A POSTMORTEM FOR A TIME SHARING SYSTEM

BY HOWARD EWING STURGIS

CSL 74-1 JANUARY, 1974

This thesis describes a time sharing system constructed by a project at the University of California, Berkeley Campus, Computer Center. The project was of modest size, consuming about 30 man years. The resulting system was used by a number of programmers. The system was designed for a commercially available computer, the Control Data 6400 with extended core store. The system design was based on several fundamental ideas, including:

specification of the entire system as an abstract machine,
a capability based protection system,
mapped address space,
and layered implementation.

The abstract machine defined by the first implementation layer provided 8 types of abstractly defined objects and about 100 actions to manipulate them. Subsequent layers provided a few very complicated additional types. Many of the fundamental ideas served us well, particularly the concept that the system defines an abstract machine, and capability based protection. However, the attempt to provide a mapped address space using unsuitable hardware was a disaster. This thesis includes software and hardware proposals to increase the efficiency of representing an abstract machine and providing capability based protection. Also included is a description of a crash recovery consistency problem for files which reside in several levels of storage, together with a solution that we used.

XEROX

PALO ALTO RESEARCH CENTER
3180 PORTER DRIVE/PALO ALTO/CALIFORNIA 94304

ACKNOWLEDGEMENTS*

First, I thank Professor James Morris, my dissertation committee chairman, for many hours of discussions and painstaking reading of many drafts. Second, I thank the other members of my dissertation committee, Professor R. S. Fabry and Professor Martin Graham. Also, I thank all of the others who have read early drafts and commented extensively, including Dr. Butler Lampson, David Redell, Dr. James Gray and Paul McJones.

For typing many drafts I thank Janet Farness, and for the illustrations Carl Stewart and Jackie Southern. The Xerox Corporation has given me generous support while writing this dissertation.

A generous thanks is due to a faculty member who rekindled my interests, at a time when I had become discouraged with the university, Professor R. Sherman Lehman.

Finally, I thank my wife, Susan Sturgis, for her years of patience while I followed the rather tortuous path that led to this dissertation.

Howard Sturgis
Woodside California
May 1973

* While writing this dissertation, the author was employed by Xerox Corporation, Palo Alto Research Center, Palo Alto, California. The project was supported, in part, by a National Science Foundation Grant, GP 7635.

TABLE OF CONTENTS

PART ONE: ASSORTED INITIAL CONSIDERATIONS

CHAPTER 1: INTRODUCTION

In 1968 the University of California Berkeley Campus Computer Center began a project to design and implement a Time Sharing System for a Control Data Corporation (CDC) 6400 computer, with Extended Core Store (ECS). The project continued until the Fall of 1971 when it was terminated due to a lack of funds. The author was a member of the project from the beginning, and was director at its termination.

The system we designed, CAL TSS, included a number of ideas proposed by other projects, that had not yet been fully tested. These included the concept of capability based protection (system maintained pointers to system objects, through which all access to those system objects must pass), and a mapped address space (all storage resides in files, and all load and store machine instructions actually access data in some file, rather than in a local memory).

In contrast to other projects, such as Multics [C3], this was a small project. At its peak, there were about eleven people involved, many part time.

This thesis contains a discussion of some of our underlying ideas, describes the system we constructed and finally some reactions to that system. Part One describes the project, the hardware and the underlying ideas. Part Two describes the system. Finally, Part Three contains my reactions to various aspects of the system.

CHAPTER 2: THE PROJECT

In contrast to other projects, this one was quite modest. It began in the summer of 1968 with a faculty advisor and four programmers from the computer center, two of whom were half time. This rose to a maximum in 1971 of about nine programmers in the core group, with maybe five others doing peripheral tasks. Some of these programmers were still part time. This should be contrasted with Multics, involving 150 to 200 man years [C3].

In terms of machine time used, during the period of maximum system development, we had access to a 6400 for 12 hours per day during the work week, and numerous hours on weekends. About half of this machine time was used for system debugging and the other half to supply basic computing services, such as editing program files and assembling them, to the system programmers and some outside users.

CHAPTER 3: THE HARDWARE

The system was designed for and implemented on a CDC 6400, with Extended Core Store (ECS) and Central Exchange Jump [C1]. The machine had 32K 60 bit words of Central Memory (CM), and 300K 60 bit words of Extended Core Store (ECS).

The 6400 CPU has about 25 hardware registers. It can perform register to register actions in about half a microsecond, and is capable of fetching two words from memory, adding them and storing the result in about four microseconds.

ECS was that feature of the hardware which had the most direct influence on the project. This is 500K 60 bit words, which can be block transferred to or from CM. The CPU can start transfers between ECS and CM with an initial access time of about 3 microseconds, and a transfer rate of about 10 60 bit words per microsecond. A transfer can be started at any word address in ECS or CM and can be of any length, as small as one word.

The protection machinery supplied by the hardware consists of a pair of registers: a relocation register and a bounds register. One such pair is supplied for central memory and a second pair for ECS. There is no other address mapping available.

Our 6400 CPU had a special instruction, central exchange jump (CEJ) [see revision M of C1]. This instruction causes an exchange of the contents of all hardware registers with the contents of some region in CM. This requires about 3 microseconds. The changed registers include the base and bounds registers. (The same action is available on standard 6400, initiated from outside the CPU.)

The CPU has two modes, monitor and user. This mode controls the location from which the registers will be loaded during a CEJ. In monitor mode the CEJ instruction contains the absolute address of the new register contents, while in user mode the address is taken from a register loaded during the previous CEJ.

Supporting the CPU and providing access to I-O devices are ten Peripheral Processing Units (PPU's). Each PPU is a computer with a 4K 12 bit word memory and a single 18 bit register. It can pick up two 12 bit words, add them, and store the result in 9 microseconds. This time is extended to 12 microseconds if the addresses are formed by indexing, the usual case. Each PPU can access CM at about 5 microseconds per 60 bit word.

Aside from magnetic tape, the only auxiliary storage (on our machine) was provided by one half of a 6638 disk. This one half could store about seven million 60 bit words. The disk rotation time was 52 milliseconds and the maximum positioning time was 110 milliseconds. The data could be transfered to a PPU at the rate of 12 bits per microsecond.

At the time the 6400 was purchased there was no suitable hardware available to connect large numbers of individual user terminals. Therefore, the computer center designed and constructed a multiplexor capable of handling a maximum of 256 individual teletypes.

CHAPTER 4: FUNDAMENTAL IDEAS

The system design was organized around a small number of fundamental ideas:

1) Specification as an Abstract Machine

As in some programming languages, the system was conceived as implementing an abstract machine which dealt with a number of different types of abstract objects. Interaction with the system was to be accomplished through virtual instructions, which were provided in addition to the standard hardware instructions. Each of these instructions was to operate on specific types of objects, and an error was to be returned to the user if the wrong type of object was presented to such an instruction. These instructions were to be understood independently of their implementation, and to be described in terms of (possibly a sequence of) atomic changes in the state of the object.

2) Capability-Based Protection System

The authority to perform actions or to reference particular objects was to be conferred by the possession of a *capability*, which is basically an unforgeable system-produced pointer to the representation of an object, together with the type of that object and a specification of the access to be permitted. This pointer could be followed only by system code, so that the representation was directly accessible only by the system.

Capabilities are stored in special regions of memory (capability lists). Virtual instructions are available to move these capabilities from one list to another. The access granted by a capability may be reduced.

3) Processes

The single hardware machine would be divided by the system into many virtual CPU's (processes). In principle, all processes would be computing simultaneously, at some fraction of the real machine speed. Each user would have one or more processes at his disposal. With the exception of those files in the process map (see 4), a program could affect objects external to its process only through virtual instructions.

4) Mapped Address Space

Each virtual computer would not have its own central memory, to be referenced by load and store machine instructions. Instead, associated with each process would be a *map* which converts each load and store memory address into a file and address within the file. A load instruction will load a register with a word from some file, and conversely, a store instruction will place the contents of a register into a file. Thus, only one concept of data storage facility need be designed, files, rather than two, files and process memories.

5) Layered Implementation

The eventual system seen by a user program would be constructed in two or more layers. Each layer would be implemented by a program which ran as a user on the virtual machine implemented by the previous layer. In general, a layer would provide new virtual objects, not provided by previous layers, with the necessary virtual instructions to manipulate the new objects. Objects implemented by previous layer' would still be available, and would be manipulated by virtual instructions interpreted by those previous layers. In particular,

"ordinary" machine instructions would be interpreted by the real hardware. Thus, the inefficiency of interpretation would only occur on virtual instructions, and even then, only the necessary system layers would participate.

6) Distributed System Code

We envisioned that some layers of the system would be implemented by system code which resided in protected regions within each users process. This code would manipulate data global to its process. In principle, many of these system representatives could simultaneously manipulate that global data.

7) Uninterpreted I-O Devices

As far as possible, we intended to provide a user program with a direct representation of each I-O device. We intended to avoid converting I-O devices into virtual objects, such as files. Such conversion would be provided by "user" programs, many of which we would write. However, since it would be possible for users with special needs to write their own, we were released from the obligation to provide for all possible uses of a given device.

Furthermore, we wanted the possibility of emulating a device by a process. Thus, any communication with a device must be interceptable by an emulating process. This ruled out special virtual instructions for communicating with I-O devices.

Origins

Most of these ideas were suggested by previous work.

A paper by Dijkstra [D2] provided us with a glimpse of the beauty of a system described as an abstract machine. His paper also suggested the use of layered design to reduce the complexity of any single layer.

Capability based protection was described by Dennis and Van Horn [D1]. The idea of storing capabilities in special regions of memory, and providing actions to move and manipulate the capabilities was provided by the Chicago machine project [F1, F2, F3].

Multics provided the inspiration for a mapped address space, protection regions within processes (Multics rings), and distributed system code [B1, G1, S1].

Finally it should be mentioned that our audacity in beginning this project which had very modest resources compared with the Multics project, derived from the example of project Genie [L2]. This project constructed a time sharing system on an SDS 940 with 3 programmers.

CHAPTER 5: REQUIREMENTS IMPOSED BY SOME KINDS OF USER LEVEL PROGRAMS

In addition to the fundamental ideas mentioned in Chapter 4, we were guided by the requirements of various special kinds of user level programs which we felt the system should support. These special programs supply services which must eventually be provided. (One alternative would have been to construct the system so that it directly provided these services. In that case, the portions of system code which provided these services would probably have made demands on the remainder of the system which are similar to the requirements described below.)

1) Scope system simulator

A large amount of necessary software, such as assemblers, compilers and loaders already existed for SCOPE, the CDC distributed batch operating system. We needed a method for running that software, and we did not want to build it directly into our system. Therefore, we decided that it must be possible to write, as a user program on our system, a simulator for the SCOPE operating system. In order to be efficient, the simulator code should only be invoked when the user program made a call intended for the real SCOPE system. The cost of making the call on the simulator should be comparable to the cost of calling a PPU in the SCOPE system. Finally, once the simulator has been called, it must have full access to the data of the users program, similar to the access available to a PPU under the real SCOPE system.

2) Text file editor

We expected most of our users to write programs for compilers and assemblers. These programs would be written as text files, then fed to the appropriate compiler or assembler. Thus we needed facilities for conveniently writing and modifying text files. We did not want to build these text editing facilities into the system. Rather, we intended to write a text editor as a user program for the system.

The amount of data that must be immediately available to the text editor should be fairly small, a couple of small text buffers and sufficient information to tell where to write the text in an output text buffer, and where to get new text from an input buffer. These buffers should need no more than a few hundred words, and the additional words should also be at most a few hundred. All in all, the total should be less than a thousand words.

On the other hand, it was expected that the program implementing the editor would be considerably larger than this, hence there must be some way to share that program among several users. That is, there should be only one copy of the code for the editor in the main store (ECS) at one time, even though several persons were using it.

3) Debugger

It should be possible to construct a user level program (debugger) which can intimately control other user level programs, even if those other programs use sophisticated system facilities. This control should include the ability to start a user's program at a specified location, run it at full machine speed, and regain control under some specified condition. These conditions should include the detection of an instruction error on the part of the user program, or a command from the user at a teletype. Once the debugger has regained control, it should be able to inspect the internal state of the user's program in detail, and make modifications.

4) Device drivers

One of the functions generally performed by an operating system is to convert an ugly I-O device into a more tractable virtual device. A typical example is to convert a line printer, with its many special functions, into a write only sequential text file. A problem with this approach is that some of the flexibility of the device may be lost.

As far as possible, we wished to avoid making such conversions in the basic system. In fact, we desired to permit a user to make his own conversion, if he wished. Thus, our system would provide an interface to each I-O device which gave a using program direct control over that device. For example, all of the special functions of the printer would be directly available. Finally, we would supply a user level program which converted the I-O device into a generally useful virtual device, but would not attempt to handle all conceivable user desires.

5) Typical user Fortran programs

We felt that the facilities necessary to support the kinds of programs mentioned above would certainly be sufficient to support an ordinary Fortran program.

PART TWO: THE SYSTEM

CHAPTER 6: BASIC ARCHITECTURAL CONSIDERATIONS

The architectural design of the system was a multistep process. We first formed a global picture of how user programs should run, then gave a general division of the necessary functions into levels. Finally we embarked on the detailed design of the various levels. This chapter describes the issues which motivated our global design.

Swapping

Our general picture of user programs included two types, the interactive ones and the noninteractive ones. The interactive programs would reside in ECS and from time to time be swapped into central memory and allowed to execute. (See Chapter 3, Hardware.) The noninteractive programs would spend most of their time on the disk. From time to time they would swap into ECS and while in ECS behave like interactive programs.

If we were able to swap a program at the full ECS transfer rate of 10 words per microsecond, then a 10K word program could be swapped in and out in two milliseconds, allowing the use of a 20 millisecond quantum with 90% efficiency (1K = 1000 base 10). A larger program would need a larger quantum to obtain the same efficiency, but even a 32K word program would need no more than a 64 millisecond quantum. Even if we assumed some degradation in efficiency of the swap, a maximum quantum size of 100 milliseconds would be sufficient. If there were 100 interactive programs, and 10% were waiting for service at one time, each would begin to receive service within one second. This seemed satisfactory.

Input-output strategy

User programs that needed access to amounts of data larger than the amount that could be swapped into central memory would use some form of access to data stored in ECS. (Eventually called ECS files, see next chapter.) This data stored in ECS could then be used as an I-O buffer. Programs which desired access to this data would read it while they were running in central memory.

Data from an input-output device would first go to a PPU. The PPU would transfer the data to a central memory buffer and start a system program in central memory, which would transfer the data to ECS. This system program would then inform the appropriate requesting program that the data was available. (The requesting program was informed by sending an event on an event channel, see next chapter.) Movement in the opposite direction was similar.

We made rough estimates of the amount of CPU time that would be used by the system programs to move the data to and from ECS. We assumed that the main I-O load would come from the disk. If we assumed that two PPU's were alternately reading the disk and then writing into central memory, central would receive words at the maximum rate of one word every five microseconds. If enough were buffered so that the main cost to the central program was the actual transfer to ECS, the central program would use one microsecond every 50. Thus, we felt this time overhead was acceptable.

We gave less thought to the space overhead. However, at 512 words per PPU (the maximum power-of-2 buffer that a PPU could hold), there would be at most 5120 words of central. On a 64K machine (which we assumed would be available for such a large system) this is less than 10% of the total space, which we thought would be acceptable. If this was too much, we could arrange for the PPU's to transfer a 512 word buffer in several sections, with a separate request to central memory system program for each one. This would reduce the necessary buffer space in central memory at an increased cost for CPU time, due to overhead in responding to each

request.

As an alternative to this design with central memory buffers, CDC offered an optional hardware feature which permitted direct exchange of data between a PPU and ECS [C2]. We referred to this feature as a "Back-door". Unfortunately, the rate of this transfer was limited, by the PPU, to one 60 bit word every 5 microseconds, and while in progress degraded the transfer rate between central memory and ECS. We were unable to determine from CDC how great the degradation would be, but there were hints it might be a factor of two. Since our estimates of program swapping overhead were heavily dependent on a high transfer rate between central memory and ECS, and the estimates for the PPU-CM-ECS scheme indicated a low overhead, we decided to avoid the Back door.

Layers

The major factor which contributed to our allocation of responsibilities within the layers was a desire to produce an interim system as soon as possible. We wanted a system which we could use to support further development, i.e., provide editing and storage facilities for the programs we were writing. Also, we felt that such a system would demonstrate to the administration that we could eventually construct a final working system. (In fact, we did demonstrate a two teletype system at the end of the first year and moved our program development under the system six months later.) To this end, all facilities which we felt were unnecessary for an interim system were pushed to higher layers.

The first layer would support the interactive programs that reside in ECS. We called this the ECS system. All data storage would reside in ECS, while access to the disk would be provided similar to that for any other I-O device. It was intended that the addition of a temporary executive would produce our interim system. This temporary executive (called the Bead) would ignore all protection problems.

