1. The Family Concept

1.1 Introduction

The design, specification, implementation, documentation, and maintenance of a general purpose operating system is without question a huge project, requiring many man-years of effort. The finished product is usually just that—general purpose. Such a system can (is designed to) behave as any or all of a batch, timesharing, communications, process control system, etc., at any time. A general purpose system cannot be as efficient in any of its roles as would be a system specifically designed for one particular purpose. Unfortunately, the development cost of even a uni-purpose system usually precludes the construction of several independent such systems.

Our research has two major goals. The first is a demonstration of the feasibility of designing a system family. The idea of a family of operating systems derives from [8] and [9]. Members of a system family are developed as far as possible along common lines to avoid as much redesign and recoding as possible. The software system family concept is somewhat analogous to the hardware concept illustrated by the IBM System/360 series or the DEC PDP-11 family, although hardware families are generally oriented towards very similar user interfaces among all family members. It might be argued that a particular computer is not well suited for more than one or two types of service, and therefore does not merit the development of different systems. While this may be true for larger machines (though the manufacturers might disagree), it is definitely not true for the proliferation of minicomputers on the market, most of which have a very large range of possible applications, and tend to cost less than the systems which run on them.

Our second goal concerns the documentation and description of the family. In [3], we proposed the description of a system at various levels of abstraction. Each partial description consists of a specification, a list of decisions, an analysis, and an implementation description. The feasibility of this method is tested by applying it to the family design. An important aspect of the description is that the specification of a system facility is separated from its implementation. As a result, the implementation may be changed without affecting programs that rely solely on the specifications.

1.2 Design Methodology

The approach used in structuring the design of the family is incremental machine design, similar to that introduced by Dijkstra in the "THE" system [1] and used in [4, 5] among others. Using incremental machine design, a system is built up level by level. Each level defines a virtual machine for use by higher levels. This machine models a hardware machine closely in the facilities it provides to its users.
The various system concepts are introduced in such an order that decisions which restrict the family are postponed as far as possible. The decisions which are made are only those of specification, allowing various family members to share the specification of a level without necessarily sharing its implementation.

Figure 1 shows a possible relationship between family members. All the systems share the specification of the address space management, process management, and synchronization levels. The process control system, however, will not need to create processes dynamically, nor to provide user facilities like the other systems. Instead, it defines the special devices it will control. The batch system may use fixed memory partitions, and not need to do any swapping as the timesharing system must. A very useful result of the design should be the compatibility of such things as file systems between both the timesharing and batch systems supplied by a vendor.

1.3 Functional Hierarchy

The fact that we use the concept of partitioning a system design into levels implies nothing per se about the interaction among those levels. In particular, the hierarchical structuring is based upon functions—not processes as employed in the "THE" system. Each level is comprised of a set of functions whose names are statically known. The levels $L_0, L_1, \ldots, L_n$ are ordered such that functions defined in level $L_i$ are also known to $L_{i+1}$ (and, at the discretion of $L_{i+1}$ to $L_{i+2}$, etc.). $L_0$ corresponds to the hardware instructions of the target machine. Each level, in fact, is regarded as providing new "hardware" to the next higher level.

It is most important that the hierarchy is among functions. One of the arguments against the "THE" design is the overhead associated with interlevel communication among processes. In a functional hierarchy where functions may actually be macros, a sequence of function calls may result in a single machine instruction (or possibly none at all) when the system is compiled. It is the system design which is hierarchical, not its implementation.

1.4 Modularization and the Grain of Hierarchy

It is noted in [6] that the notions of level and module do not necessarily coincide. We wish to elaborate upon this somewhat.

Information modules [7] are comprised of some data structures (possibly) and a set of functions which share knowledge of a particular design decision, for example the details of the data structures. A level is a set of function names which are implemented via functions in lower levels. There exists no necessary relationship between the two concepts. This not only allows the division of a single level into several distinct modules, but in addition allows for the selective spanning of several levels by a single module! For example, a process manager may be implemented above the memory manager so that it can create processes dynamically. However, the memory manager may need scheduling facilities in order to satisfy allocation requests when memory is full. This apparent demonstration of the futility of the level hierarchy can be resolved by the division of a module into more than one level.

