WORKING NOTE AC84

Alan Creak
9 October 1992

ABOUT POSS

DESCRIPTION

BACKGROUND AND AIM.

POSS, the Practical Operating System Simulator, is intended to present aspects of the behaviour of operating systems to students. I want it to do so in two ways : by supporting assignments, and by making it possible for students to experiment with – particularly – interactions between different parts of the system, which it is difficult to present convincingly otherwise. So far as can be managed, this experimentation should be unconstrained, so experimenters should be able to try out ideas not predefined in the system. Our slogan : as much as possible should be POSSible.

The proposal grew from experience with several simulators which I have used in assignments over the past few years to illustrate various aspects of operating system behaviour. In some cases, I caused the students to write their own simulators; later, when it became clear that even writing a fairly simple simulator was an onerous task for an assignment, I wrote the simulators myself and asked the students to carry out experiments with them. These proved successful, but my products were fairly rudimentary, so in 1990 I asked for programming assistance to extend them somewhat and particularly to improve their interfaces.

I had thought at the time that a combined simulator could be valuable, but realised that it would be a big job, so aimed only at the easier target of separate simulators. In discussing my request with several people academic and technical, it began to seem better to aim directly for the combined version, largely on the principle that it would be silly to go over the same ground twice. In the end, it was agreed that this approach would be better, so that's what happened.

The minimum requirement for the combined system is that it should include all the essential behaviour of my original simulators. In addition, though, it should be possible for students to write some of their own simulator code; while writing a complete simulator is a difficult task, many of the students who did so in my earlier assignments commented that they had learnt a great deal from the experience. A system which provided support for presentation and general organisation into which a student could insert the code for some system component could be valuable.

Although the simple simulators were effective in the jobs that they were designed to do, they were limited in two ways. First, pressure of time meant that I never provided as much flexibility in the simulators as I would have liked, so they were individually less instructive than they might have been. Second, and more important, they were all separate, so they could not illustrate the interactions between different parameters. These are the main defects which I wish to remedy in constructing the combined simulator.

A further useful development, and one which would become even more useful with the proposed greater range of facilities in the combined system, is in the presentation of results. The original simulators typically gave their results in the form of performance logs of various sorts, sometimes with some provision for selecting the sorts of event which were to be logged. Additional provision for graphical output of historical behaviour, and also animation of current behaviour, could be very helpful; but the logs are useful too, and must be preserved.

STRUCTURE.

It seems fairly obvious (which is not necessarily to say correct) that the best way to make sure that the simulator will work sensibly in all cases is to stick as closely as possible to simulating a real computer with a real operating system executing real programmes. The clever trick will be to find what we can leave out without seriously damaging the simulation.

I think this means we have components representing processor, memory, code, devices, etc., each with such adjustable parameters as are needed to determine different aspects of their behaviour, and which can talk to each other in standard ways which must allow for the communication of the most elaborate items of information which may be requested. Specialised versions of the components can then be used in specific simulations, tailored to suit the immediate requirements of the experiment – for example, in some cases the memory manager won't be required to do anything at all.

A point to consider is the possible requirement for extensions in the future. How easy would it be to move on to a simulation of a distributed system communicating through message passing ? I think it should be "easy", provided that we get the initial design right. An appropriate design will be be modular, so that individual components can be changed without affecting the rest of the system, and the modules must be chosen to represent all functions of the computer system – including processor and operating system – which may be of interest. For example, I include the processor in this discussion because, even though it has always been simple in the simulators which have been used so far, we shall certainly want to investigate the behaviour of multiprocessor machines in the future.

A single simulation may be interesting and informative, but commonly what we want is an understanding of how various combinations of parameters determine the behaviour of the system. For example, we might ask how the size of a process's memory requirements affects the performance of the virtual memory system, or how the number of processes using a particular resource affects the dispatcher. In other words, we often want to run a sequence of simulations, recording the value of some dependent measurements as certain independently variable parameters are varied over certain ranges of values. A higher level of control is therefore important, with which such experiments can be programmed. It must be able to identify the parameters to be changed, the measurements to be taken, and the way they are to be presented.

IMPLEMENTATION.

Elaborating on the remarks above, the simulator must operate as a set of interacting components, each largely self-contained so far as its own behaviour is concerned but depending strongly on other components in various ways. It is also necessary that most of the components (to be safe, assume all components) should be independently replaceable by other versions which perform the same function in different ways. It is also desirable that the repertoire of the simulator should not be limited by its implementation; as far as possible, a student who wishes to replace one of the provided components by a new version should be able to do so.

Perhaps the obvious, but not the only, model for such a system is that of a collection of objects. While that isn't intended to prejudge the best implementation method, object-oriented techniques must come high on the list. Each component type would be defined as an object, with the different variations defined as specialisations. The objects have the required properties : they are independent and maintain internal state, and the hierarchic organisation can be used to maintain an invariant interface independent of internal implementation. In an object-oriented system, these would naturally be incorporated as different specialisations of a parent class which defines the view seen by the rest of the system.

There are other approaches too. One which I'd look at closely if starting again from scratch is to use Linda as the coordinating and synchronising base, with the different components written as separate programmes exchanging messages through Linda's tuple space. It would still be necessary to define interfaces, but otherwise programmes would stand alone and need not fit in with any other constraints. A potentially valuable degree of freedom is that in principle any language could be used for any component.

Whatever system architecture is chosen, there are different ways of providing the programmability which I emphasised earlier. I sketch two ways in the next two paragraphs; there are undoubtedly more. Both methods give the student much flexibility in designing and constructing an esoteric version of a module of the simulator, but they achieve this end in different ways.

