An Overview of the CAL-Time Sharing System.

Table of Contents

COMPUTER CENTER
201 CAMPBELL HALL
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIF. 94720

1.0 INTRODUCTION

CAL-TSS is a large-scale general-purpose time shared operating system, implemented on the CDC 6400. The function of the system is to represent the physical resources of the computer as a <u>universe of "objects"</u>, within which large numbers of parallel computations may occur in an orderly fashion. The notion of computation is embodied in a distinguished class of objects called "processes", which can manipulate the various objects in the universe. The definition and regulation of these manipulations is a major aspect of the operating system.

A central concept of CAL-TSS is that of "<u>layering</u>"; the user-system dichotomy has been replaced by a general scheme of flexibly graded spheres of protection or "layers." Instead of one very large and totally privileged supervisory program, CAL-TSS is implemented as a small, fast, thoroughly debugged "core" system, surrounded by several layers of successively larger, slower, and more general routines. Each layer sees all previous layers as one unified system, and in turn, presents a unified extension of that system to the subsequent layers. Failure of a layer cannot destroy the system of previous layers upon which the offending layer is running. A special construct, the "operation" is used to avoid large overhead in calls to the layered system.

The innermost layer of CAL-TSS is called the "ECS system." The universe of objects defined by the ECS system includes:

a) <u>FILES</u>: sequences of addressable words ($<2^{60}$ words long)

b) <u>PROCESSES</u>: virtual processors, each with associated address space (map) and capabilities (C-list)

c) <u>C-LISTS</u>: Lists of capabilities (access priviledges) which allow the orderly distribution of protection/ privilege among processes.

d) <u>KEYS</u> (CLASSCODES): protected words used by the system as "name-tags" in various circumstances.

e) <u>EVENT CHANNELS</u>: special queue structures used for interlocking, synchronization, and communication between processes.

f) <u>ALLOCATION BLOCKS:</u> "bank accounts" allowing the orderly allocation of and accounting for system resources (ECS space and CPU time)

g) OPERATIONS: Objects used to facilitate the transfer of control between spheres of protection/privelege in an efficient, uniform way.

The ECS system allocates (and periodically compacts) ECS and schedules processes to be swapped from ECS to CM and be run. The ECS system also performs various manipulations of primitive objects on behalf of processes, and updates allocation blocks, thus serving as an accountant for the system.

The second layer of CAL-TSS is called the "Disk system." All objects, except files, in the universe defined by the disk system are identical ~~with~~ to those defined by the ECS system. The introduction of the disk improves files in two ways:

a) The total space available for files is increased by a factor of 30

b) Files become permanent objects, surviving system restarts.

Further logical structure is also introduced into files, ~~including~~ opening and closing of files, and a procedure whereby a process may declare its "working set" of fileblocks to increase swapping efficiency.

The third layer of CAL-TSS is called the "directory system," and defines a universe including several kinds of permanent objects, which are referred to by symbolic name. New objects, "directories" ~~Directories~~ hold the symbolic names of objects , and function as permanent bank accounts for funding those objects.

The outermost layer of CAL-TSS is called the "executive." It provides a civilized interface to "user-programs" and to users at consoles. Included under "user-programs" are compilers, editors, utilities, and so on, as well as "application" programs.

[redacted]

2.0 HARDWARE CONFIGURATION AND ITS IMPLICATIONS

¶ ~~Figure 1 shows the hardware configuration.~~ All hardware is CDC standard, except the teletype multiplexor, which was built by the Computer Center. Extended Core Storage (ECS) makes the 6400 especially attractive for time sharing. ECS differs from IBM's LCS in that:

a) It is not addressable as an extension of the central memory (CM) ~~address space~~

b) It performs block transfers to CM at a [redacted] very high [redacted] rate (10^7 60 bit words/sec.)

The addressing hardware [redacted] of the 6400 is very simple. CM and ECS each have reference address (RA) and field length (FL) registers which are inaccessible to the running program. ~~THis~~ This feature allows only a single contiguous block of physical core to represent the address space, as opposed to the more sophisticated "paged" memories on some machines.

The 6400 is capable of "exchange-jumping" (saving and restoring the entire state of the CPU) in 2 microseconds. This is very important for efficient multiprogramming. ~~THe~~ The standard 6400 allows only the PPUs to initiate exchange jumps. [redacted] CAL-TSS requires a CDC standard option which allows the CPU to exchange-jump itself. This option also provides hardware "user" and "system" modes to control the specification of the new CPU state on CPU initiated exchange jumps, and synchronize CPU and PPU initiated exchange jumps. [redacted]

All I/O on the 6400 must go through the 10 Peripheral Processors. ~~TNese~~ These small (4K x 12-bit) machines are totally unprotected and uninterruptible, whereas the large (32K x 60 bit) CPU is both protected and interruptible.

The design of CAL-TSS is based on the following conclusions about the hardware:

a) Since the PPUs are slow and unprotected, no sophisticated tasks are performed by them.

