A simulator is discussed that provides a language and a structure specifically designed for modeling computer systems to evaluate their performance.

The simulator provides general equipment models, and the authors discuss their experience in developing a general submodel of a multiprogramming operating system. The user assembles a system from the equipment models, specifies parameters to allow simulation of his operating system functions, and provides models of his application programs.

Simulating operating systems

by P. H. Seaman and R. C. Soucy

A suitable simulation language and structure is needed to satisfy the requirements of computer system program simulation. One simulation language frequently used to model computing systems is provided by the General Purpose Simulation System (GPSS).[1] As a result of experience with many GPSS computer system models, the conviction crystallized that much of the equipment is standard, although the programming is not. Attempts were made to provide GPSS users with equipment submodels from which they could quickly assemble total system models. However, there was no convenient facility to incorporate such a higher-level system language (which the submodels constituted) into the GPSS structure; and such attempts as were made ran rather slowly.

From this work, a specialized modeling system was developed, called the Computer System Simulator (CSS).[2] CSS is not built on GPSS but is coded afresh in basic assembler language. It uses many of the techniques of GPSS, but they have been adapted to a structure specifically designed to model computer systems. CSS incorporates general equipment models into its basic structure and makes programming the operation of this equipment the central feature of its use. Thus, a sharp dichotomy is instituted between models of equipment provided by the simulator and models of programs provided by the user. With equipment details built in, it was soon seen that a large share of modeling complexity still lay in the simulated programs, especially in the area of what is called the operating system, which controls the scheduling of operations on the equipment in processing jobs.

To simplify CSS input specifications, it would be desirable to provide a general submodel of the SYSTEM/360 Operating System that is much like the equipment submodels provided in GPSS. The user would supply only certain system parameters in addition to his application program models to simulate his entire system. The operating system submodel thus visualized would be a CSS program written within the constraints of the simulator, imitating the principal functions of the supervisor and data management services.[3]

The user's complete system model then would consist of the following interacting parts:

- *CSS base*, providing configuration definition, equipment models, input job definition, simulating capability, and statistics gathering routines.
- *Operating system submodel*, providing control functions to simulate a multiprogramming environment.
- *Application programs*, driving the above two components in accordance with the input job stream and the user's program logic.

This paper discusses our experience in developing an experimental submodel of the SYSTEM/360 Operating System for general use.

Design objectives

There are at least three cases in which one may wish to simulate a computing system in detail. First, in the development of a programming system, it is useful to have a model of the current system so that proposed changes can be evaluated by modifying the model rather than the real system. Second, in establishing a system configuration for a given workload, it is desirable to determine an optimum configuration before ordering the parts. Third, after a system is operational, it is useful to keep a model of the system that can be used to predict the effect of expected or proposed changes to the actual system. In all three cases, the requirement arises to measure job throughput, system response, or equipment loading without the availability of some aspect of the real system. In the first and third cases, quite detailed models are necessary to observe the effects of many minor changes, the cumulative effects of which may be significant. In the case of system configuration, much grosser models usually suffice.

As an initial effort, it was decided to develop a CSS model package for both installation planners and program developers by means of which they could simulate the behavior of their programs within the constraints of the programming support provided with the equipment. A tolerance of ten percent on model accuracy calibrated against a real system was aimed for in equipment utilizations and total run time. The simulated system would consist of the three major components mentioned above—configuration definition, operating system submodel, and application programs—all implemented in the CSS language.

System configuration was to be specified in terms of normal CSS input. Predefined configurations are difficult to implement because of the large number of permutations, and are not necessary since the CSS format is easy to use. The actual device functions were part of the basic CSS program.

The operating system can be premodeled, because its logic is defined rather precisely; after deciding on particular modules to be included in his operating system, the user is really subjecting his programs to a small set of fixed and predictable rules. The CSS instruction set is sufficient to write routines that model the logical functions necessary for system control. The efficiency of the CSS language for this purpose was open to study.

Finally, the application programs were to be completely user-defined. However, it was desirable that the interface with the operating system submodel be through a macro language as similar as possible to that provided in the real operating system. Thus, the application program models would call for services from the operating system submodel in the same manner as their real-world counterparts. This correlation was designed for ease of use, so that a user familiar with operating system functions could easily put together a system package largely in his own terms without having to struggle with a host of new terms. However, no attempt was made to bridge the gap between the submodel package and the user unfamiliar with the operating system who merely wanted to compare "typical" systems. At that time, it was felt there was not enough information available to usefully define what might represent a set of typical systems.