The second layer would introduce disk files and directories. It would convert the real disk into a collection of virtual disk files. We called it the disk/directory system. It would also provide disk swapping of non-interactive programs.

The third layer was to be the executive. This executive was to provide password protection for identification of users, charge accounting and convenient commands for starting programs.

CHAPTER 7: ECS SYSTEM ARCHITECTURE

The ECS system was the first layer in CAL TSS. It was a complete, but small, time sharing system. It provided a small amount of file storage. interprocess synchronizing facilities and access to physical I-O devices. (A description of the ECS system, written during an early period of the project, appears as [L1]. A more detailed description appears in [C5,C6].)

Brief sketch

A program running on the ECS system ran inside a *subprocess* (protection domain) of a *process*. Through calls on the system, it could store data in *files*, send signals to programs in other processes via *event channels* and call programs in other subprocesses of the same process.

Each subprocess had a *C-list*; a list of system provided, unforgeable, pointers to various objects, such as files and event channels. This C-list defined the set of objects which a program in this subprocess could access.

Each *physical I-O device* was represented by a set of files and event channels. A program could exchange data with the device through the files, and exchange control signals through the event channels.

The abstract machine

The abstract machine implemented by the ECS system consisted of the following 8 types of objects, and about 100 actions which could be performed on them:

ECS files
Event channels
Processes
C-lists
Capability-creating-authorizations
Operations
Class codes
Allocation blocks

The state of each existing object was represented in some region, or regions, of ECS.

The following will describe each type of object, and the kinds of actions available. Chapter 8 will give a more detailed description of the representation.

ECS files

Files were the objects designed to hold data. Basically, a file was just a sequence of 60 bit words. Facilities were provided for reading or writing any consecutive subsequence of these words. In fact, files were more complicated.

The complications were caused by the facilities we expected to provide in the disk system. There we wanted files which were stored on the disk, and which could be read or written as described above. Also, we wanted a user program to be able to *attach* a portion of a disk file, forcing that portion of the file to be brought to ECS and held there. Then, when a disk file read or write was directed to that portion of the file, the system could automatically read or write the copy in ECS.

As a means of implementing this disk system facility we proposed that an ECS file be used to hold that portion of the disk file which was in ECS. In order to make the interpretation of the disk file read or write request easier, we wanted to have the attached portion of the disk file in corresponding addresses of the ECS file. Hence we would have ECS files which would be holding small portions of very large disk files, but which themselves would have to be small. Thus the ECS files would have to have many addresses that existed and many that did not.

In order to provide this facility we divided the ECS file into blocks, each block of equal size and holding words with consecutive file addresses. Furthermore, since we felt that only a small number of blocks out of a possibly very large number would exist, we represented the file as a tree. At the head of the tree would be a block, of arbitrary but fixed size, containing pointers to the next level of blocks. After a number of levels of pointer blocks, wherein all blocks of the same level would be of the same power of two, would finally come a level of data blocks. A pointer block would only exist if one of its descendant data blocks existed.

Actions were available to create and destroy data blocks. The appropriate pointer blocks would be automatically created and destroyed, as descendant data blocks were created and destroyed. Thus the amount of ECS space required to represent a given file would vary with time.

At the time of creation of the file, a *shape* must be given. This shape specifies the number of levels of pointer blocks desired, the sizes of the pointer blocks at each level as well as the size of the data blocks. In order to permit rapid computation of the ECS address of actual data blocks, we required that the block sizes at all levels, except the head, be some power of two. Finally, it was possible to create a file with no pointer blocks, but in which the head block was itself a data block, of arbitrary size. (Figure 1 contains an example of an ECS file.)

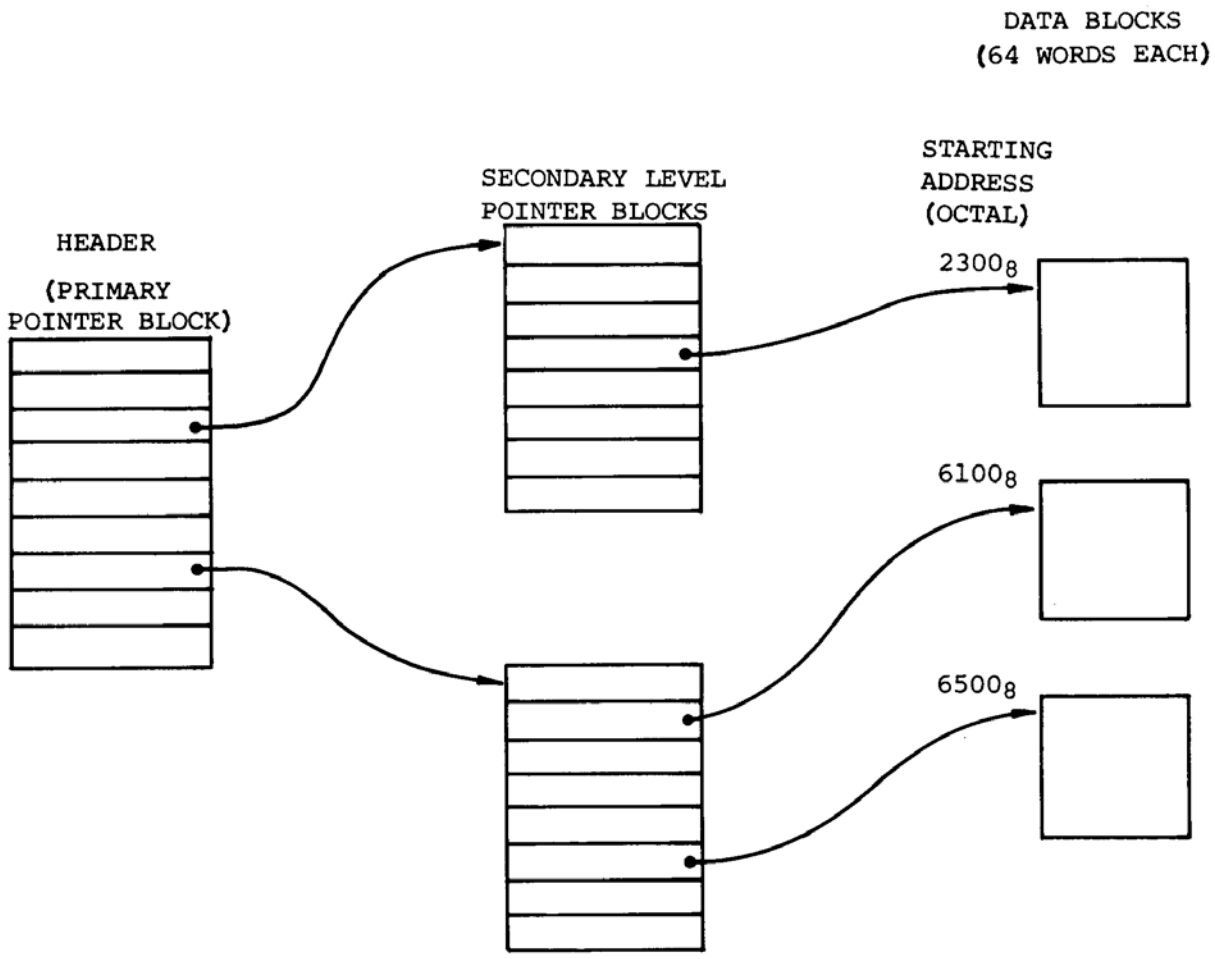

Figure 1. Example of an ECS file, with three existing data blocks and a maximum possible size of 4608 words.

Event channels

Event channels provided the means for synchronizing programs in separate processes, or a program and an I-O device. They are a generalization of the semaphores of Dijkstra [D2].

One primitive which might have been used for synchronization is a test-and-set instruction. This has the deficiency that further primitives are needed to permit a process to block when attempting to access a locked data base. Also, there must be some procedure available for a process which unlocks the data base to discover what process, if any, should be awakened. Finally, we had no hardware test-and-set instruction.

The semaphores of Dijkstra provide sufficient facilities to interlock a single data base [D2]. However, we felt that a major means of communication between distinct processes, or between a process and an I-O device, would be through a sequence of discrete messages. For example, a sequence of buffer loads could represent successive lines to be printed by a printer.

Thus, we designed an event channel to communicate a stream of 60 bit data items. These items could be the indices of buffers which carried a more voluminous message.

The basic actions available for event channels included: create, send-an-event, get-an-event, and destroy. The get-an-event action had four versions:

A) get an event from an event channel,
if no event waiting,
A1) return and so indicate, or
A2) block until one is available;

B) get an event from one of several event channels,
if no event waiting on any of them,
B1) return and so indicate
B2) block until one is available on one of the channels.

One major problem with event channels was a restriction we imposed: the waiting events must be recorded in a fixed region for each event channel. (The size of this region was specified when the event channel was created.) This imposed a maximum limit on the number of waiting events an event channel could hold. After this point, an attempt to send an event to a full event channel returned with a refusal.

This problem was helped by providing that when the last possible event was sent to an event channel, it was automatically converted to a special one, and the sender informed. Then, a program receiving one of these special events had to communicate with possible senders, to straighten things out. In general, this was moderately difficult.

There was no limit on the number of processes which could be blocked, waiting for an event to arrive at a given event channel.

Processes

Processes were the active elements in the system; they contained executing programs. Processes were composed of *subprocesses* (protection domains, performing a function similar to rings in Multics). One subprocess within the process was designated as the *root*. All other subprocesses within the process had a father, forming a *subprocess tree*. Each subprocess had a name (*class code*, see later), a local C-list and a map. The map translated a logical memory address into a file address.

Maps

In systems like Multics [D1] and TENEX [T1], the implementation of a map is assisted by hardware. We had no such hardware. The map in CAL TSS actually consisted of swapping directives. When a subprocess was to run, it was copied from ECS to CM. Each swapping directive in the map specified some portion of some file (in ECS) to be copied into some part of CM. After running, each portion would be copied back to its ECS file.

It should be understood that the maps in CAL TSS did not give the same facility as maps in other systems. For example, if the same portion of a file was mapped into different regions of CM, and the running program modified one of these CM regions, the modification was not immediately reflected in the other region, in contrast to systems like Multics or TENEX. In CAL TSS, the change would eventually appear after the subprocess was swapped out to ECS and back in to CM. In general, this was a fairly unpredictable occurrence.

The problem was more severe than might be apparent. It is plausible to assume that a single subprocess would map a portion of a file into at most one region of CM. However, two subprocesses might independently map the same portion of a file. As we will indicate later,

both subprocesses could be swapped into CM simultaneously. This would lead to precisely the problem described above. In fact, the disk-system (see Chapters 10 and 11) was composed of two subprocesses, and both mapped a common region of a scratch file. This indeed led to problems. The fix used was to explicitly read and write the common portion of the file, thus losing the automatic benefits of the map.

Subprocess call stack

At most one subprocess of a process could be in execution at a time. A facility was available for code in one subprocess to *call* a fixed entry point in another subprocess. This would create an entry on the call stack to facilitate a *return*. The called subprocess would be swapped into CM if necessary, and begin execution at a predetermined location. The call action would also transfer some data items from the calling subprocess to the called subprocess, and transfer some capabilities between their local C-lists. (More on this under *operations*.) Saving and restoring the hardware registers was up to the called subprocess. ECS system actions were available to save the registers in a specified location, as well as restore them. They were not automatically saved during the call action.

Subprocess tree

In order to facilitate the construction of debugger subprocesses, and service subprocesses such as the SCOPE simulator and the disk/directory system, a control relation was defined among the subprocesses of a process. This took the form of a tree, which we called the subprocess tree. An ancestor subprocess had complete control over all descendents.

Under certain conditions, a subprocess and a descendant could be swapped into CM simultaneously. A program running in the ancestor subprocess would then have direct access to the logical memory of the descendant. The memory of the descendant would appear at high addresses within the memory of the ancestor. Moreover, a similar relationship held for the local C-lists of the two subprocesses, as well as the maps. (The local C-list of the second appeared as an extension fo the local C-list of the ancestor.)

At all times, a certain set of subprocesses within the tree, the *full-path*, was defined. This determined which subprocesses were to be swapped into CM. The Full-path had a fairly complicated definition, which guaranteed that if a subprocess called an ancestor, both would be in the current full-path, as well as any intermediate subprocesses in the tree.

The subprocess tree was used to determine how to process *errors* and *interrupts*. Certain conditions (such as an illegal parameter) occurring during an action created an error. An error always caused some subprocess to be called, which would begin execution at a special error entry point. The subprocess to be called was determined during a scan of the tree, starting with the subprocess in execution at the time of the error, and proceeding towards the root. Each subprocess had an *error-mask* which specified which kinds of errors it wished to process. Upon calling a particular subprocess with an error, the corresponding bit in its error mask would be turned off. In order to receive similar errors later, the subprocess was required to turn the bit back on.

A program in one process could direct an *interrupt* at a named subprocess in another process. (It named the subprocess by giving its class-code.) Information was associated with each subprocess to determine if it would accept interrupts. A scan, starting at the named subprocess and proceeding towards the root, determined which subprocess would actually receive the interrupt. That subprocess was then called as soon as it or one of its descendents was executing.

The ancestor relationship had a number of intended applications. These included the construction of debugger type subprocesses and the scope system simulator. The Disk system intended to make use of the facility to perform reads and writes for portions of a file residing on the disk.

The algorithms for handling errors and interrupts attempted to prevent a descendent subprocess from unexpectedly getting control over an ancestor. Only an ancestor of a subprocess could unexpectedly gain control. We saw this as a generalization of the usual monitor-user mode facility.

Actions involving processes

The basic actions of creating a new process, and destroying a process were available. Creating a new process required a description of its root subprocess, which included its CM field length, C-list, map, class code and the relative location of its entry point within its central memory field length.

One other action directly affected a remote process, sending an interrupt to a particular subprocess within that process. Other actions on processes directly affected the process in which the program which called the action was running. These actions included the creation of a new subprocess, modification of map entries and destruction of named subprocesses. The creation of a new subprocess required the naming of its father, its class code, its local C-list, the specification of the initial contents of its map, its central memory field length and the location of its entry point within its central memory field length.

C-lists (and capabilities)

A C-list was a finite sequence of *capabilities*. A capability was a system maintained, unforgeable, authorization. Many capabilities contained pointers to the representations of system maintained objects, such as files and event channels, and authorized some actions to be performed on those objects.

A capability contained three components:

a *type*,
a set of *option bits*,
and a *value*.

In the case of capabilities which contained pointers to system maintained objects, the *type* identified the type of the object, the *option-bits* defined actions authorized through this capability and the *value* was a pointer to the object representation. For capabilities which did not contain pointers to system maintained objects, the type and option-bit components performed functions similar to those components in pointer capabilities.

In order to perform a system action, a program presented indices to one or more capabilities within its subprocess's local C-list. These capabilities, in turn, defined the action to be performed and the objects on which to perform it. Before performing the action, the capabilities presented were checked for proper type and suitable option-bits. (For more details, see *operations*.)

Available actions provided facilities for storing capabilities in C-lists other than in the subprocess's local C-list. These actions permitted copying capabilities between other C-lists and the local C-list. They also permitted an indirect specification of a capability to be used in an action: two indices would be given, the first within the local C-list to name a remote C-list,

and the second to specify a capability within the remote C-list. Other actions permitted the construction and destruction of C-list.

Capability-creating-authorization

These provided a user program with the ability to create private capabilities, with a type different from the system provided types and from other private types. Each capability-creating-authorization capability specified a type which newly created capabilities would contain. The following three actions provided the facility:

i) create a new capability-creating-authorization.
produces a capability for a capability-creating-authorization, with a specified type never before seen.

ii) create a new capability.
requires two parameters:

a) a capability-creating-authori-zation
b) a 60 bit datum

produces a capability with all option bits on, with type as specified in the capability-creating-authorization, and with the 60 bit datum as value.

iii) read a capability.
produces two words of data, containing the type, option bits and numerical value of the value part.

Using these facilities, a "user" written subsystem could construct unforgeable pointers of its own. So long as it never permitted unfriendly programs access to its capability-creating-authorization, it would know that only friendly programs created capabilities of its own type. Thus, the value of such a capability could be trusted. This value might, for example, have been the disk address of a header for a disk file. Furthermore, programs which used such a subsystem would have available the protection facility of the basic system. For example, these programs could store disk file capabilities in C-lists, and pass disk file capabilities with reduced option bits to untrusted subsystems.

Operations

Viewed as a virtual computer, the ECS system had only one virtual instruction. This instruction accepted a list of parameters, the first of which was interpreted as a pointer to an *operation*. A basic operation contained two parts; a specification of the actual action to be performed and a list of specifications for the parameters to that action.

