Massachusetts Institute of Technology
Laboratory for Computer Science

Computation Structures Group Memo 148-2

Concurrent Programming

by

Randal E. Bryant
Jack B. Dennis

This research was supported by the National Science Foundation under grant DCR75-04060 and by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract number N00014-75-C-06661.

Revised June 1978

CONCURRENT PROGRAMMING[1]

R. E. Bryant
J. B. Dennis
Massachusetts Institute of Technology

1. Introduction

Concurrency of activities has long been recognized as an important feature in many computer systems. These systems allow concurrent operations for a number of reasons of which three are particularly common. First, by executing several jobs simultaneously, multiprogramming and time-sharing systems can make fuller use of the computing resources. Second, real-time transaction systems, such as airline reservation and point-of-sale terminal systems, allow a number of users to access a single database concurrently and to obtain responses in real-time. Finally, high speed parallel computers such as array processors dedicate a number of processors to the execution of a single program to speed up completion of a computation.

In developing the software for some of the early multiprogramming systems, programmers soon discovered a need for an abstract and machine-independent means of expressing the behavior of systems which involve concurrent activities. They found that machine level programming was tedious and very difficult to do correctly. When many tasks are to proceed concurrently, the problems of allocating system resources, of scheduling the order in which tasks are performed, and of preventing concurrent activities from disasterously interfering with one another are difficult to deal with without assistance from a high level programming language.

One of the first concepts to emerge in an attempt to satisfy this need for a more abstract view of concurrent systems was the process concept. In this view, the sequence of actions performed during execution of a sequential program is viewed as an abstract entitity called a *process*, and details such as which physical processor is used and the time of execution are ignored. For example, in a typical multiprogramming system the different user jobs, the interrupt routines, and the I/O channel program executions may be viewed as separate processes. During system operation, the processors and memory may be switched among the processes, so all processes are carried forward, even though no process retains

1. This research was supported by the National Science Foundation under grant DCR75-04060 and by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract number N00014-75-C-0666l.

exclusive control of all the resources it needs nor runs in one continuous sequence.

Traditional, high level programming languages such as Fortran, Algol, and Cobol express computations as independent, noninteracting processes. The processes in a concurrent system, however, may interact with each other for several reasons. First, one process may convey data to another. This is called *communication*. Second, processes may interact to ensure a correct sequencing of events. Process interaction which serves to control the order in which processes execute is called *synchronization*. These synchronization operations may be required for several different reasons, of which two are particularly common. First, if one process must perform some task before a second can proceed, there is a *precedence constraint* between the two processes. For example, the second process may need data which is computed by the first. Conversely, if one process produces data to be used by another, then the producer process cannot produce more data than the buffer between them can hold until the consumer process has used some of the old data. Hence, precedence constraints can exist in both directions between producers and consumers of data. Second, processes which share common resources such as processors, memory locations, or input/output devices require synchronization so the resources will be allocated in a systematic way. This allocation may be a simple form of *mutual exclusion*, in which a process retains exclusive control of a resource until the process voluntarily releases it, at which time the resource is granted to any process waiting for the resource. More complex allocation schemes can involve such features as allowing several processes to use a resource simultaneously, assigning different priorities to processes contending for a resource, or allowing one process to forcibly remove a resource from the control of some other process. Traditional programming languages are not powerful enough to express these types of interactions. Instead, a program must invoke operating system routines to perform the necessary communication and synchronization with other processes in the system.

Besides the inability to express the interactions between processes, traditional, high level languages cannot express *nondeterminate* computations. That is, they can only express computations whose output values depend only on the values of inputs. In a nondeterminate computation, on the other hand, output values can depend on other factors, such as the times at which events occur in the system. For example, suppose agents at two different remote terminals of an airline reservation system both request the last seat on the same flight. One will be granted this seat and one will not, but which one receives which response depends on the relative order in which the requests are received and processed. Nondeterminacy is essential in many concurrent systems.

The need for high level programming languages which can express the operation of a system of concurrent processes has led to the development of programming constructs with which one can express these communication and synchronization operations. Some of these approaches, such as semaphores [13] and monitors [4,5,20], suppose systems in which all processes have access to a single, shared memory. Others assume that processes communicate by sending messages to one another [2,21,24]. Languages based on actor semantics [16,17,18] carry the message-passing concept even further by considering all primitive operations to be carried out by separate, message-passing processes. Other approaches to concurrent programming have been developed which, instead of viewing a system as a number of communicating, sequential processes, view a program as an unordered set of instructions and permit an instruction to be executed any time its operands are ready. This form of program execution can potentially achieve a higher degree of concurrency than is possible with sequential processes. Languages based on this approach are called data flow languages [1,10,12,25,30].

Several issues must be considered when designing programming languages to support concurrent computation. Of primary importance is expressive power. The expressive power of a language, in the context of concurrent systems, means the forms of concurrent operations, and the types of communication, synchronization, and nondeterminacy which can be expressed in the language. A language which lacks expressive power will force the programmer to rely on a suitable set of operating system routines to implement desired behaviors. A properly designed language, on the other hand, should have sufficient richness to express these functions directly. Furthermore, if the language lacks expressive power, a programmer may need to resort to awkward or inefficient programming techniques to achieve desired results.

A second issue in the design of a language for concurrent programming is the clarity of programs written in the language, that is, how easily the effect of executing a program can be understood by looking at the program. A properly designed language can provide a programmer with the tools needed to write clear and concise programs. To meet this goal, the language must allow programs to be written in a modular fashion, so that the sections of the program can be viewed independently of one another. This property is critical in concurrent system design, since the sections of the programs which are executed concurrently can often affect each other in subtle ways, and these effects can ultimately lead to deadlocks, hazards, or other forms of incorrect behavior. Furthermore, these effects may cause problems only under relatively rare combinations of circumstances, and as a result the errors may remain undetected even after a long period of system operation. Hence, a modular program in which it is quite clear how the concurrent activities in the system can affect each other would be of great value to the programmer, and to anyone who wishes to modify the program later. A programming language can also help the

programmer write clear and concise programs by providing high level constructs to express the synchronization, communication, and nondeterminacy within the system. This will not only make programming less tedious, it will reduce the chance of error and make the programs more readable. If concurrent programming languages are to describe the operation of large and complex systems, it is important for these languages to have a clarifying rather than an obscuring effect on the programs.