Computer system simulator

To understand the submodeling effort, a short description of the CSS program on which it is based is necessary.[4] CSS provides the user with a language and structure with which he can model a large variety of computer systems and at differing levels of detail. The basic input in building a CSS model of a system consists of:

- Statement of system configuration
- Description of operating programs
- Description of job environment

The above specifications make up a single input card deck. System configuration is specified by various statements, in prescribed formats, that contain information on such characteristics as size of main storage, data transfer rates for each different I/O device, and I/O device connections to each channel. Following these statements are the program descriptions in CSS language. These consist of both the user's application programs and the system's control program, and must include all timing information as well as program logic. The job environment is implicit in both the configuration and program specifications. Input message rates may be specified for each terminal, or a job stream may

be set up from an input device. Tasks may be generated by programs, and conversations defined between terminals and one or more processors.

From these input statements, the CSS program assembles a model, the only restriction being that the complete assembled model must reside in main storage. CSS then creates messages and tasks according to the input logic and commences to operate the model in an interpretive fashion. Services such as updating the system clock, maintaining a list of future events, and generating random numbers and other statistical functions are automatically provided by the simulator. The model runs until one of several user-specified stopping conditions occurs, provided an error stop does not occur earlier. The user may terminate a run after a given number of messages have been processed, after a certain time span has been simulated, or after a given amount of execution time has elapsed.

Upon termination, an output report is generated summarizing the statistics gathered during the run. The output automatically provided includes:

- Listing of input specifications.
- Usage for all units of equipment—that is, the percentage of time during which each unit was in operation.
- Statistics on system queues occuring in the system.
- Statistics on the usage of system resources, such as blocks of main storage.
- Activity statistics, such as number of messages generated at each terminal.

In addition, the user may collect data of specific interest about his system, such as distributions of various response times. This output is used to determine the extent to which system performance meets desired criteria. Changes to the system to improve performance can then be incorporated in the model, the program can be rerun, and the quantitative effect of the changes studied. The program does not optimize a design itself; it simply acts as a tool to aid the judgment of the designer.

The following equipment can be simulated in CSS:

CSS equipment simulation

- Multiple processors, including main storage and programs. Input/output (I/O) interruptions and I/O overlap are provided for.
- I/O devices connected to processors through control units and channels, including multiplexed channels and cross-channel switching arrangements. I/O devices include disks, tapes, drums, printers, card reader/punches, data cells, and generalized I/O units.
- Communication lines connected to the channels. Both contention and polling control are modeled.
- Terminals connected to the lines. There may be several terminals per line, with various polling disciplines. Both random input and conversational mode can be handled.

Figure 1 Simple system configuration

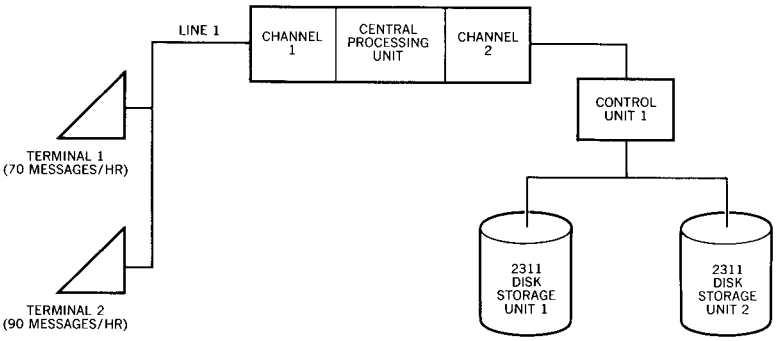


Table 1 Equipment configuration statements

Unit number	*Unit type*	*Unit characteristics*
	SYSTEM	PR1,CU1
1	LINE	14.7
1–2	TERMINAL	,,,1
	RATE	1/70,2/90
1–2	CHANNEL	1
1–2	2311	1,156000,25000
	PATH	1/1–1,2/2–1

In the above equipment, start or clutch time, line transmission delays, character transmission rates, etc., are specified by the user. Thus, any particular tape unit, disk unit, or peripheral I/O device can be modeled by specifying the appropriate values.

