose that process P has also invoked an I/O command which matches the ones of U_1 and U_7. In this case communication occurs between either the pair

Manuscript received April 10, 1980; revised September 12, 1981 and March 22, 1983. This work was supported in part by the Office of Naval Research under Contract N00014-80-K-0987 and in part by the National Science Foundation under Grant MCS 81-04017.

The author is with the Department of Computer Science, University of Texas, Austin, TX 78712.

0098-5589/83/0700-0527$01.00 © 1983 IEEE
(P, U1) or the pair (P, U7) but not both. The choice as to which pair will be selected is not known. This is another means by which nondeterminism is introduced in our proposal (the other one being Dijkstra’s guarded command [5]).

In contrast to Hoare’s proposal we allow both input and output commands to appear in guards. This, however, is restricted to the case where a process may have I/O commands in guards, but these commands may involve only those ports that it owns. The reason for this restriction is that it allows one to obtain a uniform efficient implementation of the I/O commands. We note, however, that our proposal is as powerful as the one presented by Hoare (see [21]). We feel, however, that our scheme allows a more structured approach to the problem of writing programs in distributed systems because it allows the simplification of a large class of algorithms and reduces the number of I/O commands that are needed in the implementation of these algorithms.

III. Resource Allocation Schemes

A primary aim of an operating system is to provide a safe and efficient mechanism for sharing the computer resources among competing jobs. It has become by now a common practice to provide a separate scheduler for each class of common resources (either physical or abstract). The most known construct that supports such a notion is the monitor [6], [7] concept that was developed from Dijkstra’s secretary [8] concept. If CSP is to be used as a design tool for writing operating systems it should provide an effective mechanism to allow the construction of such resource schedulers. Unfortunately, neither Hoare’s scheme, nor the port scheme are sufficient for such a task.

Hoare’s scheme suffers because of the explicit naming requirement, which implies that process identity must be known statically (i.e., at compile time). On the other hand, the port scheme suffers because of the lack of a mechanism for identifying processes which communicate through a particular port. Thus the only way to achieve resource scheduling using ports is to declare array of ports; this brings us back to the problem of explicit naming which we sought to avoid. What is needed is a mechanism that will provide the owner of a port with the capabilities:

1) to find out with whom communication took place,
2) to selectively choose those processes with whom communication may take place at some point in time.

In the following we propose a mechanism that supports such a scheme.

Let us start by considering a simple scheduling problem. Suppose that we wish to write a scheduler process, SJN, that allocates a resource among N user processes in the Shortest Job Next order.

Let F be a port declared locally to the scheduler process SJN. A user process requests allocation by executing:

\[ F! \text{Acquire}(T); \]

\[ F? \text{Ack}(T); \]

where T is an integer type variable specifying the length of time the requesting process will use the resource.

It is possible in this scheme to have several user processes awaiting an acknowledgment [e.g., each user is suspended on \( F!\text{Ack}(T) \)]. Recall that the programmer has no control as to which of these suspended processes will be selected when \( F!\text{Ack}(T) \) is executed. However for the Shortest-Job-Next example such a mechanism is needed.

If a system is composed of a number of disjoint user processes that interact only via a common scheduler through input/output commands, then in order for a scheduler to enforce a particular scheduling policy it may need to know the identity of its users so that it can grant the resource to a specific user. We propose that this should be done by assigning the identity of each user process to the port name after the successful completion of an input request to the scheduler. The owner of a port can then query the port to determine the identity of the communicating process, stored this identity in a queue for later reference, etc. This can be accomplished by allowing the owner of a port to use the name of a port within an assignment statement.

Let us be more specific. Suppose that A is a port name with owner P and users \( U_i, i = 1, 2, \cdots, 10 \). Suppose that at some point in time a match occurred between the output command of process \( U_5 \) (e.g., \( A?n \)) and the input command of process \( P \) (e.g., \( A?n \)). The effect of executing these I/O commands is equivalent to the assignments

a) \( n \leftarrow m \);
b) \( A \leftarrow U_5 \);

Note that the value of port A is affected only when the owner of A (i.e., process P) invokes an input command. The case where output commands also affect the value of a port does not seem to be useful in general.