Ultimately, one must be concerned with implementation issues. These include the ease of implementation of the language- whether it can be implemented on existing computer systems, whether slight modifications to an existing system will be sufficient, or whether it will require a whole new approach to computer design. A second factor in implementation is its efficiency, that is whether concurrency expressed in programs can be exploited without undue overhead. This desire for a language which is easy to implement, yet runs efficiently, often seems in conflict with the goals of expressive power and clarity of programs, and these two goals can themselves conflict with each other. Inevitably, trade-offs must be made, and hence the decision of which approach to use depends to a large degree on design priorities.

In this chapter, the main approaches to constructing concurrent programs will be presented and compared. As a basis for comparison, two examples of systems incorporating concurrent operations have been chosen, and programs for these examples will be presented using the different approaches to concurrent programming. Of particular interest are the semantic issues in language design, i.e. how the computation is expressed, rather than the detailed syntax of the languages. Hence, in the interest of uniformity, the example programs will be written in PASCAL [22], modified to include the necessary constructs. As shall be seen, the different approaches to concurrent programming differ greatly in their expressive power, clarity of expression, and ease and efficiency of implementation.

2. Example Systems

Two examples have been chosen as representative of systems for which concurrent programming is required. The first is an airline reservation system, in which a number of users (agents) can perform transactions interactively with a single database. In such a system concurrency in processing transactions is required to enable sharing of data, reasonable throughput, and real-time, interactive use. The second example is an input/output buffer system in which several input devices can read different files and send these files, via a buffer, to any of several output devices. By allowing the input and output devices to operate concurrently, this system can utilize hardware resources more effectively than would be possible otherwise.

The examples have been chosen to convey the basic features of concurrent systems. They have been simplified considerably to avoid the large amounts of detail typically required in real-life systems. For example, neither system has any form of error-checking, nor is there any provision for terminating system operation. Of course, it is difficult to draw conclusions about the merits of programming language features on the basis of such simple examples. In considering these programs, one must also consider how difficult it would be to add more sophistication to the system designs.

The database for the airline reservation system contains information about the flights for a single airline. Initially, each flight has 100 seats available. The system can accept two types of commands. To reserve seats on a flight, an agent gives the command ('*reserve*', *f*, *n*). If at least *n* seats are available on flight *f*, the seats will be reserved, and the system will respond with the message (**true**). If that many seats are not available, no seats will be reserved, and the system will respond with the message (**false**). To find out how many seats are available on flight *f*, a system user gives the command ('*info*', *f*). The system will respond with the number of seats which are available on the flight at the time the command is processed.

The input/output buffer system contains input devices *input*1, *input*2, ..., *input j*, output devices *output*1, *output*2, ..., *output k*, and a single buffer. During operation, the input devices read their respective blocks of data concurrently. Once a block has been read in, it is loaded into the buffer, at which time the input device can begin reading a new block. The block in the buffer is then moved to the local storage of one of the output devices and written out. Each output device is capable of writing any of the output blocks, hence a block in the buffer can be transferred to the first available output device rather than to a particular, predetermined one. The buffer can hold only one block at a time, hence the readers must contend with each other for use of the buffer. Similarly, each block is to be written out by only one output device, hence the output devices must contend for the output blocks. It is assumed that the buffering operations (i.e. moving a block from the input device to the buffer and from the buffer to the output device) are much faster than the input and output operations, so the buffer will not form a bottleneck in the system.

3. Processes Executing Within a Global Environment

The earliest organized approach to concurrent programming was to view a system as a number of sequential processes which execute concurrently in a common, global environment. This view is a natural abstraction of the operation of a multiprogramming system, which typically contains one or more central processing units and several input/output processors, all of which can access a single, shared memory. The processors communicate with one another by

reading or writing mutually agreed upon memory locations according to some convention. Thus, we can view execution of a set of instructions by a processor as an abstract process and the common memory locations as the global environment for these processes. Assignment statements with global variables on either the left or the right-hand side express the communication between processes.

Some mechanism is required to synchronize accesses to the global variables. In practice this is done using the program interrupt facility of the hardware. Examples of abstract synchronization mechanisms include the semaphores of Dijkstra [13] and the monitors of Brinch Hansen [4,5] and Hoare [20]. Other synchronization mechanisms have been developed [8,28], but none have received as much attention as semaphores and monitors. A semaphore is a special type of shared variable upon which several primitive synchronization operations can be performed. A monitor, on the other hand, is a set of programmer-defined procedures which can be called by the processes to gain access to global variables.

3.1. Process Synchronization by Semaphores

A semaphore S is an integer variable initialized to some value. Associated with the semaphore is a queue which holds names of processes. Two operations are defined on the semaphore: **wait**(S) and **signal**(S), (Dijkstra called these **P** and **V**, respectively.) If a process P executes **wait**(S), then the value of S is decremented. If this new value is negative, the name of P is placed on the queue associated with the semaphore, and P is blocked from executing. If, on the other hand, S is nonnegative, P is allowed to continue. If a process P executes **signal**(S), then the value of S is incremented. If the new value of S is less than or equal to zero, then the name of one process is removed from the queue, and this process is allowed to resume execution.

Semaphores provide a means to suspend execution of a process until certain conditions are satisfied. If processes perform semaphore operations in conjunction with their accesses of the global variables, the necessary synchronization in the system can be achieved. For example, a semaphore with initial value 1 can be used to maintain mutual exclusion of processes accessing a shared variable. A process which is updating the database in the airline reservation system, for instance, must have exclusive control of the database so that the database will remain in a consistent state during each transaction. Hence, to reserve n seats on flight f, a process would execute the following code segment:

```
        .
        .
        .
wait(mutex);
if available[f] ≥ n
  then
    begin
      available[f] := available[f] - n;
      success := true
    end
  else
    success := false;
signal(mutex);
        .
        .
        .
```

In the above program, *mutex* is a semaphore with initial value 1, and the array *available* is a global variable which represents the shared database.

