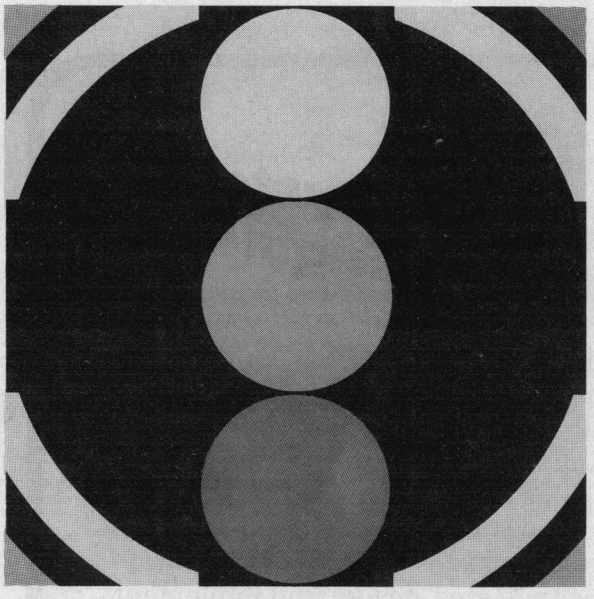


Concurrency in Operating Systems

J. William Atwood
Concordia University

Introduction

As general-purpose computing systems become larger and more complex, it has become uneconomical to restrict their use to a single user or program. This has introduced the necessity of finding ways to safely share them.

Sharing may be cooperative or competitive. It implies that mechanisms are present to permit the objects (e.g., programs) which are sharing the computer to communicate with each other, in order to coordinate their sharing. Sharing also requires adherence to certain rules of behavior which guarantee that correct interaction occurs.

In this paper we will examine the idea of concurrency as a mechanism for sharing a computing system and discuss two operating systems which implement concurrency through the use of *ad hoc* mechanisms. We will then introduce some more formal mechanisms for implementing and controlling interaction—mechanisms leading to proposals for language constructs which simplify the management of concurrency. These constructs also permit the formalization of the rules of behavior, and thus the introduction of mechanisms for enforcing correct interaction.

The need for concurrency

Early computing systems exhibited no concurrency of operation between computations and input/output operations. Typically a single program ran on the central processor, and computation was suppressed while interaction with the environment (I/O) was being effected. This represented a substantial inefficiency in the use of the processor's computing capability, since useful computation was not being performed during I/O, even though the processor was actively testing for completion of the interaction.

Subsequent introduction of the asynchronous data channel allowed the activities of computation and I/O to proceed in parallel. However, it was the responsibility of the (single) program to make use of this available concurrency by, for example, requesting input prior to the time when it was actually required. It was difficult or unnatural to write a program in which more than one or two I/O operations would proceed concurrently with the computation. Since the number of possible *sequences* of interaction was small, the problem of coordination of environmental interaction with internal computation was reducible to an enumeration of the possible cases, and *ad hoc* solutions could be formulated.

Keeping the level of concurrency down to two or three parallel operations represents an under-utilization of available resources. The variety of programs processed by a general-purpose computing system is such that if one program requires a certain set of resources, especially peripherals, other programs will require different sets. A computing system configured to meet or exceed the union of resource demands over all programs is likely to exhibit low average resource utilization.

If mechanisms can be found to increase the average utilization of the system resources, then a more cost-effective system will result, as long as the overhead in managing the mechanism does not exceed the gain in productivity. One such mechanism is to perform more than one computation at a time, providing that two programs (computations) can be found whose total resource requirements are not greater than the total available on the system. If this can be achieved for even part of the time, average utilization will rise. A side benefit is that greater throughput is achieved—i.e., more programs can be processed per unit time.

To do this, it is necessary to introduce sharing of the central processor among several programs: each program will be permitted to utilize the central processor for a period of time (for example, until it requires I/O), at which point the central processor will be allocated to another program. Thus we may say that the central processor is *time-multiplexed* among several programs. This will introduce *apparent* concurrency of programs on the processor,

and may raise the level of *actual* concurrency present in the I/O hardware.

When several programs are simultaneously active, they can no longer be permitted to access I/O equipment such as card readers directly, since the ordering of their requests for individual input records is, in general, unknown. A new "program" must be introduced (and allowed to access the central processor) whose function is to copy card images to a disk unit for subsequent (random) access. This program is, in effect, a "card reader manager."

A second reason for time-multiplexing of a processor is to allow for interactive use of the computing system. Interactive computation is characterized by relatively long periods of time (programmer think time) wherein neither the processor nor the peripherals (with the possible exception of the interactive terminal itself) are being used, followed by short periods of computation. Introduction of this mode of operation (which is very convenient for the individual user) implies sharing of the central processor among the several programs associated with the interactive users.

The process concept

The programs that participate in the sharing of the central processor are, in general, independent units, which could all proceed in parallel if enough separate central processors were available. (In fact, the single central processor is time-multiplexed among these independent units according to some strategy.) These independent units are called *sequential processes.* *

Dijkstra[2,3] has suggested that concurrent activity on one (or more) processors can be modeled by a set of processes which alternate between independent activity and periods of communication.

The concept of a community of n processes sharing a single processor may be immediately generalized to a community sharing multiple processors. The strategy for time-multiplexing must then select m processes to be executed rather than only one.

Process interaction

The restrictions on the number of available resources (e.g., tape drives, card readers) in a computing system mean that one process must necessarily influence the action of others, as it competes for resources. The need for specialization of function (e.g., card reader manager vs. "user" process consuming data) also implies that processes must, at times, cooperate to achieve the desired performance of the computing system.