- The first method is to exploit the modularity of the system, and to require that each version be separately encoded in whatever source language is used and "plugged in" to a predefined interface. This is perhaps the most flexible, though opinions differ on just how practicable it is. A significant difficulty is to define the interface, particularly if this includes not only the basic operating system function but also access for logging and monitoring routines which may be included to improve the simulator's knowledge of what's happening but which have little to do with the simulated system. I suspect, but cannot prove, that it should be possible to separate these requirements; if it is, this approach should be workable.

- The alternative which I've used in some of the simulator assignments which I've set is to have a single simulator for each component, but to make this programmable in some appropriate way so that it can produce the behaviour of a wide range of versions. The programming is generally a matter of making some choices and specifying some parameters. In practice, this means that all properties of the system which are potentially relevant to the component must be accessible, and that means for adjusting the values of such parameters as are required (and of protecting the values of such as must not be changed) be provided. The nature of the "programming" depends on the component; it may be a matter of making choices from a menu, or it may require a small programming language. An advantage of this approach is that default and other standard versions of the components can be provided as predefined programmes, so it is not necessary to write multiple versions of the simulator code for different variants of the components.

At the simulator level, it should be possible to run the simulator manually for single exploratory simulations, but there should be a way to programme experiments as outlined in the previous section. This specification is quite independent of those which define the system itself, though it may include instructions on how the system-level parameters are to be fixed. This level does not deal with the simulated operating system, but with the simulator itself. Generally, any systematic choice of simulator parameters should be programmable at this level, so that systematic studies of relationships between adjustable system characteristics and system behaviour can be investigated. Such choices include both varying a numerical parameter over a set of values ("for values of memory size from 10 to 100 in steps of 10"), and choosing different elements from a set for a non-numeric parameter ("run the simulation with programmes A, B, and C"). There must also be provision for specifying which system variables should be logged, and under what circumstances.

THE STUDENTS' VIEW.

A proper system specification should include some characterisation of the expected input to the system, and the corresponding behaviour and output. To that end, I append a draft users' manual, amusingly entitled POSSUM (Practical Operating System Simulator Users' Manual), intended for students to use. The manual is by no means complete. I have concentrated on the behaviour of the system, and have therefore left undefined matters pertaining to its implementation – such as by what means the parameters which describe the behaviour are to be specified. In other places, I have gone beyond realistic expectations for an initial implementation; see the SPECIFICATION which follows for a more practicable proposal.

I have also said little about the presentation of the results of the experiments. I would rather begin by making sure that the results can be collected and recorded, and then worry about the display. It seems to me that there are two types of display (apart from the detailed log, which is always useful) which are of interest : fairly traditional presentations showing the effect of variable parameters on different aspects of system performance, and animations of detail of the system's operation. The first of these can be addressed by presenting straightforward graphical displays – plot A against B, and more elaborate, but equally well known, variants and extensions thereof. The second is going to need a lot more work if the aim is to end up with a coherent set of displays which cover the whole system, and I am far from ready to start on that now.

SUMMARY, ADDED AFTER WRITING THE POSSUM.

In my earliest simulator assignments, I asked the students to write their own simulators. They did, and some at least enjoyed it a lot, but it was really too much work, and not everyone ended up studying the phenomena which I'd intended them to. After that, I took to writing simulators with comparatively simple specification languages; they worked better, but weren't very flexible. I still liked the idea of having students write real simulation code, though, and one of my reasons for thinking about a more general simulator was the hope that they could then achieve useful results by writing only one component of the whole system.

After writing this users' manual, I'm not so sure. My change of mind is occasioned by the complexity of the environment into which any such component has to fit. Generally, every component of the simulator has to satisfy constraints of three sorts :

- Constraints imposed by the simulated system. Any component has to fit in with the requirements of other components with which it interacts.

- Constraints imposed by the monitoring requirements. The simulator itself will require access to information (details yet to be determined) which it uses to provide the requested reports or other indications of performance.

- Constraints imposed by variable parameters. The whole simulator is to be driven by the experiment plan, which must be able to adjust any adjustable parameter in the system in order to investigate relationships between different parameters and the system performance. Any adjustable parameter in a component would therefore have to fit in with whatever discipline is adopted for control by the experiment plan.

While none of these constraints is likely to be particularly taxing, to satisfy all of them may well turn out to be a considerable task. If students are to write their own code for the simulator, I foresee a significant number of them spending a lot of time trying to get interfaces right, and otherwise occupying themselves with non-constructive activities.

A simulator constructed of programmable modules, though perhaps a little harder to implement, will at least be guaranteed to get all this communication overhead right once and for all, leaving the students free to concentrate on the phenomena of interest. It will be necessary to design the means of specifying each module with some care, but my experience with the simulators I have already used suggests that this is not a serious problem. I have used fairly conventional textual languages; some sort of form-filling technique could be just as good, though an advantage of the textual approach is that similar instructions will carry over without change to the experiment plan level. It is less obvious that a form-based method would be adequate for the experiment plan.

All these arguments apply with about equal strength to both the modular programme and Linda designs. The programmable module pattern is therefore now my favoured approach.

POSSUM :

PRACTICAL OPERATING SYSTEM SIMULATOR USERS' MANUAL

DESCRIPTION OF THE SIMULATOR.

The operating system simulator is designed to illustrate the behaviour of operating systems under a variety of conditions. It simulates an operating system which can be tailored in various ways to isolate some sorts of behaviour for separate study, or to explore the interactions between different parts of the system. The simulated system includes a set of typical system components. The system is supposed to run in a computer with characteristics which can also be varied by setting appropriate parameters; and it executes a programme – or a set of programmes – which can be written to investigate various sorts of system behaviour. Figure 1 illustrates the main components of the simulator and the relationships between them.

