A System Organization for Resource Allocation

D. M. Dahm*, F. H. Gerbstadt, and M. M. Pacelli
General Electric Company, Phoenix, Arizona

This paper introduces a system for resource management using the concepts of "process," "facility," and "event." Except for the processor no attempt has been made to give serious suggestions for the policy to be followed for resource allocation. However, a basic framework is provided in which a system analyst can express solutions to resource management problems.

The paper is divided into a tutorial presentation, a description of the system primitives, and a small collection of examples of the use of the primitives.

Introduction

Modern day operating systems are concerned with the management of resources, i.e., basic units such as processors, I/O channels, peripheral units and memory. The management of resources normally includes the allocation of resources to processes and the releasing of resources by processes. Additionally, resource management may employ a request queuing mechanism to handle peak load periods in which mean time between requests is less than mean service time.

In this paper a system for resource management is introduced that uses the concepts of "process," "facility," and "event." Except for the processor, we do not attempt to give serious suggestions for the policy to be followed for resource allocation. However, we feel that we provide a basic framework in which a system analyst can express solutions to resource management problems.

The paper is divided into a tutorial presentation, a description of the system primitives, and a small collection of examples of the use of the primitives. The tutorial presentation is intended to elucidate and provide motivation for the second part.

Some of the ideas presented here were suggested by SOL [4, 5].

1. Tutorial Presentation

1.1 Intuitive Notion of a Process

In a multiuser environment, we think of the computation ordinarily associated with a user as being a line of control, or a sequence of instructions executed for the user. We regard this sequence to be synchronous, that is to say, nonparallel. Of course, more than one line of control may be associated with a particular user; however, we wish to focus our attention upon the idea of a single line of control. We call this line of control a "process." The basic intuitive idea is the same as the intuitive notion of process given in [1]. A process is, of course, a set of states and executing a sequence of instructions for a process is a rule describing the changing of states. See, for example, [2].

Characteristically, a processor will execute a sequence of several instructions for a process until its attention is diverted (by an interrupt or a fault, for example). At some later time the processor will return and continue executing instructions for that process. During the time that the attention of the processor was diverted from the process the processor may have been executing instructions for another process.

A process may specify an action to be performed upon the occurrence of some event, e.g., by using the ON statement of PL/I. While the action is being performed, the "main program" must be suspended; that is to say, we want no parallelism between the "main program" and the action. If any case exists where parallelism is desirable, then we regard the action as a separate process.

An example of the behavior in time of a processor in a multiuser system is shown in Figure 1. For $t < t_1$ the processor is executing process A, for $t_1 < t < t_2$ the processor is executing process B, and for $t_2 < t$ the processor is executing process A again. At $t = t_1$ the system has had to cause a change of control from A to B. Similarly, at $t = t_2$ the system has changed control from B back to A.

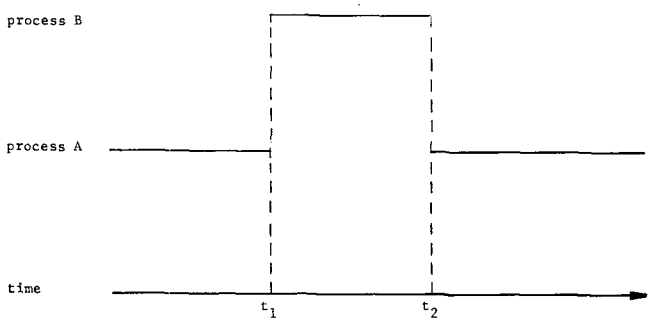

Fig. 1

In general, to change from a process A to a process B we must first save the state of A and then load the state of B into the processor. In the following we designate the change from any process to a process A by *CHANGE TO A*. This operation is similar to the operation of linkage between coroutines described in [3].

The reader will notice that no distinction has been made between code written by a user and code written by system programmers. This is deliberate. In fact, most processes will consist of some code by a user and some

* Private Consultant.

code by system programmers. In addition we wish to have all code a part of some process.

1.2 Events

When a process is running it may decide that it cannot continue until some condition is met; for example, it may need to wait until an I/O operation is completed before continuing. In order to be reinstated when the condition is met, the process will make provisions with some other process. To implement these ideas we define an entity called an *event*. The event is associated in some way with a condition.

An event, E, consists of a two-state variable, S, and a list, L, where the two states are **happened** and **not happened**, and where L is a list of processes waiting for the occurrence of the event. By occurrence of the event we mean that the event variable changes from **not happened** to **happened**. A process may cause an event to happen. When this occurs, all the processes waiting for the event now become candidates for reinstatement. Processes may initialize event variables to **not happened**. Since some of the conditions in which a process is interested may be local to the process, the process must be able to create events local to itself.

