Computer System Evaluation Through Supervisor Replication

John C. Cavours
Computing Science Department, The University of Glasgow, Glasgow G12 8QQ, UK

A class of tools that can be used for both computer system simulation and implementation is considered. Such tools can be viewed as two-level simulation programs. The outer level replicates the kernel of the operating system, contains the simulation control program and the event routines and is driven by events that are external and internal to the real computer system. The supervisor processes are replicated by coroutines, which obey the same code that these processes would obey in the real system. All user processes are modelled using a reentrant coroutine that forms the inner-level simulator. The two levels are interfaced by routines that trap the primitive calls issued by both user and system processes and schedule the corresponding events. Shadow tables associate the simulation entities with the real system variables. The tools can be used both for designing computer systems and for determining their efficiency.

1. INTRODUCTION

Experimentation with an actual system can only be applied after the system is put into operation. Furthermore, although software monitors can measure as finely as desired, in doing so they disturb the system being measured and are very expensive to run. Simulation on the other hand can be used as a tool by the designers of computer systems but has its own weaknesses and limitations. The basic drawback of conventional simulators such as those which have been surveyed by Neilsen, can be attributed to their many simplifying assumptions especially in modelling the proper resource ("facility") allocators in a computer system.

The resource allocator in a computer system is the supervisor of its operating system. The basic concept of our approach is to construct a tool which accurately replicates the supervisor of the real system in a simulation of its environment so that the overall system performance is measured by direct experimentation with the tool. Since such a tool will have the same logical structure as the software being modelled it is possible to use simulation throughout the design process. Performance measures can be used to provide feedback for the designer and the simulation code can become the implementation code.

Because the supervisor is forced to follow a quasi-identity simulation, the tool can also be used to evaluate the performance of the computer system once the latter is put into operation. The designers will have the ability to experiment accurately with a structurally valid model of the real system without the need for the physical hardware or workload to be present. Furthermore, such a tool can run as an ordinary user program without interfering with the system it is running on. Thus the method proposed combines the advantages of discrete simulation and experimentation with the actual system.

Since our tool should replicate the supervisor of the real system itself, it must be able to replicate all system processes and the real system tables and treat them in the same way as the actual system itself. In the following we are going to investigate the ways in which we can accurately model the events which drive the real system and the different stages of evolution of processes. Later on we are going to examine the implications of the absence of the actual hardware and workload.

2. MODELLING THE EVENTS WHICH DRIVE THE REAL SYSTEM

A computer system in operation is driven by external interrupts caused by events external to it and internal interrupts (traps and primitive or supervisor calls) issued by its processes. We can model such a system using an event-oriented simulation in which the occurrence of each interrupt is made an event and each interrupt service routine an event routine. If we do this, then, in general, our routines cannot always predict the next event (i.e. the type and the time this event is to occur). For example, an event routine which is invoked when a process issues a primitive call to unblock another process does not know what will happen once the call has been issued: the process which issued the primitive call (and which was executing) might continue running or it might awake a higher priority process in which case it will be preempted. If a process does not block or does not wake up a higher priority process when it issues a primitive call then our event routines can predict its next event (i.e. the time and the kind of the primitive call to be issued next) and schedule it to occur at the right time. If, however, the process blocks or wakes up a higher priority process, control in the real system is transferred to the dispatcher and our event routine cannot predict the time the process's next event is to occur, since this will be equal to the time it will be dispatched plus the time required to deal with the primitive call plus the time needed in order to issue its next primitive call, and we do not know the time the process is to be dispatched in the future (if at all).

Thus, in order to be able to predict the next event, our tool should be enhanced with an ‘uninterruptable dispatcher’ and the ability to model the set of processes which execute in the system at any time (i.e. the processes which are eligible for execution). The dispatcher should operate on the list of processes that can be dispatched.
(the ‘ready’ or ‘dispatcher’ list) and this list must contain the same processes that the corresponding real system list would have at the same point in time.

Two additional lists are required for this type of simulation, the list of next events or event list and an interrupt list. The former contains at most one entry for each eligible process in the system. The latter contains the event notices of those processes which were preempted due to the occurrence of an external interrupt.

