GENERAL LECTURE

" The Multics Concept "

Multiplexed Information and Computing System

K

Patterns of man-machine communication:

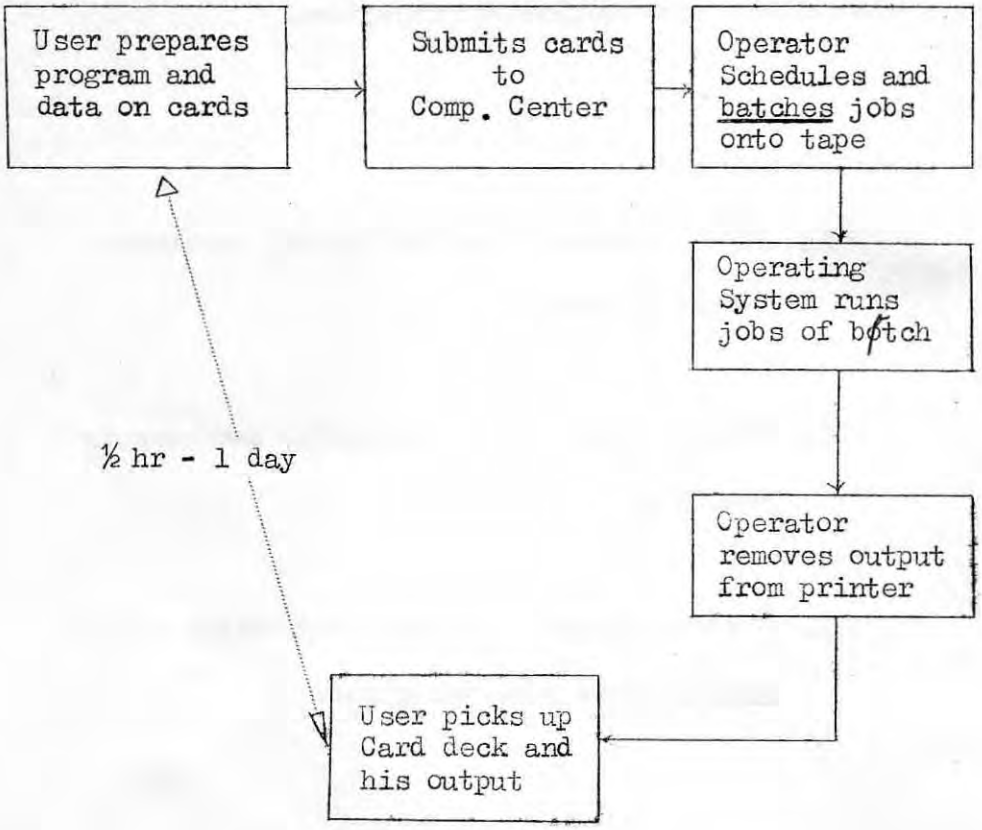
1. B$\overset{a}{\cancel{o}}$tch processing on high speed, expensive computer system

2. Direct interaction with small, inexpensive computer

3. Multics concept: Direct interaction with <u>shared</u> large computer system

X

1. B~~o~~atch processing



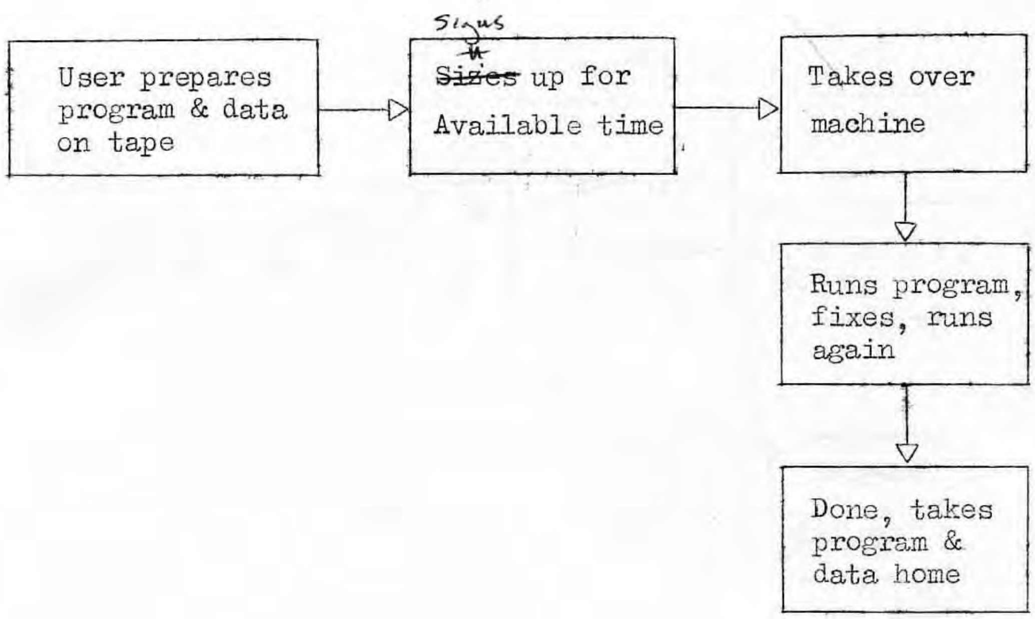
User prepares program and data on cards
Submits cards to Comp. Center
Operator Schedules and batches jobs onto tape
Operating System runs jobs of b~~o~~tch
Operator removes output from printer
User picks up Card deck and his output
½ hr - 1 day

Problem: long turn around

program checkout awkward at a distance (Core dumps)

small mistakes as disastrous as large ones

2. Direct Interaction with small computer:


User prepares program & data on tape
~~Sizes~~ Signs up for Available time
Takes over machine
Runs program, fixes, runs again
Done, takes program & data home

Problems: Uses only small fr~~i~~action of machine (1% in samples).

