WORKING NOTE AC131

Alan Creak
2001 May 24

A NEW DEFINITION OF PROCESSES

The conventional definition of a process is identified (roughly), and generalised to include all activity in a computer (or anything ?).

What is a process ? Everybody talks about processes. Even I have talked about processes, in an earlier note[1] written (originally as an E-mail message) on sabbatical leave while I was revising the notes for the lecture course which Robert and I shared for a long time and which we were hoping would become a book. There I wrote :

> To talk about what computers do, we must develop ways of talking about how we can deal with information in the computer, and how the computers behave. ... To deal with information, we invent *files* ... and *streams* ... To talk about the behaviour of computers, we invent *processes*. ...

I am not at all sure that when I wrote that I realised quite how far I was out of step in identifying processes as a form of abstraction for the behaviour of computer systems rather than as a convenient term commonly used to denote a running programme. The proposal offered below is a fairly direct descendant of this earlier suggestion.

In the wider world, processes are widely used and understood, but definitions in any but informal and practical terms are thin in the ground, at least so far as textbooks on Operating Systems are concerned. After a survey of many texts, Robert wrote[2] : "Most books have an early section or chapter on processes and their properties. Some books don't discuss process states explicitly but use the word and the concept as though readers understand what is being talked about; others only mention these things in case study chapters.".

Most recent textbook definitions approximate the notion of "a programme in execution", a phrase which seems to have caught on. A specific example is the definition given by Lane and Mooney[3] : "a program in execution under control of an operating system". This is interesting, as it makes explicit the exclusion of the operating system itself, and indeed anything but conventional code, from the notion of a process. Few other definitions include anything about the operating system, but in practice it is usually omitted when processes are discussed.

I don't remember any suggestion that the definition offered was backed by any convincing argument. (We suggested that it was folklore[2].) There is usually some explanation of the difference between a process and a programme, but rarely, if ever, any serious attempt to identify just what it is that the notion is intended to capture. The result is a definition which serves the purpose for which it was invented – to describe the workings of the operating system – but which throws little light on the broader picture of what is going on in the computer. Here are some examples :

- The operating system isn't included (as I already remarked). More precisely, how much of the operating system is included depends on the system; in a system with a smallish kernel, such as Unix, more system operations are regarded as carried out by processes.

- An execution of (say) a Java programme is regarded as a simple process running the Java virtual machine; it's true, but oversimplified. (It is true that an execution of a shell script is regarded as a single process running a shell programme, but this is not a good parallel, as the things done by a script generally happen by setting up additional processes.) (I don't know whether anyone has carefully discussed how just-in-time compilers fit into the process model.)

- A running process is suspended, then resumed. What puts it back again ? In fact, it's usually the dispatcher, which might be started by an interrupt, but none of that is included in the usual model – so a good part of the workings of the system is ignored.

None of that matters a lot if you simply want a qualitative description of the system which is sufficient to form a reasonable mental model of what's happening, but it isn't accurate or complete.

I think that if we want to investigate the activity in the system in anything approaching a rigorous way, we must at the least have enough structure in our descriptions to allow that activity to be determined. In practice, it might be that the detailed treatment is never required – you can say a lot about Turing machines without running specific programmes – but without a model which is complete in principle you are unlikely to get any reliable answers at all.

The Turing machine is a good example. It captures the essence of a certain sort of activity, and with that one can develop arguments which lead to general statements of what can and cannot be done with such systems and variants thereof. What I would like is a model which similarly captures the essence of concurrent activities in sufficient detail to model events in operating systems, and sufficiently precisely to support argument leading to reliable conclusions.

So far, I haven't found anything in the literature significantly better than the conventional process model. There are two noteworthy developments of the idea which I'll mention later, but these are not obviously worked out in any detail even in the books in which they are presented. There might be something in the more theoretical work, and I'll look for it, but I want to write down my proposal first if only to clarify it to myself. To do so, I should first address the deficiency I noted of "any serious attempt to identify just what it is that the notion is intended to capture".

WHAT IS A PROCESS ?