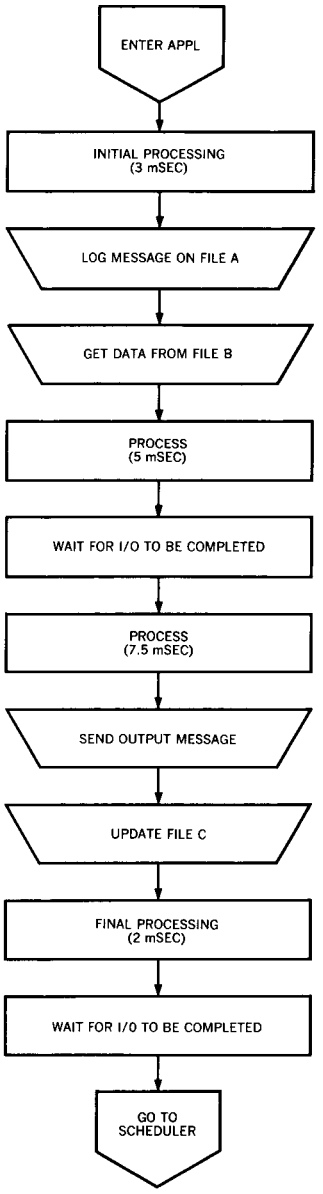
The equipment configuration statements needed to specify to CSS the system outlined in Figure 1 are shown in Table 1.

The SYSTEM statement indicates that there is one processor and one control unit for which no further specifications are needed. The LINE statement specifies that line 1 has a speed of 14.7 characters per second. The TERMINAL statement specifies that terminals 1 and 2 are connected via path 1, while the RATE statement specifies an input rate of 70 messages per hour for terminal 1 and 90 messages per hour for terminal 2. The CHANNEL statement indicates that channels 1 and 2 are both connected to processor 1. The 2311 statement specifies that there are two IBM 2311 disk storage units connected to control unit 1, with 156,000 bytes per second data transfer rate and 25 milliseconds (25,000 microseconds) rotation period. The PATH statement specifies two paths, the first connecting channel 1 and line 1, the second connecting channel 2 and control unit 1. Each of these configuration statements may convey many other characteristic parameters not shown. Other equipment models may be added similarly.

Table 2 Typical application program

APPL	PROCESS	3000	Initial processing
	WRITE	(file A)	Log message on file A
	READ	(file B)	Access file B
	PROCESS	5000	Overlapped processing with I/O
	WAIT	SCHEDL	Suspend processing until I/O completed
	PROCESS	7500	Unoverlapped processing
	SEND	(destination)	Send output response
	WRITE	(file C)	Update file C
	PROCESS	2000	Overlapped final processing
	WAIT	SCHEDL	Wait for I/O completion
	BRANCH	SCHEDL	End of program, go to scheduler

Figure 2 Sample program, APPL



After the equipment has been defined, the programs that control the processing of messages or jobs must be specified. For this purpose, CSS provides an instruction set that is used to construct flowcharts of the programs.

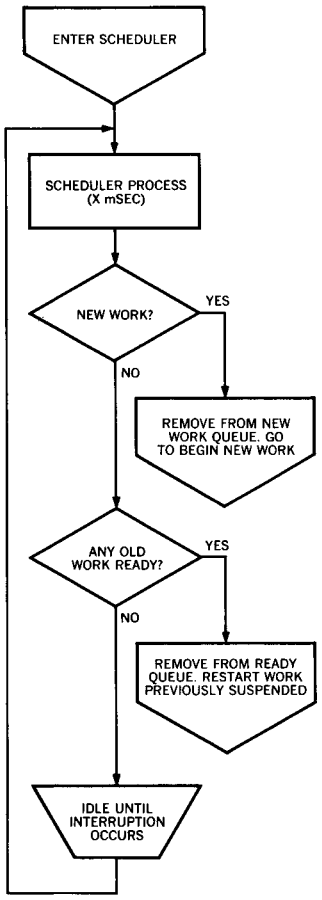
The CSS instruction set includes 40 instructions, some directly corresponding to operating system macroinstructions in the system under study (READ, WRITE, BRANCH, ALLOCATE, etc.). Others, such as PRINT and TABULATE, are used for statistical purposes. Finally, instructions are provided for manipulation of the simulation model. These include a probabilistic branch, test instructions, queuing instructions, arithmetic instructions, etc. In addition, the user may build a set of macroinstructions from the basic set, in effect creating his own language. This feature is much used in the modeling effort.