The simulation control program selects the event at the head of the event list and invokes the appropriate event routines at the right times. The event routines which service the primitive calls manipulate the ‘same’ tables in the same way as the real system primitive service routines do and exit to the dispatcher or to the interrupted process. The event routines which service the external interrupts preempt the current process if the CPU is busy (i.e. they remove its next event from the event list and insert it in the interrupt list), perform their processing and exit to the dispatcher. The dispatcher in turn, either reschedules the next event of a process which was interrupted (by removing this event from the interrupt list and inserting it on the event list) or, as in the real system, invokes the process in question.

Note that the event routines must advance the simulated time by the time taken by the corresponding real system interrupt routines. Similarly, the dispatcher must take into account the time needed to save and restore the process’s status and the time needed to dispatch the process in question. In this way, the correct flow of events is maintained.

We have called the module which contains the simulation control program, the event and interrupt lists, the interrupt event routines, the dispatcher, and the tables containing the same information as the corresponding tables in the real system, the ‘kernel interface’ of the tool, to correspond with the name which is usually given to the nucleus of the supervisor of a computer system. We now proceed to examine the ways in which to model the different stages of execution of each process in the system.

3. MODELLING THE DIFFERENT STAGES OF EXECUTION OF PROCESSES

In the real system, when a process is dispatched it performs some computation and then it issues a primitive call. The latter causes entry to the supervisor which may attempt to acquire or release resources. Thus, in order to predict the correct ‘next event’, our tool must ‘know’ the exact structure and behaviour of each supervisor process in the system, and, because these processes affect the generation of future events, the contents of the system tables and the variables manipulated by these processes.

Computer systems do not satisfy the assumptions behind event oriented simulation and, even less, the assumptions of activity oriented simulation. We have thus resorted to the third approach to simulation programming, the process interaction approach which uses coroutines to implement the various activities. This approach has been previously applied to the simulation of computer systems by Weideman. Our approach, however, is based on the fundamental requirement that the tool should have the same logical structure as the software being modelled. We have thus tried to avoid the limitations of process interaction languages by enhancing our tool, as discussed in Sections 1 and 2, with coroutines, in such a way that there is one and only one coroutine for each process in the system. Furthermore, the coroutines used to simulate the supervisor processes must provide an exact replication of these processes. This of course implies that these coroutines should communicate with each other by issuing exactly the same primitive calls that the corresponding processes in the real system issue.

3.1 Supervisor processes

Since we insist on using the same primitive calls as the real system processes use (the ‘same’ code) we should provide the mechanism which maps the primitive calls of our coroutines to the sequence of events which drive the tool. The design of this mechanism may seem difficult at first sight, but it turns out to be quite simple. Coroutine primitive calls in our tool do not cause entry to the kernel as primitive calls of the real system processes do. Coroutine primitive calls invoke procedures which schedule events, to occur at the current simulation time (i.e. the time the coroutine was invoked) plus the time the coroutine has run. Obviously the latter can be calculated in a tool of this nature (by noting the time the coroutine is invoked and the time it invokes the procedure which schedules events) if the tool runs on the same host as the real system. If this is not the case, then the difference in times must be scaled appropriately.

In most cases, the reactivation point of a coroutine is the instruction following the primitive call. Some primitive calls however, those which could cause the process which issues them to be blocked, may be associated with two reactivation points, one if the process does not block and one if it does, in which case the primitive call might have to be reissued. The primitive servicing routines of the kernel (which would know whether the process has been blocked or not) set up the correct entry point in the appropriate process descriptor field. Thus, the procedures which accept primitive calls and schedule events have to perform the following tasks:

(i) save the reactivation point(s) of the invoking coroutine,
(ii) insert the appropriate event notice in the event list, and
(iii) exit to the simulator control program by, for example, using a non-local GOTO.

If the implementation language provides procedures with different entry points then one procedure with several entry points can deal with all the primitives.

Note that this solution is similar to that used by IBM’s VM/370. Virtual machine monitors intercept certain instructions for interpretive execution rather than allowing them to execute directly on the bare machine. These interpreted instructions include I/O requests and most other supervisor calls. We will now examine the modelling of the user processes.

