Proceedings of Sixth ACM Symposium on Operating Systems Principles (November 1977) 173-180.

BEYOND CONCURRENT PASCAL

Klaus-Peter Löhr
Technische Universität Berlin

Abstract -- We take the view that operating systems should not be written in assembly language. Alternatives are machine oriented high-level languages and "safe" languages in the style of Concurrent Pascal and MODULA. A serious drawback of the Concurrent Pascal approach is the fact that those very language features that pertain to operating systems must be implemented separately, using some other language. A technique is presented which solves this problem.This technique is based on user-defined trap handling. It is exhibited by demonstrating how virtual memory systems can be constructed using Concurrent Pascal and how process management can be moved from the kernel to the Concurrent Pascal program. We demonstrate that a fundamental solution of the difficulties with Concurrent Pascal, MODULA, and similar languages cannot be found in going back to classical implementation languages, but in designing languages that are not rich with special features, but powerful with respect to extension and shrinkage.

Keywords: System implementation languages, system kernels, traps, language reliability.

1. Introduction: High-Level Languages for Operating System Construction

1.1 Most operating system designers agree, at least in principle, that operating systems should not be implemented using assembly language. However, there is still a controversy between two different attitudes: should we use a medium-level systems implementation language with some unsafe features (like explicit use of memory addresses and machine instructions), or can we do better, if at all, with a really safe high-level language?

Like assembly language, implementation languages are often advocated for their run-time and memory efficiency. But this is not the real issue. Careful design and an optimizing compiler can make a language both safe and efficient. Moreover, the use of a safe language simplifies system design, although it may prohibit some tricky fine-grain programming. An unsafe language, which has no strong type checking and allows treating pointers as integers, necessitates firewalls to guarantee the integrity of the system against programming errors of system programmers and users (not to mention penetration efforts). A safe language makes these firewalls unnecessary, reduces the need for run-time checks, and thus considerably improves efficiency.

This observation clarifies one aspect of what is meant by "the choice of the implementation language has a considerable impact on system design". It is the very issue of efficiency plus reliability that calls for the use of safe languages.

1.2 The real benefits of implementation languages in operating systems programming come from their ability to explicitly deal with the hardware, by accessing processor and device registers and by executing special instructions.

Classical implementation languages are of different degrees of insecurity. Some representatives are (in order of ascending security): BLISS [Wulf et al. 71] for the DEC 10 and DEC PDP-11, used in the implementation of the HYDRA system [Wulf et al. 74]; C [Ritchie 75] for the PDP-11, used in the implementation of the UNIX time-sharing system [Ritchie/Thompson 74]; SUE [Clark/Horning 71] for the IBM 360, used in a (not completed) operating system implementation in Toronto [Atwood et al. 72]; ESPOL [Burroughs 70a] for the Burroughs B 6700, used for the Master Control Program for that machine [Burroughs 70b]; EUCLID [Lampson et al. 77]. The higher a language ranks in this ordering, the more it deserves to be called a "machine-oriented high-level language" (MOHLL, [IFIP 74]). This term stands for a language which is safe "in principle" but is augmented with a few unsafe, machine-dependent features. A very clean solution has been adopted in EUCLID: the unsafe, machine-dependent language features are confined to so-called "machine-dependent modules". - The merits of MOHLL's are obvious, and an efficient implementation of a MOHLL is certainly superior to an efficient implementation of a low- or medium-level implementation language.

Note, however, that we still need an assembly language interface to the hardware if the language has no special features suited to solve typical low-level operating system problems. E.g. if there is no feature like an interrupt procedure, an assembly language interface must take care of the first level interrupt handling. Likewise, if the

language does not provide coroutines, special assembly code is required for stack switching. Thus, the loss in security does not pay in coherent high-level coding of the system, because the mentioned features must nevertheless be added to most implementation languages through run-time support.

1.3 With the increasing demand for special purpose system software on small computers, the need for simple and reliable production of this software becomes obvious. Since safe languages exist that allow well-structured implementation of programs, it is possible to design such languages for the special purpose of writing a whole operating system as one program. An operating system is admittedly a fairly large program, but promising techniques are emerging for the treatment of large programs. Their implementation requires languages with sound structuring aids and the possibility of separate compilation in an advanced fashion, most notably with syntactic checks at linking time.