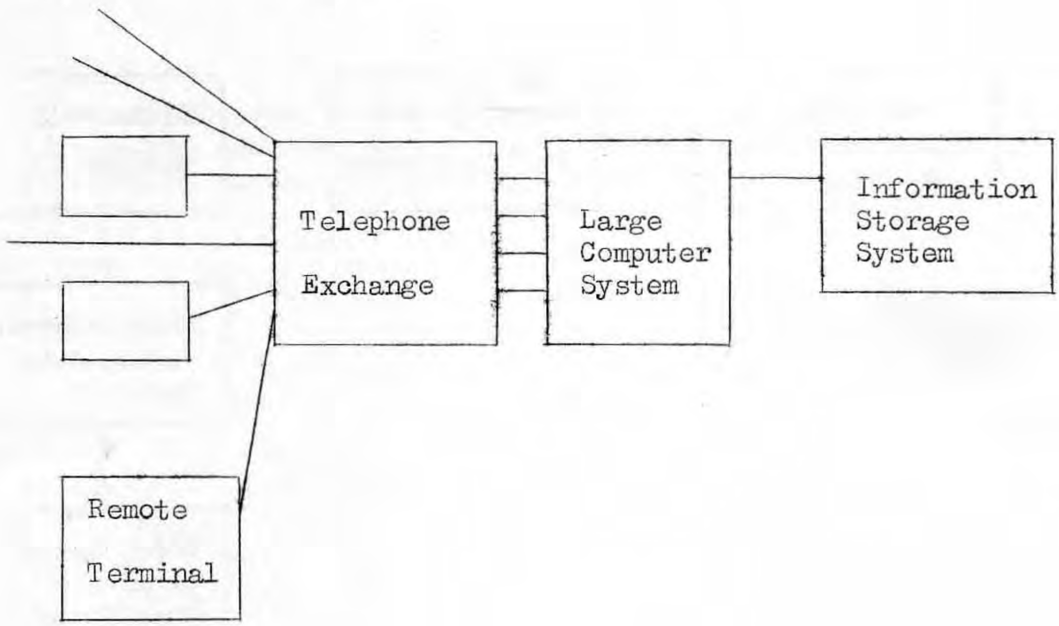
User is under time pressure.

System programs usually of limited value:

debugger & assembler.

An inexpensive computer usually cannot handle very large problems.

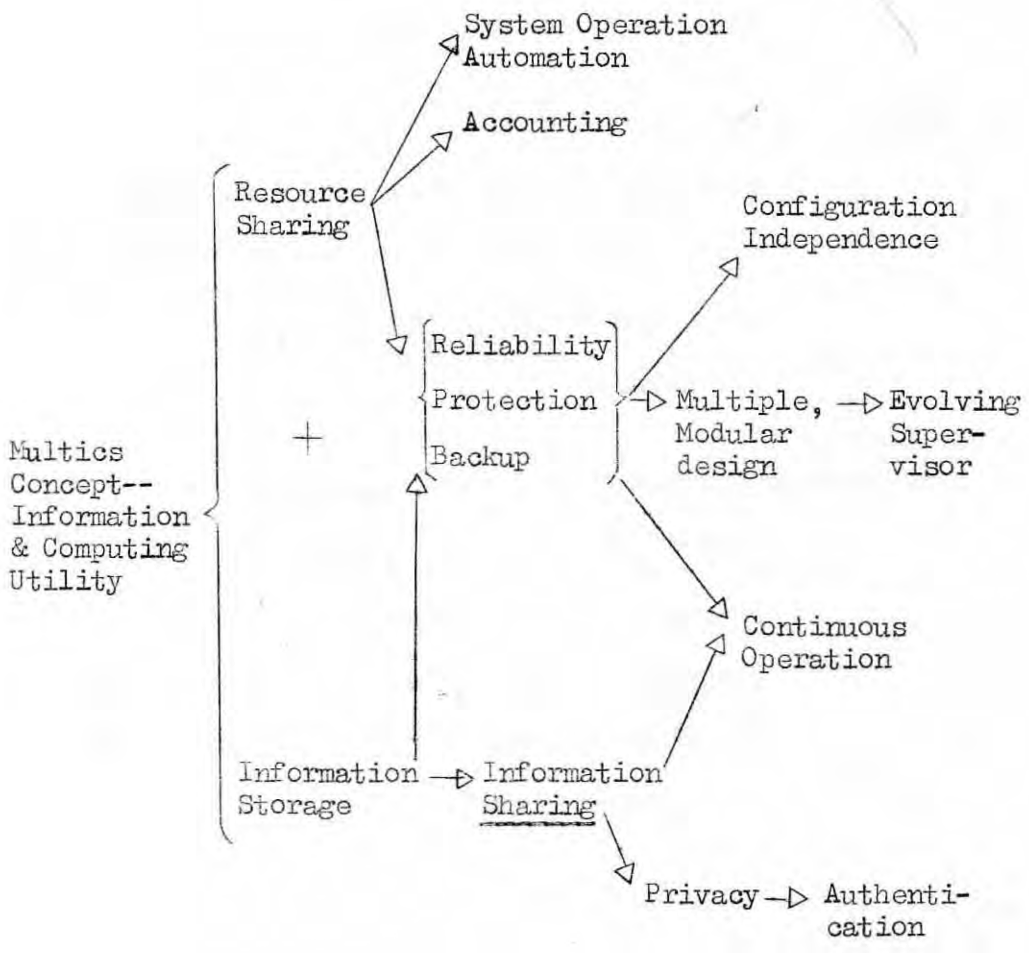
3. Multics Concept:



User dials up computer, places program in system if not already there, and runs it. When done, he leaves it there for later reference. He can use programs of other users.

Requires an elaborate Supervisor Program.