It should be pointed out that an event is purely a software construct and does not have a necessary connection with hardware signals, although there will usually exist an event for any signal.

1.2.1 *Event Primitives.* The foregoing suggests the following six primitives for dealing with the interaction of events and processes: *WAIT*, *CAUSE*, *RESET*, *HAPPENED*, *CREATE EVENT*, and *DESTROY EVENT*. Each of these primitives has as a parameter the name of the event to which it applies.

1.2.1.1 *WAIT* (E). If S is in the state **not happened**, makes an entry in L for the process which invoked *WAIT* and then suspends that process; when the process is reinstated, it resumes at the suspension point. For discussion of suspension, see Section 1.4.1.

1.2.1.2 *CAUSE* (E). Sets S to the state **happened** and queues each entry in the event-list in a queue of processes waiting to be reinstated on a processor. For discussion of queueing, see Section 1.4.1.

1.2.1.3 *RESET* (E). Sets S to the state **not happened**.

1.2.1.4 *HAPPENED* (E). A Boolean function with the value **true** if E is in the state **happened**, and **false** otherwise.

1.2.1.5 *CREATE EVENT* (E). Creates an event with the name E for the process which invokes *CREATE*.

1.2.1.6 *DESTROY EVENT* (E). Destroys the event with the name E.

1.2.2 *A More Complex Form for WAIT.* Sometimes it is important that a process be able to wait for the occurrence of any one of a number of events. For example, we may wish to wait for the completion of either a read or a write operation if we are copying a file from one media to another. This is difficult to express with the primitives introduced so far. Consequently, we need a primitive, $WAIT(E_1, E_2, \cdots, E_n)$, which will suspend processing until at least one of the events, E_i, occurs. It is fairly clear what changes must be made in our proposal to implement this. We will not give any further discussion of this primitive in this paper.

We will return to the discussion of events and the primitives *WAIT* and *CAUSE* after introducing the notion of a facility.

1.3 Facilities

In order to complete its computational task, a process requires the use of the various resources of the system, such as memory, I/O channels, and peripheral units. In a multiuser environment, processes compete for the use of resources. The management of these resources must then include mechanisms for the requesting and releasing of resources.

Moreover, the allocation of resources may take account of the relative importance of processes, i.e., the most important process should have control of a resource before a less important process. This relationship is established by assigning a priority, π, to each process. We realize that the priority may change during the course of computation. However, this is a policy matter which we will not discuss here.

Whenever a resource is free, i.e., the resource is not in use by any process, or if by priority or perhaps from hardware or software considerations the resource is interruptable, a request for the facility can be granted. However, when a resource is busy with higher priority work or is not interruptable, a requestor must be entered in a queue to await his turn. When a process no longer has use of a resource it may release the resource.

1.3.1 *Definition of Facility.* To implement the idea of requesting and releasing resources we will associate to each resource an entity which we call a *facility*. A facility may be composed of one or more homogeneous parts; that is, more than one process may use the facility simultaneously.

A process which is using a facility is called a *controller* of the facility. For example, we will consider a set of crossbarred I/O channels to be one multipart facility. We call the number of parts of a facility the *parallelism* of the facility. We say the facility is busy if all parts are in use; otherwise it is free.

When a process, A, is a controller of a facility, F, the process A holds the facility with some control strength. By this we mean that if another process, B, requests F with a control strength greater than the control strength with which A holds F, then B takes F from A.

Associated with a facility is a *queue*. When a process, A, requests a facility which is busy and all of whose controllers have control strength greater than or equal to the control strength with which A requests the facility, then A must be entered into the queue. Also, when one process

takes a facility from another process, then the latter process must be entered into the queue for the facility.

The queue is ordered according to the following hierarchy of values:

(1) control strength;

(2) queueing state, where queueing state is 1 if the facility has been taken from the process, and otherwise 0;

(3) priority, and

(4) arrival time.

The difference between control strength and priority is that control strength is used for defining interrupt classes (an interrupt class is the set of all requests with the same control strength), while priority is used for ordering requests within the same interrupt class. Control strength varies from request to request while priority is a property of the process and will tend to be relatively invariant in time. The reason for ordering according to queueing state is that we wish to maintain the order among requests of the same interrupt class regardless of requests of higher control strength. If the queueing state is not used, then it may happen that after some period of time the initial order between two requests of the same control strength will change. In some cases this is not desirable.

We note that a more general scheme would allow queueing according to an arbitrary function and interruption according to another arbitrary function. Each facility would have its own characteristic functions. This would allow more sophisticated allocation algorithms. However, the simple scheme presented is adequate for allocation of the resources we discuss here. Consequently, we will not introduce this additional complexity.

