Complexity controlled by hierarchical ordering of function and variability.

by Edsger W.Dijkstra

Reviewing recent experiences gained during the design and construction of a multiprogramming system I find myself torn between two apparently conflicting conclusions. Confining myself to the difficulties more or less mastered I feel that such a job is (or at least should be) rather easy; turning my attention to the remaining problems such a job strikes me as cruelly difficult. The difficulties that have been overcome reasonably well are related to the reliability and the producibility of the system, the unsolved problems are related to the sequencing of the decisions in the design process itself.

I shall mainly describe where we feel that we have been successful. This choice has not been motivated by reasons of advertisement for one' own achievements; it is more that a good knowledge of what -and what little!- we can do successfully, seems a safe starting point for further efforts, safer at least than starting with a long list of requirements without a careful analysis whether these requirements are compatible with each other.

Basic software such as an operating system is regarded as an integral part of the machine, in other words: it is its function to transform a (for its user or for its manager) less attractive machine (or class of machines) into a more attractive one. If this transformation is a trivial one, the problem is solved; if not, I see only one way out of it, viz. "Divide and Rule", i.e. effectuate the transformation of the given machine into the desired one in a modest number of steps, each of them (hopefully!) trivial. As far as the applicability of this dissection technique is concerned the construction of an operating system is not very much different from any other large programming effort.

The situation shows resemblance to the organization of a subroutine library in which each subroutine can be considered as being of a certain "height", given according to the following rule: a subroutine that does not call any other subroutine is of height 0, a subroutine calling one or more other subroutines is of a height one higher than that of the highest height among the ones called by it. Such a rule divides a library into a hierarchical set of layers. The similarity is given by the consideration that loading the subroutines of layer 0 can be regarded as a transformation of the given machine into one that is more attractive for the formulation of the subroutines of layer 1.

Similarly the software of our multiprogramming system can be regarded as structured in layers. We conceive an ordered sequence of machines: A[0], A[1],... A[n], where A[0] is the given hardware machine and where the software of layer i transforms machine A[i] into A[i+1]. The software of layer i is defined in terms of machine A[i], it is to be executed by machine A[i], the software of layer i uses machine A[i] to make machine A[i+1].

Compared with the library organization there are some marked differences. In the system the "units of dissection" are no longer restricted to subroutines, but this is a minor difference compared with the next one. Adding a subroutine of height 0 to the library is often regarded as an extension of the primitive repertoire which from then onwards is at the programmer's disposal. The fact that, when the subroutine is used, storage space and processor time have been traded for the new primitive can often be ignored, viz. as long as the store is large enough and the machine is fast enough. Consequently the new library subroutine is regarded as a pure extension. One of the main functions of an operating system, however, happens to be resource allocation, i.e. the software of layer i will use some of the resources of machine A[i] to provide resources for machine A[i+1]: in machine A[i+1] and higher these used resources of machine A[i] must be regarded as <u>no longer there</u>! The explicit introduction (and functional description!) of the intermediate machines A[1] through A[n-1] has been more than mere word-play: it has safeguarded us against much confucion as is usually generated by a set of tacid assumptions. Phrasing the structure of our total task as the design of an ordered sequence of machines provided us with a useful framework in marking the successive stages of design and production of the system.

But a framework is not very useful unless one has at least a guiding principle as how to fill it. Given a hardware machine A[0] and the broad characteristics of the final machine A[n] (the value of "n" as yet being undecided) the decisions we had to take fell into two different classes:

1) we had to dissect the total task of the system into a number of subtasks
2) we had to decide how the software taking care of these various subtasks

should be layered. It is only then that the intermediate machines (and the ordinal number "n" of the final machine) are defined.

Roughly speaking the decisions of the first class (the dissection) have been taken on account of an analysis of the total task of transforming A[0] into A[n], while the decisions of the second class (the ordering) have been much more hardware bound.

The total task of creating machine A[n] has been regarded as the implementation of an abstraction from the physical reality as provided by machine A[0] and in the dissection process this total abstraction has been split up in a number of independent abstractions. Specific properties of A[0], the abstraction from which we wanted to implement, were:

1) the presence of a single central processor (we wanted to provide for multi-programming)

2) the presence of a two level store, i.e. core and drum (we wanted to offer each user some sort of homogeneous store)

3) the actual number, speed and identity (not the type) of the physically available pieces of I/O equipment (readers, punches, plotters, printers etc.)

The subsequent ordering in layers has been guided by convenience and was therefore, as said, more hardware bound. It was recognized that the provision of virtual processors for each user program could conveniently be used to provide also one virtual processor for each of the sequential processes to be performed in relatively close synchronism with each of the (mutually asynchronous) pieces of I/O equipment. The software describing these processes was thereby placed in layers above the one in which the abstraction from our single processor had to be implemented.

The abstraction from the givven two level store implied automatic transports between these two levels. A careful analysis of, one the one hand, the way in which the drum channel signalled completion of a transfer and on the other hand the resulting actions to be taken on account of such a completion signal, revealed the need for a separate sequential process -and therefore the existence of a virtual processor- to be performed in synchronism with the drum channel activity. It was only then that we had arguments to place the software abstracting from the single processor below the software abstracting from the two level store. In actual fact they came in layer 0 and 1 respectively. To place the software abstracting from the two level store in layer 1 was decided when it was discovered that the remaining software could make good use of the quasi homogeneous store. Etc. It was in this stage of the design that the intermediate machines A[1], A[2],... got defined (in this order).

At face value our approach has much to recommend itself. For instance, a fair amount of modularity is catered for as far as changes in the configuration are concerned. The software of layer 0 takes care of processor allocation; if our configuration would be extended with a second central processor in the same core memory then only the software of layer 0 would need adaptation. If our backing store were extended with a second drum only the software of layer 1, taking care of storage allocation, would need adaptation, etc.

But this modularity (although I am willing to stress it for selling purposes) is only a consequence of the dissection and is rather independent of the chosen hierarchical ordering in the various layers, and whether I can sell this, remains to be seen. The ordering has been motivated by "convenience"....

The point is that what is put in layer 0 penetrates the whole of the design on top of it and the decision what to put there has far reaching consequences. Prior to the design of this multiprogramming system I had designed, together with C.S. Scholten, a set of sequencing primitives for the mutual synchronization of a number of independent processors and I knew in the mean time a systematic way to use these primitives for the regulation of the harmonious co-operation between a number of sequential machines (virtual or not). These primitives have been implemented at layer 0 and are an essential asset of the intermediate machine A[1]. I have still the feeling that the decision to put processor allocation in layer 0 has been a lucky one: among other things it has reduced the number of states to be considered when an interrupt arrives to such a smaal number that we could try them all and that I am convinced that in this respect the system is without lurking bugs. Fine, but how am I to judge the influence of my bias due to the fact that I happened to know by experience that machine A[1], with these primitives included, was a logically sound foundation?