The simulator proper is in the central box. Its operation is controlled by a set of parameters, some of which will usually be set to some required values before a run, leaving the rest with default values. When it runs, it reproduces – more or less realistically – the behaviour of the system defined by the parameters, and produces a record of what happened during the run. This can be recorded as text in a log file, or it can be displayed directly on the screen. You can also watch the behaviour of the system as it runs using a collection of animation displays which depict the current states of various aspects of the system.

For a single simulation, you can set the parameters from the keyboard, but for a systematic investigation of the effect of some parameter on the system's behaviour, you will need to conduct more detailed experiments. To do so, you can draw up an *experiment plan*, which specifies the sequences of parameter values which you wish the system to investigate and the results you wish to collect, and have the whole plan executed by the experiment manager.

It is a fairly complex package, but for many purposes you may not need to worry about much of the detail. So far as is possible, all parameters are supplied with sensible default values which should lead to plausible system behaviour. You can therefore concentrate on just one part of the system, and assume that the rest will work sensibly.

The simulator simulates two things : processes, and the operating system environment. The two are essentially independent, and both must be described if you want to use anything but the default versions. In general, therefore, you have to provide both a programme and a configuration before the simulator will do anything useful for you.

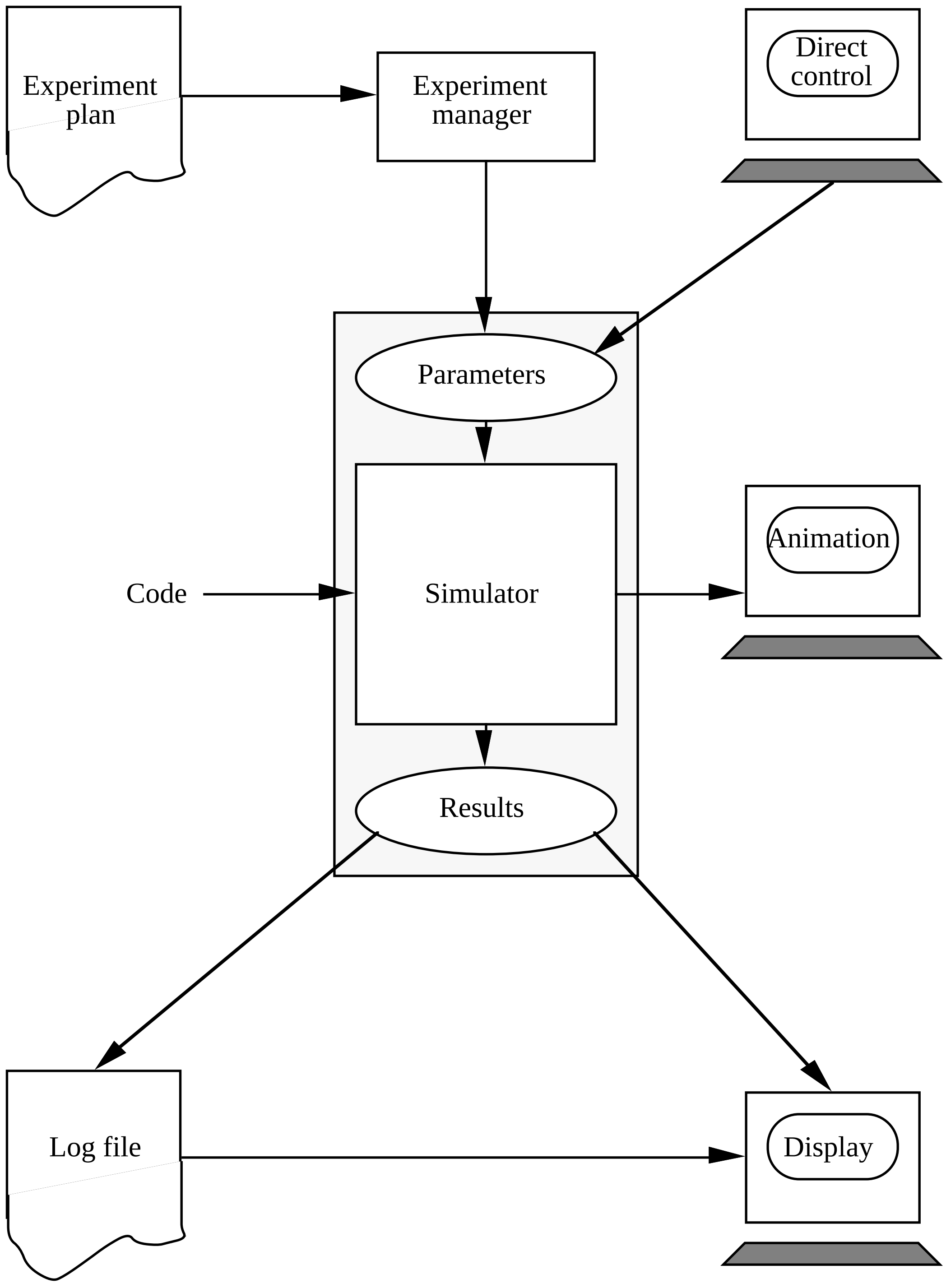

Figure 1.

The function of a **programme** is to represent a process's pattern of resource requests. The resources concerned are the processor, memory, and various devices, and a primitive programming language is used to define the number and sequence of requests, and the programme structure. The language does not provide any facilities for useful computation, but you can say how long each part will take.

The function of the **configuration** is to prescribe the system environment in which the programme will run. This includes any parameters available : memory size, time slice, scheduler queue discipline, memory management algorithm, how many processors, etc. Many of these are associated with specific parts of the system, and will be described in the appropriate sections below.

You may also provide an **experiment plan**, which will specify a sequence of runs of some process or processes, executing a specified programme, or combination of programmes, with characteristics which may be fixed or may vary from run to run according to the values of certain parameters. The configuration must also be defined either explicitly, in the configuration specification, or in terms of parameters set by the experiment plan, or by default. Then the system will be started, and the experiments conducted as directed. The experiment plan must also specify what measurements of performance are to be taken, and how these are to be displayed or recorded.