Notions important to the Multics Concept



x

1. Utility -- Notion of service to a variety of customers; available on demand from remote locations. (via telephone switching networks)

2. Resource Sharing -- Technique to lower the cost of a computing and information service. (Analogy to telephone operator or dial equipment) Implies need for protection between users of the system.

3. Information Storage -- The system remembers (by magnetic disk & drum memories) files of information placed in it by a user. Information may be programs or data. The system catalogs all information given it so that it may locate the information later.

4. Information Sharing -- The system can permit different users access to common files of information. Users may thus share programs and dat$\overset{a}{\cancel{e}}$ when desired. Implies need for protection and privacy between users of system.

X

5. Backup -- Since users depend on system to store information, it must be reliable. Technique used is to duplicate all information stored in the system by users on a detachable medium (magnetic tape) and store it permanently. In case of catastrophe, slightly outdated information can be retrieved from backup tapes.

6. Protection -- Users must be protected from mistakes or errors of judgement of other, independent users.

7. Privacy -- Since information can be shared, there must be interlock to guarantee privacy of information to users desiring such privacy. Ideally, system should meet requirements of military security (= company confidential). Implies positive authentication of persons using the system.

8. Accounting -- Since resources are shared, system must keep records of how much of each resource is used by each user, so that charges are fair.

   Objective: 1. Receipts should pay for system.

   2. Charges should be proportioned to service rendered.

X

9. Multiple, Modular Design -- Permits system to continue operation in spite of failure of any single component. (Also allows growth of system capacity to meet needs of users.)

10. Configuration Independence -- (See Multiple, Modular design) Users must not be affected by changes in machine configuration (faster processor, different type of memory, etc.) or else system cannot support continuous operation or evolution.

11. Continuous Operation -- required to complete notion of a utility available on demand. Dependability of service available is essential for acceptance of an information storage system. (Users cannot take work elsewhere if all information is frozen inside a temporarily disabled system.)

12. Evolving System

a) hardware. As technology changes, processors will become faster, intercommunication will improve, I/O channels will change characteristics, and ~~mix~~ mass storage devices will become faster, more accessible, and cheaper. The system must be capable of accepting hardware evolution without affecting users (except to provide improved service).

b) supervisor. The supervisor is never debugged; one always wishes to add new features; superior understanding from experience can suggest more effective techniques. Therefore the system must be able to support both change in the supervisor and dynamic debugging of new supervisor pieces. This is accomplished by organizing most of the supervisor procedures so that they operate in user modes, with all system safeguards in operation.

13. System operation automation -- The system must operate by itself with a minimum of procedures required of human operators. In particular, job scheduling and time accounting must be automatic (with exceptions permitted) -- If an operator can make a mistake, he will --

Specialized lecture

Topics

I Hardware

1. Organization of major system components

2. Segment and Page organization

II Information Storage System

1. Information Storage Hierarchy

2. File System; Segment/File Correspondence

3. Backup/Reload/Salvage

4. Multilevel Storage Management

III Central Supervisor

1. Processes

2. Traffic Controller

3. Interprocess Communication

4. Users; Login and Logout

5. Metering/Accounting/Performance Monitoring

6. Reconfiguration

7. Protection, walls -- rings

IV Command System

1. Shell...Command Language Interpreter

2. Dynamic Linking and Search

3. Command Library

x

V I/O System Organization

VI System Operation

1. Transactor
2. Absentee jobs
3. Peripheral operators
4. Initiali~~c~~zation

VII Development

1. Management of R&D effort
2. Development Techniques

Introductory Lecture:

Organization of the Computer Utility

The term "computer utility" by its very nature implies marketing of a useful resource in a usable form. Although immense computing power, sharable secondary storage, and flexible access to input and output devices are indeed useful resources, the primary function of the computer utility is to organize such resources into a usable, and thereby marketable, form.

From one point of view the marketing of computer resources is much the same as the marketing of candy bars. The man on the street would be quite pleased to purchase his candy bar direct from the factory at the candy jobber's prices. On the other hand, his enthusiasm wanes when he discovers that he must take not one candy bar but a carload, and delivery will require six weeks. In much the same way the ordinary computer user is quite unprepared to tackle the problems of managing several processors, I/O interrupts, and disk track organization, even though his particular problem might require sizable amounts of computer time, input-output, and secondary storage space.

Again using the candy bar example, we observe that the candy bars pass through several hands: the jobber, the wholesaler, the distributor, before they turn up on the drugstore counter. At each of these levels the product of the previous level is transformed into a resource with a wider market. The carload of candy bars is wholesaled in gross cartons; the distributor once a week provides the drugstore with boxes of 24 candy bars. Finally, the man on the street wanders in and purchases just one, whenever he likes. In a very similar manner, we may view the resources of the computer utility as being transformed three times, each time producing a resource that is successively more "marketable":

1. Starting with the basic hardware resources available, the "hardware management" procedures have the function of producing hardware independence. They do so by similating an arbitrarily large number of "pseudo-processors" each with a private segmented address space (which may contain segments shared with other pseudo-processors), easy access to a highly organized information storage hierarchy, and smooth input/output initiation and termination facilities. The resulting resource is independent of details of hardware or system configuration such as processor speed, memory size, I/O device connection paths, or secondary storage organization.
2. Working with these pseudo-processors and the information storage hierarchy, the "resource management" procedures

allocate these resources among "users", providing accounting and billing mechanisms, and reserving some of the resources for management services, such as file storage backup protection, line printer operation, and storage of user identification data.

3. Finally, these allocated and accounted resources can be used by the ultimate customer of the computing utility either directly by his procedures or to operate any of a large variety of library commands and subroutines. Included in this library are a command language interpreter, a flexible I/O system, procedures to permit simple parallel processing, language translators, and procedures to search the information storage hierarchy and dynamically link to needed programs and data.