A "Master PPU" maintains clocks, enforces quantum overflows, and exchange-jumps the CPU on request by other PPU's. The other PPUs function as sophisticated data channels, rather than sattelite computers.

b) Since the CPU is fast, is protected, has a high speed link to ECS, and can exchange-jump itself ("CEJ" instruction), the body of the system, including I/O control, runs on the CPU.

c) Since the simple addressing hardware complicates the storage allocation software, and since the ECS transfer rate saturates CM, so that swapping and computing cannot be overlapped, only one process is allowed in CM at a time.

d) Since ECS is randomly accessible, complex data structures and small inter-related objects can be used to advantage without incurring large overhead costs due to latency.

e) Since the memory-mapping hardware is very simple, no attempt is made to provide an extended or segmented "virtual memory". Instead, a software mapping mechanism is provided, which flexibly maps ECS into CM during swapping.

3.0 [redacted] MECHANISMS OF PROTECTION AND PRIVILEGE

3.1 ~~A.~~ Capabilities: tickets to use of objects

Within the context of a multi-user system, it is essential that access to objects which are maintained by the operating system (eg. files, processes, directories, etc.) be strictly controlled by the system. Within the ECS system, capabilities are the entities which authorize access to and manipulation of the objects existing within the time sharing system. A capability identifies the object to which it refers. It also controls the kinds of manipulations (eg., for files: read, write, destroy, etc.) which the "owner" of the capability may perform on the object. Since capabilities authorize access to objects within the system, they may never be fabricated by the user. Therefore, capabilities are gathered together in arrays of capabilities called capability lists (C-lists) where the user may refer to them but not directly modify them. C-lists are primitive objects created and maintained by the ECS system. Moving capabilities from one C-list to another, while (possibly) downgrading the set of manipulations allowed on the object, facilitates the sharing of objects and provides precise and flexible control over the ~~exact~~ limits of access to shared objects.

The ECS system maintains a Master Object Table (MOT). It contains the unique name and ECS address of each primitive object in ECS. It is the only critical table in the ECS system. All capabilities point to objects ~~thru~~ through the MOT. This design facilitates ~~garbage collection~~ the process of compacting ~~of~~ ECS, and the unique name in the MOT provides a necessary ~~level of redundancy to~~ check on the validity ~~or~~ of capabilities.

Thus, a capability for an object consists of 1) unique identification, 2) an MOT index, 3) a type, and 4) a set of allowed manipulations. Whenever a new object is created for a process, the ECS system returns to the process C-list a capability for the new object, which authorizes all possible manipulations of the object.

~~Thus, we have seen that~~ To summarize, control of access to objects within the system is enforced by the mechanism of capabilities. A program running on [redacted] CAL-TSS can manipulate an object maintained by the ~~basic~~ ECS system only by presenting a capability authorizing access to the object and permitting the ~~contemplated~~ requested manipulation.

3.2 ~~B.~~ Processing environment or Sphere of protection/privilege

The processing environment is the context in which a program executes instructions within CAL-TSS ~~the time sharing system~~. It consists of 1) a set of capabilities, 2) the size of the address space, and 3) the contents of the address space.

The set of capabilities which may be directly invoked by a program defines the access privileges of the program. This set of capabilities (called the full C-list) is the logical concatenation of one or more C-lists. Capabilities may be referred to by their index in the array of capabilities which makes up the full C-list. The ~~This~~ set of objects and allowed manipulations given in the full C-list defines the privileges enjoyed by the associated program.

The CAL-TSS [redacted] address space is a one level vector of words. The size of the address space and its contents (including the program) are [redacted] clearly part [redacted] of the processing environment. The content[redacted] of the address space is defined by a map which associates areas ~~intervals~~ of the address space with areas ~~intervals~~ of files. The map is interpreted as a swapping directive when the process is activated. The address space is a sequence of addressable words in which the addresses can be interpreted by the hardware. A file is a sequence of addressable words whose addresses must be interpreted by the ECS system.

Whenever a program is to run on the CPU, its address space must be constructed from files which are residing in ECS. Only the required portions of the files need be resident in ECS. To reduce the overhead involved in constructing the address space (swapping) the logical map entries (file, file address, address space address, and word count) are "compiled" to the absolute ECS addresses of the file data. Portions ~~Intervals~~ of the address space which are either "pure" procedure or a constant data base, are [redacted] not [redacted] copied back to their respective files when the address space is being swapped out of central memory. ~~Thus, "read only" map entries are provided to gain the associated efficiences. In addition, shared "pure" procedures are protected by this feature against inadvertant modification.~~

The configuration the processing environment clearly limits the privileges of a program. [redacted] Programs [illegible] [redacted] protected [redacted] from one another by associating with each [redacted] its own processing environment.