THE SYSTEM SIMULATED.

The computer system consists of several modules, which perform independently as far as possible. They are :

Hardware;
Processor manager;
Process manager;
Memory manager;
User interface manager;
File manager;
Communications manager;
Device manager;
Clock.

A variety of system parameters and tables are available for general use. Some can be set to control the system's behaviour; others can be read, but not changed.

THE PROGRAMMES.

The programmes are necessary to define patterns of resource requirements. The simplest requirement is for a processor; next we might want to define how much memory is required, what external devices are used, how memory is used, and so on. A *very* primitive programming language is used to define a programme; it exhibits enough programming language features to model common programming practice, but has no data manipulation instructions – though it *does* have data declarations, so that unshared memory can be defined, and corresponding statements of the size of code areas to specify potentially shared memory. Individual instructions occupy no space, and, except for the RUNFOR instruction, take no time to execute. Here is a list of the instructions available (Alan and Bruce, documents around August 1991).

PROGRAMME *NAME* ... END *NAME*
: Introduces and names a programme. One or many programmes may be used.

SUBROUTINE *NAME* ... END *NAME*
: Introduces and names a subroutine. Any number of subroutines may be used.

SIZE *SIZE*
: Used in any programme or subroutine to declare the size of the code. If no SIZE declaration is present, it is assumed that the code is not significant for memory management – in effect, it is assumed to be of size 0 and always present.

DATA *SIZE*
: Used in any programme or subroutine to declare the size of the data. If no DATA declaration is present, it is assumed that no additional data space is required. DATA areas are assumed to be unshared, so if several processes use a programme, the code areas may be shared but data areas are not.

CALL *SUBROUTINE*
: Used to enter a subroutine. There is no limit on the depth of nesting, but recursion is not permitted. (As there is no conditional instruction, it wouldn't work anyway.) There is no explicit instruction for returning from a subroutine – the return happens when the end of the subroutine's code is reached.

REPEAT *NUMBER* TIMES ... END REPEAT
: The instructions between TIMES and END are repeated as instructed. REPEAT instructions can be nested; REPEATs and ENDs are associated in the corresponding way.

REPEAT FOREVER ... END REPEAT
: The instructions between FOREVER and END are repeated indefinitely.

REQUEST [*NUMBER*] *RESOURCE*
: One or more of the specified resources are requested. The process is suspended and the appropriate device manager invoked. What happens then depends on the device manager.

RELEASE [*NUMBER*] *RESOURCE*
: One or more of the specified resources are released. The process is suspended and the appropriate device manager invoked. What happens then depends on the device manager.

RUNFOR *NUMBER* TICKS
: Undefined processing continues for the length of time specified. This is the only time-consuming instruction; if you want any other instruction to take time, associate it with an appropriate RUNFOR. Interrupts are possible between ticks.

RUN
: Provided for convenience – equivalent to RUNFOR RUNLENGTH TICKS, where RUNLENGTH is a system variable which has the value 1 by default, and can be set from the experiment plan.

WAIT *SEMAPHORE*, SIGNAL *SEMAPHORE*
: Wait and signal on semaphores as defined. The semaphores themselves are set up as part of the system configuration.

The *default programme* makes no demands on any resource except the processor (not even for memory); it runs for ever – or, at least, until stopped by the experiment plan's TIMEOUT.

```
PROGRAMME DEFAULT
REPEAT FOREVER
        RUN
END REPEAT
END DEFAULT
```

THE CONFIGURATION.

At this level, you define the sort of computer and system that you want. This includes the size of the computer, the algorithms used by the operating system modules, and any parameters needed to define the details of these specifications. Each module has its own requirements, so they are dealt with in turn below.

Processor.

With a single processor, there isn't much to specify. (Perhaps the length of a tick ? – maybe later, but not now.) Eventually, we shall want to say how many processors are present, and perhaps something about how they interact.

Default : One processor.

Memory manager.

The two fundamental decisions needed to specify the memory behaviour are the size of the memory and the management technique to be used. After that, each memory management technique has its own requirements, but only one of these is needed in any single simulation.

The management techniques available are those based on paged memory, segmented memory, paged segments, and overlay areas.

Default : 10 pages, paged.

For *paged memory*, you must specify a page replacement algorithm. Three standard methods are supplied – cyclic, first-in, first-out, and least-recently-used. You can also write your own algorithm which identifies the page to replace by evaluating some function depending on a set of attributes of pages maintained by the simulator.

Default : cyclic.

For *segmented memory management*, strategies for both choosing an available segment to use and choosing a segment to replace must be provided. Algorithms provided for choosing a vacant segment are cyclic first fit, best fit, and worst fit; replacement strategies are as for paged memory.

Default : cyclic first fit for vacant segment searching, cyclic search for replacement.

For *overlay management*, a rather different sort of information is needed. You are required to specify how many overlay areas you require, and which groups of subroutines of the programme should be allocated to which area.

Default : none. There is no simple way to derive an overlay allocation automatically.

Process manager.

There are three active components in the process manager : the scheduler, which decides when to start a new process and which process to choose; the dispatcher, which manages the ready queue and decides which ready process should be executed; and the interrupt manager, which decides what should be done after an interrupt.

The scheduler must be given a list of programmes, and instructions as to how frequently one is to be started, and how it should choose the programme to start. There must also be a specification of which programmes are to be running at the beginning of the simulation.

Default : The DEFAULT programme is running at the beginning of the simulation; no other programme is started.