We now wish to study each of these transformations in more detail.

Hardware Management.

The basic hardware resources available to the utility are the following:

1. One or more identical processors.
2. Some quantity of addressable primary (probably core) memory. The processors are equipped with hardware to allow addressable memory to appear to be paged and segmented. It is not necessary that all possible memory addresses correspond to core locations. One might expect to have 100,000 words of core memory for each processor.
3. A substantial amount of rapidly accessible secondary storage. This secondary storage might consist of a large volume, slow access core memory, high speed drums, disks, data cells, or any combination thereof which proves to be economical. The total amount of accessible secondary storage might be on the order of 100 million words per processor, although this figure can easily vary by more than an order of magnitude.
4. Channels to a wide, in fact unpredictable, variety of input and output devices, including tapes, line printers

and card readers, typewriter consoles, graphic display consoles, scientific experiments, etc. In an installation committed primarily to interactive usage, one might find 200 typewriter channels, plus a few dozen other miscellaneous devices. Each of these channels can produce signals indicating completion or trouble. The signals are transmitted to the system in the form of processor interrupts.

5. Various hardware meters and clocks suitable for measuring resource usage.

The hardware management routines must do two very closely related jobs. First, they must shield the user of the system from details of hardware management. The user should be essentially unaware of system changes such as addition of a processor, replacement of processors by faster models, or replacement of a date cell by an equivalent capacity disk memory. Except for possible improvements or degradations of service quality, his programs should work <u>without</u> <u>change</u> under any such system modification. Second, the hardware management routines must handle the multiplexing of system resources among users in such a way that the users may again be unaware that such multiplexing is going on. Included in this second job is the necessary protection to insure that one user cannot affect another user in any way without previous agreement between the two users.

The strategy chosen here to implement this hardware management is the following. Using the hardware resources listed above and two major program modules, the traffic controller and the basic file system, simulate (by multiplexing processors and core memory) an arbitrarily large number of identical pseudo-processors, and an information storage hierarchy in which data files are stored and retrieved by name.

The information storage hierarchy is a tree-like structure of named directories and files which is shared by all users of the system. Access to any particular directory or file is controlled by comparing the name and authority of the user with a list of authorized users stored with each branch of the tree. This structure allows sharing of data and procedures between users, and also complete privacy where desired.

The pseudo-processors look, of course, very much like the actual hardware processors, except that they are missing certain "supervisory" instructions and have no interrupt capability. Each pseudo-processor has available to it a private two-dimensional address space. Within the address space are a number of supervisor procedures capable of carrying out the following basic actions upon request:

1. "Mapping" any named file or directory from the storage hierarchy into a segment of the address space. Files appearing in the information storage hierarchy are identified by a tree name which is a concatenation of the name of the file within its directory, the name of the directory, the name of the directory containing this directory, etc., back to the root of the tree. As we will see below a utility program named the "search module" may be used to establish the tree name of a needed segment so that the map primitive may be used. The search module itself operates by temporarily mapping directories into addressable storage in order to search them. Use of the map primitive does not imply that any part of the file is actually transferred into core storage, but rather that the file is now directly addressable as a segment by the pseudo-processor. When the pseudo-processor actually refers to the segment for the first time, the basic file system will gain control through missing-segment and missing-page faults and place part or all of the segment in paged core memory. Except for the fact that the first reference to a portion of a segment takes longer than later references, this paging is invisible to the user of the pseudo-processor. The same file can appear as a

segment in the address space of any number of processors, if desired; options allow the processors to share the same copy in core, or different copies.

2. Blocking, pending arrival of a signal from an I/O channel or some other pseudo-processor. A pseudo-processor blocks itself because the process which it is executing cannot proceed until some signal arrives. The signal might indicate that a tape record has been read, that it is 3:00 p.m., or that a companion process has completed a row transformation as part of a matrix inversion.
3. Sending a signal (here known as a "wakeup") to another pseudo-processor or to an input/output channel. (From the point of view of a pseudo-processor, an I/O channel looks exactly like another pseudo-processor.) The wakeup facility, in combination with the ability for pseudo-processors to share segments, permits application of several pseudo-processors simultaneously by a single user. A user may thus specify easily parallel processing and input/output simultaneous with computation.
4. Forcing another pseudo-processor to block itself. This primitive, named "Quit", allows disabling a pseudo-processor which has gotten started on an unneeded or erroneous calculation.

All of these primitive functions are constructed as closed subroutines which are called using the standard call stack described in chapter one.