3.2 User processes

Since in most general purpose systems the nature and behaviour of user processes may be unknown, these
processes have to be modelled in a probabilistic manner. The number of active user processes in a system may be quite large and there may be several independent events associated with each user process. In general, these include I/O requests, creation and deletion of subsidiary processes, segment or page faults and termination (normal or abnormal). The order in which these events occur varies from process to process and their occurrence is conditional on the corresponding coroutine being invoked. Thus we cannot predict such events beforehand and put them on the event list whose entries are ordered by the absolute simulation time. 

We have to ensure that each user process coroutine takes care of the sequence of events within its associated process and, when it is invoked, schedules at most one next system event or issues one primitive call. We can accomplish this by making the user processes's coroutines implement an independent simulation program within our tool. Each user process coroutine can implement its own simulation control program, simulation routines and event list. This list need only contain an entry for each different event type and these entries need to be ordered relative to each other only. By adopting a separate event list for each user process in the system, there is no need to keep the process identifier in each entry of the list since this information is kept in the corresponding process descriptor and the 'choice' of the coroutine is accomplished by the dispatcher. 

Since there are only a small number of different event types within each user process the event lists of these processes can be kept small and the use of this approach leads to significant reductions in running time. Note that this approach is similar to the 'indexed list algorithm' suggested by Vaucher and Duval. Although not the same, 'the basic idea of grouping notices to reduce scan time is sound'. Furthermore, unlike their algorithm, our 'grouping' does not need self-monitoring and a feedback mechanism to keep track of the 'optimum' values of the time interval sizes and to reorganize them when necessary. 

Our tool now consists of a two-level simulation: a simulation within a simulation. The 'inner simulation' models the execution of the user processes. The 'outer simulation' is partly driven by the former and partly by itself (includes the interrupt routines and the system processes) to model the overall behaviour of the system. The simulation control program of the user processes coroutines always removes the event notice at the head of its list and invokes the appropriate inner simulation event routine. There are four types of such routines. 

The first type deal with the events which are internal to the process in question only, i.e. events which do not occur in the real system and hence the outer simulation. For example, in a demand paging system we could have a 'working set change' event which could 'occur' at fixed length time intervals to determine the size of pages which are going to be accessed within the next interval. Each such routine performs its processing (which can include scheduling other events), schedules the next event of that type and returns to the inner simulation control program. Notice that this is the only type of coroutine which exclusively schedules events internal to the process in question. For the remaining types which can also schedule outer simulator events, the outer (absolute) simulation time (T) displacement is calculated from the inner (relative) event time (t) using the statements:

\[ T = \text{process remaining time} - \text{process run time} + t; \]
\[ \text{process remaining time} = \text{process remaining time} - T; \]

At \( t = 0 \), i.e. the first time the process in question is invoked, its remaining time is set equal to its total run time.

The second type of inner event routines deal with 'unconnected' events such as page faults. Each such routine inserts the notice in the outer simulation list and returns to the outer simulation control program. Routines of the third type deal with events whose occurrence implies other events. Such events include I/O and creation and deletion of son processes. Each such routine inserts the event in the outer simulation list, predicts the event that should follow and inserts its notice in the inner simulation list before returning to the outer simulation control program. Notice that in addition to the above functions, routines of type two and three have to set the reactivation point of the user process coroutine to its inner simulation control program. When the coroutine is reinvoked the process will be repeated.

The fourth and final type is concerned with the issuing of primitive calls that can cause the process to be blocked. Such routines include 'waiting for I/O', 'waiting for creation', etc. Each such routine has to insert the notice in the outer simulator list, set the two reactivation points of the user process coroutine and exit to the outer simulation control program. If the user process is entered at the first reactivation point (i.e. if it has been blocked) then the primitive call is immediately reissued (scheduled to occur by invoking, say, the routine that traps primitive calls and schedules events). When the user coroutine is entered at its second reactivation point (the process was not blocked or has now been unblocked) then some post processing of the primitive is performed, the next event that should occur is scheduled, i.e. its notice is inserted in the inner simulation event list, and the routine returns to the inner simulation control program.

Note that system processes can be simulated using 'non-reentrant' coroutines, i.e. a coroutine code module for each different system process. (There might be an exception in the case of device managers which share their code in the real system). In fact, by definition, system processes are monitors, i.e. 'non-shareable' processes. All user processes must be simulated by a 'reentrant' coroutine program, i.e. one sharable code module and separate data modules for each. If this is not done we will have as many code modules as the number of entries in the process descriptor table and our method would be infeasible in practice. As all user processes' coroutines perform the same functions and use the 'same' variables this can easily be accomplished.