Turning our attention from implementation to design, we observe that operating system design problems are also not much different from those encountered in design of other software. Accordingly, the programming language used for operating system construction should at the same time be a design language, i.e. should assist the designer and encourage a proper "programming-in-the-large" by supplying language constructs that reflect generally acknowledged design principles.

The use of such a language facilitates the construction and maintenance of an operating system considerably. The problem of interfacing such a language with the hardware is solved in the following way. For the sake of security, there is no explicit access to memory locations and registers, nor is it possible to insert machine instructions into the high-level code; a small interface, serving as a run-time system for the language, builds a convenient abstract machine on top of the concrete hardware. The run-time system can be conceived as the lowest level or kernel for all operating systems expressible in the language.

The relevance of this approach has been demonstrated by the widely acclaimed success of the programming language Concurrent Pascal (subsequently abbreviated CPASCAL) [Brinch Hansen 75a]. We can write non-trivial operating systems in CPASCAL within a few man-months. The implementation distributed by the authors also demonstrates the possibility of a limited portability of small operating systems. Another recent example of such a language is MODULA [Wirth 77] which is superior to CPASCAL mainly due to the flexible structuring aid of the "module" and the tight control over input/output.

2. Limitations of the Concurrent Pascal Approach

2.1 We do not want to discuss here CPASCAL as a programming language. As in any language there are strengths and flaws. At issue is the CPASCAL approach to operating system construction. The CPASCAL approach is characterized by 4 or 5 essentials:

1. The language is high-level, with the exception of some tiny security leaks (this looks inevitable at the present state of the art).

2. A complete operating system (not including files on backing store) can be written as one compilation unit if desired.

3. The language incorporates features considered fundamental for writing operating systems, namely processes and synchronization.

4. A medium-size run-time system supports these language features as well as input/output. Operation of the run-time system is triggered both by traps/interrupts and by explicit program action through language features or standard procedures. The run-time system serves as an invariant kernel for all operating systems written in the language.

(5. Portability of the operating systems written in the language is achieved by making the compiler generate virtual code which is executed by an interpreter incorporated in the kernel. - This characteristic is not essential for the purpose of this paper.)

2.2 There is no doubt about the usefulness of the language as a design tool, but this usefulness is limited by the fact that processes and synchronization constructs are predefined in the language (essential no. 3). This forces the designer to use monitors for mutual exclusion and synchronization, whether he likes them or not. We take the view that using monitors, with the inherent rule that any shared data object must be realized as a monitor, is not just a matter of taste, but is a serious design decision. E.g. we may be willing to live with the strict exclusion provided by monitors running on an uniprocessor system, whereas in a multiprocessor configuration time-critical tasks might suffer from the strict exclusion principle which imposes unnecessary timing constraints on co-operating processes. Taking the CPASCAL approach, the language designer has already made design decisions for you, and you are forced to build on top of those.

Worse than that, we have to live not only with given design decisions but also with a given implementation of these (essential no. 4). The designer who wants to implement process management and device drivers (in MODULA process management only) by himself, must re-program the system kernel, which is written in another language. This additional language is not necessarily assembly language[a]. Anyway, the goals stated in 1.3 are not achieved because we now have two language processors to work with.

---

[a] At the Technical University of Berlin a CPASCAL kernel has been implemented on an IBM 370/158 under VM 370 using SUE-360 [Clark/Horning 71] and a small assembly language interface as mentioned in 1.2 .

2.3 Adopting the CPASCAL approach to operating system construction requires a choice from somewhere between two extremes:

- A relatively poor language leads to a small kernel, but is hard to program a "real" operating system in. Example: (Sequential) PASCAL with a run-time system that runs on the bare hardware.

- A relatively rich language includes half of the operating system in the kernel. Example: A hypothetical variant of CPASCAL where the compiled programs are running within a virtual memory instead of residing in main memory; in that case the kernel must be drastically enlarged for the necessary virtual memory management.

(CPASCAL with its original kernel lies between these extremes.) Whatever decision we make, we cannot overcome the annoying trade-off between small kernels and attractive systems. And making our choice, we have lost much freedom of design, and freedom of implementation is achieved only at the expense of kernel modification.

3. A Case Study: Virtual Memory Management in Concurrent Pascal

3.1 There is a way out of the previously mentioned dilemma. Let us stick to CPASCAL and try to devise an operating system that supports virtual memory. On the one hand, the security of CPASCAL should not be decreased by adding features for the realization of the virtual memory management; on the other hand, the virtual memory management should not be included in the kernel.

