This part of the paper discusses the control-program functions most closely related to job and task management.

Emphasized are design features that facilitate diversity in application environments as well as those that support multitask operation.

The functional structure of OS/360

Part II Job and task management

by B. I. Witt

One of the basic objectives in the development of OS/360 has been to produce a general-purpose monitor that can jointly serve the needs of real-time environments, multiprogramming for peripheral operations, and traditional job-shop operations. In view of this objective, the designers found it necessary to develop a more generalized framework than that of previously reported systems. After reviewing salient aspects of the design setting, we will discuss those elements of OS/360 most important to an understanding of job and task management.

Background

Although the conceptual roots of OS/360 task management are numerous and tangled, the basic notion of a task owes much to the systems that have pioneered the use of on-line terminals for inventory problems. This being the case, the relevant characteristics of an on-line inventory problem are worthy of review. We may take the airline seat-reservation application as an example: a reservation request reduces the inventory of available seats, whereas a cancellation adds to the inventory. Because a reply to a ticket agent must be sent within a matter of seconds, there is no opportunity to collect messages for later processing. In the contrasting environment where files are updated and reports made on a daily or weekly basis, it suffices to collect and sort transactions before posting them against a master file.

Three significant consequences of the on-line environment can be recognized:

- Each message must be processed as an independent task
- Because there is no opportunity to batch related requests, each task expends a relatively large amount of time in references to the master file
- Many new messages may be received by the system before the task of processing an older message is completed

What is called for, then, is a control program that can recognize the existence of a number of concurrent tasks and ensure that whenever one task cannot use the CPU, because of input/output delays, another task be allowed to use it. Hence, the CPU is considered a resource that is allocated to a task.

Another major consideration in on-line processing is the size and complexity of the required programs. Indeed, the quantity of code needed to process a transaction can conceivably exceed main storage. Furthermore, subprogram selection and sequence depend upon the content of an input message. Lastly, subprograms brought into main storage on behalf of one transaction may be precisely those needed to process a subsequent transaction. These considerations dictate that subprograms be callable by name at execution time and relocatable at load time (so that they may be placed in any available storage area); they also urge that a single copy of a subprogram be usable by more than one transaction.

The underlying theme is that a task—the work required to process a message—should be an identifiable, controllable element. To perform a task, a variety of system resources are required: the CPU itself, subprograms, space in main and auxiliary storage, data paths to auxiliary storage (e.g., a channel and a control unit), interval timer and others.

Since a number of tasks may be competing for a resource, an essential control program function is to manage the system's resources, i.e., to recognize requests, resolve conflicting demands, and allocate resources as appropriate. In this vein, the general purpose multitask philosophy of the OS/360 control program design has been strongly influenced by task-management ideas that have already been tested in on-line systems.[1] But there is no reason to limit the definition of "task" to the context of real-time inventory transactions. The notion of a task may be extended to any unit of work required of a computing system, such as the execution of a compiler, a payroll program, or a data-conversion operation.

Basic definitions

In the interests of completeness, this section briefly redefines terms introduced in Part I. Familiarity with the general structure of SYSTEM/360 is assumed.[2]

From the standpoint of installation accounting and machine

room operations, the basic unit of work is the *job*. The essential characteristic of a job is its independence from other jobs. There is no way for one job to abort another. There is also no way for the programmer to declare that one job must be contingent upon the output or the satisfactory completion of another job. Job requirements are specified by control statements (usually punched in cards), and may be grouped to form an input job stream. For the sake of convenience, the job stream may include input data, but the main purpose of the job stream is to define and characterize jobs. Because jobs are independent, the way is open for their concurrent execution.

job step

By providing suitable control statements, the user can divide a job into *job steps*. Thus, a job is the sum of all the work associated with its component job steps. In the current OS/360, the steps of a given job are necessarily sequential: only one step of a job can be processed at a time. Furthermore, a step may be conditional upon the successful completion of one or more preceding steps; if the specified condition is not met, the step in question can be bypassed.

task

Whenever the control program recognizes a job step (as the result of a job control statement), it formally designates the step as a *task*. The task consists, in part or in whole, of the work to be accomplished under the direction of the program named by the job step. This program is free to invoke other programs in two ways, first within the confines of the original task, and second within the confines of additionally created tasks. A task is created (except in the special case of initial program loading) as a consequence of an ATTACH macroinstruction. At the initiation of a job step, ATTACH is issued by the control program; during the course of a job step, ATTACH's may be issued by the user's programs.

From the viewpoint of the control system, all tasks are independent in the sense that they may be performed concurrently. But in tasks that stem from one given job (which implies that they are from the same job step), dependency relationships may be inherent because of program logic. To meet this possibility, the system provides means by which tasks from the same job can be synchronized and confined within a hierarchical relationship. As a consequence, one task can await a designated point in the execution of another task. Similarly, a task can wait for completion of a subtask (a task lower in the hierarchy). Also, a task can abort a subtask.