4. IMPLICATIONS OF THE ABSENCE OF THE HARDWARE

Event-oriented simulation of the operation of 'active' hardware units such as peripheral devices is usually straightforward. We need to observe, however, that in our tool the simulation of device operation should be
done within the corresponding 'INITIATE IO', 'TEST IO' and 'transfer completion interrupt' event routines.

In a virtual memory system, when a process is dispatched its next 'event' may be a page or segment fault. In addition, all pages which were in core when a process is suspended and which were replaced by the page throwing mechanism might be required to be brought in before or after the next event of that process takes place (this could well be a 'page or segment fault' event).

'Primary' page faults, i.e. page faults which would occur if the process is not suspended, can easily be modelled by the user coroutine program invoking its 'working set change' event routine 'immediately' when it is entered for the first time. The 'working set change' routine will cause specific page faults to occur during fixed length time intervals, as explained in the previous section. If, however, the process under consideration is not permanently memory resident, multiple 'secondary' page fault events could occur both before and after the 'next event'. Such events cannot be rescheduled by the user coroutines themselves for, by definition, these coroutines have different reactivation points and can be suspended at any stage.

Thus, in order to model correctly a virtual memory system we must first of all enhance our dispatcher with the ability to 'know' which page or segment the process was accessing at the time it was suspended and whether this page is now core resident or not. If the page in question is not in core then our dispatcher will have to cause the page fault immediately. This takes care of the page the process was accessing when it was suspended. For the other thrown pages (at least those which belong to the process's current working set), the most straightforward solution is to let the page throwing mechanism (of the 'store manager process' coroutine) schedule their secondary faults, i.e. to insert them in a simulation list. Now the question arises as to which list these event notices should be inserted into.

Obviously, since the process has been suspended, the 'secondary' page fault events cannot be kept in the outer simulation event list. This list contains at most one next event for each process and its notices are ordered on the absolute simulation time. Putting the secondary page fault events together with the 'interrupted' next event in the 'interrupt list' is also not desirable for the following reasons:

(i) If this list has more than one event for a process it would have to be ordered.
(ii) The number of entries will become very large and the arguments put against the use of one overall event list will apply here as well.
(iii) Since secondary page fault events can occur not only before but also after the interrupted next event, our dispatcher would have to repeatedly compare the entries of this list with the entry at the head of the user process's list.

For the last reason, it is not worth introducing an interrupt list for each process either. The solution which satisfactorily solves our problems is to insert all secondary page fault notices and the notice of the interrupted next event in the event list of the process itself. The notice of the interrupted next event should be marked as suspended and, provided that the page the process was accessing when it was suspended is in core, all our dispatcher has to do is remove entries from the user process's list and insert them in the outer simulation list until it finds the suspended notice. At this point the process is marked as resumed and from there on the user process coroutine is invoked as before. Note that care should be taken not to mark as suspended an interrupted next event of a process which has already been suspended (has not yet been resumed).

A final point needs to be elaborated. When an event is interrupted, we can calculate the remaining time $T$ of this event by subtracting the current time from the time the event should have occurred. The user process's list is ordered on the relative times $t$; $t$ is calculated from $T$ using the 'opposite' statements of (1):

$$\text{process remaining time} = \text{process remaining time} + T; \quad (2)$$

$$t = \text{process runtime} - \text{process remaining time} + T.$$

5. IMPLICATIONS OF THE ABSENCE OF THE WORKLOAD

One can recognize two modules which provide interfaces between a real system and its workload. The first is providing an interface between the users (jobs) and the system. We have called the corresponding module of our tool the 'job scheduler interface'. The second module provides the interface between processes and their associated programs and is called the 'process creator interface'.

5.1 The job scheduler interface

The job scheduler interface of the tool should not only perform the functions of the corresponding module of the real system (i.e. create, select for activation and delete jobs, initiate and drive the spoolers, respond to user requests etc.) but it should also have the ability to predict the arrivals of users at their terminals and the requirements of all the jobs in the system, i.e. it must include all the functions of the 'job generator' routine$^5,7$ used in conventional computer system simulators. Most of these simulators assume that jobs and not processes acquire and release resources. Clearly, this assumption is invalid in any modern system.

