OVERVIEW OF THE HYDRA OPERATING SYSTEM DEVELOPMENT

W. Wulf, R. Levin, C. Pierson[1]
Carnegie-Mellon University
Pittsburgh, Pa.

Abstract: An overview of the hardware and philosophic context in which the Hydra design was done is discussed. The construction methodology is discussed together with some data which suggests the success of this methodological approach.

Key Words and Phrases: Operating systems, kernel, policy/mechanism separation, capability-based protection, modular decompositions, programmer productivity.

1. Introduction

This paper has two objectives: first, to set the context for the two companion papers in these proceedings, and second to discuss the methodology used in the implementation and our experience with it. To fulfill the first objective we first briefly discuss the hardware environment on which Hydra was implemented, then discuss the philosophy on which the system is based, and finally exhibit some of the ways in which the philosophy is instantiated. The final section discusses the construction methodology.

2. The Hardware Context

C.mmp is organized as a canonical multiprocessor computer system; it consists of a number of equal, asynchronous central processors (Pc's) that share a large primary memory. C.mmp differs from earlier multiprocessors such as the Burroughs D825, IBM 360/67, Honeywell 645 (Multics), etc. in two essential respects:

1. It is designed to have up to 16 Pc's (presently there are 6) while other multiprocessors usually have no more than 2.

2. It is constructed with minicomputer Pc's (DEC PDP-11's [DEC73]) rather than the larger (32 to 48 bits/word) Pc's used in the other multiprocessors.

In other words, the effective use of C.mmp requires that we find and exploit a much higher degree of parallelism than has been needed by other multiprocessors in the past. The

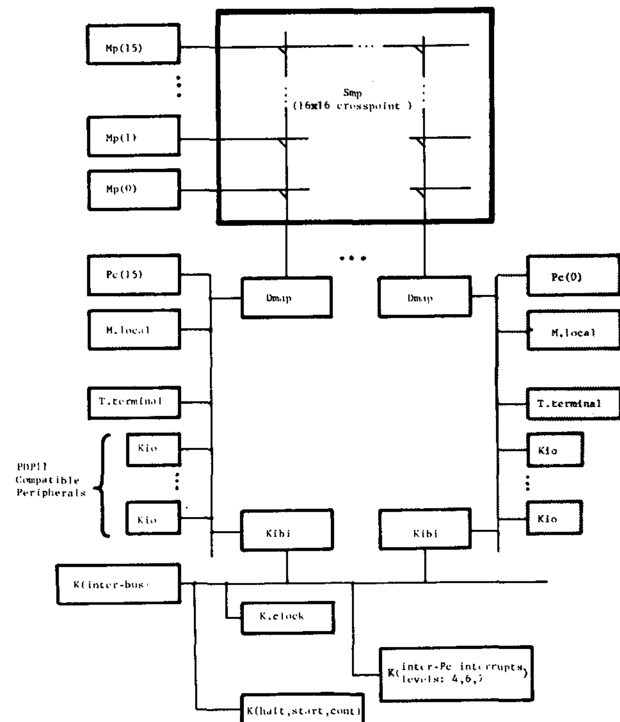

Figure I

only other current multiprocessor comparable to C.mmp is the 14 Pc Pluribus computer system at BBN [Hea73]. To date Pluribus has been successfully applied to the task of digital communications; it remains to be seen how well Pluribus can be applied to a wider range of applications.

The C.mmp computer system is illustrated in figure I. As may be seen, the principal components are 16 modules of shared memory, Mp(0:15); a 16 x 16 switch, Smp; 16 processors, Pc(0:15); address relocation hardware, Dmap, associated with each processor; and an interprocessor bus with special devices attached to it (K.clock, K.halt, etc.).

It is crucial to an understanding of C.mmp to appreciate that, from the outset, it was envisioned as a distributed system. All components were envisioned as a pool of resources to be shared among whatever tasks were to be done. This was to be (is) true of processors, I/O, and memory singly and in combination. There was to be no master-slave relation between the processors, for example, and any process (user job) was to be able to execute on any processor at any instant.

1. This work was supported by the Defense Advanced Research Projects Agency under Contract F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research.

Two features of the hardware organization significantly impact the operating system structure and thus are discussed in somewhat more detail: (a) the memory address translation (relocation), and (b) the interprocessor communication mechanism.

2.1 Memory Mapping and Relocation

Probably the greatest problem in building a large computing system from minicomputers is their small address space. In C.mmp we must be able to address several million bytes of primary memory from the processors. The basic PDP-11 architecture, on the other hand, is only capable of generating 16 bit addresses. Although the processor (i.e., programs operating on it) may generate only a 16 bit address, the Unibus supports an 18 bit address, and the shared memory uses a 25 bit address. Somewhat arbitrarily we chose to divide these address spaces into 8K-byte units called pages. Thus processor generated addresses are divided into 8 pages, Unibus addresses are divided into 32 pages, and the shared memory is divided into 4096 pages. Recall that the processor generates a 16 bit address but that 18 bits are present on the Unibus. As shown in figure II, the two extra bits are obtained from the program status register (PS) in the processor. As we shall see in a moment, these bits may not be altered by a user program. Thus user programs are actually bound to operate within the eight pages described by a subset of the relocation registers. Such a subset is called a space and is named by the two bits, e.g. '00' or '11' space.

The address mapping registers and PS register are themselves located in the peripheral page and therefore can only be accessed by a process in the '11' space (i.e. a process executing with the PS bits 8 and 9 set to '11'). Thus a user program cannot directly alter the address mapping. Operating system operations are provided, however, so that after appropriate validation, the user may manipulate it.