Although a job stream may designate many jobs, each of which consists of many job steps and, in turn, leads to many tasks, a number of quite reasonable degenerate cases may be imagined; e.g., in an on-line inventory environment, the entire computing facility may be dedicated to a single job that consists of a single job step. At any one time, this job step may be comprised of many tasks, one for each terminal transaction. On the other hand, in many installations, it is quite reasonable to expect almost all jobs to consist of several steps (e.g., compile/link-edit/execute) with

no step consisting of more than one task.

In most jobs, the executable programs and the data to be processed are not new to the system—they are carried over from earlier jobs. They therefore need not be resubmitted for the new job; it is sufficient that they be identified in the control statements submitted in their place as part of a job stream. A job stream consists of such control statements, and optionally of data that is new to the system (e.g., unprocessed keypunched information). Control statements are of six types; the three kinds of interest here are *job, execute,* and *data definition* statements.

control statements

The first statement of each job is a job statement. Such a statement can provide a job name, an account number, and a programmer's name. It can place the job in one of fifteen priority classes; it can specify various conditions which, if not met at the completion of each job step, inform the system to bypass the remaining steps.

The first statement of each job step is an execute statement. This statement typically identifies a program to be executed, although it can be used to call a previously cataloged procedure into the job stream. The first statement can designate accounting modes, conditional tests that the step must meet with respect to prior steps, permissable execution times, and miscellaneous operating modes.

A data definition statement permits the user to identify a data set, to state needs for input/output devices, to specify the desired channel relationships among data sets, to specify that an output data set be passed to a subsequent job step, to specify the final disposition of a data set, and to incorporate other operating details.

program structure

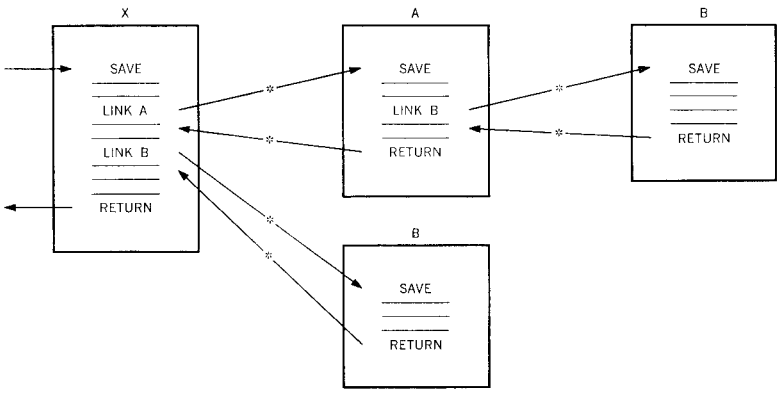
In OS/360, a ready-for-execution program consists of one or more subprograms called *load modules*; the first load module to be executed is the one that is named in the execute control statement. At the option of the programmer, a program can take one of the following four structures:

Simple structure. One load module, loaded into main storage as an entity, contains the entire program.

Planned overlay structure. The program exists in the library as a single load module, but the programmer has identified program segments that need not be in main storage at the same given time. As a consequence, one area of storage can be used and reused by the different segments. The OS/360 treatment of this structure follows the guide lines previously laid down by Heising and Larner.[3] A planned overlay structure can make very effective use of main storage. Because the control system intervenes only once to find a load module, and linkages from segment to segment are aided by symbol resolution in advance of execution, this structure also serves the interest of execution efficiency.

Dynamic serial structure. The advantages of planned overlay tend to diminish as job complexity increases, particularly if the selection

Figure 1



of segments is data dependent (as is the case in most on-line inventory problems). For this situation, OS/360 provides means for calling load modules dynamically, i.e., when they are named during the execution of other load modules. This capability is feasible because main storage is allocated as requests arise, and the conventions permit any load module to be executed as a subroutine. It is consistent with the philosophy that tasks are the central element of control, and that all resources required by a task for its successful performance—the CPU, storage, *and programs*—may be requested whenever the need is detected. In the dynamic serial structure, more than one load module is called upon during the course of program execution. Following standard linkage conventions, the control system acts as intermediary in establishing subroutine entry and return. Three macroinstructions are provided whereby one load module can invoke another: LINK, XCTL (transfer control), and LOAD.

The action of LINK is illustrated in Figure 1. Of the three programs (i.e., load modules) involved, X is the only one named at task-creation time. One of the instructions generated by LINK is a supervisor call (SVC), and the program name (such as A or B in the figure) is a linkage parameter. When the appropriate program of the control system is called, it finds, allocates space for, fetches, and branches to the desired load module. Upon return from the module (effected by the macroinstruction RETURN), the occupied space is liberated but not reused unless necessary. Thus, in the example, if program B is still intact in main storage at the second call, it will not be fetched again (assuming that the user is operating under "reusable" programming conventions, as discussed below).