A process may release a facility. This means that the process is no longer a controller of the facility and if the facility queue is not empty, then the first process in the queue (according to the given order) becomes a controller of the facility.

For each facility there exist three procedures which we call *ALLOCATE*-, *INTERRUPT*, and *UNALLOCATE*. When a process becomes the controller of a facility, the procedure *ALLOCATE* for that facility is invoked, when a process releases a facility the procedure *UNALLOCATE* is invoked, and when a facility is taken from a process the procedure *INTERRUPT* is invoked. We emphasize that the procedures will vary from one facility to another. Any of these procedures may be null for some facility, e.g., a null *INTERRUPT* procedure means that if the facility is to be taken from a process, there is no special work to perform other than the normal queue manipulations.

In summary, a facility consists of the following:

(1) a queue of processes which have requested the facility,

(2) a vector of controllers of length equal to the parallelism of the facility, and

(3) a set of three procedures, *ALLOCATE*, *UNALLOCATE*, and *INTERRUPT*.

At this point it is clear that facilities are a software construct and may have a correspondence with either hardware or software resources.

1.3.2 *Facility Primitives.* The primitives used in dealing with facilities are: *REQUEST*, *RELEASE*, *CREATE FACILITY*, *DESTROY FACILITY*, and *FREE*.

1.3.2.1 *CREATE FACILITY* (*F*, *N*, *Pa*, *Pu*, *Pi*). Establishes a facility *F* with parallelism *N*, with the associated set of procedures *Pa* for *ALLOCATE*, *Pu* for *UNALLOCATE*, and *Pi* for *INTERRUPT*.

1.3.2.2 *DESTROY FACILITY* (*F*). Destroys the facility *F*.

1.3.2.3 *REQUEST* (*F*, π, *NM*, *CS*, *INFO*). Requests the facility *F* for the process named *NM* with a control strength, *CS* and priority π. The process name is included as a parameter so that a process may request a facility on the behalf of another process (refer to Figure 2). *INFO* is used to pass parameters to the procedures of the facility.

1.3.2.4 *RELEASE* (*F*, *m*). Releases the part of *F* identified by the ordinal *m* (refer to Figure 3).

1.3.2.5 *FREE* (*F*, *m*). A function yielding the result **false** when *F* is busy; otherwise the result is **true** and the value of *m* is the ordinal number representing one of the free parts.

1.4 Use of Primitives

Given this model as a basis, the task of the analyst is to define processes, facilities and events involved in his system. The policy for the management of each particular resource is embodied in the three procedures *ALLOCATE*, *UNALLOCATE*, and *INTERRUPT*. Housekeeping funcions such as logging can be inserted in these procedures as desired by the system analyst.

Frequently there are events associated with a particular facility, usually the event that the facility has been allocated to a given process and the event that the faclity has been unallocated for a given process.

The creation of such events by a process establishes a means of communication between the process itself and any other process which will cause these events.

1.4.1 *Processor Facility.* In order to define more precisely the *WAIT* and *CAUSE* mechanism, we now introduce a facility associated with the processor resource. This association seems natural since we need a queue of processes which are ready to run. Moreover there is at any moment a controller of the processor which may be interrupted by another process. In this case the interrupted process has to be reentered into the queue of processes which are ready to run and a new process will take control of the processor. The mechanism for switching from one process to another is described by the statement *CHANGE*.

If a running process decides that it cannot proceed further until some condition is satisfied, the process must

release the processor. At this moment a new process will become the controller of processor, namely, one of the processes in the processor queue.

From the point of view of queueing, it appears clear that a processor and a facility have a similar behavior.

In the case of a one-processor system we can define a corresponding facility, *PROCESSOR*, in the following way:

N, the parallelism, is 1;

the procedure *ALLOCATE* is defined to be: *CHANGE* to the process that has now become the controller;

the procedure *INTERRUPT* is null;

the procedure *UNALLOCATE* is null.