Each relocation register also contains a number of control and status bits:

The non-existent memory bit can be set by the operating system to prevent access to shared memory through the register. This permits the system to place a small user job in the machine without allocating a full 64K byte address space.

The write protect bit, when set, permits read cycles to proceed through the register while blocking write cycles. This feature can be used to guarantee the integrity of reentrant code.

The written into ('dirty') bit is an indicator which flags write cycles through a register. This provides an inexpensive mechanism to facilitate paging out only those pages which have been changed.

2.2 The Interprocessor Communication Mechanism

Interprocessor communciation is an important consideration in controlling a multiprocessor. Furthermore, in a fully distributed multiprocessor it is necessary for each processor to control these functions on every other processor. This control is provided by an interprocessor bus, a controller for it, and interfaces to it (see Figure I).

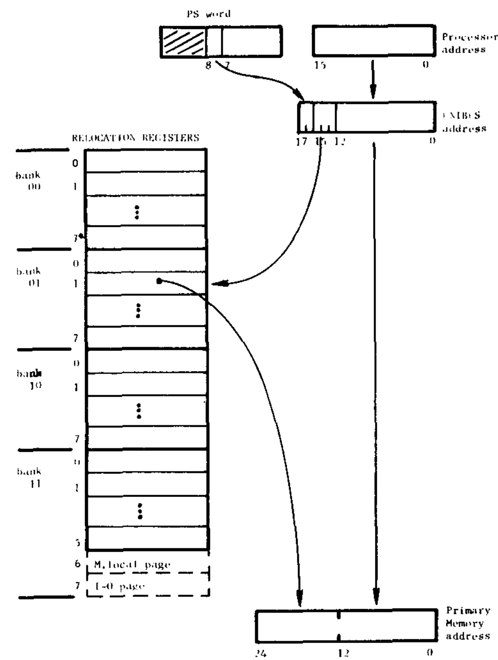

Figure II

The interface allows a processor to evoke a certain function on any subset of the processors, including itself, by simply 'ORing' a mask into the interface register associated with that function. The interface currently contains six such registers, one each for HALT, START, CONTINUE and three for different levels of interruption. Each of these function registers is 16 bits wide. Setting the i†[th] bit of the register (to one) associated with one of the functions will evoke that function on the ith processor. Thus, for example, moving a mask of all 1's into the halt register will stop the entire machine. In addition to these functions, the bus provides other facilities to the processors; such as a (per processor) programmable interval timer and a 56-bit, one-microsecond resolution, time-of-day clock.

3. The System Philosophy

The basic philosophy upon which the specific Hydra mechanisms rest is a desire to allow nearly all of the facilities one normally associates with an operating system to be built as "normal" user programs. This central goal suggests that at the heart of the system one should build a collection of basic, or "kernel", mechanisms of "universal applicability" -- a set from which arbitrary user-visible operating system facilities can be conveniently, flexibly, efficiently, reliably, and quickly constructed. Moreover, lest the flexibility be constrained at any instant, it should be possible for an arbitrary number of systems created from these mechanisms to co-exist simultaneously.

This is obviously a tall order. Nevertheless, Hydra is an attempt to provide just such a set. Whether or not the particular Hydra mechanisms satisfy these criteria and whether or not they form the best set, are, of course, still open

questions. The answers will come, if at all, only after extensive use.

We can easily rationalize two properties that the kernel mechanisms must possess: (1) protection and (2) no policy. The Hydra mechanisms to support these properties are discussed in two companion papers; here we focus only on the rationale for them. Consider for the moment two common descriptions of the purpose of an operating system:

1) An operating system provides a "virtual machine" which is more hospitable than the base hardware for two reasons: (a) it makes available certain "virtual resources" such as files, directories, virtual memory, etc., absent from the base hardware. (b) It makes certain unpleasant hardware features, such as interrupts, from the user and maps them into more acceptable ones, such as P-V synchronization primitives.

2) An operating system manages the physical resources of the computer, such as primary memory, processor, channels, etc. so as to improve their utilization.

Even though these descriptions are quite different they are not incompatible -- they merely express two quite different views of what is a single object with multiple goals.

From the first of these descriptions we see that an individual (program) must be able to behave as though it is running in isolation; that is, as though it has exclusive access to the machine. In practice, of course, we relax this slightly by saying "except for possible differences in real-time behavior". With this exception, however, we see that a uniform requirement of all "operating systems" is that they provide protection. In our case, since operating systems are themselves user programs, the only candidate for providing the necessary protection is the kernel.

From the second description we derive a negative criterion on the kernel mechanisms -- namely that they should not impose a policy on the way in which (physical) resources are used. If the kernel mechaisms were to do this they would preempt the possibility of specifying these at the user level -- and hence preclude an important dimension of operating system variation. We refer to this negative criterion as the principle of "policy/mechanism separation"; Brinch Hansen [Bri70] has made cogent arguments for this separation.

The fact that the kernel should provide protection but should not define resource policies does not of itself provide sufficient information on which to base a design; it merely specifies some properties that the design must have. To develop the appropriate basis for the design we choose to turn away from traditional operating system design considerations and to look instead at some of the more recent results of "structured programming".