If several processes wish to access the database without changing the database's state, these accesses can proceed concurrently. Furthermore, if a process wants to read only one word in the database, there is no danger of finding the database in an inconsistent state, hence this access can proceed even while other processes are updating the database. To find out how many seats are available on the flight, a process would simply execute the statement

$$n := available[flight].$$

Of course, in a more realistic airline reservation an agent would want to know more about a flight than the number of seats available. Hence, processing an '*info*' request would require reading several words of memory. If the database is altered in the middle of these reads, the information returned to the agent may contain inconsistencies. To program a more sophisticated reservation system, we would divide the types of transactions into two classes: those which only read the database (the readers), and those which alter the database (the writers.) A number of readers can proceed concurrently, but a writer must have exclusive control of the database. Programs which solve the readers-writers problem [9,17,20] are considerably more complex than our simple example.

Semaphores can also be used to control the order in which processes access resources. For example, the input and output processes in the input/output buffer system would execute codes as follows:

$Input_j$

```
while true do
begin
   read(inj, infilej);
   wait(free);
   buffer := inj;
   signal(loaded)
end
```

$Output_k$

```
while true do
begin
   wait(loaded);
   outk := buffer;
   signal(free);
   write(outk, outfilek)
end
```

Initial values: *free* = 1, *loaded* = 0.

In the above program, the global variable *buffer* serves as the buffer between the input and output processes. The semaphores *free* and *loaded* are used to maintain correct sequencing between input and output processes. Furthermore, the semaphore *free* is used to guarantee that only one input process can load a value into the buffer at a time, and the semaphore *loaded* guarantees that only one output process will print a particular buffer value. Thus, the two semaphores enforce both precedence constraints and mutual exclusion in the system.

The semaphore construct is sufficient to solve a wide variety of process synchronization problems, although sometimes with great difficulty. Two concepts which are found in many computer systems, however, are noticeably lacking. The first is the concept of the time at which events occur. For example, a process cannot pause for a specified amount of time before continuing execution. The second is that one process cannot force another process to stop execution. These two features were left out intentionally, since the process abstraction removes the time at which events actually occur in the system from the programmer's control, and a process can be affected by other processes only when it makes reference to the global environment.

One type of system whose operations cannot be fully expressed with a semaphore program is a system in which the processes do not execute within a single, global environment. If the system consists of processors connected together by a communication network [27], the processes execute within a number of local environments and hence cannot access global variables or semaphores. The notion of a global environment does not reflect the architecture of such a system. For example, in the airline reservation system, one cannot cause information to be transferred between the remote terminals and the central computer except by calling on the operating system to perform these operations.

The semaphore concept was a major step forward in making programs involving process synchronization easier to understand, but it still has several flaws as a programming tool. The first is the primitiveness of the semaphore operations. Semaphores provide a very simple form of process synchronization. It is left to the programmer to develop conventions about how semaphores will

be used to provide the desired behavior. Complex forms of process synchronization, in which different processes have different priorities, such as the various solutions to the reader's-writer's problem [9], typically have very obscure semaphore programs. Unless the conventions are carefully documented, the programs may be difficult to modify at a later date. Moreover, if just one process fails to obey the conventions as to how resources are to be accessed, the system may deadlock or in some other way behave improperly.

The second flaw is a total lack of modularity in the programs. Information about how a shared resource is utilized and how the synchronization is provided is distributed throughout the programs for the individual processes. For example, it is difficult to locate all sources of nondeterminacy in the system. The processes in the input/output buffer system would have the same programs if there were only one input process and one output process as it does with several input and several output processes. In the first case, the system is determinate, whereas it is not in the second. This lack of modularity, coupled with the primitiveness of semaphore operations, makes it very difficult for someone looking at a semaphore program to determine whether a resource is being accessed properly.

Regarding implementation, semaphores and their corresponding synchronization operations can be implemented without great difficulty on any system whose architecture reflects the idea of a global state, such as a multiprogramming system. The T.H.E. system of Dijkstra [14] is an example of a simple, but elegant operating system which uses semaphores to synchronize processes.

3.2. Process Synchronization by Monitors

Monitors were developed to allow a more structured format for concurrent programs than is possible with semaphores. Unlike semaphore programs, all information about a set of shared resources and how they are used is contained in a single area of the program: the declaration of a *monitor*. The declaration of a monitor includes a number of procedures which define operations on the shared resources. These procedures are available to all processes in the system. When a process wishes to access a shared resource, such as a global variable or a shared hardware resource, it must do so by executing one of the procedures of the corresponding monitor. It should be emphasized that a monitor does not itself cause any action in the system. Instead, it is merely a collection of procedures which can be executed by the processes in the system. This idea of limiting the ways in which a shared resource can be accessed to the operations performed by a small set of procedures was originally proposed in conjunction with *conditional critical sections* [19,3].

Monitors are implemented in such a way that the execution of the procedures of a particular monitor are mutually exclusive. Hence, a process retains exclusive control of the resources of a monitor while executing one of the monitor's procedures, until it surrenders its control. A process can surrender its control of the monitor in one of several ways. First, it can complete execution of the monitor procedure, at which time some other process can begin execution of one of the monitor's procedures. This form of control-passing is sufficient to implement mutual exclusion of processes. The airline reservation system, for example, utilizes only this form of control-passing. Other forms of control-passing are provide by *condition variables* along with the operations **delay** and **continue** (Hoare calls these **wait** and **signal**.) A condition variable has no visible value, although it does have an initially empty queue associated with it. When a process executes the statement **delay**(*cond*) in the body of a monitor procedure, the process' name is placed on the queue for *cond*, the process is blocked from executing further, and control of the monitor is released. When a process executes the statement **continue**(*cond*), this process is temporarily blocked (unless the queue for *cond* is empty), and one of the processes on the queue for *cond* is resumed. Once this reawakened process leaves the monitor procedure, the process which executed the **continue**(*cond*) statement is resumed.