Robert and I are still supposed to be writing the book, and I am still supposed to be tidying it up for publication. This has turned into a massive revision as I have tried to follow through the intended principle of starting from a definition of operating systems as means of supplying computing services to people who want them. In my struggles through the undergrowth of what is still too tortuous a development, I found it useful to identify the "elements of operating systems" as hardware, programmes, files, streams, and conventions. After that, I wrote down these passages :

1 : "Processes become necessary when you wish to give each activity a name so that you can analyse the progress of each, investigate their interactions, and so on. It is then important to have a way of identifying a continuing activity which might involve several programmes, or which might share the same programme with other similar activities, which might have periods of inaction between its periods of activity, and which might even move from one computer to another as it proceeds, but nevertheless retains its identity throughout.".

2 : "... And that brings us back to another notion we introduced earlier, that of *activity* – both in the selection and development of the design of the system, which we shall not discuss, and in the system itself, which we shall. We noticed then that activity was different in nature from the elements in that it (obviously) requires something to be active, so depends on at least a selection of the elements. Nothing happens with elements on their own; conventions must apply to something else, hardware and programmes are symbiotic, files and streams cannot exist without hardware. We have still not discussed this drawing together of the elements.

To do so, we reintroduce the notion of the *process*. It has not appeared very much in this section, which supports our view of the elements as in some way at a lower level than processes, but we used it in previous sections to describe activity in the system. In returning to processes, though, we exalt them to a much higher status, for our discussion suggests that the idea of a process is not merely a convenient way to describe what goes on; instead, it is central to the functioning of the system, in that nothing happens except through a process.

Henceforth, then, we regard the operating system as a means for ensuring the orderly and efficient execution of processes. ..."

(... and about then I stopped to write this note.)

I think that the critical notions in those stirring passages are, first, in the spirit of my original suggestion[1], the identification of a process with the activity of the computer system, and, second, the assertion that "nothing happens except through a process". The object of an operating system is to manage that activity in an orderly and productive way, so processes seen in that light are clearly important; and the second

notion implies that if we can adequately define and describe processes, then we have included everything of significance in the performance of the system.

It is reasonable to suppose that if we want to find some rigorous analysis of the activity, we mustn't leave bits out; we might think of trying to organise traffic through a large and complex road junction while ignoring some of the connected roads. This is the defect illustrated in each of my three objections; we can therefore not blithely ignore the activity of the operating system itself; we cannot regard all Java executions as identical; we cannot leave out the activity associated with interrupts or other exceptions.

> *A possible response to that suggestion might include reference to other complicated programmes; it is not only language interpreters that can behave very differently depending on their data. Am I requiring an analysis in which every single programme in the system must be specified in detail, together with every possible input sequence ? Yes, I think I am. I chose Java as an example in which the effect of the input on the performance was particularly clear, but I accept that the principle is general. It isn't as bad as it sounds; a Turing machine won't do anything without a specific programme, but that hasn't made the notion any less fruitful.*

Consider the Java example further. What is "really" happening when a Java programme runs ? In fact, two things happen. (I shall ignore complications from just-in-time compilers; I am cautiously confident that they will fit into the model which I eventually suggest, and I hint at a possible treatment in a note below somewhere.) First the Java compiler converts the source programme into intermediate code (unhelpfully called "byte-code"); this is a conventional compiling operation. Then the Java virtual machine executes the intermediate code. In both cases, the hardware directly executes part of the Java software system, but in the first operation the behaviour of the running programme is primarily determined by Mr Java's compiler code, while in the second operation the behaviour is primarily determined by the source programme compiled. In the conventional view, though, the processor is always running Mr Java's code. This will be true even if you are a little eccentric, and run a dual-processor system with one processor always running the Java virtual machine.

Suppose now that someone builds you a hardware Java machine which executes byte-code directly, and provides it as a coprocessor which you can attach to your machine instead of your second processor. This will certainly change the system in several ways; memory for the virtual machine is no longer required, fewer processor instructions will be executed, and so on. At another level, though, nothing has changed, as the Java byte-code is still being executed by the second processor, and in the execution there will be the same sequence of demands for memory, for file access, for terminal use. Now, though, the conventional expectation is that the behaviour of the coprocessor part of the system will be determined by the compiled form of the original source programme. Why should the description be so significantly different ?

I think it shouldn't, and I want my description of processes to reflect this reality. In fact, in both cases (at least) two things are happening concurrently – a Java processor is operating, and it is executing a byte-code programme. Each of these levels might generate demands for system resources which depend on the implementation of the level, but both are significant and to describe the system completely both should be taken into account. How could this be done ?