3.1 Program Structuring

It is unfortunate that the term "structured programming" has too often been equated with "goto-less programming" or "top-down design". Far more central to the issue is the concept of "abstraction". Several authors have noted the close relation between many programming abstractions and the concept of "type" as it appears in programming languages [DDH74,Bri73,Wul74b]. Specifically, the concept of a "class" in Simula '67 [Dah66] and its extension to "monitors" [Hoa74,Bri75], "clusters" [Lis74], and "forms" [Wul74b] seems especially well suited to expressing these abstractions. A class in Simula defines an abstract data type by specifying both an underlying storage structure and a set of operations which operate on it.[2] Thus, for example, the abstract concept of a set of integers might be introduced into a language by a definition of the form[3]

```
type intset =
  begin
    var a: array[1:100] of integer, n: integer;
    op union(u,v: intset) returns(intset); begin ... end;
    op intersect(u,v: intset) returns(intset); begin ... end;
  end;
```

Such a definition is intended to describe how any particular variable of type intset is to be represented and how operations on this type of variable are to be performed. Thus the declaration "var a: array[1:100] of integer, n: integer;" defines how storage is to be allocated for each variable of type intset. The operator definitions, e.g. that for "union", define how such variables are manipulated. An important property of such definitions is that all the representational information is localized and "hidden" [Par72a,Par72b,Par72c] in the type definition; the only way to manipulate variables of a defined type is by invoking the operations defined in the type definition.

After having made such a definition, the programmer may write such things as declarations of variables of type intset and statements which operate on these sets, e.g.

```
var a,b,c: intset;
a := union(b,c);
```

This style of programming captures the notion of abstraction because it effectively separates the application of the abstract "primitives" from the details of their implementation. The programmer, working at a level where intsets are an appropriate medium of expression, need never concern himself with the details of how they are represented or manipulated. Conversely, the implementor of the realization of the type intset may freely alter that realization (to improve efficiency, for example) without concerning himself with the details of how it is used, so long as he preserves the functional properties of the operations.

It is not our purpose here to advocate a particular approach to structuring programs. However, the brief description given above is the model on which Hydra is based. Except for a slight change in terminology, extensions to provide protection, and a more dynamic definition of types than is common in programming languages, the Hydra kernel mechanisms were chosen to support this model. The same structuring philosophy is also used in the implementation of the system.

---

2. The definition of a data type in terms of both its representation and the operations on it, is called an "extensive" definition; an "intensive" definition is one in which only the objects of the given type are defined.

3. We have purposely chosen a neutral syntax whose meaning should be clear; it is not Simula '67 or any other specific language.

Earlier in this discussion we used the phrase "virtual resources" to describe some of the facilities provided by an operating system (e.g. files). Notice that this phrase is essentially identical to the word "type" as used in the immediately preceding discussion. A virtual resource (e.g. file, directory, semaphore, ...) is an abstract concept with a set of operations defined on it (e.g. read, write, append, open, close, ...). Moreover, the virtual resource has some realization in terms of more primitive concepts (e.g. disk segments). Just as with well-structured programs, we want the user of the file system to be unconcerned with the details of its implementation. Conversely, we want the implementor of the file system to focus on the issues related to that specific realization without concern for the details of the idiosyncratic use of a particular file.

Without yet concerning ourselves with the details of the Hydra mechanisms, we proceed by analogy with the programming language model and list further properties which these mechanisms must have (the first two are copied from the earlier discussion for completeness):

- protection
- policy/mechanism separation
- creation of new kinds of virtual resources (new types)
- specification of the representation of, and the operations on a virtual resource (type)
- creation of instances of a resource (type)
- application of operations to an instance of a resource (type)
- certain "generic" operations, e.g. "storing", which are applicable to all resources (types)

3.2 The Protection Mechanism

In the sequel we shall often use the phrase "subsystem" when speaking of operating system facilities; it will mean essentially the same thing as "type definition" in the previous discussion. That is, a subsystem is a collection of information which specifies the representation of a virtual resource (type) and the nature of the implementation of various operations on that type of resource. All knowledge about these representational and operational details are contained and "hidden" within the subsystem. In those cases where resource allocation (policy) issues are involved, these policies are also embedded in the subsystem. Global knowledge about a specific type of virtual resource is limited to that supplied in the external specifications of the subsystem which implements that resource. Manipulation of the representation of a resource is restricted by the protection mechanism to only that code which defines the operations within a subsystem.

At this point we can pose a question about the protection structure of the system which we purposely avoided previously, namely "what should be protected and against what"? This apparently simple question is complicated by two issues; one endemic to operating systems, the other arising from the primary goal of Hydra.

First, we recognize that sharing is as important as protection. That is, we don't really want complete isolation of the virtual machines seen by various users. Users want to selectively share files, pages, directories, semaphores, or any of the other virtual resources provided to them. This is true in any "computing utility" [Gra68], but especially so in a multiprocessor where a single user will wish to divide his job into parallel cooperating processes and share resources between these processes. Second, because we wish to provide virtual resources through user-level programs, we don't know _a priori_ what kinds of resources will exist. Hence we don't know what sorts of things will need to be protected, or what sorts of access should be granted (or inhibited) to them.

Both of these questions can be answered in terms of the model posited above. The objects to be protected are instances of virtual resources. Since we shall insist that _only_ the operations defined to operate on a type may manipulate the representation of objects of that type, we shall provide protection against the application of those operations to instances of that type. Thus, for example, suppose type 'file', with associated operations 'read', 'write', 'append', 'open', etc. has been defined. The protection mechanism will allow application of, for example, the 'write' operation to specific instances of files to be selectively granted or inhibited. (Clearly, similar protection must be (and is) provided against application of the generic operations provided by the kernel, e.g. 'store'.)

The "capability" based protection mechanisms [Den66,Lam69,Fab74, GD72] are ideal for supporting the philosophy espoused above. With two extensions these mechanisms were adopted for Hydra. The Hydra version of capability protection is discussed in [Wul74a] and is further elaborated in the companion paper [CJ75]. It is desirable, however, to mention here the two extensions and their relation to the previous discussion.

(a) Capabilities in Objects: In order to implement new abstractions, the representation of these abstractions must be expressed in terms of extant object types, and the operations on the new abstraction must be expressed in terms of the operations of the representing objects. In the general case the representing objects will be arbitrary abstractions realized in terms of yet further abstract types. To support this Hydra allows an object to be represented in terms of both simple data and capabilities (protected references) for other objects.