In the airline reservation system, accesses to the database would be controlled by a monitor *database* with procedures *reserve* and *info* as follows:

```
monitor database;
var available: array [1..limit] of integer; i: integer;

procedure entry reserve (f, n: integer; success: boolean);
  begin
  if available [f] ≥ n
    then
      begin
        success := true;
        available [f] := available [f] - n;
      end
    else success := false
  end reserve;

procedure entry info (f, n: integer);
  begin n := available [f]
  end info;

begin
  for i = 1 to limit do available [i] := 100
end.
```

The monitor *database* controls all accesses to the array *available*, where *available* [*f*] is the number of seats available on flight *f*. During system operation, some process initializes the monitor by executing the statement **init**

database. This causes the body of the monitor program to be executed, setting all elements of *available* to 100. Then, to reserve n seats on flight f, a process executes the statement

database.reserve(f, n, *success*),

and to find out how many seats are available, it executes

database.info(f, n).

For the input/output buffer system, the buffer would be controlled by a monitor *I/O_buffer* with procedures *deliver* and *retrieve* as follows:

```
monitor I/O_buffer;
var buffer: block;  inuse: boolean;  free, loaded: condition;

procedure entry deliver(in: block);
  begin
    if inuse then delay(free);
    buffer := in;
    inuse := true;
    continue(loaded)
  end deliver;

procedure entry retrieve(out: block);
  begin
    if not inuse then delay(loaded);
    out := buffer;
    inuse := false;
    continue(free)
  end retrieve;

begin
  inuse := false
end.
```

During system operation some process must initialize the monitor by executing the statement **init** *I/O_buffer*. This causes the variable *inuse* to be set to **false**. Thereafter, the input and output processes execute programs as follows:

Input$_j$

```
while true do
begin
  read(inj, infilej);
  I/O_buffer.deliver(inj)
end
```

Output$_k$

```
while true do
begin
  I/O_buffer.retrieve(outk);
  write(outk, outfilek)
end
```

The expressive power of monitors is equivalent to that of semaphores in the sense that one can write a program for a monitor *semaphore*, with procedures *wait* and *signal* which models the behavior of a semaphore, and conversely one can write a semaphore program which models the behavior of a monitor.

However, if one wishes to follow the convention that a shared resource in the system can be accessed only by calling a procedure of the corresponding monitor, then all accesses to that resource must be mutually exclusive. For example, in the airline reservation system several processes cannot execute the procedure *database.info* concurrently. Only by relaxing the restrictions, so that the database could be accessed directly by the processes could the full concurrency in the system be realized. This, however, would compromise the goal of collecting together all information about how a resource is utilized into one section of the system specification.

The monitor construct provides more modularity than semaphores, and this yields more understandable programs. The ways in which a resource may be accessed are contained in a single section of the system specification, rather than in the programs for each process. This modularity also makes the system easier to modify. For example, if we wish to modify the input/output buffer system so that several blocks could be buffered at once, we need only modify the monitor procedures. The change would not affect the process programs.

The mutual exclusion of procedure calls, while it is a restriction in terms of expressive power, helps make monitor procedures easier to write than the equivalent semaphore programs. Monitor procedures are less susceptible to subtle timing errors than they would be if several processes could access the resources controlled by the monitor simultaneously. Perhaps a carefully designed extension to the monitor formalism could be developed which allows procedure calls to proceed concurrently under some circumstances, while retaining the modularity and clarity of the monitor concept.

As with semaphores, monitors can be implemented without major difficulties on a multiprogramming system. The Solo operating system of Brinch Hansen [6,7] is written mainly in Concurrent Pascal [5], an extended version of Pascal which supports monitors. The ability to write an operating system in a high level language, including the communication and synchronization between processes, is an important advance in concurrent programming.

4. Processes Communicating by Message Passing

In one more modular view of concurrent systems each process executes within a local environment that cannot be accessed or altered by any other process. For two processes to interact with each other, one process must send a message to the other, and the receiving process must accept the message. One of the first system designs which followed this approach was the Regnecentralen RC4000 computer system [2] in which the system contained a single CPU yet supported a number of independent, message-passing processes.

To illustrate how message-passing semantics might be supported by a programming language, we shall use a language extension in which a message is a triple (*destination*, *source*, *contents*), where *destination* is the name of the receiving process, *source* is the name of the sending process, and *contents* is the information which the message is to convey. Messages in this language are of type record. Thus, for example, the contents field of a message *m* is referenced by the expression *m.contents*. Execution of the command **send**(*m*) by process *P*, where *m* is of type message, will cause a message (*m.destination*, *P*, *m.contents*) to be sent to the process *m.destination*. Each process has a single input queue into which all incoming messages are placed. Execution of the function **receive** will first cause the process to wait until a message is placed in its input queue, if one is not already present. Then the first message is removed from the queue and returned as the value of the function.

These two message-passing operations are sufficient to solve the airline reservation system problem. Whereas in the global environment approach, the database is a global variable accessed by a number of different processes, with the message-passing approach we shall define a process *transact* which has sole access to the database. All transactions are initiated by sending messages to *transact*. The contents fields of these messages can have one of two formats:

('*reserve*', *flight*, *number*),

and

('*info*', *flight*)

The program for the process *transact* is as follows:

```
process transact;
var available: array[1..limit] of integer;
      request, reply: message; f, n: integer;

begin
for n := 1 to limit do available[n] := 100;
while true do
begin
  request := receive;
  case request.contents.type of
  'reserve': begin
               f := request.contents.flight;
               n := request.contents.number;
               if available[f] ≥ n
                 then
                   begin
                     reply.contents := true;
                     available[f] := available[f] - n;
                   end
                 else
                   reply.contents := false
             end;
  'info': reply.contents := available[request.contents.flight]
  end;
  reply.destination := request.source;
  send(reply)
end
end.
```

Notice that this program does not realize all potential concurrencies in the system. The database transactions are processed sequentially, much as they were in the monitor program, because the process *transact* has exclusive access to the database, and it is a sequential process.

