Communicating Sequential Processes for Centralized and Distributed Operating System Design

M. ELIZABETH C. HULL and R. M. McKEAG
The Queen's University of Belfast

This paper demonstrates how the notation of Communicating Sequential Processes may be used in the design of an operating system. It goes further to show how such an approach assists in the design and development of a system distributed over a network of computers. The technique uses a well-defined design methodology.

Categories and Subject Descriptors: C.2.4 [**Computer-Communications Networks**]: Distributed Systems—*network operating systems*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Techniques**]: Language Constructs—*concurrent programming languages*; *Pascal-plus*; D.4.0 [**Operating Systems**]: General; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *scheduling*; *synchronization*

General Terms: Design, Languages

Additional Key Words and Phrases: Operating systems, parallel programming, program design, communicating sequential processes, distributed systems

1. INTRODUCTION

A widely accepted view is that the function of an operating system is to provide a virtual machine within which a user can run his own programs. The development of the THE multiprogramming system [3] laid the foundation for our present understanding of operating systems, that is, that an operating system should be structured as a series of levels, each of which would provide a virtual machine for the higher levels. P. Brinch Hansen [1] and C. A. R. Hoare [4] have both shown that Pascal can be extended with processes, monitors, and abstraction mechanisms (classes or envelopes) to provide a structured programming language suitable for computer operating systems. These languages [1, 6, 8] have been used for the construction of operating systems.

A rather different type of language involves processes that communicate with one another directly, thereby avoiding the need for monitors through which the processes communicate. Several language proposals have been based on the notion of directly communicating processes [2, 5]. It seems reasonable to choose

Authors' addresses: M.E.C. Hull, School of Computer Science, Ulster Polytechnic, Shore Road, Newtownabbey, Co Antrim BT37 0QB, Northern Ireland; R.M. McKeag, Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland.

such a notation to represent a multiprogramming system and to use it to investigate the design of an operating system as a set of communicating processes. The most influential of these is the Communicating Sequential Processes (CSPs) notation [5], developed by Hoare, and it is this notation that we choose to use as a design tool in this paper. This notation has been proposed as a means of representing a system as a set of processes executing in parallel and communicating only by means of input/output operations.

We give consideration, in this paper, to how such a notation may be used in the design of an operating system, and we go further to consider the construction of an operating system for a network of computers. Various multicomputer operating systems have indeed been programmed, but little in the way of general rules for their design has been published. In this paper we consider some of the problems of distributing an operating system and how they may be overcome in a reasonably methodical manner.

2. AN OPERATING SYSTEM

We shall take as an example the conventional single-computer operating system described by Welsh and McKeag [7]. This owes a lot to the THE system, and its structure is not unlike that of many other operating systems. In outline it takes the following form, expressed in Pascal-plus [6]:

```
program operating system;
  monitor module processor;
  ...;
  monitor module mainstore;
  ...;
  monitor module typewriter;
  ...;
  monitor module filestore;
  ...;
  monitor module cardreader;
  ...;
  monitor module lineprinter;
  ...;
  process userprocess;
    procedure runuserjob;
      {declare instances of the virtual resources needed—these, together, constitute the
      virtual machine in which the user's job is run}
      begin
        :
      end;
    begin
      while {system switched on} do runuserjob
    end;
  instance user: array[1 .. maxuser] of userprocess;
  begin
    {no global variables to initialize}
  end.
```

The principal component of the operating system is the "userprocess", for which we declare "maxuser" instances, each of which will run a succession of jobs for the users of the system. To run a user's job, various resources are required, and to administer the resources of each type we program a Pascal-plus

monitor. Thus there is one monitor to administer the "filestore", another to administer the "typewriters", and so on.

In general, the structure of a typical monitor to administer the resources of some type is as follows:

```
monitor module resourcetype;
  const
    resourcemax = {resources of this type};
  type
    *status = ( *success, *failure);
     resourcenum = 1 .. resourcemax {to identify resource};
  monitor module resourcescheduler;
    ...;
    procedure *acquire(...);
    ...;
    procedure *release(...);
    ...;
  monitor controller (r: resourcenum);
    ...;
    procedure *operation(...);
    ...;
    process module resourcehandler;
    ...;
  instance
    resourcecontroller: array[resourcenum] of controller ((1) (2) (3) ... (resourcemax));
  envelope *virtualresource;
    var * result: status {records result of the last operation on the resource};
          resource: resourcenum {records identity of real resource being used};
    ...;
    procedure *virtualoperation;
    ...;
  begin
    .
    .
    .
  end  {resourcetype};
```

The monitor contains some constant and type definitions pertaining to the resources and a "resourcescheduler" monitor, with a procedure to "acquire" a resource from a pool of available resources, and one to "release" a resource back to the pool.