3.3 Process and Subprocess or Combining program environments

As a logical follow-up to the discussion of ~~Having seen~~ the mechanisms used by the innermost layer [redacted] of ~~the~~ CAL-TSS [redacted] to define a processing environment, it will now be ~~we shall~~ shown ~~discuss~~ how a number of related programs with different processing environments are combined into one process. A process is 1) a CPU state (registers, etc), 2) a set of state flags [redacted] 3) a set of subprocesses, and [redacted] 4) a call stack. Associated with each Subprocess is a "local" processing environment which consists of a C-list, the size of the address space, and a map. So one makes each program a subprocess in order to build protection walls between them.

The subprocesses within [redacted] a process are organized in a rooted tree structure. The unique path through the subprocess tree from any subprocess to the root of the tree defines a set of subprocess called the "ancestors" of the subprocess. In addition, a subprocess is defined to be its own "ancestor". In general, [redacted] subprocesses closer to the root of the subprocess tree are thought of as being more "powerful" than subprocesses near the leaves of the tree.

At any given time, there are two distinguished (not necessarily distinct) subprocesses within a process. These are called the 1)"current" subprocess and the 2) "end-of-path" subprocess. The "current" subprocess is always an "ancestor" of the "end-of-path." It is the subprocess currently in control (ie. the subprocess) whose program is running). The subprocesses between the "current" and "end-of-path" subprocesses (inclusive) are called the full path. The processing environment of the"current" subprocess is the concatenation of the "local" processing environments of all the subprocesses in the full path. Thus, the full C-list is the concatenation of all the C-lists of all the subprocesses in the full path. The size of the full address space is the sum of the sizes of all the "local" address spaces in the full path and the contents of each "local" address space is defined by the map of the corresponding subprocess.

The concept of the full path implies that less "powerful" subprocesses (i.e. near the leaves of the subprocess tree) may, under the proper full path conditions, have their "local" processing environment annexed to the "local" environments of other, more "powerful", subprocesses. The most obvious application of this concept is one in which a "debugging" subprocess annexes the processing environment of the "debuggee". Another application allows two subprocesses to be "protected" from each other if they are on different branches of the subprocess tree.

Control may pass from one subprocess to another by mechanisms discussed in section 3.4. As control passes between subprocesses, the full path is defined dynamically by the relationship between the subprocess receiving control and the "end-of-path". The subprocess receiving control becomes the "current" subprocess. If the new "current" subprocess is an "ancestor" of the present "end-of-path", then the "end-of-path" remains unchanged. Otherwise, the "end-of-path" is set equal to the new "current" subprocess.

To keep track of the flow of control each process maintains a call stack. The call stack records the "current" and "end-of-path" subprocess and the P-counter of the "current" subprocess. This information is sufficient to reconstruct the processing environment and to restart a program which has been interrupted by calling another program.

In the example of the debugging subprocess, we can assume that the debugger is a proper "ancestor" of the "debugee". We see that, if a breakpoint has been inserted in the "debuggee" which causes control to be transferred to the "debugger", the full path includes both the "debugger" and the "debugee" (with the debugger being in control). Considering subprocesses in different subtrees, one subprocess may never annex the environment of the other since the two subprocesses do not have any descendants in common so each is protected from the other. Therefore, the transfer of control from one subprocess to the other will always result in the "end-of-path" being reset to be the same as the subprocess receiving control.

Transfer of Control or Protected Calls

Often, one program may wish to initiate execution of another program which requires a different processing environment. This occurs not only in the case of transferring control between subprocesses, but also, when a program calls upon the system to perform some manipulation on an object maintained by the system. To accomplish these transfers of control, we wish to provide a clean interface between programs running in different processing environments which, not only facilitates the call of one program by another, but also obscures the distinction between the calls to the basic system and calls which activate subprocesses. If we can accomplish this, calls on the basic system may have the same format as calls upon subprocesses which may perform "system like" actions.

When transferring control from one program to another, the parameters of the call must also be transferred to the environment of the program being called. Parameters come in two varieties: datum parameters representing numerical values or pointers; and capability parameters which refer to objects within the system. At the calling interface it is desirable to do checking on the capability parameters. A program expecting a capability to write on a file would surely be in trouble if it received, instead, the capability for a C-list or a file capability without write access. Thus, we wish to check capability parameters to insure that they are of the correct type and that they permit the required manipulations of the object.

Next, the calling interface must control the number of parameters transferred to the environment of the program being called. Since the parameters must occupy some space in the new environment, the called program must allocate this space. Should an arbitrary number of parameters arrive, other information within the new environment would be destroyed.

Finally, the entry point (the address at which execution is initiated) of the called program must be implicit in the specification of the program to be called. This consideration is necessary to protect the called program from being initiated at other than its expected starting point.

The mechanisms to manage the transfer of control between sphere's of protection are incorporated in the primitive objects called operations. Operations specify the program which is to be entered and provide parameter checking information. An operation is invoked by calling the ECS system (executing an exchange jump-CEJ) and passing a pointer to a parameter vector. The zeroth element of the vector is the capability-list index of a capability for the desired operation.

Parameter checking is controlled by a set of parameter specifications contained in the operation. The parameter specifications direct the processing of the parameter vector. Datum parameters are simply copied from the parameter vector. Capability parameters are denoted in the parameter vector by their index in the user's full C-list. The capability is checked, using the parameter specification, for the correct type and required set of permissible manipulations.