Having provided a simple mechanism for finding out the identity of user processes, we focus our attention now on how this can be used in the construction of resource schedulers. Returning to our Shortest-Job-Next example, it is clear that the scheduler needs a mechanism for selectively choosing among these users that required an allocation. This can readily be done by adding a new interpretation as to how I/O command matching occurs.

Let P be the owner of a port A. Suppose that P has encountered an I/O command. Further suppose that the value of port A is \( U_5 \). In this case a match can occur only with an I/O command of process \( U_5 \). Note that this function is carried out by the port mechanism and hence does not require explicit programming.

Since the above interpretation of matching is not always required, we will allow the programmer to designate those ports which employ the selective function described above. Moreover, we will allow a user to specify whether such a selection will be accomplished during the execution on input command, output command, or both. To do this we extend the port declaration as follows:

\[ \text{port } [1] \text{--selection function invoked only when output command executed,} \]

\[ \text{port } [2] \text{--selection function invoked only when input command executed,} \]
port ![, ?] \dash selection function invoked when either an input or output command executed.

We emphasize that only for those ports declared with the selective attribute does the compiler use the above interpretation.

Let us illustrate our concepts by writing the Shortest-Job-Next scheduler. For simplicity, we assume that process-id's are integers varying between 0 and N - 1, and that the largest time that can be specified by a custom process is L - 1.

**SJN:**

- **F:** port ![]
- **Time:** (0..N) integer;
- **Busy:** Boolean;
- **Count, T, I, Min, Next:** integer;
- **Busy := false;**
- **Count := 0;**
- **I := 0;**
- **[* [I < N \rightarrow \text{TIME}[I] := L; I := I + 1] **
- **[* F?Acquire(T) \rightarrow [\text{Busy} \rightarrow \text{ENTER}

  \[ \text{!\neg Busy} \rightarrow \text{Busy := true;}
  \]
  
  \text{F!Ack()}

  \]
- **[* F?Release( ) \rightarrow [\text{Count > 0} \rightarrow \text{REMOVE}

  \[ \text{!\neg Count} \rightarrow \text{Busy := false} \]

  \]

Where **ENTER** and **REMOVE** are an abbreviation for

**ENTER** = Time[F] := T;  
Count := Count + 1;

**REMOVE** = I := 0; Min := L;  
[* [I < N \rightarrow [\text{TIME}[I] < \text{Min} \rightarrow \text{Min} := \text{TIME}[I]; Next := I;  

  \[ \text{!\neg Time}[I] \rightarrow \text{Min} \rightarrow \text{Skip} \]

  \]

  I := I + 1;  

  ]

Time[Next] := L;  
Count := Count - 1;  
F := Next;  
F!Ack();

Let us examine in greater detail the Shortest-Job-Next scheme. In this example, the integrity of the scheme rests upon the correct use of the SJN scheduler; that is, each process wishing to operate on the shared resource must observe the sequence,

- **F!Acquire(T);**
- **F!Ack();**
- `<use resource>`
- **F!Release( );**

Unfortunately, the process concept alone cannot guarantee that such a sequence will be observed. In particular,

1) a process might operate on a resource without first gaining access permission to it (via a direct call to the shared resource);
2) a process might never release a resource once it has been granted an access right to it;
3) a process might attempt to release a resource that it never requested;

4) a process might request the same resource twice (without first releasing it), etc.

These modes of system behavior are particularly undesirable in that correctly written processes can suffer due to the errant performance of a single incorrect process, even though an access control mechanism has been provided.

We will show now how the mechanisms proposed thus far can be effectively used in resolving the above difficulties. There are two issues that need to be dealt with.

1) A mechanism must be available to allow the programmer to dynamically discriminate among different processes that can access a common resource.

2) A mechanism must be available for associating a set of resources (or a single resource) with a particular "Allocator" process. Once such an association has been established one must ensure that only the Allocator process can dictate a policy as to who can use its controlled resources.

We will show below (through an example) how our proposed mechanism can be used in dealing with the above two issues.