A reasonable approach might be to begin with the nature of processes, and to include in the description enough detail to account for the behaviour. Thinking again about our association of processes with the "elements" of operating systems, it is seen that conventional description of a process includes information about the processor state (which confounds the hardware and the programme) and the input-output facilities (confounding streams and files). The information carried in the process control block, generally accepted as the abstract structure representing a process, varies somewhat depending on who defined it, but it commonly contains, or is linked to, some fairly detailed information about files used by the programme. Streams are rarely perceptible as such, but stream information is kept with the file information. In contrast, there is usually very little, if any, explicit information about programme and processor; it is usually assumed that the processor need not be identified, and that all we need to know about the programme is what is implied by the processor registers.

I think that an improved process structure can be constructed by making these components explicit; each process should include an explicit declaration of its programme and its processor. (I am less sure about the programme than I am about the processor, but there is certainly an argument for some properties of the programme to be included, so I err on the safe side.) I also change the definition of the processor – or it might be more correct to say that I introduce a definition of the processor rather than simply assuming it to be given. This includes, but broadens, the conventional notion of a processor.

As an illustration, the processor which executes the Java source programme in the example is a Java compiler; if this produces byte-code, the processor which executes the byte-code is a Java machine in both cases. If we want to dig deeper (which, for a complete description, we do), we can worry about the nature of the processor. Here our two cases differ, with the Java machine implemented in virtual machine software in one case, and in hardware in the other case. It is quite likely that the processor which executes the Java virtual machine (and the compiler) is a conventional hardware processor. The byte-code will (or, at least, should) run on any Java virtual machine, so its processor is unambiguous (which is one good reason for having the virtual machines); but different virtual machines are required for different hardware processors.

What about the process which was suspended and then resumed ? When it was running, its processor was the machine hardware – or anything else, in principle, but the hardware will do. Now it is suspended, the next thing that happens to it is determined by the dispatcher – or something, and again it doesn't much matter what. It is common to describe these different modes by distinguishing between different parts of the process's state. These are sometimes called the *internal* state, to do with the "real" processor and the programme, and identifying the current position of the computation, and the *external* state, which is to do with the process's relationships with other parts of the system, and traditionally includes descriptions such as ready, running, blocked, suspended, and so on. We see that the "real" processor operates on the process's internal state, while other system components (usually, though not necessarily, software) operate on the external state.

Is it necessary to preserve this distinction between different parts of the system which operate on processes ? Now that we have broadened the definition of a processor, could we not reasonably say that our specimen process's processor is now the dispatcher ? Is that significantly different from saying that the processor for a byte-code programme is the Java virtual machine ? A consistent definition of a programme's current processor is "that part of the system which is next going to change the programme's state".

A processor, then, can be hardware or software; in any case, a processor, being active, is also a process. In most cases, it is a different process. A few processes are autonomous, in that they are their own processors : hardware processors are obvious examples. Their programmes are implicit in their internal structure. This class includes not only what used to be called the arithmetic-logical unit (or units), but also the interrupt system and the system clock, which must operate independently of the primary computing elements. Depending on how far you want to stretch the description, computer operators might be included too. Observe that the process's programme determines its next internal state, but the processor might or might not take note of that – so a hardware computing processor will move a software process to the next state determined by the process's programme, while the dispatcher – hardware or software – will change the state of the process from suspended to ready, without regard for the process's programme. It is also likely to change the details of some data structures, such as system queues or tables, just as a "real" processor changes the details of a programme's internal variables.

It was my original guess that a processor could be associated with only one process at once, but I couldn't convince myself that it was an essential property. I thought of superscalar architectures, etc., and that perhaps I can do more than one thing at once (though perhaps to do so I use different processors ?). I would like to restrict each processor to a single process, because then I can say that if process A is happening, then process B, which shares a parent, must be stopped somewhere. In fact, it is probably essential that processors be defined without the restriction; the dispatcher must be able to manage many processes at once, and the same is true of any processor commonly implemented with a queue or other set of processes. It is possible that experience will show that it is useful to define a special class of single-process processors; a more orderly alternative might be to include the maximum number of processes (perhaps infinite) in the definition of a processor.

In summary : everything that happens in the computer system does so as a part of some process. Every process has a current processor, and to change the state of that process its processor must operate, which in turn requires that the processor's processor must operate. The obvious chain stops when it reaches an autonomous process. Only the autonomous processes do any real work, any chain which does not reach an autonomous process is dead, for its state will never change.