Operations must also specify the program which is to be called. For ECS system programs it is sufficient to provide an integer to identify which ECS system program should be called. The specification of a subprocess to be called involves more subtle considerations. [redacted] All layers of the system outside the ECS sytem are implemented as [redacted] subprocesses which are [redacted] part of every ordinary [redacted] process. In order to ~~We should like~~ to avoid creating separate operations for each [redacted] process, ~~Thus, we need~~ a "naming" facility is needed ~~by which we can~~ to identify subprocesses. With such a facility, the operation may carry the subprocess "name". Operations may be shared by all [redacted] processes which are equipped with the "named" subprocess.

~~Our~~ The "naming" facility must 1) produce unique "names" [redacted] (lest the wrong subprocess get called); and 2) provide for mechanisms of protection and restricted access to the "names" (so that the careless user cannot use "names" already assigned to other subprocesses). By making these "names" primitive objects within the basic system, ~~we achieve~~ the protection and access control of the capability mechanism is achieved. ~~In short,~~ The basic system provides objects (called class codes [redacted]) which ~~amount to~~ are protected 60 bit data items. The content of a class code [redacted] (i.e. the 60 bit datum) is [redacted] used by operations to identify the subprocess to be called; while the class code [redacted] (i.e. the object) is used to construct operations or to "name" subprocesses when they are created. ~~We shall see in section~~ An discussion of how class codes are also used to identify users and authorize access to file directories is given below (See Section 4.0)

~~Having discussed~~ The above discussion of the mechanisms involved when one program calls another ~~, we shall proceed~~ leads to the question of how control is returned when a program has completed its function.

A ~~program~~ sub-process may complete either by performing its computation or manipulation to completion or by discovering it cannot complete the desired computation or function. (This distinction is analogous to the success and failure transfers in SNOBOL when trying to match a pattern.) For example, a file read by the ECS system will fail if ~~some~~ the portion of the file referenced by the read is not currently in ECS.

When a program completes successfully, it should initiate a normal return (by calling the ECS system with an operation for return). A return causes the call stack (Section 3.3) to be popped.

The environment specified by the new top of the call stack is established. Execution is resumed at the location obtained by adding the P-counter saved in the call stack to the low order 18 bits of the CEJ instruction word originally used to initiate the transfer of control.

If a program fails, ~~we may wish~~ it may be desirable to provide for some other program to attempt to complete the function of the first program. Within the basic system, the mechanism of "F-returns" provides this feature. To achieve this result, it is necessary to extend the notions embodied in the operation mechanism. Operations ~~actually~~ may specify a sequence of programs to be called in case of F-returns. When a program initates an F-return, the next program in the sequence specified by the original operation is called. The program specifications for alternative actions must be restricted to subprocess calls to protect the integrity of the ECS system. Another feature of the F-return mechanism is to provide for additional parameter specifications with each program specification. This allows additional parameters to be passed to the subsequent programs. If the sequence of alternative subprocess specifications has been exhausted by one or more (possibly) repeated F-returns, a [redacted] return must be made to the originating program. However, the P-counter is not offset in this case as in the normal case of the regular return, and serves to ~~This~~ notify the originator of the call (s) that the requested function was not performed.

The F-return mechanism is useful ~~in that~~ since system action requests are first attempted at the lowest (most efficient) levels of the system. Unusual conditions are automatically reflected to higher layers of [redacted] the system. [redacted]

Hierarchies of processing and data structure manipulation can be embedded in the return mechanisms while appearing to be single operations from the point of view of the calling program.

4.0 Process communication and synchronization

4.1 Event Channel: co-ordinated processing

A program within a process must have available to it a mechanism by which it can wait for some external event to occur. For instance, a tape-reading program may wish to discontinue processing until some buffer is full, or a program which is servicing several other programs may wish to wait for more requests from its customers when it runs out of things to do. In case the desired event has already occurred, the mechanism must also provide for queuing of the events until they are requested by processes. [illegible]

Thus, to synchronize running processes, the ECS system creates and maintains "event channels". An event channel consists of two queues: a queue of "events" and a queue of waiting processes. Only one of these queues may be non-empty at any given time. The occurrence of an "event" is marked by the sending of an "event" to an event channel.

An "event" is simply a 60-bit datum ~~(plus the name of the sending process and the name of the [illegible])~~ which is to be passed to a requesting process.

When an "event" is sent to an event channel, either it is passed to the first process on the waiting-process queue or, if there are no waiting processes, it is appended to the queue of events. Similarly, when a process requests an event from an event channel, either it is passed the first event on the event queue or, if there are no queued events, it is appended to the waiting-process queue. When a process is added to or removed from a waiting-process queue, additional actions are taken to schedule or de-schedule the process. ~~[illegible]~~

~~can interface to the process ... to activate and ... activate processes when they enter and wait ... the queue of waiting processes on an event channel.~~

Clearly, event channels can be used to synchronize several otherwise asynchronous processes. The 60-bit datum "event" provides additional information for the reciever of the event. Another use for event channels is to enable allocation and locking. For instance, given an event channel with 5 events in the event queue, corresponding to the 5 tape drives, a process could reserve a tape drive simply by getting an event from this event channel. If no drives were available, the process would wait until some other process returned an event to the event channel (released the tape drive)

~~[illegible]~~.