(b) Rights Amplification: The operations provided by a subsystem, in general, may require quite different accesses to an object than those required by the user of the abstraction. In order to achieve the goal that representational information should be 'hidden', and thus that there should be operations which the subsystem can perform but users cannot, we provided a mechanism for expanding the rights to an object when a subsystem is invoked.

3.3 Summary of the Philosophy

The basic philosophy on which the design of Hydra rests is essentially the admission that its designers were incapable of anticipating all of the ways in which it might be used. Thus there has been a conscious attempt to permit almost all of the functions normally associated with an operating system to be (re)defined by the motivated user -- both in terms of the facilities provided and the policies which govern the use of the

resources of the system. As a corollary of this philosophy, Hydra attempts to preempt as few resources and modes of using them as possible.

There appear to be limits beyond which this philosophy cannot be pushed and still permit simultaneous access by competing users. Thus, for example, Hydra defines a basic protection mechanism which must be used by all. We have chosen a capability mechanism because of its close relation to what we believe to be good program structuring principles. Although, for example, some user may define an authority-based file protection mechanism on top of the capability scheme provided by Hydra, he can never be totally unaware of the underlying capability-based protection mechanism. Thus there are limits on the variability possible within the Hydra scheme; we did not try to provide a "virtual machine" identical to the C.mmp hardware.

There are also limits placed on the users with respect to the policies which may be defined to govern resource utilization. Both for protection and efficiency reasons, limits are placed on the policies which may be defined by the user (see [Lev75]).

Within these limitations however, we believe, and our early experience corroborates, that users will be able to define a broad spectrum of facilities and policies.

4. The Construction Methodology

The previous sections have dealt with the hardware and philosophic context in which the Hydra design was done. In this section we shall focus on the approach to the detailed design and implementation and summarize some of our experiences.

4.1 Stages in the Design

Perhaps the earliest major design decision was the choice of the kernel approach. To determine the precise set of primitives to include in the kernel, we concentrated heavily on the "virtual machine" Hydra should provide. Since we wished to provide maximum flexibility in the use of the multiprocessor facilities, we emphasized the notion of protected access to resources, from which evolved the abstracted resource (object), access control mechanism (capability), and protected environment (LNS). Object types and rights amplification supported the subsystem concept, which we viewed as the key idea in construction of operating system facilities outside the kernel.

Once the important kernel primitives had been postulated, we turned to the task of designing selected subsystems (e.g. directory). This initial "user system" design would, we hoped, expose the shortcomings of the kernel primitives and permit us to iterate the kernel design before implementation proceeded too far. In fact, several iterations occurred and at least one more is presently planned. It should be noted, however, that much of the kernel and subsystem design could and did proceed in parallel. Furthermore, since not all of the kernel facilities were required by these initial user subsystems design of these additional primitives could also occur in parallel. In fact, the individuals within the development group worked as both designers and implementors, with some implementing one part of Hydra while designing another. The feasibility of this overlapping derives from the modular decomposition approach, to be discussed next. However, it should be emphasized here that the stages in the system design were therefore not strictly sequential in time. Figure III presents a time-line of the construction of major system components.

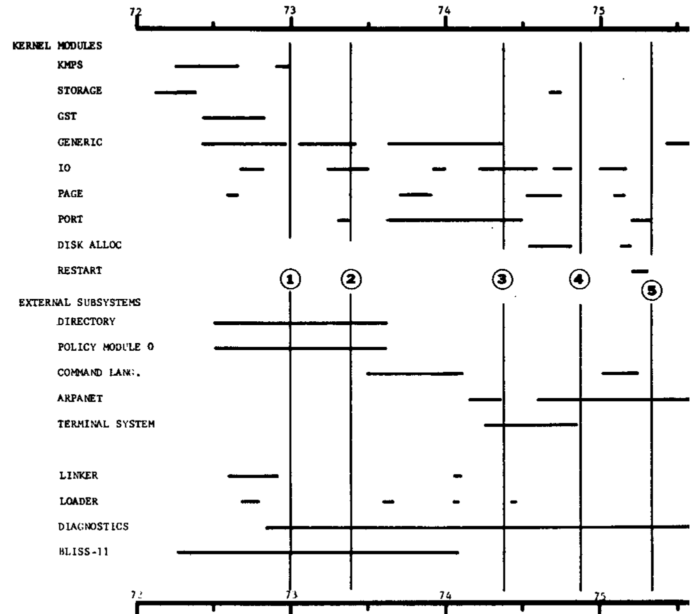

Figure III

4.2 The Decomposition Strategy

The subsystem notion is pervasive in Hydra and extends to the implementation of the kernel. In the early phases of the implementation, when the subsystem concept was a central concern, we expended considerable effort to decompose the kernel into a set of carefully-specified modules. We were strongly influenced by the work of Parnas [Par72b], but did not, for reasons to be discussed later, adopt the specification technique he proposes. We did, however, establish some rules for module construction, which might be regarded as informally defining our notion of "module"; these are a subset of the Parnas criteria:

1) A module performs a well-defined set of actions, and specifies the necessary conditions for successful operation of those actions.

2) The actions of a module are made available to other modules as functions, and these functions constitute the only access to the services of the module. In particular, the data manipulated by the module is only made available to other modules by function invocations; other modules have no direct knowledge of the location or representation of any data used by the module to implement its actions.

3) Modules detect conditions which violate their specifications and prevent execution of functions when the necessary conditions are not met.

The first two rules obviously relate closely to the subsystem concept and follow directly from the earlier discussion. The third rule stresses reliable operation of modules so that a local failure will frequently be detected before it can spread and cause serious disruption [Par72d].