As suggested by Figure 2, XCTL can be used to pass control to successive phases of a program. Standard linkage conventions are observed, parameters are passed explicitly, and the supervisor

Figure 2

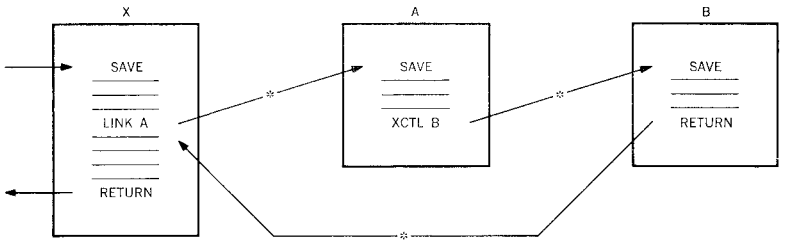
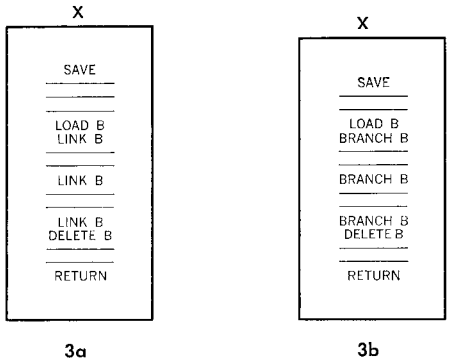


Figure 3a, 3b



functions are similar to those needed for LINK. However, a program transferring via XCTL is assumed to have completed its work, and all its allocated areas are immediately liberated for reuse.

The LOAD macroinstruction is designed primarily for those cases in which tasks make frequent use of a load module, and reusable conventions are followed. LOAD tells the supervisor to bring in a load module and to preserve the module until liberated by a DELETE macroinstruction (or automatically upon task termination). Control can be passed to the module by a LINK, as in Figure 3a, or by branch instructions, as in Figure 3b.

Dynamic parallel structure. In the three foregoing structures, execution is serial. The ATTACH macroinstruction, on the other hand, creates a task that can proceed in parallel with other tasks, as permitted by availability of resources. In other respects, ATTACH is much like LINK. But since ATTACH leads to the creation of a new task, it requires more supervisor time than LINK and should not be used unless a significant degree of overlapped operation is assured.

program usability

Load modules in the library are of three kinds (as specified by the programmer at link-edit time): *not reusable, serially reusable,* and *reenterable.* Programs in the first category are fetched directly

from the library whenever needed. This is required because such programs may alter themselves during execution in a way that prevents the version in main storage from being executed more than once.

A serially reusable load module, on the other hand, is designed to be self-initializing; any portion modified in the course of execution is restored before it is reused. The same copy of the load module may be used repeatedly during performance of a task. Moreover, the copy may be shared by different tasks created from the same job step; if the copy is in use by one task at the time it is requested by another task, the latter task is placed in a queue to wait for the load module to become available.

A reenterable program, by design, does not modify itself during execution. Because reenterable load modules are normally loaded in storage areas protected by the storage key used for the supervisor, they are protected against accidental modification from other programs. A reenterable load module can be loaded once and used freely by any task in the system at any time. (A reenterable load module fetched from a private library, rather than from the main library, is made available only to tasks originating from the same job step.) Indeed, it can be used concurrently by two or more tasks in multitask operations. One task may use it, and before the module execution is completed, an interruption may give control to a second task which, in turn, may reenter the module. This in no way interferes with the first task resuming its execution of the module at a later time.

In a multitask environment, concurrent use of a load module by two or more tasks is considered normal operation. Such use is important in minimizing main storage requirements and program reloading time. Many OS/360 control routines are written in reenterable form.

A reenterable program uses machine registers as much as possible; moreover, it can use temporary storage areas that "belong" to the task and are protected with the aid of the task's storage key. Temporary areas of this sort can be assigned to the reenterable program by the calling program, which uses a linkage parameter as a pointer to the area. They can also be obtained dynamically with the aid of the GETMAIN macroinstruction in the reenterable program itself. GETMAIN requests the supervisor to allocate additional main storage to the task and to point out the location of the area to the requesting program. Note that the storage obtained is assigned to the task, and not to the program that requested the space. If another task requiring the same program should be given control of the CPU before the first task finishes its use of the program, a *different* block of working storage is obtained and allocated to the second task.

Whenever a reenterable program (or for that matter any program) is interrupted, register contents and program status word are saved by the supervisor in an area associated with the interrupted task. The supervisor also keeps all storage belonging to the

task intact—in particular, the working storage being used by the reenterable program. No matter how many intervening tasks use the program, the original task can be allowed to resume its use of the program by merely restoring the saved registers and program status word. The reenterable program is itself unaware of which task is using it at any instant. It is only concerned with the contents of the machine registers and the working storage areas pointed to by designated registers.