As well as scheduling the use of the resources, we need to control their use. In Pascal-plus we express this, using a monitor "controller" of which we declare as many instances as there are resources. Each such monitor contains variables to buffer input/output and for recording results of data transfers. A small process "resourcehandler" initiates each data transfer, waits for the completion interrupt, and checks for and reports data transfer failures.

The only other component of the monitor is a Pascal-plus ***envelope***, which is effectively a definition of a type representing a virtual resource. Instances of this type are declared by processes needing to use such a resource, thus:

```
instance R: resourcetype.virtualresource
```

The only visible aspects of this virtual resource are the variable "*R*.result" and the procedure "*R*.virtualoperation"; the envelope instance "*R*", the scheduler, and the appropriate controller hide everything else: the identity of the resource, the delay until it is available, the details of data transfers, the interrupts, the fault handling, and so on.

To consider a typical example, a virtual card reader should provide its user with a procedure to read a card. The interface is therefore quite simple, complicated only by the need to provide for a "status" check after reading a card, that is, success, failure, and end of file detected.

```
envelope *reader;
  var
    cr:crnum {records identity of real card reader being used};
    *result: status {records result of the last operation on the card reader};
  procedure *read (var c:card);
    begin
      {read a card from card reader "cr" and record "result"}
       crcontroller [cr].read(c, result)
    end;
  begin
    crscheduler.acquire (cr); result := success;
    ***
    {the eventual return of the card reader to the scheduler's pool is left to the "crhandler"
     process in the controller monitor}
  end;
```

There may be minor variations on this theme; for example, the "filestore" monitor may offer its users several envelopes, such as "sequentialfile" or "randomaccessfile", and it may have more than one scheduler: one to schedule the use of the sectors of the disc and the other to schedule access to the disc for data transfers. But, in general, for each type of resource, there is a scheduler, there is a controller (containing a small device handling process) for each actual resource, and there is a virtual resource envelope. Thus a complete operating system will contain several such "resourcetype" monitors and a number of "userprocesses"; it may also have a number of small service processes, for example, to log the jobs that have been run or to provide the operators with statistics of resource usage, but such service processes need not concern us here.

3. FEATURES OF CSP FOR OPERATING SYSTEM DESIGN

Before designing an operating system, the structures required of, and available in, the CSP notation must be considered. Hoare's attempt to unify all the different program structures into processes seems acceptable for processes and monitors. The representation of abstraction mechanisms seems to be more difficult, however.

One easily recognizes that an operating system, unlike the normal application program, involves a degree of parallelism, its purpose being to share the resources that it controls among a number of users who make unpredictable demands upon these resources. We have already shown that any operating system requires, for each type of device, a scheduler, a controller for each device of that type, and a way of providing the user with virtual resources.

In CSP input and output provide the sole method of communication between processes running in parallel. They do not communicate with each other by updating global variables. A process may communicate with another process which it names—if the latter is subscripted, then it communicates with the specified element of that process array. The one-to-one type of communication is quite straightforward

```
singleuser :: [. . . ; resource ! message( ); . . .]
//
resource :: [. . . ; singleuser ? message( ); . . .]
```

However, in the one-to-many type of communication shown below, the "resource" process provides a service to any member of an array of "user" processes, the point being that in CSP this element must be named; this is quite different from Pascal-plus where communication is through global data and the identity of one process is not known to another.

```
resource :: [. . . ; (u: 1 . . usermax) user(u) ? message( ); . . .]
//
user(u : 1 . . usermax) :: [. . . ; resource ! message( ); . . .]
```

This can easily be extended to an *n*-to-*m* type of communication, where we have a specific element of "resource" communicating with a specific element of "user".

```
resource (r: 1 . . resourcemax) :: [. . . ; (u: 1 . . usermax)user(u) ? message( ); . . . ]
//
user (u: 1. .usermax) :: [. . . ; resource(r) ! message( ); . . .]
```

Often, the "user" process is not interested in a *particular* element of "resource", but rather in *any* element. It therefore would "acquire" such an element, that is, choose *r*. This approach is therefore convenient when considering the representation of envelopes as a user can now communicate with a "virtual process".

This, for *notational purposes only*, can be represented by the following:

```
virtualresource =df [. . . ; ? message( ); . . . ]
//
user (u: 1 . . usermax) :: [resource :: virtualresource
                            // . . . ;
                            resource ! message ( );
                            . . .
                           ]
```

However, in CSP we are not provided with the facility of defining process types or dynamically declaring instances of them. We must therefore find some way of representing this in CSP. In fact, we declare instances to some maximum of the "virtualresource" process required and then use them as necessary through a scheduling process. In CSP, for communication to succeed between two processes, both tags and value lists must correspond in the input/output commands. That is, in CSP, each process must name the other in order to communicate, while in Pascal-plus an instance of an envelope is unaware of the identity of the process invoking it—a CSP program therefore makes explicit what a Pascal-plus compiler must deduce from the program.