Although these module specifications were produced in detailed written form for only a few modules, the techniques served as the basis for the entire Hydra implementation. We believe that the localization of data structure manipulations and the careful encapsulation of functions as module actions strongly encourage both coherent decomposition and flexible maintenance of the system software. We did not follow the rigid regime of formal specification advocated by Parnas in part because of a desire to "get on with it", but also because our early attempts to follow this regime were frustrated by several problems:

1) Recursive structures. Recursive structures commonly cause trouble in any hierarchical decomposition, because one is essentially forced to produce simultaneous specifications for two or more modules. In Hydra, this situation arises, for example, in connection with the protection data base. A capability references an object, which in turn contains a list of capabilities. It is difficult to separate objects, capabilities, and C-lists in a formal specification in which this recursive relationship obtains.

2) Evolution. All operating systems, even the most carefully designed ones, evolve. A precise, formal specification has difficulty accommodating such perturbations; informal ones are less brittle. Our experience suggests that the effort necessary to update formal specifications (compared to informal ones) is often not expended,[4] leaving only the original, obsolete ones. These are undoubtedly worse than up-to-date, informal specifications.

3) Context conditions. Not all calling conditions are easily expressed as restrictions on the parameters; some are best described as limitations on the context in which the functions may be invoked. For example, certain routines may only be callable inside a critical section, or may not be executed as part of an interrupt servicing operation. Formal descriptions of such conditions tend to be represented in terms of "state funntions" which are constructed to capture context for which no explicit data representation is available. Such devices are of obvious use if the specifications are to be subjected to mechanical verification. Conversely, if the specifications are intended to guide the implementation and debugging processes, less formal and more intuitive context conditions are preferable.

4. This results at least in part from the nature of our research environment.

It should be noted that even an informal specification technique requires descipline. If one adopts the formal specification approach and fails to follow through, at least a partial specification results. With our technique, a lack of discipline results in no useful specification at all. Where we failed to complete specification, we later encountered trouble in debugging because some case had been overlooked as a direct consequence of our informal approach. Thus this technique has its risks.

It is important that the source language version of the implementation reflect the module decomposition of the design. To do this most effectively it would have been ideal if the implementation language had provided a type definition mechanism similar to the Simula "class" concept (see section 3.1); unfortunately such a mechanism was not available to us and the expense of designing and implementing a new language did not seem justified. Thus we set out to achieve the same effect by imposing an appropriate discipline upon our coding practice.

In particular, each abstraction (module) is implemented as (at least) two files. One of these files defines the data structures and routines which implement the operations of the module. This file may be (is) compiled separately. The second file contains the declarations necessary for another module to create instances of the abstraction and apply the operations, functions, defined in the first file to these instances of the abstraction. This second file is not, indeed cannot be, compiled separately. Rather, upon request it is automatically included as part of the compilation of other modules which use the abstraction. This rather simple discipline achieves two goals: First, it localizes all of the code dealing with a particular abstraction to a single module, usually two files, thus making changes relatively easy. Second, it "hides" all the representational decisions in these same files; the only access to the abstraction is through the interfaces declared in the second file.

(Note, however, that the interfaces are often macros so that the ultimate object code is often distributed quite differently than is apparent from the source code. This is essential to the efficiency of the system.)

4.3 Stages in the Construction

Construction of a complex system is a difficult task, even with a carefully specified design. We can summarize the lessons we learned in a single maxim: "Use the best tools you can reasonably obtain. If the tools aren't available when you start, build them first." This is another of those "motherhood" statements and is difficult to dispute. When one is actually in the throes of implementation, however, it is easy to ignore. This section indicates the cases in which we heeded this rule and reaped the benefits, and points up the times when we ignored it and paid the price.

The task of system construction was complicated by the embryonic state of our eventual multiprocessor configuration, and the instability of the hardware available. (See Figure IV.) We needed a more stable tool than C.mmp on which to establish a base of development operations. Happily, we had a PDP-10 system available, and we used it extensively for programming Hydra and suppport software.

Having established the hardware to be used, we next selected an implementation language. The arguments for building systems in higher-level languages are well-known; we will not repeat them here. We chose BLISS [Wul71].

Our next problem was the near impossibility of debugging Hydra itself, because of the lack of adequate tools. We therefore chose to implement the initial modules of the kernel (and several subsystems) in a compatible subset of Bliss/10 (for the PDP-10) and Bliss/11 (for PDP-11's), and to run and debug these modules on the PDP-10. Because we were concerned with synchronization errors, the PDP-10 version was designed to switch (simulated) processors before and after every critical section, thereby increasing the probability of actually encountering a latent synchronization error. This facility enabled us to detect a number of subtle mistakes. Clearly, we could not detect all the timing-dependent problems using this approach, but the vast majority of other logical errors were found and corrected.

To facilitate check-out, we also incorporated a debugging mechanism which allowed us to control the execution of the several "processors", start and stop selected processes, inspect process state, and trace major actions. (This mechanism was later developed into a sophisticated, BLISS-oriented debugger called SIX12.) It required approximately four weeks of debugging to get the entire "basic" kernel operational on the PDP-10.

When the initial C.mmp configuration came up, we recompiled the kernel using BLISS/11 and brought the system over to C.mmp. The only new software necessary was machine-dependent initialization, and over half the debugging time (at this point) on C.mmp was expended on these procedures. After that, the rest moved swiftly - the entire PDP-10 kernel was transferred to C.mmp and operational in about two weeks, with debugging tools little more sophisticated than the console switches! However, we now realized a major error in our implementation method - we had no tools to aid our further debugging on C.mmp. Work then began in earnest on the construction of SIX12 for C.mmp, but kernel development was decidedly impeded until it was available (a period of about two months).