When modeling programs in CSS, two classes of programs must be defined: application and control. The first of these represents user-written programs to perform a specific job. The second represents the operating system that controls execution of the application program and regulates both the machine, including interruption handling and task scheduling, and external conditions, such as arrival rate of jobs.

An example of a simple application program is shown in the flowchart in Figure 2, and it can be written in CSS as shown in Table 2.

The time unit in this example has been taken as 1 microsecond; thus, a process time of 3 milliseconds appears as 3,000 microseconds. The unit may be changed at the option of the user. The I/O information, which is indicated here only by the parentheses, specifies the location and amount of data to be moved. The WAIT instruction causes processing of the task to be suspended until its outstanding I/O operation is completed. During this time, processing can be carried out for other tasks in a multiprogramming environment. Thus, control is returned to the operating system scheduler (here called SCHEDL) to find such work. Similarly,

Figure 3 Simple task scheduler



at the end of the program, all work on the current task having been completed, control is returned to the scheduler to begin a new task.

Such a scheduler, and other control programs like it, provide the framework within which the application programs are executed. Control programs are coded in the CSS instruction set in the same manner as the application programs. Virtually any operating discipline may be accounted for. For example, a flowchart for the simple scheduler discussed above is shown in Figure 3. Each block represents one CSS instruction.

Another important part of an operating system is made up of the interruption-handling programs. These are automatically branched to, and any program that is being executed is interrupted, whenever an I/O operation, such as a seek or data transfer, is completed. Such an interruption-handling program is shown in Figure 4, providing control for a teletype line.

Again, every box represents a CSS instruction. Note that the processing time to handle the interruption must be specified by the user. Oftentimes, when considering a control module like this, prepared by a remote developer, the user has little idea what the detailed control logic is, or what time values to insert. In such cases, a previously prepared module would be a convenient way to complete his model.

Control functions, such as task scheduling and interruption-handling, are standard within the framework of the SYSTEM/360 Operating System (OS/360). Therefore, for the large number of users wishing to study a system using OS/360, standard blocks of code modeling its functions could be provided to facilitate model building and ensure that the operating system is modeled with correct logic and timings. At the same time, submodels of control functions like these would provide the OS/360 developers with a convenient way to try out proposed changes. Such submodels are the subject of this study.

Scope of modeling effort

In the existing operating system concept, it is necessary to define the relationship between a device and its program support, since each device type is dependent upon the availability of such support. Some program modules are device dependent and remain in main storage only if that device is active. Others are not device dependent and may be called by many device support modules. It was decided that IBM 2311 direct-access storage devices, IBM 2400 series magnetic tape units, and IBM 1050 terminals would be supported in this initial modeling effort. Therefore, the programming support for only these devices was simulated, while other modules would be considered later.

Figure 5 illustrates the scope of the modeling effort. The functions in the solid boxes were included. Eighteen basic submodels were produced, including both basic and queued access methods