3.1 Scheduling

Consider the problem of scheduling a single resource between *n* users. In CSP this could be trivially expressed as

```
scheduler :: * [(i: 1 . . n) user(i) ? acquire( ) →
                                  user(i) ? release( )
               ]
```

Table I

Pascal-plus	CSP

acquire —
  if pool = [ ] **then** WAIT;
  “take item from pool”
	acquire —
	  [pool = [ ] → WAIT
	  □ pool ⟨ ⟩ [ ] → “take item from pool”
	  ]

release — “put item into pool”	**release** — “put item into pool”; [queue = [ ] → skip □ queue ⟨ ⟩ [ ] → SIGNAL; “take item from pool”]

Note that the user identification enforced by CSP automatically ensures that a release signal is accepted only from the user currently using the resource.

A single process may communicate with any element of an array of processes. Consequently, Hoare's monitor may be regarded as a single process that communicates with one or more elements of an array of user processes. However, if more than one device is available, then a more general solution for the scheduler is required—the availability of the devices being represented by a set.

```
scheduler ::
  [free: set of resource; free := [1. .R];
    * [free ⟨ ⟩[ ]; (i:1. .n) user(i) ? acquire( ) →
              r: resource; r := 1
              * [not (r in free) → r := r + 1];
              free := free − [r];
              user(i) ! acquire(r)
     □ (i:1· ·n) user(i) ? release(r: resource) →
                            free := free + [r]
     ]
  ]
  //
  user (i:1· · n ) ::
  [r: resource;
    . . . ; scheduler ! acquire( ); scheduler ? acquire(r);
      . . . ; scheduler ! release(r); . . .
  ]
```

When considering scheduling, an algorithm, for example, “first come, first served”, should be provided so that processes waiting to use the resource may be queued. In the above solution, the scheduler repeatedly inputs from any of the n users, not necessarily conforming to a particular algorithm.

In CSP we must, therefore, introduce some form of explicit queuing if that is what is desired for the particular application. That is, in CSP we must explicitly queue the processes that partake in, for example, a first-come, first-served algorithm, rather than rely on the built-in synchronization techniques of Pascal-plus.

Let us consider the fundamental difference in style in programming a scheduler in Pascal-plus and CSP. Table I shows the acquisition and release of a resource in skeletal form.

WAIT occupies the same position in both, as does "put item into pool". Because in CSP the scheduler has a life of its own, the requesting process must be treated as passive data when it is queued. No resumption address can be readily stored in the queue, and so it has to resume being processed in the **release** procedure, not in the **acquire** procedure; therefore we have the inevitable duplication of "take item from pool".

In the design of an operating system a more general technique is required when considering the whole problem of queuing. We cannot rely on the underlying "run-time" support for such algorithms as scheduling the movement of a disc head. We must therefore define a range of queue types that can take into account the fact that each process can be on only one queue at any moment. Any process (normally a scheduler) wishing to use such a queue declares an instance of it and performs the required operations. Using the process array declaration of CSP, we can define a queue element for each process that could require queue suspension.

```
discscheduler ::
[ readqueue(direction: (up, down)) :: priorityqueue
                                   {two processes to represent priority queues}
//
  writequeue :: fifoqueue {process to represent FIFO queue}
//
[{code of "discscheduler" process with such commands as readqueue (up) ! wait
                                                          (cylinder number)}
]
]
```

3.2 Virtual Resources

A monitor is an envelope that guarantees mutual exclusion. Since a monitor has already been represented by a process, an envelope can also be represented by a process. Following the ideas of operating system design, we wish the user to be unaware of an actual resource, but rather to be aware of some corresponding virtual resource.

A user declares "myresource" to be of type "virtualresource". In the example given below, the function of "virtualresource" is perhaps rather unnecessary apart from hiding the identity of the resource. However, it demonstrates the structure required for a more realistic example.

```
virtualresource =df
  [r: resource;
     scheduler ! acquire( ); scheduler ? acquire(r);
     * [? transfer( ) → {use the resource r}];
     scheduler ! release(r)
  ]
//
user (i: 1. .n) :: [myresource :: virtualresource
                  // * [. . . ; myresource ! transfer ( ); . . .]
                  ]
```