In Figure 2, the memory and process management modules are interleaved in such a way as to not violate the functional hierarchy. The two pieces of the memory manager are part of the same module because they share knowledge about how virtual memory structures are implemented (e.g. segment tables and descriptors). The use of a functional hierarchy may therefore (and in some cases does) result in a proliferation of levels which produces a finer grained hierarchy than those found in systems previously developed. As mentioned earlier, this does not necessarily add to run-time expense if macros are used.

1.5 Virtual Machine Definition

A good example of a firm boundary between levels is the boundary between a hardware machine and its programs. In this case the hardware provides a set of data registers (memory, device control words, status words, accumulators, etc.) and a set of instructions for manipulating those registers. A program written for this particular machine is considered to be at a higher level, and it may or may not use the hardware "correctly." There is no opportunity for violation of the order of the program and the hardware.

The hardware can be used in a variety of ways, some of which have been anticipated, and others which are erroneous. Examples of erroneous use of hardware are attempts to branch to an invalid address and division by zero. Nevertheless, the hardware is considered to be correct if the following types of statements hold:
Using the term "facilities" to mean the registers, in-
machine can apply to its base may be classified as:
instructions, and asynchronous activities of a given
machine. In general, a virtual machine is an incremental
modification of a lower-level machine called its base.
Using the term "facilities" to mean the registers, in-
sitions, and asynchronous activities of a given
machine, the possible modifications which a virtual
machine may attempt to be tolerant of intermittent errors or
failures in a restricted portion of the hardware).
The hardware analogy provides a prototype for
virtual machine interfaces. A virtual machine is a
programmable computer with registers, instructions
and specified actions for all improper uses of the ma-
chine. In general, a virtual machine is an incremental
modification of a lower-level machine called its base.
Using the term "facilities" to mean the registers, in-
sitions, and asynchronous activities of a given
machine, the possible modifications which a virtual
machine can apply to its base may be classified as:
(1) the hiding of a subset of the facilities, i.e. making
them unavailable to higher levels;
(2) the definition of new facilities;
(3) the systematic modification of a subset of the exist-
ing facilities.

Some of the new registers may serve the function of
trap or interrupt words for higher level programs.
(We differentiate traps, which result directly from
program actions, from interrupts, which result from
external asynchronous events.) A trap word provides
an address to which control is to be passed if the trap
or interrupt condition occurs. The side effects of a
trap are part of the specifications of the virtual
machine. Using this mechanism, the higher level program
can exercise the functions of the machine, handle er-
roneous uses (which in some cases may be desir-
able, for example, page faults), and process the results

1.5 Implementation Alternatives

The question arises as to whether the rules implied
by the level structure and the modular structure should
be checked at compile time or at run time. We decided
to check functional hierarchy at compile time and
module boundaries at run time. A justification of these
decisions follows.

A compile time check has the obvious advantage
that the check is performed only once, before execution
starts. Moreover, a compiler can optimize the code
across level or module boundaries and gain a reduction
in space and time requirements. It seems that for this
reason both level hierarchy and modular structure
should be checked at compile time. Regarding hier-
archy, we can afford to check the validity of a function
call at compile time because of the earlier design de-
cision that function names are noncomputable objects
and can specify the level at the definition site. That is,
function names behave as constants which are known at
compile time. (This decision does not preclude that the

Fig. 2. Modules and hierarchy.
parameters given in a function call, or their types, may trigger the invocation of a particular version of the function, but all these versions carry the same function name and are at the same level.) Since a level is defined as a set of function names, the rules defining hierarchy can be checked at compile time. Thus, a compiler can optimize code across level boundaries. As a result, it may be hard to find hierarchy in the compiled code, a situation not unlike nested control structures which are compiled into jump instructions.

Within modules, addresses of objects are computed at run time. This means that a program may generate an incorrect address and unintentionally modify arbitrary locations. In order to limit the damage which results from incorrect address computations, a module is associated with a collection of memory cells addressable only by the functions which belong to the module. (This corresponds very closely to the "invisible" registers accessible by arithmetic logic or microcode in hardware.) Since addresses are computed at run time, the check that a generated address is within the bounds of a module must be made at run time. Intermodular function calls require a change of environment which may or may not be expensive, depending upon the appropriateness of the hardware. However, data local to a module is completely protected from external addressing errors. Intramodule calls, since they do not require a change of environment, can be compile-time optimized even across levels.