In modelling a process organized system there are two further considerations. Since a process's service requirements depend on their associated job they can be sampled and generated from the corresponding characteristics of the job under the constraint that, say, the sum of them does not exceed the corresponding job characteristic. Furthermore, since these processes can have a hierarchical structure and since the creation of the processes of a group is not performed in a sequential fashion but it is interleaved with the creation of processes from other groups, it is obvious that this hierarchical order (tree) of modelled processes belonging to a group should exist before the processes themselves are created in the system. Thus, the job scheduler interface has the additional responsibility of setting up the tree of modelled processes and filling its nodes with their characteristics.
Since our tool should give reproducible results in different runs and since the same jobs can be chosen at different times in different systems (for example because of a different job scheduler), we must associate a pseudo-random number with each job in the system. These random numbers uniquely specify the job's characteristics as well as its associated processes' characteristics. Again, since multiaccess and batch jobs could arrive at different times in different systems (for example, because of different speeds of input devices, different I/O managers and/or different user attributes), we must associate a random number with each type of workload from which the individual random numbers of jobs (and subsequently processes) are generated. Thus, no matter which system is being tested, exactly the same jobs and processes will be generated but not necessarily accepted (contrast this with some other simulators or analytical models).

Although it is convenient to structure the tree of modelled processes in the same way as the corresponding tree of the real system (the links of which are kept in fields of the process descriptor table), the nodes of the two trees do not need to contain the same information. The latter tree contains the information which is used or gathered during the execution of the process itself and it is an exact replication of the real system tree. The former tree contains primarily the 'unknown' simulation characteristics of the process, i.e. those which must be known beforehand in order that the latter can be filled correctly at the time the process is created and those which are used exclusively by its inner simulation. Borrowing VM/370 terminology we have called the tree of modelled processes the 'shadow process table'.

Note that if we want all our tables to be an exact replication of the real system tables we should also have a 'shadow job table' which contains the simulation characteristics of the job. The necessity to separate the 'simulation' from the 'real' characteristics is not so vital, in this case, as the jobs in a system are not usually connected or ordered in any way. Strictly speaking in VM/370 terminology the name 'shadow' is given to 'combined' tables. In a similar way, other 'shadow' tables can be envisaged. This brings us to the system module which provides the second interface with the workload: the module that creates the processes.

5.2 The process creator interface

'Process creation' is usually allocated to a monitor outside the kernel since it might involve reading the associated program file. If no file needs to be read then the following discussion applies mutatis mutandis to the 'create process' primitive. Since in our tool program files do not exist, it follows that although our process creator should have the same logical structure and follow the same logical steps as its real system counterpart it should operate in a slightly different fashion. Functions such as determining a 'free' descriptor in the process descriptor table and setting up 'memory' and other tables can be done in exactly the same fashion as in the real system. Reading of the program file can also be accomplished by invoking a 'disk manager' process. Our process creator interface, however, will have to take its information from our shadow process table and not from a program file.

Furthermore, it has to 'connect' the shadow process table entry and the descriptor of the process created (allocate a pointer to each) so that our tool will be able to simulate–execute the creator process.

6. CONCLUSION

Our research effort has shown that a tool based on the concept of replicating the real system supervisor by embedding it in a simulation of its environment so that the overall system performance can be tested by direct experimentation can in general be constructed and should consist of the following modules:

(i) A kernel interface which is driven by events which have a one-to-one correspondence with the real system events (external and internal interrupts). This interface includes the event-oriented simulation control program, the event routines each of which provides an effective replication of the corresponding real system interrupt routine and an 'uninterruptable' dispatcher (which provides a replication of the real system's low-level scheduler). In brief, this interface models the nucleus of the real system supervisor and the physical hardware and, therefore, its design is influenced by the absence of the latter from the tool.

(ii) A reentrant coroutine program which independently models the execution of all user processes. This is an event oriented simulator with its own simulation control program, event lists and events. The design of this coroutine program is influenced by the design of the user processes' run time support system and hence the actual system itself.