Such a reference to "myresource" can only be made within the scope of the declaration of the "user" process. The input command "? transfer( )" accepts an input command to use the resource from its (anonymous) creator. Note that the "scheduler" process now accepts input commands from any of the "virtualresource" processes, not directly from the users. As stated earlier, we declare instances to some maximum "vmax", say, of the process type required, and then use them as necessary through a scheduling process.

```
virtualresource (v:1··vmax) ::
  [r: resource;
   scheduler ! acquire( ); scheduler ? acquire(r);
   *[(i:1··n) user (i) ? transfer( ) →
                    {use resource r}];
   scheduler ! release(r)
  ]
//
virtualscheduler ::
  [{some straightforward scheduler}]
//
user (i:1··n) ::
  [myresource: 1··vmax;
   virtualscheduler ! acquire( );
   virtualscheduler ? acquire(myresource);
   * [. . . ; virtualresource(myresource) ! transfer( ) . . .];
   virtualscheduler ! release (myresource)
  ]
```

Because of the structure of a CSP process, the "virtualresource" process, unlike the envelope construct, can govern the ordering of operations performed on that resource. This is particularly useful when writing a process to administer such resources as sequential files.

3.3 Controllers

The need for explicit controller processes in such a system is at first sight a debatable one—one can argue that, for reasons of abstraction and readability, the inclusion of a process, one for each resource, is necessary to control the use of the device

```
controller(r: resource) ::
   *[(v:1. .vmax) virtualresource(v) ? usedevice( ) →
      {perform transfer on device and await
       interrupt}
   ]
```

But there is also merit in the argument that the structure of CSP is such that the functions of the envelope and controller can be collapsed into one process—in our case the "virtualresource". However the virtual resource process has a temporary existence (in principle, at least), and the resource generally needs a permanent controller process to look after it, even when it is unallocated—to field its interrupts, to report its failures and to wait for them to be corrected, to hide its idiosyncrasies from the world, and so on. These tasks cannot be performed by the succession of virtual resource processes to which it is allocated.

One saving is made, however. In Pascal-plus we have one monitor and, local to it, one process, for each resource; in CSP we can amalgamate these into a single process.

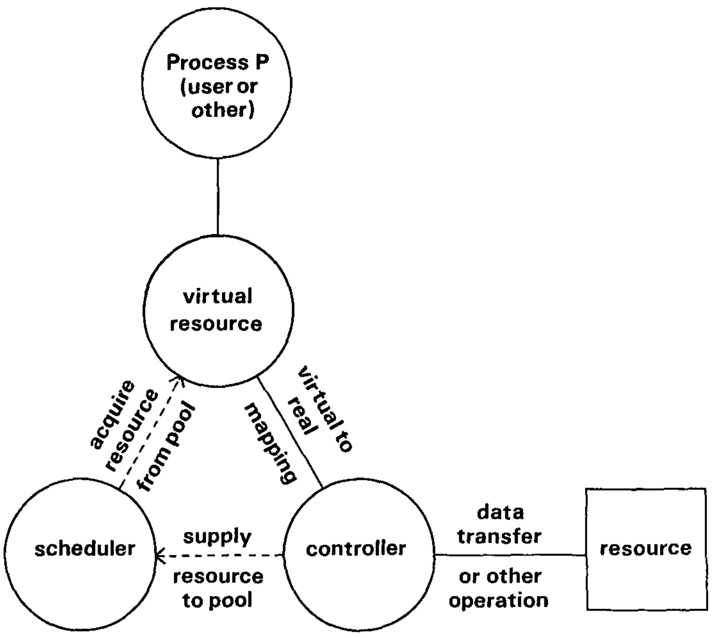

Fig. 1. Resource administration.

4. A NETWORK

Increasingly, there has been a trend toward the interconnection of several computers to produce a system as powerful as a mainframe, but at a reduced cost. This, of course, is only one advantage. It also provides the facility to expand the system, coupled with the attraction of a reliable one.

The operating system we examined in the earlier part of this paper was developed for a single computer with one or more processors and a common store. The language that we used there, with its monitors (which are effectively shared variables, together with the mutually exclusive operations that can be performed upon them), is ideally suited for a common store. Indeed in Pascal-plus the only way in which processes may communicate with one another is through monitors. It follows that if we are to implement an operating system on a network of computers without any common store it would be helpful to drop the use of monitors.

It would appear that the software notation already described in this paper is ideally suited to a distributed operating system, principally owing to the concept of no shared data. This section, therefore, investigates what changes to the process structure are necessary if we choose to implement the system on a network of computers and to investigate the suitability of CSP for this new system.

As far as the structure of resource administration is concerned, we have seen that in both approaches a process *P* wishing to use a resource declares an instance of a "virtual resource" process, which acquires a free resource from a scheduler process. This can be represented diagrammatically as shown in Figure 1. Following our technique in the previous section of this paper, we shall replace each monitor by a process and permit processes to communicate directly with each other. This was one of the reasons that led Hoare to develop his CPS notation. Any monitor already containing a process, chiefly each resource controller monitor with its resource handler process, can be merged with its local process.