Two kinds of actions could be specified by an operation: a built in ECS system action, or a call on a named subprocess. The possible parameter specifications included:

datum,
capability of given type with certain option bits,
fixed datum,
fixed capability,
block capability and
block datum.

(The last two were only used for subprocess calls.)

The fixed datum and capability specifications carried a value for that parameter in the operation itself. The user calling such an operation never saw these parameters. One intended use for fixed parameters, particularly fixed datum, was to distinguish between different kinds of calls on a single subprocess. In general, the fixed parameters allow projection of an operation.

During either a built in ECS action, or for a call on a subprocess, all capability parameters were automatically checked for correct type, and at least the specified option bits. If the check failed, an immediate error was generated. For a call on a subprocess, all parameters were then copied into the address space of the called subprocess, the data into its memory and the capabilities into its local C-list.

An immediate consequence of specifying an action by pointing to an object was the ability to control what actions were available to each subprocess. This was a generalization of one aspect of the usual monitor-user mode facility on actual computers, that of a restricted instruction set under user mode.

F-returns (failure-return)

An ECS action, or a subprocess, could return with an *F-return*, distinct from an error. In particular, the ECS system returned with F-return when an attempt was made to reference data in a non-existent portion of a file. This F-return was processed in either of two ways. The simplest was to reflect it to a program jump in the calling program.

Multi-level operations

The more sophisticated use of F-returns was to make the original call with a *multi level operation*. This was an operation with several possible actions. If the first returned with F-return, the second was automatically tried.

The major user of this feature was to be the Disk system. It would provide a two level operation, in which the first level was an ECS file read or write, and the second was a call on the Disk system subprocess. Thus, a user could attempt to read the ECS version of a disk file with this operation. If the desired portion of the file was in ECS, the action would proceed exactly as if it were an ordinary ECS action (i.e., fast). If not, the ECS system would F-return, and an automatic call would be made on the Disk system to handle the situation. This would be an inherently slow subprocess call. Thus, an initial expensive call on the Disk system was avoided if the data were actually in ECS, but it all appeared as one simple action to the user.

Actions on operations

Only operations which made subprocess calls could be constructed by user programs. The operations which contained ECS actions existed at the beginning of the world. Actions existed for constructing new operations which called given subprocesses with desired parameter specifications. Actions also existed for constructing a new operation built from an old one by adding a new level.

Class-codes

Class-codes were subprocess names. They were used in constructing operations and in sending interrupts to other processes.

As we conceived the system, each user process would contain a representative from each of several classes of subprocesses (e.g., each user process would contain a representative of the

disk/directory system). We wished to construct operations which would, for all processes, name a representative of the same class of subprocesses (e.g., the representative of the disk/directory system). One possibility would have been to name a subprocess by its position in the subprocess tree. This would have been undesirable for three reasons:

1) It was necessary to prevent an arbitrary program from constructing an operation that called an arbitrary subprocess with arbitrary parameters. Hence a subprocess name to be used in constructing an operation must be protected.

2) It was necessary to prevent arbitrary programs from sending interrupts to arbitrary subprocesses in other processes, again the subprocess name must be protected.

3) It was conceivable that in different processes, subprocesses designed to be called by a given operation might appear at different positions in the tree, or might not exist at all.

A capability for a class code did not contain a pointer, but contained a representation of the name as its value part.

Actions were available to create new class codes, to construct subprocesses at specified points in the tree (specified by class code of father), to construct operations which called subprocesses of specified class code and to send interrupts to subprocesses of specified class code.

Allocation blocks

As we shall see below, the representation of each ECS object occupied space in ECS. It was necessary to ration this space among prospective users. Since we expected that a single user might be associated with more than one process, we decided to ration ECS space through a more general entity, an allocation block.

Each ECS object, including allocation blocks, had to belong to some allocation block. (A special root allocation block was exempted from this requirement.) All actions that created objects required an allocation block as one parameter. This allocation block was checked for sufficient free space, which was then allocated to the new object. For objects which could change size (files and processes) space was moved to or from their allocation block.

In addition to rationing the use of ECS space, allocation blocks metered the ECS space used by objects. This meter recorded the time integral of the ECS space used by objects belonging to the block.

Moreover, other resources were to be rationed and metered by allocation blocks. These were to include MOT slots (see chapter 8) and CPU time. In fact, meters for both were installed, but rationing was never installed for CPU time. (Also, the eventual accounting system read the meters for ECS space-time and CPU time, but not MOT slots.)

New allocation blocks could be created, and space could be moved about the tree of allocation blocks. A special action was implemented, to be used only by the disk system, which could move space from one allocation block to another, while the space continued to be included in the space time integral of the first block. This permitted the disk system to borrow space from a user and have the user continue to pay for it.

CHAPTER 8: STATE REPRESENTATION IN THE ECS SYSTEM

A general storage allocator provided arbitrary sized blocks of contiguous space in ECS With the exception of files, each ECS system object was represented in a single block. An *MOT* (Master-Object-Table) provided the absolute ECS addresses of the representation of each ECS object. A capability for an object pointed to the object by giving its MOT index.

Storage allocator

A general storage allocator was written which provided arbitrary sized blocks of contiguous space in ECS. If no free block of contiguous storage was large enough to satisfy a request, and the total amount of free space was sufficient, a compactor moved all in use blocks to one end of ECS, and thus combined together all free blocks.

The compactor was made possible by two conventions:

i) For each block in use, there would be exactly one primary pointer which contained its absolute ECS address, and the block itself contained the absolute ECS address of this primary pointer

ii) Secondary pointers which contained an absolute address could exist, but they would have to be accompanied by:

   a) Sufficient information to recompute the absolute address through a succession of primary pointers,

   b) The count of compactions which had occurred up to the time of computing the absolute address.

Thus, during compaction, the primary pointers could be found and updated. Also, before following a secondary pointer, its associated compaction count was compared with the actual compaction count. If they disagreed, the pointer was recomputed.

Unique name

Each object, when created, was assigned a never before seen unique name. (We had enough unique names to conservatively last several years of continuous operation. These names were re-assigned after each dead start.)

Master object table (MOT)

When an object was created, it was assigned a free entry in the MOT. This entry contained the unique name of the object and the primary absolute pointer to the representation of the object. When an object was destroyed, the unique name in the MOT entry for the object was replaced by the unique name to be used by a new object with the same MOT index.

The value part of a capability for an object contained the index of the MOT entry for the object, and the unique name of the object. Whenever the absolute address of the representation of an object was to be obtained from a capability, the unique name in the capability was checked against the unique name in the MOT entry. Thus, there was no need to find and invalidate all capabilities for an object that was destroyed, as subsequent unique name checks would fail.

Capability list

A C-list had the simplest representation: a single block containing 2 word entries for each capability. Figure 2 sketches a C-list with one capability for a second C-list, and another capability for a destroyed C-list.

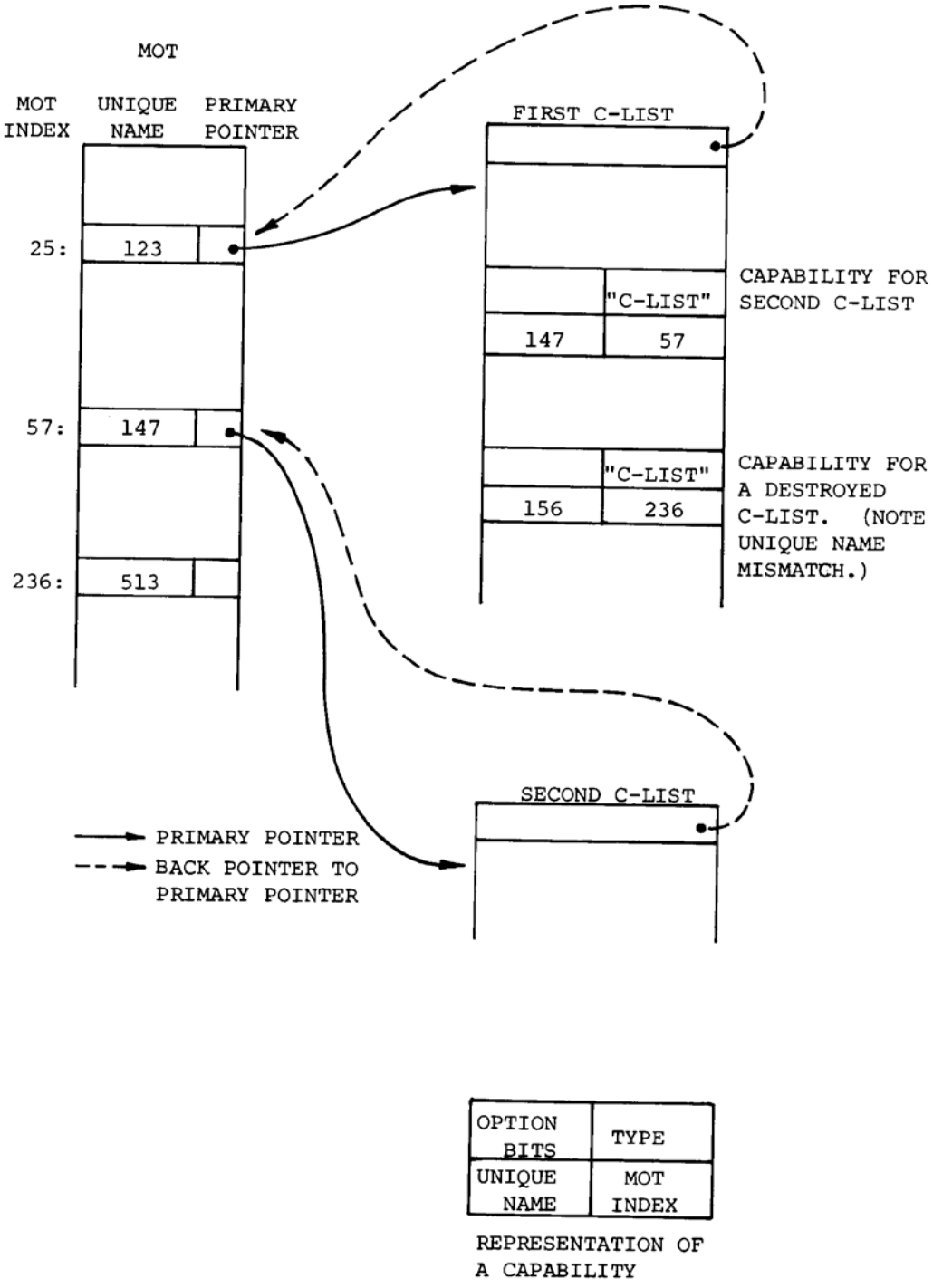

Figure 2. Example of MOT with 2 C-lists.

Files

Files were represented by a number of distinct blocks, one for each existing data or pointer block. Pointer blocks contained primary absolute pointers to descendant blocks.

Event channels

Event channels were represented in single blocks. These blocks contained a fixed size buffer area for storing events that had been sent, but not received. They also contained the head of a circular doubly linked chain of processes which were blocked waiting for events. This chain was held together by MOT-index-unique-name style pointers, rather than absolute addresses. A single process could be on more than one such chain if it was waiting for an event from several event channels. There was an area in the representation of each process that contained these chain pointers. (In general, events could be waiting for processes, or processes could be waiting for events, but never both on the same event channel.)

Other types of objects

Among other objects, processes had a very complex and not very interesting structure, while most other kinds of objects were fairly straightforward. One item of interest was the representation of subprocess maps. These were maintained in two forms, logical and compiled. The logical form specified, for each entry, a file (through MOT-index-unique-name pointer), file address, logical memory address, count and read only flag. The compiled version converted these entries to absolute ECS and CM addresses. Associated with the compiled version was a compaction count. Thus, the compiled version constituted a secondary form of absolute pointers.

CHAPTER 9: ECS SYSTEM I-O FACILITIES

The major consideration in the design of the I-O system was having the code in the heart of the system as simple as possible. There were a number of reasons for this, but the paramount one was to allow as much of the system as possible to continue running, even in the face of an error in some infrequently used or special purpose part. For example, we wanted to be able to add code to drive special purpose I-O devices without a long check out period. If most of the code for the device had to be within the system, then a large effort would be required by system programmers, whereas if most of the code could be part of a user's process, then the system programmers could be responsible for only a small interface program in the system.

Another consideration was experience derived from the SCOPE operating system which indicated that PPUs were very feeble machines. The SCOPE operating system depended on having large amounts of special purpose code in the PPUs, and significant problems developed trying to fit the code into the memories of the PPU's. Hence we felt that the more complicated device code should be in central.

The general character of the I-O system was controlled by earlier decisions, and the hardware. There had to be some sort of device drivers in the PPUs. These would transfer data to and from buffers in central memory. As necessary they would signal special code in the ECS system which would transfer data to and from buffers in ECS. These ECS buffers would exist in ECS files, accessable to programs in user processes. Finally, as necessary, this special code would send and receive signals to and from user processes via event channels.

Under the SCOPE operating system, PPU's were assigned to individual user programs for an I-O task. Each time such an assignment was done, the appropriate program had to be loaded into the PPU from some storage medium, usually the disk. This system had a number of drawbacks, the worst being the time required to load the program into the PPU. As one watched the system run, the same PPU program could be seen to move from PPU to PPU. It seemed that much more service could have been given if the program remained in one PPU and serviced several users. However, as the system designers had chosen to put a large amount of code in the PPU's, there was far more code than could fit in the 10 PPU's available. Thus they were essentially forced to swap programs into the PPUs.

Aside from the overhead of swapping the PPU code, there was a second reason for avoiding the style used in the SCOPE system. We expected to have a large number of active processes, all frequently doing I-O. In the SCOPE system there were only 7 Control Points (or processes). Thus 10 PPU's could service these 7 processes moderately well, but would have trouble with more processes.

We decided that the code in the PPUs should be simple, so that it could be resident at all times. As a hedge we held open the possibility of having one or two PPUs hold transient code, but this never proved necessary. Since the code in the PPUs had to be simple, and since we wanted to make all the potential uses of a particular device available to users, the interface between central and the PPUs had to be a logically complete description of the device. We did not want to convert the device to some virtual device, for example we did not want to implement the idea of files on magnetic tape within the PPU. Attempts to do this sort of thing in the SCOPE system had led to large amounts of code to implement the logical constructs of files, and left some of the features of tapes unavailable. For example, under the SCOPE system, it would be impossible for a central program to write "in-place" on a tape, whereas by direct control of the tape drive it is possible.

This same argument can be used to show that the same sort of interface must be preserved

between the central code and the user processes. We wanted to keep the amount of code within the system at a minimum, and attempts to implement the idea of files at that level could lead to the same problems.

Thus the user process was to be presented with an interface that presented the full logical facilities of the device. In fact these ideas had to be modified in the face of timing constraints. It would be impossible, for example, to allow a user process to decide exactly what to write on the disk at each sector position, as that sector position came by. The reaction time of a process would be too large. Consequently, some logically equivalent interface would have to be found, if possible.

The following are short descriptions of how we handled some I-O devices.

Teletype I-O interface

The multiplexor interface was designed to present each TTY as a separate device. The TTY was run in full duplex mode, with an echo for each typed character. This permitted a visual check that the character had actually been received by the computer. Wherever possible the echo was done by the PPU itself. Logically, this echo should be provided by the receiving process when it receives the character. The process can then do special purpose functions, such as echoing a different character than it received, or not echoing at all. These functions are useful when interpreting non printing control characters as special signals, and when receiving passwords. Also, the echoes can appear at appropriate places in the output of the process.

It is desirable, if possible, to echo a character immediately after it is typed, since unexpected delays in the echoes are unnerving to a user at a teletype. Unfortunately, it is very expensive to permit a user process to echo each character as it is typed.

The PPU had tables of 'break' characters, one table associated with each TTY. If a character arrived from the TTY that was not a break character it would be automatically echoed. If it was a break character, it would not be echoed, subsequent characters also would not, and a signal would be sent to the process involved. Furthermore, if a character arrived from a TTY while ordinary output was in progress on that TTY, the echo would be prevented and a signal for the first such character sent to the user process. Thus characters typed during output could be echoed by the receiving process at the time it actually received them.

As characters were received they would be packed into one word buffers held in central memory, and a central action would only be required when the one word buffer filled, or when a break character arrived. Thus we attempted to hand the full duplex facility of a TTY to a user process, and still keep the number of interactions low. Except under unusual circumstances, interactions with a central program would occur only once per one word buffer. Thus the number of interactions was reduced by a factor of five (the number of raw teletype characters which could be held in one central memory word). The central program itself transferred the words to and from a buffer in an ECS file, and only interacted with the user process when that buffer was full or empty, another reduction. (Of course, there had to be interactions for break characters etc.)

Magnetic tape I-O interface

A reduction in interaction rate was also attained for magnetic tapes, but in a different manner. A single user request on the magnetic tape system was permitted for a series of similar read or writes. The read or writes would proceed sequentially into or out of a buffer in an ECS file, and would return a single response to the user process at the end of the sequence. The sequence would be terminated early if an error occurred, such as parity or end of file. The response

would indicate the reason for termination and the number of reads or writes successfully completed.