With the PDP-11 memory management hardware in mind, we can imagine an - admittedly crude - paging system in which pages of 4K words are subject to swapping. Suppose the kernel contains a virtual memory manager, VMM. In that case no harm is done, if a CPASCAL program does not fit in main memory. The system is loaded and initialized on a swapping device; the kernel swaps the necessary pages into main memory on a demand basis or using a clever working-set oriented scheme. Note that this would not require any change in the CPASCAL compiler. Now, if we wish to implement the VMM as part of the CPASCAL program that constitutes the whole operating system (except kernel), some interface must be added to the kernel that triggers the operation of VMM upon the occurrence of page fault traps and of interrupts from the swapping device. What is the nature of these interfaces?

3.2 Since there is no inherent reason to implement a VMM using a demand paging scheme as a process, we first think of a monitor implementation. Unfortunately, it is impossible to simulate a monitor call upon occurrence of a trap in CPASCAL. This is mainly due to the fact that the name of an entry procedure of an object cannot be passed to the kernel (which is necessary for trap indirection). It is not possible to remedy this situation by making changes in the kernel.

Next, we try a process implementation for VMM. Since the message queue for this process cannot be implemented as a monitor either (for the reasons just mentioned), it must be implemented as part of the kernel. Reading from the queue can be accomplished via the standard procedure io. This procedure is originally intended for input/output, but can in fact be used as a general purpose "kernel call". We define a "device" called "swapqueue" and provide appropriate "input/output" operations:

```
PROCEDURE io
   (VAR fault: RECORD processno: integer;
                      pageno: integer
               END;
    CONST operation: (input,output,lookup);
    CONST device: (swapqueue)   )          (a)
```

An anonymous operation on swapqueue is simulated by the kernel upon occurrence of a page fault trap. The identity of the process that caused the fault is entered into the queue, and with it the identity of the page at fault. The trapped process is delayed. Next, operation "input" should be issued by VMM for removal and delivery of the process identity as well as the identity of the page that should be swapped in for that process. In fact, VMM is delayed upon an input operation until swapqueue is nonempty. As soon as the page is available, VMM should perform the operation "output" which signals the completion of the page transfer to the process and allows it to continue(b). - Thus, swapqueue serves as a special purpose process communication device which is not programmed in CPASCAL as a monitor but is part of the kernel.

The task of handling the swapping device can be performed by applying the procedure io in the usual manner just for that device (a slightly different view will be presented in 4.2). The insecurity of the io feature (only limited checks are performed at compile time) allows any page transfer between the device and a selected page frame in main memory.

3.3 Our problem is not solved yet. We have just managed to establish VMM within a functional hierarchy (see [Parnas 74], [Parnas 76]) using the trap detour. However, execution of VMM functions does not only require bookkeeping of page frames and controlling page transfer, but also includes maintenance of page descriptors. The latter task cannot be performed in a straightforward way, because the descriptor must be available to the kernel: since any process switch requires changes to be made in the address map(c) the kernel must have the descriptors at hand and must know about their representational details.

(a) CONST is not standard CPASCAL; it has been added for the sake of clarity.

(b) More elaborate operations on swapqueue could be useful if VMM was to operate on the swapping device according to some non-FCFS strategy. This is omitted here for the sake of simplicity.

(c) Note that this is implementation dependent. We can conceive of CPASCAL implementations on machines different from the PDP-11 where changing the address map is not necessary.

This way we get into a situation that has been described by different authors as "interleaving of functional hierarchy and module structure" or simply "sandwiching" ([Parnas 76], [Habermann et al. 76]). The kernel may be considered as consisting of two functional levels, namely maintenance of address spaces and, above this, process management. The latter need not know about the representational details of the former, it only uses its functions (this structure is neatly reflected in the quasi-PASCAL comments in the original CPASCAL kernel). Now we get a third level, VMM, which cannot do without knowledge of the representational details of the first level, although functionally dependent on the second level.

In the CPASCAL environment there is no way for VMM to get at the inner of the kernel - and this is alright. A solution ot the problem comes once more from the io feature. We are able to provide a special io function that reads and manipulates descriptors. This io figures as an operation on the abstract data type "descriptor" and thus allows changing the representation of descriptors without affecting the operation of VMM. A possible way of realizing the io operation is:

```
PROCEDURE io
   (VAR descinfo: RECORD base: integer;
                         resident: boolean;
                         dirty: boolean
                  END;
    CONST operation: RECORD op: (read,write);
                             descno: integer
                     END;
    CONST device: (desctable)      )
```

All descriptors reside in a central descriptor table called "desctable". They are identified by their position in that table. The address map associated with a process is given by a sub-block of 8 entries within the process control block which point to descriptor table entries. - Accessing desctable through io causes no delay for the executing process.