Job management

The primary functions of job management are

- Allocation of input/output devices
- Analysis of the job stream
- Overall scheduling
- Direction of setup activities

In the interests of efficiency, job management is also empowered to transcribe input data onto, and user output from, a direct-access device.

In discussing the functions of OS/360, a distinction must be made between job management and task management. Job management turns each job step over to task management as a formal task, and then has no further control over the job step until completion or abnormal termination. Job management primes the pump by defining work for task management; task management controls the flow of work. The functions of task management (and to some degree of data management) consist of the fetching of required load modules; the dynamic allocation of CPU, storage space, channels, and control units on behalf of competing tasks; the services of the interval timer; and the synchronization of related tasks.

schedulers

Job management functions are accomplished by a *job scheduler* and a *master scheduler*. The job scheduler consists mainly of control programs with three types of functions: read/interpret, initiate/terminate, and write. The master scheduler is limited in function to the handling of operator commands and messages to the console operator.

In its most general form, the job scheduler permits priority scheduling as well as sequential scheduling. The *sequential scheduling system* is suggested by Figure 4. A reader/interpreter scans the control statements for one job step at a time. The initiator allocates input/output devices, notifies the operator of the physical volumes (tape reels, removable disks, or the like) to be mounted, and then turns the job step over to task management.

In a *priority scheduling system*, as suggested by Figure 5, jobs are not necessarily executed as encountered in an input job stream. Instead, control information associated with each job enters an input work queue, which is held on a direct-access device. Use of this queue, which can be fed by more than one input job stream, permits the system to react to job priorities and delays caused by

the mounting and demounting of input/output volumes. The initiator/terminator can look ahead to future job steps (in a given job) and issue volume-mounting instructions to the operator in advance.

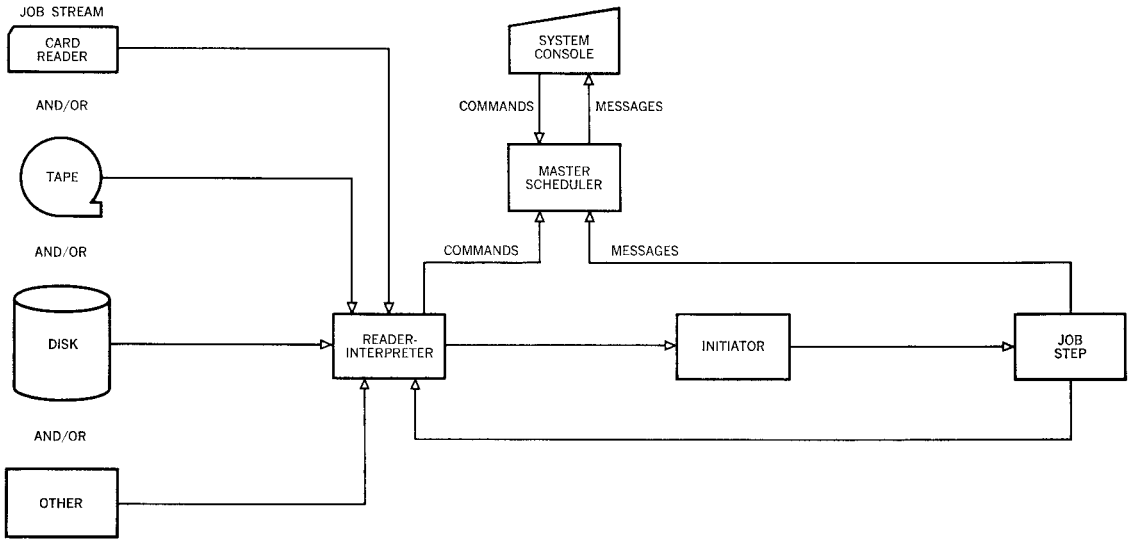
Some versions of the system have the capability of processing jobs in which control information is submitted from remote on-line terminals. A reader/interpreter task is attached to handle the job control statements, and control information is placed in the input work queue and handled as in the case of locally submitted jobs. Output data sets from remote jobs are routed to the originating terminal.

For each step of a selected job, the initiator ensures that all necessary input/output devices are allocated, that direct-access storage space is allocated as required, and that the operator has mounted any necessary tape and direct-access volumes. Finally, the initiator requests that the supervisor lend control to the program named in the job step. At job step completion, the terminator removes the work description from control program tables, freeing input/output devices, and disposing of data sets.

multijob initiation

One version of the initiator/terminator, optional for larger systems where it is practical to have more than one job from the input work queue under way, permits *multijob initiation*. When the system is generated, the maximum number of jobs that are allowed to be executed concurrently can be specified. Although each selected job is run one step at a time, jobs are selected from the queue and initiated as long as (1) the number of jobs specified