Let us start by programming the shared resource controlled by the SJN scheduler.

**Resource:**

- **S, R:** port [?];
- **Q:** process-id;
- **R := SJN;**

Let us define the following functions:

**S := Resource;**

[* R?Set(Q) \rightarrow S := Q  

  \[ \text{!\neg R?Reset( )} \rightarrow S := \text{Resource} \]

  \[ S?Use(\ldots) \rightarrow \text{PERFORM SERVICE FUNCTION} \]

Note that the process Resource can be used (i.e., perform the service function) only by the process whose identity is stored in the port S. The setting and resetting of this variable is solely controlled by the scheduler SJN. We emphasize that these are guaranteed to hold by our port selection mechanism.

We can now reprogram the SJN process. For brevity we only present those code segments which need to be revised. We also introduce one additional item variable "Current" of type process-id, to record the identity of the process currently using (owning) the resource. The variable Current is initialized to SJN.

[* F?Acquire(T) \rightarrow [F = \text{Current} \rightarrow F!Ack()] \]

[* F \neq \text{Current} \rightarrow \text{[Busy \rightarrow \text{ENTER}}

Authorized licensed use limited to: Yale University. Downloaded on February 23,2022 at 15:47:34 UTC from IEEE Xplore. Restrictions apply.
We note that in the above example we have guaranteed the following:

1) A user process can request service from the process Resource, only if it had gained access permission to it from the SJN process.
2) A user process can release the process Resource only if it has gained access permission to it (which was not yet released).
3) A user process cannot rerequest the resource without first releasing it.

The only mischief that can be done (in the above example) by a malfunctioning user process that will affect other processes is:
1) to acquire an access to the resource process, and refuse to release it afterwards,
2) to execute F!Acquire(T) without following it with F?Ack().

Circumstance 1) can be easily remedied using our proposed mechanism by preemptive revocation. Whenever the SJN process wishes to regain access to the previously allocated Resource process (because of some policy governing use of the resource process, such as time-outs, etc.), it simply changes the value of port S in the process Resource by executing R!Reset(). This will prevent the user process, that had previously acquired access to Resource, from accessing it.

Circumstance 2) cannot be handled without the introduction of new mechanisms. The difficulty stems from the fact that the user and SJN processes run asynchronously and therefore SJN may not be able to determine whether the user will eventually execute F?Ack() or not. In order to ensure that the process SJN will not be delayed forever on the output statement F!Ack(), we must enclose F!Ack() within a guarded command. If no matching occurs then the resource is released and SJN continues with its computation. The only form this guarded command may assume is:

```plaintext
[]¬Busy → Busy := true;
Current := F;
R!Set(F);
F!Ack();
]
[]F?Release() → [F ≠ Current → skip
 []F = Current → [Count > 0 → REMOVE
 []Count = 0 → Busy := false;
 R!Reset()]
]
[]F?Ack() → skip;
[]true → [Count > 0 → REMOVE
 []Count = 0 → Busy := false;
 R!Reset()]
]
```

The reason a "true" guard must be used is because CSP does not provide a mechanism for detecting whether a matching I/O command has been invoked.

The above proposed solution however has the undesirable effect that the user process may be delayed forever on F?Ack(). This may happen under two different circumstances.

1) The guard "true" may have been selected even though the user process is waiting on F?Ack() (guard selection is nondeterministic).
2) The guard "true" was selected because at the time the guards were evaluated F?Ack() has not been invoked yet.

Difficulty 1) could be easily resolved by introducing a deterministic control construct that will allow the programmer to specify the order in which the various guards should be evaluated [an alternative is to introduce a mechanism for detecting whether a matching I/O command has been invoked; although this would resolve difficulty 1), it does not, however, resolve difficulty 2)].

Difficulty 2) could be resolved by allowing a user of a port (say A) to invoke several I/O commands (associated with A) within a single statement, as in:

```
A(!Acquire(T); ?Ac(k));
```

The interpretation is that these I/O commands are executed in sequence but information concerning these commands is registered simultaneously so that no time dependent errors may occur. We note that the abstract implementation of the port concept introduced in [2] could be extended in a simple and efficient manner to handle this new construct.