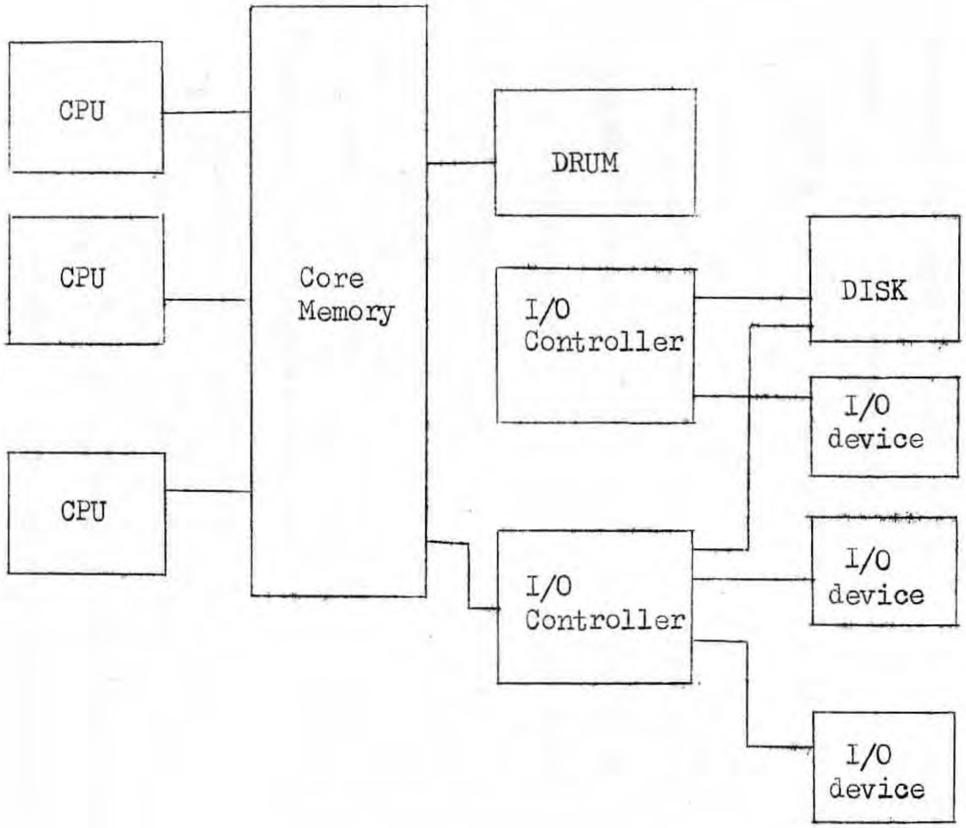
Figure 2.1 shows a typical hardware configuration of the utility, while figure 2.2 indicates the apparent system configuration after the hardware management procedures have been added. An important difference between these figures is that while figure 2.1 may change from day to day (as processors are repaired and a disk is replaced with a drum) figure 2.2 always is the same, independent of the precise hardware configuration.

When a pseudo-processor calls the "map" entry of the basic file system, the file system establishes a correspondence between a segment number of the pseudo-processor address space and a file name on secondary storage by placing an entry in a segment name table belonging to this pseudo-processor. It does not necessarily, however, load any part of the file into core memory. Instead, it sets a missing-segment bit in the appropriate descriptor word in the descriptor segment of the pseudo-processor. This bit will cause the pseudo-processor to fault if a reference is made to the segment.

Sometime after calling the "map" entry, the pseudo-processor may attempt to address the new segment. When it does so, the resulting missing-segment fault takes the pseudo-processor directly back to the segment control module of the basic file system, which now prepares for missing page faults by locating the file name corresponding to the segment number in the segment name table, placing the secondary storage location of the file in an active segment table, and creating in core memory a page table for the segment. This page table is filled with missing-page bits, and none of the file is actually loaded into core memory yet.

The pseudo-processor is then allowed to continue its reference to the segment. This time, a missing-page fault takes the pseudo-processor to the page control module of the basic file system. Page control must locate two items: a space in core memory large enough for the missing page, and the location on secondary storage of the missing page. Establishing a space in core memory may require unloading some other page (possibly belonging to some other pseudo-processor) onto secondary storage. A policy algorithm in the "core control" module decides which page or pages in core are the best candidates for unloading, on the basis of frequency of usage of the pages.

Having established space in core memory for the page, and initiated the transfer from secondary storage, page control



(lines show communication paths.)

Figure 2.1 -- Typical hardware configuration.