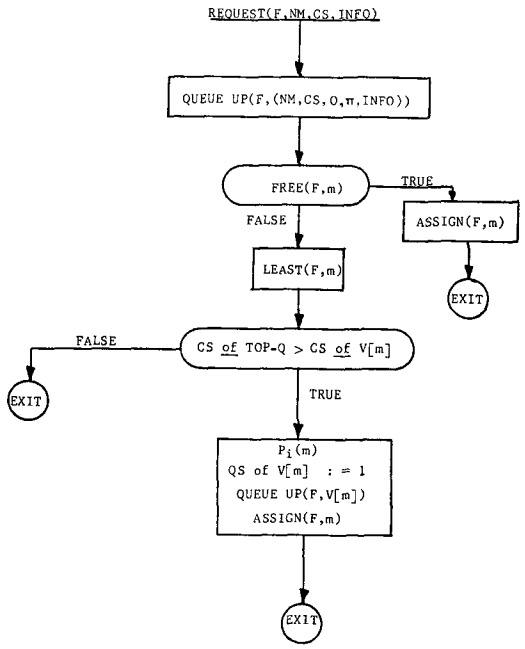

FIG. 2

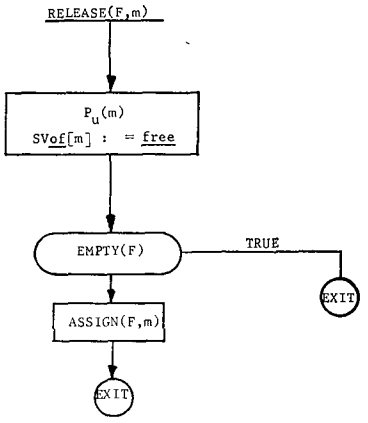

FIG. 3

Referring now to the description of *WAIT* and *CAUSE* in Sections 1.2.1.1 and 1.2.1.2, we can define the suspension and the reinstatement of a process. When a process has to suspend itself, the process has to execute *RELEASE* (*PROCESSOR*, 1). When a process B wants to reinstate another process A, as in *CAUSE*, the process B has to execute *REQUEST* (*PROCESSOR*, π, A, *CS*, *INFO*).

To assure that the processor queue is never empty, we assume the existence of a process which is always ready to run. This is called the *spin process*. It has a priority lower than that of all other processes in the system. The reader should refer to Figures 4 and 5, which give flowcharts for *WAIT* and *CAUSE*. The list associated with an event may be ordered according to priority so that control will be given to waiting processes in priority order.

1.5 The Primitive *WHEN*

Let us now explain how a process can specify that an action, named *AN*, is to be performed on the occurrence

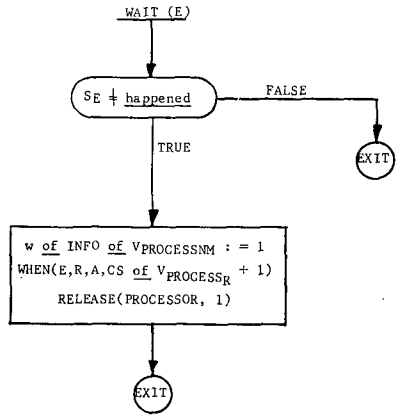

FIG. 4

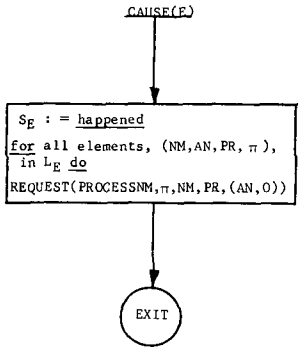

FIG. 5

of some event, E. A natural notation might be *WHEN E DO AN*. Consider the case in which a process specifies both *WHEN* E_1 *DO* AN_1, and *WHEN* E_2 *DO* AN_2. If E_1 occurs, and then E_2 occurs while AN_1 is being performed, should we suspend AN_1 to do AN_2 or should we do AN_2 only upon the completion of AN_1? It is easy to convince oneself that for some events the answer is the former and for others the answer is the latter. Hence we will add another parameter to *WHEN* to specify the preference between actions. We will write *WHEN* (*E*, *NM*, *AN*, *PR*) where E is the event, *NM* is a process name, *AN* is the name of the action, and *PR* is an integer giving the preference.

In some process *NM*, let us regard the "main program" as an action, *MP*, with preference 0. The meaning of *WHEN* (*E*, *NM*, *AN*, *PR*) where $PR > 0$ is that if E occurs then the action *MP* is no longer eligible for processing until the action *AN* is completed.

In general, let AN_1 be an action belonging to process *NM* which is eligible for processing with preference PR_1. The execution of the statement *WHEN* (*E*, *NM*, AN_2, PR_2) implies that if at any later time the state of E is **happened** then the following occurs:

(1) If $PR_2 > PR_1$ then AN_1 cannot be eligible for processing until the action AN_2 is completed.

(2) If $PR_2 \leqq PR_1$ then AN_1 remains eligible for processing and AN_2 cannot be eligible until after the completion of AN_1.

The presence of the *WHEN* primitive in a process imposes a nested structure upon that process. This structure has all the characteristics of a facility if we identify the control strength with the preference for the actions.