Process cooperation. As mentioned in the Introduction, process cooperation implies that communication is taking place between the processes. If two processes are going to cooperate by communicating with each other, it is clear that their address spaces must overlap in some way—i.e., provision must be made for both processes to access the "message." It is apparent that confusion may result if both processes attempt to access (and especially to update) the shared address space at the "same" time. (On a single processor this means in an interleaved fashion. On a multiple processor system, access may in fact be simultaneous.) This problem is typical of the timing-dependent errors which may occur in the interaction between concurrent processes. Later we will examine rules and language constructs which permit the prevention of these classes of errors.

Typical examples of process cooperation involve two processes which bear a producer/consumer relationship to each other. This often occurs in the context of I/O—e.g., a card reader manager process and a user process.

Process competition. In systems with multiple concurrent processes, the presence of resources such as unit record peripherals and tape drives, which must not be used simultaneously by several processes (if program operation is to be correct), introduces the requirement for exclusive access to these devices. This requirement may also be imposed on shared objects such as disk files during periods of updating.[7] Processes desiring exclusive access to a resource must compete for it.

The same competition arises concerning access to what may be called *virtual* resources—e.g., tables recording the system state, communications buffers between two cooperating processes. Since it must be guaranteed that both processes do not access the buffer simultaneously, exclusive access to the buffer must be ensured. This exclusiveness of access is called *mutual exclusion* between processes.

It is important to note that resource allocation (request for mutual exclusion on the use of a resource) implies communication of the desire to reserve or release the resource. Similarly, process cooperation (e.g., communication) requires mutual exclusion at a lower level (e.g., accessing the shared message buffer). Cooperation and competition may both be implemented if a mechanism is provided for process coordination or synchronization.

Process Scheduling. The above requirements for process cooperation and competition have obvious implications on short- and medium-term scheduling of the central processor.[8] If a desired resource or object is not available, the process expressing the desire to use it must be prohibited from further activity until it becomes available. For example, a process which requires a tape drive must suspend operation (temporarily) if none is available.

There are two extreme levels of exclusiveness here—requirements for deferral of access to a data structure (virtual resource), which may often be of short duration; and requirements for a (perhaps substantial) delay until a real resource becomes available. If the delay is short, it is not worthwhile to shift the attention of the processor from the process which is running on it to another process. If the delay exceeds the time required to switch the processor, the ability to shift attention may be vital to efficient utilization of the processor.

Processor sharing. The sharing of the central processor is achieved by placing the several processes together in central memory and providing a mechanism for rapidly switching the attention of the processor from one process to another.

In the case of interactive computing, an additional level of complexity is introduced, since the processes which are currently not making use of the processor may have been placed in memory image form on secondary storage and must therefore be loaded into central memory when computation resumes. As long as this loading time is negligible with respect to the cycle time of the think-compute cycle, the system will operate effectively—i.e., the phrase "rapidly switching" still applies.

*A complete discussion of the concept of a process may be found in Horning and Randell.[1] Early use of the process concept appears in Dijkstra,[2,3] Morris and Detlefsen,[4] and Brinch Hansen.[5] A more recent discussion is presented in Presser.[6]

Responsibility for managing process scheduling, processor multiplexing, and other related functions is normally assigned to a separate *control program.*[8] The control program may be co-resident in central memory with the several processes, or it may reside on separate processor(s) dedicated to the control program function. It may itself be a process, a community of processes, or a collection of control program functions which are invoked by user processes.

The state of a processor (and thereby the state of the active process) is defined by the set of processor registers. *Context switching* is the swapping of the contents of these registers for another register image. If this register image represents the (previously saved) state of another process, then the attention of the processor is switched from one process to another when a context switch takes place.

Context switching between processes is typically invoked when the active process is unable to proceed further without requesting a service provided by the control program, such as I/O, or when a process requires cooperation from another process. In addition, a context switch may occur when the user and control programs coexist on the same processor, and a control program function is invoked by the environment.

Sharing of the central processor introduces three subordinate problems: (1) protection of the resources of one program from willful or accidental damage by other programs; (2) provision for communication among programs, and between user programs and the control program; and (3) allocation of resources among processes so that resource demands can always be fulfilled.

Protection. One process can be protected from another through a variety of mechanisms. Essentially the goal of protection is the achievement of inviolate firewalls between any two processes. The requirement is absolute —i.e., it must be highly reliable. Efficiency considerations therefore imply a hardware solution, although purely software schemes are possible in special cases.[9] Typical hardware solutions involve some sort of key and lock mechanism, coupled with a privileged state for the control program.

We will not explore the protection problem any further here. An excellent discussion may be found in Saltzer and Schroeder.[10]

Communication. Communication among processes implies provision of (controlled) mechanisms for circumventing the protection features of a computing system. For example, although the address spaces of two processes are normally separate, they must share access to the buffer in which the message will be placed. It is the role of the communications mechanism to verify that objects desiring to communicate with each other have the right to do so, and that they do it in permitted ways.

Resource allocation. If two or more processes wish to access a resource (or a pool of identical resources) within a computing system, it is necessary to allocate the resources in such a way that the total resources of the system are not exceeded. In addition, if it is possible for a process to acquire a portion of the resources that it requires, and then subsequently make a request for more, it is necessary to ensure that future demands can always be satisfied. As an example, suppose that a computing system has one card reader and one printer. Process 1 requests the card reader and process 2 requests the printer. If processes 1 and 2 subsequently request the printer and card reader, respectively, then the system is in a state where two processes are blocked indefinitely. This situation is called *deadlock* or *deadly embrace,* and methods for detecting and preventing it are thoroughly discussed in Holt.[11] The point here is that although communication is necessary to achieve resource allocation, correct communication is not sufficient to guarantee correct (deadlock-free) resource allocation.

