Designers and users of multitasking operating systems must be alert to the problem of task deadlock, which prevents the affected tasks from being completed.

This paper describes the conditions that can result in task deadlock in any multitasking systems. Also discussed are techniques for avoiding deadlock in both operating system and application program design. Finally, it is shown how these techniques were applied in the design of the SYSTEM/360 *Operating System job initiator, the part of the system that allocates major resources needed to execute jobs.*

Avoiding deadlock in multitasking systems

by J. W. Havender

Modern operating systems, by permitting more than one data processing task to be performed concurrently, make possible more efficient use of system resources. If a program that is being executed to accomplish a task must be delayed, for example, until more data is read into the computer, performance of some other completely independent task can proceed. The central processing unit can execute another program or even execute the same program to accomplish a different task.

In the competition for system resources, such as main storage space or data sets (files), however, all multitasking systems are subject to a condition referred to as *deadlock*. This condition prevents the affected tasks from being carried out to completion. Several conditions must exist for tasks to become deadlocked. Consider a simple example involving only two tasks that are being performed concurrently. Assume that each task has been allocated a system resource which has been used in partially completing the task. Assume also that allocated resources are released only after completion of the task. If completing each task requires an additional resource and if the additional resource has been allocated to the other task, neither task can be completed; that is, task deadlock exists.

Such impasses can arise in many forms involving many tasks, and when task deadlock does occur, there is no known general technique for correcting the condition. Recovery procedures, if

they exist, are dependent on the particular processes that give rise to the situation. Thus, task deadlock situations are not now amenable to system recovery procedures. However, the problem has been dealt with effectively in the SYSTEM/360 Operating System[1] by systematically avoiding task deadlock. Throughout this paper, the term "operating system" implies that version of the SYSTEM/360 Operating System that provides multiprogramming with a variable number of tasks (MVT).

Although deadlock problems between jobs have been overcome within the operating system, deadlock is a real possibility that must be considered by designers of other multitasking systems and by users of this system who plan to exploit its multitasking capability in application programs. In the following discussion, some background information about the operating system is provided, the deadlock problem is defined more completely, some techniques that have been useful in circumventing deadlock situations are described, and, finally, examples taken from the job initiator design are used to illustrate some of these techniques.

Background

Although the multitasking concept of the operating system is dominated by the theme of executing tasks independently, some interdependence among tasks is necessary. To accommodate this interdependence, the abstract concept of an *event* was evolved. Completion of a task may depend on the occurrence of one or more events, such as transferring data between an I/O device and main storage. While the original task is deferred until the occurrence of such an event, the system can proceed with other tasks. We will call these interdependencies among tasks *interlocks*, and provisions are made in this operating system for two basic kinds of interlocks among tasks, which we call explicit and implicit.

Two system macroinstructions have been provided to permit explicit interlocks. The WAIT macroinstruction permits a programmer to request that a task be delayed until the occurrence of one or more events; the POST macroinstruction permits him to signal that an event has in fact occurred.

Implicit interlocks are provided when a service or resource that is under operating system control is required for a task. The system forces the task to be deferred until the needed resource is available. For example, if a serially reusable program that is currently being used to perform one task is required to perform a second task, the system delays the second task until the event "program has become available" occurs.

It was also realized that a mechanism was required to coordinate the use of resources not under system control. Provisions were incorporated to permit such resources to be defined and named, and the macroinstructions ENQ and DEQ were provided to control access to them.[2] The ENQ macroinstruction causes a request to be put on the queue for a named resource; the DEQ

macroinstruction requests removal from the queue because use of the resource is completed.

A further refinement came about when it was realized that a resource might or might not have to be used serially. To accommodate this situation, use of a named resource may be defined as exclusive or shareable. The system coordinates use of such a resource so that exclusive use precludes any other concurrent use, while any number of shareable uses can be made concurrently.