Turning therefore to CSP, we may describe our system as follows:

```
operatingsystem ::
  [resourcetypeR :: [{declaration of process for a particular
                       resource type}
                    ]
  //
  resourcetypeS :: [. . .]
  //
  resourcetypeT :: [. . .]
  //
  .
  .
  .
  //
  user(u: 1. .maxuser) :: [{declaration of user process}]
  ]
```

Each "resourcetype" process takes the following form:

```
resourcetypeJ ::
  [resourceschedulerJ :: [{declaration of scheduler process}]
  //
  resourcecontrollerJ(j: 1. .jmax) ::
                          [{declaration of an array of controller
                            processes}
                          ]
  //
  virtualresourceJ =df [{definition of virtual resource}]
  ]
```

As a result of this transformation we have a number of "user" processes and, for each type of resource, typically a "scheduler" process, a "controller" process for each resource of the type, and some number of instances of a "virtual resource" process.

Just as in Pascal-plus, so in CSP we can add some structuring to provide a better understanding of the design. In the former solution we were able to "wrap up" the processes, monitors, etc. for each resource type within a single monitor, and so in CSP we can nest the processes to produce a similar structuring effect.

In a network, we must consider the distribution of these processes. It suffices here to consider a simple configuration of a main computer running user programs and a satellite computer. Both computers support peripheral devices and are connected by a full duplex link. Such a design can be modified to permit more generality of use.

4.1 Administration of the Link

For the administration of the link two processes are required in each computer. Obviously the two in the main computer deal with traffic to and from the satellite, while the two in the satellite deal with traffic in the reverse directions. The sets

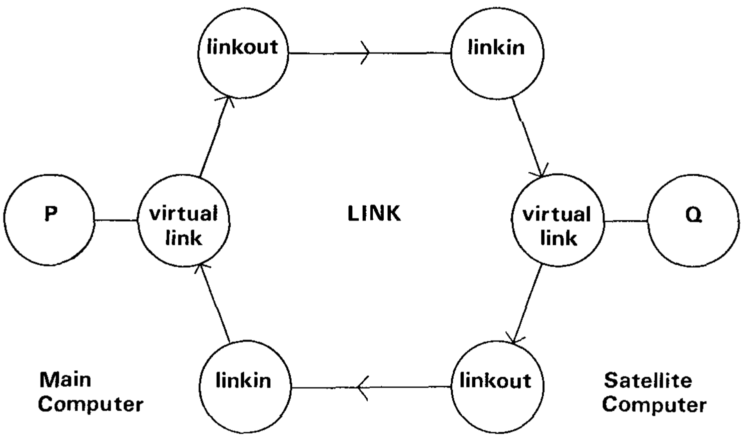

Fig. 2. The link processes.

of processes are identical and so we place a copy of each in each computer. These processes have the following skeletal structure, if we assume that they communicate with any process *P*:

```
tothelink :: * [P ? message ( ) → {send along link}]
fromthelink :: * [{accept message from link} → P ! message( )]
```

Since successive transmissions along the link may not be for the same device nor indeed for the same process, it is necessary to identify the destination of such transmissions. Furthermore, the structure of these processes should be intuitively trivial (i.e., the link processes should not be interested in which peripheral, function, etc. they are administering), and this suggests that our approach should be rather different.

To this end, we treat the link as a resource like any other and supply each user process (or each process wishing to use the link) with a "virtual link". The actual handling of the outgoing channel and incoming channel that constitute the link is carried out by two processes in each computer, "linkout" and "linkin". It may also be necessary to introduce a process to schedule the use of the outgoing channel, especially if some messages have priority over others. (See Figure 2.)

Since many virtual links will be mapped onto the real link we need some protocol to match up a "virtual link" process in one computer with the corresponding "virtual link" process in the other. We therefore introduce a coding system, that is, we associate a unique code with each of the virtual links established at any moment and this code will accompany every signal or message transmitted over the real link; thus the receiving computer's "linkin" process can match the code accompanying the transmission with that quoted by the "virtual link" process expecting the message or signal.