Disk I-O package

The ECS system disk I-O package was divided into the usual two pieces, one consisting of PPU code and the other in central memory. It provided the ability to read or write records starting at any sector boundary and of lengths 64, 129, 257, or 513 60-bit words. Automatically included in each record, invisible to the user, was the disk address of the start of the record. This acted as a check on the disk positioning mechanism. Also provided were buffers of these same sizes in an ECS file. Event channels were provided to hold lists of available buffers. Up to 512 requests could be pending at one time. These were sorted by disk arm position and starting sector. The arm was moved elevator fashion back and forth, and upon reaching a disk position with pending requests, all such requests, pending at the time the position was reached, were serviced. The arm was then moved to the next position with outstanding requests. Within a given position requests were selected approximately in the order of disk rotation. The algorithm used was to select a request at some distance beyond the current rotational position, prepare for the transfer, check that the I-O request could still be made without an intervening full rotation, then execute that actual I-O instruction. The only suitable final check was to insist that the rotational position be more than one sector in advance of the desired sector. For reads, this led to choosing a request about 3 sectors beyond the current disk position. Write requests required more preparation time than reads as it was necessary to move the data from ECS to the PPU before the write could proceed. This move required as much time as needed to write the data on the disk. I-O requests with the same arm position and starting sector were handled in the order made. This was the only order condition satisfied by the algorithm.

Sixty-four words was the maximum size record that could be written in one sector position, allowing for two internal address check bytes. The 129 and 257 word records fit in 2 and 4 sectors respectively, while the 513 word records fit in 7 sectors. Except for the 64 word records, the others were a power of 2 plus one, allowing one check word to be used by the next level of the system, and still provide power of two record sizes to the ultimate user.

To make a read request, the higher level disk system would first obtain a slot from a slot event channel for permission to make the request. The disk system would then send an event, on a request event channel, which would contain the disk address for the request, the size of the request, an internal number to identify the response and the index of the slot for the request obtained from the slot event channel. The request would be stored by the ECS I-O system code in an internal table indexed by slot number. These entries were chained together for equal arm positions and eventually for equal starting sector. The response would appear on a response event channel, and would contain the internal number of the request, the index of a buffer containing the data and a completion code which indicated if any errors had occurred. The higher level disk system would eventually release the buffer by sending an event containing the index of the buffer on a buffer event channel.

To make a write request the higher level disk system would first obtain the index of a buffer from a buffer event channel, and would write the data in that buffer. Then, as in the case of a read request, it would obtain a slot number and send the request containing the buffer index, slot index, internal number, disk address and size of the request. The response event would contain a completion code, and in the event that there were no errors the buffer would have been automatically released. (Figure 3 diagrams the flow of events between the higher level disk system and the ECS system disk driver described here.)

Information as to the current position of the disk arm and current direction of motion was made available to the higher level disk system so that most writes could be in a position expected to be serviced soon. This, together with the fact that a read buffer was not assigned

until the actual read was about to start, reduced the amount of ECS space required for buffers. It was expected that under heavy load a complete scan of the disk might take several seconds, possibly up to 10. However, any given block of data would be held in buffers for only a fraction of this time.

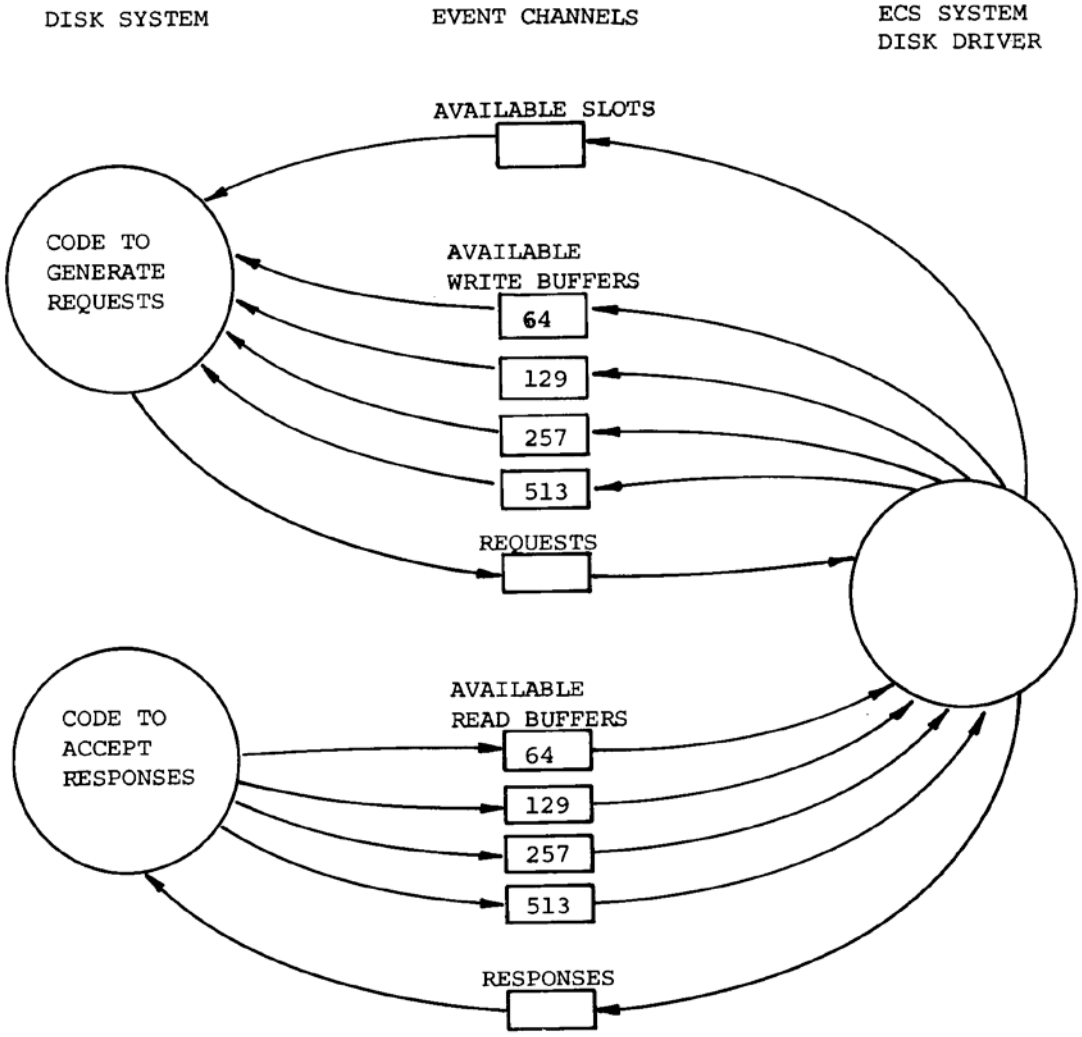

Figure 3. Sketch of event flow between disk system code and ECS system disk driver.
The request and response channels carry a 60 bit coded request or response. The other channels carry indices of available items.

CHAPTER 10: DISK/DIRECTORY SYSTEM

The disk/directory system provided the user machine. (The ECS system was merely intended as a tool for the production of the disk/directory system.) Further levels above the disk/directory system were thought of more in terms of executives than artificial machines.

Objects that existed in the disk/directory system were called disk objects, of which the two major ones were files and directories. The files provided a place to store data much as in the ECS system and the directories provided a naming structure for the disk objects. Also, the directories stored disk level capabilities, corresponding to the function provided by C-lists in the ECS system. A number of other disk objects existed and will be described subsequently.

Disk files

A disk file had the same structure as an ECS system file. A portion of the file could reside on the disk and a portion in ECS. Which portion was in ECS varied with time. If a read or write request was made to a portion of the file residing in ECS, the action would proceed as fast as if it were a read or write on an ECS file. On the other hand, if the request was made to a portion not in ECS the request would proceed more slowly. An action was available to attach one or more blocks of a file. This action resulted in those blocks being moved into ECS, but computation could proceed while the move was being made. If subsequently the program attempted to read or write data in the attached blocks, the program would be forced to wait until the blocks were in ECS, and then would be permitted to continue. Thus the system essentially provided a buffering facility. Figure 4 gives an example.

Another action that could be performed on disk files was to place parts of them in subprocess maps. This had an effect similar to placing ECS files in a map, portions of the disk file became accessable through load and store instructions. One step in achieving this result was to implicitly attach the blocks containing the mapped data, thus moving them into ECS.

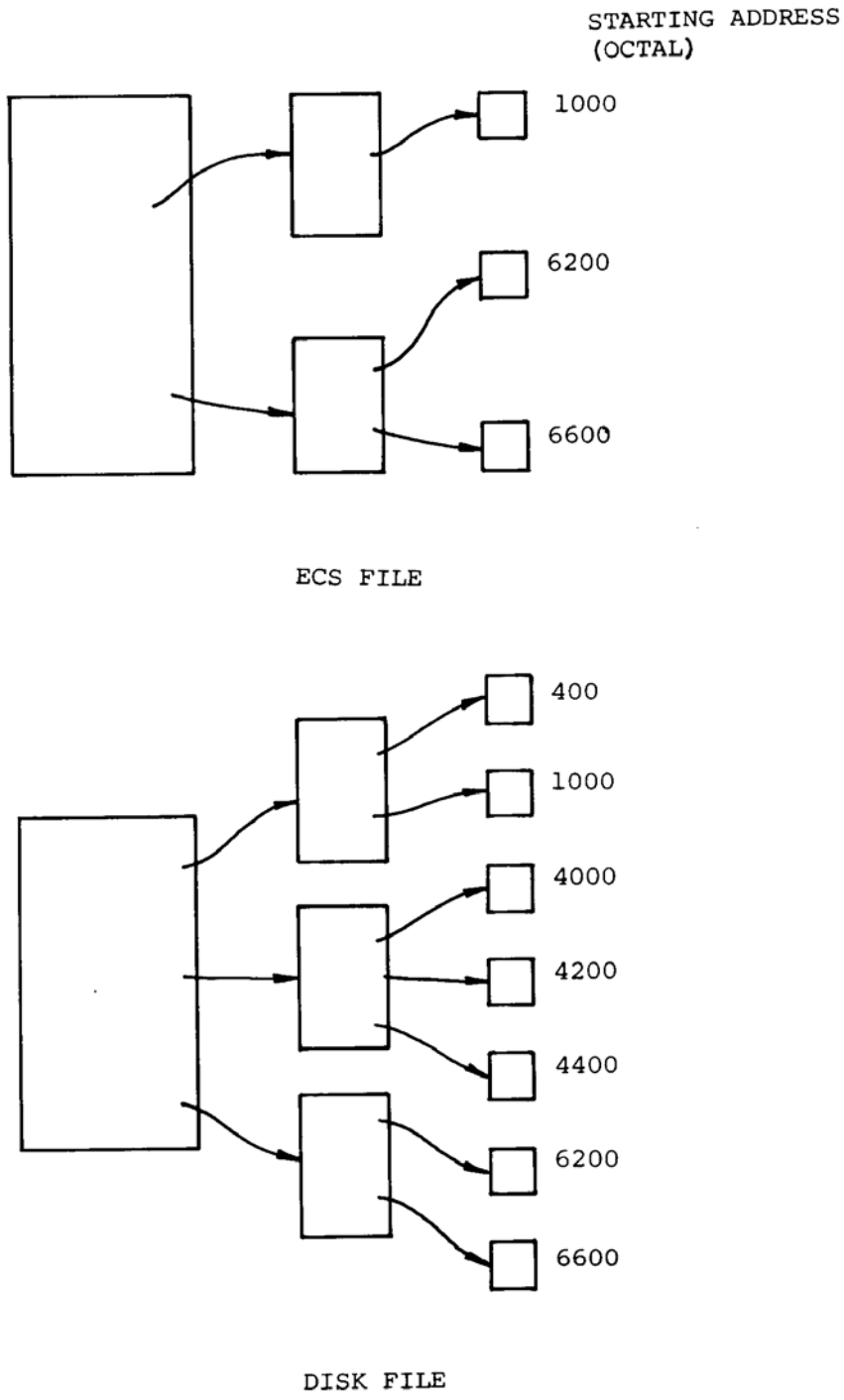

Figure 4. Example of an ECS file representing a portion of a disk file (128 word data blocks, 8 pointers in each second level pointer block).

It was assumed that the total space desired in ECS by all processes would exceed the capacity of ECS, so we intended to provide two states for a process, swapped in and swapped out. When swapped out, those portions of disk files being held in ECS for the process would be moved out to the disk and the activity of the process would be suspended. We further assumed that space

required in ECS for a process beyond that held in disk files would be very small, so that many processes could exist in a swapped out state.

Access keys and locks

Access keys were capabilities that contained an integer. They were to be used like keys, which could fit certain locks. Associated with entries in directories were lists of access key numbers which acted as locks. In order to access a given entry in a directory, one had to present a key which fitted one of the locks on the entry.

Directories

A directory was a disk object which consisted of a list of entries. Each entry contained a *symbolic name*, an *object specification* and a list of *access-locks*.

a) The *symbolic name* was merely a sequence of characters.

b) The *object specifications* could be one of three things:

   i) an owned entry;

   ii) a hard link, i.e., there would be a pointer to a disk object not owned by this entry;

   iii) a soft link, specified by a pointer to another directory, a text name to be used for look up in that directory and an access key.

c) An *access-lock* was a pair:

   i) a number to be matched against an access key

   ii) a set of option bits.

Each disk object, except a root directory, had exactly one ownership entry in some one directory. Thus the directories formed the objects of the disk system into a tree structure.

The access action required three arguments: a directory, a symbolic name, and an access key. If an entry was found in the directory with the symbolic name, the access-lock list for the entry was scanned. If a number in the access lock list was found which matched the number in the presented access key, then an ECS system user type capability for the object specified at the entry was returned with the option bits associated with the given access key number. (As described in the chapter on implementation of the disk/directory system, the directory system maintained private capability-creating-authorizations for creating user capabilities for disk objects.)

The basic directory access action just described had a number of variations. In order to reduce the length of access lock lists, we defined a special access key, the null access key. The number in this key occurred implicitly in all lock lists and had an associated set of option bits, of which all others were subsets. The null access key was accepted as the third argument only if the directory capability in the first argument had an appropriate option bit on. A capability with this bit on was generally only available to the owner of the directory.

If the object at the entry was found to be specified by a soft link, a further directory lookup was automatically made with the directory, text name and access key supplied by the entry. The

intended purpose of this facility was to provide pointers to objects whose identity would change from time to time.

The modification of the access-lock list for a given entry did not require possession of the indicated key. An ordinary datum containing the number of the key was sufficient (in addition to a suitable capability for the directory, of course). Thus, one user could provide access to one of his files for a second user, without ever possessing access to that second user's access key. The second user need only give the number of his access key to the first user (e.g., in a written note).

Other actions on directories included provision for listing the contents of a directory, creating and deleting entries, etc.

Disk space control

We placed the control of disk space into the directories. Rather than have accounting blocks to control space as in the ECS system, we decided it would be simpler to assign disk space to the directories. When a directory was created it was specified to be an accounting directory or not. Whenever disk space was needed to create or enlarge a file or other disk object, the disk tree was scanned starting with the object and moving towards the root directory until an accounting directory was found. This accounting directory was then checked to see if it had sufficient available space for the action intended, and if so, the available space was reduced. (To save time, a pointer was placed in each directory to point back to its associated accounting directory.)

Each user was assigned an accounting directory and given space in it. The user could build as large a tree as he desired from this directory, within the limits of the space he was given. He could even create subuser directories. Using this mechanism we could delegate the authority for making user directories to individual departments of the university, thus reducing the clerical work of the computer center.

Name-tags

No facility was provided for entering ECS system objects into directories. In fact, the underlying ECS system was destroyed daily and recreated the next day. Thus it was impossible to conceive of a directory containing ECS objects. As a way around this difficulty, two types of disk objects were provided, static and dynamic name-tags. Name-tags could be associated with ECS system objects, and actions were provided to obtain a capability for the object associated with a given name-tag. Static name-tags always referred to the same (or isomorphic) ECS system objects; and dynamic name-tags could be associated with arbitrary ECS objects by user programs. (In fact, the association was with capabilities, hence not restricted to ECS objects.) Elaborate conventions were required by the user programs to make sure that the dynamic name-tags were in fact associated with appropriate ECS objects, since the association was destroyed daily.

The system used name-tags to provide directory entries for the ECS files and event channels associated with I-O devices. During each system startup, the appropriate associations were reconstructed.

Subprocess descriptors

A facility provided by the directory system for use by the higher levels of the system was that of subprocess descriptors. There was a need to construct a subprocess within a users process, at the request of a user, which had access to capabilities not available to the user. One purpose

of this was to provide for actions which manipulated objects within the system to which the user could not be given direct access.