The original multitasking concept of the operating system envisioned relatively unrestrained competition for resources to perform a number of tasks concurrently. The system acted mainly as a dispenser and collector of these resources. But, as the system evolved, many instances of task deadlock were uncovered, most of which were centered about the single resource—main storage. The planned solution was "roll out." In this procedure, the program that is to be executed to perform the task is rolled out of main storage onto external storage to make more main storage available. However, the problems encountered with roll-out required a redirection in thinking, leading to the current version. In this operating system, a subset of system resources is allocated to the job step. The job initiator, one of the job management routines, obtains the subset of resources needed for a job step in a manner designed to avoid deadlocks.

The deadlock problem

Deadlock situations arise when a group of tasks becomes interlocked in such a way that they cannot be completed. Consider first a simple example involving explicit interlocks. Assume that completion of task A depends on the occurrence of event b; task B on event a. Also assume that only the program being executed to accomplish A can record (post) the occurrence of event a; only the program for task B can post event b. Clearly, this design makes deadlock inevitable. But this type of problem is not likely to occur in practice, because the designer had to take explicit steps to create the problem. However, interlocks involving system resources that must be used serially can produce deadlocks in much more subtle ways.

Using a resource that must be used serially to accomplish separate tasks necessarily results in interlocks among the tasks. If a task requires a resource that is not available, that task becomes dependent upon some other task for release of the resource. And if the task to which the resource has been allocated requires, in turn, a resource allocated for the original task (or otherwise becomes interlocked with the original task), a deadlock situation occurs.

In the earlier example of explicit interlocks, the two events, a and b, must first be defined and then the programs must be designed so that the occurrence of each of the two events depends

explicitly upon occurrence of the other. However, if we consider a task that first requires a serially reusable resource, A, and later an additional resource, B, it is not so obvious that deadlock is possible. Yet deadlock can occur if there is another task or group of interlocked tasks that first requires resource B and then resource A. If, in addition, the order of execution of the programs is such that each progresses beyond the point of acquiring their first resource before either requests their second resource, deadlock becomes inevitable.

serially reusable resources

Although deadlocks can arise from any type of task interlock, the emphasis in the following discussion is on deadlocks arising from requests for serially reusable resources. Most system resources are serially reusable. Consider the following examples:

Storage media (not to be confused with the information they contain). These qualify as serially reusable resources—serial because they contain only one set of information at a time; reusable because they can contain other information when the original information is no longer required. (Although not all storage media fulfill these requirements, the ones of interest here, such as tape, disk tracks, and main storage, do.)

System components (such as tape drives, disk drives, access mechanisms, and central processing units). Some components, such as tape drives and unit record equipment, are commonly considered to be serially reusable. Disk drives are intrinsically shareable in performing independent tasks, but when one contains a volume totally dedicated to an application, it must be considered to be serially reusable.

Information. In a static state, information is intrinsically shareable (although it may be defined not shareable for security reasons). Examples of shareable information are tables of values or read-only programs. But, while information is being changed, it is typically nonshareable. The classical example of this is updating a record in place. Only one updating process can take place at one time. More complex information transitions than this occur in practice, but they have the characteristic property that two such processes on the same information cannot overlap. (The ENQ–DEQ facility should prove to be useful in preventing the overlapping of such processes.)

Avoiding deadlock

To recapitulate, we can see that, as performance of a task progresses, resources are required. If requests are made for more resources for a task to which resources have already been allocated, the possibility of task deadlock must be examined. The pertinent question to be asked is: Is it possible for another task, B, to exist at the same time as this task, A, and that (1) task B has a resource allocated to it that is required for task A, and (2) task B may

require resources that have been allocated to task *A* before the resources allocated to task *B* can be relinquished? Typically, the answer is affirmative, and task deadlock may occur. However, this points the way to the first approach to avoiding the deadlock problem:

Approach 1. In designing a program, request a resource for a task while another resource has been allocated to it only if it can be demonstrated that no other group of interlocked tasks will exist concurrently that (1) have been allocated the required resource and (2) will later require the resource allocated to the original task. Simply stated, ensure that resources are requested for all tasks in the same order.