This system can best be explained by an example. To reduce the amount of traffic along the link and to avoid having to buffer an arbitrary amount of information in the receiving computer, the protocol we choose is that the driving force behind the transmission of a message will be the process that is to receive the message. Let *P* be a process in the main computer and let it transmit to *Q* in the satellite computer. *Q*, via its "virtual link," transmits a *request* signal to the

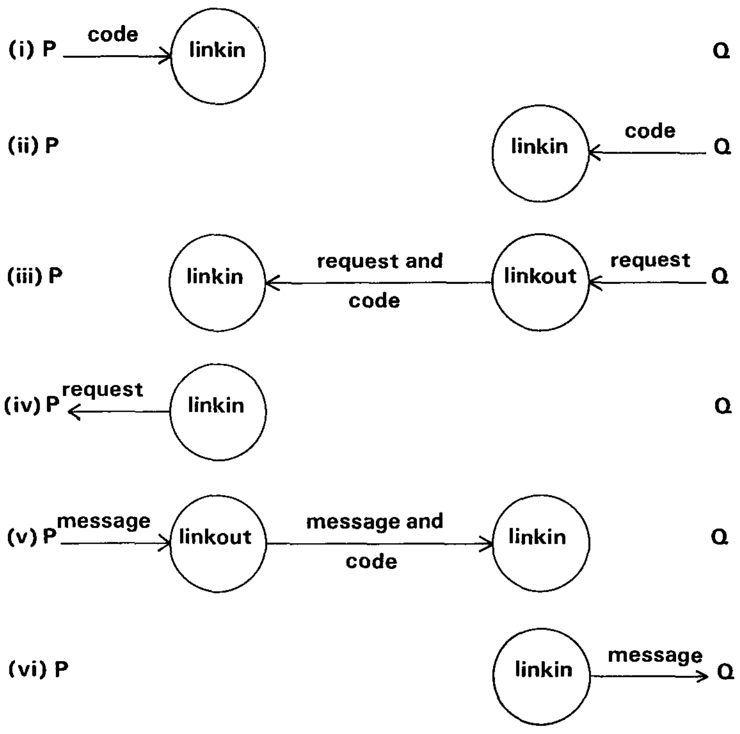

Fig. 3. The link protocol.

"linkin" process in the main computer and *P*, via its "virtual link," also communicates with this "linkin" process. Both *P* and *Q* will have supplied codes, which are used by "linkin" to match *P* and *Q*. *P* then transmits the message, accompanied by the virtual link code, to the satellite computer's "linkin" process and *Q* also communicates with that process, again quoting the code. The "linkin" process can then direct the message to *Q* and so avoid having to buffer it. (See Figure 3. Note that as we ensure step (ii) precedes step (iii) we are not concerned with timing constraints.) The only buffering required is in the "linkin" process, which stores codes rather than the messages, and this leads to greater economy and efficiency.

A "user" process wishing to use the link, therefore, does so in the following way:

```
user ::
  [message: array [1..n] of char;
   mylink :: virtuallink (code: codetype)
   // . . .
   mylink ! transmitting (message);
   :
   mylink ! request to receive ( );
   mylink ? receiving(message);
   :
  ]
```

The code may refer to a single process or to a family of processes.

4.2 Distributing the Processes

In a distributed system a user may wish to be able to use resources on the local computer and on the satellite computer. Two points are immediately evident:

(i) The link should be invisible to the "user" process.
(ii) The "controller" process for any resource must obviously be placed in the computer to which that particular resource is attached.

Let us suppose that the resource that user *P* wishes to use is not local but a peripheral on the satellite computer. By using (i) and (ii) above the "virtual resource" and "controller", processes are placed as shown in the main computer and the satellite computer, respectively.

The position of the "scheduler" process is perhaps less obvious. As regards the amount of communication with other processes, it could really go in either machine. However, we should like the "scheduler" process to be unaware of the link, and, further, if the satellite computer serves processes running on itself, or indeed on a third computer, it seems logical to place the "scheduler" into the same machine as the resources it schedules; if the resources are physically distributed, this solution is of course not possible.

We have said that the "scheduler" should remain unaltered and unaware of the link; this criterion should also apply to the "controller" process. This, in turn implies that the "virtual resource" process should be in the satellite computer. On the other hand, we argued above that it should be placed with the "user" process, enabling it to hide the link and the protocols regarding its use from the user. So it should be placed in the main computer. The solution is to split the functions of the "virtual resource" process and to make it, alone, aware of the link. This is in line with the function of any virtual resource process, which is to hide any necessary housekeeping.

In the main computer we therefore have a "virtual resource" process, which will accept requests from a process *P* and code them into messages for transmission along the link. In the satellite we have a "shadow virtual resource" process, which receives messages from the link, decodes them, and passes them on to the "scheduler" and "controller" processes. (See Figure 4.)

4.3 Distributed Schedulers

At any moment a great many virtual resources of a particular type may be required although there is only a limited number of real resources of that type. All but this limited number of virtual resource processes are waiting in the scheduler: Thus the scheduler must be able to queue many requests that will, in general, come from a variety of computers in the network, and, corresponding to each nonlocal request, there will be (in the scheduling computer) a "shadow virtual resource" process and a "virtual link" process, both of which will be inactive most of the time.