This is a simplified presentation of the facts. In reality VMM needs additional information about the page states when looking for a victim to swap out. Also, efficiency can be improved, if several descriptors can be operated upon instead of just one (e.g. the set of descriptors of all core resident pages).

The above approach has been taken for design and implementation of a small time-sharing system for a PDP-11/40E at the Technical University of Berlin [Gräf et al. 77]. The system is called MUSIC for multi-user system in Concurrent Pascal and is based on SOLO [Brinch Hansen 75b]. It has been in operation since March 1977 and has proved to be a valuable tool for experimenting with operating systems, especially with swapping strategies.

3.4 What have we achieved? Did we change the language or did we not? - At least we did not make any change in the compiler. One may argue that by extending the kernel we silently removed the restriction that any program must fit in main memory. With the new kernel, we find that the language provides for virtual memory in the following sense: a program written in the language is compiled for, and will be running within, a virtual memory; however, the run-time system does not contain a VMM; it only contains an interface that allows the construction of a VMM as part of the program.

The VMM is not unlike a user-defined trap handler. The programmer is not forced to provide it, but if he does not, his program may crash. The following may happen with our virtual memory kernel:

- If the program fits in main memory, a page fault will never occur and nobody will ask for a VMM.

- If the program does not fit, page faults may occur. These will produce entries in the swapqueue. If a process is provided that empties the queue, it is supposed to do the swapping as a VMM. But:

- If no such process is provided, the swapqueue will never be read, and some (or all) processes may get hung up. This would be the analogon to the above mentioned program crash.

Note that placing the swapqueue in the kernel is not essential to the above approach. The real problem with CPASCAL is that devices (of which the processor is one) are not treated in the same way as ordinary objects in the language. MODULA could have provided a "device module" for the processor which turns a page fault trap into a send message operation for an explicitly programmed swapqueue.

3.5 There is one serious flaw in our considerations up to this point. Since VMM is an integral part of the program which is subject to swapping, the necessary functional hierarchy is not enforced. (Compare this with a trap handler that may cause traps of the very type it handles.) A clean solution requires a language amplification which tells the compiler which parts of the system are swappable. Appropriate information is then put into some of the descriptors. These descriptors are hidden from the VMM by the desctable io, and VMM must ask the kernel which page frames are available for swapping.

In the MUSIC system a pragmatic approach has been taken. The programmer of MUSIC has some idea of how memory is distributed over the system parts, so he can see to it that VMM will never swap out parts of itself.

The most notable aspect of the approach just sketched is the fact that it is applicable not only for one special system level, namely just VMM. We have implicitly presented a general principle that solves the implementation problem demonstrated at the end of 2.3 . The answer to the language designer can (need not) be: "provide many nice features for operating system design" and to the implementor of the run-time system: "don't provide ready made implementations of those features, but do provide 'links' like swapqueue and desctable". We will embark on this principle in the next section before radically turning around in section 5.

4. Generalization: Programming Process Management in A Language that Provides Processes

4.1 The applicability of the general principle mentioned above is elucidated by trying it on another language feature, namely processes/synchronization. Taking this very feature has the advantage that we can stick to CPASCAL for presentation (although this is, again, immaterial to the subject). Moreover, this may be more convincing than taking some feature "above" virtual memory, say files, which might be "too familiar" to the reader as a language feature.

So, imagine the process management (PM) is removed from the CPASCAL kernel. It should be replaced by an appropriate mechanism that allows to implement it as part of any operating system written in CPASCAL.

The nature of such a mechanism can be derived from section 3 if we emphasize that processor and memory can both be viewed as resource types. The operating system deals with managing the available instances of resource type "CPU" as well as those of resource type "main memory frame". PM provides virtual processors by multiplexing the real CPU (or several CPUs). VMM provides virtual address spaces by multiplexing main memory frames. Common to both is the phenomenon of preemptions.

Note that if the system fits in main memory, no VMM is required. Thus, if the number of processes does not exceed the number of processors, it suffices for PM to provide a simple-minded synchronization mechanism based on busy waiting. For these cases the kernel could arrange an appropriate allocation of memory and processors to the system components at initialization time.