If, as is often the case, the first approach is not applicable, there are three alternatives:

Approach 2. Request resources collectively. Do not proceed with the task until all required resources have been obtained.

Approach 3. If holding a formerly obtained resource may prevent acquisition of an additional resource, release the original resource before obtaining the additional resource. If the original resource is still required, re-request it collectively with the additional resource.

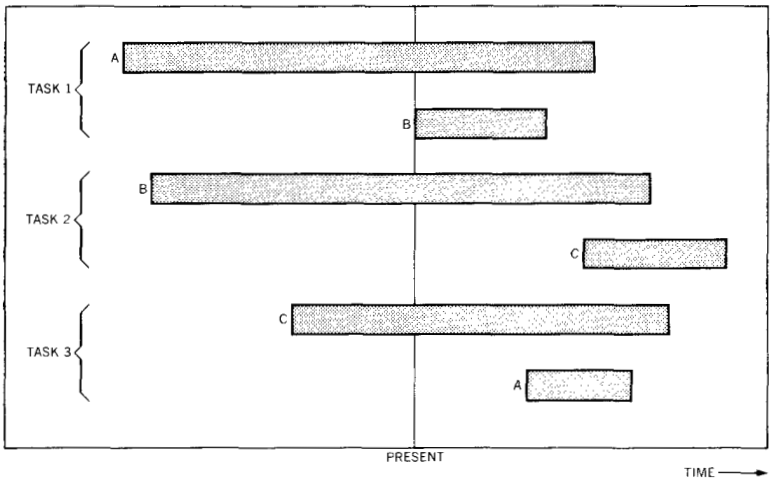
Approach 4. When a request for a resource is denied and when Approach 1 is not applicable, be prepared to take an alternative course of action. Do not wait for the needed resource while retaining other resources.

Observe here that a task should not be delayed for a requested resource even when no known threat of deadlock exists. Consider the example in Figure 1, which involves three tasks and three resources. Time is shown horizontally in the figure, and the vertical line indicates the present time. Use of resources *A*, *B*, and *C* is represented by horizontal bars, which, when they cross the vertical line, indicate that the resource is presently being used. Resource bars beginning to the right of the vertical line indicate that use is planned for the future.

At the present time, resource *B* has been denied to task 1, because it has been allocated to task 2. The question the designer must answer is: Should task 1 be delayed until resource *B* is made available? The answer is negative, because inspection of the figure reveals that the future resource requirements of interlocked tasks 2 and 3 will ultimately result in deadlock if all tasks are delayed until resources become available. Thus, task 2, which cannot be completed, will never relinquish resource *B* for task 1.

It is theoretically possible for the control program to detect deadlocks from requests for resources under its control, but only as they arise—too late to avoid terminating one task. However, interlocks can be established among tasks completely independently of the control program. For example, when a task awaits an

Figure 1 Three tasks competing for resources



unposted event, the control program does not know which task will post the occurrence of the event. Therefore, it must be assumed that deadlock is generally undetectable. Of course, designers of application programs are in a position to be aware of interlocks among nonsystem tasks and can take advantage of this knowledge.

Case histories

The cases to be considered involve the job initiator, the function of which is to process a stream of incoming jobs. This involves repeating the basic cycle of selecting jobs from an input job queue, acquiring for each job step, in turn, the resources required to execute it, and attaching each job step as a task to be done under control of the supervisor. Multijobbing is achieved by concurrently executing multiple copies of the initiator program as independent tasks. Thus, steps from different jobs are initiated for concurrent execution. However, because the initiator tasks include acquiring three resources, one at a time, for each job step, deadlocks among initiator tasks could occur if appropriate precautions were not taken.

The three major resources acquired for jobs by the initiator are: devices and auxiliary storage, data sets, and main storage. We consider first the problem of allocating these three resources individually and then of allocating them in combination.