A subprocess descriptor was simply a capability for a directory, but with a different type. An action was provided to the higher levels of the system, not made available to the users, which could convert this capability into a directory capability. One of the actions to be provided by a higher level of the system was to take such a capability, convert it to a directory capability, look up a special name within the directory, take the resulting capability to be that of a disk file; and interpret the contents of that disk file as a description of a subprocess to be built within the users process. Part of this description would be text names. During construction of the subprocess (done by the command processor, see chapter 13) these names would be looked up in the directory defined by the subprocess descriptor, and the resulting capabilities were placed in the local C-list of the new subprocess.

Scan lists

It was expected that a user would actually want to look up a name in a sequence of directories (e.g., first a local directory associated with his current process; second, a directory containing files that live from session to session and third, a system provided directory containing generally available subsystems). In order to automate this procedure, we introduced the idea of a scan list. This was simply an ordinary C-list in which the capabilities alternated between directories and access keys. An action was provided by the directory system which would accept two parameters, a text name and a C-list. The directory system would assume the C-list was in the form of a scan list, and look up the text name in the successive directories contained in the C-list, and check against the associated access keys. An error was returned if the directory system did not find a capability for a directory or access key when appropriate.

CHAPTER 11: IMPLEMENTATION OF DISK DIRECTORY SYSTEM

The disk/directory system was composed of a number of components. These included three subprocesses within each user process, a number of special processes, and some miscellaneous packages for startup and shutdown.

Disk-directory system code within a user process

The disk/directory system implemented a two level virtual machine on the ECS system. The two levels were the disk system and the directory system. The disk system implemented the concept of disk files, which the directory system in turn used to implement directories. The original purpose for this division was to modularize the implementation, and make debugging easier by isolating the problems. A later intention was to combine the two layers once the system was debugged, but this point was never reached.

The disk system consisted of two subprocesses within each user process, and a number of ECS system processes, invisible to the user. (The user process tree is described in chapter 14, "A Short Tour of a User Process".) One of the two subprocesses within the user process was intended to handle F-returns on disk file read and write requests, that is, reads and writes directed to portions of files not currently in ECS. This subprocess was placed in the tree so that it would be in the full path of any subprocess causing such an F-return. Thus it had direct access to the core of the requesting subprocess. Because the entire full path had to be capable of residing in central memory, any subprocess in the full path of a user subprocess reduced the maximum possible size of that user subprocess. Therefore, much of the disk system code was placed in a second subprocess which sat off to one side of the users full path.

Special system processes (non user processes)

There were three kinds of special processes. The first kind handled the interpretation of responses from the ECS system disk I-O code. These responses were sent by the ECS system to a single event channel. (We did not want the ECS system to have to know about the many user disk processes.) In order to send a response to the process originating the request, an intermediate process examined the response to determine appropriate further processing. If necessary, an event would then be sent on an event channel looked at by the originating process.

A second kind of special process initiated disk I-O for such functions as closing files. File closing was a two step procedure in which all blocks except the header were first written to the disk. When these writes had been successfully completed the header was written. Since the modified pointer and data blocks were written at new locations on the disk, the effective disk version of the file showed no change until the header itself was written. Thus the disk representation of the file was always a good disk file. This procedure took time, and it was desirable to permit the user process to proceed with other business. Hence a special process performed the step by step procedure.

The third kind of special disk system process controlled the total disk space allocated to particular user directories. The information about space held by these directories was all held in one disk file, and maintained by this special process. This process had in itself a disk system which treated the special file just as any other disk file. For historical reasons this process was a part of the disk level and accepted commands from the directory level.

Disk file capabilities

From the the ECS system's viewpoint, a capability for a disk file was a user type capability. The capability-creating-authorization for these disk file capabilities was retained by the disk/directory system and not made available to the general user. To open a disk file, a user program presented to the disk system a capability for a disk file. The disk system read the data part of the capability to obtain a disk address and unique name. Since there was no way user programs could fabricate such a capability, the disk system knew that the disk address and unique name had been put there by itself. As a final check during the open operation, the disk file header was read from the given disk address and checked for the given unique name. This was necessary because the file might have been destroyed, and that disk address used for some other purpose. These disk system unique names were distinct from the ECS system unique names. A different one was associated with each disk file. The contents of the files were dumped to tape each night and reloaded the next morning. During reloading, new disk files were created to hold the old contents, and consequently unique names were reassigned. Thus the unique name of a file differed from day to day.

Directory system directories

ECS capabilities for disk files were created by the directory system. The directory system was implemented as a single subprocess within a user process. The directory system maintained each directory in a single disk file. Directory capabilities were another special kind of ECS system user capability, for which the capability-creating-authorization was retained by the directory system. These directory capabilities contained the same information in their data parts as disk file capabilities, but since they were of the wrong type they could not be used as disk file capabilities by user programs.

Dynamic name-tags

The only form of name-tags actually implemented were dynamic-name-tags. (The desired effects of static name-tags were obtained through the use of dynamic name-tags.) A dynamic name-tag was represented by a unique name, carried in the data part of a capability. A hash table, maintained in an ECS file by the directory system, provided an index into a C-list, private to the directory system. This C-list contained the capabilities, if any, currently associated with each dynamic name-tag.

CHAPTER 12: A CONSISTENCY PROBLEM FOR DISK FILES

We made three decisions early in the design of the system, which together, had unforseen consequences. These were:

i) The current version of some portions of a disk file may be in ECS, with no copy on the disk (e.g., attached blocks).

ii) After a crash, we must be able to restart the system using only data on the disk. (It was felt that the structures in ECS were probably too fragile and complicated to reconstruct after a crash. Also, one of the more frequent causes of a crash was failure of ECS.)

iii) Vital information, necessary to the integrity of the system, would be stored in disk files. This included directories, with access control information, and the system accounts. (Once disk files had been invented, we saw no reason to invent other disk storage facilities.)

The resulting problem was that the contents of a file after recovery from a crash may not be the same as before the crash. Moreover, it is conceivable that they may not represent the contents at *any* previous time (i.e., one portion may represent the contents of a different previous time from another portion).

Initially we felt that this would just be "tough luck" for some unfortunate user, and it was his responsibility to maintain backup facilities. Unfortunately, we forgot decision iii) above.

We eventually found a way around the problem, described below, but it greatly increased the system overheads involved in the maintenance of the system accounts.

The problem

The current contents of a given disk file are defined by data residing at many different locations in the physical machine. Some data is on the disk, some data is in ECS and some data may be in CM. In order to completely restore the contents of a disk file after a crash to the contents immediately before the crash, all of this data must be available.

We made the somewhat arbitrary decision to ignore the data in CM and ECS during crash recovery. This decision was motivated by three considerations. First, it was easier to write a recovery procedure that relied solely on data stored on the disk. Second, a more complicated procedure must include a procedure depending only on the disk, in the event that data in CM and ECS proved inconsistent. Finally, we felt that in most crashes the data in CM and ECS would be unreliable, thus we must be prepared to recover from the disk alone.

It is conceivable that the system could have been designed so that the data on the disk represented a “snapshot" of each file, taken by request of a user program. (That is, make sure the disk version of the file correctly represents the current contents.) At one time during the design we attempted to do this. However, we decided that it was wasteful of disk space. Consequently, the system took snapshots from time to time, unpredictably. Moreover, these snapshots only included data in ECS, they excluded data in CM. Thus, a snapshot might include some changes to the file made later than other changes which were not included, if the earlier changes were still in CM.

In an attempt to permit a program which was using disk files to protect itself over crashes, we

provided an action that forced a snapshot. At the completion of this action, the data on the disk would be a faithful representation of the file. (This was true only if all data representing the file was in ECS or on the disk, and no other program was modifying the file.)

The problem will also occur in other systems

In other systems, if we assume that the contents of a file will be represented by data at many locations within a physical machine, and that some of these locations will be unavailable after a crash, then there will be a similar problem. At best, the system may offer snapshots that are under complete control of the user. However, if a user has a data base stored in more than one file, after a crash these files will contain snapshots taken at different past times.

Consequences of the problem

A using program is not interested in the bits stored in a file for themselves, but generally uses them to represent some useful logical structure. For example, an accounting program may intend to update a number of accounts to reflect a sequence of transactions. The data in one or more files may represent the state of the accounts, how much money each has, while another file may represent the sequence of transactions, "move so many dollars from one account to another".

If this accounting program took no precautions, and a crash occurred while the accounts were being updated, the contents of the represented accounts after recovery will have no predictable relationship to the contents before the crash.

A simple solution would be to have backups for all the files, and return to the backups after a crash. There still remains the question of how to identify which files are current. If this information is maintained in files (where else?), and a crash occurs while it is updated, chaos may still occur.

If the accounting records are extensive, and the transaction file is long, it may be too expensive to maintain a complete backup of all accounting records while the transaction file is processed.

A solution

The solution we chose was to prescribe a careful, multistep method for making changes to files. In essence, one first writes one's intentions into the file, then carries them out, and finally removes the intentions. The idea is to arrange things so that if a crash occurs, the disk version of the file will appear to have either no changes, or all of the changes. This is accomplished by an agreement, among all programs that maintain the file, to perform any changes to the file that they find recorded as intentions.

The recorded intentions must satisfy a number of conditions. It must be possible to repeat them, even if they have been partially, or completely, carried out; leaving the file in the intended state. (A simple list of file addresses with intended new contents would satisfy this condition.) After a crash, the file must not have a partial list of intended corrections. (A flag that signifies the presence of intended changes will satisfy this condition, if it is not turned on until the entire list of intended changes are guaranteed to be in the file.)

The algorithm for making changes is:

i) lock out other programs from the file (or files).
ii) If the bit signalling the presence of intentions is off, go to step vii).
iii) (A crash has occurred, we are now recovering.) Perform the intended changes.

iv) snapshot the file(s).
v) turn off the intentions bit.
vi) snapshot the file containing the intentions bit.
vii) read the file(s) to determine the desired changes.
viii) write a list of intended changes at some known place in the file(s).
ix) snapshot the file(s) containing the list of intended changes.
x) turn on the intentions bit.
xi) snapshot the file containing the intentions bit.
xii) make the changes.
xiii) snapshot the changed file(s).
xiv) turn off the intentions bit.
xv) snapshot the file containing the intentions bit.
xvi) unlock the file(s).

Discussion

Various simplications of this algorithm are possible in special cases, but I believe that in the most general case, all of the snapshots are needed. If the list of intentions all appear in one file block, so that the entire list is either present or not after a crash, then it is possible to dispense with the intentions bit. If all changes will be in a single file, as well as the intentions list, and the system will guarantee that all of the contents of a file after a crash did exist simultaneously at some time before the crash, then all snapshots except step xv) can be removed.

It should be noted that the logical state represented by the file(s) changes at step x), but if a crash occurs before step xi) is completed, the representation may return to the old logical state.

Finally, snapshots require a significant amount of real time. Disk operations must be started, and completed. Some of the vital information (e.g., user accounts) in CAL TSS was maintained using a simplified version of this algorithm, and this contributed to our system overheads.

CHAPTER 13: COMMAND PROCESSOR

The final level of the system, the command processor, was constructed under strong time pressure, and was difficult to view as an artificial machine. It provided a large assortment of functions, mostly determined by the fact that no previous level had provided these functions.

Some of the functions included at the command processor level were octal debugging, subprocess construction from subprocess descriptors, teletype line editing, looking up of compound names in a number of directories, money accounting on a user by user basis and pass word protection for entry to the system.

Octal debugger

The octal debugger permitted the inspection of the contents of disk files by name and address, and the modification of those files. It permitted the inspection of the contents of the core of an interrupted user subprocess and the modification of that core. It also permitted the inspection of the C-list of such a user subprocess and the modification of that C-list. It permitted the examination of directories by name and the modification of those directories, as permitted by the capabilities obtained from the names. It permitted the interruption of a running subprocess by hitting certain keys on the users TTY. It also obtained control over a user's subprocess if the subprocess committed an error and did not catch the error itself.

Subprocess construction

A general facility was provided for constructing a subprocess from a description in a given file. This description contained such information as entry point address, size of a scratch file, names of files to be used in map entries, etc. Various conventions were used to determine in which directory to look up a name.

A special case of this facility accepted as input a subprocess descriptor (as described in chapter 10, "The Disk/Directory System"). The command processor had had access to the action which converted the descriptor to a directory. The command processor then looked up a special name in the resulting directory, obtaining a file which it processed as above.

TTY line collector

The original intention in the design of the system was to permit the user to construct his own I-O interface routines, if he so desired. Owing to the time pressures we were never able to implement some of the features necessary for this, and even so we would still have had to supply standard interfaces for the user who desired them. For these reasons we supplied a standard subprocess in all user processes which collected characters from the TTY and transmitted characters to the TTY. The program in this subprocess normally collected a complete line up to a carriage return before returning to the caller. This program had a fairly sophisticated built in editing routine that permitted copying portions of a previous line, skipping portions of a previous line and replacing portions of a previous line. These functions could be controlled on the basis of the characters in that line, for example, skip up to a given letter. These functions were controlled by the control keys on the TTY, and were arranged in a simple regular pattern on the keyboard. For the purpose of editing lines already existing in files, a subprocess could call the line collector with a line to be edited as if it had been a previous line.

Naming

Obtaining an object by name within our system was somewhat complicated. There were two general reasons for this. First, we had a multiplicity of directories, and a user was generally confronted with at least three of them. Second, we tried to minimize the access a subsystem was given to a user's permanent directory.

A user was confronted with at least three directories. There was a directory which contained publicly available system provided subsystems (system directory), such as the editor, SCOPE simulator and a printer driver. There was a directory which contained the users files which lived from session to session (permanent directory). Finally, there was a directory which contained temporary files associated with his current logged on teletype (temporary directory); such as scratch files containing the memory of subsystems he was using.

It was necessary to maintain both a temporary and permanent directory for two reasons. First, two teletypes could be logged on under the same user name. Thus, they would have access to the same permanent user directory. A directory associated with the teletype was necessary in order to prevent naming conflicts among temporary files constructed by subsystems. Second, we associated disk space control with directories. Due to the limited amount of disk space available, it was necessary to severely limit the space occupied by files that lived from session to session. On the other hand, some subsystems required an enormous amount of temporary file space while running. Thus, the users permanent directory was provided with a small amount of space, while his temporary directory was provided with several times that amount of space.

In general, whenever a name was presented to the system to be looked up in directories, it was intended to be looked up first in the temporary directory, then the permanent directory and finally the system directory. That is, try the most local scope first, and then try larger and larger scopes. In order to automate this, these three directories were placed in a scan list.

Since a scan list was merely a C-list, if this scan list was provided to a subsystem, that subsystem would have access to the directories contained in the list. If those capabilities were strong enough, the subsystem could list all the names occurring in the directory, and using those names, delete all the files. We had a vision of a run away subsystem destroying, in a few seconds, all of a users permanent files. Also, it was necessary to provide some name look up facility to subsystems, e.g., for use by assemblers with facilities to include text in other files named by the one being assembled.

In an attempt to prevent this somewhat unlikely catastrophe, while providing a name look up facility for subsystems, we provided two scan lists, one weak and one strong. The strong one would be used in response to commands typed on the teletype, while the weak one was supplied to the subsystems.

Passwords and accounting

The command processor supplied the entire password and money accounting portions of the system. The logon procedure required a user to name a permanent directory. This permanent directory contained a system pointer to a user profile maintained by the command processor. This profile contained, among other things, a password. The user was requested to type in a password and this was checked against the profile. If it matched, the logon procedure was permitted to proceed. Once logged on, the user could change his password via a special command available in the octal debugger. At the end of a session, the command processor could examine the various totals of space time and CPU used by the process, convert these to dollars and subtract the result from a running balance maintained in the profile. A user who had

subdirectories which were also permanent directories could transfer funds to and from those directories.

Each permanent directory profile contained permission to construct a certain number of descendent permanent directories. This permitted the administration to construct one permanent directory for a class and let the instructor construct the necessary student permanent directories under the class permanent directory.

CHAPTER 14: A SHORT TOUR OF A USER PROCESS

The subprocess tree of a user process contained 9 system subprocesses, one initializing subprocess that destroyed itself after starting the process, and one or more user subprocesses. The following is a list of those subprocesses with a brief statement of their function. (See figure 5.)

Root

All subprocess trees had a root. Instead of having a functional subprocess for a root we had a very tiny (72 words of CM, all shared) subprocess which could catch disastrous errors in the system subprocesses for later analysis. It also contained some code for the final destruction of the user process.

Builder

The Builder was a transient subprocess that constructed the system portion of the subprocess tree. It was guided by a descriptor file which described the subprocesses to be constructed, and their location within the tree. Files to be used in the subprocess maps, and capabilities to be placed in their local C-lists were specified by text names. The builder searched a global list of names for each text name supplied. A global C-list contained a capability corresponding to each position in the global name list. This was essentially a linking operation, similar to that provided by a linking loader for user programs in many systems. After constructing the system subprocesses, the Builder destroyed itself and started the appropriate system subprocess.