Figure 4



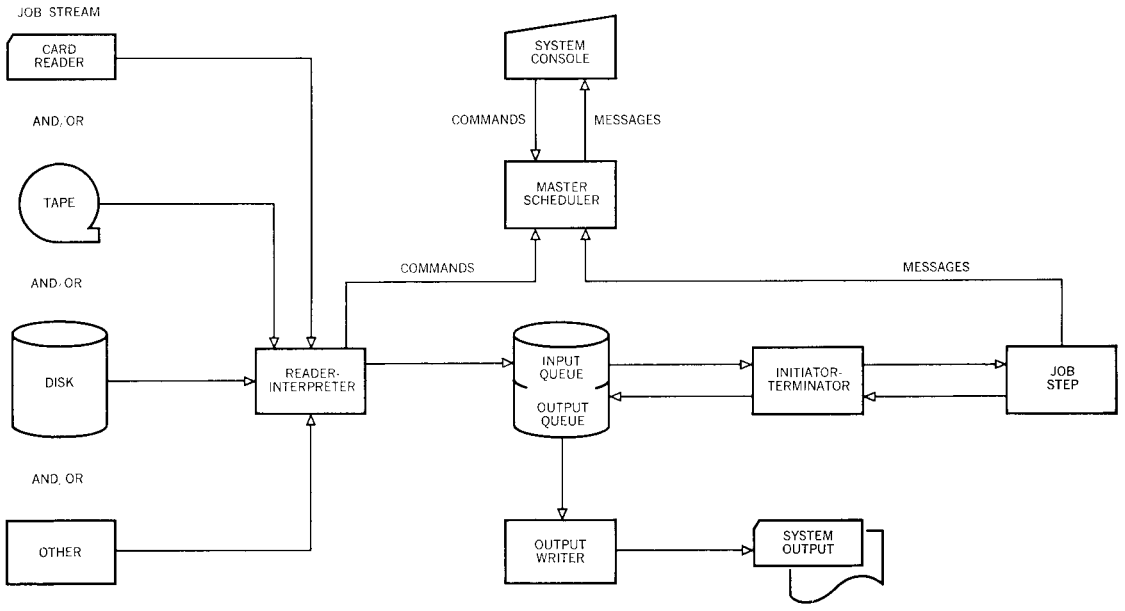
by the user is not exceeded; (2) enough input/output devices are available; (3) enough main storage is available; (4) jobs are in the input work queue ready for execution; and (5) the initiator has not been detached by the operator.

Multijob initiation may be used to advantage where a series of local jobs is to run simultaneously with an independent job requiring input from remote terminals. Typically, telecommunication jobs have periods of inactivity, due either to periods of low traffic or to delays for direct-access seeks. During such delays, the locally available jobs may be executed.

During execution, output data sets may be stored on a direct-access storage device. Later, an output writer can transcribe the data to a system output device (normally a printer or punch). Each system output device is controlled by an output writer task. Moreover, output devices can be grouped into usage classes. For example, a single printer might be designated as a class for high-priority, low-volume printed output, and two other printers as a class for high-volume printing. The data description statement allows output data sets to be directed to a class of devices; it also allows a specification that places a reference to the data on the output work queue. Because the queue is maintained in priority sequence, the output writers can select data sets on a priority basis.

In systems with input and output work queues, the output writer is the final link in a chain of control routines designed to ensure low turn-around time, i.e., time from entry of the work

Figure 5



statement to a usable output. At two intermediate stages of the work flow, data are accessible as soon as prepared, without any requirement for batching; and at each of these stages, priorities are used to place important work ahead of less important work that may have been previously prepared. These stages occur when the job has just entered the input work queue, and when the job is completed with its output noted in the output work queue.

Note that a typical priority scheduling system, even one that handles only a single job at a time, may require multitask facilities to manage the concurrent execution of a reader, master scheduler, and a single user's job.

Task management

As stated earlier, job management turns job steps over to task management, which is implemented in a number of supervisory routines. All work submitted for processing must be formalized as a task. (Thus, a program is treated as data until named as an element of a task.) A task may be performed in either a *single-task* or *multitask* environment. In the single task environment, only one task can exist at any given time. In the multitask environment, several tasks may compete for available resources on a priority basis. A program that is written for the single-task environment and follows normal conventions will work equally well in the multitask environment.

single-task operation

In a single-task environment, the job scheduler operates as a task that entered the system when the system was initialized. Each job step is executed as part of this task, which, as the only task in the system, can have all available resources. Programs can have a simple, overlay, or dynamic serial structure.

The control program first finds the load module named in the EXEC statement. Then it allocates main storage space according to program attributes stated in the library directory entry for the load module, and loads the program into main storage. Once the load module (or root segment, in the case of overlay) is available in main storage, control is passed to the entry point. If the load module fetched is the first subprogram of a dynamic serial program, the subsequent load modules required are fetched in the same way as the first, with one exception: if the needed module is reusable and a copy is already in main storage, that copy is used for the new requirement.

When the job step is completed, the supervisor informs the job scheduler, noting whether completion was normal or abnormal.

