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

SAFEGUARD Data-Processing System:

Process Design in the Structure of Real-Time Software Systems

By W. S. DOYLE and J. R. GIBBONS

(Manuscript received January 3, 1975)

Process design, structuring the real-time program for the CLC, was one of the difficult aspects of SAFEGUARD software development. Initially, there were no significant guidelines or criteria. In the course of the project, basic process-design rules were developed and significant experience was acquired. Some techniques that emerged are the use of short-running, asynchronous tasks; overlays to minimize storage requirements; and multiple storing of programs to minimize processor queuing.

I. INTRODUCTION

Process design involves defining the characteristics, interrelationships, and organizational structure of the tasks that comprise the operating system and the applications software. It was one of the difficult aspects of SAFEGUARD software development. Initially, there were no specific criteria to be followed. Several iterations were required to converge on the final process design. The purpose of this paper is to present some of the basic guidelines that evolved in the course of the SAFEGUARD project. The guidelines included are those believed to be most workable and most applicable to a wide range of real-time software systems.

II. GENERAL PROCESS-DESIGN GUIDELINES

Major efforts in the process design involved selecting from among the available methods of enablement for tasks, selection of the time frames in which they would execute, and the definition of task priorities. (For a description of tasks and processor management, see Ref. 1.)

2.1 Task structure

Initial investigation of possible process structures led to the use of both synchronous (time-enabled) tasking and asynchronous (event-triggered) tasking. It was clear that critical processing had to be given high priority, and it was generally of a synchronous nature. Asynchronous tasks were to be used to fill the time slots between critical synchronous tasks and to provide a uniform distribution of processing among the available processors. This general approach had to be modified by a few additional considerations. First, low-priority asynchronous tasks must have a short run time or they will hold a processor too long, denying access to high-priority tasks. Second, it is generally more difficult to design and test a process which utilizes asynchronous tasks. Further, it is not always necessary to achieve a uniform work distribution, e.g., during the process initialization and termination sequence. An almost totally synchronous design was chosen for process initialization and termination tasks to facilitate design and testing.

It is inefficient to enable a synchronous task, only to find that the task has no data to process because a peripheral device has not completed its transfer or because other tasks have not generated it. Ultimately, synchronous tasks were utilized when critical and periodic response was required and when the availability of data at the same frequency as task enablement could be guaranteed.

The asynchronous, event-triggered task is enabled by the completion of an I/O transfer or by the successful completion of processing by a predecessor task or tasks. Each predecessor task can conditionally enable one or more successor tasks. A successor task is absolutely enabled, i.e., ready to run, only after all conditional enablement criteria have been satisfied. The predecessor-successor relationship of conditional enablement can also help alleviate data interference problems. Table I depicts some of the process-design questions that were faced and the type of tasks used to answer these questions.

Table I — Process design

Problem Description	Task Description
Support high-frequency, high-accuracy endoatmospheric target track.	Synchronous task whose frequency is at least as high as the update requirements.
Process intersite communications message traffic.	Asynchronous tasks whose trigger for enablement is the arrival of intersite communication messages.
Generate time-ordered, simulated radar replies during an exercise.	Both synchronous and asynchronous tasks. Tasks that generate the replies are synchronous. These tasks conditionally enable an asynchronous task which time-orders and outputs the simulated replies.

2.2 Parallel processing

There were several cases where identical processing had to be repeated for several items in a short time frame. In this case, the throughput requirement exceeded that of a single processor. The solution to the problem was to parallel process, i.e., to define several tasks executing identical code. Since the code was re-entrant, only one program copy was required even though each instance of the task could be separately controlled and separately enabled. Again, the structure of this processing could be synchronous, asynchronous, or a combination of both. It was found necessary to parallel process different types of tasks to take full advantage of the multiprocessor environment.

Obviously, multiple-instance task use may cause processor queuing problems. These can be alleviated by storing one program copy for each task. The critical consideration determining the number of program copies needed is the response requirement on the tasks involved.

2.3 Data interference

One of the primary design goals was to maximize throughput of the processing system. A natural implication of this was an attempt, in the beginning, to multiprocess everything. This immediately triggered task-to-task data-interference problems. Reviewing the task-response requirements made it obvious that not only was it not necessary to multiprocess all tasks, but in many instances it was impossible.

This observation led designers to take a closer look at task time-frame design and the serial-processing relationship among tasks. From these investigations evolved two basic task-design guidelines for avoiding data interference. If possible, competing tasks should be assigned to nonoverlapping time frames of possible execution.* If this could not be done, an attempt was made to establish predecessor-successor relationships among them. These techniques could be used only infrequently when tasks were competing for data.