¶ An event channel with a single event in the event queue can act as a lock. To lock a data base or other shared object, a process simply removes the event from the event queue. To unlock, the event is returned to the event channel. If the lock is already "set", any process attempting to get the event will have to wait until it is returned to the event queue.

It is important to note that use of the event channel mechanisms requires the co-operation of the processes using the facility. Processes must wait on event channels and send events to event channels in an orderly manner if the facility is to be useful.

4.2 Process interrupt: asynchronous processing

Under certain conditions, it is necessary for one process to "get the attention" of another process. The event channel mechanisms are not sufficient for this since they depend on voluntary co-operation between the processes involved. Therefore, ~~[illegible]~~ "process interrupts" are provided. Using this feature, one process may force another process to transfer control to a specified subprocess. For example, the system operator can force each process to type out a message on its console (e.g. "system going down in 10 minutes.")

A process interrupt is initiated by one process (the "interrupting process") and directs another process (the "interrupted process") to activate a specific subprocess (the "interrupt subprocess"). However, the interrupt subprocess will not be activated until the normal flow of control passes to one of its "descendents", since it would be improper for an interrupt subprocess to pre-empt one of its more powerful "ancestors". This "interrupted subprocess" is then effectively forced to call the interrupt subprocess, which is always one of its "ancestors."

<insert F>

¶ The transfer of control to the interrupt subprocess must obviously make provision for the eventual return of control to the interrupted subprocess. When the interrupt subprocess returns, the interrupted subprocess resumes exactly as if it had never been interrupted

If the interrupted process is waiting on an event channel at the time of the interrupt and the interrupt subprocess is an "ancestor" of the "current" subprocess in the interrupted process, the process must be removed from the waiting-process

Insert F

Since the interrupt subprocess may not be called immediately, the interrupt must be recorded in the process. If there are pending interrupts, the "ancestors" of the new "current" subprocess must be checked for interrupts at every subprocess transfer. From the time the interrupt is first sent until the interrupt subprocess is called, interrupts to the same subprocess are disabled (have no effect).

~~Futhermore,~~ Since a subprocess may interrupt itself (it is its own "ancestor" by definition) there must be a facility to inhibit interrupts in which the "current" subprocess interrupts itself. This facility is called ~~a~~ "interrupt inhibit" ~~and only prevents the an interrupt subprocess~~ ~~which is the same subprocess~~ ~~the current~~ ~~from being called as long as it is in effect~~.

The interrupt inhibit is set automatically whenever ~~th~~ an interrupt subprocess is called, and may be cleared and set by the program.

queue of the event channel and scheduled to run. The interrupt subprocess will be called immediately. Whenever a process enters the waiting-process queue of an event channel, the P-counter of the "current" subprocess is "backed-up" one word. Thus ~~we see that~~, if there is an ~~in the case of an~~ interrupt to a process which is waiting on an event channel, when the interrupt subprocess returns, the process will re-executed its call to get an event from the event channel. ~~[illegible]~~

In summary, inter-process synchronization and communication can procede in two modes. The pre-emptive mode of the process interrupt, and the co-operative mode of <u>event channels</u> provide flexible facilities to define the relationships between processes.

(4.0 continued)

4.3 Illustrative examples
Communication between processes.

When a user pushes the panic button on his teletype (currently the ~~control-shift-p~~ (CTRL) (SHIFT) (P) button) an interrupt is sent to the executive (root of subprocess tree) of the process owning that teletype. This causes control to pass to the root. ~~This is vital in getting programs out of loops.~~ ~~This is a asyncronous form~~ This is an asyncronous form of interprocess communication and is vital for getting programs out of loops. It can come at any time and in general will come at the wrong time (e.g., when the teletype buffer is locked because the pointers are being manipulated). handling interrupts correctly can be very subtle. (For this reason). They should be reserved only for ~~very~~ drastic situations.

Event channels provide a more graceful mechanism for communicating between processes. Their simplest use is as a lock. Currently there is one printer, driven by a PPU ~~#~~ ~~we have one printer. There is a PPU~~ which accepts commands from the CPU. ~~and drives this printer.~~ In order to prevent two processes from driving the printer simultaneously, an event channel with one event was created. A process wishing to drive the printer ~~The process which is driving~~ reserves it ~~the printer~~ by removing the single event from the event channel. ~~[illegible] this event~~. No other process should attempt to drive ~~touches~~ the printer until ~~they~~ it has been able to get the "lock-out" ~~have this~~ event. Indeed, if the printer is already active, processes ~~they~~ will hang on the event channel waiting for the event ~~it~~. When a process finishes with the printer it returns (sends) the event to the channel. This will wake up the next process waiting in the queue, if there is one. If there are no waiting processes, the printer is free. This mechanism results in a FIFO scheduler. It generalizes to n printers by providing n distinct events. More elaborate schedulers require the addition of a scheduling process. The salient feature here is that the lock is voluntary. A malicious process can ignore it if it likes. Event channels are designed for cooperating processes.