Two example systems

In this section we will examine the architecture of two large-scale computing systems, and the basic facilities provided by their control programs to support concurrency. The examples are chosen to illustrate the impact that hardware architecture can have on the design of the control program software.

CDC 6000 and CYBER Series: Kronos

System architecture. The CDC 6000, CYBER 70, and CYBER 170 series architecture[12,13] consists of one (in some cases two) large central processors assisted by up to 20 peripheral processors. The central processor is well equipped for arithmetic calculation, but it is not capable of explicitly communicating with its environment (e.g., it has no I/O instructions in its instruction set). The peripheral processors, on the other hand, are designed to handle interaction with the environment through a number of I/O channels which may all be accessed by any peripheral processor. The central memory is accessible to both the central and peripheral processors. Each peripheral processor has a small local memory which is inaccessible to any other processor.

Control program structure. The control program (Kronos[14]) may be viewed as a collection of cooperating processes, augmented by a number of "monitor" functions which are shared by these processes. The control program processes run on the peripheral processors. Certain monitor functions (e.g., scheduling, accounting, disk track allocation) which require substantial arithmetic or logical calculation are run on the central processor, at the request of a peripheral processor. These functions, plus certain functions provided for user processes (see below), are known collectively as CPUMTR. The remaining shared functions, which usually involve coordination of potentially conflicting control program process activity, are provided by a process called MTR, which runs on a dedicated peripheral processor.

One other peripheral processor is dedicated to a specific function (process): control of the master console (DSD). The remaining peripheral processors form a pool for the execution of other control program processes. Scheduling of the activity of these peripheral processors is the responsibility of a peripheral processor allocator function (part of CPUMTR), which executes on the central processor.

Context switching. The central processor of all CYBER processors has two operating states: *monitor* mode and *program* mode.

Peripheral processors request execution of central processor functions by issuing a MONITOR EXCHANGE JUMP (MXN) instruction. This forces the central processor to being execution in monitor mode, in response to the function request. When the service has been completed, the requesting peripheral processor is informed (see the section on communication, below). The issuing peripheral processor will normally wait while the function is in progress.

If a peripheral processor completes its current activity, it will use MXN to invoke the allocator function on the central processor, so that the peripheral processor may be reassigned.

While the central processor is in monitor mode, other requests for entry into monitor mode will not be honored—i.e., monitor mode activation may be viewed as a resource to which mutual exclusion of access is applied.

User programs. User programs run as processes on the central processor(s). These processes compete for access to a *control point,* which represents the right to be in central memory. Scheduling of processes to control points is the responsibility of the job scheduler (1SJ), which runs on a pool peripheral processor.

A user program requests control program services by issuing a CENTRAL EXCHANGE JUMP (XJ) instruction. This causes the central processor to begin execution in monitor mode—i.e., CPUMTR is activated. If the request is for a central processor (CPUMTR) function, it is serviced. If the request is for activation of a peripheral processor process, a pool processor is assigned, if available; otherwise the request is queued. Note that user programs may not initiate peripheral processor activity directly, but must ask CPUMTR to do it for them.

Certain CPUMTR functions are quite lengthy. They are executed in program mode by a special system control point—i.e., the monitor mode exclusion is released. This system control point has highest priority, so these program mode routines are guaranteed first access to the central processor when the monitor mode is exited.

Return from monitor mode to program mode occurs when a CPUMTR routine running in monitor mode issues an XJ instruction. Return is always to the highest priority waiting control point. Multiprogramming of user programs is achieved when a control point explicitly vacates the central processor by requesting a control program function, when the real time clock manager (part of MTR) forces the central processor into monitor mode on completion of a prescribed time interval, or on completion of a peripheral processor function, as described above.

Process communication. Communication between a central process running at a control point and CPUMTR is achieved through a message buffer (word one of the central memory segment assigned to the control point). Requests for control program services are made by placing a three-character name in the message buffer, and then issuing XJ. Additional parameters or addresses of parameters may be in the rest of the message buffer.

Communication between CPUMTR and a peripheral processor occurs through two message buffers in the low addresses of central memory—an "input register" and an "output register."

When a pool processor is to be assigned (e.g., as a result of a user program request for a control program service), CPUMTR will place the three-character name of the required process in the input register of the assigned peripheral processor. The pool processor, which has been periodically examining its input register, will locate and load the named process, and begin execution of it.

If a peripheral processor requires CPUMTR services, it will place the function code of this service in its output register and issue MXN to activate CPUMTR. Additional parameters may be placed in a 6-word central memory area adjacent to the output register. Completion of the service is indicated when CPUMTR clears the leading 12 bits of the output register to zero. (Results may be returned in the remaining 48 bits or in the 6-word area.)

Completion of peripheral processor activity is indicated by requesting the CPUMTR function DROP PERIPHERAL PROCESSOR, in which case both the input and output registers are cleared by CPUMTR, and the peripheral processor enters its idle loop, waiting for reallocation.

If a peripheral processor requires MTR services, it will use the output register as for CPUMTR services, except that it will not issue the MXN. Since there is no mechanism for a peripheral processor to force a context switch in another peripheral processor, requests for MTR services must be "noticed" by MTR as the result of a periodic scan of all the output registers. If MTR notices a nonzero output register requesting a service which it does not provide (e.g., a CPUMTR function), it will ignore the request. Completion of the service is again indicated by clearing of the leading 12 bits of the output register.

On some 6000 series processors, the MXN/XJ instructions are not implemented, and the monitor/program mode distinction is not made. In this case, MTR must react to *all* output register requests and pass CPUMTR function requests to CPUMTR using the EXCHANGE JUMP (EXN) instruction. This forces an unconditional context switch on the central processor and activates CPUMTR. (The mutual exclusion on the activation of CPUMTR is thus implemented by funneling the requests for activation through a single peripheral processor.)