For the I/O buffer example, we shall use a process *buffer_control* to control the buffering between input and output processes. An input process will send a message containing the input block to *buffer_control* which in turn will send this block to one of the output processes. Each output process must notify *buffer_control* when it is ready to receive a block, or else *buffer_control* would have no way of knowing what output processes are free. This can be accomplished by sending a '*ready*' message. Hence, the *contents* field of messages sent to *buffer_control* can have one of two formats:

('*data*', *inblock*),

and

('*ready*').

Unlike the processes in the airline reservation system, the process *buffer_control* cannot always service its input messages in the order received. For example, it may receive several '*ready*' messages before receiving any '*data*' messages. Hence, some means of storing messages in internal queues is required. For this

reason we will use a data type **queue** on which the operations **enqueue** and **dequeue** are defined, as well as the boolean-valued function **empty**. The program for *buffer_control* is as follows:

```
process buffer_control;
var dataq, readyq: queue of message;
      inputm, outputm, datam, readym: message;

begin
while true do
begin
  inputm := receive;
  case inputm.contents.type of
  'data': if empty(readyq) then enqueue(inputm, dataq)
          else
            begin
              readym := dequeue(readyq);
              outputm.contents := inputm.contents.inblock;
              outputm.destination := readym.source;
              send(outputm)
            end;
  'ready': if empty(dataq) then enqueue(inputm, readyq)
           else
             begin
               datam := dequeue(dataq);
               outputm.contents := datam.contents.inblock;
               outputm.destination := inputm.source;
               send(outputm)
             end
  end
end
end.
```

The input and output processes execute the following codes:

Input$_j$

```
begin
mj.destination := 'buffer_control';
mj.contents.type := 'data';
while true do
begin
  read(inj.infilej);
  mj.contents.inblock := inj;
  send(mj)
end
end.
```

Output$_k$

```
begin
mk.destination := 'buffer_control';
mk.contents.type := 'ready';
while true do
begin
  send(mk);
  outmk := receive;
  write(outmk.contents, outfilek)
end
end.
```

Note that in the above set of programs, there is no means of limiting the number of blocks buffered by *buffer_control*. If the input processes send blocks to *buffer_control* at a higher rate than *buffer_control* sends them to the output processes, the number of blocks stored in the queue *dataq* will grow without limit. In order to limit this buffering, additional control messages must be sent between the input processes and *buffer_control*. For example, an input process

may send a message '*ready_to_send*' to *buffer_control* which, when it had sufficient space, would reply '*send*'. Only when an input process receives permission would it send a block. Thus, message-passing can accomplish synchronization as well as communication between processes.

This view of processes as independent entities which can interact only by sending messages to one another is certainly more modular than the view of processes executing within a global environment. As a result, it is much clearer to the programmer exactly how the processes can affect one another. Furthermore, this view corresponds more closely to the way in which processes are implemented on a distributed computer system. For example, the program for the airline reservation system very naturally expresses the way in which such systems are implemented. In a typical system, remote, "intelligent" terminals assemble messages requesting operations on the database. These messages are then sent to a central computer, which performs the operations and sends back reply messages. Control messages such as the ones sent between processes in the input/output buffer system correspond closely to the control signalling between the components of a distributed system. When the programming language reflects the underlying system design, a programmer can understand more fully how the program will be executed and hence can design programs which run efficiently on the system. Both the modularity and the closeness to the implementation make this approach to concurrent programming attractive for many important applications

The message passing operations described so far are clearly too primitive for a high level programming language. Like semaphores, they provide only a simple form of process communication and synchronization, leaving the programmer to determine what types of processes are required, what types of control and data messages must be sent between processes, and at what points in the programs the messages should be sent.

More sophisticated languages have been proposed [21,24] which provide the programmer with a higher level view of the cooperation of message-passing processes. Whereas the illustrative language used for our examples requires a separate program for each process, a program written in either Kahn's [24] or Hoare's [21] language specifies the operation of a number of processes. A program is a set of coroutines, where each activation of a coroutine may be executed by a separate process. This approach provides a more concise view of the system and also eliminates some of the duplication in effort needed to write separate process programs. One can specify a set of similar computations as a "coroutine array," in which a set of processes execute the same coroutine program with different input parameters. Processes are dynamically created and terminated by invoking or completing execution of a coroutine. Kahn's language achieves an additional degree of semantic elegance by treating the sequence of messages sent from one process to another as a single data object

called a *stream*. He defines primitive operations on streams which are analogous to the commands **send** and **receive**. However, since the sequence of messages sent between each pair of processes is a separate stream, a process can decide which process to receive its next message from. This enables the programmer to limit the sources of nondeterminacy in the system. In fact, programs written in Kahn's language are inherently determinate. A process must decide in advance which stream to remove the next message from, hence the order in which messages arrive at a process has no effect on the outcome of the program. If the language were modified so that several processes could enter messages concurrently into a single stream, however, nondeterminate computations could be expressed. Both Hoare's and Kahn's languages are at very preliminary stages of development and implementation. More work will be required before these concepts are fully developed and become tools of programming practice.

Hewitt and Atkinson [17] have proposed a program structure called a *serializer* to provide a more structured and higher level view of concurrent programming in a message-passing environment. The purpose of the serializer construct is to provide the programmer with a general framework for resource controllers which is then customized to fit a particular application, much as the monitor construct provides a general framework for a resource controller operating in a global environment. In addition the serializer design tries to correct some of the weaknesses in monitors, such as the complexity of the operations **delay** and **continue**, and the limited amount of concurrency. The behavior of a serializer is defined in terms of the *actor* model of computation [15,16,18], a model in which message-passing is viewed as the fundamental operation. In this model every action is performed by an actor, where each actor behaves like a message-passing process. That is, it receives input messages, performs an operation on the input, generates output messages, and possibly changes its internal state. Unlike processes, however, actors can be dynamically created and abandoned. With this model a wide variety of activities can be expressed, such as concurrent operations, dynamic system creation and reconfiguration, and nondeterminacy. Furthermore, the actor model allows highly concurrent computations to be expressed more naturally than the sequential process model does, because the only sequencing constraints between actor activities are those imposed by the messages. This great expressive power of the actor model allows a serializer to have a much more sophisticated behavior than can be expressed in a programming language such as PASCAL extended with message-passing commands. Furthermore, since the designers of the serializer were not constrained by the limited types of behavior exhibited by sequential, message-passing processes, they could develop a cleaner structure with greater potential for concurrency. Serializers, as well as the actor model are still in an early stage of development. Their influence on future language design and programming practice remains to be seen.