A second example of the use of event channels is as an interprocess communication. For example, the PPU-CM interface for the teletype has two event channels: one for PPU to CM messages, and

Processes commonly hang waiting for "more input" or "more room in output buffer" events

The other for CM to PPU messages, such as "more output", "more room in input buffer", "resume echo", "change echo table", "purge", etc.... The actual lines are passed in buffers since the volume of information ~~did~~ does not make the overhead of sending each 60 bit word as an event economically ~~fiesible~~ feasible (~250 μsec). In the future this decision ~~will~~ may be ~~reversed~~ reconsidered.

4.4 ~~11.3.2~~ Capabilities and Keys

?. Classcodes are objects which identify ~~some~~ classes of users. Their primary purpose is to obtain capabilities for objects. ~~However~~ Also within a process, they function in lieu of capabilities for subprocesses since subprocess are not objects and thus cannot be pointed to by capabilities.

An essential reason for keys is the ~~feeling~~ concept that having the capability for a directory (~~a file~~) does not give one ~~the~~ capabilities for the entries in that directory. Certainly it does not give the same capabilities for all the objects in the directory. So the directory structure is built with a list of two word entries associated with each directory (called the access list for that entry) entry. ~~When the file system is asked~~ Associated with a request on the directory system to deliver a capability for ~~a certain~~ some object in the directory, ~~a key~~ must be ~~presented~~ a key. The key (60 bit datum) is matched against

The elements of the access list for equality. If a match is found the second of the two words specifies the set of allowed ~~option~~ manipulations ~~bits~~ ~~for~~ provided by the capability ~~will provide all~~ ~~to be delivered~~. For example the source for the EDITOR might have two keys. One a read only key which is public and another a read-write key which is owned by the systems programmer who ~~created~~ maintains the editor ~~it~~. ~~This points out an application of Keys~~. All the public systems programs will reside in a central directory. Not all systems programmers should be able to ~~mess with~~ alter ~~my~~ the editor, nor should the author of the editor ~~and I should not~~ be able to ~~mess with theirs~~ meddle in their domains. Keys provide ~~this~~ a facility for preventing such occurrences

A second important ~~reason~~ use for keys is that rather than ~~hand~~ a subprocess receiving all the capabilities to which it is entitled, it can be given just one object which it can use to obtain exactly those capabilities it needs. This facility considerably shortens capability lists.

(A)

5 ~~4.3.3~~ Subprocess structure

The subprocess structure of CAL is its most radical feature. It is in some respects self explainatory. ?? For example the concept of "full path" is needed to allow more powerful subprocesses (such as debuggers and student grading programs) to look

(A)

Sharing of objects is done via capabilities. Seveal people can share one ~~file~~ object by giving each person a capability for the file. Further, not all people need have the same capability for the object. For example ~~two people can have a cap own a file one~~ suppose there is a file of systems messages. The system will have a write capability for this file ~~and~~ but all ~~users~~ other processes will have a read capability for ~~this file~~ it.

at the address space of less powerful subprocesses. This ~~is not~~ scheme does not ~~radically different~~ differ greatly from the multics ring structured protection. In fact if the subprocess structure were a list structure rather than a tree structure ~~we would have~~ it would constitute a ~~[illegible]~~ restricted ring structure protection mechanism. However, The power comes ~~in~~ from having a tree structure. In a ~~ring~~ ring structure ~~one~~ each subprocess must be more or less powerful than ~~another~~ all others. Utilities must ~~run~~ run in low numbered rings (3–10) and users in high numbered rings 50–64. Now suppose two competitors have different programs and I deal with both of them. Then what ring should they be put in. Clearly they fall in the same ring. But in that case each can inspect the other and has the power of the other. This is solved in CAL by putting them as separate subtrees under the executive. In this configuration neither one can see the other because neither one is given the class code of the other. (For a deeper discussion of this see the paper by Prof. Butler Lampson in FJCC 1969.)

4.6 Operations + the flow of control

One view of the ECS system is that it schedules processes to run, compacts ECS, does accounting and, using the operation mechanism, moderates

The flow of control and the protection walls within processes. ~~by the operation mechanism~~. ~~Everything else~~ All other activities (eg.: send/get events, read/write files, etc) ~~produce capability~~ are ~~just~~ merely actions which may be thought of as canned subprocesses. In fact, ~~this is the way~~ it is with this approach that the ECS system is implemented.