A primary function of MTR is thus to periodically examine the message buffers that it shares with other processes, to see if a request has been made or a service has been completed. Since MTR can only examine (and respond to) one request at a time, the problems of timing-dependent errors due to "simultaneous" requests disappear. In effect, correct interaction is ensured by requiring all potentially conflicting communication to take place through MTR and defining a protocol to be followed for accessing the message buffers.

Event notification. Two types of control program services are available to a user process: those which can or must be completed before the user process can proceed (e.g., inquiry as to time of day), and those which may admit parallelism (e.g., buffered-ahead request for I/O). In the first case, completion of the service is indicated by zeroing of the leading 48 bits of the message buffer. In the second case, completion of the *initiation* of the service is indicated by the above mechanism, and notification of completion of the function is achieved by setting the rightmost bit of a central memory word designated by the requesting process, which can be periodically checked by that process. Alternatively, the process can request the CPUMTR function RECALL, or it can ask for recall along with the function request. This will cause it to be quiesced until CPUMTR detects that the designated bit has been set. Safety is ensured by the fact that the central process must clear the bit prior to invocation of the function and will do nothing but examine it until it is subsequently changed. (This is another example of a protocol to be followed to achieve correct interaction.) Protection against the case where the central process does not clear the bit first must be provided by the control program routines at the time the function is invoked.

Subordinate control points. Recent versions of Kronos permit the creation of a subordinate control point within a control point. A program running at a subordinate control point has some of the characteristics of a child process. However, it operates in an extremely limited environment. Activation of a subordinate control point causes the father process (running at the control point) to quiesce. All subordinate control point requests to CPUMTR (the subordinate control point makes requests using the same message buffer mechanism as a

control point) are trapped by CPUMTR, and cause reactivation of the control point, rather than provision of the requested service. The father process must then interpret the request, and reissue it to CPUMTR if a control program function is actually required.

An example of the use of subordinate control points is the Tranex (Transaction Executive) subsystem of Kronos.[15] Each subordinate control point represents a transaction being performed for a (probably interactive) user. The Tranex system, running at a control point, is responsible for I/O to terminals and files, memory management within the control point memory allocation, and time slice allocation among the subordinate control points. It regains control from the subordinate control point each time I/O is required, or when an allocated time slice has been depleted. The subordinate control point is an example of a set of restrictions on process activity which ensure that timing-dependent interactive errors do not occur but which still permit a program to be run in a supervised environment.

Resource allocation. Allocation of certain system resources is achieved by funneling requests through MTR. For example, if two peripheral processors attempt to activate a single I/O channel at the same time, both will deadlock. The right to access I/O channels is granted only by MTR. In other cases, exclusive access will be granted by MTR to the table (shared variable) in which resource allocations are recorded. This implementation of mutual exclusion thereby guarantees that in no case do two (or more) peripheral processes simultaneously allocate a resource requiring exclusive access. Note, however, that there is no enforcement of these mechanisms at the system level of programming. The programs are all written in assembler language and the mechanisms are just software conventions that are adhered to by the system designers. Any enforcement of the mechanisms must be through run-time checking.

IBM 360/370 Series: OS

System architecture. The IBM 360/370 Series architecture[16,17] normally consists of a single central processor assisted by a number of extremely specialized processors called I/O channels, which are capable of executing programs of I/O commands (channel command words). These channels are activated when an I/O instruction is performed on the central processor. IBM channels do not have the general-purpose calculation capability of a CDC peripheral processor. As a result, many of the computations related to I/O, which are done in the peripheral processor of a CDC system, must be done on the central processor of an IBM system prior to the issuing of the I/O instruction. Central processor and channel programs are both stored in the commonly accessible central memory.

Context switching. All 360/370 central processors have two operating states: a *supervisor* state, in which all instructions are available, and a *program* state, in which instructions that communicate with the environment and those that alter the state of the processor are forbidden. [One exception is the instruction SUPERVISOR CALL (SVC) which invokes the control program.]

Context switching begins as the result of SVC, as the result of a programming error or machine error being detected by the hardware, or as the result of action being completed in the environment causing an I/O or external interruption. In each case the processor is switched to supervisor state, the external and I/O interrupts are disabled, and a portion of the control program is activated. The activated routine must complete the context switch by storing the central processor registers in central memory. The control program may thus be said to be interrupt-driven: each service is implemented as an effective procedure call on a function provided by the control program. Completion of a function is normally indicated by execution of the LOAD PROGRAM STATUS WORD instruction, which forces a context switch (and typically returns the processor to program state).

The context switching (interrupt) mechanism on the IBM 360/370 Series is similar to the central exchange jump/monitor exchange jump feature on the CDC CYBER, except that certain classes of interrupt source can be suppressed while allowing others to proceed.

Control program structure. The control program (OS[18]) is a collection of monitor functions which run in supervisor state on the central processor. The functions are divided into four classes. Class I functions are resident in central memory. They may not invoke other control program functions using the SVC mechanism, although other functions may be invoked as procedures. Class II functions are also resident in central memory, but may invoke Class I functions through SVC. Classes III and IV are nonresident. They are loaded into *transient areas* in central memory on demand. They may request Class I and II functions through the SVC mechanism. All SVC functions begin execution in supervisor state, with I/O and external interrupts disabled, but they may choose to release any of these exclusions. Control program functions are always invoked by explicit request (interrupt). This interrupt may come from a central process; a Class II, III, or IV control program function; or from the environment (e.g., completion of I/O activity).