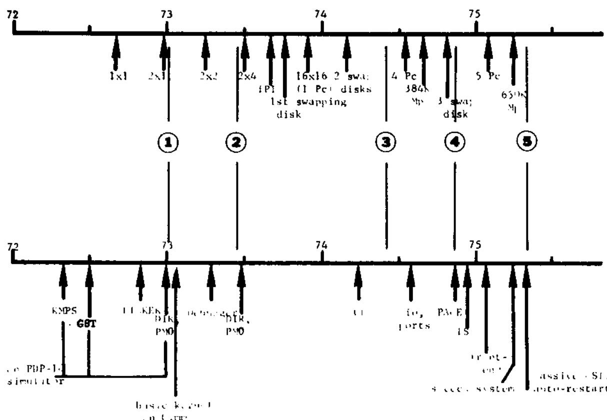

Figure IV

4.4 Productivity

Although we would like to present some conclusive data demonstrating the success of our design and implementation approaches, the figures we have are only indicators. It is a difficult task to measure the progress of a research project in quantitative terms, and a still harder one to compare it to the progress of other projects. After all, what is a reasonable measure of productivity? Perhaps the most obvious choice is rate of production of debugged code, say in instructions per man-day. This is a common metric, but it is difficult to apply fairly. We may have trouble identifying what parts of the system legitimately may be included in the instruction total, and we may be unable to obtain an accurate measure of the human effort required to produce the final product. Before presenting the productivity statistics for the Hydra project, we must examine these sources of error in more detail.

We indicated earlier that the system construction process often may (should) involve the production of software tools. These support facilities may require substantial design and programming effort by themselves, so one may legitimately ask whether such code should be included in the final product total. Plausible arguments exist supporting both sides of this question, but it is difficult to compare projects if a consistent accounting method is not used. For the figures reported below we included only the code in the end-product; support software, intermediate versions, stopgap kludges, and debugging aids were not counted.

Similar questions may be raised concerning the amount of effort expended. We chose to include all design and implementation effort until system "stabilization" (see next paragraph), including that required to build tools, intermediate versions, etc. The notion is to compute the total human effort required to produce a "finished" product and compare that ratio with other research system-building projects. This method of comparison ignores differences in projects due to methodology; only the end-result is important. Thus it is essential to establish exactly what constitutes that "finished product".

When is a system complete? Certainly, not when it first turns over; but how about when users first begin to work on it? Should we include the time until it "stabilizes", whatever that may mean? Do we keep the meter running until all bugs which significantly impact user productivity are removed? These moments in the system's history are difficult or impossible to pinpoint, and they may be widely separated in time. Thus considerable subjective judgment is involved in establishing the duration of the system construction period. For Hydra we attempted to choose a "stabilization point" identifying a time when the system was available to users and capable of performing useful work on their behalf. It still crashed relatively often and was far from stable in the sense that the term might be applied to a commercial system. Nevertheless, software construction on the initial version of the system was complete, and enough bugs had been fixed that the system was minimally useful.

We hope that we have chosen a rather conservative measure of productivity. We include the time to produce tools and other support software, but exclude these items from the finished product. We believe that this still yields a fair basis for comparison, and, in a way, provides an indication of the value of producing such "secondary" software. On the other hand, all productivity figures we have are based on estimates, and thus must be interpreted with care. With these caveats in

mind, let us turn to the data on Hydra. Our productivity at several points in the development is summarized in the following table (the "checkpoints" are noted in Figure IV):

Date	Instructions (x1000)	Man-Months
Checkpoint 1	12	28
Checkpoint 2	21	49
Checkpoint 3	34	73
Checkpoint 4	42	89
Checkpoint 5	53	115

The overall productivity at the fifth checkpoint is an average of 21 instructions per man-day. A commonly accepted "industry average" for complex systems is 5-8 instructions per man-day, although it has been significantly lower for some projects [Wol74]. In an informal survey of eight research operating systems we found an average productivity of eight instructions per man-day -- measured at roughly the same state as Hydra's checkpoint 5.

A number of explanations for the productivity of the Hydra group might be advanced; we will examine these in more detail below. However, two obvious ones are the use of a high-level language and the decomposition methodology used. We believe the latter to be the more important; in effect we believe the decomposition methodology has decreased the complexity of the task to a point where the higher productivity rate is common (see [Met71]). To support this belief let us consider the hypothesis that it is the use of the high-level language which caused the productivity increase.

It is widely believed that the number of lines of code written per man-day is independent of the language in which it is written. (Although this belief is open to challenge, let us accept it for a moment; further corroboration of the conclusion we shall draw using it will follow.) From this premise one must assume that the Bliss/11 "expansion factor", object instructions generated per line of source code, is in the range 2.6-4.5 to account for the observed productivity (i.e. 2.6x8 ~ 4.2x5 ~ 21). However, in a sample of over 43,000 lines of source code the expansion factor was observed to be less than 1.5. In other words, the observed productivity in terms of souce lines per man-day is in excess of 14. Under the assumption above this can only be accounted for by a reduction in the complexity of the task -- that is, by the decomposition methodology.

(It should be noted at this point that the low expansion factor of Bliss/11 is <u>not</u> the result of Bliss being a "low level" language. Excepting formatted i/o, which is uncommon in an operating system, Bliss programs are generally line-for-line comparable with, for example, PL/I. The low expansion rate is almost exclusively due to the optimization quality of the Bliss/11 compiler [Wul75a].)