By clearly distinguishing among tasks, the control system can allow tasks to share facilities where advantageous to do so.

multitask operation

Although the resource allocation function is not absent in a single-task system, it comes to the fore in a multitask system. The system must assign resources to tasks, keep track of all assignments, and ensure that resources are appropriately freed upon task completion. If several tasks are waiting for the same resource, queuing of requests is required.

Each kind of resource is managed by a separate part of the control system. The CPU manager, called the *task dispatcher*, is part of the supervisor; the queue on the CPU is called the *task queue*. The task queue consists of task control blocks ordered by priority. There is one task control block for each task in the system. Its function is to contain or point to all control information associated with a task, such as register and program-status-word contents following an interrupt, locations of storage areas allocated to the task, and the like. A task is *ready* if it can use the CPU, and *waiting* if some event must occur before the task again needs the CPU.

A task can enter the waiting state directly via the WAIT macroinstruction, or it may lapse into a waiting state as a result of other macroinstructions. An indirect wait may occur, for example, as a result of a GET macroinstruction, which requests the next input record. If the record is already in a main storage buffer area, the control program is not invoked and no waiting occurs; otherwise, a WAIT is issued by the GET routine and the task delayed until the record becomes available.

Whenever the task dispatcher gains control, it issues the Load Program Status Word instruction that passes control to the ready task of highest priority. If none of the tasks are ready, the task dispatcher then instructs the CPU to enter the hardware waiting condition.

By convention, the completed use of a resource is always signaled by an interruption, whereupon the appropriate resource manager seizes control.

Let *subtask* denote a task attached by an existing task within a job step. Subtasks can share some of the resources allocated to the attaching task—notably the storage protection key, main storage areas, serially reusable programs (if not already in use), reenterable programs, and data sets (as well as the devices on which they reside). Data sets for a job step are initially presented to the job scheduler by data definition statements. When the job scheduler creates a task for the job step, these data sets become available to all load modules operating under the task, with no restriction other than that data-set boundaries be heeded. When the task attaches a subtask, it may pass on the location of any data control block: using this, the subtask gains access to the data set.

synchronized events

We have mentioned the ways by which an active task can enter a waiting state in anticipation of some specific event. After the event has occurred, the required notification is effected with the aid of the POST macroinstruction. If the event is governed by the control program, as in the instance of a read operation, the supervisor issues the POST; for events unknown to the supervisor, a user's program (obviously not part of the waiting task) must issue a POST.

A task program may issue several requests and then await the completion of a given number of them. For example, a task may specify by READ, WRITE, and ATTACH macroinstructions that

three asynchronous activities be performed, but that as soon as two have been completed, the task be placed in the ready condition. When each of these requests is initially made to the control program, the location of a one-word *event control block* is stated. The event control block provides communication between the task (which issued the original request and the subsequent WAIT) and the posting agency—in this case, the control program. When the WAIT macroinstruction is issued, its parameters supply the addresses of the relevant event control blocks. Also supplied is a *wait count* that specifies how many of the events must occur before the task is ready to continue.

When an event occurs, a *complete* flag in the appropriate event control block is set by the POST macroinstruction, and the number of complete flags is tested against the wait count. If they match, the task is placed in the ready condition. A *post code* specified in the POST macroinstruction is also placed in the event control block; this code gives information regarding the manner in which completion occurred. After the task again gains control, the user program can determine which events occurred and in what manner.

Requests for services may result in waits of no direct concern to the programmer, as, for example, in the case of the GET macroinstruction previously mentioned. In all such instances, event control blocks and wait specifications are handled entirely by the supervisor.

Another form of synchronization allows cooperating tasks to share certain resources in a "serially reusable" way. The idea (already invoked in the discussion of programs) may be applied to any shared facility. For example, the facility may be a table that has to be updated by many tasks. In order to produce the desired result, each task must complete its use of the table before another task gains access to it (just as each task had to complete its use of a self-initiating program before another task was allowed to use the program). To control access to such a facility, the programmer may create a queue of all tasks requiring access, and limit access to one task at a time. Queuing capabilities are provided by two macroinstructions: enqueue (ENQ) and dequeue (DEQ). The nature of the facility, known only to the tasks that require it, is of no concern to the operating system so long as a queue control block associated with the facility is provided by the programmer. ENQ causes a request to be placed in a queue associated with the queue control block. If the *busy indicator* in the control block is on, the task issuing the ENQ is placed in the wait condition pending its turn at the facility. If the busy indicator is off, the issuing task becomes first in the queue, the busy indicator is turned on, and control is returned to the task. When finished with the facility, a task liberates the facility and posts the next task on the queue by issuing DEQ.