The presence of the single (general purpose) processor means that all process concurrency is only apparent. Concurrency is little used within the control program itself, the preferred method of operation being to disable all possibility of externally-forced context switching while the control program is running.

User programs. The user programs run as processes, in the program state, on the central processor. These processes are called *tasks* by IBM. A *region* is a contiguous area of central memory allocated to a task.

Process communication. Communication between a user task and the control program is achieved by placing parameters in the processor registers and requesting context switching by issuing the SUPERVISOR CALL instruction. (This same mechanism is, of course, used by Class II, III, and IV functions when invoking SVC.)

Substantial use is made of the fact that actual concurrency is not present. Therefore, the problems of simultaneous updating of shared variables are reduced. In effect, correct interaction is ensured by requiring all control program activity to take place strictly in sequence.

Event notification. As with Kronos, two types of control program services are available to a user task: those which can be completed quickly (i.e., before the next instruction after SVC is executed) and those which admit or require parallelism. In the first case, completion of the operation is assumed when the user task resumes. In the second case, notification of the completion of actions performed by the control program is signalled by the use of a control program function called POST, which sets the leftmost bit of a central memory word (EVENT CONTROL BLOCK) designated by the requesting task

and checks to see if any task is waiting on the event. Waiting for an action to complete is achieved by invoking the function WAIT. The ECB is also used to contain an address identifying the waiting task when there is one. Use of the ECB as a signalling mechanism is subject to the same criticism as that mentioned earlier in the context of process communication for the CDC 6000: protection against inadvertent error must be provided by the system functions when invoked. For example, setting an ECB to zero immediately *after* requesting a service that uses an ECB to report completion, and then invoking WAIT, may result in task deadlock if the service was in fact completed before the task regained control after issuing the SVC.

Subtasks. OS permits a task to create a subtask within the central memory region allocated to the task. This subtask may proceed independently of the originating task, is permitted to issue its own SVC requests, and competes with all other tasks for system resources, such as central processor time, in accordance with its allocated priority.

Communication between a task and its subtask is implemented using the POST/WAIT functions mentioned above, either explicitly, or implicitly at subtask termination (the control program can be requested to issue POST to a designated ECB, when the subtask terminates).

Resource allocation. Physical resources are allocated by control program procedures which rely for their correct operation on the fact that externally-forced context switching is inhibited. Where mutual exclusion is required on shared resources such as disk files, which are being updated by various tasks, two control program functions called ENQUEUE and DEQUEUE are provided. Normally, if a task requests an ENQUEUE function on a certain resource, and it is not available, the task is delayed until the resource becomes available. However, it is possible to request control only if it will be immediately granted. Requested control may be exclusive (no other task permitted to access the resource while control is held), or shared (always granted unless another task currently has, or is waiting to establish, exclusive control). The resource is represented by a pair of identifiers supplied as parameters to the ENQUEUE or DEQUEUE function, and no protection is provided against inadvertent use, at the system-wide level, of the same pair of identifiers by two different tasks. For example, if a user program uses the identifiers, which are used by control program functions to indicate enqueuing on the volume table of contents of a disk volume, then it will cause deadlock if it attempts to open any file on that volume.

Process coordination

In previous sections we have examined some concepts related to process coordination and resource management, and two implementations which have formulated rules to guarantee correct interaction between processes. These rules are extremely dependent on the architecture of the system for which they were formulated, and they rely on the careful following of conventions.

There is a need for formalisms to describe these concepts, in ways which are amenable to *a priori* demonstration of the correctness of the implementation. Use of these formalisms leads to proposals for language constructs which aid in their correct use.

In this section we will present a formalism for process coordination, and examine its utility in the control of concurrency in operating systems. In the following section we will examine two concepts that are more useful for resource management, and the language constructs that may be derived from them.

Critical sections. In the CDC 6000 and CYBER series, any peripheral processor can access any physical I/O channel. In order to ensure that no two peripheral processors access a channel at the same time, it is necessary for a process running on a pool peripheral processor to request access rights to the channel (from MTR) prior to use, and then release it immediately after use.

This is a typical pattern for use of a resource which must be allocated to only one process at a time:

establish exclusive access;
use the resource;
release exclusion.

The section of program text during which the necessary exclusive access holds is called a *critical section.*

Establishment of exclusive access. When establishment of exclusive access is being attempted, the requesting process must be delayed if the required resource is not currently available. Similarly, releasing the exclusion may result in the resumption of a waiting process.

In Kronos, the process running on the pool processor, which has requested exclusive access to a resource, delays itself by periodically testing for an answer to its request until the request is satisfied. On a shared processor, however, this "busy waiting" is extremely undesirable, since it represents a consumption of processor resource that might otherwise be avoided. It may also prevent the process which will respond to the request from ever executing, thus causing indefinite blocking (deadlock) of the waiting process. It is preferable to have a basic operation which will establish the exclusive access if the resource is available, and place the process in a blocked state otherwise.[8]

In Kronos, requests for access to channels take the form of function requests (messages) to MTR, containing the request number and the desired channel number. In this approach there is one communication path (the output register), and several possible messages, such as REQUEST CHANNEL, ASSIGN EQUIPMENT, REQUEST JOB SEQUENCE NUMBER, etc.

An alternative approach is to provide a toggle (reserved/not reserved) associated with each channel. This toggle would then be accessed directly by any processor. If the toggle were already set to "reserve," then the accessing processor would be delayed until the toggle was reset. This corresponds to having a number of communication paths (toggles) and a single message (reserve the channel).

Access to the toggle associated with each channel is access to a shared "variable," and the mutual exclusion on this access is required in order to guarantee correct operation—i.e., it must not be possible for a second process to access the toggle between the time the toggle is tested by the first process and the time it is set to "reserved."