(iii) A set of coroutines, one for each supervisor process in the system. The structure of some of these coroutines (job scheduler, process creator and user processes' interfaces) is influenced by the absence of the real jobs (workload), program and other files and actual user programs/processes. The remaining coroutines (store manager, medium-term scheduler, device managers, spoolers, file system monitor et cetera)—those which only deal with the execution of processes and the medium-term management of the hardware resources—are an 'exact' replication (include the same code) of the corresponding system processes of the real system. This is accomplished by:

(iv) Routines which trap the primitive calls from the coroutines and convert them into scheduled events before returning to the overall system simulation control program in (i) above. These routines perform the same function as the control program (CP) or IBM's VM/370 virtual machine monitors (VMM) entirely in software.

The overall tool can be considered as a two-level simulation. The inner level is provided by (ii) above, whereas the outer level ((i) above) is partly driven by the former, partly by the simulated system coroutines ((iii) above) and partly by itself to model the overall system. The connection between the various modules is provided by a 'shadow' process table which contains the unknown (simulation) characteristics of the modelled user processes and by the replicated real system (process,
page/segment, device and job descriptor etc.) tables. Finally, the connection between the shadow process table and the replication of the (real) process descriptor table is established by the 'process creator' (coroutine) module of the tool.

![Figure 1. Layout of the software tools.](image)

The overall structure of the tool is shown in Fig. 1. Upper case shaded modules correspond to the simulation proper, upper case to the coroutines which provide interfaces, lower case to the coroutines at the same level of abstraction and lower case shaded to the routines which transform primitive calls and schedule events.

Our software tools make use of several different types of tables. The entries of these tables are connected with each other as shown in Fig. 2. Shaded fields contain information which is used by the 'simulation' part of the tool, whereas unshaded fields contain information which is used by the real system. To simplify the diagram all fields and table entries have been drawn as having the same length.

Notice that the method used to collect statistics in such a tool is basically the technique followed by the 'event stream' approach used in many real-life systems.

7. CRITICAL EVALUATION

For a general discussion of our method as a computer system design methodology the reader is referred to Ref. 8. In this section we evaluate our technique in terms of the conventional simulation and experimentation with the real system approaches.

7.1 Advantages over conventional discrete simulation approaches

(i) Flow of control (sequence or events) for a tool constructed using our method is exactly the same as that in the real system. The modules of such a tool are interfaced using the same communication primitives as the real system.

(ii) If the tool and the operating system to be evaluated are written in the same language, existing system subroutines can be used as an integral part of the tool and vice versa.

(iii) The tool can simulate 'reentrant coroutines': if a set of coroutines have similar code and use no STATIC variables other than the set of data associated with their primitives, then they can 'share' the same program. For example, one coroutine program module can be used for each type of I/O handler (dedicated, shared and terminal system).

(iv) Software development: run-time support systems, database systems and other operating system components can be developed and tested in the tool before they are included in the real system.

(v) The method allows the natural breaking up of the 'event list' of conventional simulators into several 'private' sublists, resulting in an efficient tool.

(vi) Statistics gathering routines and options can be (and should) be delegated to the real system itself. Thus the 'simulator' part of the tool needs only to plot and analyse these statistics. Furthermore, the number of parameters required by such a tool is far smaller than the number required for conventional simulators. Since the tool includes the real system, it can compute a large number of its 'parameters' and can still use 'trace-driven data' if such data are available.
(vii) Similarly, the simulation and the real system parts of the tool can take care of their own input and output. Thus, a completely parameterized program producing a vast amount of output can be implemented in a simple and modular fashion. Furthermore, default cases for the hardware and operating system parameters can be readily incorporated.

Note that within a given framework, a variety of hardware and operating system configurations can be simulated by changing the parameters of the tool. Similarly, a variety of device characteristics, sizes of system tables, operating system overheads and parameters of scheduling algorithms can be tested in the framework of the computer system introduced. If, however, a different set of devices, different communication and synchronization primitives or different algorithms are required, these have to be rewritten, recoded and reintegrated in the tool.

7.2 Advantages over experimentation with the real system

(i) Design, compilation, deadlocks and other logical errors can be detected and corrected in the tool which can run as an ordinary user program without interfering with the system it is running on or with the normal execution of the machine.

(ii) Evaluation of system behaviour: a simulator workload is reproducible (using the same parameters, pseudo-random numbers and timing considerations). Thus a controlled, repeatable environment can always be provided without the need of additional hardware.