1.7 Description and Documentation

Description and documentation form an integral part of the design task. The resulting family will not merely exist as so many lines of code, but as a document which describes modules of the systems at various abstract levels. The data associated with a module is described in terms of type definitions. A type definition describes to the user of an object the abstract states of such objects and the functions which manipulate them. The most abstract description specifies the data types used in a module. A more detailed description specifies the operations of each data type and an even more detailed description lists the programs for implementing the data types.

The description is set up to facilitate the understanding and modification of modules. It allows a programmer to get acquainted with the system family without having to go through the tedious experience of deriving meaning from the code. Moreover, the type definitions separate the implementation issues from the specification issues. It is then possible to understand part of a module without having to understand every detail of the implementation. In addition, the implementation of the abstract states of an object, or the implementation of the operations defined in a type definition, can be changed without affecting the users of the typed object provided that the specifications remain unaltered. An introduction to the use of data types as a design tool may be found in [2].

Dijkstra observed in [1] that level hierarchy facilitates the debugging process. The hierarchy makes it possible to debug the levels one by one, starting at the lowest level. The type definitions provide an additional debugging tool, since the operations on typed objects can be tested independently of their call sites. Moreover, if bugs are found, the programmer can be sure that their extent is limited to the type definition in which they occur. The strict separation of specification and implementation makes it impossible that an implementation change in one place may require additional changes in an arbitrary number of other places.

2. Family Implementation

2.1 Introduction

The three methods of transition among the various modules of a system can be summarized as:

(1) simple intramodular function calls,
(2) intermodular function calls,
(3) virtual traps and interrupts.

The concepts involved in this statement are considered to be universally basic to the family, more so than any others. Therefore, it was decided that a protected version of these facilities should comprise the lowest system level. Price [9] has shown that such a basis is sufficient to guarantee adequate protection for the system and its users. His prototype implementation established that intermodular calls would not be overly expensive on a machine with appropriate hardware of a simple nature. The implementation described in this paper is a follow-up to Price's work. The design is somewhat simplified with respect to inter-module connections. An important extension is the processing of virtual interrupts as described previously. The integration of this level with the design of higher levels has led to more extensions and a growth of confidence in the utility of the features provided.

2.2 Address Space

A module is characterized by

(1) the information for which it is responsible,
(2) the set of functions it provides to other modules for manipulating that information,
(3) the set of modules of which it knows the existence.

The instantiation (execution) of a module is characterized by

(1) the function invoked and the subset of information it needs to operate;
(2) any additional information passed to it via the parameter mechanism.

The concept of address space is introduced to implement these notions. When a module is defined, a static address space (SAS) is created for it (Figure 3).
Contained in the SAS are a segment table (ST) which represents information local to all instantiations of the module, and a function table (FT) which is a vector of information about the invocation of each of the possible module entry points. The ST is a vector of segment descriptors, each of which identifies a segment of memory, either by means of a physical address or indirectly via a reference to a segment in another SAS. The latter case allows for the sharing of segments among address spaces. Also provided in the SAS is the known address space table (KAST) which consists of a vector of SAS names.

The functions in a module have no direct access whatsoever to any of these tables, since an instance of a function runs in virtual memory, i.e. all addresses are relocated. The virtual instructions provided by the virtual memory level (VM) include ASCALL and ASRETURN, which comprise the mechanism for inter-module calls.

ASCALL takes as parameters
1. a KAST index (i),
2. an FT index (j),
3. a list of parameters in the form of ST indices (k, l, m, ...).

In effect, an intermodule call can be read as “call the ith module I know about, invoking the jth function it provides. Pass as parameters to that module the kth, lth, mth, etc., segments which I know about.”