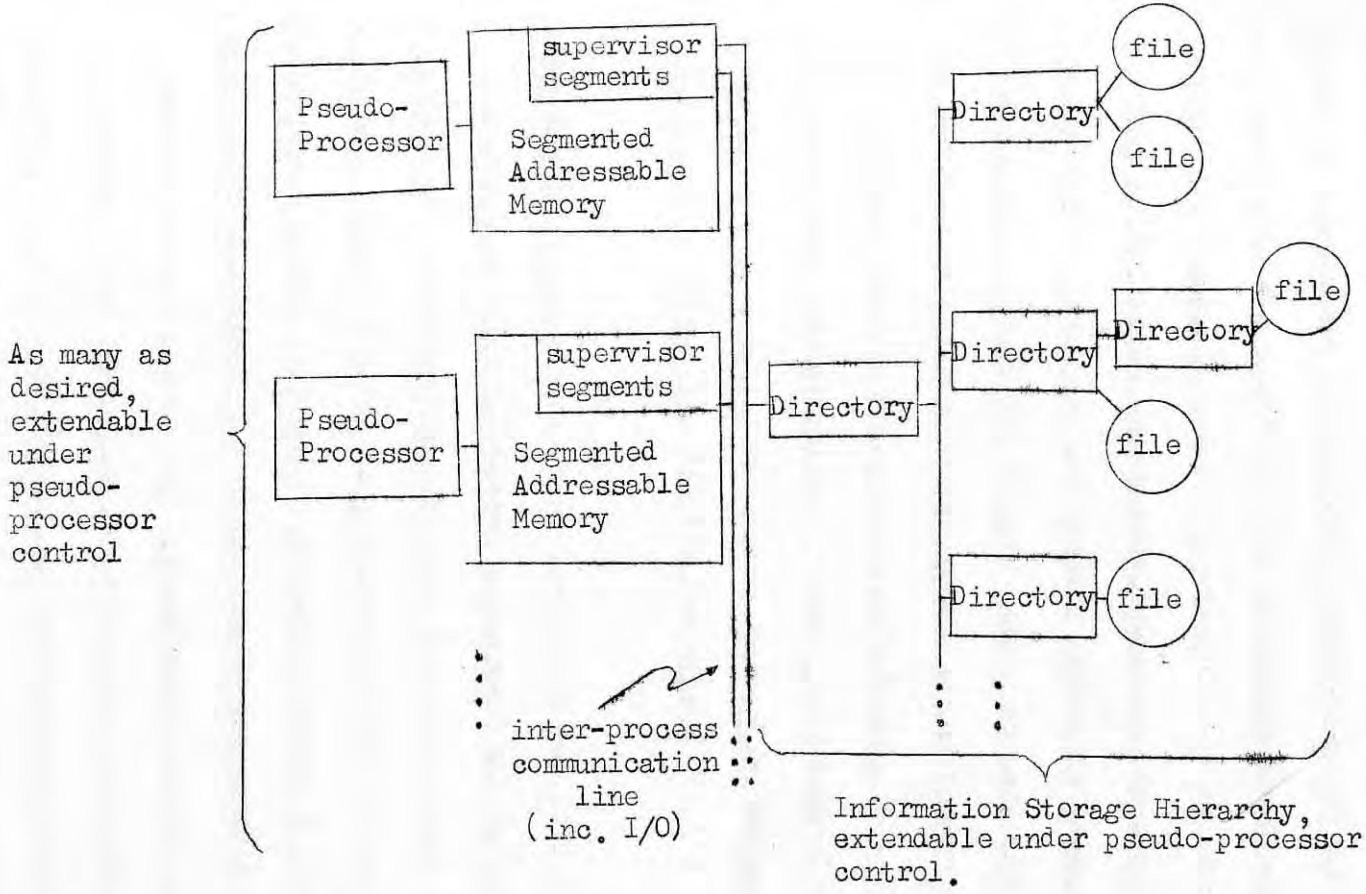

As many as desired, extendable under pseudo-processor control
Pseudo-Processor
supervisor segments
Segmented Addressable Memory
Pseudo-Processor
supervisor segments
Segmented Addressable Memory
inter-process communication line (inc. I/O)
Directory
Directory
Directory
Directory
Directory
file
file
file
file
file
Information Storage Hierarchy, extendable under pseudo-processor control.

Figure 2.2 -- Apparent system configuration after hardware management.

blocks the pseudo-processor pending arrival of the page. When the page is in, this pseudo-processor is re-awakened by the basic file system operating for some other process, page control returns to the point at which the missing-page fault occurred, and the pseudo-processor now completes its reference to the segment as though nothing had happened. Future references to the same page will succeed immediately, unless the page goes unused for a long enough time that the space it is holding is reclaimed for other purposes by core control. If the space is reclaimed, core control sets the missing-page bit in the page table _on_, and writes out the page onto secondary storage. A later missing-page fault will again retrieve the page.

As we will see in chapter four, some segments cannot take part in the paging in-and-out procedure; these segments must be "wired down" (that is, they are not removable) since their contents are needed, for example, in order to handle a missing-page fault. A general property of the file system organization is that a missing-page fault cannot be encountered while trying to handle a missing-page fault. The reason for this organization is not that a recursive missing-page fault handler is impossible to organize, but rather that the depth of recursion must be carefully controlled to avoid using up all of core memory with recursion variables (at least the call stack must go into a wired down segment.) The method chosen here to control recursion depth is to prevent recursive missing-page faults in the first place.

The method of implementing the secondary storage hierarchy, the "map" primitive, and core memory multiplexing has been described in a paper on the basic file system by Daley and Neumann (4) and the reader interested in more detail is referred to that paper. The multiplexing of hardware processors to produce many pseudo-processors is the function of the traffic controller, and is the subject of the remaining chapters of this thesis.

Resource Management.

The hardware management programs transform the raw resources of the computer system into facilities which are eminently more usable, but these facilities must be made available (allocated) to users of the system before those users can accomplish anything. Also, certain of the transformed facilities must be reserved for the system's own use in operation, administration, and preventive maintenance. Finally, a flexible, fair, and accurate accounting mechanism must be provided to determine how and by whom the system is actually being used.

The most important function of resource management is to define the concept of a "user" of the utility. A user, is, roughly, a person, working on a project, who signs out a portion of the system facilities by "logging in." He may work in concert with other users of the system on a single larger project, but his coming and going is independently noted in system logs. The definition of a person working on a project must be relaxed slightly to include the possibility of a so-called "daemon" user (1) which is not directly associated with a person. The definition of a daemon user is that it is automatically logged in to the system when the system is initialized; one cannot identify any particular person who claims to be this user. The daemon generally performs periodic housekeeping functions. (Most daemons, in fact, are creations of resource management, but there are also applications for customer-provided daemons.)

To get the flavor of the techniques used by resource management, we may consider the path followed in logging in from a typewriter console. One pseudo-processor is reserved for a daemon user to which we give the name "answering service". This pseudo-processor is given access to every

(1) "<u>dae·mon</u>, <u>n</u>. in Greek mythology, any of the secondary divinities ranking between the gods and men; hence, 2. a guardian spirit." (Webster's New World Dictionary, 1958.)

typewriter channel which is not presently in use. The process operating on the pseudo-processor activates every attached typewriter channel so that the channel will return a signal when a console dials up, or turns power on in the case of direct connections. The process then blocks itself awaiting a signal from some typewriter channel. When a person dials up to a channel, that channel wakes up the answering service process which immediately brings into play two more pseudo-processors. One pseudo-processor is assigned the typewriter channel and a typewriter management process is initiated on that pseudo-processor. A "listener" process is initiated on the other pseudo-processor. The listener process reads from the typewriter by asking the typewriter manager process for the next line of input. The listener may have to wait if a line has not yet been typed. The listener can take any desired action upon the line, including establishing a process on yet another pseudo-processor to perform some computation. The programs executed by the listener and the typewriter manager come from the library, which is discussed in the next section, so we will not go into any further detail here. Their first action is, of course, to execute the "login" command to establish the identity of the user and his authority to use the system.

Logging in is accomplished by comparing the typist's credentials with a list of all authorized users which is stored in the secondary storage hierarchy. (As we will see, the storage hierarchy is used extensively for administrative purposes.) When a match is found, information stored there indicates this user's access privileges, authorities, and the section of the directory structure in which he keeps his private files. The system log (a file in the storage hierarchy) is updated to show that this user is logged in, and the typist may now begin typing commands.