Sequential processes that communicate by message passing can be implemented without great difficulty. The processes can be carried out by a number independent processors, such as one typically finds in a distributed computer system, or even by a more traditional multiprogramming system, such as the RC4000 computer system. By extending the RC4000 system with semaphore operations, Lauesen [26] was able to develop an operating system which is provably free of deadlocks. Few operating systems which use machine synchronization instructions can claim this achievement.

A system consisting of a small number of sequential, message-passing processes can achieve only a limited amount of concurrency, as was seen in the airline reservation example. Since a resource can be accessed by only one process, and this process operates sequentially, concurrent accesses to a single resource cannot be expressed. In some cases, a large resource can be partitioned into a number of parts, and each part managed by a separate process. For example, the information about each flight in the airline reservation system could be maintained by a separate process. However, if we want to add new flights to the database or remove old ones, some method of dynamically creating and abandoning processes is required. When the system is divided into many small parts which can be dynamically created and abandoned, it no longer seems justified to call these parts processes; rather they are more like actors. Exactly where the dividing line between the process model and the actor model lies is a matter of debate, as are many other issues in developing highly concurrent systems which operate in a message-passing environment.

5. Data Driven Program Execution

The programming languages discussed so far (with the exception of those based on actor semantics) have been based on the concept of communicating, sequential processes. That is, a system is viewed as a number of processes which can proceed concurrently, but within each process only one action is performed at a time. Programming languages designed to express the behavior of these systems are similar to traditional languages, with constructs added to express process communication and synchronization. An alternative to sequential processes is to view a program as an unordered set of instructions, each of which defines how a set of values is to be computed and what identifier is to be associated with each value. Within an environment, an identifier must refer to a unique value. Rather than executing in strict, sequential order, instructions can be executed as soon as their input operands are ready, i.e. as soon as the values required to compute the expressions have themselves been computed. This form of program execution is said to be *data driven*, since the arrival of the operands, rather than the indication of a program counter determines when an instruction will be executed. Languages which express programs for data driven execution are often called data flow languages [1,10,12,25,30].

To express an unambiguous computation, instructions in a data flow language must be side-effect free. That is, the effect of executing an instruction can only be to compute a set of values for a set of identifiers. It cannot alter the definition of any other identifier in the program. Furthermore, the program must obey the "single assignment rule", meaning that each identifier is defined only once within an environment. Considering the importance of side-effects and multiple assignments to variables in traditional programming languages, one naturally wonders how a language could eliminate both of these properties and yet be able to express useful computations. Data flow languages can make up for these restrictions with recursive procedures and with *data streams* [23,30]. Recursion eliminates the need for iteration, a control structure which relies heavily on side effects and multiple assignments. Streams allow the programmer to view a sequence of elementary data values as a single entity. Thus, by writing a procedure that accepts inputs that are data streams, one can express program units which perform operations on entire sequences of input values. Procedures which have streams as inputs and return streams as results will be called *modules* to differentiate them from procedures which operate on individual data values. For the airline reservation system example, we shall define a module *transact* with inputs *request_stream*: **stream of message** and *available*: **array** [1..*limit*] **of integer**, which will compute an output *reply_stream*: **stream of message**. That is, the module will receive a sequence of requests from the remote terminals and an initial state of the data base, and it will produce a sequence of replies.

To make use of streams, we must define some operations on them. To extract the values from a stream s, we define two functions: **first**(s) which returns the first value in the sequence, and **rest**(s) which returns the stream consisting of all elements in s except for the first one. To construct a stream, we define a function **cons** where the value of **cons**(x, s) is the stream consisting of x (which cannot be a stream) followed by the elements of stream s. Furthermore, we must define a rule for procedure invocation in data flow. In the earlier definitions for data flow languages [10], a procedure $P(x, y, z)$ cannot be invoked until all input arguments x, y, and z are ready. With streams, however, this rule is modified somewhat. If, for example, x is a **stream**, then P could be invoked as soon as the first element of stream x is ready. Hence the module *transact* can be invoked as soon as the first request has arrived.

With a few modifications to the PASCAL syntax, we can arrive at a language which is suitable for expressing data flow programs. Most importantly, to emphasize the idea that an instruction is a definition of how a set of values is to be computed, assignment statements

<*id*>:= <*exp*>

will be replaced by identifier definitions

let <*id*> = <*exp*>.

Furthermore, a side-effect free analogy to "updating" the array *available* is

required. We will define the function modify(*A*, *i*, *v*) which returns an array which is identical to *A*, except that the *i*th element is equal to *v*. Despite the syntactic similarities, however, the semantics of the data flow language are entirely different from PASCAL. In particular, the order in which statements are listed does not dictate the order in which they are executed.

The program for *transact* is as follows:

```
module transact(request_stream: stream of message;
          available: array[..limit] of integer);

returns reply_stream: stream of message;

var request, reply: message; f, n: integer;
            newstate: array[1..limit] of integer;

begin
  let request = first(request_stream);
  case request.contents.type of
  'reserve': begin
                let f = request.contents.flight;
                let n = request.contents.number;
                if available[f] ≥ n
                  then
                    begin
                      let reply.contents = true;
                      let newstate =
                          modify(available, f, available[f]-n)
                    end
                  else
                    begin
                      let reply.contents = false;
                      let newstate = available
                    end
             end;
  'info': begin
            let f = request.contents.flight;
            let reply.contents = available[f];
            let newstate = available
          end
  end;
  let reply.destination = request.source;
  let reply_stream =
        cons(reply, transact(rest(request_stream, newstate)))
end.
```

