Section III

REAL-TIME SOFTWARE DEVELOPMENT

THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Central Logic and Control Operating System

By J. P. HAGGERTY

(Manuscript received January 3, 1975)

The Central Logic and Control (CLC) is the digital computer that controls SAFEGUARD. This paper describes the novel features of the CLC operating system, presents its design rationale, and points out its limitations. Emphasis is on the characteristics that make the operating system suitable for applications other than SAFEGUARD. These include its ability to control as many as ten processors, its ability to initiate the execution of a program within milliseconds of an event, and its ability to detect and isolate faulty hardware racks without manual intervention.

I. INTRODUCTION

The Data-Processing System (DPS) at a SAFEGUARD installation is controlled by a stored program computer, the Central Logic and Control (CLC). CLC software can be divided into a set of applications programs plus an operating system. From the point of view of the operating system, all applications programs are simply the *user* or the user *process*.

Although assemblers, compilers, and linkage editors are usually considered part of an operating system, the CLC operating system provides none of these. All program preparation takes place on a separate support computer, currently an IBM System 370.* The programs compiled and link-edited on this machine, including the operating system itself, are brought to the CLC on magnetic tape as *load modules*.

II. THE ARCHITECTURE OF THE CLC†

The CLC consists of one to ten identical processor units sharing a common memory system, two Input/Output Controllers (IOCs), and two Timing Generators (TGs). Processors are independent of one

* The reasons for this are discussed in Ref. 1.
† A more complete hardware description appears in Ref. 2.

another in the sense that each executes its own instruction stream without knowledge of the instruction stream being executed by any other processor. An *interrupt* causes a processor to switch instruction streams in response to an error condition it has detected, such as arithmetic overflow. Memory is of two types: program store, read-only core from which processors may fetch instructions but not data; and variable store, ordinary core which processors may read or write. Memory racks are shared and not associated with a particular processor so that any processor can reference any memory location. Processors always reference program store by absolute address; they may reference variable store either by absolute address or through base registers.

Data transfer between the CLC and its peripheral devices is performed by an IOC that operates independently of the processors. IOC programs residing in variable store may be initiated either by a processor or by a peripheral device; these programs may perform elementary storage-to-storage operations, such as setting or clearing bits in variable store, as well as I/O.

The IOC controls a variety of peripherals. Some of these are conventional data-processing devices such as the disc drive units, the magnetic tape transports, the card reader, and the line printer of the recording subsystem; the teletypes; and the cathode-ray tube displays of the display subsystem. Other equipment such as the radar subsystem, the missile subsystem, the TG, and the Maintenance and Diagnostic Subsystem (M&DSS) are also considered peripheral devices only because they communicate with the IOC rather than with the processors directly.

The TG, part of the CLC, contains a time-of-day clock incremented every 200 ns. The TG can cause the initiation of an IOC program when a specified time of day has been reached. By suitable IOC programming, this notification may be made repetitive.

The M&DSS is particularly important to the operating system. It can inject logic signals into and sense logic signals within DPS racks at predefined M&DSS test points. Under the proper conditions, the M&DSS can control DPS equipment by means other than their normal interfaces. For example, an M&DSS instruction that places the proper pattern on IOC test points could cause an I/O operation to be performed. M&DSS instructions can originate from various sources, only one of which will be mentioned here: the M&DSS read-only core memory. M&DSS executes instructions from this source in response to one of three stimuli: manual intervention, failure of the CLC operating system to reset a *sanity timer*, or an explicit request from CLC software.

The SAFEGUARD data-processing system includes standby equipment. There is one extra processor, program store, variable store, IOC,

and TG. A given peripheral device is either duplicated and wired to a particular IOC or switchable under program control to either IOC. Software can establish a *green partition* and an *amber partition* such that equipment in the green partition cannot communicate with equipment in the amber partition, and vice versa. The amber partition has two purposes. Spare equipment is partitioned amber, so it may be used as a pool of inactive equipment from which replacements for green units are drawn; for example, the operating system can substitute the amber IOC for the green IOC. When it contains sufficient equipment, the amber partition may function as an independent computer. The operating system then executes independently in each partition.