Exclusive access for "testing and updating" of a memory location is provided on the IBM 360/370 Series by the TEST AND SET instruction. This instruction tests the leftmost bit of the addressed memory location and sets the addressed memory location to all ones. No other access to the addressed location is permitted between the moment of fetching and the moment of storing all ones. In this way there is no possibility of another process intervening and testing the shared variable. This mechanism does not provide for any automatic action; if the resource is not available, the process must suspend itself. The TEST AND SET instruction is used to effect coordination between two central processors in a multi-

processor system.* The waiting processor cycles in a "spin loop," repeatedly issuing TEST AND SET, until the tested memory location is cleared by the other processor.

On the CDC CYBER, an *interlock register* is available as an option. Bits or words in this register can be tested and updated by any peripheral processor, with guaranteed exclusion of access. Operation of this interlock register is similar to TEST AND SET, except that several optional actions are available (e.g., READ, TEST, TEST AND SET, TEST AND CLEAR, etc).

Semaphores. Dijkstra, in a now classic paper,[2] has offered a formal mechanism for the solution of the problem of establishing exclusion with the introduction of two primitive operations, called P and V, and a specialized integer data type called a *semaphore.*

When a process executes a P-operation on a semaphore S and the value of S is greater than zero. S is decremented by one and the process continues. If the value of S is not greater than zero the process is blocked from further execution. The value of S is also decremented by one either (1) as soon as the resulting value would be non-negative[2] or (2) immediately: the magnitude of the (negative) value of the semaphore indicates the number of processes waiting.[3] If the process is allowed to continue, then the P-operation is indivisible. If the process is to be blocked, then the test and the blocking of the process constitute an indivisible operation and the "deciding to release" and the actual release of a waiting process also constitute an indivisible operation.

When a process executes a V-operation on a semaphore S, then S is incremented by one, and the process continues. In addition, if any process is waiting on that semaphore, exactly one is removed from the waiting queue and permitted to continue execution. The V-operation is also indivisible.

The indivisibility of these two primitive operations is the required guarantee of mutually exclusive access to the shared variable (the semaphore).

A semaphore (initialized to 1) can be used to establish and release exclusive access to a resource. If RES is a semaphore representing a resource (compare with the toggle discussed above) then the use of the resource is preceded by a P-operation to reserve the resource and followed by a V-operation to release it:

```
P(Res);
use the resource;
V(Res).
```

If the first definition of the P-operation is used, then the semaphore will have value 0 while the resource is in use; an attempt by a second process to access it will result in a delay of the second process until a V-operation is performed by the first process. At this point the semaphore becomes greater than zero and is immediately decremented again, granting access to the second process and completing its P-operation. This form of the semaphore is called a *binary* semaphore, because the only values it can have are 0 and 1.

Process cooperation. Two cooperating processes can coordinate their access to a shared address space by only accessing that space while they are within a critical section delimited by P and V. Producer/consumer relationships can also be controlled with semaphores—if a portion has not yet been produced, the consumer will be delayed. The initial value for a producer/consumer semaphore, representing available portions, is zero. Since the producer may produce several portions before the consumer can consume them, the semaphore may take on arbitrarily large values. This form of semaphore is called a *general* semaphore.

Process Competition. The mutual exclusion required for resource allocation can also be implemented using semaphores. If there is only a single instance of a nonsharable resource, the associated semaphore will be binary. If there is a pool of identical resources, the semaphore will be general. In both cases the initial value of the semaphore is equal to the number of resources.

Note that the ENQUEUE/DEQUEUE functions of OS can be used in very similar ways for resource allocation. They are more complex to analyze in that the enforced waiting is conditional.

Process Scheduling. Semaphores have obvious implications on the scheduling of processes—a V-operation on a specific resource semaphore will result in the release of a process, if any is waiting for that resource (so that it can then complete its delayed P-operation). A P-operation by a process may result in the delay of that process until the resource which it desires becomes available. Semaphores are defined in such a way that if several processes are waiting to complete a P-operation, only one of them will be selected to proceed when a V-operation is performed. The only requirement is that the selection of a candidate by "fair"—i.e., that no one process be indefinitely overtaken by others.[19]

Semaphore implementation. On a uniprocessor system, semaphores can be implemented by the control program using the global masking of interrupts to guarantee the indivisibility of the P- and V-operations. On a multiprocessor system, special hardware is needed to guarantee mutual exclusion of access to the semaphore (i.e., "cycle stealing" by another I/O or central processor must be prohibited while the semaphore is being updated). Both the 360/370 TEST AND SET instruction and the CYBER interlock register satisfy this constraint.

Recent extensions and adaptations. The semaphore as proposed by Dijkstra may be viewed as a signaling mechanism, but one which only signals that the desired event has occurred. Associated with a semaphore is always a queue (empty if $S > 0$) of processes waiting on that semaphore. It is customary to associate the pointer to that queue with the semaphore itself. (This may be compared with the address of the task descriptor recorded in the OS EVENT CONTROL BLOCK.) This association is used in the Venus system,[20,21] in which semaphores are known to the microprogram of a small computer—i.e., P and V are instructions on the machine.

A further extension is used in the Honeywell Series 60/Level 64.[22] Here a firmware convention is adopted concerning the availability of a short (16-byte) message area (from a pool) associated with each V-operation. This is useful, for example, for making a status report on completion of an I/O operation available directly to the process which initiates the request for I/O, without the need for either a shared, dedicated status area in central memory or a separate buffer management semaphore. Various other extensions are discussed by Presser.[6]

It may also be noted that the availability of semaphores makes the I/O interrupt superfluous. Completion of an I/O operation can be indicated by a hardware- or firmware-generated V-operation, which is "noticed" by a software process as the result of a P-operation on that same semaphore. Thus the equivalence of external (real) concurrency and internal (apparent) concurrency is established.