The dispatcher needs a specification of how many queues it should operate, and how these are related. It needs to know how to decide in which queue a process should be placed when it is first made ready, and how to decide from which queue it should select a process for execution. Queues may be associated with different timeslice lengths.

Default : One queue; no other decisions are necessary.

The interrupt manager must be told what to do in response to any interrupt. This will usually be a matter of moving a process from a waiting queue to another – often the ready queue – within the system. The clock interrupt is slightly special; the interrupt manager must know how to decide to which queue a process should be transferred if it is interrupted by a clock interrupt,

Default : For a clock interrupt : replace the interrupted process on the end of the active queue of lowest priority. No other behaviour is defined.

The process manager also manages semaphores. An arbitrary number of semaphores may be set up when the simulator is configured.

Default : No semaphores.

Device manager.

The device manager is mainly concerned with queues and timing. Its function is to accept requests for service from processes, to estimate the time required for service, to place the process on its request queue, and to generate an interrupt at the correct time.

Default : Only disc defined; all requests take 100 ticks to satisfy.

Clock.

The clock can either issue regular interrupts separated by a constant number of ticks, or it can be programmable.

Default : Regular interrupts, one at every tick.

THE EXPERIMENTS.

In specifying the programmes and configuration, you describe the system which is to be simulated. Once that is done, you want the simulator to run – but you also want to know what it did and what it's doing. You therefore need an additional level of control at which you can say what has to be done and what you want to get back.

At the lower levels, the computer systems, operating system, and executing programmes are fully defined; at the experiment level, you can define experiments which you wish to conduct at the lower levels. Typically these will involve taking measurements of one or more indicators of the system's behaviour for several combinations of values of various parameters. For example, you might wish to determine the relative effectiveness of two different dispatching disciplines as the number of processes active in the system increases from a lower to a higher limit. Provided that you can define the experiments you wish to perform by a reasonably simple algorithm which varies the lower level parameters in a systematic way, you should be able to specify them to the experiment controller.

Once you have the results, you will want them presented in some form. You may want to plot graphs, or to tabulate values in some way. You can specify these too at the experiment level. You may give these specifications before the experiment starts, when they may involve any available variables, or after it is completed, when you may only use variables which have been saved.

SPECIFICATION FOR POSS

This specification is informal, based on qualities which I identify as experience and common sense. The project is big enough to be worth proper analysis and specification, and I would welcome any contributions to that end. I have tried to demonstrate that, though the whole project is very large, it can be implemented incrementally as something like a series of prototypes.

COMPONENTS.

The simulator system comprises two major components, the **simulator** and the **experiment manager**. The first requirement is the simulator, which should be able to stand alone executing specifications entered directly by the experimenter. The experiment manager is an "automatic experimenter", issuing instructions and collecting results much as a human experimenter would. It is driven by an *experiment plan*, a programme of experiments devised by the experimenter.

SUGGESTED DEVELOPMENT PROCEDURE.

Main scheme :

```
Construct the simulator skeleton.
Implement one module.                        –   Whichever happens to be convenient.
Refine the module.                           –   See below for Refine.
Construct the experiment manager skeleton.
Extend the manager to handle the new module.
Tune the extension.                          –   See below for Tune.
For all the other modules                    –   In whatever order happens to be convenient,
                                                 obviously taking into account dependencies
                                                 between modules.
    Implement the module.
    Refine the module.
    Extend the manager to handle the new module.
    Tune the extension.
```

Refine a module :

```
Repeat until happy
    Test the module.                         –   Both performance and interface.
    If not happy
        Change as required.
```

Tune an extension to the experiment manager :

```
Repeat until happy
    Test the extension.                      –   Both performance and compatibility with the
                                                 direct control interface.
    If not happy
        Change the extension OR Refine the module as required.
```

SPECIFICATION FOR THE SIMULATOR

COMPONENTS.

The simulator components fall into three groups. The **simulator proper** comprises those components directly involved in simulating the computer system, while the **interface** is concerned with setting the system parameters and finding out what it does during the simulation. The third component is the **skeleton**, the framework composed largely of data structures defining the system which maintains the current system state and makes it accessible to the rest of the programme.

Skeleton.

Simulator proper :

Simulator core;
Processor manager;
Process manager;
Memory manager;
User interface manager; – The simulated computer's interface, not the simulator interface.
File manager;
Communications manager;
Device manager.

Interface :

Input interface;
Animation interface;
Output interface.

The relationships – or, perhaps better, a possible set of relationships – between these parts of the simulator are illustrated in Figure 2. The interactions between the components are of two sorts, those primarily involving data requests or alterations, and those involving control. Apart from external transfers to and from terminal devices and files, the data transfers are – at least notionally, but see later comments on the skeleton – between the skeleton and the separate components of the interface and the simulator proper. (Notice that this separation of function may solve the "complicated interface" problem which raised a question as to the practicability of replacing complete modules with new code.) Control interactions radiate from the input interface. The diagram shows the major control interactions, intended to support a normal simulation.