In a multitask operation, competing requests for service or resources must be resolved. In some cases, choices are made by

considering hardware optimization, as, for example, servicing requests for access to a disk in a fashion that minimizes disk seeking time. In most cases, however, the system relies upon a priority number provided by the user. The reason for this is that the user can best select priority criteria. He may reconcile such factors as the identification of the job requestor, response-time requirements, the amount of time already allocated to a task, or the length of time that a job has been in the system without being processed. The net result is stated in a priority number ranging from 0 to 14 in order of increasing importance.

task priority

Initial priorities, specified in job statements, affect the sequence in which jobs are selected for execution. The operator is free to modify such priorities up to the time that the job is actually selected. Changes to priorities may be made dynamically by the change priority (CHAP) macroinstruction, which allows a program to modify the priority of either the active task or of any of its subtasks. Means are provided whereby unauthorized modification can be prevented.

When the job scheduler initiates a job step, the current priority of the job is used to establish a *dispatch priority* and a *limit priority*. The former is used by the resource managers, where applicable, to resolve contention for a resource. The limit priority, on the other hand, serves to control dynamic priority assignments. CHAP permits each task to change its dispatching priority to any point in the range between zero and its limit. Furthermore, when a task attaches a subtask, it is free to set the subtask's dispatching and limit priorities at any point in the range between zero and the limit of the attacher; the subtask's dispatching priority can however exceed that of the attacher. For example, were task A, with limit and dispatching priorities both equal to 10, to attach subtask B with a higher relative dispatching priority than itself, it could use CHAP to lower its own dispatching priority to 7 and attach B with limit and dispatching priorities set to 8.

It is expected that most installations will ordinarily use three levels of priority for batch-processing jobs. Normal work will automatically be assigned a median priority. A higher number will be used for urgent jobs, and a lower one for low-priority work.

Normally, programs are expected to signal completion of their execution by RETURN or XCTL. If the program at the highest control level within the task executes a RETURN, the supervisor treats it as an end-of-task signal. Whenever RETURN is used, one of the general registers is used to transmit a return code to the caller. The return code at task termination may be inspected by the attaching task, and is used by the job scheduler to evaluate the condition parameters in job control statements. It may, for example, find that all remaining steps are to be skipped.

task termination

In addition, any program operating on behalf of a task can execute a macroinstruction to discontinue task execution abnormally. The control program then takes appropriate action to liberate resources, dispose of data sets, and remove the task con-

trol block. Although abnormal termination of a task causes abnormal termination of all subtasks, it is possible for abnormal subtasks to terminate without causing termination of the attaching task.

main storage allocation

The supervisor is designed to allocate main storage dynamically, when space is demanded by a task or the control program itself. An *implicit* request is generated internally within the control program, on behalf of some other control program service. An example is LINK, in which the supervisor finds a program, allocates space, and fetches it. To make *explicit* requests for additional main storage areas, a user program may employ the GETMAIN or GETPOOL macroinstructions.

Also provided are means for dynamic release of main storage areas. Implicit release may take place when a program is no longer in use, as signaled by RETURN, XCTL, or DELETE. Explicit release is requested by the FREEMAIN or FREEPOOL macroinstructions.

Explicit allocation by GETMAIN can be for fixed or variable areas, and can be conditional or unconditional:

- *Fixed area.* The amount of storage requested is explicitly given.
- *Variable area.* A minimum acceptable amount of storage is specified, as well as a larger amount preferred. If the larger amount is not available, the supervisor responds to the request with the largest available block that equals or exceeds the stated minimum.
- *Conditional.* Space is requested if available, but the program can proceed without it.
- *Unconditional.* The task cannot proceed without the requested space.

storage protection

The operating system uses the SYSTEM/360 storage protection feature to protect storage areas controlled by the supervisor from damage by user jobs and to protect user jobs from each other. This is done by assigning different protection keys to each of the job steps selected for concurrent execution. However, if multiple tasks result from a single job step (by use of the ATTACH macroinstruction), all such tasks are given the same protection key to allow them to write in common communication areas.

Each job step is assigned two logically different pools, each consisting of one or more storage blocks. The first of these is used to store non-reusable and serially-reusable programs from any source, and reenterable programs from sources other than the main library. This pool is not designated by number. The second pool, numbered 00, is used for any task work areas obtained by the supervisor and for filling all GETMAIN or GETPOOL requests—unless a non-zero pool number is specified.

When the highest-level task of a job step is terminated, all storage pools are released for reassignment. However, when a task attaches a subtask, and makes storage areas available to the

subtask, it may suit the purposes of the task not to have the storage areas released upon completion of the subtask. To provide for this possibility, programs may call for the creation of pools numbered 01 or higher. Such a pool may be made available to a subtask in either of two ways—that is, by *passing* or *sharing*. If a pool created by a task is passed to a subtask, termination of the subtask results in release of the pool. On the other hand, subtask termination does not result in the release of a shared pool. In both cases, the subtask that receives a pool may add to the pool, delete from it, or release it in the same way as the originating task.

passing and sharing

Because Pool 00 refers to the same set of storage blocks for all tasks in a job step, it need not be passed or shared, and is not released until the job step is completed.