III. THE SUPPORT MODE AND THE PROCESS EXECUTE MODE

The SAFEGUARD data-processing system is used for three different activities with distinct requirements:

(*i*) Tactical execution of a user process.
(*ii*) Debugging of a user process.
(*iii*) Utility operations such as saving the contents of disc packs on magnetic tape.

The operating system reconciles conflicting requirements between these three environments by functioning in the *process execute mode* for item (*i*) or the *support mode* for items (*ii*) and (*iii*).

In the support mode, the CLC operating system reads requests from job control cards to invoke utility programs. Some of these programs allocate space on DPS disc volumes; others install load modules created on the support computer onto DPS disc. Still others temporarily or permanently patch load modules.

Debugging is easier in the support mode than in the process execute mode. In the process execute mode, program testing is hampered because manual interactions such as a teletype input cannot be exactly reproduced for each test run and because the cause of an error is difficult to determine when several processors have been executing simultaneously. In the support mode, on the other hand, the operating system allows simulated manual inputs to be generated as specified by a card deck, each card tagged with the time of day it is to be processed. Also in the support mode, the operating system allows all processors but one to be idled when a programmer-specified condition occurs. Only one user job can run at a time, although that job may use more than one processor. A more detailed discussion of the operating system support mode debugging capabilities appears in this volume.[3]

The second mode of the CLC operating system, the process execute mode, is discussed in depth in Sections V through IX.

IV. RECONFIGURATION, LOADING, AND DPS RECOVERY

Selection of either the support mode or the process execute mode is under the control of the CLC data-processing system operator at the time the system is initialized. The major events following a request for the process execute mode will now be examined.

First, the operating system attempts to identify faulty hardware, such as an IOC that appears unable to reference a particular variable store. Next, it establishes a green partition sufficiently large for the user process (by examining tables stored on disc along with the process), and partitions amber all equipment not needed. Finally, it loads the user process from disc, it enables the sanity timer, and execution begins.

The same sequence of events can also be initiated manually or automatically during execution when DPS sanity is in question, in which case it is called *DPS recovery*. The reason for this operation is discussed in Section IX.

Both manually initiated loading and DPS recovery involve the M&DSS. Each causes the M&DSS to execute a program that idles all processors, causes the IOC to load a portion of the operating system into memory, and restarts all processors. The remainder of the load or the recovery is performed by the operating system as described above.

V. THE PROCESS EXECUTE MODE

Two fundamental constraints are placed on the CLC operating system in the process execute mode:

(*i*) *Timing*. Certain user process computations are required as often as every 6.5 ms.

(*ii*) *Error Control*. The incidence of a hardware or software failure must not cause the operating system to lose control.

The following sections of this paper examine how the four operating system functions of processor management, main storage management, I/O management, and error recovery are performed as a consequence of these constraints.

VI. PROCESSOR MANAGEMENT

The problem of processor management is simply stated: How shall the CLC processors (as many as ten) be best utilized to perform the SAFEGUARD process control calculations within the real-time constraints imposed by system requirements? To provide the necessary throughput, the multiple processors must be permitted to perform certain calculations in parallel, but how shall this capability be provided to the programmer? Shall the programming language allow statements to

be executed in parallel as in ALGOL 68? The answer is no. Parallelism is excluded from the language, and instead the operating system is allowed to execute simultaneously as many "independent" programs as possible, as in conventional multiprogramming systems.

Recognizing that it may be necessary to prevent one program from interfering with another through alteration of shared data, the operating system provides functions equivalent to Dijkstra's[4] P and V so that "independent" programs may cooperate, thus becoming no longer truly independent. Programmers can write parallel algorithms involving several programs. The operating system will assign programs (now called *tasks*) to processors so that as many processors as possible are busy. The assignment algorithm is sketched later.