Fake bead ghost

The BEAD GHOST subprocess (described later) was provided for aiding in the debugging of user subprocesses. The Fake Bead ghost was a stripped down version installed to provide debug facilities for the system portion of the process.

Disk F-return read write

This was a disk system subprocess which would be in the full path of any user subprocess requesting disk actions. It performed the actual read or write of the data in the user subprocess address space.

Disk

The disk subprocess contained the main body of the disk system code residing in a user process. It was placed out of the full path of a user subprocess so as to increase the maximum CM field length available to the user subprocess. This was necessary since all subprocesses within a full path must be capable of residing in CM simultaneously.

Directory

The directory subprocess contained directory system code. It provided all directory system services for the user.

Bead Services

An interim monitor available in 1969 and ’70 had been called the Bead. We felt it necessary to

continue the services provided by that monitor and placed the necessary code in this subprocess. Also, Bead Services supplied a number of specialized services for the command processor, such as recording charges against a users funds.

Bead ghost

This was a subprocess placed in the full path of the user's subprocess to provide debugging facilities, and an interface to Bead Services.

TTY line collector

This subprocess provided a standard interface to a user's teletype. It collected single lines, and output single lines. It contained an editor for correcting a typed line before it was transmitted to the using program. (Thus typed input for all programs could be corrected in a uniform manner.) As an option, it allowed operations in units smaller than single lines.

CMMD

This subprocess contained the command processor and debugger. It provided complicated naming conventions beyond those provided by single directories. It provided facilities for calling named programs. It provided commands for access to user programs being debugged. It provided general system services such as construction of new user directories and transfer of resources from one user to another.

User

This was the location for a user subprocess, if one existed. Any further user subprocesses would have this one as an ancestor.

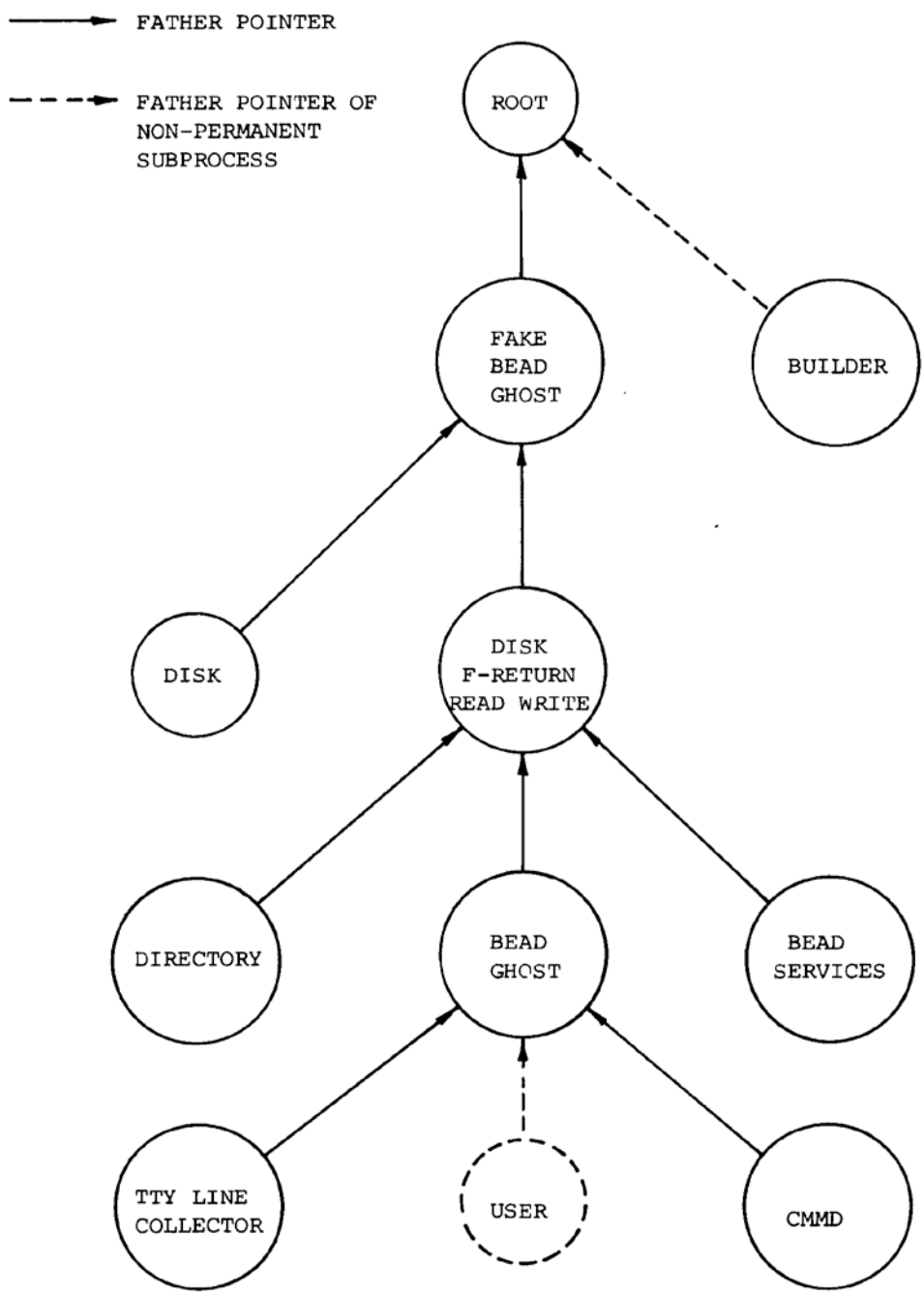

Figure 5. The subprocess tree of a user process.

PART THREE: ASSORTED REACTIONS

CHAPTER 15: DISCUSSION

Part Three discusses how the ideas mentioned in Part One worked out. This chapter presents an overall summary, while subsequent chapters go into more detail on some of the issues.

Disappointments

As with many other operating system projects, the system we actually constructed disappointed us in several ways. It was larger, slower and considerably more complicated than we expected. It has been difficult to explain why this happened, although one possibility is that our expectations were too naive. Chapter 17 details the external manifestations of these problems.

The fundamental ideas

The fundamental ideas of Chapter 4 naturally divide into two groups: those with which we were satisfied, and those with which we were not. The ideas which worked out well were:

The concept of an abstract machine,
Capability based protection,
Processes,
Layered implementation,
Uninterpreted I-O devices.

The unfortunate ideas were:

Mapped address space,
Distributed system code.

Probably the worst disaster of the project was to attempt to implement a mapped address space on an unsuitable machine (i.e., no mapping hardware). Also, distributed system code turned out to be considerably more difficult to design than we anticipated, leading to some very complicated and not very well understood programs. These problems are more fully discussed in Chapters 20 and 21.

The concepts of an abstract machine and capability based protection so permeated our thinking that it is impossible to conceive of the project without them. They provided the essential framework supporting all of our design work. These ideas are explored more fully in Chapter 16.

The ECS system was the cleanest realization of these ideas. Unfortunately, as implemented, the computation cost of an ECS system call (virtual instruction) was higher than we anticipated. Chapter 18 explores improvements in the implementation which would have substantially reduced this cost, while Chapter 19 proposes hardware modification to the CPU which would further reduce the cost.

Support for special user level programs

All of the special purpose programs described in Chapter 5 were supported as intended. The SCOPE system simulator ran as the first user level subprocess, with its user as a direct descendent. The ECS system contained a special feature which generated an error whenever cell 1 within a subprocess field length became nonzero. (This is the form of a user call in the real SCOPE system.) The SCOPE simulator subprocess intercepted this error, and when

called, the user subprocess would be in its Full Path. Thus the SCOPE simulator would have direct access to the user program's memory.

The debugger (most of which existed in the command processor) gained access to the local memory and C-list of a subprocess through the Bead Ghost, a small subprocess which was the immediate ancestor of the user subprocess. (As in the case of the SCOPE system simulator, this access was provided by the Full Path.)

Both the SCOPE system simulator and the debugger gained access to a subprocess local data through the Full Path. I feel an alternative to the Full Path was desirable.

The full path was a device which would permit the memory of a subordinate program to appear as an extension of the memory of a more powerful program, e.g., a user program running under the SCOPE simulator or the subject program of a debugger. An unexpected effect of the full path was to force an unnatural division of system programs into two parts, one which had access through the full path to a subject program, and a second larger part which did not.

Actual references to the memory of the subject program were infrequent, and would have been adequately supported by a special virtual instruction. In fact, the division of system programs into two parts could be thought of as an attempt to simulate such an instruction. However, the map facility made such a virtual instruction difficult to implement directly in the ECS system. In Chapter 20 we discuss other problems with the address map facility, and propose an alternative.

CHAPTER 16: A SUCCESS

The most successful aspect of CAL TSS was the concept of an abstract virtual machine with abstract objects referenced by capabilities. This success manifested itself in a number of ways:

It provided a unified conceptual framework.
It provided an easy to understand protection mechanism.
It provided modularity with clean interfaces.
It resulted in an almost error free ECS system.

Unified conceptual framework

The basis for all of our design work was the concept of a set of objects, each of which can be accessed only through a small set of primitives. Any design proposal was implicitly expected to consist of three parts:

i) an abstractly specified set of states for an object,

ii) a set of primitive actions on the object and their effects on the states,

iii) a representation for the states.

The concept of C-list combines and generalizes a concept which appears in many operating systems, that of a list of entities which a process may reference. For example, consider TENEX [T1]. Associated with each job is a list of files which processes within that job may access, while associated with each process is a list of processes which may be referenced. Neither of these lists provides the full power of a C-list, and presumably they are managed by entirely separate system routines. Within CAL TSS, there are no explicit lists of this kind. Rather, they are implicitly represented by the capabilities that appear in the local C-lists of the user subprocesses.

Furthermore, associated with each process in TENEX is a set of "capabilities" (binary flags) which control which system calls are permissable for the process. The same control in CAL TSS is provided by controlling which operations (accessable only through capabilities) are available to a particular subprocess. This is possible since the only operations available to a subprocess are those in its C-list, and in C-lists accessable to the subprocess.

Easy to understand protection

In CAL TSS, all protection is founded on one idea: possession of a pointer to an object (capability) grants access to the object as specified by the pointer. Two subprocesses within a single process have different access rights because they have access to different sets of pointers. In general, access rights are transferred from program to program by moving pointers.

Access keys and locks are a somewhat different mechanism. However, they are implemented using the capability machinery, and possession of a capability for an access key is necessary to open a matching lock. Furthermore, even though an object in another directory may have a lock matching a given user's access key, the user must explicitly obtain a capability for the object (using his key) before he can access the object.

A capability based protection system seems to provide features that are difficult to obtain in other systems, such as Multics. Two such features seem to be:

1) The ability to provide for mutually suspicious subsystems within the same process;

2) The ability for new layers of system to be constructed, which provide new virtual objects, and permit protection for these objects to be controlled using the same machinery as used for more basic objects.

CAL TSS provides for mutually suspicious subsystems simply by placing them in subprocesses neither of which is an ancestor of the other. One of these subprocesses may touch objects accessable to the other only through parameters passed in a call, or with the assistance of a common ancestor subprocess (presumably a trustworthy bystander).

The addition to the disk/directory system of the ability to construct "user" disk/directory system types would have made it possible for new system layers to construct new types of virtual objects. (This facility was provided by the ECS system, and was used by the disk/directory system to provide disk/directory objects. It is a quite simple feature to implement.) Since capabilities for these new type objects could be placed in directories, access to them could be controlled in exactly the same ways as access is controlled to disk files and directories. (For example, this feature would have permitted nametags to be implemented by system code written upon the disk/directory system, rather than directly implemented in the disk/directory system.)

Modularity with clean interfaces

The user's program views an abstract object through a small number of functions which modify the state of the object. Any changes in the algorithms used for these functions, or the internal state representation, are not directly visible to the using program. Thus only a small number of programs depend on the internal state representation, and they can be collected together in a single module. This is very similar to the form of modularization proposed by D. L. Parnas [P1].

It resulted in an almost error free ECS system

During the last three months of operation an error report was made for most system crashes. An examination of reports, 18 in all, showed one crash for unknown reasons, three for suspected hardware causes, and the rest identified as high level system errors (disk/directory system or command processor). During this period the system was run for at least 8 hours each working day, with a fairly continuous load of several users. Even if the one unknown and three suspected hardware crashes are attributed to the ECS system, I feel that this represents an excellent record.

The system change log for the last six months of operation records 41 system modifications, of which 7 contained changes to the ECS system proper, and 6 contained changes to the I-O drivers. Only 4 of the 7 changes to the ECS system proper were for repairing errors. The others were for adding new features or changing assembly parameters.

CHAPTER 17: SOME DISAGREEABLE FACTS

As with many other operating systems, the system we actually constructed disappointed us in several ways. In particular, it was large, slow and difficult to use. This chapter surveys most of our disappointments. It must be kept in mind that this is a description of the results of the first implementation of the system. Other systems, notably Multics, have had many rewrites which greatly improved their performance. We feel the same could be expected for CAL TSS, and in the next chapter some immediate improvements will be suggested.

A major problem for this description is the lack of detailed measurements of system overhead, either in memory space or CPU time. This is due to the very strong pressure we felt towards producing a working system. We assumed that once we had a system up and working, we could then analyze the system, look for trouble spots and clean them up. We never reached that stage of development.

Large

The system was large in a number of ways. First, it occupied a large amount of central memory, thus reducing the available field length for the user. The ECS system, together with I-O buffers, required about 7K words (1K 1000 base 10). The process descriptor together with the subprocesses in the path between the user subprocess and the root required another 4K words. On our 32K machine, this left about 21K words for the user. Furthermore, a user of the SCOPE simulator was penalized another 1K words, leaving him about 20K words. This last can be compared with the real SCOPE operating system, as run at U.C. Berkeley campus computer center in 1971, which occupied 12K words of C.M. On a 32K machine this also would provide a single user with a maximum of 20K words of central. Thus, in some absolute sense, the SCOPE system preempted as much CM space as our system. However, the SCOPE system was run on a machine with 64K of CM, while we had only 32K.

Second, the system has a fixed overhead in ECS of 140K words. (Almost half of the available ECS.) This overhead was composed of system code and tables. Most of the space seemed to be occupied by the disk system, but we never had a chance to do a detailed accounting. It was fairly clear that the ECS system accounted for a fairly small fraction of this overhead, and most was due to higher levels of the system.

Third, there was a per process overhead in ECS of about 10K words. This figure was better understood than the fixed ECS system overhead and was expected to decrease. It was composed of two major parts:

i) about 3K was consumed by the local C-lists and storage for the 8 system subprocesses, along with the space necessary to define the process structure and provide a subprocess call stack,

ii) about 7K was used to provide an ECS image of portions of disk files attached by the process.

It is probable that the implementation of process swapping by the disk system would have reduced this per process ECS overhead to around 3K (only “swapped-in” processes would require the 7K for ECS images of disk files). Further reductions would have required a redesign of some portions of the system. Even without process swapping, various developments under way at the termination of the project would have reduced the overhead, possibly by as much as 4K, leaving a 6K overhead.

Thus, since only 300K of the 500K ECS was available for the system (the rest was dedicated to the computer center's batch system), at most 16 user processes could exist, even if they were idle. Process swapping would increase this to around 50.

Slow

A user perceived the system as slow in at least two ways. The first was during the execution of a moderate size program. For example, a typical 50 page assembly on CAL TSS, using the CDC assembler running under the SCOPE simulator, required about two and one half times as much CPU time as under the real SCOPE system. The second was the time required to start a program. For example, with no other users on the system it required about 15 seconds of real time to start the SCOPE simulator, assemble a null program with the assembler and return.

The major contribution to the system CPU cost for running a program was from disk file I-O, either explicit file reads and writes, or implicit via placement in map entries.

Shortly before the termination of the project, a small test program was written to investigate the disk file I-O speed problems. This program read data from one file, did a small amount of computation, and then wrote data onto an output file. It wrote as many words as it read, and computed for about 50 microseconds per word. This test program was able to maintain a transfer rate, to and from the disk, of about 6K words per second. (A similar program, running alone on the real SCOPE system, could transfer about 10K words per second. Due to disk conflicts on the real SCOPE system, two such programs running simultaneously could maintain a combined rate of less than 5K words per second.)

This test program was run only a few times, and gave results varying by almost a factor of 2. Due to the termination of the project, no improvements on these numbers were obtained. Hence, the figures in the following discussion are very approximate.