SLIGHTLY FORMAL.

I don't think I'm ready for really formal axioms and such, but here are a few collected notions which I think any such axioms should embody. I have introduced a few names to avoid the sort of vocabulary I was using above.

1	All activity in a computer system is effected by *processes*.	
2	Every process has a *state*, which is a collection of *attributes*.	Includes everything changeable.
3	A subset of the attributes might be distinguished as *primary attributes*.	Internal variables, etc. – what the process is for.
4	At any time, a process has exactly one *effector*, which can change its state.	The process's current processor.
5	A process's state can *only* be changed by its current effector.	The effector is changeable, therefore part of the state.
6	Every process has a *programme*, which is an algorithm whereby a specific effector can alter the values of the process's primary attributes.	Software, ROM, processor wiring ...
7	The effector associated with the programme is the process's *executer*.	I avoided "interpreter", which is better, but has other meanings.
8	Every effector is a process.	Because it is active.
9	A process which is its own effector is an *autonomous process*.	Hardware, people.

That leaves me with some unresolved questions.

- I haven't said anything about transitions of processes from one executer to another. That's because I'm not sure what I want to say, or how to say it.

- I haven't said anything about interactions between processes. In practice, this is obviously important, but I have assumed that it can be built on top of the process foundation. That's what happened in the development of operating systems; when people found that they needed interactions, they provided machinery for that purpose within the system structure they were using. I am not convinced that this appeal to history counts as a sound argument, though.

- I haven't said anything about the hardware structure. That's because I don't know what, if anything, ought to be said. I think there is something, because what happens is constrained by what the hardware can do. Two hardware effectors can communicate directly only if there is a link between them; this is likely to affect what can and can't be done – consider the requirement for routing in a computer network.

- I haven't said anything about data. I'm not sure whether or not I must. Any data involved are implied by the programmes; is it enough to leave it at that ?

- I have not addressed one question which might or might not raise complications. My examples – the Java virtual machine and the Unix shell interpreter – are simple in the sense that a programme running on either is unambiguously under the control of its software executer, with only the executer running on the "real" processor. In many other cases described as virtual machines, the "virtuality" is provided by something amounting to a subroutine library; in these circumstances, some of the programme is executed by the "real" processor, and some by the subroutine library.

Other such hybrid cases can be devised. (Perhaps the just-in-time compilers I mentioned are included ?) I am not sure what to do about them, though I think (or hope) that they can be accommodated in the scheme suggested. I think I would regard the subroutine call as moving the process to a new effector (from <old-processor> to <old-processor-plus-subroutine>), as the state of the original process is now determined by the subroutine. More careful analysis is required. (A curious example from elsewhere[4] (p215) : "Conceptually, the [device] manager and the device are one process, executing part of the time as a software process (the manager) and part of the time as a hardware process (the device)". I don't think I agree, but it's interesting to find the idea from another source.)

- I am sure that I haven't listed all the unresolved questions, because I don't know what they are.

WHY IS THAT GOOD ?

- ... because it accounts for *all* the activity in a computer system. At last I can (assuming that I accept the scheme set out above) prove a principle of which I am quite fond : "A computer can do only two things : it can run a programme, or it can break down".

  A helpful scrutineer of a draft of this note[5] has pointed out that it can also be switched off. That is undeniably true, but it is something done to the computer, not something the computer does. Insofar as it reacts to the switching off – for example, by carrying out some built-in power-fail sequence – it is running a programme, which happens to be built into the hardware. I take this opportunity of acknowledging many other helpful comments from Robert, which I have incorporated in the text hoping that people will think they're my ideas.

- ... because it gives a way of finding just what can and can't run simultaneously. Assuming that processors can only run one process each, two processes which share a processor in their hierarchy of effectors cannot run at once. That's why the system clock and the interrupt system must be separate processors.

- ... because it doesn't oversimplify the structure of the activity in the computer. It will, in principle, cope with an arbitrary variety of programmes and processors in a uniform framework.

- ... because it is extensible. For example, you can include the attached devices if you want; they are "simply" additional processors.

- ... because I like it.

SOME EARLIER SUGGESTIONS.