(iii) Programs can be developed and debugged for machine configurations which are different from those of the host.

(iv) Concurrent testing of dissimilar executive/operating systems with the same nucleus. For example, one designer may be checking out a new release while others are using a tried and proved version.

(iii) and (iv) above are also advantages of VM/370 over conventional (‘non-virtual’) operating systems. Our kernel interface can however be made to trap any instruction desired whereas some systems do not lend themselves to ‘virtualisation’.

7.3 Disadvantages compared with experimentation with the actual system and the VM/370 concept

(i) Our tool does not really run the operating system, since it does not consider real files, programs and commands. Similarly, the environment generated does not simulate the low-level problems such as the programming of I/O processors. However, it does allow the designers to specify the system processes and their interactions without being concerned with details such as to which data sets space is to be allocated on backing storage.

(ii) Our approach ignores protection mechanisms as there is no isolation between independent processes. The tool has no way of determining whether a coroutine is accessing and altering the data (or even the code!) of another existing coroutine. This can be a problem if the host machine has no hardware protection features and the tool is implemented in a (high-level) language which provides no such features.

(iii) Certain modules of the tool, such as the interfaces, could require a larger amount of code or a higher degree of complexity than the corresponding modules of the real system. The run-time efficiency of a tool produced by our method depends directly on the efficiency of the operating system which it models.

(iv) The above discussion assumed that a tool needs to have all the replicated real system tables in the primary store. This is not the case in most real-life systems. Although a tool can use the host machine’s backing storage, this could introduce problems of: (a) communication; (b) dependence on the host machine/operating system; and (c) artificial replication.

8. APPLICATION

Our method has successfully been applied in the construction of a tool for a hypothetical, medium-scale, virtual memory, multiprogrammed, spooled, general purpose job shop operating system. In implementing the tool, the aim was not only to show the applicability of the method presented but also the feasibility of writing it in a high-level language.

The computer system chosen embodies the nucleus and hierarchical approaches. The supervisor monitors were allocated to different levels according to the urgency of the tasks they have to perform and the ‘grain of time’ in which they operate. The kernel interface implements processes, process synchronization and communication (using a mailbox message passing scheme) and short-term scheduling. The job scheduler interface deals with both batch and multiaccess users, provides long-term scheduling (using time since arrival within external priority group) and communicates with the medium-term scheduler and the spoolers. The process creator interface has the responsibility of creating and deleting processes.

The (medium-term) store manager is capable of dealing with both segmentation and paging and uses an elaborate replacement policy based on special cases, process priorities, LRU approximation and the working set principle. The medium-term CPU manager (process scheduler) adjusts the internal priorities of processes (using simplified policy-functions), provides load control and prepares the dispatcher list. Finally, the I/O managers are separated into dedicated, sharable and virtual device managers and a terminal system manager. A trivial file system monitor was implemented in order to provide the correct flow of events.

The complete tool at this stage occupies more than 71 pages of PL/1 listing and corresponds to around 2700 source statements. When run it occupies about 300 Kbytes (IBM 370/168, PL/1 F compiler) or more depending on the sizes specified for the system tables. It can have a ratio of simulation time to real time of less than 2 given appropriate workload characteristics, working set quantum time, policy functions and other factors which affect the eligible set size. Note that none of the multitasking facilities of PL/1 were used. For more
details on implementing such a tool in a high-level language with no multitasking facilities the reader is referred to Ref. 15.

As the efficiency of our method directly depends on the efficiency of the operating system it evaluates, the above figures were considered reasonable for the excessive degree of detail provided. The operating system designed is roughly ten times larger than the RC4000\textsuperscript{13, 16} and about ten times smaller than MULTICS.\textsuperscript{17} It should be noted that because our method ‘insists’ that the system be resident in core and because of our modest attempt (approximately four man years) compared to the mighty MULTICS, this difference is not unreasonable.

Acknowledgements

My thanks are due to Professor D. C. Gilles, Mr W. Findlay and Dr J. Haselgrove of the Computing Science Department of the University of Glasgow and to Dr R. H. Davis of the Department of Computer Science of Heriot–Watt University.

REFERENCES

15. J. C. Cavouras, Implementing a simulation tool in a high-level language with no multitasking facilities. To be published.

Received June 1982