IV. DISCUSSION

In the previous section we have presented a safe and efficient Shortest-Job-Next resource allocation scheme. For sake of brevity we did not include any other examples. We note, however, that our proposed mechanisms were successfully applied to such scheduling problems as the disk head optimizer, readers and writers, alarm clock, etc. We have found that our simple mechanisms are sufficiently powerful to solve these types of problems in an effective manner. For example, in solving the reader/writer problem we needed to allow several processes to gain access to the resource (file) in parallel. This can easily be accomplished in our scheme by explicitly storing the identity (ID) of all these processes in the resource (data structure) and by checking whether the "calling" process ID is in this list. This differs from our implementation of the Resource process where our port selection mechanism ensured that only the process which has gained access permission can indeed use the resource.

We have thus shown that with simple extensions to the port concept, CSP could be effectively utilized in the construction of safe and efficient dynamic resource management schemes.
CORRESPONDENCE

Weighted Processor Sharing—Results for Hyperexponential Servers

MICHAEL J. FERGUSON

Abstract—In a recent paper by Fayolle, Mitran, and Iasnogorodski [2], some general multidimensional integral equations were derived in order to solve for the mean response time of each of several classes in a queue whose service discipline was weighted processor sharing. The arriving processes were Poisson. The weighting means that each job within a class k is given an amount of processing proportional to the priority weight gk associated with that class. For exponential service times, the general equations were solved. In this note, a simple observation allows use of the exponential solution directly for the case of hyperexponential servers. As a result, it is possible to state the following.

* Characterization of a server in terms of its mean and coefficient of variation is not sufficient to predict even the mean response time for a class using weighted processor sharing. In unweighted or egalitarian processor sharing, only the mean is sufficient.

I. INTRODUCTION

Processor sharing is the name given to those job scheduling strategies where each job present at the service center is receiving service simultaneously, with individual rates depending on the system state. Unweighted or egalitarian processor sharing [1], [4], [5], [9] divides the processing cycles evenly among the jobs. Thus, if there are N jobs in the system, and the processor operates at C cycles/s, then each job will receive C/N cycles/s. The jobs may belong to different classes with different Poisson streams and service distributions. Kleinrock generalized the strategy to include weights gk so that a job of class k, under the condition that there were Ni jobs in class i for (j = 1, M), would receive service at a rate

\[ r(N_1, \ldots, N_M) = \frac{Cg_k}{\sum_{j=1}^{M} g_j N_j} \]  

(1.1)

These disciplines are not purely academic. It is possible to realize, given the knowledge of the class service distributions, any response time vector by choosing the appropriate gs.

In the recent paper by Fayolle et al. [2], a solution to the weighted processor sharing discipline was given. In the course of this solution, it was shown that the previous solution to the problem by Kleinrock [5] and O’Donovan [7] was incorrect. The solution was given in the form of a multidimensional integral equation that must be solved for the response time vector. Although, in principle, this set of equations could be solved for any set of rational service distributions, explicit solutions were given only in the exponential case. From the form of the equations, it was noted that, in general, the mean response time for weighted processor sharing depends on the entire service distribution and not just the first moments. In addition, they noted [2, (4.14)] that the response times satisfied

\[ \sum_{K=1}^{M} \rho_K W_K = \frac{1}{1 - \rho} \sum_{j=1}^{M} \frac{\lambda_j}{\mu_j} \]  

(1.2)

They justified the equation by noting that weighted processor sharing is work conserving and does not use any information about individual service times. This correspondence notes that the argument is insufficient. For hyperexponential service distributions, the weighted processor sharing discipline is work conserving, and also does not use any information about the individual service times, but the left-hand side of (1.2) is not a constant. However, for exponential service distributions and for any work conserving queue discipline, preemptive or not, Gelenbe and Mitran [3] showed that the left-hand side of (1.2) is a constant and that the right-hand side is the queuing time for a FCFS queue.

In this note, it is shown that, with respect to the response time process of a job in a queue with weighted processor shar-