The module *transact* receives its input requests in the form of a single stream. This stream is composed of elements produced by a number of separate modules that transmit request messages from agent terminals. So far, no means for generating such a stream has been discussed. In fact, the data flow language which has been presented can express only determinate computations: the result

of program execution depends only on the values of the inputs, and not on the order in which they are received. The airline reservation system, however, behaves nondeterminately, hence some means expressing nondeterminate operations in the language is required. For this purpose, we will define a primitive operation **merge**, where the value of **merge**(*s1*, *s2*) is a stream containing all elements of streams *s1* and *s2*, such that the ordering of elements from *s1* is preserved, as is the ordering of elements from *s2*, but the order in which an element from *s1* and an element from *s2* occur is arbitrary. This operation is sufficient to express a wide variety of nondeterminate computations. For example, suppose the airline reservation system contains three terminal modules which produce streams *requests1*, *requests2*, and *requests3*. We can write the program which computes the three output streams as follows:

```
module system(request1, request2, request3: stream of message;
                        available: array[1..limit] of integer);

returns replies1, replies2, replies3: stream of message;

begin
   let r1 = tag(request1, 1);
   let r2 = tag(request2, 2);
   let r3 = tag(request3, 3);
   let requests = merge(r1, merge(r2, r3));
   let replies = transact(requests, available);
   let replies1, replies2, replies3 = sort(replies)
end.
```

In this program the messages in the three streams of input requests are first tagged with the stream number. These three tagged streams are merged together into a single stream which serves as the input stream to *transact*. The output stream from *transact* is sorted according to the tag values into three streams of replies- one for each terminal module.

A data flow program for the input/output buffer system will not be given here, because it does not demonstrate any new concepts.

Data flow languages seem very promising for expressing computations for concurrent execution, since the only restrictions on the concurrency are those imposed by data dependencies. Although side-effects and identifier redefinition are excluded, the combination of recursive procedures and data streams yields a surprisingly rich language. Furthermore, the single, nondeterminate operator **merge** is sufficient to express numerous types of nondeterminate system behavior. Not enough experience has been gained, however, to fully evaluate the expressive power of the language. Suggestions for extensions have been made [1], for example, which allow communication links between modules to be created dynamically. Just how necessary such a feature is, and how important other features may be, are open questions.

Data flow languages permit programs to be written which are far more modular than is possible with traditional languages. Each module of a program can be described fully in terms of its input/output behavior. Due to the absence of side effects, sections of the program can interact only in limited and well-defined ways. In fact, each instruction executes in its own local environment: it computes a result based only on its operands. This high degree of modularity leads to programs which more clearly describe what computations the system is to perform. In addition, data flow languages allow the programmer to explicitly limit the sources of nondeterminacy in the system. Nondeterminacy can occur only where it is explicitly allowed through the use of the **merge** operator. Considering that unwanted nondeterminacy is a major source of errors in concurrent systems, a means of controlling it is of great significance.

The implementation of data flow languages is currently at a rather primitive state. Due to the high degree of concurrency and the asynchronous nature of instruction execution, these languages may require totally new forms of computer architecture. Several designs have been proposed [11,29], but numerous problems remain to be solved before practical data flow machines can be realized. Hence the state of the art for data flow language design is well ahead of the state of the art for architectures which support these languages.

6. Conclusion

The three major approaches to concurrent programming discussed here differ greatly in their fundamental views of how a computer system operates. With the global environment approach, one views a system as a number of processes which execute "under one roof" and communicate with one another by altering the surrounding environment. With message-passing processes, one views a system as a number of processes which execute under their own roofs and send telegrams to one another. With data-driven program execution, the system is viewed as a network of operators, each of which receives data values, computes new data values, and sends these output values to the next operator in the network. Furthermore, this network dynamically expands as recursive procedures are invoked and contracts as they are completed. The three approaches differ in the amount of concurrency which they can achieve, the clarity of the programs, and the ease with which they can be implemented given the current state of computer system design.

No system composed of communicating, sequential processes can realize the full degree of concurrency latent in high level programs. However, a number of processes can often proceed concurrently. With semaphore-based programs, the number of active, concurrent processes is limited only by the cleverness of the programmer subject to the need to maintain a consistent global state. With monitor-based programs, one must choose between completely protecting each

resource with a monitor and hence precluding concurrent accesses to this resource, or allowing processes to access a resource directly, thereby compromising the modularity provided by the monitor concept. With message-passing processes a resource can be directly accessed by only a single process. Hence, unless the resource can be partitioned into a number of parts, each of which is managed by a separate process, concurrency in the system is restricted. In contrast to programming languages based on sequential processes, data flow languages and actor-based systems can express all forms of concurrency allowed by the algorithm, although no existing machine architectures can fully exploit their benefits.

Evaluating how clearly each approach can express the operations of a system is a subjective judgement. However, such features as modularity, limited sources of nondeterminacy, and high-level language constructs are clearly desirable goals. In terms of modularity, the approaches to concurrent programming have been presented in order of increasing modularity. First, a semaphore-based language allows little modularity- the processes can affect each other in numerous and often subtle ways. Next, monitors provide more modularity by restricting the ways in which each process can access global resources. Languages based on message-passing processes carry the modularity one step further by eliminating the global environment altogether. Finally, data flow languages, by eliminating all side effects achieve a degree of modularity in which each program module can be viewed as defining a function from input values to output values. As for limiting the sources of nondeterminacy, only Kahn's stream language and data flow languages provide means of stating explicitly where nondeterminacy is allowed in the system. Operations on semaphores, global variable accesses, monitor procedure calls, and message-passing, on the other hand, are all potential sources of nondeterminacy. When nondeterminacy is not wanted, the programmer must be careful to use these operations in a way which will not allow nondeterminate behavior.

With the exception of monitors and serializers, high-level language support for concurrent programming is largely nonexistent. With both semaphore-based systems and message-passing systems, the language constructs presented express very elementary forms of process communication and synchronization. The programmer must devise conventions for using these constructs to achieve the desired behavior. Data flow languages would also benefit from more sophisticated constructs. For example, a construct similar to a monitor has been proposed for data flow languages [1] which eliminates the need for the programmer to construct a tagged stream from several input streams and then to sort the output stream into its constituent parts. Designing high level programming tools which are sufficiently general and modular, yet do not restrict the concurrency exploitable in their implementation is one of the most difficult challenges to the designer of future high-level languages.