I haven't found anything quite like this proposal in any other literature. Perhaps that's because I'm looking at the wrong literature, but I've been alert to operating system topics since around 1985, and have looked at a fairly wide range of current literature, including *Operating Systems Review* between 1985 or so and 1998. The bits I've missed are the overtly theoretical computing topics, so there might be something there. There follows a selection of relevant comments from books which are within easy reach; I emphasise that I have made no attempt at a complete historical review.

The notion of a process seems to have been fairly slow to develop. Reading articles on both programming languages and systems in a compendium from 1967[6], it seems that until around 1970 the emphasis was on constructing the code, and therefore weighted towards programming languages, compilers, and subroutine libraries. That's not unreasonable, as both languages and systems are parts of the basic job of getting your programme running, and the language is perhaps the more obvious part of this task. Once you have a programme, then you can worry about running it.

I am happy to note that I wrote the preceding paragraph before coming across this comment from a book[7] published in 1973 : "In the years since 1969, the study of computer systems has assumed a role nearly equal in importance to 'theory of computation' and 'programming' in computer science curricula. In contrast, the subject of computer operating systems was regarded as recently as 1965 as being inferior in importance to these two traditional areas of study.".

My memory of that time is that entities we would now call processes were recognised, but simply regarded as programmes which were running; multiprogramming systems were well known. In fact, they had been for a long time, as is demonstrated by this excerpt from a 1961 paper on Atlas[8] : "While one program is halted, awaiting completion of a magnetic tape transaction for instance, the co-ordinator routine switches control to the next program in the object program list which is free to proceed. In order to maintain full protection, it is necessary to preserve and recover the contents of working registers common to all programs ...". The notion of a process is clearly there, but the comment is presented as a matter of engineering necessity, not as a recognition of a significant abstraction. The emphasis is simply on running programmes, and coordinating their activity with pieces of the supervisor (that is, the operating system).

In a survey by Rosen from 1964[9], the original emphasis on programming languages, rather than on operating systems, is underlined in a comment introducing a second feature of my proposal : "Perhaps the most important contribution of [the Univac] group was the emphasis they placed on the programming system rather than on the programming language. In their terms, the machine that the programmer uses is not the hardware machine, but rather an extended machine consisting of the hardware enhanced by a programming system that performs all kinds of service and library maintenance functions in addition to the translation and running of programs.". Here the idea that a processor need not be made of hardware is apparent. (I am fairly sure that someone must have said corresponding things about the Atlas system, in which many functions were implemented as "extracode", looking like machine instructions to the programmer, but I haven't found one. I haven't looked at all hard.)

In a book from 1968 by Wegner[10] the description is still in terms of programmes, but the significant status of programmes being executed is clear, and something like a process emerges as the "instantaneous description of a program" :

> The instantaneous description of a program may be thought of as consisting of three parts :
> 1. A program part, representing the program to be executed;
> 2. A data word ...;
> 3. A stateword, representing the information in the computer processing unit ...

Wegner also explicitly erodes the one-to-one association between programmes and processes : "... it has been found convenient to allow program segments to be executed by several different higher-level programs simultaneously." – and he follows the earlier lead in his view of the processor : "Each user of a multiprogrammed computer system has at his disposal a virtual computer ..."

An explicit definition of a process is presented by Brinch Hansen[11] in 1973 (p55) : "A process is a sequence of operations carried out one at a time. The precise definition of operation depends on the level of detail at which the process is described.". Here he seems to make allowance for some sort of nesting of processes; in fact, he is merely permitting descriptions at different levels, but there is more to this than meets the eye. He introduces the notion of a virtual machine very early in his account (p3) : "An operating system makes a virtual machine available to each user ..." – but does not restrict the notion to entities which are recognisable in the final system as hardware or programmes. Instead, the virtual machine notion is extended into the design process, and different levels of abstraction in the design are regarded as virtual machines; he gives an example (p 50) of the Banker's algorithm represented as a hierarchy of five virtual machines and the physical hardware, but the implementation of this is a single programme rather than a nested set. This contains some elements of my treatment, but used in a rather different way.

On the other hand, Brinch Hansen's collected definitions tell a rather different story. In his vocabulary section (pp 335, 336) he offers :

Virtual Machine : A *computer* simulated partly by *program*.
Computer : A physical system capable of carrying out *computations* by interpreting *programs*. A computer consists of ... and one or more **Processors**, physical components which can carry out *processes* defined by stored programs.
Program : A description of a *computation* in a formal language ...
Computation : A finite set of *operations* ...
Process : A *computation* in which the *operations* are carried out strictly one at a time.
Operation : A rule for deriving ... **Output** from ... **Input**.