Figure 4 Line interruption handler

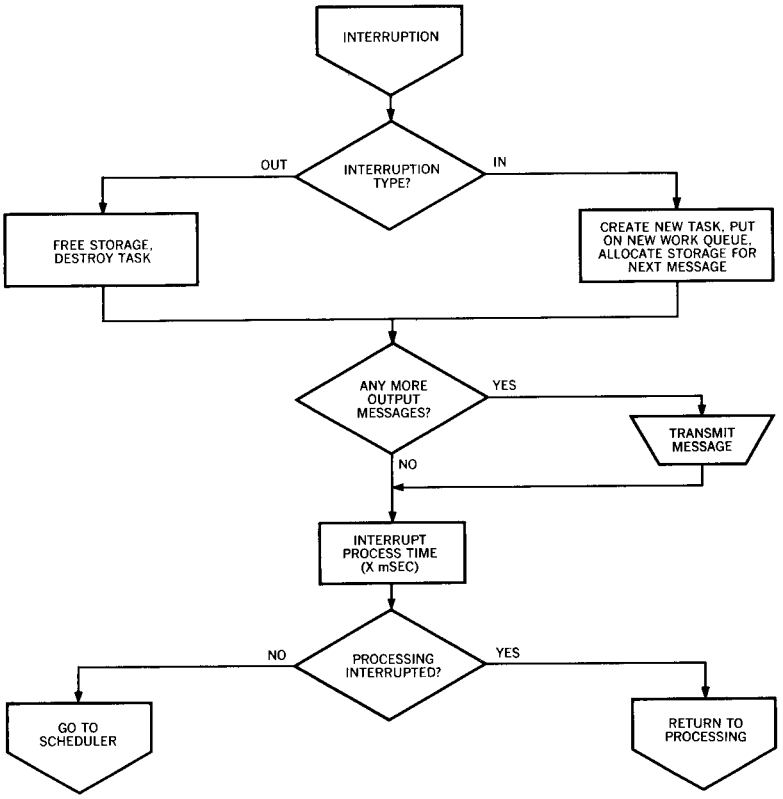
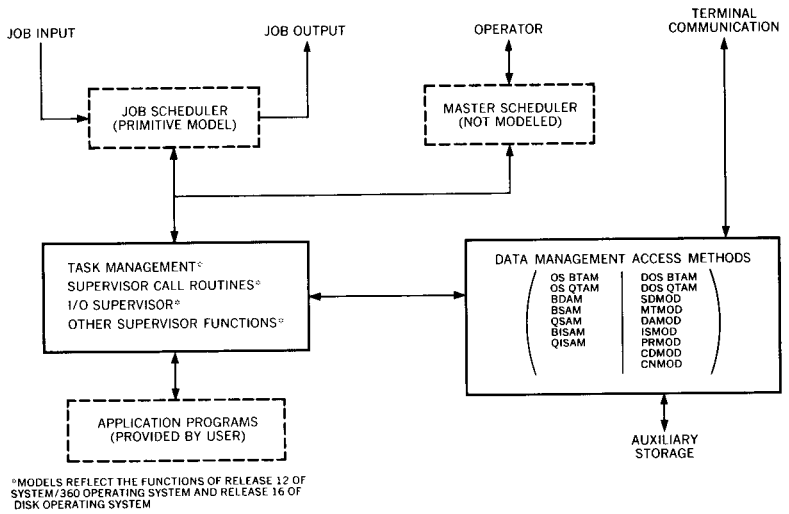


Figure 5 Schematic of OS/360



*MODELS REFLECT THE FUNCTIONS OF RELEASE 12 OF SYSTEM/360 OPERATING SYSTEM AND RELEASE 16 OF DISK OPERATING SYSTEM

Table 3 Simulated macroinstructions

Macroinstruction	*Function*
GET	Obtain next logical record
PUT	Write next logical record
RELSE	Release current buffer
TRUNC	Truncate an output buffer
DCB	Data control block

Table 4 File defining parameters

Blocksize
Buffers
Locate, move, and update modes of operation
Fixed- and variable-length, blocked and unblocked record format
Write verify

Table 5 Actual and simulated programs

Actual program

	LA	6,1000	Set up counter
LOOP	GET	INPUT	Read next logical record
	BAL	14,PROCESS	Process it
	PUT	OUTPUT	Write record
	BCT	6,LOOP	Branch per document
INPUT	DCB	(file definition information)	
OUTPUT	DCB	(file definition information)	

Simulated program

	1	REFTL	N22,,1/(file definition information)	
	2	REFTL	N25,,1/(file definition information)	
		MOVE	1000,SV9	Set up counter
LOOP		GET	$INPUT	Read next logical record
		PROCESS	1000	Process it
		PUT	$OUTPUT	Write record
		BRAD	SV9,1,LOOP	Branch per document
INPUT		DCB	1	Refer to file information in reference table (REFTL) 1
OUTPUT		DCB	2	Refer to file information in reference table (REFTL) 2

(each access method representing one module) operating under both the OS/360 and Disk Operating System (DOS)[5] supervisors. Schedule requirements and frequency of use were the bases for determining the functions covered. Functions required to start, schedule, and service devices in a multiprogramming environment were included, as well as supervisor calls, input/output routines, and access-method modules. The supervisor functions had to be carefully defined so as to be shareable by all access methods. User program entry and exit conditions were specified as in any programming system.