Great ~~c~~ care has been taken that there be as few mechanisms in the system as possible. ~~The operations mechanism~~ ~~are one of these few mechanisms~~ ~~is no exception~~. Operations provide the uniform interface between user/user and user/system. It is impossible for the user to distinguish the system actions and calls from calls on systems running below him (eg. the executive), an arrangement which ~~This~~ lends itself to a very clean interface structure. Operations are directives which check ~~and~~ parameters, invoke actions, and pass the buck to ~~the next~~ an alternative action if an action fails. For example suppose one asks for a file to be read in. First the ECS file read action is initiated. If it ~~fails~~ fails, the file may be on the disk so a disk-read action is initiated; if the disk read action fails ~~[illegible]~~ an error has probably occurred so an F-~~RETURN~~ return is passed back from the operation. If not the user is never aware that he had to go to disk to get the ~~block~~ file. In actual practice this has proven to be an extreemly convient way of handeling control since only those actions which are actually required are invoked.

~~Section V~~

5.0 external I-O

For the ECS system, the external world consists of all the devices ~~equipment~~ which can be connected to the computer through data channels. These include disks, printers, cardreaders, tape drives, the consol display and Teletypes. **ECS is not an external device.** ~~to implement the base ECS system~~.

The user~~'s~~ process is provided with no special operations to access ~~these external objects,~~ these devices ~~this external equipment~~ ~~Instead we make certain files and~~ Instead **"pseudo-processes" are created to interface to each device**, Since communication with processes is done via files and event channels, communication with ~~external equipment is via files and event channels.~~ these pseudo-processes (i.e. external devices ~~equipment~~) is via files and event channels.

Since the only direct connection between external devices ~~equipment~~ and the computer is through the PPU's, However, special system code had to be written to interface between the ppu's and the files and event channels for ~~that~~ each device ~~equipment~~. ~~The external interface consists of all that code residing in ppus or in the system whose function is to simulate the special processes.~~ The external interface consists of this special system code plus ~~the~~ code residing in the PPUs ~~ppus~~ for controlling particular devices ~~pieces of equipment~~.

The basic strategy of the external interface is as follows. Each PPU will contain a resident program controlling one or more ~~pieces of equipment~~ devices. This program will have some local buffering in the PPU itself, plus some more ~~buffering~~ in special fixed buffers in central memory (CM). When the CM buffers become full, empty, or at completion of requested actions, the PPU program will request the master PPU ([illegible]) via a data channel, to force the CPU to switch to a special system ~~code~~ routine for ~~this equipment~~ the device. The PPU program then waits for a request for more action, which is posted in a particular CM communication word.

Each special system routine ~~will be~~ is able to transfer data between ECS files and the ~~special~~ fixed CM buffers. They can also make requests on the PPU programs through the CM communication words and can cause events to be sent on ECS event channels.

~~Finally, special ECS objects called pseudo processes have been provided.~~ The special system routines use ~~these~~ pseudo-processes to get events from event channels. If no event is present, the pseudo-process is placed in the process queue. When an event is sent to an event channel on which a pseudo-process is waiting, the system ~~unchains~~ de-queues the pseudo-process, places the event data in the pseudo-process and, via a special CM communication word, requests the master PPU to cause an exchange jump to the appropriate special routine.

The ~~ECS~~ special routines and PPU programs use various methods of intercommunication. There is ~~no~~ little overall design. The user interface with any particular ~~equipment~~ devices usually consists of an event channel ~~to send~~ for sending requests to the ~~equipment~~ device, an event channel ~~to get~~ for getting responses from the ~~equipment~~ device and a file in which to buffer the data. Beyond that there is no general design.

In order to permit ~~handing~~ passing control of a particular piece of external equipment to different processes at different times, it is intended that the unique names of the files and event channels ~~would~~ be changed in the Master Object Table (MOT) ~~[illegible]~~ when control of the piece of equipment is to be removed from a process. Hence any capabilities for the files and event channels available to that process would no longer be usable.

October 11, 1969

6.0 Progress Report on 6400 CAL Time-Sharing System (10/11/69)

Jim Gray

In order to understand the status of TSS, it is necessary to understand its architecture. TSS is built up in three layers called the ECS layer, the disk layer and the executive layer. The ECS layer is the core of the system. Its design has strong implications for all higher layers. Its function is to create, manage and destroy objects in ECS and to provide protection walls and and communication paths between processes and other TSS objects. It also includes the process scheduler and the ECS-CM swapper. The disk layer reflects ECS files up into the disk store. It provides facilities for creating, managing and destroying disk files as well as opening and closing them. The executive consists of a command processor, log in-log out procedures, accounting routines and a directory system. Its duties are comparable to those at SCOPE except that the objects that it manipulates are the disk/ECS objects created by the low-level systems. Compilers, interpreters, editors and user-constructed subsystems run "on top of" the exec just as the exec runs "on top of" the disk system.

Currently the ECS system is operative. About four months of work and an equal amount of documentation remain to be done on it. There is a provisional executive program running on top of the ECS system allowing TSS to be written on itself (see Figure 1). Currently TSS has enough CPU to support 60 systems programmers (or about 150 ordinary users). However, there is only enough ECS for about 10 active processes. There are 6 teletypes connected to TSS. We are confident that TSS will gracefully support 1000 student users when it is complete.