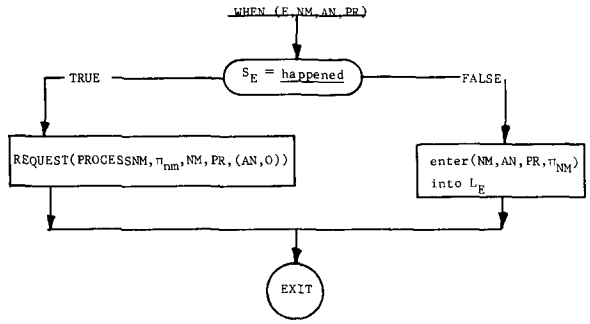

FIG. 6

We will not develop this idea further here. In Section 2 we describe in detail how it is possible to define a facility for each process to order the actions specified by the *WHEN* primitives. Naturally the *CAUSE* primitive will be influenced by the introduction of the *WHEN* primitive. We also have chosen in the next part to specify the *WAIT* primitive in terms of *WHEN*. A flowchart is given in Figure 6.

2. System Description

2.1 Processes in the System

A computing system may be regarded as a process. The set of all possible states which may be assumed is the set Σ. For a state of $\sigma \in \Sigma$, $S(\sigma)$ is the set of possible successor states. In general, $S(\sigma)$ is a set and not an element since a system may be influenced by external signals. An *action* is a rule which selects a particular state from $S(\sigma)$. The special action, processor, is the rule which defines how the system proceeds in the absence of external signals or interrupts.

In the following sections we consider such a process. We will call the process, the *system*. Such a system could be organized so that it appears to be a collection of subprocesses. We will then give a partial description of such an organization, using the term *processes* for subprocesses.

This description takes the form of a set of primitives or procedures that a collection of subprocesses may use to organize a process. These primitives will be given in a combination of English and a stylized ALGOL. The primitives assume a data structure that includes variables, lists, arrays, and queues.

In the following sections we will want to refer to processes, actions, and states by name. Consequently, we assume that each process, each action, and each state has a unique name.

We will assume that for each process A, there is an integer π_A called the priority of A.

2.2 Facilities

A facility element, e, is defined to be the ordered 5-tuple (*NM*, *CS*, *QS*, π, *INFO*) where:

NM is a process name;
CS is an integer called the control strength of the element;
QS is a Boolean variable called the queueing state of the element;
π is an integer called the priority of the process;
INFO is additional information defined for each particular facility element.

A facility, F, is defined by the triple (V, Q, P) where:

(1) V is a vector of N elements. N is the parallelism of the facility. Each element of V has two parts, a facility element and a variable, *SV*, with two states, **free** and **busy**. If *SV* = **free** the corresponding facility element is empty.

(2) Q is a queue of indefinite length, comprised of facility elements, ordered according to the following hierarchy of values:

(a) *CS*
(b) *QS*
(c) π

(3) P is a set of three procedures:

(a) $Pa(m)$ *ALLOCATE* procedure,
(b) $Pu(m)$ *UNALLOCATE* procedure,
(c) $Pi(m)$ *INTERRUPT* procedure,

where m is the ordinal number of a facility element in V.

We denote the mth element of V by $V[m]$ and the first element of Q by $TOP\text{-}Q$.

A process, B, is called a *controller* of F if NM **of** $V[m] = B$ for some m.

The facility F is said to be **free** if SV **of** $V[m]$ = **free** for some m; otherwise the facility is **busy**.

Where necessary, the constituents of a facility will be subscripted with their facility name, e.g., V_F, Q_F, etc., to distinguish them from the corresponding components of other facilities.

2.2.1 *Facility Primitives.* A process can request and release facilities. In order to describe these procedures we introduce the following primitives.

2.2.1.1 $QUEUE\ UP(F, e)$. Places the facility element e into Q, using CS, QS, π and arrival time for the ordering rule.

2.2.1.2 $ASSIGN(F, m)$. Removes the facility element $TOP\text{-}Q$ and assigns it to $V[m]$ and sets SV **of** $V[m]$ to **busy**. Then it invokes $Pa(m)$.

2.2.1.3 $FREE(F, m)$. A Boolean function yielding the value of **false** if there does not exist an m such that SV **of** $V[m]$ is **free**. Otherwise the value is **true** and m is set to the ordinal value such that SV **of** $V[m]$ is **free**.

2.2.1.4 $LEAST(F, m)$. Sets $m := j$ where j is such that CS **of** $V[j] \leq CS$ *of* $V[n]$ for any n.

2.2.1.5 $EMPTY(F)$. A Boolean procedure with the value **true** if Q is empty, otherwise **false**.

2.2.1.6 $CREATE\ FACILITY(F, N, Pa, Pu, Pi)$. Creates a facility named F with parallelism N and the procedures Pa, Pu, Pi.