4.2 We continue to discuss the multi-processor case for the sake of generality. When a process is ready to run, it competes for the resource "processor" and has to queue up in what is commonly known as "ready list". The ready list can be seen as a straight analogon to the swapqueue. It is most conveniently thought of as being served by a separate "processor management process"which is itself not subject to, but part of PM (process management on the CD Cyber series comes close to this view since a separate processor is used).

Note that PM consists of two parts, the processor management which serves the ready list, and the synchronization management (plus timer) which produces arrivals on the ready list. Comparison with VMM shows that demand swapping has no analogon to the latter part of PM; arrivals on the swapqueue are caused by hardware traps.

PM knows about allocation of the processors to the processes, just as VMM knows about allocation of memory frames to (pages of) processes. As VMM becomes active upon a non-empty swapqueue, so PM is activated upon arrival of an entry in the ready list. PM then decides, according to some scheduling strategy, whether a processor should be preempted from some selected process and allocated to the requesting process. Compare this with the activity of a VMM for demand swapping!

The CPASCAL VMM described in section 3 makes use of

```
PROCEDURE io
   (CONST pageframe: RECORD pageno: integer;
                            frameno: integer
                     END;
    CONST operation: (swapin,swapout);
    CONST device: (memory)     )
```

for preemption and allocation of memory frames to pages[a]. Accordingly, PM should provide a procedure for allocation and deallocation of processors to and from processes, e.g.

```
PROCEDURE io
   (CONST proc: RECORD processno: integer;
                       processorno: integer
                END;
    CONST operation: (allocate,deallocate);
    CONST device: (cpus)     )
```

Preempting a processor from a process amounts to stopping the processor and deallocating it from the process; deallocation means copying the processor's state information into the process's control block. Allocation means loading the processor with state information from a process control block and then starting the processor.

Now, it is not necessary for the processor management part of PM to be implemented as a separate "meta-process". It can simply figure as a system part which is executed by the processes themselves. If a process has entered PM, it may recognize that it has to relinquish its processors, either due to a release of that resource (blocking) or to preemption. In both cases, after having performed the necessary bookkeeping operations (i.e. juggling with lists of process control blocks), the process should deallocate the processor from itself and allocate it to another process which is taken from the ready list. Conceptually, this is achieved by

```
io( (currentprocess,currentprocessor),
     deallocate, cpus);
currentprocess:= selectprocess;
io( (currentprocess,currentprocessor),
     allocate, cpus)
```

This is not quite correct, though; the above operations must form a single

```
PROCEDURE io(CONST processno: integer;
             CONST operation: (resume);
             CONST device: (processor)    )
```

which essentially is a coroutine exchange jump and is commonly known as

```
PROCEDURE resume(CONST processno: integer)
```

Our first conclusion is: CPASCAL must provide a coroutine feature which allows explicitly control over the scheduling of processors. Of course, this result is not surprising. However, note that in our context

---

(a) Each page is supposed to have its fixed location on the swapping device for all time.

- the exchange jump is just a special case of a procedure for resource allocation/deallocation offered as an io operation by the kernel,

- the language proper is not changed in any respect.

4.3 PM must be entered upon occurrence of one of the following events:

- start and completion of an input/output operation executed on behalf of an io call;

- entry to an exit from a monitor, operations "delay" and "continue";

- time slice end;

- operation "init" for a process or a monitor.

In the original CPASCAL version, these events result in activation of the kernel via traps or interrupts; the kernel will then perform all the necessary PM operations. In our version it is also the case that the kernel is activated. However, the kernel acts only as a first-level interrupt handler which passes control to a PM written in CPASCAL. This is done by simulating calls of entry procedures to an object PM contained in the CPASCAL program(a).

Although shared, this object cannot conveniently be conceived as a monitor, because one of its duties is to provide for monitors. Thus PM should be implemented as a class object, which also prevents processes from entering it by explicit calls. However, lest PM is entered recursively (or by more than one process in a multi-processor system), it must operate in a low-level mutual exclusion state. It is the task of the PM interface in the kernel to establish this state before simulating the call to PM. The mutual exclusion state must be left upon return from PM. This can most conveniently be achieved by the following technique: the process does not return directly to the interrupted program but makes a little detour; the PM interface adjusts the process' stack in such a way that normal return from the class object PM results in leaving the mutual exclusion state before returning to the interrupted program.