Given the current state of computer design, one has little choice of which programming approach to use if a practical implementation is required. Both approaches which assume a global environment fit most naturally on a multiprogramming system consisting of processors sharing memory. Such systems are common, and as a result a large proportion of the work in concurrent programming has been directed toward this global environment approach. Message-passing processes, on the other hand, describe most naturally the operation of a system of independent processors connected by communication channels. Such systems are becoming increasingly common, due largely to a desire to distribute the processors geographically, and also to the availability of small, low-priced processors. Most programming of these systems is still done at the machine language level. No machine-independent languages for message-passing processes have come into accepted use. Finally, languages which express higher degrees of concurrency than can be achieved by communicating, sequential processes, such as actor-based and data flow languages have not yet been implemented to take advantage of this greater concurrency. Whereas the other approaches could be implemented by modifying existing machine designs, these high concurrency languages appear to require totally new approaches to computer design if the latent concurrency is to be realized. While the design of languages for concurrent programming is an interesting field of study in its own right, a language is of little use unless it can be effectively implemented. Hence, the design of computer systems to support languages which express high degrees of concurrency is also an important field of study.

References

1. Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, Monitors, and Dataflow," Proceedings of the Sixth ACM Symposium on Operating Systems Principles, Operating Systems Review, 11-5, ACM, New York (November, 1977), pp. 159-169.
2. Brinch Hansen, P., "The Nucleus of a Multiprogramming System," Communications of the ACM, 13-4, ACM, New York (April, 1970), pp. 238-241,250.
3. Brinch Hansen, P., "Structured Multiprogramming," Communications of the ACM, 15-7, ACM, New York (July, 1972), pp. 574-578.
4. Brinch Hansen, P. Operating System Principles, Prentice Hall, Englewood Cliffs, N. J. (July, 1973).
5. Brinch Hansen, P. "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering, SEI-2, IEEE, New York (June, 1975), pp. 199-207.
6. Brinch Hansen, P., "The Solo Operating System: a Concurrent Pascal Program," Software-Practice and Experience, 6-2, John Wiley and Sons, New York (April-June, 1976), pp. 141-150.

7. Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall, Englewood Cliffs, N. J. (July, 1977).
8. Campbell, and Habermann, "The Specification of Process Synchronization of Path Expressions," Lecture Notes in Computer Science, Volume 16, Springer Verlag, New York (1974).
9. Courtois, P. J., F. Heymans, and D. L. Parnas, "Concurrent Control with Readers and Writers," Communications of the ACM, 14-10, ACM, New York (October, 1971), pp. 667-668.
10. Dennis, J. B., "First Version of a Data Flow Procedure Language," Programming Symposium: Proceedings, Colleque sur la Programmation, B. Robinet, Ed., Lecture Notes in Computer Science 19, G. Goos and J. Hartmanis, eds., Springer-Verlag, New York (1974), pp. 362-376.
11. Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York (January, 1975), pp. 126-132.
12. Dennis, J. B., "A Language Design for Structured Concurrency," Computation Structures Group Note 28-1, MIT Laboratory for Computer Science, Cambridge, Mass. (February, 1977).
13. Dijkstra, E., "Co-operating Sequential Processes", Programming Languages, ed. F. Genuys, Academic Press, New York (1968), pp. 43-112.
14. Dijkstra, E., "The Structure of the THE Multiprogramming System," Communications of the ACM, 11-5, ACM, New York (May, 1968), pp. 341-346.
15. Goodman, N., Coordination of Parallel Processes in the Actor Model of Computation, Technical Report TR-173, MIT Laboratory for Computer Science, Cambridge, Mass. (December, 1976).
16. Greif, I., Semantics of Communicating Parallel Processes, Technical Report TR-154, MIT Laboratory for Computer Science, Cambridge, Mass. (September, 1975).
17. Hewitt, C., and R. Atkinson, "Parallelism and Synchronization in Actor Systems," Principles of Programming Languages, ACM, New York (January, 1977), pp. 267-280.
18. Hewitt, C., and H. Baker, "Laws for Communicating Parallel Processes," Information Processing 77, IFIP, North Holland Publishing Company, Amsterdam (1977), pp. 987-992.
19. Hoare, C. A. R., "Towards a Theory of Parallel Programming," Operating Systems Techniques, ed. C. A. R. Hoare, Academic Press, New York (1972).
20. Hoare, C. A. R., "Monitors: an Operating System Structuring," Communications of the ACM, 17-10, ACM, New York (October, 1974), pp. 549-557.
21. Hoare, C. A. R., Communicating Sequential Processes, Internal Memo, Queen's University, Belfast (April, 1976).

22. Jensen, K., and N. Wirth, PASCAL: User Manual and Report, 2nd ed., Springer Verlag, New York (1974).
23. Kahn, G., "Semantics of a Simple Language for Parallel Programming," Information Processing 74, IFIP, North Holland Publishing Company, Amsterdam, (1974), pp.471-475.
24. Kahn, G., and D. MacQueen, Coroutines and Networks of Parallel Processes, Information Processing 77, IFIP, North Holland Publishing Company, Amsterdam (1977), pp. 993-998.
25. Kessels, J. L. W., "A Conceptual Framework for a Nonprocedural Programming Language," Communications of the ACM, 20-12, ACM, New York (December, 1977), pp. 906-913.
26. Lauesen, S., "A Large Semaphore Based Operating System," Communications of the ACM, 18-7, ACM, New York (July, 1975), pp. 377-389.
27. Metcalfe, R. M., Packet Communication, Technical Report TR-114, MIT Laboratory for Computer Science, Cambridge, Mass. (December, 1973).
28. Reed, D., Processor Multiplexing in a Layered Operating System, Technical Report TR-164, MIT Laboratory for Computer Science, Cambridge, Mass. (June, 1976).
29. Rumbaugh, J., "A Data Flow Multiprocessor," IEEE Transactions on Computers, C26-2, IEEE, New York (February, 1977), pp. 138-146.
30. Weng, K., Stream-Oriented Computations in Recursive Data Flow Schemas, Technical Memo TM-68, MIT Laboratory for Computer Science, Cambridge, Mass. (October, 1975).