EWD275 - 0

Structure of an extendable operating system.

For an operating system of the scope and purpose as the THE-multiprogramming system, the conception of a linearly ordered hierarchy of layers has proved to be unsurpassed sofar: instead of "*complexity growing as the square of the number of system components*" (Aron), each next layer is easier to add, because the environment in which it is to ne understood is closer to our "ideal machine", i.e. our final goal. Knowing of no alternative arrangement with that pleasant property -and stronger: finding the existence of such an alternative hard to conceive!- I take the linearly ordered *structure as my starting point.*

In two vital respects, however, its structure calls for generalization.

1) The THE-system has been designed to bridge a pre-defined, constant gap, i.e. the gap between the given hardware (an EL-X8 with given *types of peripherals and backing store) and a given target*, i.e. five parallel ALGOL-machines. Construction of the system and use of the system were, at that time, viewed as completely different activities, different in nature and taking place at different times. In the mean time, I regard the intermediate product, consisting of the hardware, covered by a few of the lower layers, as " a system" with its own right of *existence and I regard the activity of adding a next layer no longer* so radically different from what a user does when he feeds his ALGOL-program into the completed system. In other words, I would like to treat wherever possible, system embellishemt and system usage as much as possible on the same footing. In this case, the natural extension of a fixed set of layers becomes a stack of layers.

2) *The target machine of the THE-system is not only constant in* time, it is also homogeneous in the sense that we produce a set of equivalent ALGOL-machines. (In actual fact, one of the five is slightly special purpose, but that is irrelevant in this discussion.) We wish to create in parallel different environments. The obvious generalization of the stack of layers -corresponding to strictly nested environments- is a tree of layers. And if I were to design an operating system, I would do my utmost best to conceive it in such a fashion, that I can recognize such a tree structure, and this for two compelling reasons: as just shown, a tree structure is the very least we can hope to come out with, on the other hand it is my repeated experience that a tree-structure is about the most complicated structure of hierarchical nature that I can cope with. I don't feel like having a choice anymore.

Note. My above use of "parallel" refers to conceptual parallellism, or "independence", not to parallel execution in time of a number of sequential processes: what is regarded as a single layer, may contain a number of co-operating sequential processes! (The function of such a layer is to transform one machine into another: both source- and target-machine may contain asynchronous elements!)

I have given much thought -not all of it conclusive- to the question what abstraction from the hardware should be implemented in what layer. In particular I have been thinking of the question whether the bottom layer should implement the sequential processes and the next one the virtual store (as in the THE-system) or the other way round. The argument for the THE-arrangement was that the implementation of the virtual storage required -or at least could make good use of- a sequential process (the so-called "segment controller") synchronized with the drum interrupts. If we take as a backing store a device with such a high transfer rate that demand paging is no longer a motivation for change in processor allocation, that argument is no longer valid. Yet I am inclined to try to put processor allocation as low as possible, because above that level, the interrupts have disappeared and the sooner that is achieved, the better. In the THE-system processor allocation at level zero is easy, because the number of processes is constant and the relevant information is kept resident in core. In an extendable operating system, however, the total number of processes (and semaphores or what you have) will vay in time and fixed allocation in core seems out of the question. My present idea is to look for a mixture: layer zero (and this may be in the hardware) will contain the acces mechanisms for virtually addressed information "provided it is in core". Layer one will contain processor allocation, described in terms of virtual addresses, whereas the next layer will be storage allocation, which has the duty never to remove (but possibly to move in core) information pertaining to level one. As a result, level one will hardly make fixed core committments, yet will never produce a page fault.

Note. The concept of a virtual store is not restricted to multilevel storage machines. Also on a machine with a single level storage it is conceivable that virtual store is a very useful concept. In the latter case. level two would restrict its activity to rearrangement of the core store contents.

The function of a layer in the THE-system is to provide some facilities in terms of some others. Take for instance the layer in which peripherals are allocated. Below this layer we have unbuffered

physical peripherals, above it we have information streams, i.e. buffered virtual peripherals. Above that layer a maintenance engineer has no longer the possibility to introduce a program for the unbuffered exercising of a specific peropheral, for it "is no longer there". If we wish to create that facility, one of the peripherals should be taken out of the common pool serving the virtual units and suddenly we have two environments: one environment for programs referring ot virtual units (to be served by the remaining physical units) and one for the maintenance engineer. In a situation like that I see the first traces of a tree of environments emerging (environment is here used as "interface between software layers").

1) the environment with the physical, unbuffered peripherals
2) the environment with the specific unbuffered peripheral retained, plus virtual peripherals (to be served by the pool of the remaining physical peripherals)
3a) for the users the environment of the virtual units
3b) for the maintenance engineer the environment of the specific unit. (This example may be too simple: in practice, the maintenance engineer will probably use the remaining virtual units as well. OK, then we have the identity transformation -probably implemented by an empty layer- between 2 and 3b.)

A layer performs the transition from one environment to an environment on top of it. A translator has the function to produce such a layer. (In the THE-system we have at the users disposal an ALGOL-environment: when his program has been translated he has temporarily added a new layer, which provides the environment in which his date can be "interpreted"). This implies that a translator can only work when the environment, which serves as the "underface" of the layer it has to produce, is known, i.e. for its activation we must give "the underface" as a parameter. For a variety of reasons I am strongly inclined to relate this "underface parameter" very closely to the environment from which the translator is called. Traditional ALGOL-translators supply the environment of the Library as a constant presetting of (the bottom of) the name stack, as if the library has been declared in an (implicitly embracing) outer block. I would like to repeat this arrangement for inner blocks: a translator first translates the outermost block, thereby creating the environment in which the next set of inner blocks are to be understood. Then -and this closely resembles the start of the execution of the declarative part of the outermost block- the translator is called again for the translation of the procedures declared in the outermost block; these translator calls will get the newly created environment as the underace parameter. Here we have an arrangement in which the creation

of a next layer can be considered as an activity to be understood in the environment which equally serves as the underface for the level to be created and this repeatedly. It is essential that the new environment can again accept programs. It is by such means that I should like to achieve that "building up and tayloring" of the system is not essentially different from using the system, that I hope to treat "job control language" and normal programming language on exactly the same footing.

Acknowledgements are due to Harry Whitfield, Edinburgh University, XXXX whose experience with software design for an ICL745 (a 360/67 XXX type machine) had made him think of more attractive alternatives.

november 1969 Edsger W.Dijkstra