The design of the disk system is almost complete. Implementation has begun recently and should be complete by Feb. 1. This project is in series with a disk driver program which will be available in mid-December. With the advent of the disk system, a new porivisional executive will be written. At that point TSS will be able to support many (~ 60) users. We plan to offer TSS to persons who can provide their own teletypes and who are developing subsystems for TSS (e.g., Basic, CAL, APL, FORTRAN, ...). A manual on the system is being prepared for this eventuality.

The executive is in the preliminary design stages. A reasonable guess of its delivery date is mid-summer 1970.

A background batch system is in development. It will run simple SCOPE jobs (no tapes) and will be SCOPE-compatible. It requires routines to drive card readers and printers, a display driver and a dayfile generator. Almost all other work to interface SCOPE with TSS is done in the SCOPE simulator now running.

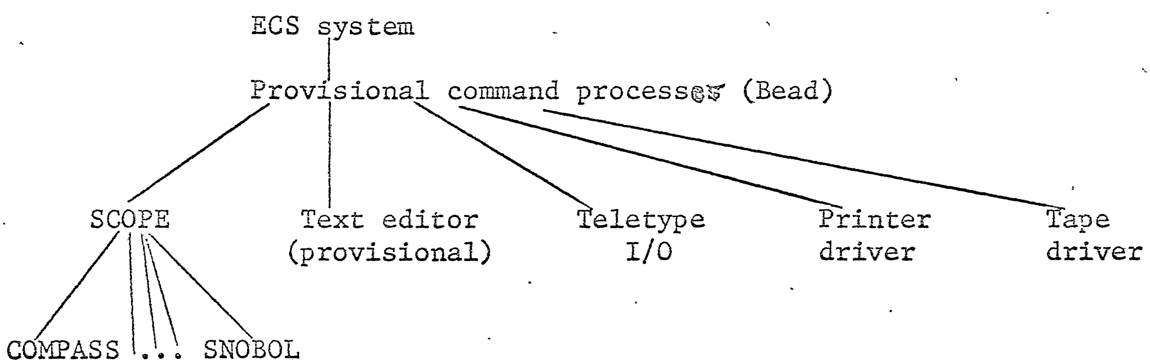
To facilitate systems programming one software subsystem (not part of TSS) is being implemented. It is an assembler/debugger called Cool Aid. The assembler has an Algol syntax and an elegant macro-facility. It is designed to be very fast (~ 10 times faster than Compass) and compact, and is re-entrant. It will feed a loader which is SCOPE-compatible. There will be a run time interactive debugger which will allow the teletype to examine and modify (symbolically) a running program without complete reassembly.

Also in development is a sophisticated editor. Members of the CS and EECS departments are supervising the development of a BASIC and an APL.

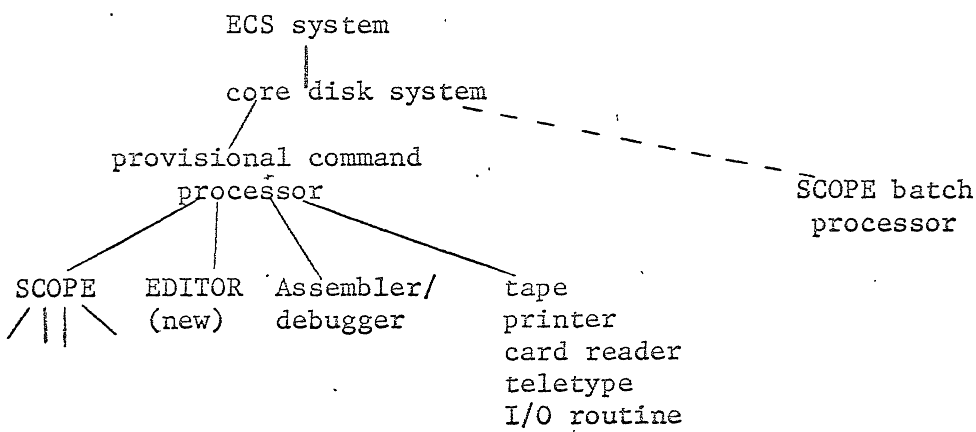
I plan to implement a JOSS-like language next spring (with the help of CS undergraduates) and to supervise FORTRAN and ALGOL syntax checkers at that time.

The developers of Cool Aid have expressed an interest in producing an interactive SNOBOL 4.

Current Status (October 10, 1969)



February 28, 1970



The current personnel allocation is

Malbrain McJones	Debugger/assembler (Feb. 28)
Redell Bentley Vaughan	Complete and document ECS system (Jan. 1)
Lampson Lindsey Morris Redell Sturgis	Design executive system (mid-summer)
Lindsey Redell	Implement core disk (Feb. 1)
Sturgis	Implement disk driver (Jan. 1)
Gray	Design and Implement editor (Jan. 1)
Standiford	Implement batch system (April 1)