The point of the description of logging in is to illustrate the techniques used in resource management, not the details. The most important feature of these techniques is that they are based on usage of the pseudo-processors and information storage hierarchy provided by the hardware management programs. They may, therefore, be debugged and replaced while the system is operating, in exactly the same way as any user program. They are also relatively independent of the configuration of the system.

A number of similar operations are carried out by resource management in other areas. For example, a daemon user continually copies newly created files in the storage hierarchy out onto tape for added reliability in case of some catastrophe. Another daemon user periodically wakes up and "checks out the system" by running test and diagnostic procedures. An example of an ordinary user dedicated to resource management is the operator in charge of detachable input and output devices such as tape and disk packs. At his typewriter console he receives messages requesting him to mount reels; he may reply when the reel is mounted or it cannot be found.

Finally, within every address space, certain resource management procedures are inserted in the path between a user procedure and the supervisor routines described under hardware management. These resource management procedures perform resource usage accounting for this process. A system of accounts is maintained within the storage hierarchy, which allows a project supervisor to allocate resources to group leaders who can in turn allocate to individual users. Every pseudo-processor draws on some account in this hierarchy. Also, among the library procedures available to any process are "system transaction programs" which allow the user to arrange special classes of service, sign up in advance for tape drives, etc.

Dynamic Linking, Hierarchy Search, and the Library.

So far, the hardware management procedures have insulated the user from the details of the system configuration and secondary storage management, and resource management procedures have established doors through which a user may enter and leave the system, and have his resource usage accounted for. Before the system is useful to the average user, however, a variety of utility and service (library) programs must be available to him. The library is merely a collection of procedures stored in one section of the information storage hierarchy. This library is built upon the foundations laid by hardware and resource management. It is flexible and open-ended, and procedures drawn from the library operate in exactly the same way as any user provided procedure drawn from elsewhere in the information storage hierarchy.

Fundamental to the usage of the system are dynamic linking and storage hierarchy search procedures. The pseudo-processor provided by hardware management has the capability of producing a _linkage fault_ when a procedure attempts to refer to a segment which has never been mapped into addressable storage. When establishing a new pseudo-processor, one normally places a linkage fault handler in the new address space. When the new pseudo-processor encounters a linkage fault, the linkage fault handler (_linker_) locates the needed segment in the information storage hierarchy by calling the search module. The linker then maps the segment into addressable storage with the "map" primitive discussed earlier, and resets the inter-segment linkage pointer which caused the fault so that faults for that reference will not occur in the future.

By providing an appropriate algorithm to search the information storage hierarchy for needed segments, the user can arrange that a newly established pseudo-processor execute any desired sequence of procedure. The search may, of course, include those sections of the information storage hierarchy containing library procedures provided by the utility.

For example, consider the sequence of linkage faults and searches implicit in the logging-in procedure described earlier. The answering service establishes a new pseudo-processor to run the "listener" process, initially mapping into its address space the standard system linker, a search algorithm which looks at the system library, and a one-instruction procedure which attempts to transfer (through a linkage fault) to a program named "listen". The pseudo-processor is started at the planted transfer instruction. Of course, it immediately gets a linkage fault, and the linker calls the search module to locate the "listen" program. The search module finds a procedure by this name in the system library, the linker maps it into addressable storage, and the transfer instruction is continued. This time it completes execution, and the "listen" procedure is now in control of the pseudo-processor. As it calls on various subroutines, for example to communicate with the typewriter manager process, it gets more linkage faults, and triggers appropriate searches through the library. As needed, the address space of the pseudo-processor collects the subroutines and data segments required to operate a listener process.

An important library procedure is the "Shell", a command language interpreter which is called by the listener to interpret the meaning of a command line typed by the user. The Shell takes a typed command to be the name of a subroutine to be called with arguments, e.g., if the user types the command

PL/I ABCD

the Shell would take this to mean that it should call a subroutine named "PL/I" with one argument, the character string "ABCD". It therefore sets up linkage to a subroutine named PL/I (with a linkage fault in the path, of course) and attempts to call the subroutine. The resulting linkage fault causes a search of the library for, and linkage to, a procedure segment named "PL/I". When the PL/I compiler ultimately begins execution, it will similarly search for and link to the file named ABCD and (presumably) translate the PL/I program found there.

Among the library procedures commonly executed as commands are procedures to help type in and edit new files to be stored in the information storage hierarchy, translators for program files, and commands to alter the search algorithm, for example to search a portion of the hierarchy containing the user's own data and procedure segments. Note that through the mechanism of the Shell, any procedure segment, public or private, appearing in the information storage hierarchy and to which a user has access rights can be called as a command from the console.

Other library procedures include an input/output control system which allows symbolic reference to input and output streams and a substantial measure of device independence. These procedures include necessary inter-process communication facilities required to overlap input/output with other computation.

Through the mechanism of the linker and the search module an arbitrarily elaborate collection of utility programs may be established, yet all such programs are on an identical footing with the user's own programs. That is, they may be checked out and replaced while the system is in operation using the full resources of the system to aid in the checkout. The open-endedness of the library means that it is likely that there will be some users who never execute anything but procedures from the library. It is even possible, through the mechanism of access control provided in the information storage hierarchy, to have a user who, since he has no access to any compilers or input editors, can _only_ execute commands found in some library.

Summary.

We have in this chapter seen a brief overview of several aspects of the organization of a computer utility. In this overview, we have seen how the raw resources of the system are successively transformed, first into configuration- and detail-independent resources consisting of pseudo-processors and an information storage hierarchy, secondly into allocated and accounted resources ready to be put to work, and finally, through a linker, search module, and system library, into a full scale, flexible operating system with a multitude of readily accessible utility procedures. Our overview has necessarily been too broad-brush to go into much detail on how these various techniques are implemented. The reason for the overview has been to give enough of a framework so that we can study in detail the particular problem of processor multiplexing, one of the fundamental aspects of hardware management. Chapter three begins our study of this topic.