(Type styles are approximately as in the original; I have changed the order of presentation, and omitted things – including a definition of **Operating System** – if I didn't want them. Many of the ellipses include references to data, which I have omitted because they play no part in my discussion.) I have presented these definitions in some detail, because they cover or imply a great deal of my proposal, though Brinch Hansen does not interpret them as I have. The only missing element (I think) is a link between process and processor, and that is missing only because he has not said that a processor offers a specific set of operations; this extension would imply that a computation must be run on a processor which matched the set of operations required.

Coffman and Denning[7], also from 1973, comment on the invention of processes (p8) : "The term process as the abstraction of the activity of a processor appears to have emerged in several projects during the early 1960s ..., so that the concept of 'a program in execution' could be meaningful at any instant of time, irrespective of whether a processor was executing instructions from the program at that instant.". They remark on the many definitions which have been offered, but then contrive to avoid making any specific definition of their own by sticking to a level of discussion at which they don't have to worry about implementation details. They are liberal about processors – "any device which performs transformations on information or carries out the steps of a process. ... central processing unit ... input/output processor" (p3) – but I haven't found any further discussion of this definition.

Madnick and Donovan[12], writing in 1974, introduce processes right from the start (p5), where they define a process as "a computation that may be done concurrently with other computations". ("Computation" seems not to be defined.) They are uncompromising about the nature of processors (p4) : "A processor is a hardware device". This is perhaps rather surprising, because their book is dominated by IBM systems (proof : count references to manufacturers' names in the index), and includes discussion of IBM virtual machine methods. They do identify "extended machines" in much the same way as Rosen did, and describe a "hierarchical machine" (p 16) as a model of an operating system. In fact, their hierarchical machine is just the onion model. They – perhaps reasonably enough – make no special distinction between the extended machines implemented by supervisor calls and IBM's virtual machines – so they describe (p549) "... a Virtual Machine Monitor (VMM) system. A VMM is a special form of operating system that multiplexes only the physical resources among the users – no other functional enhancements are provided. In particular, the extended machine ... provided by a VMM is identical to the bare machine on which the VMM runs !". On the other hand, they do recognise that different sorts of processor are possible, and specifically mention I/O processors.

The closest approximation to the views offered in this note are perhaps those put forward by Lister[13] in 1979. He begins by defining the operating system itself from a rather unusual viewpoint (p12); an operating system is a set of activities (such as scheduler, IO handler), each of which is performed by the execution of one or more programmes. A process is defined as a formalisation of this idea as "a sequence of actions, performed by executing ... a program ... whose net result is the provision of some system function". This definition is then extended to include "user functions", so includes all programmes. A process can involve many programmes, and programmes can be involved in many processes. A process proceeds because it has a processor; this might not be hardware. Examples of processors are the CPU, a Basic interpreter, and a VM system à la IBM (p109), though in the cases of the interpreter and virtual machine my hierarchy of processes is not recognised; instead, the processor is taken to be the combination of hardware and interpreter. It is recognised too that programmes can be hardware; channels and other peripheral devices are examples. Process hierarchies are mentioned (p106), though these are seen as relationships between fairly ordinary processes rather than the encapsulation of one process by another envisaged in my scheme. It is not so clear that this broad vision is fully worked out in the main body of the book, which otherwise follows fairly conventional lines.

Janson[14], from 1985, offers a rather different view which seems to be strongly influenced by the IBM series of virtual machine systems. This leads him to identify different sorts of processor, and correspondingly different sorts of process, and consequently to identify processors at one level with processes at another. For example, he writes (p24) "A virtual processor is an abstract processor that implements a unit of parallelism, a locus of control that evolves through a user instruction stream. ... virtual processors are often called ... processes in other contexts" – though in fact his use of the term "process" there does not seem to fit well with its more conventional interpretation. This is more closely identified in a later statement (p72) : "The concept of a user process is supported by a mechanism called the process management mechanism. ... multiplexes virtual processors among user processes ...". The similarity between the levels is explicitly pointed out : "Everything applicable to virtual processors with respect to physical processors can be stated about user processes with respect to virtual processors." Nevertheless, the different levels of process are different sorts of thing, for the "virtual processors" have to be limited in number (p71 – though that seems to be because of the table size ?). The possibility of further levels is suggested (p65) : "... one can talk about a processor state at various levels. For a physical processor, the state between two instructions is completely defined by the ... hardware registers. ... in the middle of an instruction, it is necessary to include all internal registers ... Similarly, ... to describe the state of a virtual processor ... current state of the physical processor ... state of the microcode processor.".