The instructions/man-day metric is not an altogether satisfying measure of the productivity in a large software effort -- partially for the difficulties in obtaining comparable figures, but also because one suspects that the number may change during the history of the system's development, maintenance, and use. Specifically, one suspects that after the inevitable enhancement and repair cycle, the effort which will be expended per new instruction, or per incremental increase in size, will increase. Saying the same thing another way, it seems reasonable that if one computes the productivity as the ration of the total number of instructions 'in the system' to the total effort expended, as we have, this ratio should decrease with time.

Belady and Lehman [Bel71] have built a macro-economic model of this phenomenon. They derive an expression for the total work invested in a system at its ith release, which is of the form

$$W_i = w(i) + C \cdot 2^{F(i)}$$

where

w(i) is all the effort related to <u>new</u> features in the ith release, and to general overhead, except for that effort explicitly acconted for in the second term.

C is the average effort related to the correlation between simultaneous activities and/or between current ones and previous components of the system.

F(i) is a complex expression in their presentation and accounts for the number of simultaneous activities, the extent to which the total system is complete and correct, the positive effects of increasing familiarity with the system, and so on.

The complete model as presented in [Bel71] is considerably more elaborate than the expression given above. For our purposes, however, it is the form of the expression rather than its details which are important. It predicts that the effort invested is exponentially related to time. This result has both frightening implications and a certain intuitive appeal; that is, it predicts the observed behavior of many extant systems.

Figure V plots the size vs. effort at five points during the development of Hydra. Each point represents the size and effort (measured as discussed above) measured at an identifiable 'stable' point; that is, a point at which some portion of the system was complete and 'operational' in the sense that it reliably performed its intended function.

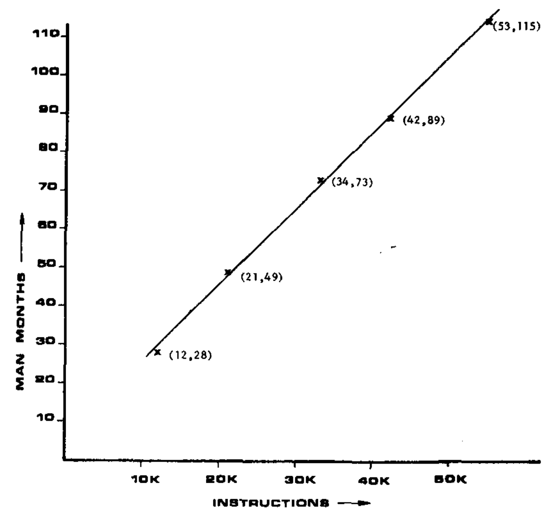

Figure V

As is obvious, the productivity rate has been approximately linear. It has not required more effort to produce the later modules than the earlier ones. This result is somewhat surprising, even to us, although we are certainly pleased by it. It seems to be at variance with the Belady-Lehman model.

Several things might explain the discrepancy between the predicted and observed behavior. First, the model may not be applicable. The model was devised to predict the effort on successive releases of a 'finished' system, while our own graph displays development effort. There may be some truth to this, but because of the style of our development - in which design, coding, debugging, documentation, redesign, recoding, and so on were carried on in parallel on separate portions of the system -- the major assumptions of the model seem to apply quite well. Specifically, for example, the period covered by the graph includes two substantial design and coding iterations on the kernel interface module.

Second, the system may simply not be large enough to exhibit the predicted behavior -- but then this simply becomes an argument for the kernel approach.

Third, the model may in fact be applicable, but the values of C and/or F(i) are small enough that the second term does not dominate. It would be nice if this were true, since it would constitute direct evidence that the methodology significantly reduces the intellectual complexity of the task.

In reality, of course, probably all these factors, and others, are responsible for the observed behavior. We don't know the relative contribution of each of the factors, but we believe the methodology to be an important influence.

The fact that most user-visible operating systems are built as user-level programs, which can be viewed as an extension of the decomposition methodology used inside the kernel, is another important factor. It is well known that the productivity rates for systems programs such as compilers are significantly higher than those for operating systems. What we have done, in effect, is to elevate more code to the user level, make it more like a compiler than like an operating system, and thus realize the correspondingly higher productivity rates.

4.5 Shortcomings

Up to this point we have dealt mostly with the successes of Hydra. It is appropriate to mention some of our shortcomings, in the hope that other system-builders will be able to profit from some of our mistakes. We can identify two classes of difficulties: hardware and software.

The most significant hardware difficulty is the small address space, a restriction imposed by the choice of processor. A machine with at least 24 address bits would provide a much more comfortable user environment than is currently possible. A segmented address space is more appropriate for Hydra. An i/o architecture carefully integrated with the segmentation scheme could eliminate much of the kernel i/o mechanism and permit some direct user control of devices. Reliability is compromised by the lack of error detection on the Unibus, and by the trusting nature of some standard peripherals. The lack of protection facilities in the processor creates awkward situations in the user program environment -- in particular, the resulting restrictions on the use of the stack page are quite unattractive. A processor designed to support multiprogramming can incorporate these protection restrictions more naturally and make many of them transparent to the user program. It can also minimize the cost of the process context-swap, which is substantial on C.mmp. Finally, a tagged architecture would eliminate much of the software checking necessary to support capabilities.

Software shortcomings are less obvious, largely because our operating experience with Hydra to date has been limited. We do know, however, that the initial user subsystems (directory, command language, terminal handling) were not given sufficient attention in the design phase. We are presently involved in a substantial re-design of these facilities. Several important issues which were consciously ignored earlier are now surfacing as major problems, including debugging of an interconnected set of processes and checkpointing a collection of processes. Some of these problems are currently under study, others remain to be tackled, but many should have been considered earlier. We can only attribute these shortcomings to lack of foresight and, in some cases, lack of personnel.