Since the ready list is incorporated in PM, a special io operation like the one applied to the swapqueue is not required. An io operation for access to data administered by the kernel is not required either, for the following reasons:

1. monitor data are maintained within PM;

2. process control blocks are maintained within PM;

3. the exchange jump io accomplishes the necessary context switching;

4. if a process issues an io operation, PM is informed about this and can book that process as waiting for completion of io; in due time PM will be informed by the kernel about completion and can take appropriate action.

The only io operation that is required in addition to that on device "processor" is an operation on device "timer", e.g.

```
PROCEDURE io(CONST time: integer;
             CONST operation: (set);
             CONST device: (timer)     )
```

Note that the kernel does not know of processes, but only of coroutines. Regarding 4 the kernel only delivers the identity of a coroutine to PM which in turn associates a process control block with that coroutine. The correspondence between process control blocks and coroutines is established at system initialization time, by execution of the "init" operations for process objects.

System initialization proceeds along the same lines as in the original CPASCAL kernel. There is one additional issue, though; the PM object must be initialized, and all its entry procedures must be made known to the kernel. This task causes the same difficulties that urged us to implement VMM as a process instead of a monitor. Without changing CPASCAL, these difficulties are insurmountable. - We will not remedy this deficiency in our treatment of the subject because the underlying problem seems to be of a very trivial technical nature.

One further remarks concerning initialization is in order. If not all the PM functions are really used in some system, they need not be present and will consequently be omitted during initialization. E.g. if there are no more processes than processors, no processor preemptions and thus no time slicing is required; this means that if the timer is not used for other purposes, it will never be set by PM; in this case it is not necessary for PM to provide an entry procedure for timer interrupt handling.

4.4 Implementing PM outside the kernel in the way just described diminishes the kernel to an extent where it is responsible only for truly hardware dependent operations, e.g. driving a peripheral device or passing the processor to another coroutine. Exceptions only arise with "pseudo-devices" like swapqueue for VMM. Here the kernel also acts as a general servant which compensates technical difficulties.

This approach of a minimal kernel seems appealing not only for aesthetical reasons, but also because it effectively overcomes the annoyance with given fixed implementations of system parts like PM and VMM. The merits of the minimal kernel with respect to system portability are obvious.

If portability is of no concern the compiler may generate machine code for the target processor. In this case, if the quality of the compiled code can compete with that of good assembly code, efficiency is not notably degraded by removing PM from the kernel.

A look at the language MODULA with the above considerations in mind reveals a curiosity. The

(a) More sophisticated structures than a single monolithic object would be preferred in practice.

machine specific language feature called "device module" allows writing device drivers in the language, whereas process handling and synchronization are predefined (language features called process, interface module) and pre-implemented. This is made possible by treating the device drivers as special-kind processes.

5. Conclusion: Towards Really Safe Languages for Operating System Design and Implementation

The technique that has been exhibited in sections 3 and 4 is based on the attitude that a rich language is available (including a compiler) or will be designed and implemented in advance to system construction.

Such an approach is feasible and useful with languages like CPASCAL and MODULA for the construction of small-scale systems. However, if the construction of more powerful and versatile systems is intended, the language becomes more and more complex since it has to reflect all the system features. Furthermore, this implies that design of the language is tightly connected with system design. Since the latter is evolving only gradually, there is no point in having available or constructing in advance a super-language for operating system construction.

If we refuse to go back to the MOHLLs, two alternatives remain. The first one is working with, and at the same time designing, a language family the members of which are generated during system evolution, along with the functional hierarchy. The second and probably more attractive one is working with only one language that is not rich with special features, but powerful with respect to extension and shrinkage.

Extensibility can be achieved starting off with well-known data structuring facilities such as type and module. The notion of shrinkage means the hiding of system features that have been introduced previously by extension or existed a priori by support of a minimal kernel in the sense of 4.4. An example for the former features is given by a facility for disk access by block number that works with a built-in disk scheduling algorithm; this facility should be invisible "above" the file system, i.e. to all system parts that can use the file system. An example for the latter features is given by a "resume" operation for coroutines which must be invisible "after" it has been used to implement multiprogramming.

The possibility of shrinkage is absolutely necessary if the language is to support system integrity, i.e. to prohibit access to "dangerous" system features. An ad-hoc technique to establish programmer-controlled scopes is the use of exported/imported names ([Koster 76], [Wirth 77], [Lampson et al. 77]). - Another desideratum, selective access rights to system objects, can be enforced by simple technique as well [Jones/Liskov 76].