2.2.1.7 $DESTROY\ FACILITY(F)$. Destroys the facility F.

2.2.1.8 $REQUEST(F, \pi, NM, CS, INFO)$.

```
begin
  QUEUE UP(F,(NM,CS,0,π,INFO));
  if FREE(F,m) then ASSIGN(F,m)
else begin
  LEAST(F,m);
  if CS of TOP = Q > CS of V[m] then
  begin
    Pi(m);
    QS of V[m] := 1;
    QUEUE UP(F,V[m]);
    ASSIGN(F,m)
end end end;
```

2.2.1.9 $RELEASE(F, m)$.

```
begin
  Pu(m);
  SV of V[m] := free;
  if ¬ EMPTY(F) then
    ASSIGN(F,m);
end;
```

2.3 Events

An event E is defined to be the ordered pair (S, L) where:

(1) S is a variable which has two values, **happened** and **not happened**;

(2) L is a list of elements, each element being an ordered 4-tuple (NM, AN, PR, π), where

(i) NM is a process name;

(ii) AN is an action name;

(iii) PR is an integer called the preference of the element; and

(iv) π is an integer which is the priority of the process.

The list L is ordered by priority.

Where necessary S and L will be subscripted, e.g., S_E and L_E, to distinguish them from the variables and lists of other events.

In order to define how events influence the behavior of processes we introduce several facilities and primitives for manipulating events and these facilities.

2.3.1 *Processor Facility.* To each process, NM, we associate an element of a vector named $STATE$; thus $STATE[NM]$ has as a value the last stored state of NM. We also assume the global variables σ, $NAME$, and $SUCCESSOR$ whose values are system states, process names, and action names, respectively. For the $PROCESSOR$ facility and the following $PROCESS$ facility, $INFO$ will consist of the pair (AN, w), where the values of AN are action names and the values of w are either 0 or 1.

The facility, $PROCESSOR$, is created by:

$CREATE\ FACILITY\ (PROCESSOR, 1, Pa, Pu, Pi)$

where

```
Pa ≡ SUCCESSOR := AN of INFO of V_PROCESSOR[1];
     AN of INFO of V_PROCESSOR[1] := P;
     AN of INFO of V_PROCESS NM[1] := P;
     STATE[NAME] := σ;
     σ := SUCCESSOR(STATE[NM]);
Pu ≡ NAME := NM of V_PROCESSOR[1];
Pi ≡ Pu.
```

The procedure Pa, which corresponds to $CHANGE$ of Section 1, assigns the action in $INFO$ to be the $SUCCESSOR$ function for the process NM, sets the actions of $INFO$ to P, the action, processor, defined in 2.1, stores the state, σ, of the current process and determines the next state of NM by applying $SUCCESSOR$ to the current state of NM. The procedure Pu saves the name of the current process for subsequent use in Pa.

It is necessary that $Q_{PROCESSOR}$ never be empty. Therefore we introduce a process, $SPIN$, which has the properties that:

(1) π_{SPIN} is less than π_{NM} for any other process NM, and

(2) $SPIN$ never executes $RELEASE(PROCESSOR, 1)$.

2.3.2 *Process Facilities.* For each process, A, we create a facility called $PROCESS\ A$ by:

$CREATE\ FACILITY\ (PROCESS\ A, 1, Pa, Pu, Pi)$,

where

```
Pa ≡ if w of INFO of V_PROCESS[1] = 0 then
     REQUEST(PROCESSOR, π, A, π, (AN of INFO of
     V_PROCESS A[1], 0));
     if NM of V_PROCESSOR[1] = A then
     RELEASE(PROCESSOR, 1);
Pu ≡ delete all facility elements with NM = A from
     Q_PROCESSOR ;
Pi ≡ Pu.
```

To illustrate how *Pa*, *Pu*, and *Pi* work, consider the following situations.

(1) In process A an action AN_1 which controls *PROCESS A* releases *PROCESS A* when $Q_{PROCESS\ A}$ is nonempty. By virtue of AN_1 executing a statement, AN_1 must be controller of the processor. Note that it is possible that the action AN_2 associated with the element in *TOP-Q* may be waiting for some event, which implies that the processor cannot be requested for AN_2. If AN_2 is not waiting, signified by $w = 0$, then the processor must be requested for AN_2. Furthermore, the request processor for AN_2 of A must be issued using the same or less $CS_{PROCESSOR}$ as that of AN_1; if a higher *CS* is used the *PROCESSOR* facility would be interrupted which would requeue AN_1 which contradicts the release by AN_1.