The real-time constraint can be approached in two ways. "Time" often suggests "interval timer," the expiration of which usually causes a processor interrupt, followed by the initiation of the time-dependent computation. This method becomes decidedly unattractive if the time-dependent computation must be performed on more than one processor, for the operating system would have to decide which processors to interrupt, save the previous state of each, initiate new tasks on several processors, and later restore the processors to their original tasks. Therefore, this interrupt-driven approach is discarded in favor of a simpler method that is suggested by the following observation. Assume the time-dependent calculation must be completed within 6.5 ms from the time of request and further that it can be structured as P tasks each having an execution time T of less than 6.5 ms. Then if, among the tasks that are already running at the instant the time-dependent calculation is requested, at least P of them finish within $6.5 - T$ ms, sufficient processors will be available to complete the desired computation. By restricting task run times to the millisecond range, the desired behavior can be produced without timer interrupts because processors become free every few milliseconds.

If a computation cannot be completed in milliseconds, it is divided into pieces (tasks) that can be completed in the allotted time, and each task is executed in turn. This requires the operating system to recognize predecessor conditions, e.g., that Task B cannot run until Task A completes. It is useful to allow more complex situations, such as those represented by Fig. 1. Here, Task A is said to *enable* Tasks B, C, and D, and Task E cannot execute until *conditionally enabled* by both C and D. Enablement is a generalization of the "wake-up" operation of other operating systems.[5]

What conditionally enables Task A? It could be some other task not shown, or it could be the operating system. One particularly important feature of the operating system is that it can be requested to enable a

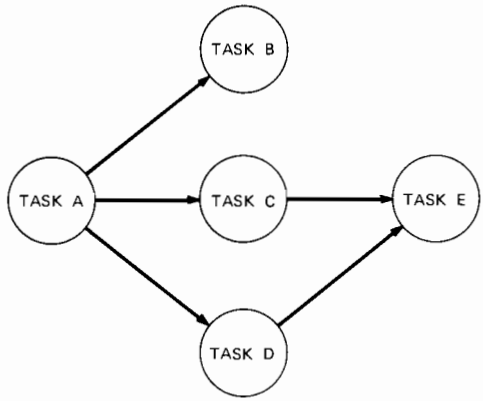

Fig. 1—Predecessor conditions among tasks.

given task approximately every 6.5 N ms ($N = 1, 2, 4, \cdots, 64$). Sets of tasks initiated this way are called *timed arrays*, structures that form the basis for almost all time-dependent computations performed in the process execute mode.

Assuming that each task is assigned a unique *priority* relative to all other tasks, the following algorithm decides which task will run next on a given processor:

(*i*) Of all the tasks whose predecessor conditions have been satisfied but which are not executing yet, execute the task of highest priority.
(*ii*) Allow each processor to perform step (*i*) independently of all other processors.

If each processor performs this operation whenever the task it is currently executing terminates, then no one processor is master over another, and the operating system is not sensitive to the number available. In fact, the number of processors can be increased or decreased during execution.

The way in which the operating system keeps track of the 6.5-ms intervals can now be explained. In Section II it was stated that the IOC can alter bits in memory, that an IOC operation can be initiated by a peripheral device, and that the timing generator may be programmed to signal the IOC at intervals of 6.5 ms. Let the IOC program "satisfy the predecessor conditions" of a task (i.e., set bits in an operating system table), and let this task be of high priority. The above algorithm then ensures that this task will execute as soon after the timing generator command as a task on any processor terminates. Although an exact 6.5-ms synchronism is not possible, the simplification of the operating

system achieved by not using timer interrupts for this purpose outweighs the disadvantage of having to account for a slight timing jitter when real-time deadlines are being planned.

The previous paragraph implies that, in the process execute mode, the operating system is itself executed as a set of tasks. This is indeed the case. In fact, the processor management algorithm makes no distinction between operating system tasks and those of a user, nor are system tasks necessarily of higher priority. In this way, execution of the CLC operating system is distributed over all the processors and the loss of a processor simply results in its load being equally distributed among those that remain.