It is clear that the word "process" conveys rather different – thought not *very* different – things to different people. The notion shows some sign of developing through the years, generally beginning with the idea that a process is something that happens when you run a programme, and growing as extensions to the basic idea (for example, the hierarchy of processes, and virtual machines) gain general acceptance, with each author adding his own pet ideas. This is all very laudable, but I have two concerns, which are my own pet ideas. These are, first, that there is little variation on the original theme, and, second, that in all cases the definitions appear to be ad hoc, with processes defined rather as an aid to describing what's happening than as a means of analysing the behaviour of the system.

There are one or two counterexamples to my first concern, with those I've noticed following a minor, though significant, variation of the basic theme. In this view, processes are seen as entities in their own right, and as the agencies by which programmes are executed. Finkel[15] writes (p2) : "A process is a fundamental entity that requires resources in order to accomplish its task, which is to run a program to completion.". (He also identifies himself as a man of clear perception : "The process notion is both central to operating systems and notoriously hard to define".) Here, the process is responsible for the programme's execution, an idea which is clearly expressed in a later comment (p241) : "Typical batch multiprogramming systems initialize themselves to have a fixed number of processes. Each process runs one job ... to completion, and then looks for another job ....". (I add that this organisation is not restricted to batch systems; the – unfortunately deceased – Digital Equipment Corporation's TOPS-10 interactive system worked in this way, with each process associated with a terminal. You could still run additional processes, but to do so you had to define a "virtual terminal".)

A rather similar, though muted, position is taken by Holt[4], who agrees with Lister in identifying processes in the first instance with operating systems functions, but then extending the notion (p8) : "Each user's program can be managed and executed by a process" – but then tries to have it both ways by also saying (p11) : "A process ... can be thought of as the execution of a program by a CPU". It is interesting that the second quotation continues : "However, the CPU may actually be a virtual CPU that is implemented by multiplexing one or more physical CPUs among many processes". Holt also accepts the notion that not all processes are the same (p9) : "Often the processes that manage devices are quite different from those that execute user jobs; on the CDC6000 systems the device managers do not even run on the CPU – they use special 'peripheral processors'.".

Despite their non-standard approach, though, neither Finkel nor Holt develops the idea to any extent. Both follow precedent by continuing to use processes as a convenient means of description of the operation of operating systems rather than as a basis for their analysis.

As for my second concern, I have found no previous definitions of processes which have any consequences other than convenience. It is as though atoms were defined as "little bits of matter", without further qualification; with such a definition, one can describe events in terms of atoms when it happens to be convenient, but one is not obliged to seek atoms in all matter, and it is of no concern if you can't account for some phenomena in terms of atoms. Adding the principle that "all matter is composed of

atoms" makes a big difference; now any phenomena which appear not to be compatible with atomic theory become highly significant, and require attention, and the theory becomes fruitful.

I want to do the same trick with processes, and that's what I've tried to present in this note. If I do, I am forced to seek processors and programmes for all the activity that goes on in a system, and I can find them. The result is the basis for a theory of all activity in an operating system.

A DUAL APPROACH ?

Several of the authors whose work I have mentioned allow for something like the hierarchical structure of computing by proposing what amounts to a hierarchy of virtual machines. A possible view (which I shall attempt to refute forthwith) is that my proposal is essentially the same, but seen from the other end. So far as describing the system is concerned, there is something in this view. In both cases, the "real" process is seen as running in an environment which itself is expressed in terms of a more elementary environment, and so on.

I think that the significant difference is that the other proposals (so far as I understand them) all regard *the* process running on *the* top-level system-defined virtual machine as the real behaviour, while I insist that all the processes at every level must be considered as operating together. (It is just this characteristic of the other proposals which leads to the Java virtual machine being considered the real process in our original example, with the contribution of the Java programme ignored.) Perhaps this difference comes about because I emphasise throughout the *activity* taking place in the system, while many other authors seem to concentrate on describing the system itself. From that point of view, the description as a set of nested processors is quite sensible, and serves their purpose – but as a processor is a thing which sits there waiting to be used, no activity is implied.