In small systems this will not matter, but if the scheduling computer *Y* is serving not just one computer *X* but many, the overhead may be unnecessarily large. In such circumstances there may be a case for the scheduler in computer *Y* to limit itself to queuing a limited number of requests from each computer, *X*, say two, and from these it selects the most deserving request for service. We thus need to introduce into each computer *X* a "shadow scheduler" process to filter the requests before they are passed to the main scheduler. (See Figure 5.)

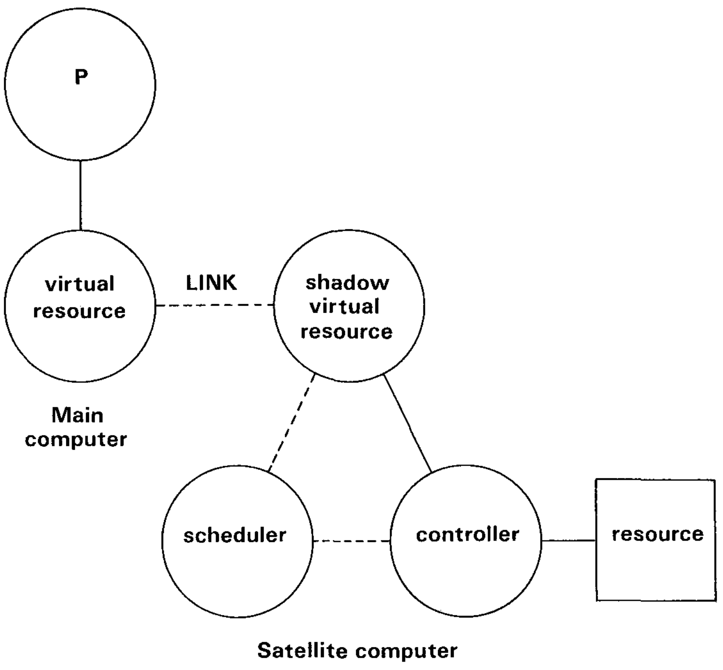

Fig. 4. Splitting the virtual resource.

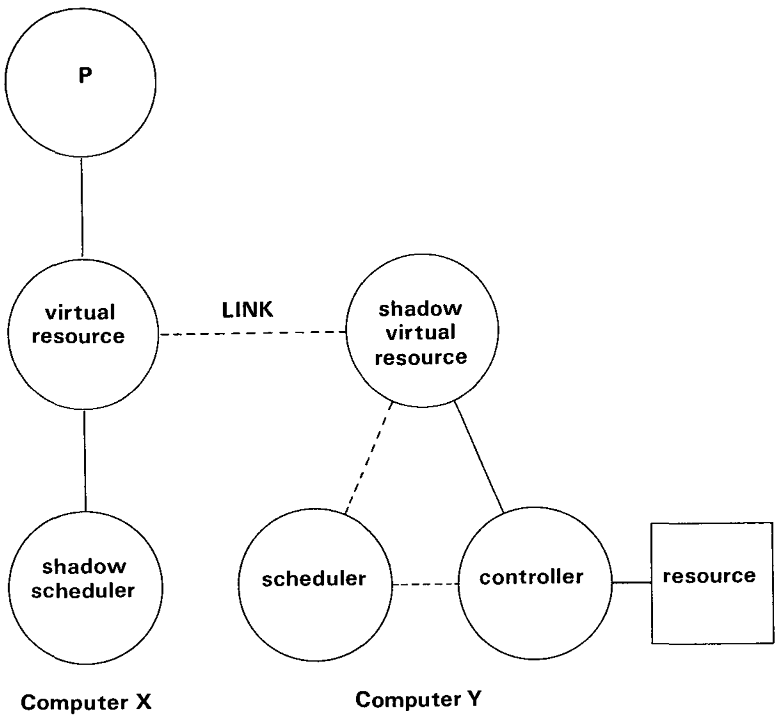

Fig. 5. Splitting the scheduler.

We need to consider carefully what the policy of each "shadow scheduler" should be. If the main scheduler operates a strict priority policy, the shadow can do likewise. If one desires to operate a first-come, first-served policy, it is not enough to ensure that the main scheduler and its shadows implement that policy independently; rather, each should operate a priority policy where each request uses as its priority the real time at which it is passed to its shadow scheduler. It is much more difficult to distribute a scheduler that operates the elevator algorithm for a moving head disc; such a policy requires the scheduler to serve the request for the nearest cylinder in the current direction of travel. Although the main scheduler can operate such a policy, the shadow schedulers do not in general know which cylinder is currently under the read/write heads, nor do they necessarily know the current direction of travel. Thus, if an excessive amount of interprocess communication is to be avoided, the best approximation is for the shadow schedulers to operate a first-come, first-served algorithm; in practice this should be perfectly acceptable.