The notion of safe language encompasses a large spectrum of safeguards against inadvertant access to objects, calls of procedures, use of types etc., that are conceptually not available to the respective programs. CPASCAL sets an example for restrictive scopes but is poor with respect to hiding of low-level features. The worst concept of all is the use of standard procedures (notably "io") that are available throughout the programs.

Most desirable however, for operating system construction - and least considered by language designers until now - is the possibility of treating the hardware devices as objects in the language. The attraction of such an approach is due to language economy and true hardware independence. The programmer builds his operating system on top of a "standard prelude" which reflects a slightly embellished hardware. This embellishment is accomplished by a minimal run-time system that does not even contain device drivers but only machine dependent realizations of operations like "startio". This operation is one of the set of operations that apply to an abstract data type, say "printer". Hiding of this "real" printer from higher system levels will then be achieved by no other techniques than are used for any "normal" type.

Acknowledgement: The author owes much to disdussions with the students who realized the MUSIC system, Norwin Gräf, Horst Kretschmar, and Bernt Morawetz. Good advice from Peter Neumann and Nico Habermann helped to polish the paper.

References

[Atwood et al. 72] J.W. Atwood, B.L. Clark, J.J. Horning, M.S. Grushcow, K.C. Sevcik, R.C. Holt, D. Tsichritzis: Project SUE Status Report. TR CSRG-11, Univ. of Toronto (1972)

[Brinch Hansen 75a] P. Brinch Hansen: Concurrent Pascal Report. Calif. Inst. of Techn. (1975)

[Brinch Hansen 75b] P. Brinch Hansen: The SOLO Operating System. Calif. Inst. of Techn. (1975)

[Burroughs 70a] Burroughs Corp.: B 6700 ESPOL Reference Manual. The Burroughs Corporation, Detroit (1970)

[Burroughs 70b] Burroughs Corp.: Master Control Program Reference Manual. The Burroughs Corporation, Detroit (1970)

[Clark/Horning] B.L. Clark, J.J. Horning: The System Language for Project SUE. ACM SIGPLAN Notices 6,9 (1971)

[Gräf et al. 77] N. Gräf, H. Kretschmar, K.-P. Löhr, B. Morawetz: How to Design and Implement Small Time-Sharing Systems Using Concurrent Pascal. TR 77-09, Fachbereich Informatik, TU Berlin (1977)

[Habermann et al. 76] A.N. Habermann, L. Flon, L. Cooprider: Modularization and Hierarchy in A Family of Operating Systems, CACM 19,5 (1976)

[IFIP 74] W.L. van der Poel, L.A. Maarssen (Ed.): Machine Oriented Higher Level Languages. North-Holland Publishing Company, Amsterdam (1974)

[Jones/Liskov 76] A.K. Jones, B.H. Liskov: A Language Extension for Controlling Access to Shared Data. IEEE Trans. Softw. Eng. 2,4 (1976)

[Koster 76] C.H.A. Koster: Visibility and Types. ACM SIGPLAN Notices Special Issue 8,2 (1976)

[Lampson et al. 77] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, G.L. Popek: Report On The Programming Language EUCLID. ACM SIGPLAN Notices 12,2 (1977)

[Parnas 74] D.L. Parnas: On A "Buzzword": Hierarchical Structure. Proc. IFIP Congress (1974)

[Parnas 76] D.L. Parnas: Some Hypotheses About the "Uses" Hierarchy for Operating Systems. Fachbereich Informatik, Techn. Hochschule Darmstadt (1976)

[Ritchie/Thompson 74] D.M. Ritchie, K. Thompson: The UNIX Time-Sharing System. CACM 17,7 (1974)

[Ritchie 75] D.M. Ritchie: C Reference Manual. Bell Telephone Laboratories, Murray Hill (1975)

[Wirth 77] N. Wirth: MODULA: A Language for Modular Multiprogramming. Software - Practice and Experience, Vol. 7 (1977)

[Wulf et al. 71] W.S. Wulf, D.B. Russell, A.N. Habermann: BLISS: A Language for Systems Programming. CACM 14,12 (1971)

[Wulf et al. 74] W.S. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack: HYDRA: The Kernel of A Multiprocessor Operating System. CACM 17,6 (1974)