Consider first the allocation of devices to a job step. It is intuitively appealing to permit free contention among initiator tasks for individual devices, until all devices required for a step have been accumulated. However, this design would permit two or more initiator tasks to become deadlocked. This and other considerations led to the current design in which all devices required for a job step are acquired collectively. While an initiator program

is collecting devices, an interlock incorporated into the control program prevents other initiators from acquiring devices. (Of course, initiators can release devices at virtually any time.)

data set allocation

The next case to be considered is the allocation of data sets, which, in turn, involves an additional problem of data set integrity. To maintain data set integrity, job steps must not be scheduled for concurrent execution if they use common data sets in such a way as to impair either the data sets or the results of the job. Fortunately, many data sets do not enter into the data set integrity problem, including: (1) those that are totally local to the job, such as work files; (2) program libraries that are usually used in a read-only fashion; and (3) those that may be written into but to which access by jobs is controlled by a record or track hold mechanism. However, the possibility of global data sets must be considered (especially in systems having large data bases). Global data sets, which are accessible by all jobs in the system, are sometimes accessed in a fashion designed to prevent their concurrent use by several jobs. Thus, the deadlock problem must be considered.

The deadlock problem hinges on the treatment of global data sets that are passed from one job step to another. Therefore, the first question to be answered is: Should the integrity of a passed data set be maintained from step to step? It is certainly conceivable that a global data set could be read in the first step of a job, that the information in the data set be processed in one or more succeeding steps, and that finally an updated version of the data set be written out in the last step.[3] Therefore, it was decided that the integrity of a passed data set would be maintained from step to step of a job. Thus, we define a passed data set to be in continuous use from the first to the last step that refers to it.

The possibility of deadlock arises when we consider the fact that another global data set may be required for the second step. If each of the data sets is requested within the job step in which it is required, a deadlock can clearly arise if another job that has been scheduled for concurrent execution requires the same data sets in the opposite order. Because this possibility cannot be effectively excluded, Approach 1 is not applicable. Approach 3, releasing formerly acquired data sets and then requesting the new group of data sets, has also been ruled out because of the need to retain data set integrity from step to step. Thus, Approach 2 must be used.

A list of all external data sets referred to in a job is compiled as the job is read into the system. For each data set referred to, the user must also declare whether use of the data set is shareable or exclusive. Then, the ENQ macroinstruction is used to control concurrent execution of the jobs based on these declarations.[4]

device allocation

Allocating devices for a job presents a problem similar to that of allocating data sets; as execution of the job progresses, it may require additional devices. Two jobs being executed concurrently could reach a point where each required some of the devices allo-

cated to the other to proceed to completion. Here, allocating devices for the whole job would avoid deadlock, just as with data sets. However, the requests of two jobs for devices are far more likely to conflict than requests for data sets, so the alternative of allocating all devices required for the total job was not chosen. Instead, Approach 3 was used: all devices allocated to a step are released at the end of the step. Then, the requirements of the next step are considered. This approach is permissible because there is no integrity problem with devices (i.e., a step of another job can intervene between the steps of the original job and a device can be used in all three steps without detrimental results).[5] However, this approach opens up a potential problem, since it will sometimes require dismounting and mounting of volumes. But this problem is addressed in a different manner.[6]

main storage allocation

The next resource that the initiator obtains for a job step is the region of main storage in which the step will be executed. Because main storage is a valuable resource, allocating a region large enough for the largest step in the job is out of the question. Therefore, a region is obtained for each job step. The region for one step may be larger than that for the previous step, but the previously obtained region is freed prior to requesting the next one. Hence, there is no deadlock problem related to regions alone.