VII. MAIN STORAGE MANAGEMENT

The hardware design of the CLC processor restricts the main storage management that the operating system can easily perform. The design does not allow the creation of a virtual memory since program store and variable store are both referenced by absolute addresses embedded within machine instructions. For the same reason, code is not easily relocatable, and a main storage management technique that assigns the same program to different locations in memory at different times is not feasible. A static allocation for all main storage is therefore implied. With a minor exception for part of variable store, this is the case.

Since programs are placed in fixed locations in memory, it is desirable to make this assignment only once, prior to task execution. The Execution Preparation Facility,[1] executing on the support computer, performs this function, and the load module brought to the CLC is not relocatable. This implies that the CLC memory rack configuration assumed at link-edit time must be available when the load module is read into core, and it is the responsibility of the reconfiguration and loading function of the operating system, described in Section IV, to ensure this.

The operating system provides a limited overlay mechanism. Two or more programs in the load module may be bound to the same address, and one or the other read into core as desired. The operating system performs the disc transfer, but it is the responsibility of the user to request the operation explicitly and to keep track of the current contents of overlay areas. Data base overlays may also be performed.

The operating system provides up to ten pushdown stacks in variable store, one for each processor. The stacks are used in the ordinary way for passing subroutine parameters, saving return addresses, and providing local storage for subroutines. A processor's stack is initialized to empty whenever a new task begins.

The CLC operating system provides no other forms of dynamic memory allocation. All other variable store usage, like all program store usage, must be declared at compile time. The decision not to allocate variable store dynamically meant that the maximum amount of data to be passed between two tasks would have to be decided at design time. This decreased operating system overhead and ensured the existence of a data structure large enough to handle the specified traffic level.

VIII. I/O MANAGEMENT

The traditional I/O management functions of an operating system are I/O scheduling, buffering, I/O completion processing, reservation and allocation of devices, and protection of one user from another. But an operating system can also provide other services such as concealing differences between devices so that one device can be substituted for another or altering the appearance of the device so that it is easier to program. In any case, an operating system should ensure the reliable performance and efficient use of the peripheral devices.

The CLC operating system deals with two general categories of devices. The first set consists of the conventional devices and includes the magnetic tape transports, the disc drive units, the cathode ray tube displays, the teletypes, the card reader, and the printer, and the second consists of the special-purpose devices such as the radar subsystem and the missile subsystem. These latter units are considered first.

For the special-purpose peripherals, the CLC operating system is only concerned with I/O completion and reliable performance. In particular, device characteristics are not camouflaged, and no attempt is made by the operating system to ensure the efficient use of the unit. Buffering is generally limited to providing an input area for devices that send data to the IOC of their own accord, under hardware rather than software control. The operating system functions of I/O scheduling, reservation, allocation, and user protection for this class of peripherals are simple. There is only one on-line unit of each type, and the user must do everything himself. Finally, an attempt is made to ensure the reliable performance of each device by monitoring some error indications it can produce and informing the user if trouble is being reported. These reports deal generally with the IOC-peripheral interface; the user is responsible for sensing and responding to device-dependent error conditions. The operating system was designed this way because the users were not sure at the time of how they wanted to program the special-purpose devices.

While the operating system management philosophy for special-purpose peripheral devices is generally one of minimal intervention, its approach for conventional devices is almost the opposite. Emphasis is placed on I/O scheduling and reliability and, in some cases, on altering the appearance of the device to the user. For example, to increase disc drive efficiency, read and write requests received by the operating system are reordered to minimize access delays. To increase reliability, each disc write is performed to two units so that if a subsequent read on one unit fails, a duplicate copy is available. The magnetic tape transports are another example. In this case, the appearance of the device is altered so that the user sees a tape capable of recording at up to four times the hardware rate of an individual transport. This is accomplished by directing suitably buffered output not to a particular transport but to a pool of four, capitalizing on the ability of the IOC to overlap writes on as many as four transports. The designers of the operating system knew how the conventional peripherals would be used, so they were able to plan more sophisticated support for them.

Neither the conventional nor the special-purpose peripheral devices generate processor interrupts when they complete a request. Instead, every 6.5 ms the operating system tests whether any I/O has completed. It then notifies the user via the conditional enablement of a user task. Since processor management uses no interrupts, neither does I/O management.