I begin with the activity, and define processes so that they include all the activity. This leads me to a description of the same structure in which activity at all levels is explicitly taken into account. I think it's better.

A WIDER SIGNIFICANCE ?

Casti[16] identifies a class of complex systems characterised by their behaving in surprising ways, and their irreducibility in the sense that their behaviour is not merely the sum of the behaviours of their components. Inspection of instances of such systems shows that they have certain features in common. Some of these are :

- They consist of a "medium" number of interacting agents – too many to be handled by explicit mathematical approaches, but too few for statistical methods to be useful;
- The agents are "intelligent" – at least to the extent that their behaviour depends on circumstances, typically according to some set of rules;
- The agents adapt their behaviour depending on local knowledge of their environments, not knowledge of the complete system.

Operating systems show all these characteristics. In an earlier report[17], I wrote of the conventional operating systems introductory course : "It doesn't work. The course material in fact doesn't account at all well for the actual behaviour of the system. Alan's ... experience with measurements on real running systems taught him that system behaviour is much more complicated, and very hard to analyse. Everything interacts inscrutably. The course material isn't wrong, but isn't sufficient to predict the behaviour of practical systems. It describes how all (well, some of) the parts work – but it doesn't add up to how the system works (and wouldn't even if the other parts were added).". (Some of the "experience with measurements on real running systems" is recorded, very scrappily and incompletely, in earlier notes[18, 19].)

You can quibble a bit about the extent to which operating systems conform to Casti's set of conditions. Perhaps it would be more accurate to suggest that they are only just complex systems; certainly comparatively minor changes can remove much or all of the unpredictability. In real-time operating systems, guarantees of performance can be made provided that certain constraints are imposed –

notably, restricting the programmes to a known set with known properties, usually resulting in a fully predefined sequence of requests for system resources.

Casti further asserts that until some new mathematical language which can deal with these systems is developed, they will remain obscure. Perhaps processes are a part of this language for operating systems.

REFERENCES.

1 : G.A. Creak : *Introducing processes in a course on operating systems*, Working Note AC113 (1997 August 25).

2 : G.A. Creak, R. Sheehan : "A Top-Down Operating Systems Course", *Operating Systems Review* **34#3**, 69-80 (July 2000).

3 : M.G. Lane, J.D. Mooney : *A practical approach to operating systems* (Boyd & Fraser, 1988).

4 : R.C. Holt : *Concurrent Euclid, the Unix system, and Tunis* (Addison-Wesley, 1983).

5 : R. Sheehan, in conversation (2001 May 14).

6 : S. Rosen (Ed) : *Programming systems and languages* (McGraw-Hill, 1967).

7 : E.G. Coffman, P.J. Denning : *Operating systems theory* (Prentice-Hall, 1973).

8 : T. Kilburn, R.B. Payne, D.J. Howarth : "The Atlas supervisor", *Proc. Eastern Joint Computer Conference* **20**, 279-294 (1961), reprinted in reference 6, pp 661-682.

9 : S. Rosen : "Programming systems and languages – a historical survey", *Proc. Eastern Joint Computer Conference* **25**, 1-15 (1964), reprinted in reference 6, pp 3-22.

10 : P. Wegner : *Programming languages, information structures, and machine organization* (McGraw-Hill, 1968).

11 : P. Brinch Hansen : *Operating system principles* (Prentice-Hall, 1973).

12 : S.E. Madnick, J.J. Donovan : *Operating systems* (McGraw-Hill 1974).

13 : A.M. Lister : *Fundamentals of operating systems* (Macmillan, 2nd edition, 1979).

14 : P.A. Janson : *Operating systems structures and mechanisms* (Academic Press, 1985).

15 : R.A. Finkel : *An operating systems vade mecum* (Prentice Hall, 2nd edition, 1988).

16 : J.L. Casti : *Would-be worlds* (Wiley, 1997), Preface, and pp 213-215; and observations thereon from Casti during a seminar in Auckland in 2001 February.

17 : G.A. Creak, R. Sheehan : *A new structure for an operating systems course* (Auckland University Computer Science Department, Technical Report #162, December 1999).

18 : G.A. Creak : *Improving the DEC-10's performance*, Working Note AC32 (1982 July 7).

19 : G.A. Creak : *Monitoring the DEC-10's performance*, Working Note AC40 (1984 August 29).