We can also identify some issues in methodology. Our failure to construct adequate tools for ourselves on some occasions during the implementation process must be considered a serious failing. In particular, the lack of a debugger probably delayed the system by several months. It can also be argued that our approach to the task of system-buildin was incorrect. We chose to iterate the design of a system component until it appeared satisfactory, then implement it. Of course, we have on occasion found it necessary to redesign and reimplement after the initial version proved inadequate -- this evolution is inevitable. However, an alternate approach to system building [New71] stresses early implementation and experimentation with the prototype. In this approach, iterations on the design frequently involve reimplementation; thus, the design rarely reaches an advanced stage without receiving "the acid test". This technique can prevent design oversights from becoming costly iterations, but it relies heavily on a flexible system-building environment with a short "code-compile-debug" loop. Such an environment was not available when the Hydra project began. Whether this approach would have produced a Hydra system without the above shortcomings we cannot say, nor can we determine if it could have done so with less total effort.

5. Conclusion

We are learning, albeit slowly, that some programs we can conceive may be so complex as to be intellectually unmanageable. Concurrently we have generated ample evidence that no operating system, however sophisticated, can be perfect for all applications. Some contemporary researchers, observing these facts, have chosen to retrench and consider only very simple systems with a single style of use in mind. Hydra, on the other hand, attempts to respond to this same situation by providing a kernel and a view of how user-specific features may be built using that kernel. The same view is used in the construction of the kernel, and there seems to be some evidence of its success in that context. The test of its success outside the kernel is still in progress. Initial subsystems for directories, files, terminal handling, a command language, scheduling, and so on have been built and have gone reasonably smoothly, but the real test lies in the future when many more people are using the system.

6. References

Bel71 Belady, L. and Lehman, M., **Programming System Dynamics**, IBM Thomas J. Watson Research Center Report RC 3546, Yorktown Heights, N. Y., July 1971.

Bri70 Brinch-Hansen, P., "The Nucleus of a Multiprogramming System", **Communications of the ACM 13**, 4 (April 1970).

Bri73 Brinch-Hansen, P., **Operating System Principles**, Prentice-Hall, Englewood Cliffs, N. J., 1973.

Bri75 Brinch-Hansen, P., "A Programming Methodology for Operating System Design", **Proceedings of the 1975 International Conference on Reliable Software**, 1975.

CJ75 Cohen, E. and Jefferson, D., "Protection in the Hydra Operating System", **Proceedings of the 5th Symposium on Operating System Principles**, Austin, Texas, Nov. 1975.

DDH74 Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., **Structured Programming**, Academic Press, New York, 1974.

Dah66 Dahl, O.-J., and Nygaard, K., "Simula - An Algol-Based Simulation Language", **Communications of the ACM 9**, 9 (September 1966).

DEC73 Digital Equipment Corporation, **PDP-11/05/10/35/40 Processor Handbook**, Maynard, Massachusetts, 1973.

Den66 Dennis, J. and E. Van Horn, "Programming Semantics for Multiprogrammed Systems", **Communications of the ACM 9**, 5 (May 1966).

Fab74 Fabry, R., "Capability-Based Addressing", **Communications of the ACM 17**, 7 (July 1974).

Gra68 Graham, R. M., "Protection in an Information Processing Utility", **Communication of the ACM 11**, 5 (May 1968).

GD72 Graham, R. and Denning, P., "Protection - Principles and Practices", **AFIPS Conference Proceedings 40**, SJCC 1972.

Hea73 Heart, F. et. al., "A New Minicomputer/Multiprocessor for the ARPA Network", **Proceedings AFIPS National Computer Conference** 42, 1973.

Hoa74 Hoare, C. A. R., "Monitors: An Operating System Structuring Concept", **Communications of the ACM 17**, 10 (October 1974).

Lam69 Lampson, B. W., "Dynamic Protection Structures", **AFIPS Conference Proceedings**, FJCC 1969.

Lev75 Levin, R., Cohen, E., Corwin, W., Pollack, F., Wulf, W., "Policy/Mechanism Separation in HYDRA", **Proceedings of the 5th Symposium on Operating System Principles**, Austin, Texas, Nov. 1975.

Lis74 Liskov, B., **A Note on CLU**, Computation Structures Group Memo 112, MIT Project MAC, Nov. 1974.

Met71 Metzelaar, P., **Cost Estimation Graph**, TRW Systems Group, Redondo Beach, Calif., April 1971.

New71 Newell, A. et. al., "The Kernel Approach to Building Software Systems", **Computer Science Research Review 1970-1971**, Carnegie-Mellon University, September 1971.

Par72a Parnas, D., "A Technique for Software Module Specification with Examples", **Communications of the ACM 15**, 5 (May 1972).

Par72b Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", **Communications of the ACM 15**, 12 (December 1972).

Par72c Parnas, D., "Information Distribution Aspects of Design Methodology", **Proceedings of the IFIP Congress 1971**, Vol. 1, 1972.

Par72d Parnas, D. L., **On the Response to Detected Errors in Hierarchically Structured Systems**, CSD Report, Carnegie-Mellon University, 1972.

Wol74 Wolverton, R., "The Cost of Developing Large Scale Software", **IEEE Transactions on Computers C-23**, 6 (June 1974).

Wul71 Wulf, W. et. al., "Bliss: A Language for Systems Programming", **Communications of the ACM 14**, 12 (December 1971).

Wul74a Wulf, W., et al., "HYDRA: The Kernel of a Multiprocessor Operating System", **Communications of the ACM 17**, 6 (1974).

Wul74b Wulf, W., **Alphard: Toward a Language to Support Structured Programs**, Carnegie-Mellon University Technical Report, 1974.

Wul75a Wulf, W. et. al., **The Design of an Optimizing Compiler**, American-Elsevier Publishing Co., New York, 1975.