combined allocation

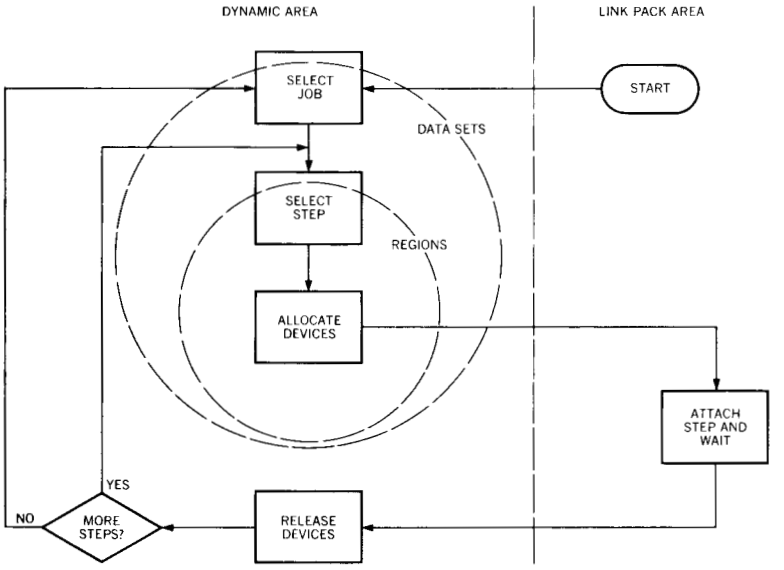
Up to this point, the allocation of data sets, devices, and regions have been considered individually in relation to the deadlock problem. It now becomes necessary to consider acquisition of all three of these resources in combination. But to do this, it is necessary to digress and describe certain design characteristics of the operating system.

Two areas of main storage for the operating system are of interest. The dynamic area is a pool of main storage from which regions are allocated both for job steps and for such system tasks as executing input readers, output writers, and certain operator command routines. The link pack area, which is fixed at the time the operating system control program is first loaded into the computer, contains selected programs from system libraries. Although certain system programs must reside in the link pack area, the user may choose to keep some additional system programs there, based on available storage space and frequency of use.

As each initiator selects a job step, it requests a main storage region in which the step will later be executed. If the request, which is made to the supervisor main storage management routines, cannot be satisfied, that initiator task must wait until a region of the required size becomes available.

A design objective for the operating system was to execute the initiator in the region that would later be used for the job step, in order to reduce the amount of main storage required for the job scheduling function. When a job step is ready to be attached (formally designated as a task to be coordinated under control of the supervisor), the initiator relinquishes the region to the job step by transferring control to a small module in the link pack

Figure 2 Initiator allocation points not yet established



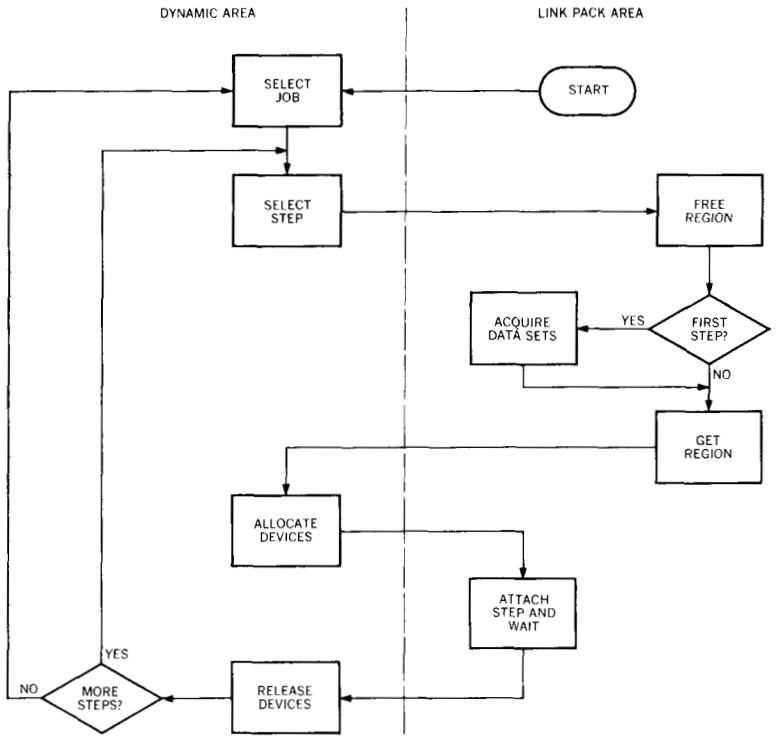
area. This module actually issues the ATTACH macroinstruction, and then waits for the job step to be completed. When the step is completed, this initiator module transfers control back to the region, where step termination procedures are carried out.