The result of an ASCALL is the creation of a dynamic address space (DAS). A DAS consists of a working set (WS), a parameter segment table (PST), and an SAS reference (Figure 4). The WS is a vector of segment descriptors which comprise the virtual address mapping at any instant. The WS is loaded with information taken from the called address space’s FT, i.e. the code segment and a local data segment. A virtual address of the form \((w, d)\) indicates the wth WS segment with displacement \(d\). The PST is loaded with indirect segment descriptors which represent the parameter segments passed by the calling address space (access control may be restricted).

When a function executes an ASRETURN instruction, its DAS is erased and the DAS which called it is resumed after its ASCALL. During execution, a function may load and unload segments between the WS and the ST via the instructions SEGLOAD and SEGUNLOAD.

The behavior of ASCALL is illustrated in Figure 5. The calling program would like to do teletype I/O. That program’s static address space (SAS) contains a segment table which has descriptors for at least the three segments shown—a code segment, data segment, and buffer segment. The known address space table of SAS contains a reference to the teletype address space (SAS). The teletype ST contains descriptors for (at least) a code segment and data segment. The data segment might contain the device assignments currently in effect. When the program is executing, it has a dynamic address space (DAS) whose working set entries contain the ST indices of the segments currently addressible. DAS has a reference to SAS, and a parameter segment table (which is not important here). When the program executes ASCALL(kast[3], 1, st[6]) a new DAS (DAS) is created for the teletype module (kast[3] = SAS). The function to be executed is the first in the FT of SAS. Its code segment is found in the second slot in the ST. The new PST is loaded with a descriptor for the sixth segment in the SAS segment table (the buffer segment). DAS then executes the code segment, loading its data segment and the parameter buffer segment into its WS. When the called function completes and does an ASRETURN, the teletype DAS will be erased and the program DAS resumed.

2.3 Processes and Modules

Although the concept of process is unknown to the VM level, some thought must be given to the relation-
ship between processes and modules in order that the higher levels have appropriate "virtual hardware" with which to work. Roughly speaking, a process can be thought of as a flow of control which passes among various modules, some of which are user-written and some of which are part of the system. A request for I/O, for example, involves the process actually executing system code. In systems with more standard virtual memory facilities, a segment table is associated with the process control block. This means that any code executed by that process can reference only those segments in the segment table, for the life of the process (possibly with incremental changes). The code for one function can (inadvertently) destroy information important to another function. Moreover, if one function needs access to "privileged" information, any function belonging to the process has that same access. The need for protection leads to the treatment of system calls as executions for the process while it is suspended. In contrast, an intermodule call in our family causes a radical change of environment. The process performing a call on an I/O module

1. has immediate access to appropriate I/O information, and
2. cannot cause data not important to the I/O operation to be altered.

The asynchronous types of system calls which are useful (e.g. I/O operations) can be modeled as fork and join operations.

This leads to a view of processes flowing between modules in a call/return manner, and allows for more than one process to execute a given module at the same time. Any necessary synchronization is defined in the module itself. This concept turns out to be particularly "clean" in terms of context swapping.

2.4 Virtual Stack Registers

For purposes of higher levels, VM defines two virtual stack registers as part of its virtual machine. The first of these is the address space history (ASH) register, which names a segment in memory. When a function performs an ASCALL, VM saves the calling address space state at the top of the ASH stack and then pushes an entry for the called address space. The ASRETURN instruction pops the ASH stack, and resumes the execution of the address space currently at the top of the stack. Should a higher level (e.g. the dispatcher) decide to switch processes, it need only reset the value of the ASH register. The other virtual stack register is called the interrupt address space history (IASH) register and the relationship between the two registers is explained in the next section.

2.5 Virtual Interrupts

The need for virtual interrupts and traps was discussed previously. All hardware interrupts are modified by VM. Those that are important to the system are looked up in a virtual interrupt vector. Entries in this vector contain a reference to an address space and the FT index of an interrupt routine. VM stacks this address space on the IASH stack and then causes it to execute. Should other interrupts of higher priority occur during execution of the interrupt routine, the new interrupt address spaces will be pushed on the IASH stack and executed. When all of the interrupt routines return, the address space at the top of the ASH stack is resumed. Should the value of the ASH register now be different than when the first interrupt occurred, a context swap will have been effected.