*Note that multiprocessing is not supported by regular versions of OS.

Resource management

Semaphores may be viewed as powerful but relatively low-level constructs. When they are used for resource management, they are typically used in certain "standard formats." Failure to use them in precisely the right way can lead to errors which are particularly difficult to detect and diagnose. In addition, it can be argued that the variables which represent a resource, and which must be manipulated within critical sections distributed among all of the competing processes, are not properly part of the state space of *any* of these processes.[23] In this section we will introduce two mechanisms for resource management (and, therefore, the control of concurrency) which attempt to make the design of operating systems safer. Both do this by isolating the resource management from the users of the resources: the first, by creating a separate process to manage the resource; the second, by defining precise properties of a control program function to manage the resource, and removing the variables that represent the resource from the address space of the requesting processes.

Secretaries. The actions of reserving and releasing a resource may be viewed as procedures *reserve* and *release* acting on a variable (or a data structure) which represents the resource. A possible move in the direction of increased safety is to associate the procedures with the data structure on which they operate, in a separate process. In this way it can be guaranteed that the data structure is *only* operated on by the defined procedures, because the variable(s) representing the resource are no longer part of the address space of the requesting processes.

Resource allocation can then be likened to the actions of a group of directors (competing processes) being scheduled by a secretary, which is a special, high-priority process which implements the procedures acting on the data structure.[19] A secretary is in fact a "semi-sequential" process,[19] in that her procedures represent the allowed operators on the data structure, and the order in which her actions are performed is undefined—i.e., it depends on the order in which the directors ask for her services.

This concept may also be used recursively: a process which is a secretary to one group of processes may act as a director to another. This allows the construction of hierarchical systems.[24,25]

Monitors. Another approach to the idea of combining shared variables with their operators (procedures) is provided by the "class" concept of SIMULA 67.[26] To this concept we may add the requirement that access to the procedures of the class be restricted to strictly one-at-a-time—i.e., the mutual exclusion, which we have previously associated with the distributed critical sections, is now associated with a centralized object.

The combination of the shared variables, the set of meaningful operations on them, and the mutual exclusion, is called a *monitor.*[27] The shared variables become a data structure which is syntactically local in the scope of the monitor itself. The mutual exclusion of procedure execution is implied and need not be explicitly coded. The procedures thus appear to be critical sections within a process, but they are different in that the shared variables are no longer in the address space of the invoking processes.

A monitor may be viewed as an *abstract data type*—i.e., a language notation which associates the procedures with the data structure on which they operate.

Monitors are used primarily to construct resource schedulers. If the desired resource is not available when the monitor is entered (this is reflected in the state of the local variables), then the calling process is delayed until the resource is released by another process. The mutual exclusion on the access to the monitor procedures is also released, for otherwise no other process would be able to enter the monitor and release the desired resource.

The concept of a monitor is similar to the implementation of control program functions in OS—the explicit disabling of the possibility of externally-forced context switching provides the mutual exclusion aspects of a monitor. However, in OS there are only four structural levels (the four classes of SVC). Monitors have greater potential for the construction of hierarchical systems.

Monitors have been illustrated and examined extensively in the recent literature.[28-30] There appears to be agreement that the basic idea of a class-like structure is sound, but there are various proposals concerning the detailed syntax of the abstract data type, and on the syntax and semantics of the delaying operator. For the present, insufficient experience has been gained to evaluate the relative merits of these proposals.

Language proposals. Control of concurrent activity is, in general, more difficult to achieve than control of sequential activity. It is therefore desirable that a higher-level language be used in the development of control programs. The programming language Pascal[31,32] provides a base with elegant data structuring capability, in which the addition of abstract data types is quite natural. Recent proposals[30,33] to extend Pascal with processes (which could be secretaries), monitors, and queues of waiting processes are now being used in pilot projects to test the usefulness of these constructs.

The advantage of monitors and secretaries over other mechanisms is that they package the operations and the data structures so that they must be used correctly. Specifically, it is possible to check for correct usage at compile time, and thereby avoid the need for extensive execution-time verification of parameters, etc. Also, the operations on the data structure are defined in only one place, and are therefore less prone to errors in coding.

Summary

As computing systems have become increasingly complex, real and apparent concurrency have been introduced in order to increase utilization. This has necessitated the development of mechanisms for controlling concurrency and preventing errors due to arbitrary interleaving of sequences of instructions from separate processes.

In this paper we have examined the structure of two control programs and the ways in which they have solved the problem of the management of concurrency. These solutions have been based on rules which must be explicitly followed in the coding of the control program.

These rules form the basis for proposals for coordination primitives and language constructs. Semaphores are ideally suited to the establishment of short-term mutual exclusion for process coordination. Secretaries and monitors represent attempts to derive a superior mechanism when the problem is one of scheduling of resources. Neither semaphores nor monitors are inherently more powerful, since each can be implemented in terms of the other, but the semaphore is the more primitive concept. For this reason it is more likely that semaphores, or some similar construct, will be implemented within the basic instruction set of new computers. Monitors are more likely to be a feature of systems programming languages, because of the way in which they package data structures and operations (procedures) into an abstract data type. ■

Acknowledgements

The assistance of J. Vaucher, M. Collins, J. Woodrow, S. Bush, T. Davidson, and the reviewers is most gratefully acknowledged.

The writing of this paper was supported in part by the National Research Council of Canada.

References

1. J. J. Horning and B. Randell, "Process Structuring," *Computing Surveys,* Vol. 5, No. 1 (March 1973), pp. 5-30.

2. E. W. Dijkstra, *Cooperating Sequential Processes,* Technological University, Eindhoven, The Netherlands, 1965. Reprinted in *Programming Languages* (F. Genuys, ed.), Academic Press, New York, 1968.