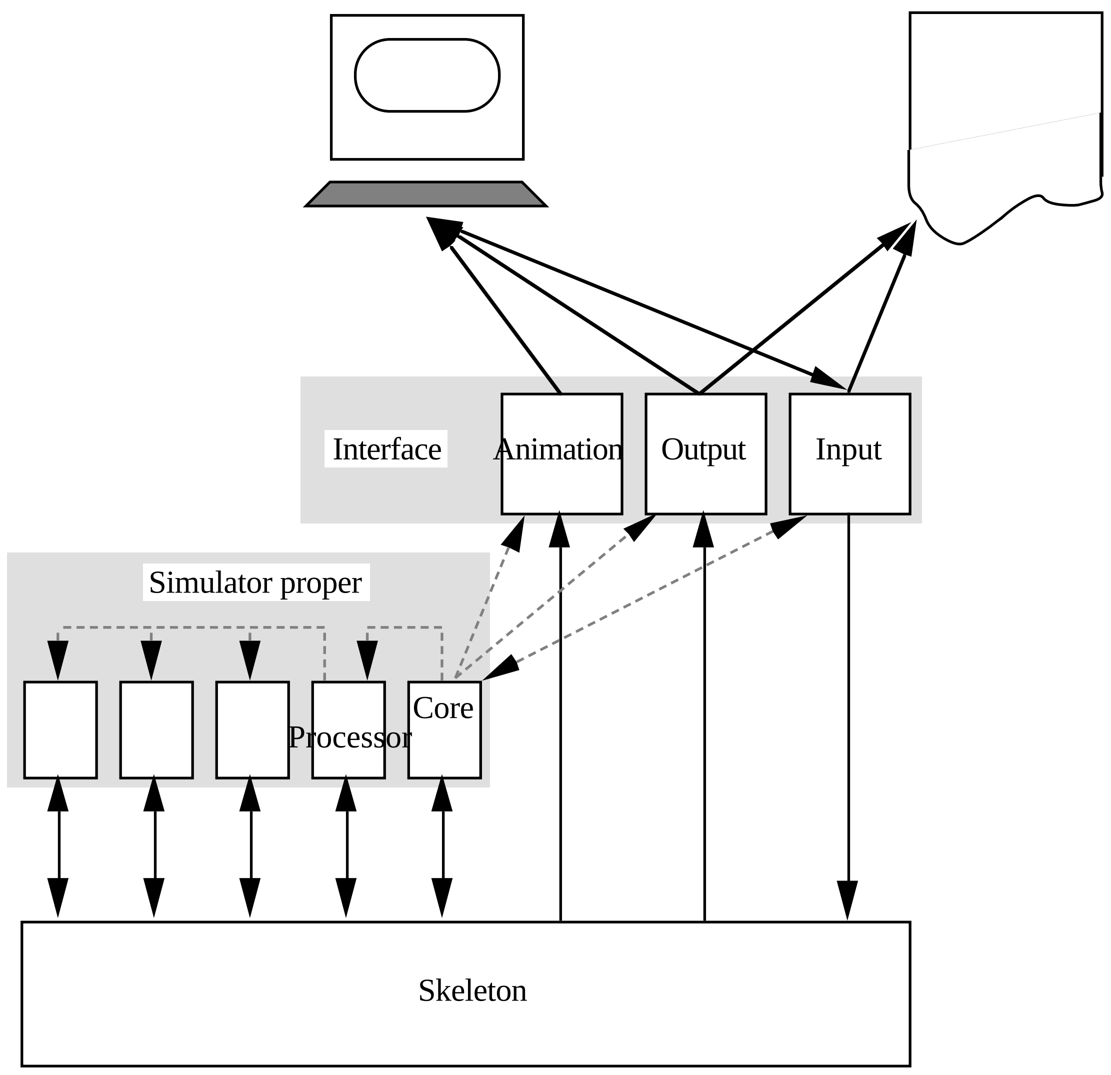

Figure 2

The sequence of events begins with instructions from the experimenter (or, equivalently, the experiment controller – the distinction is irrelevant at this level) which will establish the nature of the experiment to be simulated by setting the values of certain parameters in the skeleton, and then instruct the simulator core to begin the simulation. The simulator core will read the first item in its event queue. The queue is a part of the skeleton, and will probably have been set up to contain a processor event, and perhaps a clock event, if the clock is to be used. The simulator core will therefore execute the processor – or perhaps one of the processors, depending on the configuration.

The processor state is determined by the contents of its memory and its programme counter, also parts of the skeleton and established during the definition of the experiment. The processor will "execute" the instruction identified by its programme counter, perhaps calling on other resources in so doing. When the interpretation is complete, it will modify the programme counter accordingly, schedule another processor event at the time appropriate to the instruction, and return control to the simulator core.

The simulator core will then check its queue again. If the first item has the same scheduled time as the item just completed, the simulator will execute that too; if not, it will poll each of the interface components in case they have any actions to perform at the current time, or the experimenter has interrupted the interpretation; then, unless instructed otherwise, it will continue with its next scheduled event.

The interface components are polled at each change to the simulator time. The output and animation components can retrieve information from the skeleton to generate new output or to change the display. The input component must also be polled in case the experimenter wishes to stop the simulation or change parameters.

DESCRIPTIONS.

The skeleton is not trivial, as it must include all the basic structure which ties the rest of the system together. The most important function of the skeleton is communication; it holds the record of the current state of the system, which is made accessible as appropriate to such components of the whole as need to use or to change the information. The skeleton therefore defines a standard view of the system, as seen by all the modules it contains. That is valuable because we want the incremental steps of development to focus on the new components as they are added; we do not want to have to alter the fundamentals in every cycle. Despite its importance, the skeleton need not appear as a separate unit in the simulator. In an object-oriented implementation, it may take the form of a collection of interface definitions which must be maintained by the appropriate elements of the other system components.

INTERFACE : The skeleton must be able to accept requests for data which it holds, and changes to the data.

Configuration :

Running :

Setparameter(<table>, <item>, <value>)

Readparameter(<table>, <item>)

The simulator proper is the active part of the model. Its task is to execute the instructions in the programme or programmes which are executing in the simulated computer, and to maintain the system state tables held in the skeleton. The simulated system is represented as a set of tables and other items, described more fully in a later section entitled THE SYSTEM MODEL. The machinery of the simulator is also included under this heading; the descriptive note above gives some idea of how it works, and further details can be found in a subsequent section on THE SYSTEM MODEL.

The simulator core is the simulator's controller. It must manage the simulator cycle itself, incorporating the event queue, and the event interpreter. These are described further in a later section entitled THE SIMULATOR.