2.6 Virtual Traps

The virtual trap mechanism provides a means for the "escape" of a virtual machine to higher-level software when a condition occurs which that virtual machine is not prepared to handle. There are a set of entries in the virtual interrupt vector which are available for this use. In the case of allowing context swaps, an interrupt routine can, upon completion, take one of these traps which has been previously set by the dispatching module. This enables the dispatcher to change the value of the ASH register, and then return through the proper sequence of interrupt routine completions before the new process is given control. The "trapped-to" address space is pushed on whichever stack the machine is currently processing. Hence, traps are recorded on the current stack while interrupts are always recorded on the IASH stack. No system state is ever saved on the ASH stack.

This mechanism is sufficiently general to provide to users. Each user address space then, may provide its own interrupt vector (through a virtual register accessible by the user) via which it can process the more standard faults (such as floating point errors) and also arrange for its own user-generated virtual traps to be handled.

2.7 Typed Segments

One of the problems arising from the desire to restrict data structure manipulation to a particular module is where the actual data should be kept. As an example, consider a module which implements semaphores. This module provides P- and V-operations, along with operations to create and destroy semaphores. One of the alternatives for the effect of a semaphore-create operation is that the resulting segment be kept in the environment of the semaphore module. This would imply that it reside in the module's SAS, since the DAS disappears when the operation is finished. Since all instances of the module share the same SAS, they would need to make some agreement about where these dynamically created segments go in the segment table. Worse, the segment table could easily fill up and require expanding, and billing for the segments would be difficult. Worse still, from a security viewpoint, a bug in some obscure, rarely invoked operation of the

Communications May 1976
of Volume 19
the ACM Number 5
A module could run rampant through the segment table and destroy data belonging to other users who had no intention of invoking that operation.

The other alternative is to let semaphores reside in the environment of the caller. This has good properties with respect to billing and security, but violates the reason for the existence of modules since the user can now read and write the new segments indiscriminately.

A solution involves the concept of “typed segments.” Each segment is identified by a type which is simply the name of the SAS which was responsible for its allocation. The rule is enforced that no segment may be loaded into a working set (and hence read or modified) unless its type is the same as that of the loader. In this way, users may “own” semaphore segments, but they are prevented from changing them except via operations provided by the module governing semaphores.

**Conclusion**

Our first experience with the design of a system family is favorable. We are confident that we will be able to design several differing family members. Fundamental to our design are the notions of level and module. The levels are constructed via incremental machine design. This method greatly enhances the design and debugging processes because it becomes possible to concentrate on one level at a time. The module concept leads to an unconventional ordering of the levels. Traditionally, one finds the multiprogramming and processor allocation facilities immediately above the hardware. However, since protection of modules is common to all family members whereas the processor allocation strategies may differ from one member to another, the level placed on top of the hardware is the one which implements the protected address spaces in which modules operate. Other levels which have been designed include a virtual clock level to reside immediately above VM, and the process definition level which resides above that.

The virtual memory system described has been implemented on a PDP-11/45 with segmentation feature. We argue that it is a virtual machine as we have defined, by providing a definition of the base machine and the modifications made to it by this first level.

The VM machine is the PDP-11/45 with

1. the program status word, relocation registers, segmentation status registers, register set 0, and emulate trap word hidden and therefore unavailable to the user. Also, the halt, wait, reset, and emulate instructions are no longer available.
2. new complex registers added, namely address spaces, working sets, known address space tables, the ASH, etc. Also new instructions, namely, `SEGLOAD`, `SEGUNLOAD`, `ASCALL`, `ASRETURN`, etc. are added to the instruction set.
3. all memory references by instructions systematically altered from 16-bit physical addresses to `{working set slot, displacement}` pairs. All interrupt and trap vectors are systematically altered from 16-bit physical addresses to `{address space, FT index}` pairs.

A module is described at various abstract levels so that its meaning does not have to be derived from the code. The building blocks for modules are type definitions. These allow us to separate specification from implementation issues. Type definitions provide yet another protection tool by limiting the extent of bugs. Continuation of the research effort will produce several running family members with highly nontrivial differences, including batch and timesharing systems with widely differing storage management strategies.

**References**


Communications

May 1976

of Volume 19

the ACM

Number 5