The major emphasis was on the simulation of teleprocessing jobs, with and without background job interference. The real-time program execution (i.e., polling terminals and processing the resultant message inputs) can be investigated by sampling intervals during which the teleprocessing job has no beginning or end. Therefore, the scheduling of jobs is not necessary to study the effects of background interference on the high-priority teleprocessing job. However, a partial facility to initiate background jobs was provided. Similarly, the functions of the master scheduler were not pertinent to the teleprocessing study and were omitted.

language

As previously mentioned, the simulated system is driven by user logic. After defining the configuration by means of regular CSS input, control is passed to the highest-priority program to be executed. User programs are executed in the user-defined partition and in user-preferred order. All statistics are based on the logic of the user's program, consisting of macroinstructions (GET, PUT, READ, WRITE) whose form and operands are similar to existing assembly language macroinstructions. The tabular module technique (using DCB's in OS/360 and DTF's in DOS)[6] was incorporated in the models in order to define the files. The various macroinstructions available in the simulated version of QSAM are listed in Table 3, and file parameters that may be specified are shown in Table 4. Similar instructions and parameters exist for the other access methods. The similarity of the simulated language to its assembler language counterpart is indicated in Table 5. Much of the ease of using CSS is derived from the ability to define user-oriented macroinstructions within the CSS language.

library

A library facility was incorporated in CSS to contain the various modules of the operating system model package. A MODEL statement allows the user to call those models that he wishes to use during a simulation run. For example, the statement

```
MODEL L4,OSSPVR,OSQTAM,OSS40,OSQS40
```

calls in the operating system supervisor, the operating system queued telecommunications access method QTAM, and the SYSTEM/360 Model 40 timing modules for both. Timing modules were independently specified so that the extensive logic in each submodel would not have to be duplicated for each different machine. CSS assembles the user programs with the models specified in his MODEL statement.

Implementation

The modeling project required the coordinated effort of two independent groups. One group, close to simulation development, was responsible for the detailed coding of the models, whereas the other group, close to actual system development, was responsible for logic definitions and timings. It was found that the spirited interchange of ideas produced a much more effective result than if either group had attempted the project alone.

To verify the logic and timings employed in the models, test programs were measured and compared with their simulated versions, both for execution time and equipment utilization. Discrepancies between the measurements led to the conclusion that the simulation results are extremely sensitive to device timing characteristics, such as tape interrecord gap or arm motion time. Further, measurements on several identical devices of actual characteristic timings showed that variances in these timings from their nominal values exist. If the measured device characteristics were used, the simulation run results were within three percent of the real program results. Fortunately, device timings are normal input parameters to CSS and values other than nominal specifications can easily be inserted. However, this indicates that there is little value in extreme accuracy in a general model if equipment tolerances are not correspondingly tight.

Model characteristics

The total operating system model library of 18 modules, plus associated timing modules, consists of over 10,000 CSS instructions, or roughly 150,000 bytes of storage. Most modules contain between 500 and 800 CSS instructions, while the associated timing modules contain about 100 individual timing segments each. These figures indicate the great amount of detail of these modules.

Fortunately, only the modules required by the user need reside in main storage. Table 6 shows the typical storage requirements for a large teleprocessing system, indicating that it would be rare to require main storage of more than 256K bytes to simulate a system. Also note the small amount of user coding compared to the space taken by the models. However, in addition to the application code, the user must supply the equipment configuration statements. In this example, the user supplied one-sixth of the total model statements, the rest being obtained from the library.

Models of systems with no terminals run on the order of five to ten times slower than real time when executed on an IBM SYSTEM/360 Model 40 processor. Including lines and terminals, which require time-consuming polling procedures, model running times drop to twenty to fifty times slower than real time when executed on a Model 40 processor. Execution time may be shortened by a factor of nine by running on a Model 65 processor, since CSS is processor bound. Use of the modeled Autopoll feature also greatly decreases model execution time.