INTERFACE :

Configuration :

Settimeout(<time>)

Running :

Run( )

Stop( )

The processor manager manages all processors in the simulation. It maintains the processor states of all processors, executes instructions when required – possibly calling on other services for actions required – and handles process switching.

INTERFACE :

Configuration :

Running :

Step(<processor>)

Interrupt(<processor>)

Switchprocess(<newprocess>)

The process manager administers the process table. (I hope it will eventually be able to cope with several computers in a loosely coupled distributed system, when there will be several process tables.) It handles any change in process state, including changes to the process's memory table and the administration of system queues. The process manager also sets up and maintains any semaphores required in the simulator configuration.

INTERFACE :

Configuration :

Running :

Newprocess( )

Removeprocess(<process>)

Setstate(<process>, <newstate>)

Makesemaphore(<name>)

Swapout(<memoryarea>, <discaddress>)

Swapin(<memoryarea>, <memoryaddress>)

The memory manager is the system side of memory management. (The process's view is maintained by the process manager.) It maintains the memory map for the physical memory, responds to requests for memory, and handles virtual memory as required.

INTERFACE :

Configuration :

Setmemorysize(<size>)

Setusedbit({ off | on })

Setdirtybit({ off | on })

Setmemorymodel({ overlay | paged | segmented | pagedsegments })

Setpagereplacement({ cyclic | FIFO | LRU })

Setsegmentsearch({ firstfit | cyclicfirstfit | bestfit | worstfit })

Running :

Requestmemory(<size>)

Releasememory(<address>)

Usememory(<address>)

Changememory(<address>)

The user interface manager is the manager for the simulated computer's interface, not the simulator interface. This component is *NOT YET DEFINED*, but will be.

The file manager looks after each process's file table and file transactions. It could not unreasonably be considered as a part of the process manager, in the same way as the process's memory view, but is kept separate partly for convenience and partly because it will often not be used.

INTERFACE :

Configuration :

Running :

Fileopen(<fileidentification>, <mode>)

Fileclose(<fileidentification>)

Fileread( )

Filewrite(<size>)

The communications manager is concerned with interprocess communication, and, particularly, communication between processes on different computers in a distributed system. But that's later : this component is *NOT YET DEFINED*.

The device manager controls transactions between memory and devices. No data are moved, though the volume of data may be specified. The manager's function is either to suspend the process for a suitable period of time or to cause an "operation complete" interrupt if the process continues, and to take care of resource allocation if appropriate.

INTERFACE :

Configuration :

Makedevice(<name>, <howmany>, <latency>)

Paralleldevice(<name>, { yes | no })

Running :

Readdevice(<name>, <size>)

Writedevice(<name>, <size>)

Interface :

The three components of the system interface have less complex interactions with the rest of the system, but are likely to be more complicated internally. (This apparent simplicity may be illusory – certainly as more elaborate displays are required it is likely to become more complex, but for the moment it isn't too bad.) Except for the poll, necessary to guarantee the components time to perform any actions they need to do in a single-processor single-process machine, the transactions are between the input interface and the others, and are instructions to do with the material to present as results. As the animation interface remains UNDEFINED, and the output interface is deliberately kept in a primitive state for the immediate future, things can only get worse.

The input interface

INTERFACE :

Configuration :

Running :

Pollinput( )

The animation interface.

NOT YET DEFINED.

The output interface records system variables as instructed before, during, and after an experiment. Values during an experiment are only written if there is a change from the previous set written; the time is always included. The values are written to a log file as simple text.

INTERFACE :

Configuration :

Reportfile(<filename>)

Reportbefore(<variable>)

Reportduring(<variable>)

Reportafter(<variable>)

Running :

Polloutput( )

SPECIFICATION FOR THE EXPERIMENT MANAGER.

COMPONENTS.

Skeleton;

Input interface;

Display manager.

DESCRIPTIONS.

The **skeleton** of the experiment manager contains most of the works. Its job is to interpret an experiment plan expressed as a textual programme, presenting instructions as appropriate to the simulator, and retrieving and collating the result files produced for retrieval by the experimenter.

Initially, the programme representing the experiment plan will be constructed as a conventional text file by the experimenter; later, as the direct control interface to the simulator evolves, the plan language should evolve compatibly, at which stage it will probably require its own **input interface**. To maintain compatibility, the input interface should translate the instructions received into a textual programme.

The **display manager** is not necessary in the original implementation, but should be included in the development schedule after two or three simulator modules have been implemented, at which point the system will be capable of generating complex results. Its function is to facilitate exploration of the results produced during the experiments. This is likely to require extensions to the skeleton so that it forms indexes of the files which it collects from the simulation. The basic function of the display is to present the data collected in graphical form as requested by the experimenter. A basic set of graphical presentation functions will be available from the simulator's output interface, but additional facilities extending the display repertoire to include ways of presenting the combined results of many experiments will also be useful.

THE SYSTEM MODEL.

The model of the computer system itself is organised as a set of structures representing different elements of hardware, software, and abstractware which the system needs for its operation. In this section, some nomenclature and definitions are introduced.

A COMPUTER SYSTEM is a set of system components, a set of resource groups, and a set of process groups. It is a state machine, with the states defined by the current association between resources and processes. We are interested in the transitions between states. Transitions may be initiated either by actions of the system components.

A SYSTEM COMPONENT is an active thing belong to the system. Processors, clocks, and devices are common system components. A process is not a system component; its actions are initiated by a processor executing its programme. Every system component class contains one or more component instances and a component behaviour, which is a description of the component's activities. All system components operate independently, and they are the only parts of the system which can initiate actions. They do so by inserting events in the simulator's event queue.