The principal reason for introducing shadow schedulers is to ensure that the number of housekeeping processes in the computer housing the main scheduler is reasonably independent of the numbers of processes in the other computers.

4.4 General Structure

Let us consider a process in the main computer wishing to use a resource S in the satellite computer, as well as a local resource M in the main computer. As far as resource S is concerned, we see that a "satellite virtual resource" process must be declared in the main computer, and a "shadow virtual resource" in the satellite computer.

A certain uniformity therefore appears if each computer allows administration of both *local and remote* resources. Each computer therefore requires the following processes for administration of a particular resource type:

(i) local virtual resource;
(ii) remote virtual resource;
(iii) shadow virtual resource;
(iv) local resource controller (one per resource);
(v) local resource scheduler;
(vi) shadow resource scheduler (to perform scheduling algorithm).

This uniformity eases the "what goes where" decisions necessary for a distributed system.

The structure in the main computer for a resource M being used locally and S remotely is as follows:

```
maincomputer ::
  [resourcetypeM ::
    [resourceschedulerM :: [...]
    //
     resourcecontrollerM (m:1..mmax) :: [...]
    //
     localvirtualresourceM =df [...]
    ]
  //
resourcetypes ::
  [remotevirtualresource$S$ $=_{df}$ [{uses link to communicate with "shadowvirtual-
                                        resource$S$"}]
  //
   shadowresourcescheduler$S$ :: [...]
   ]
//
link ::
   [linkout :: [...]
  //linkin :: [...]
  //virtuallink $=_{df}$ [...]
   ]
//
   user :: [{declares instances of "virtuallink",
            "localvirtualresource$M$" and
            "remotevirtualresource$S$"}
           ]
]
```

and the structure in the satellite computer is as follows:

```
satellitecomputer ::
  [resourcetype$S$ ::
    [resourcescheduler$S$ :: [...]
    //
     resourcecontroller$S$ ($s$: 1..$s$max) :: [...]
    //
     shadowvirtualresource$S$ $=_{df}$ [{communicates with "remotevirtualresource$S$" across
                                        the link}]
    ]
  //
  link ::
    [linkout :: [...]
   //linkin :: [...]
   //virtuallink $=_{df}$ [...]
    ]
  ]
```

5. CONCLUSION

The purpose of this paper was to investigate how an operating system could be designed using a highly parallel notation. It must be stressed that CSP was used here as a design tool for a structuring method and was not intended as an implementation language. We have shown how an operating system can be clearly defined as a hierarchy of communicating sequential processes in a methodical way. The only programming construct in CSP is the process. We have discussed ways in which the well-understood structures of envelopes, monitors, and processes in Pascal-plus can be represented by processes in CSP without loss of clarity and with little difficulty. We believe that using an abstract notation such as CSP is indeed an attractive technique and it has been shown that clear and correct programs result.

This paper also investigated how a distributed configuration would benefit from an operating system designed in this way. If we consider a system where each process is running on its own processor, and where one process can communicate with another process by name, then it is easily seen that such a design approach is indeed attractive. Generally, we would not have such a configuration. Rather, we would have several computers linked to form a network in a predefined way. In this paper we have shown how and where processes should be placed in such a system and what extra processes must be provided. Furthermore, we have shown how links between the computers are used in the same way as any other resource, and we have devised protocols for communication to take place between processes in one machine and those in another machine.

ACKNOWLEDGMENT

The authors wish to thank the referees of an earlier version of this paper for their very helpful comments.

REFERENCES

1. BRINCH HANSEN, P. The programming language concurrent Pascal. *IEEE Trans. Softw. Eng. SE-1*, 2 (June 1975), 199–207.
2. BRINCH HANSEN, P. Distributed processes: A concurrent programming concept. *Commun. ACM 21*, 11 (Nov. 1978), 934–941.
3. DIJKSTRA, E.W. The structure of the THE multiprogramming system. *Commun. ACM 11*, 5 (May 1968), 341–346.
4. HOARE, C.A.R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.
5. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
6. WELSH, J., AND BUSTARD, D.W. Pascal-plus—Another language for modular multiprogramming. *Softw. Pract. Exper. 9* (1979), 947–957.
7. WELSH, J., AND MCKEAG, R.M. *Structured System Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1980.
8. WIRTH, N. MODULA: A programming language for modular multiprogramming. *Softw. Pract. Exper. 7* (1977), 3–35.

Received April 1982; revised February, July 1983, accepted August 1983