Returning to allocation, the three resources—main storage regions (REG), data sets (DS), and devices (DEV)—must be allocated in an order that precludes deadlock. The flowchart in Figure 2 shows the program logic of the initiator. The points in the flow at which devices are allocated and released are shown as fixed, but only general areas are shown for the acquisition of data sets and regions, since it is precise placement of these functions that must be established.

Because data sets are allocated for the life of the job, while the other two resources are allocated only for a step, data sets for the job should be allocated first. The reason for this can be seen by considering the consequences of allocating devices for step one and then data sets for the job, resulting in an order of DEV → DS. For step two, data sets would have been allocated previously, so that only devices would now be allocated, an order of DS → DEV. Thus, the order of allocation would have been reversed in going from step one to step two, creating the possibility of deadlock. Similar reasoning can be applied to the acquisition of regions before data sets. Thus, we have the order DS → (DEV, REG).

The problem remaining then is to decide on the order for acquiring regions and devices. At first glance, it appears reasonable to acquire devices first and defer acquiring a region until just before attaching the job step, since the initiator must relinquish

Figure 3 Initiator allocation points in final design



the space in the dynamic area at this time in any case. This sequence would be DEV → REG. Because the initiator tasks are always carried out in this order, the requirements of Approach 1 would appear to be satisfied. Unfortunately, this is not the case, as can be seen from Figure 2. After a job step has been completed, CPU control is passed to the routine that allocates devices, so that the devices needed for the next step can be acquired. However, this routine is executed in the region acquired for the previous job step. Thus, the order for acquiring devices and regions has been effectively reversed to REG → DEV, and deadlock can occur. In the final design, regions are allocated before devices, and the sequence for acquiring the three resources is DS → REG → DEV, as shown in Figure 3.

Summary comment

Deadlock problems can arise in many subtle ways in a multitasking system. System designers must be constantly alert to the deadlock possibilities of any proposed design. Users designing application programs in which multitasking capabilities are used face similar problems. Once a deadlock situation has been recognized, it can probably be circumvented by one of the techniques described.

The operating system supervisor could be modified to detect deadlocks after they had actually occurred. However, the added time required to perform this function does not seem justified since a task would have to be terminated nonetheless. The reason that the supervisor cannot anticipate deadlocks is that no means presently exist for communicating to the supervisor the future interlock plans for a task. Theoretically, the supervisor could detect and circumvent deadlocks before they occurred if the future plans for a task were specified. Of course, the supervisor would have to be apprised of all future plans, not just those involving future plans for resource use. Only in this way could a really complete solution to deadlock be achieved.

CITED REFERENCES AND FOOTNOTES

1. Many of the terms and concepts of the SYSTEM/360 Operating System are explained in the three-part article "The functional structure of OS/360" particularly in B. I. Witt's part on "Job and task management," *IBM Systems Journal* **5,** No. 1, 12–29 (1966).
2. *IBM SYSTEM/360 Operating System: Supervisor and Data Management Services* C28-6646-0, Data Processing Division, White Plains, New York.
3. The generation data group presents an added facet to this problem, since concurrent execution of jobs, some or all of which are creating new generations, could cause unpredictable shifts in the generation number base as the jobs were executed.
4. A request from within the job step for access to data sets in addition to those obtained by the initiator is not prohibited. However, such requests must be made with great care, since main storage, devices, and probably other data sets have already been allocated to the job step. Thus, such requests are fraught with deadlock potential.
5. Printers using the universal character set feature are the first devices to present an integrity problem, since they have a loadable buffer that defines the print chain characteristics.
6. The job scheduler has been designed to minimize this operational problem. First, when devices containing passed data sets or volumes that are to be retained are released at the end of a job step, the volumes are not dismounted. Second, the device allocation program has been designed to avoid allocating devices containing passed data sets or retained volumes if they are not to be used in the step for which devices are being allocated. When this cannot be avoided, the affected volumes must be dismounted and later remounted.