3. E. W. Dijkstra, "The Structure of the 'THE'—Multiprogramming System," *CACM,* Vol. 11, No. 5 (May 1968), pp. 341-346.

4. D. Morris and G. D. Detlefsen, "A Virtual Processor for Real Time Operation," *Proc. COINS Symposium,* 1969. (Also published as a report from the Department of Computer Science, University of Manchester, U.K.).

5. P. Brinch Hansen, "The Nucleus of a Multiprogramming System," *CACM,* Vol. 13, No. 4 (April 1970), pp. 238-250.

6. L. Presser, "Multiprogramming Coordination," *Computing Surveys,* Vol. 7, No. 1 (March 1975), pp. 21-44.

7. P. J. Courtois, et al., "Concurrent Control with 'Readers' and 'Writers'," *CACM,* Vol. 14, No. 10 (October 1971), pp. 667-668. See also C. A. R. Hoare, *CACM,* Vol. 17, No. 10 (October 1974), Section 6, pp. 209-215.

8. Bunt, R. B., "Scheduling Techniques for Operating Systems," *Computer,* Vol. 9, No. 10 (October 1976), pp. 10-17.

9. C. G. Bell and A. Newell, *Computer Structures: Readings and Examples,* McGraw Hill, New York, 1971, p. 76.

10. J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE,* Vol. 63, No. 9 (September 1975), pp. 1278-1308.

11. R. C. Holt, "Some Deadlock Properties of Computer Systems," *Computing Surveys,* Vol. 7, No. 3 (September 1972), pp. 179-196.

12. *Control Data CYBER 70/Model 73 Computer System Reference Manual,* Publication No. 60347200, Control Data Corp., Arden Hills, Minnesota, 1971.

13. *Control Data 6000 Series Computer Systems Reference Manual,* Publication No. 60100000, Control Data Corp., Arden Hills, Minnesota, 1971.

14. *Kronos 2.1 Reference Manual,* Publication Nos. 60407000 (Vol. 1) and 60448200 (Vol. 2), Control Data Corp., Arden Hills, Minnesota, 1975.

15. *Transaction Subsystem Reference Manual,* Publication No. 60407900, Control Data Corp., Arden Hills, Minnesota, 1973.

16. *IBM System/360 Principles of Operation,* Form GA22-6821, IBM, Poughkeepsie, New York, 1970.

17. *IBM System/370 Principles of Operation,* Form GA22-7000, IBM, Poughkeepsie, New York, 1970.

18. *IBM System/360 Operating System MVT Supervisor PLM,* Form GY28-6659, IBM, Poughkeepsie, New York, 1970.

19. E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica,* Vol. 1, No. 2, 1971, pp. 115-138. Reprinted in *Operating System Techniques,* (C. A. R. Hoare and R. H. Perrott, eds.) Academic Press, New York, 1972.

20. D. C. Tsichritzis and P. A. Bernstein, *Operating Systems,* Academic Press, New York, 1974.

21. B. H. Liskov, "The Design of the Venus Operating System," *CACM,* Vol. 15, No. 3 (March 1973), pp. 144-149.

22. T. Atkinson, "Architecture of Series 60/Level 64," *Honeywell Computer Journal,* Vol. 8, No. 2 (1974), pp. 94-106.

23. C. Bron, "Trends in the Synchronization of Parallel Processes." Paper presented at Annual Symposium of the Dutch Computer Society, Technological University, Eindhoven, The Netherlands, February 6-7, 1974.

24. K. C. Sevick, et al., "Project SUE as a Learning Experience," *AFIPS Conference Proceedings, 1972 FJCC,* Vol. 41, pp. 331-338.

25. R. C. Holt and M. S. Gruschow, "A Short Discussion of Interprocess Communication in the SUE/360/370 Operating System," *Proc., SIGPLAN/SIGOPS Workshop on Programming Languages and Operating Systems,* Savannah, Georgia, April 1973.

26. O. J. Dahl, B. Myhrhaug, and K. Nygaard, *Simula 67—Common Base Language,* Norsk Regnesentral, Oslo, Norway, May 1968.

27. P. Brinch Hansen, *Operating System Principles,* Prentice Hall, Englewood Cliffs, New Jersey, 1973, p. 121.

28. C. A. R. Hoare, "A Structured Paging System," *Computer Journal,* Vol. 16, No. 3 (1973), pp. 209-215.

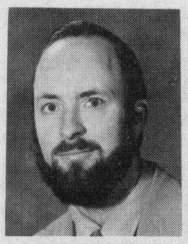
29. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *CACM,* Vol. 17, No. 10 (October 1974), pp. 549-557.

30. P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering,* Vol. SE-1, No. 2 (June 1975), pp. 199-207.

31. N. Wirth, "The Programming Language Pascal," *Acta Informatica,* Vol. 1, No. 1 (1971), pp. 35-36.

32. K. Jensen and N. Wirth, "Pascal—User Manual and Report," Springer Study Edition, Springer-Verlag, Berlin and New York, 1975.

33. D. W. Bustard, "Progress towards a Structured Operating System," Technical Report, Department of Computer Science, The Queen's University of Belfast, May 1975.



J. William Atwood is with the Department of Computer Science at Concordia University, where his interests include operating systems, computer architecture, performance measurement, and interactive program editing.

Atwood received his Ph.D in electrical engineering from the University of Illinois at Urbana in 1970. He subsequently joined the Department of Electrical Engineering at the University of Toronto, where he worked on the design and implementation of the Input/Output Supervisor for Project SUE.