The system CPU costs for the test program run under CAL TSS were over 70 microseconds per word. (Hence, the high CPU costs for running under CAL TSS.) These CPU costs were caused by the computation necessary to move each data block to or from the disk. (This corresponds to the computation to move a page in other systems.) This amounted to approximately 15 to 25 milliseconds per block, divided about as follows:

1/4 millisecond	ECS system time on behalf of user (ECS system time spent moving data between ECS and CM, in response to user program requests)
2 milliseconds	ECS to CM swap time (ECS system time spent swapping process memory between ECS and CM. Principally during disk system calls from user code, and when the process blocked waiting for disk I-O to complete.)
4 milliseconds	non ECS system, system time (Time consumed by the disk system, viewed as a user program running on the ECS system.)
12 milliseconds	ECS system time, on behalf of disk system (Time consumed by the ECS system in response to requests from the disk system: principally general "book-keeping" by the disk system, and sending disk I-O requests to the ECS system disk driver.)

The total ECS system time required to communicate with the ECS system disk I-O driver adds up to about 3 milliseconds. The rest of the ECS system time occurring on behalf of the disk system (about 9 milliseconds), must be bookkeeping overhead within the disk system itself. Other information indicates that about half of the disk system ECS system time was spent on

event channel sends and receives and half on ECS file reads and writes. (We eventually expected to reduce this cost by giving the disk system direct machine instruction access to a portion of ECS, which would have reduced the ECS system time to about 6 milliseconds. See Chapter 18.)

One final remark on this test: the test program called on the disk system only once per 16 blocks of data. If it had called once per block, the overheads would have probably been substantially higher. (Unfortunately, more exhaustive tests were never made, and this conclusion was never verified.)

A second point where the system was slow was during user subprocess construction. This would generally occur in response to a command typed to the command processor by the user. This would result in a flurry of activity at the command processor level. First the name of the subsystem had to be looked up in a directory, rather, in a succession of directories. Then several names of files needed by the subsystem itself had to be looked up. Each of these directory references resulted in disk file actions to read the necessary information. Finally some scratch files had to be constructed for the subsystem. In sum, on a system with only one user, it generally required between 5 and 15 seconds of real time before the subsystem itself was ready to begin. (We considered installing an associative window for the directory references, but never began a serious design. It is not clear how much this would have helped.)

Difficult to use

The main difficulty for the average user resulted from a multiplicity of naming conventions. One naming convention was a leftover from an early experimental system, since the software originally written for it had not been changed. Other naming conventions resulted from the fact that in the final system the user had to be cognizant of at least three directories that might contain the file he wanted, his temporary directory, his permanent directory, and a system directory. We supplied one naming facility which gave full access to his permanent directory, and another which did not, in order to protect him from undebugged subsystems (a protection not generally provided in other operating systems).

Finally, a user who desired to write his own subsystems was in severe difficulty, since we had no complete manuals covering all of the conventions he had to know. Consequently the only successful subsystem writers were our own staff and a few determined and inquisitive users.

CHAPTER 18: SPEED UPS

As indicated in the previous chapter, the system overheads were quite high (e.g., on the order of 20 milliseconds per data block transferred from the disk). In this chapter we will consider the possible effects of some improvements we had intended to implement. In the next chapter we will consider possible hardware improvements.

We had two intended improvements, both to the ECS system:

Direct ECS Access

> A subprocess could select a single data block in a single ECS file and be given direct hardware access to its contents.

Fast Actions

> A small number of ECS actions would be recoded in an ad hoc, but hopefully more efficient, manner.

It is probably that these two changes, together with a minor change in the disk system, would have reduced the per block overhead for disk transfers to the order of 10 milliseconds for multiblock transfers.

Direct ECS access

The CDC 6400 hardware has a base and bounds register which controls access to ECS (in addition to similar base and bounds registers for CM). In the system as implemented, the ECS bound was always set to zero while user code was running. After solving some minor bookkeeping problems, we could have given a subprocess direct access to a single file data block through this single base bounds pair. Thus the cost for each ECS file access to such a data block would be reduced from about 300 microseconds to about 3 microseconds.

Fast actions

The execution of an ECS action had three major steps:

1) enter and leave the system
2) find and check types of all parameters
3) perform the action

Steps 1 and 2 together consumed from 200 to 250 microseconds for most actions. For many actions step 3 consumed about 100 microseconds. One of the parameters which had to be found and checked was the operation (an object), which specified the action. This consumed from 40 to 50 microseconds. (The next chapter will contain a more detailed description of parameter fetching.)

We proposed to invent a new type of capability, that for a 'fastaction'. All capabilities for such actions would contain a pointer to code which would fetch and check the parameters, and then execute the action. We expected that this together with some miscellaneous improvements in the code for the actions would reduce the CPU time for such actions by 100 to 150 microseconds, a reduction by about 50%.

Effect on the disk system

For the system as a whole, ECS system action times were about evenly divided in the following groups:

read and write files
send and receive events
call and return from subprocesses
miscellaneous

If we assume that most of the file and event channel activity was on behalf of the disk system (which we suspected but never attempted to prove), and observe that the disk system rarely made subprocess calls, then we see that a substantial portion of the disk system time is spent on files and event channels. Most of the disk system's file actions were directed to a single file which contained its global data base. If this file were made directly accessable, great saving would result. (The disk system was coded so that only a re-assembly was necessary to take advantage of the direct access.)

This we expected to reduce most disk system ECS file action times by about 90%, the remainder by about 50%, and event channel times also by about 50%. The approximate figures in Chapter 17 indicate that, of the 18 milliseconds of CPU time required by the disk system for a single block transfer, 12 milliseconds are consumed in ECS system actions. The above figures suggest that Direct ECS access and fast actions will reduce this to about one third, or 4 milliseconds. Thus the per block overhead for disk transfers would reduce to about 10 milliseconds, for multiblock transfers.

CHAPTER 19: HARDWARE HELP

As has been indicated, some of the more frequently used ECS actions spend half of their time obtaining and checking parameters (e.g., ECS file read and writes, and event channel actions). This chapter contains a proposal to reduce this overhead. It is a hardware capability mechanism, similar to that in MAGNUM and System 250, with the addition of type information. Before examining the proposal, we first examine the details of obtaining and checking parameters in CAL TSS.

Sketch of entry to a system action

When the existing system is entered in response to a user call, the basic information presented to the system is a pointer to an input parameter list (IP list), in the user's address space. This list contains information which defines the objects involved in the action, including the operation itself as the first object. There are two means of defining an object, direct and indirect. A direct definition consists of an index in the local C-list of a capability for the object. An indirect definition consists of an index in the local C-list of a capability for another C-list, and an index in that other C-list of a capability for the desired object.

Many base bound pairs

Figure 6 diagrams the situation for a system call with an indirect reference to the operation, a direct reference to an object and a single datum. There are 10 pointers involved, each of which is relative to some implied base address and must be checked against some implied bound. Furthermore, there are three capabilities involved, each of which must be checked for the correct type and sufficient access bits, as well as continued existence of the object (correct unique name in the appropriate MOT entry). Finally, any data to be manipulated within the defined object will be addressed relative to a base address (the ECS address of the object) and checked against a bound (the size of the object).

On the CDC 6400, given that the necessary data is already in the central registers, the time required to access a word in central memory by a pointer relative to a base bounds pair is twice that required to follow a direct pointer. Thus, the addition of hardware instructions which provide memory access by pointers through base bounds pairs could greatly reduce system overhead. However, additional time would still be spent checking types, access bits and unique names.

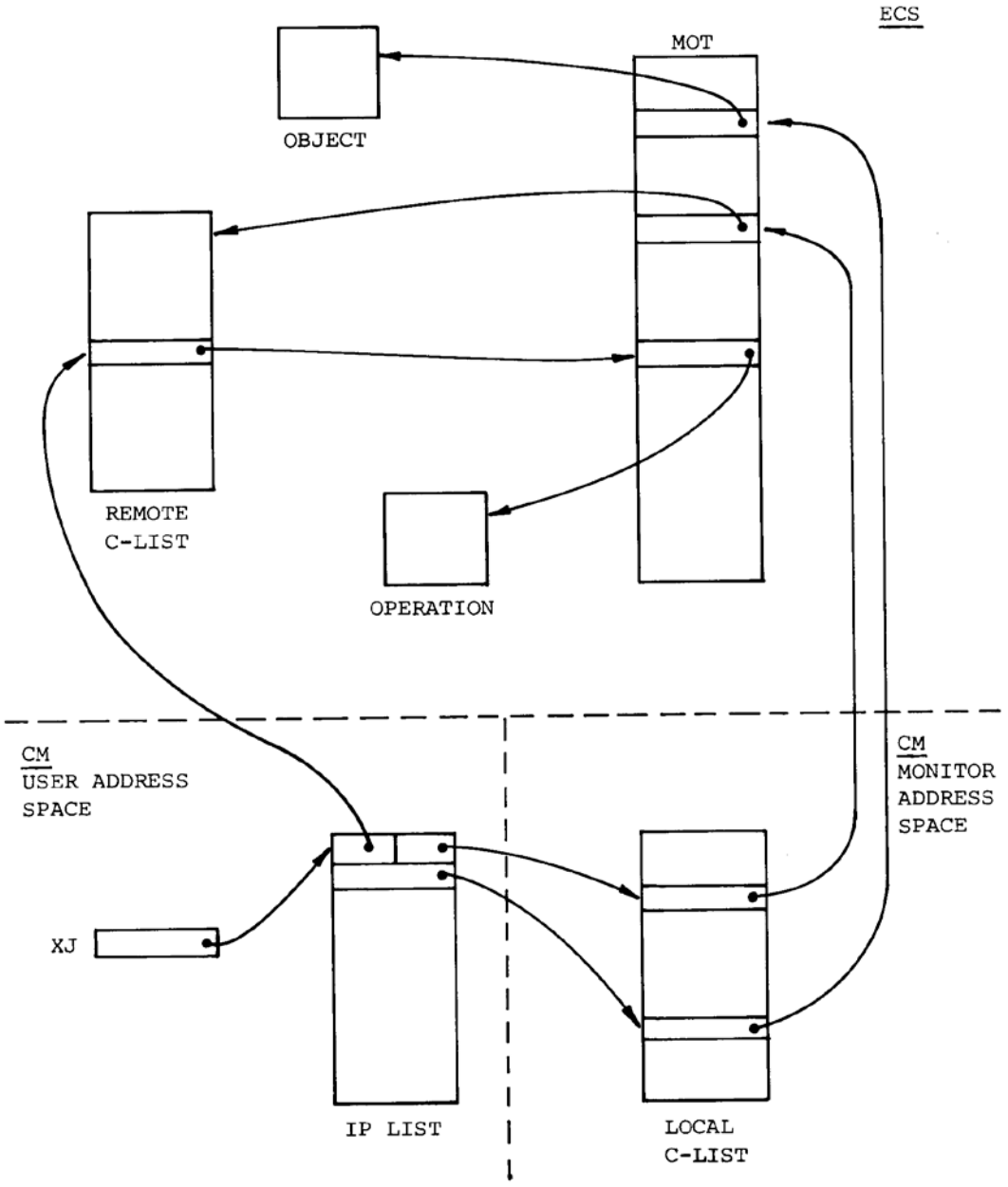

Figure 6. Pointer structure for a typical ECS system call, with an indirect reference to the operation.

Hardware capabilities

The following hardware proposal combines the protected-base-bound-pairs of a system like MAGNUM or System 250 with the abstract typed objects of CAL TSS. In particular, it permits a program with proper authority to directly convert a capability for an abstract object into a capability for a base-bound-pair describing the storage area for the representation of the object. It combines the basic capability hardware of MAGNUM [F1], the MOT of CAL TSS (similar to an idea in System 250 [C4, E1]), the user type facility of CAL TSS, and a proposal of David

Redell and Bruce Lindsay (the lock facility, intended by them for software implementation of such features of CAL TSS as the subprocess-descriptor).

The Central Processor will have an architecture closer to that of MAGNUM or System 250 than that of the CDC 6400. There will be two classes of *registers*, *capability* and *data*. There will be four classes of *instructions*:

capability memory ref
data memory ref
data manipulation
capability manipulation

A *capability-register* will contain three fields:

type
access bits (These correspond to option bits in CAL TSS capabilities.)
datum

There will be a number of *types*, six of which will be:

capability segment (C-list)
data segment
datum
create new lock and key authority
lock
key

For capability segment and data segment capabilities, the datum field will contain two subfields:

base
bound

For lock and key capabilities, the datum field will contain three subfields:

source type
target type
access field check bits

Capability memory reference instructions

Capability memory reference instructions will either load capability registers from memory, or store their contents into memory. The actual memory address will be computed in two steps, first an index is computed by the program and left in a data register, then the actual address is computed from this index and the contents of a capability register specified in the instruction. This specified capability-register must contain a capability of type capability-segment, and its base will be added to the given index. (If capabilities occupy more than one word of memory, the index will be suitably modified to guarantee that the resulting address will be that of the first word of a capability.) The index will also be checked against the bound. If a load is to be done, the capability-segment capability must have the load access bit on. Similarly, if a store is to be done, the store access bit must be on.

In order to avoid having absolute addresses within the capabilities (as stored in memory), and to remain close to the CAL TSS ECS system, the same trick as used in the System 250 can be used. Instead of a base bound pair, a segment type capability, stored in memory, will contain a

unique name and an index into an MOT. The indexed MOT entry will contain the same unique name (for checking purposes) and the actual base bound pair. When a segment type capability is loaded into a register, the MOT index will be followed, the unique name checked, and the actual base bound pair loaded. In this case, such a register will have to contain the original unique name and MOT index for subsequent storage. (Of course, this MOT index and unique name need not be kept in hardware registers, but in a known part of memory, as in the System 250.)

On a machine like the CDC 6400, with an extra memory, any base bound pair could state which memory contained the address, thus reference to either memory would be the same. This would be true for both data segments as well as capability segments.

Data memory reference instructions

Data memory reference instructions will either load data registers from memory, or store their contents into memory. As for capability memory reference instructions, the actual memory address will be computed from a program computed index and the base bound pair in a specified capability register. This specified capability register must contain a capability of type data segment, with the appropriate load or store access bits on.

Data manipulation instructions

These are the standard bit manipulating and arithematic instructions seen on all computers. They work on data stored in the data registers.

Capability manipulation instructions

There are four subclasses of capability actions:

reduce access bits
read or write datum part of a datum capability
lock or unlock a capability
create a new lock and key

The last three subclasses represent a departure from previous hardware capability systems.

Reduce access bits

Any capability in a capability register may be replaced by a capability identical to the original, except that some access bits are turned off which were on in the original. Thus access available through a capability may be reduced before passing it to less privileged programs.

Read or write the datum part of a datum capability

Any program may explicitly read the datum part of a datum capability. Similarly, any program may arbitrarily replace the datum part of such a capability. For example, a re-implementation of the CAL TSS disk system might place the disk address of a disk file in a datum capability. (The capability must then be locked, as described below, before passing to a user program.)

Lock or unlock a capability

A capability whose type matches the source-type subfield of a given *lock*, and has at least the same access bits on as in the access field check bits subfield of the lock, may be converted to a

capability with:

> those *access-bits* on which are on in the given lock,
> *type* as given in the target-type subfield of the given lock,
> *datum* part unchanged.

This is referred to as *locking* the capability with the given lock.

Similarly, a capability may be *unlocked* if its type matches the target-type subfield of a given key. We also require that the access bit field of the capability have at least the same bits on as the access field of the key. The resulting capability will have:

> *access-bits* as in the access field check bits subfield of the key
> *type* given as source-type in the key
> *datum* part unchanged

Thus, the condition:

> any two capabilities, of type either lock or key, with the same target type, have identical source types and access field check bits;

will assure that any capability resulting from an unlock operation will have previously existed, possibly with more access bits on. (We explain below how to assure this condition.)

If the MOT-index unique name representation for segment capabilities is used, then a locked segment capability will contain only the MOT-index and unique name. Unlocking such a capability will cause the base-bound pair to be re-computed.

Figure 7 contains an example of locking and unlocking a capability.