Table 6 Typical storage requirements for large teleprocessing systems

	Storage requirements (kilobytes)
CSS equipment (500 terminals, 70 lines, etc.)	20
Transient entity pools (messages, tasks, events)	58
CSS models (OS/supervisor, QTAM, QISAM)	44
User application programs	3
CSS/360 program	42
OS/360 (including I/O buffer areas)	43
Total	210

The length of a simulation run is of concern to the model analyst, both from the economic viewpoint and from the fast turnaround time required to seriously consider the effects of many small changes. Many factors affect the running time. To meaningfully discuss these, one must first define the run time ratio, R, as:

$$R = \frac{\text{actual model execution time}}{\text{simulated elapsed time during model execution}}$$

Thus, if it requires ten minutes of execution time to simulate 1 minute of model time, $R = 10 : 1$, i.e., the model runs ten times slower than real time. This ratio may often be usefully expressed as the product of three factors:

$$R = (\text{average macro time}) \times (\text{model burden}) \times (\text{event density})$$

Each of these factors is discussed below with specific reference to CSS and the modeling effort.

average macro time

Macro here refers to the instructions of the model language, e.g., the CSS instructions. The execution time of such a macro is a function of the complexity built into it and the efficiency of coding it in the basic machine language (i.e., machine instructions per macro), as well as the speed of the machine executing the model. Since the timing for various macros varies greatly, an average must be struck for a typical mix. For instance, in CSS, the number of machine instructions per CSS instruction ranges from 30 up to 1,000, with the usual average of 200. Operating on a Model 40 processor, CSS averages about 15 microseconds per machine instruction, resulting in an average CSS instruction time of three milliseconds. This value drops to about 0.34 milliseconds on a Model 65 processor.

model burden

Model burden represents how many macros must be executed to support an event. An event is defined as one of the major occurrences in the model, the sequence of which constitutes a

run. For example, the processing of a message in a teleprocessing model would be an event. In this case, the burden would represent the number of CSS instructions executed to process a message from beginning to end, including all control service plus polling and interruption-handling instructions. It may be simply calculated after one run by dividing the total number of instructions executed by the total number of messages processed during the run. The burden is a function of several secondary considerations. One is macro complexity. For simple macros, the burden is large—that is, many simple macros must be executed to represent a complex event. However, this may be compensated for by fast macro time resulting from the simplicity. A similar consideration is the closeness of the macro operations to the real system functions being modeled. This is where CSS gains its principal advantage over GPSS in modeling computer systems. Another obvious consideration affecting model burden is the complexity of the model. Incorporating fine details adds many instructions per event. Not so obvious is the inclusion of fixed scans, such as polling, which do so many operations per minute, regardless of message load. This results in an increasing burden with decreasing message rate.

event density

Events per unit of model time is the rate at which system events, such as message arrivals, occur in the real system. Since these events must be processed serially by the simulator, even though parallelism is being modeled, the run time is at least proportional to the occurrence rate. That is, to model one hour's operation of a system processing 10,000 messages per hour takes at least twice as long as the same model processing only 5,000 messages per hour. It will probably take longer, because the model burden will be increased due to system procedures invoked to handle overflows and queues resulting from congestion at the higher rate. This may also result in longer average macro time, because the new macros called into play may be more complex than the regular mix. Thus, all three major factors may be related.

Applying these factors to a typical CSS run employing the operating system models, it is found that average macro time is three milliseconds on the Model 40 processor, model burden is 2000 CSS instructions per message, and event density is 1.5 messages per second. Thus,

$$R = \left(0.003 \frac{\text{execution seconds}}{\text{macro}}\right) \times \left(2000 \frac{\text{macros}}{\text{event}}\right) \times \left(1.5 \frac{\text{events}}{\text{model second}}\right) = 9 \frac{\text{execution seconds}}{\text{model second}}$$

This running time is largely a result of the extreme detail included in the operating system. It means that realistically only a few minutes of such a system's operation can be studied. This is no handicap with most teleprocessing-oriented systems, since stable operation can be determined within this time span. However,

studies of daily or weekly work schedules using this technique are out of the question. For such efforts, a much less detailed level is required, perhaps based on formulas rather than simulation, though quite possibly various operating parameters might be derived from the detailed run.

There is some hope that job shop environments which are not teleprocessing-oriented can be modeled. Here the job arrival rate (event density) is typically in minutes rather than seconds, resulting in a theoretical sixty-fold decrease in the run-time ratio. This would require a model of job scheduling functions, which were not incorporated into the experimental models. However, work along this avenue appears fruitful.