In the process execute mode, the CLC operating system makes no attempt to conceal the differences between devices, and programs are usually device-dependent. For tactical execution, this is permissible, but in other circumstances, it is a handicap. This is discussed further in Section X.

IX. ERROR DETECTION AND RESPONSE

The operating system detects errors in many ways and provides both local and system responses to these errors, depending on the circumstances. Local error responses consider the frequency with which an error is reported. If the frequency exceeds a given threshold, then extensive corrective action is assumed to be required. For error conditions that are treated in this manner, the operating system may make a particular response before the threshold is reached, but a different response after it is exceeded. For example, before the threshold is reached, a device reporting errors may be reset; after it is exceeded, further use of the device may be prevented.

This latter action suggests a general technique called "severing." If a peripheral device or a software function is declared severed, the

operating system rejects all future requests for that device or function. This procedure is applicable to a variety of error conditions; its intent is to minimize snowballing by preventing a second failure from occurring as a result of the first. In the face of many errors, severing produces a relatively gradual loss of operating system capabilities and is appropriate in situations in which the consequences of DPS recovery cannot be tolerated.

Some operating system functions, especially processor management and I/O management for the special-purpose devices, are never severed. These functions execute moderately elaborate error recovery code that attempts to prevent unrelated calls of the same type from failing.

System level responses are provided in but not initiated by the operating system. A more complete discussion of SAFEGUARD error control can be found in Ref. 6.

X. DEFICIENCIES OF THE OPERATING SYSTEM

A single mechanism for peripheral device substitution, a feature commonly found in general-pupose operating systems, is not in the CLC operating system. Initially, this was felt to be an unnecessary complication because the important peripherals, the special-purpose devices, cannot be mimicked by any other peripherals. Later, several operating-system designers needed particular instances of this capability, and each built his own version. Allowing commands to be read from the card reader rather than from a teletype (in the support mode) and permitting the use of one teletype in place of another (in the process execute mode) are both instances of peripheral device substitution, yet two different mechanisms were coded.

The operating system does not provide for communication between tasks, and it should. An extension of conditional enablement would be to allow a parameter list to be passed by each predecessor task. Communication between tasks does take place, but each programmer devises his own mechanism.

Whenever a particular subroutine was needed by one class of users, it was made part of the operating system and accessible to all users, thus penalizing those who did not require the subroutine by costing them core. A subroutine library established on the support computer would have avoided this.

XI. CONCLUSION

The CLC operating system is not intended to be general purpose and cannot easily be made so. Criteria that might be used to judge the adequacy of a general-purpose operating system do not apply to it, such as the ease of learning its job control language or the number of

jobs it can process per hour. Since the real-time performance of the SAFEGUARD Data-Processing System depends not only on the CLC operating system but also on the user process, the operating system would have to be considered a failure no matter how elegant it was if the overall real-time performance of the DPS were not achieved. But since the required performance has been achieved, the CLC operating system can be termed a success.

The operating system's most innovative and greatest success is its approach to processor management. The approach taken provides a rapid response time without the conventional use of processor interrupts. It also sets a logical framework in which it is possible to design, code, and test real-time programs taking advantage of up to ten independent processors.

REFERENCES

1. R. R. Conners, "SAFEGUARD Data-Processing System: Support Software and Support Computers: An Overview," B.S.T.J., this issue, pp. S149–S160.
2. J. W. Olson, "SAFEGUARD Data-Processing System: Architecture of the Central Logic and Control," B.S.T.J., this issue, pp. S41–S61.
3. A.K. Phillips, "SAFEGUARD Data-Processing System: Debugging a Real-Time Multiprocessor System," B.S.T.J., this issue, pp. S133–S145.
4. E. W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages*, New York: Academic Press, 1968, p. 68.
5. E. I. Organick, *The Multics System: An Examination of Its Structure*, Cambridge, Mass.: M.I.T. Press, 1972, pp. 275–281.
6. L. J. Gawron, "SAFEGUARD Data-Processing System: System Error Control," B.S.T.J., this issue, pp. S123–S131.