Issues

1. Notion of Utility

2. Hardware detail Independence (configuration)

3. Protection and Privacy between users

4. Continuous operation/Reliability

5. Backup

6. System remembers information

7. Automation of all system operations

8. Hardware Management--Resource Management--Library organization

9. Notion of Sharing: Procedure & data

10. Evolving Supervisor / On-line debugging

Smaller Ideas

1. ASCII Character set standard & escape conve$\overset{n}{\cancel{r}}$tions

2. Canonical Form

3. Multi-tasking

4. On-line documentation of system

5. Consoles & Displays (Hardware)

6. Command language Synt$\overset{a}{\cancel{o}}$x

7. User Authentication

8. Tape registry

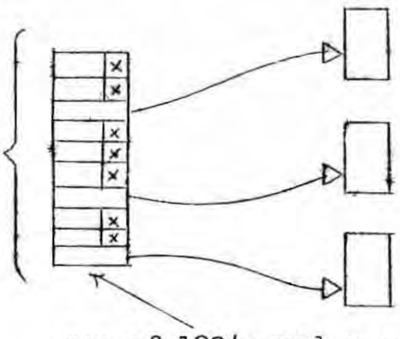
Virtual memory

Problem:

1. Provide programmes with constant system no matter what memory size is used or other users are in competition.
2. Allow programmers a <u>large</u> addressable memory.
3. Share Core memory among many users.

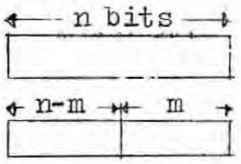
Technique: Virtual memory hardware

View memory thru map



map of 1024-word pages

Processor produces address:



split so:

Done by hardware

a. use n-m ~~or~~ on index of map.

b. use contents of map to replace <u>n-m</u>

If ☒ in map, fault.

Virtues of Virtual memory

1. Permits flexible allocation of core memory blocks on basis of need.

2. Permits Paging, e.g., partial loading of a user's memory, bringing in only those parts needed.

3. Each programmer may have a virtual memory longer than actual system memory.

4. Change in configuration of system memory need only be reflected in core allocation routines. No user is affected.

Information Storage System

Stored on the disks and drums are two kinds of entities:

directories

and files

A file is a string of bits containing information of value to some user; he may interpret it as he likes. A file has a symbolic name.

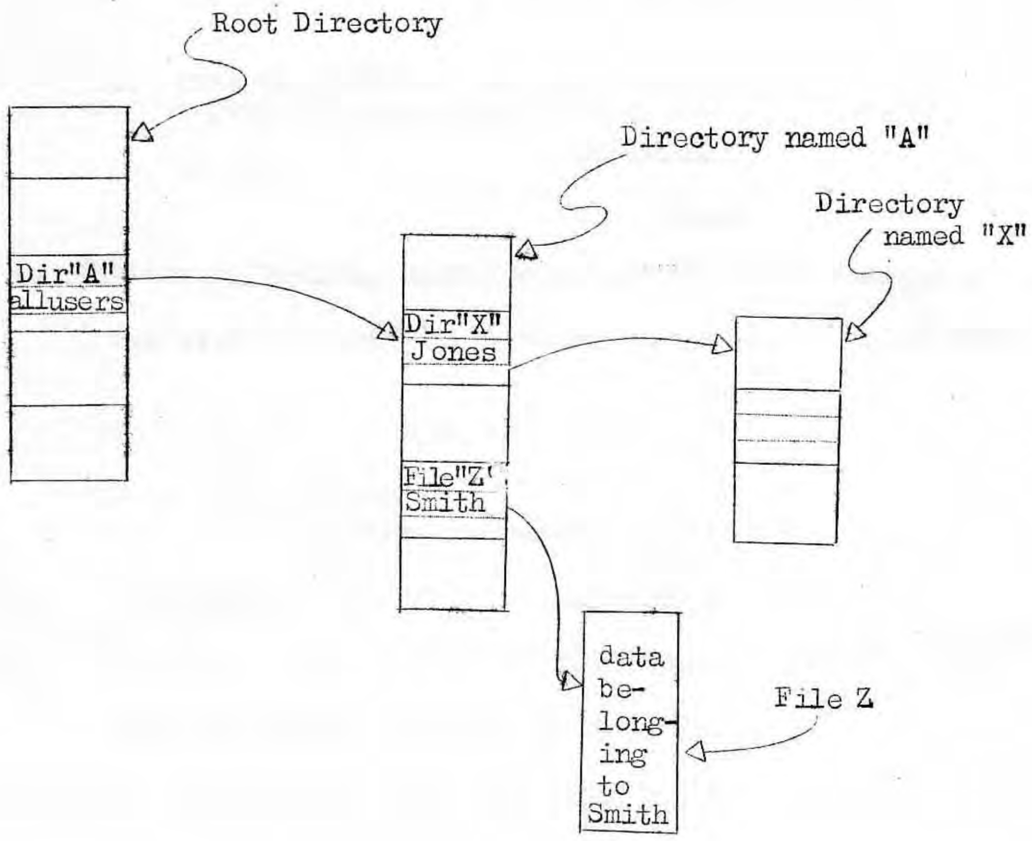
A directory is a list of three-part entries:

1. symbolic name of a file or a directory
2. access control list
3. Pointer(s) to physical location of file
   (indirect them file map) (Disk module 7 track 300
   + module 1 track 107
   etc.)

A directory may also have a symbolic name.

The system always knows the track address of one directory, the root.

All other information in the system is branched from the root directory.



Tree name of file "Z" is Root > A > Z

One can identify any file in the system by giving a tree name.