Since a large number of data-interference problems were not solvable by either of these techniques, attention was directed to data-base design. Many interference problems arose when only two tasks were in competition, one loading the data and the other processing them. In those instances where the competing tasks were accessing a variable number of data items each time executed and the response requirements on the task were not critical, a circular queue with an access mechanism called a take-load pointer was used. With this mechanism, the loading task uses the load pointer to control the writing of data. It never

---

* A time frame is a time "window" in which a task is allowed to execute.

writes beyond the take pointer. The processing task uses the take pointer to control the reading of the data. It never takes beyond the load point. This technique alleviated about 10 percent of the interference problems.

When two high-frequency tasks with critical response-time requirements were competing for data, a double-buffering technique was useful to avoid data interference. In this case, two tasks both execute at a high frequency and in the same time frame. One loads the data and the other processes it. The competition question was solved by dividing the data area into two identical buffers, one of which was being loaded while the other was being unloaded. When unloading was complete, the buffers were switched. This technique works, but was of limited applicability.

As a final resort to solving interference problems, locking and unlocking conventions were used. These conventions required use of predefined program-logic sequences to lock and unlock data areas. These sequences relied on a special CLC instruction called a "biased fetch" which was implemented for this purpose. (For a more complete description, see Ref. 2.) Locking will always work, provided locking conventions are observed and enforced. Improper use of locking has caused the integration effort many headaches. The improper use of locks will manifest itself in a thousand disguises. However, it was necessary to use locking to solve more than half of the interference cases.

2.4 Discussion

How well is the process working? How close does the process conform to the process-design requirements? These are two questions that were constantly asked. To answer them, a process performance-monitoring capability was implemented. The implementation relied on constant monitoring of "probe" or test points within the process. Implantation of these probes into the process and interpretation of the resulting data proved useful for fine tuning the design and verifying that the basic requirements were being met. This should have been done much earlier in the design cycle. Probes should be capable of furnishing such data as routine and subroutine execution timing; the time differential between when a task is enabled and when it actually acquires a processor; minimum, maximum, and average task run times, etc.

This section would be incomplete without a few words about the position of the process designer. It became obvious that the process designer must participate in program design and integration. He must do this to guarantee that the program designers do not stray from the process-design requirements on program timing and interfaces. He must be part of the integration effort to ensure that the process design is actually implemented in the process. Furthermore, it was found that the process designer required this program design experience and integration experience to be able to accurately interpret performance data and to use it to refine the design of the process.

III. SYSTEM SIZING CRITERIA

Estimates of the number of processors, program stores, and variable stores needed to do the job were continually monitored in the light of the mission to be fulfilled by the system. System sizings were an iterative effort. As requirements solidified and understanding of them improved, as routine, subroutine, and data-base estimates improved, and as simulation tools for forecasting system loading improved, sizing estimates changed.

3.1 System operating points as design input

It was the process designers' responsibility to map system performance requirements into the number of instructions needed to code these requirements, the amount of variable store required to support the data base, and the number of processors needed to meet throughput requirements. The design effort attempted to balance, on a system cost basis, the inevitable trade-offs among these three resources.

To facilitate evaluation of the impact of the various trade-offs on process design, a contour or envelope of possible system operating points was developed. Points on this contour reflected maximum usage of one or more resources and/or maximum processing capability of one or more process functions. It soon became clear that there were not enough resources to support the "worst-case" condition for all process functions. Further, it was not only impossible to support the worst case, but not necessary, since all functions do not peak simultaneously. Once the contour was identified and a feasible and reasonable set of operating points selected from it, trade-offs could be thoroughly examined.

After the operating point was selected, it was the responsibility of the process designers to ensure that the design supported it. It was this effort that required the continual resizing of the system to guarantee that it would fit into the resources available.

3.2 Minimizing core requirements by the use of overlays

As design proceeded, program storage resources were rapidly exhausted. Further investigation showed that there were certain sets of programs that were not required to be in core simultaneously since their functions were mutually exclusive. Another set of programs had such "loose" timing requirements that they could be called in from a peripheral storage device prior to execution. Examples of such sets are hardware test programs, display update programs, and system initialization programs.

3.3 Load balancing

One of the most critical factors that influenced selection of the system operating point was the need to maintain a balance between the capability of the application process and the exercise process; that is, the exercise process must be capable of driving the application process at or above the system operating point.[3]

When planning for load balancing, two factors must be studied. These factors are the "immediate-response" processing requirements, representing a maximum allocation of resources applied for a short time, and the "long-term" or residual processing requirements, representing the load over a typical processing cycle.

Since the process had two basic time frames, one approximately 5 to 10 ms and one approximately 50 to 100 ms, two levels of load balancing were needed, short term and long term. Experience showed the most critical need for load balancing to be at the short-term level. It was also the most difficult to satisfy. Once the short-term problem was solved, the long-term problem disappeared. Short-term balancing was found to be extremely sensitive to changes in routine and subroutine execution times, and tuning the balance was always required.