It has been suggested that an increase in speed could be attained by coding the models directly in machine language rather than using the higher-level CSS language. However, an investigation has shown that, except for isolated cases, the gain would not be so dramatic as the CSS/GPSS gain because the CSS instructions largely do what must be done in the operating system logic in an expeditious manner with little superfluous overhead resulting from generality. However, certain small operations, which are highly repetitive and consist of a series of elementary steps, have been coded in machine language to speed up the models. Even so, such an operation rarely accounts for more than ten percent of a total run time, and eliminating it altogether does not speed up the run by more than that ten percent. The trouble with the models is that they do so much, not that they are especially inefficient. Also, to code the operating system models in machine language would make them much less flexible, reducing their utility in aiding the design of new operating systems.

This raises an interesting question—for whom is the general model intended? It is now clear that the developers and the installation planners represent two incompatible user groups. The developers wish to use the model to judge how well a new system will perform and the effect of proposed changes to the structure. For developers, then, the model must be very detailed to be sensitive to minor changes. However, speed is not a critical factor. The installation planner, on the other hand, is willing to accept a much less accurate model, as long as it is fast and easy to use. The uncertain accuracy of his input data from the field makes any fine model detail unwarranted.

These models have been found relatively easy to use by both development and installation planning groups. The direct macro capability allows a user familiar with the supervisor and an access method (and having a superficial knowledge of CSS) to set up a complex model in a day that would formerly have taken him several weeks. However, these models are more suitable for the laboratory. A different modeling package from that discussed here may be required for general use in installation planning, one with grosser logic (and thus somewhat less accuracy) but an order of magnitude faster.

diagnostics

The basic CSS program makes a great many checks for erroneous conditions throughout the execution phase. The operating system model makes relatively few checks, partly as a result of its experimental nature and partly because such checks are just too costly in time to make in the high-level CSS language. As a result, debugging a model employing the library submodels can be cumbersome. This is a common problem in any system implemented in a higher-level language. In order for such a system to find the general use for which it is intended, the naive user should be shielded from the intricacies of low-level debugging as well as the labyrinth of operating system logic.

Summary comment

The operating system models have been used for both development and installation work. They can be modified to simulate the effect of a proposed change, as well as simulating the activity of a specific configuration of equipment to be installed. Since the models are miniature systems, they can serve as an educational tool, to elucidate the functions of the operating system and its relationship to user programs. The models have been written for general use. However, feedback from the pilot users indicates that two versions of the same package are really required: a detailed version similar to that described above, and a gross version with additional input options.

Some effort was made to speed up the package over that indicated. By rewriting the modules, coalescing many segments while still holding to the initial ten percent accuracy criterion, simulation running time was cut in half.

The development of a trace-edit feature would enhance the modeling effort by providing a trace of the execution of a user program and automatically editing it to CSS format. The initial package requires a user to define his own application programs. It is necessary to define a program's logic in this way if the program code does not exist.

In addition, a trace-edit program could be used to create a library of jobs. Typical jobs such as FORTRAN and PL/I could be traced and placed in edited form in the library to be referred to by a user calling for representative work loads. The feature could also be used to trace control program functions, such as job scheduling, and include them as an integral part of the existing models. The resultant input options could allow the user to simulate the scheduling of typical or existing jobs within a multiprocessing environment, with or without a teleprocessing load.

CITED REFERENCES AND FOOTNOTE

1. R. L. Gould, "GPSS/360—An improved general purpose simulator," *IBM Systems Journal* 8, No. 1, 16–27 (1969).
2. The Computer System Simulator is an IBM proprietary program.

3. G. H. Mealy, B. I. Witt, and W. A. Clark, "The functional structure of OS/360," *IBM Systems Journal* **5**, No. 1, 3–51 (1966).
4. P. H. Seaman, "On teleprocessing system design, Part VI, The role of digital simulation," *IBM Systems Journal* **5**, No. 3, 175–189 (1966).
5. G. Bender, D. N. Freeman, and J. D. Smith, "Function and design of DOS/360 and TOS/360" *IBM Systems Journal* **6**, No. 1, 2–21 (1967).
6. A. R. Cenfetelli, "Data management concepts for DOS/360 and TOS/360," *IBM Systems Journal* **6**, No. 1, 22–37 (1967).