A RESOURCE GROUP is a resource (a set of things), a resource manager which looks after them, a resource management protocol which defines what the resource manager does given the current system state, a resource table which records the current disposition of the things which constitute the resource, and a resource behaviour which describes what the things do. The "things" which constitute a resource may be concrete (devices, processors) or abstract (memory segments, locks).

A RESOURCE MANAGER accepts requests for its resource from processors running processes, and responds according to the protocol. The response will probably involve some change to the resource table, and may result in some change in the state of the process. While resource managers could be system components, they are more realistically seen as processes which require the services of a processor to run. Nevertheless, it may be convenient to implement default managers as separate components, so that they can be operated without interfering with the rest of the system.

A RESOURCE TABLE contains an entry for each unit of the resource it describes. Entries can be added to, removed from, and modified within all tables. Different resources may need tables with different structures; some may represent sets, some queues, and so on. A resource table is visible throughout the simulator, as any component may need information about the system state, and we cannot predict beforehand just what information the different components may need. A resource table may only be changed by its own resource manager.

A RESOURCE BEHAVIOUR must describe the stimuli which a resource can accept, and what it does in response. A disc can accept a read instruction, and will respond with some data after 10 milliseconds. (It's up to the resource manager to decide how to handle the response.)

The PROCESS GROUP includes a set of processes, a process manager, and a process table.

A PROCESS is an abstract entity which may be created or destroyed. It is characterised by a process behaviour, otherwise called a programme, which describes a sequence of operations which may involve requesting and releasing resources. A process does nothing unless associated with a processor. All current processes are listed in the process table. It will probably be reasonable for the simulator to impose a limit on the number of processes which can be active, but the limit should be an adjustable parameter.

The PROCESS MANAGER administers the process table. All changes in the state of a process are effected by the process manager.

THE SIMULATOR.

The simulator's job is to accept a system model of the sort described and prescriptions for an environment and an experiment, and to perform the experiment. It must also provide certain services : logging, statistics, repetitive experiments, and anything else that seems like a good idea. The model is independent of the simulator.

Generally, the simulator must set up the computer system according to parameters specified in the environment and experiment, start the computer system by executing an appropriate resource manager, and monitor the ensuing events.

In principle, the computer system should be self-contained; in practice, this is a simulation, not a real system, and there will be loose ends. The simulator has to tie up these loose ends and make them look like proper parts of the computer system so far as the rest of the system is concerned. For example, the simulator has to provide new requests for processes as required by the experiment.

PAGE REPLACEMENT ALGORITHMS.

I append this section as an example of a programmable resource manager.

Looking at a collection of page replacement algorithms in Finkel's book (and in one by Maekawa et al., which didn't add a lot), and adding an element of introspection, it seems that this set of parameters attached to each "real" memory page covers most of the possibilities.

Link for available queue.
Link for FIFO queue.
Link for LRU queue.
"Available" bit.
"Dirty" (changed) bit.
"Used" bit.
"Can be swapped" bit.
Last use time. (Time = clock tick count.)
Number of references.
Priority.
Owner (process identifier, for multiprogramming simulations).

Most of these are maintained solely by the simulator, but a few ("Can be swapped" bit, Number of references, Priority) can be changed by the page replacement algorithm.

A few system parameters must also be available :

Time now.
Identity of the process requesting memory.

THEREFORE :

These quantities must be available to people trying to investigate original page replacement algorithms. An original algorithm can be presented either

(a) as a piece of Pascal, plugged in as a specialisation of a SelectPageForReplacement object, or

(b) as a built-in specialisation of the same object which will accept some sort of programme.

If (b) is chosen, some sort of degfinition language must be devised. The rest of this section illustrates how such a language can be defined.

A LANGUAGE.

A language in which these programmes can be written must have names for each of the parameters, expressions with which they can be combined if need be, and ways of writing the selection algorithm to be used.

Names for parameters.

That's fairly easy. Some suggestions :

Link for available queue :	QAvailable.<position>
Link for FIFO queue :	QFifo.<position>
Link for LRU queue :	QRecency.<position>
"Available" bit :	BAvailable
"Dirty" (changed) bit :	BDirty
"Used" bit :	BUsed
"Can be swapped" bit :	BSwappable
Last use time :	NLastuse
Number of references :	NReferences
Priority :	NPriority
Owner (process identifier, for multiprocessing simulations) :	Owner

<position> ::= First | Last | <number>

(For lazy people, the capitals are unambiguous. Case is, of course, not significant.)

Expressions.

An ordinary set, such as (not necessarily exhaustive) :

(<expression>)
if <logical> then <expression> else <expression>

(<logical> is better than conventions about 1 = true, etc.)

<number> + – * / > < = <number>
- <number>
<logical> and or <logical>
not <logical>

Selecting a page for replacement.

The material above has defined only the information available to the page replacement algorithm, and the operation which can be used with the information. *Only what follows must be supplied by the experimenter*. I have presented it here in textual form; it would obviously be easy enough to devise a menu interface to do the same job, but it would be harder to integrate such a menu into the experiment manager. I think that there are four ways to select the page :

Take the first or last or Nth entry in a queue.

```
Use <queue>.<position>
```

Search memory for the page with the maximum value of some function of the parameters.

```
Maximise <expression>
```

Follow a queue looking for the next page to satisfy some condition. (The search may be unsuccessful.)

```
Seek <condition> in <queue>
```

Work through memory, perhaps cyclically, looking for the next page to satisfy some condition. (The search may be unsuccessful.) (I originally lumped this method with the previous queue search, but I think that the potentially cyclic search makes it sufficiently different to merit special treatment, if only because it implies that there may be some sort of "current position" marker which makes no sense with the queues.)

```
Seek <condition> in memory [ cyclically ]
```

- or, of course, perform a sequence of such operations until one works.

```
<algorithm> or <algorithm>
```