IV. ALLOCATION OF RESOURCES

Consideration of possible process structures led to three basic alternatives for the allocation of the most critical system resources, processor and radar time. The first alternative is fixed allocation in which the execution time frame of each task is fixed in nonreal time by the process designer. The second alternative is real-time allocation in which the execution time frame of each task is determined dynamically by a synchronous allocation task included in the process. The third alternative is a combination of the previous two.

Initially, fixed allocation with its heavy reliance on synchronous tasking was favored because it appeared to be easier to design and test, and its reactions to traffic were easier to predict. After study, this design was rejected because it resulted in a nonuniform distribution of the work which, it was thought, would result in unacceptable system performance.

The second alternative to a process structure centered on attempting to allocate almost all resources in real time. This technique yields a much more uniform distribution of work among the processors and a better utilization of resources; however, designing and testing this type of process appeared to be very complex. In addition, it was decided that the uniformity of the distribution of work was not as critical as first thought.

Process design eventually included both types of allocation. This combination allowed the process to be designed and tested in a timely manner and yielded a nearly uniform distribution of work, giving reasonable processor utilization.

V. OVERLOAD RESPONSE REQUIREMENTS

SAFEGUARD process designers had to answer the question of what to do when there were more requests for service than could be accommodated. Because it was felt that the inherent overload handling of the priority tasking structure was not sufficient, a predefined, fixed-response technique was developed.

In this approach, a tunable processing load point was defined at which overload-response rules were invoked. The exact rule to be used depended on the outcome of an overload function which "predicted" processor usage for the next cycle. This prediction was done by summing selected system-traffic components weighted by an appropriate factor. Depending upon predicted processor usage, the execution of certain lower-priority tasks was curtailed. The higher the predicted usage, the more tasks were curtailed. Once the system entered overload, it remained there for the duration of the engagement.

This technique eliminated the additional testing and design required to implement a feedback type of overload response. The feedback technique was tried in the prototype system and was found to be impractical.

VI. MULTIPROCESSOR QUEUING PROBLEMS

Minimizing task run times was of critical importance for certain process functions; e.g., endoatmospheric tracking. Generally, functions with critical response times were also those functions selected for multiprocessing. This quickly led to a realization of the impact on task run time of processors queuing for instructions.

A decision had to be made either to use multiple copies of multiple-instance parallel tasks or to divide the program into subunits. The final decision was based on each task's response requirement. For example, in one instance five identical tasks executing from a single program copy ran 77 percent longer than single-processor run time. The same programs were suitably subdivided and partially distributed to five independently addressable storage units and run time was reduced to a level about 25 percent greater than single-processor run time. Of course, if five complete copies were stored in five different independently addressable storage units, there would be no increase in the parallel-tasking time versus single-processor execution. The final decision made was to use multiple program copies only for those tasks that always had to execute at maximum efficiency. This was done to conserve program storage. More commonly, large programs were divided into subunits distributed among program storage units in such a manner as to equalize the number of accesses per storage unit per time interval. This general technique was found to be sufficient for a large number of applications.

VII. SUMMARY

Initially, there were no significant guidelines to process design; these were developed as design progressed. No claim is made that the criteria which evolved in our design are exhaustive, but they should be applicable to a wide spectrum of real-time software systems.

It was good design practice to use short-running, low-priority, asynchronous tasks wherever possible. This helped alleviate task scheduler conflict problems, which arose when there were a large number of high-priority synchronous tasks. It helped guarantee that high-frequency, high-priority tasks would execute at their specified frequency, and it also aided in achieving a more uniform work distribution.

Data-interference problems arise naturally in a multiprocessing environment. The most useful technique to solve these problems was consistent use of software locking conventions; however, improper implementation of these techniques caused problems during integration.

To minimize system overhead and to avoid wasting processing time, tasks should be enabled only when they have work to do. Synchronous tasking should be used only if data are available to be processed at the same frequency as the enablement.

Since it was essential to maintain a balance of capabilities between the application process and the exercise process, it was required that the interfaces between these processes be established as soon as possible and that their integrity be rigidly maintained.

Because it was necessary to measure how well the process was working, it was found that performance probes should be included in the initial design and considerable thought should be given to their correct placement. Performance probes proved invaluable throughout the system-integration process, particularly in helping to identify task-timing and queuing problems. Resolution of these problems requires that the process designer become deeply involved in the test-and-integration effort.

Finally, process design is iterative. For this reason, it is important that the design be kept as simple and straightforward as possible. This standard guideline of programming is even more important in process design because of the inherent complexity of the multiprocessing environment.

REFERENCES

1. J. P. Haggerty, "SAFEGUARD Data-Processing System: Central Logic and Control Operating System," B.S.T.J., this issue, pp. S89–S99.
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. B. P. Donohue III and J. F. McDonald, "SAFEGUARD Data-Processing System: Process-System Testing and the System Exerciser," B.S.T.J., this issue, pp. S111–S122.