(2) If an action AN_1 of process A requests *PROCESS A* for an action AN_2 with a sufficiently high $CS_{PROCESS\ A}$, we find that AN_1 must request the processor for AN_2 with the same or less $CS_{PROCESSOR}$ than that of AN_1 by an argument similar to the reasons in example (1) above. This relationship between the control strengths of AN_1 and AN_2 allows AN_1 to release the processor so that AN_2 may eventually become controller of the processor.

(3) Suppose some action of a process B requests *PROCESS A*, similar to example (2) above. Before B requests the processor for A it is necessary to eliminate the possibility of having more than one element in $Q_{PROCESSOR}$ for A by removing any element in $Q_{PROCESSOR}$ belonging to A. Depending on the relative values of $CS_{PROCESSOR}$ of A and B the request processor issued by B will either cause B to be requeued in $Q_{PROCESSOR}$ and A to take control of the processor, or simply enter A in $Q_{PROCESSOR}$ and leave B in control of the processor.

We have chosen the policy of using π_A for the *CS* of all processor requests for A; i.e., the processor is to be allocated first to processes with highest priority. To allocate the processor on another basis, another expression for *CS* in the request processor in *Pa* may be employed provided that the *CS* of the request is always the same or less than the *CS* of the controller of the processor.

2.3.3 *WHEN(E, NM, AN, PR)*. The primitive *WHEN* is defined by:

```
if S_E = happened then
        REQUEST(PROCESS NM, π_NM, NM, PR, (AN, 0))
else enter (NM, AN, PR, π_NM) into L_E.
```

Thus if the event E has happened, *WHEN* requests the processor for action AN of process NM with control strength PR; otherwise, the parameters are saved in L_E and the request is deferred until a *CAUSE(E)* is executed.

2.3.4 *WAIT(E)*. The primitive *WAIT* is defined by:

```
if S_E = happened then
  begin
    w of INFO of V_PROCESSOR NM[1] := 1;
    WHEN(E, NM, AN*, CS of V_PROCESS NM[1]+1);
    RELEASE(PROCESSOR, 1)
  end;
```

where NM = NM of $V_{PROCESSOR}[1]$ and AN^* is an action which is defined by:

```
find the first element e in Q_PROCESS NM such that CS = CS of
  V_PROCESS NM[1]−1;
w of INFO of e := 0;
RELEASE(PROCESS NM, 1);
```

Thus if the event, E, has not happened, *WAIT* indicates that the process NM is waiting for an event by setting $w = 1$, provides an action AN^* to reset w (AN^* wil run when E happens), and releases the processor.

2.3.5 *CAUSE(E)*. The primitive *CAUSE* is defined by:

```
S_E := happened
for all elements, (NM, AN, PR, π), in L_E do
REQUEST (PROCESS NM, π, NM, PR, (AN, 0));
```

CAUSE sets the event, E, to **happened** and requests the processor for every element in L_E for action AN of process NM with control strength PR.

3. Examples

In Section 2.3, a *PROCESSOR* facility and a set of *PROCESS* facilities were introduced in order to explain the influence of events upon processes. Now some other facilities are introduced, which are of importance in an operating system.

3.1 Interlocks

When many processes are running concurrently, it is sometimes necessary that one process inhibit the use of a set of data by other processes, e.g., linked lists are generally not readable while the links are being altered. We can create a facility, T, for such a set of data in the following way:

CREATE FACILITY(T, 1, Pa, Pu, Pi)

where

Pa = CAUSE(DATA-ACCESSIBLE)
Pu = null
Pi = null

where *DATA-ACCESSIBLE* is an event local to the sequence *LOCK* described below.

A process A, may use this data after executing the following sequence:

```
LOCK:
RESET(DATA-ACCESSIBLE);
REQUEST(T, π, A, 0, DATA-ACCESSIBLE);
WAIT(DATA-ACCESSIBLE).
```

After using the data, to allow access to the data by another process, A may execute the following statement:

UNLOCK: RELEASE(T, 1).

Clearly, all processes must use the conventions for *LOCK* and *UNLOCK* to insure the integrity of the data associated with T. In particular, all requests for T must have the same control strength, say 0, so that T will not be interrupted.

This scheme forces a process to wait, i.e., give up control of the processor, if the process wants access to data which is in use by some other process. Notice that if processor control is actually lost, control will be regained through the eventual execution of *Pa*: *CAUSE(DATA-ACCESSIBLE)*. A scheme can be devised using *WHEN* in place of *WAIT* such that the processor need not be released.

