Table of Contents

## 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 universe of "objects", 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A central concept of CAL-TSS is that of "layering"; the user-system dichotomy has been replaced by a general scheme of flexibly graded spheres of protection or "layers".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. FILES: sequences of addressable words ($< 2^{60}$ words long)

b. PROCESSES: virtual processors, each with associated address space (map) and capabilities (C-list)

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

d. CLASS CODES: protected words used by the system as "nametags" in various circumstances

e. EVENT CHANNELS: special queue structures used for interlocking, synchronization, and communication between processes

f. ALLOCATION BLOCKS: "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/privilege 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 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 becomes 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", 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.

## 2.0 <u>HARDWARE CONFIGURATION AND ITS IMPLICATIONS</u>

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);

b. it performs block transfers to CM at a very high rate ($10^7$ 60 bit words/sec.).

The addressing hardware 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 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 standard 6400 allows only the PPUs to initiate exchange jumps. 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 in CPU initiated exchange jumps, and to synchronize CPU and PPU initiated exchange jumps.

All I/O on the 6400 must go through the 10 Peripheral Processors. These small (4K 12-bit) machines are totally unprotected and uninterruptible, whereas the large (32K 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 sophiticated tasks are performed by them. A "Master PPU" maintains clocks, enforces quantum overflows, and exchange jumps the CPU on request by other PPUs. The other PPUs function as sophisticated data channels, rather than sattelite computers. —sp —sp

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.

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 flexible maps ECS into CM during swapping.

## 3.0 MECHANISMS OF PROTECTION AND PRIVILEGE

### 3.1 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 (e.g., 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 (e.g., 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 facricated 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 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 through the MOT. This design facilitates the process of compacting ECS, and the unique name in MOT provides a necessary check on the validity 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.

To summarize, control of access to objects within the system is enforced by the mechanism of capabilities. A program running on CAL-TSS can manipulate an object maintained by the ECS system only by presenting a capability authorizing access to the object and permitting the requested manipulation.

## 3.2 Processing environment or sphere of protection/privilege

The processing enviroment is the context in which a program executes instructions within CAL-TSS. 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 set of objects and allowed manipulations given in the full C-list defines the privileges enjoyed by the associated program.

The CAL-TSS address space is a one level vector of words. The size of the address space and its contents (including the program) are clearly part of the processing enviroment. The content of the address space is defined by a map which associates areas of the address space with areas of files. The map in interpreted as a swapping director 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 of the address space which are either "pure" procedure or a constant data base are not copied back to their respective files when the address space is being swapped out of central memory.

The configuration of the processing environment clearly limits the privileges of a program. Programs can be protected from one another by associated with each its own processing environment.

## 3.3 Process and Subprocess or Combining program environments

As a logical follow-up to the discussion of the mechanisms used by the innermost layer of CAL-TSS to define a processing environment, it will now be shown 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, 3) a set of subprocesses, and 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 a process are organized in a rooted tree structure. The unique path through the subprocess tress from any subprocess to the sp root of the tree defines a set of subprocesses called the "ancestors" of the subprocess. In addition, a subprocess is defined to be its own "ancestor". In general, subprocesses closer to the root of the subprocess tree are through 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 (i.e., 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 applications 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 enviroment and to restart a program which has been interrupted by calling another program.

In the example of the dubugging 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 "debuggeee" which causes control to be transferred to the "debugger", the full path includes both the "debugger" and the "debuggee" (with the debugger in control). Considering subprocesses om 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.

## 3.4 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 there transfers of control, it is desirable to provide a clean interface between programs running in different processing enviroments in order not only to facilitate the calling of one program by another, but also to obscure the distinction between calls to the basic system and calls which activate *subprocesses*. Given this clean interface, callson the basic system may have the same format as calls upon subprocesses which may perform "system like" actions.

When control is transeerred from one program to another, the parameters of sp 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. In 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, capability parameters must be checked 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 environemnt, the called program must allocate this space. Should an aribtrary 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 spheres 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 zero-th 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 con-