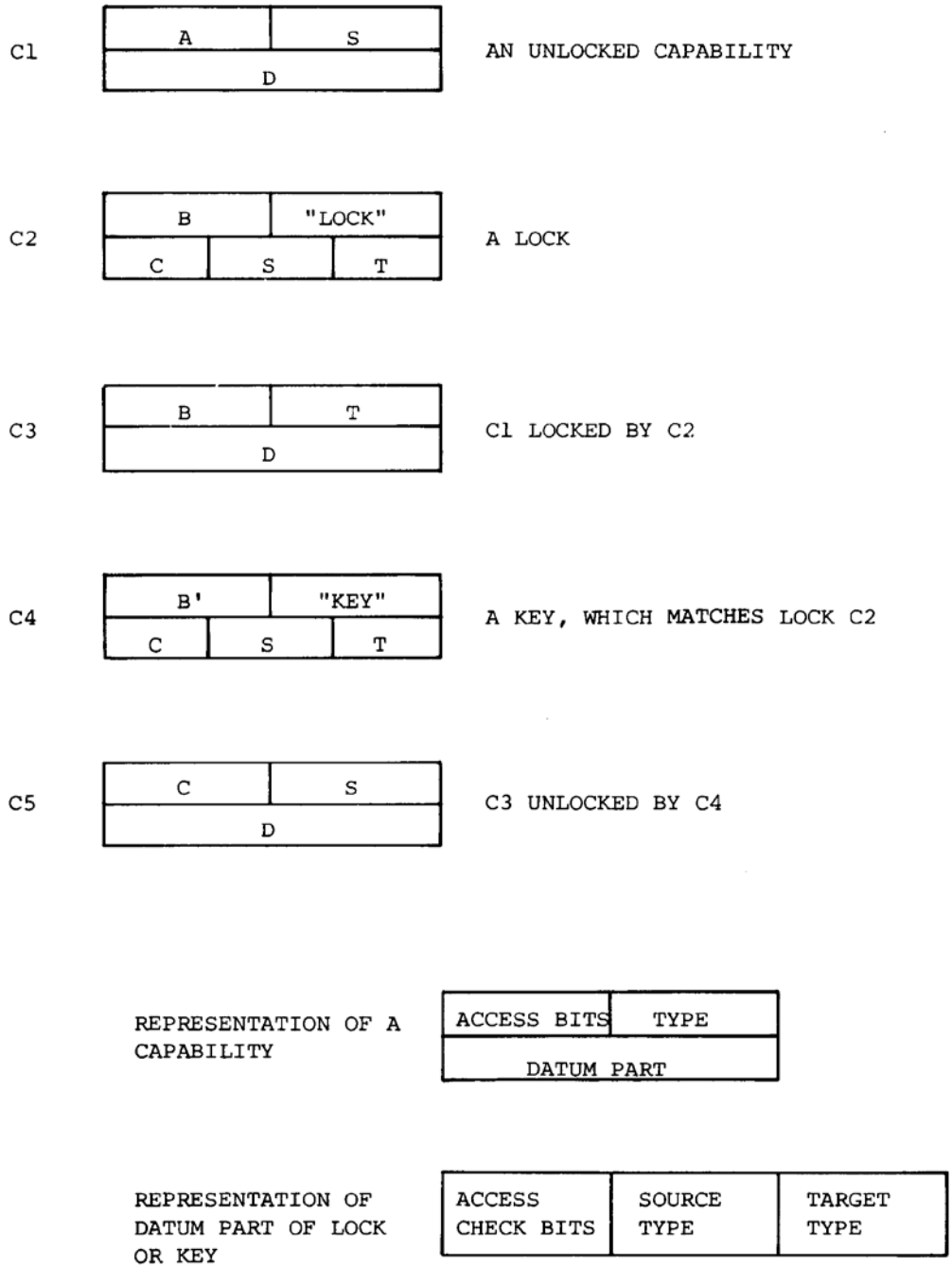

Figure 7. Example of locking and unlocking.

Create a new lock and key

Using a create-new-lock-and-key authorization, and presenting a source-type representation and a set of access-field-check-bits, a program may obtain two new capabilities. One will be a lock, the other a key, otherwise they will be identical. The target-type subfield of each will be a new, never before seen type. The source-type and access-field-check-bits will be as given. All access bits will be on in both.

Any lock or key capabilities created subsequently, with the same target-type, must be copies of these just created capabilities. They will have the same source type and access-field-check bits, since all copy operations leave the datum parts of a capability unchanged. Thus, the condition required above is met.

Example

In order to explain the use of these hardware features, we explain how the ECS system of CAL TSS might be re-implemented.

The state of each object will be represented in one or more segments. A single data segment might suffice for some simple objects, e.g., a file with a single data block. More complicated objects would be represented by a capability segment with capabilities for subparts. A locked capability for the representing segment is then given to user programs.

A user program will call the system with parameters in hardware registers. Upon entry, the system will pick up the appropriate keys and unlock each expected parameter. In one step, this will check for correct parameter type, sufficient access bits, correct unique name at the specified MOT index; and produces a segment descriptor for the representation of the object.

CHAPTER 20: A REPLACEMENT FOR OUR MAP FACILITY

Our attempt to provide a mapped address space was probably the worst disaster of the project. In this chapter we discuss the manifestations of that disaster, and then suggest an alternative.

Disaster

Our attempt to provide a mapped address space required a lot of very complicated code, and it failed. The complicated code was most evident in the disk system, but also was evident in our complicated file structure and multi-level operations. We failed because we did not correctly simulate a mapped address space.

The decision to simulate portions of disk files residing in ECS by ECS files led to structuring ECS files as a sequence of blocks, only some of which might exist. (If ECS files had been merely consecutive blocks of storage, they would have been much simpler.) Multi-level operations were supplied to permit an unsuccessful ECS file read or write action to automatically initiate a more expensive disk file action. This was the only place where multi-level operations were actually used. The major complications were in the disk system, which maintained complicated global tables recording which portions of which files were attached by which process.

We were successful in our attempt to make ECS files appear to represent the ECS portions of disk files. However, the main intention in providing this facility was to make possible subprocess maps which pointed into disk files.

The subprocess map facility failed, even at the simple level of maps into ECS files. This failure occurred under the following conditions:

i) two subprocesses in the same process attempt to share some data by mapping into some *common* file, and

ii) they both can be in CM simultaneously (one is a descendent of the other), and

iii) at least one of them is permitted to modify the data.

Under these conditions, a subprocess has to explicitly read and write the common data in order to insure that his copy is correct. This totally defeats the purpose of the map.

The reason for this failure is quite simple: the hardware did not provide address mapping. We attempted to simulate mapping by copying the mapped data into the appropriate regions of central memory. Thus, there could occur two independent copies in central memory of the same region of a file.

This same problem can occur under other conditions:

a) Two different regions in the same subprocess address space map to the same region in some file. Then a change in the data in one region will not be immediately reflected in the other.

b) If the system were run on dual 6400's with a single ECS, and subprocesses in two different processes map into the same file. In this case, the problem can be prevented by a potentially very complicated mechanism which prevents the two subprocess from swapping into different CM's simultaneously. (We intended that our system might eventually run on the two 6400's owned by the computer

center.)

c) Two subprocess in the same process, one maps into a file, and the other attempts direct reads from the file. If one is a descendent of the other, the problem appears.

d) Same as c, except the subprocesses are in separate processes, but under a dual 6400 system.

An alternative

This alternative I am about to describe was considered at the start of our project, but rejected as not general enough. It acknowledges the one successful use of maps, the use of shared, read only programs. All other uses are discarded.

There will be two forms of data storage:

i) ECS files
ii) disk files

An ECS file will be a simple sequence of words, there will be no division into data blocks. A disk file will be a simple sequence of equal sized records, each of which is a sequence of words. (This directly reflects the physical structure of the disk.) Actions will be available to read the contents of one or more consecutive, complete, disk file records into continguous words of an ECS file. There will be a similar action to write consecutive disk file records from an ECS file. There will be separate actions to transfer consecutive words between an ECS file and the local memory of a subprocess.

A subprocess local memory will be specified in two parts. (The vestiges of a map.) Each part will be an entire ECS file. When a subprocess is swapped into CM, the two ECS files will be copied into CM; and when swapped out, one of the files will be copied back to ECS. Thus one of the files contains read only code, while the other contains data local to this version of the subprocess.

CHAPTER 21: DISTRIBUTED SYSTEM CODE

The design of the outer layers of CAL TSS (disk/directory and command processor) depended heavily on distributed system code. I feel this was a mistake, and that there was a better alternative.

In this thesis I have used the phrase "distributed system code" to describe the following idea: global data (which represents the state of the virtual machine) that will be manipulated by code which runs in protected domains (subprocesses) within a user process; several instances of this code may, in principle, run simultaneously. This idea has been around for a long time, and a detailed account of its use may be seen in Saltzer [S1].

It is difficult to argue persuasively for distributed system code. Saltzer gives the argument that distributed system code makes it easier to provide a different appearing system for each user. An individual user's process would contain that version of the system code which he desired. (I would view this as providing different sets of virtual instructions.)

In practice it is difficult to take advantage of this idea. If this portion of the system code is in fact part of a distributed system and manipulates global system data, then it is sensitive system code. Hence it must be checked out by whatever painstaking methods are used for all system code. Consequently it is unlikely that two versions of a portion of the system would be completed.

A major problem introduced by constructing a distributed system is that of interlocks. Since there are many representatives of the system attempting to read and modify some global data, they must avoid interfering with each other. (For example, while one process is reading, adding one, and rewriting a count, another process may attempt the same action.) At the worst, if the interlocks are designed carelessly one may be confronted with a 'deadly-embrace'.

Examples of the difficulties one can get into are provided by the disk system, our major attempt at a distributed system. The majority of the CPU time consumed by the disk system was in calls on the ECS system. We have evidence that about one half of this time was spent on ECS file read and write actions, with the other half going to event channel actions. At least half of these event channel actions must have been involved with interlocks. Thus on the order of 25% of the CPU time spent by the disk system was involved with interlocks.

As an extreme example, at one point in the development of the system, we discovered that while one disk system representative had an item of data locked, two others could get into a loop asking each other for permission to use the locked data. This was, of course, fixed, but demonstrates that much care must be taken.

I believe that we would have been better off to avoid distributed system code, at least in as complicated a form as the disk system. (The proposal made in the preceding chapter would have relieved us from providing the disk system.) Necessary global data bases could be manipulated by dedicated processes, which receive coded instructions from event channels. Any interlocking of modifications to the data base would then be purely internal to the process, and thus simpler to achieve.

CHAPTER 22: SUMMARY AND PARTING WORDS

Summary

In this thesis we have discussed an operating system project at the University of California, Berkeley, Computer Center. We have discussed many of the initial ideas and objectives of the project, the actual system constructed, and a number of reactions to that resulting system.

The project was modest in size, involving about 30 man years. Except for a necessary item of peripheral hardware, the system was designed for a commercial computer, the Control Data 6400 with an Extended Core Store (ECS).

We attempted to include a number of fundamental ideas in our design, including:

specification of the entire system as an abstract machine,
a capability based protection system,
a mapped address space for each virtual computer (process),
layered implementation including distributed system code,
and uninterpreted input output devices.

It was intended that many of the services that must be supplied by a complete system would be provided by "user" level programs, whose special needs would be provided by system features. These features would be made available to "ordinary" user programs, as a matter of principle.

The system made heavy use of ECS, a large core store with a fast block transfer rate (10 60 bits words per microsecond) to central memory. The state of each user process was stored in ECS, copied to central memory for execution and then copied back to ECS. The state of the system was maintained as the state of abstractly defined objects, whose representation was stored in ECS. Also, user programs accessed all real input output devices through the state of these abstractly defined objects.

The system was designed in layers. The first layer (ECS system) provided 8 types of abstractly defined objects and about 100 actions to manipulate them. Subsequent layers provided a few (but very complicated) additional types of objects. (The entire second part of this thesis is devoted to a description of these various layers, and the abstract machines they define.)

My major reaction to the resulting system was that it was disappointingly large, complex and slow. (Possibly this was due to naive expectations.) Many of our fundamental ideas served us well, particularly the concept of an abstract machine and capability based protection. However, mapped address space (on an unsuitable machine) and distributed system code were unsuccessful.

Part 3 is devoted to an elaboration of my reactions to the system, along with a number of proposed improvements which would have mitigated some of the difficulties. In particular, I include a hardware proposal (a modification of ideas used on other projects) which would have greatly increased the efficiency of our capability based protection and abstract machine implementation.

Parting words

There are two firm beliefs that I have acquired from this project:

i) The concepts of designing the system as an abstractly defined machine together

with capability based protection were extremely useful and I strongly advocate their future use;

ii) The concepts of distributed system code and mapped address spaces introduced many complications unnecessary for the provision of an adequate time sharing system, and I would recommend at most limited use of these ideas in future systems.

BIBLIOGRAPHY

[B1] Bensoussan, A., Clingen, C.T., and Daley, R.C. The Multics Virtual Memory: Concepts and Design, *CACM*, Vol. 15, No. 5, May 1972, pp. 308-318.

[C1] Control Data Corporation. 6400/6600 Computer Systems, Reference Manual, Publication No. 60100000.

[C2] Control Data Corporation, 6641-A ECS/MASS Storage Adapter. Publication No. 60334200.

[C3] Corbató, F.J., Saltzer, J.H. and Clingen, C.T. Multics – The First Seven Years, *SJCC*, 1972, pp. 571-583.

[C4] Cosserat, D.C. A Capability Oriented Multi-processor System for Real-time Applications, *ICC* Conference, Washington, D.C., Oct. 1972.

[CS] Computer Center, CAL Time-Sharing System Users Guide, University of California, Berkeley, Nov. 1969.

[C6] Computer Center, CAL-TSS Internals Manual, University of California, Berkeley, Nov. 1969.

[D1] Dennis, Jack B., and VanHorn, Earl C. Programming Semantics for Multiprogrammed Computations, *CACM*, Vol. 9, No. 3, March 1966, pp. 143-155.

[D2] Dijkstra, Edsger W. The Structure of the "THE" – Multiprogramming System, *CACM*, Vol. 11, No. 5, May 1968, pp. 341-346.

[El] England, D.M. Architectural Features of System 250, Infotech State of the Art Report on Operating Systems, 1972.

[F1] Fabry, R.S. A Users View of Capabilities, ICR Quarterly Report No. 15, Nov. 1967, The Institute for Computer Research, The University of Chicago.

[F2] Fabry, R.S. Preliminary Description of a Supervisor for a Machine Oriented around Capabilities, ICR Quarterly Report, No.18, August 1968, The Institute for Computer Research, The University of Chicago.

[F3] Fabry, R.S. List-Structured Addressing, Thesis, The University of Chicago. March 1971.

[Gl] Graham. Robert M. Protection in an Information Processing Utility, *CACM*, Vol. 11. No. 5, May 1968, pp. 365-369.

[L1] Lampson, B.W. On Reliable and Extendable Operating Systems, Techniques in Software Engineering, NATO Science Committee Workshop Material, Vol. II, Sept. 1969. (Also published as: An Overview of the CAL Time-Sharing System, Computation Center, University of California, Berkeley, Sep. 1969)

[L2] Lampson, B.W., Lichtenberger, W.W., and Pirtle, M.W. A User Machine in a Time-Sharing System, Proceedings of the IEEE, Vol. 54, No. 12, Dec. 1966, pp. 1766-1774.

[P1] Parnas, D.L. On the Criteria to be Used in Decomposing Systems into Modules, *CACM*, Vol. 15, No. 12, Dec. 1972.

[S1] Saltzer, J.H. Traffic Control in a Multiplexed Computer System, MAC-TR-30, July 1966, Massachusetts Institute of Technology.

[T1] Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., Tomlinson, R.S. TENEX, a Paged Time Sharing System for the PDP-10, *CACM*, Vol. 15, No. 3, March 1972.

APPENDIX A: PROJECT HISTORY

Summer 1968	Time sharing project starts. One faculty advisor and 4 computer center programmers, two half time.
December 1968	Layered structure of system chosen, with responsibilities of each layer determined. Major components of ECS system specified. Actual programing begins. 2 persons from computer science department join project unofficially.
Summer 1969	Portions of ECS system and a temporary executive (Bead) completed. A SCOPE simulator and a text editor written. Public demonstration of system, exhibiting editing, compiling and execution of Fortran programs from two teletypes simultaneously. Disk I-O interface and disk/directory system design begun. Several more programmers join the project.
January 1970	Disk I-O interface completed. A temporary program provided which moves entire files between the disk and ECS. System will support nearly 10 users (editing large text files and assembling them), if they cooperate closely on use of ECS space. No file protection between users. Several persons invited from computer science department to use system on an experimental basis.
Summer 1970	Disk/directory system design completed. Programming begun during spring 1970. Most components of ECS system completed. Final drive begun on remaining portions, mostly I-O interfaces. Some redesign and reprogramming of ECS system started. Command processor design starts.
Spring 1971	Disk/directory system and command processor sufficiently completed to permit use of new system. Bead system scrapped. Many components not completed (disk swapping of user processes, accounting and others).
Fall 1971	Accounting completed. Disk swapping of user processes not completed. System difficult to use, partially due to multiplicity of conventions carried over from temporary (Bead) system. Will support at least 15 simultaneous users of BASIC subsystem. ECS size limits number of users, rather than CPU time.
November 1971	Project terminated for lack of funds.

APPENDIX B: PROJECT MEMBERS

The initial advisor to the project was Professor Butler Lampson. Other official and unofficial members were:

William Bridge
Dr. James Gray
Bruce Lindsay
Karl Malbrain
Gene McDaniel
Paul McJones
Professor James Morris
David Redell
Charles Simonyi
Keith Standiford
Howard Sturgis
Vance Vaughan