3.2 Input-Output

To describe a typical input-output mechanism in terms of facilities, assume that we have k tape units connected to j input-output channels in a cross-bar fashion; i.e., information may be transmitted between any tape unit and any channel. We postulate that a channel $C[m]$

(i) operates in parallel with the processor and independently of other channels;

(ii) can execute an input-output command, initiated by the processor, only if $C[m]$ is not busy; and

(iii) executes *CAUSE(TRANSMISSION-COMPLETE* $[m]$*)*, where *TRANSMISSION-COMPLETE* is a j-element array of events, to notify the processor that $C[m]$ is not busy.

Postulate (i) allows a process to use the input-output resources and processor simultaneously; (ii) forces a particular discipline in issuing input-output commands, and (iii) furnishes an event to signify that a channel has become free.

We define two facilities:

(1) a *CHANNEL* facility defined by:

CREATE FACILITY (CHANNEL, j, Pa, Pu, Pi),

where

Pa ≡ *RESET(TRANSMISSION-COMPLETE*$[m]$*);*
WHEN(TRANSMISSION-COMPLETE$[m]$*, NM, IO-COMPLETE, CS+1);*
do hardware instructions to initiate *IO-COMMAND*;
Pu ≡ *null*
Pi ≡ *null*

where *IO-COMMAND* is a part of *INFO* of $V_{CHANNEL}[m]$,
$NM = NM$ **of** $V_{CHANNEL}[m]$, and
$CS = CS$ **of** $V_{PROCESS\ NM}[1]$.

(2) a tape unit facility, T, defined by:

CREATE FACILITY (T, 1, Pa, Pu, Pi)

where

Pa ≡ **if** *QS* **of** $V_T[1] = 0$ **then**
REQUEST (CHANNEL, π, NM, 0, (UNIT-FREE, IO-COMMAND))
Pu ≡ *CAUSE (UNIT-FREE)*
Pi ≡ *null.*

Associated with each tape unit is a facility defined analogously to T.

If a process, *NM*, wishes to perform an *I/O* on the tape unit associated with T, the following sequence is executed by *NM*:

```
RESET(UNIT-FREE);
REQUEST(T, π, NM, 0, (UNIT-FREE, IO-COMMAND));
⋮
WAIT(UNIT);
```

That is, the process *NM* initializes an event, *UNIT-FREE*, to **not happened** and requests the facility T passing the input-output command, *IO-COMMAND* and *UNIT-FREE* as parameters in *INFO*. The process may continue using the processor, perhaps requesting facility T. When the process requires that the input-output command, *IO-COMMAND*, be finished, the process may execute the *WAIT* shown above.

When the transmission of data is completed, the *WHEN* statement in *Pa* of the *CHANNEL* facility and the *CAUSE* executed by the channel cause the execution of the following action, called *IO-COMPLETE*:

```
RELEASE(CHANNEL, m);
if TRANSMISSION-UNSUCCESSFUL then IO-ERROR;
RELEASE(T, 1);
```

The release *CHANNEL* causes the queue for *CHANNEL* to be served, i.e., the next input-output command is initiated. The input-output transmission is checked for the occurrence of a malfunction; if so, the recovery procedure *IO-ERROR* is invoked. To remedy some malfunctions, *IO-ERROR* may need to request T for special input-output commands which must be performed before any normal commands already queued in Q_T. To insure proper ordering of commands, a $CS = CS$ **of** $V_T[1] + 1$ is used when requesting T for these special commands. Finally, whether *IO-ERROR* is invoked or not, T is released so that the queue for T may be served, i.e., the next request *CHANNEL* issued.

4. Concluding Remarks

In conclusion, several observations are in order.

The overall scheme is able to accommodate a large class of queueing problems that arise in implementation of operating systems. One of the advantages of this kind of approach is that one has a uniform structure for all the queues in the system.

In an actual implementation various optimizations are possible; however, we feel that in any case the uniformity of queue structure should not be violated.

If the operating system will reside in a multiprocessor system, the processor facility could be a multipart facility. This somewhat complicates *Pa*, *Pu*, and *Pi* for both *PROCESSOR* and *PROCESS* facilities.

Received December, 1966; revised August, 1967

REFERENCES

1. Vyssotsky, V. A., Corbató, F. J., and Graham, R. M. Structure of the multics supervisor. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, 1965, 203–212.
2. Conway, M. E. A multiprocessor system design. Proc. AFIPS Fall Joint Comput. Conf., Vol. 24, 1963, 139–146.
3. Conway, M. E. Design of a separable transition-diagram compiler. *Comm. ACM 6*, 7 (July 1963), 396–408.
4. Knuth, D. E., and McNely, J. L. SOL—A symbolic language for general purpose systems simulation. *IEEE Trans. EC 13* (Aug. 1964), 401–408.
5. Knuth, D. E., and McNely, J. L. A formal definition of SOL. *IEEE Trans. EC 13* (Aug. 1964), 409–414.