If two or more job steps are being executed, and one requires more additional main storage than is available for allocation, the control program intervenes. First, the supervisor attempts to free space occupied by a program that is neither in use nor reserved by a task. Failing that, it may suspend the execution of one or more tasks by storing the associated information in auxiliary storage. The storage and retrieval operations occasioned by competing demands for main-storage space are termed *roll-out* and *roll-in*.

roll-out and roll-in

The decision to roll out one or more tasks is made on the basis of task priorities. A main storage demand by a task can cause as many lower-priority tasks to be rolled out as necessary to satisfy the demand. If the lowest-priority task in the system needs more space to continue, it is placed in a wait state pending main storage availability.

During roll-out, all tasks operating under a single job step are removed as a group. Input/output operations under way at the time of the roll-out are allowed to reach completion.

Roll-in takes place automatically as soon as the original space is again available, and execution continues where it left off. Since its task control block remains in a wait status and its input/output units are not altered, a task may still be considered in the system after roll-out.

Significance of multitask operations

It may be expected that multitask operations will not only provide powerful capabilities for many existing environments, but will also serve as a foundation for more complex environments for some time to come.

Fast turnaround in job-shop operations is achieved by allowing concurrent operation of input readers, output writers, and user's programs. It is possible to handle a wide variety of telecommunication activities, each of which is characterized by many tasks (most of them in wait conditions). Also, complex problems can be programmed in segments that concurrently share system resources and hence optimize the use of those resources. With some versions of the job scheduler, multitask operations permit

job steps from several different jobs to be established as concurrent tasks. To serve such current multitask needs, the structure of the control system consists of two primary classes of elements: (1) queues representing unsatisfied requirements of tasks for certain resources, and (2) tables identifying available resources. Some of the control information is in main storage; some is in auxiliary storage. This structure facilitates dynamic configuration changes, such as addition or removal of programs in main storage, and attached input/output devices.

Perhaps more important for future systems, the structure may prove adaptable in the management of additional CPU's. For example, if multiple CPU's were given access to the job queue (now stored on a disk), each CPU could queue new jobs as well as initiate jobs already on the queue. Similarly, if multiple CPU's were given access to main storage, each CPU could add tasks to the task queue and dispatch tasks already on the task queue. That is, a system could be designed wherein, by executing the task-dispatcher control routine (which itself is in the shared main storage), any CPU could be assigned a ranking task on the queue; and while executing a task, any CPU could add new tasks to the queue by means of the ATTACH macroinstruction.

Summary

In OS/360, for which the basic unit of work is the task, resources are allocated only to tasks. In general, resources are allocated dynamically to permit easier planning on the part of the programmer, achieve more efficient utilization of storage space, and open the way for concurrent execution of a number of tasks.

Users notify the system of work requirements by defining each job as a sequence of job-control statements. The number of tasks entailed by a job depends upon the nature of the job. The system permits job definitions to be cataloged, thereby simplifying the job resubmittal process. Reading of job specifications and source data, printing of job results, and job execution can occur simultaneously for different jobs. Job inputs and outputs may be queued in direct-access storage, thereby avoiding the need for external batching and permitting priority-governed job execution. In its multijob-initiation mode, the system can process a number of jobs concurrently.

CITED REFERENCES AND FOOTNOTE

1. An historic review of operating systems, with emphasis on I/O control and job scheduling, appears in Reference 4. Operating systems that provided for multiprogramming are described in References 5, 6, and 7. One on-line inventory application is described in Reference 8, and some indication of techniques used in its solution are given in References 9 and 10.
2. G. A. Blaauw and F. P. Brooks, Jr., "The structure of SYSTEM/360: Part I—outline of the logical structure," *IBM Systems Journal* 3, No. 2, 119–135 (1964).
3. W. P. Heising and R. A. Larner, "A semi-automatic storage allocation

system at loading time," *Communications of the ACM* **4**, No. 10, 446–449 (October 1961).
4. T. B. Steel, Jr., "Operating systems," *Datamation* **10**, No. 5, 26–28 (May 1964).
5. E. S. McDonough, "STRETCH experiment in multiprogramming," *Digest of Technical Papers*, ACM 62 National Conference, 28 (1962).
6. E. F. Codd, "Multi-programming," *Advances in Computers*, Volume 3, Edited by Franz L. Alt and Morris Rubinoff.
7. G. F. Leonard, "Control techniques in the CL-II Programming System," *Digest of Technical Papers*, ACM 62 National Conference, 29 (1962).
8. M. N. Perry and W. R. Plugge, "American Airlines SABRE electronic reservation system," WJCC *Proceedings*, 593–601 (May 1961).
9. M. N. Perry, "Handling of very large programs," *Proceedings of IFIP Congress 65*, Volume 1, 243–247 (1965).
10. W. B. Elmore and G. J. Evans, Jr., "Dynamic control of core memory in a real-time system," *Proceedings of IFIP Congress 65*, Volume 1, 261–266 (1965).