Classification: 000/0
Date: 5/1/78
Doct=1 Vers=2

Lecture 1

The first section covers pieces and the relationships between system components, the basic interface, and some common tables. This section will not cover the internal details of any one interface or any one piece; they will be covered later. Thus, for example, some of the SVCs in the task-to-supervisor interface will be mentioned, but not necessarily all of them.

Looking at the system as a whole, the first question is "who's in control?" Usually the answer is, "The supervisor." But the supervisor, doesn't seem to have any entry point--it never begins and ends. "How does the supervisor get entered?" An I/O interrupt or a program interrupt occurs. Who causes the interrupts? A task that's running. Who starts a task? It's the supervisor. It's the chicken-egg problem. Essentially, the whole system is interrupt-driven. Someone initiates things, usually the operator pressing the request key on the console typewriter.

This section is an overview of what could be called the Steady State System. In other words, it is assumed that the system is there, loaded and running. How the system is built, how it is loaded from disk once it has been built, and things like that will be covered later.

Figure 1 is the picture around which discussion in this section will revolve. Most of the items here are covered in detail in later sections. For now they will be treated as black boxes and only the connections discussed. The connecting lines are the interfaces, and the numbers on each line are solely to identify the interface for discussion.

At the bottom of this picture is the hardware machine. Above this is a box called supervisor. Note that this picture is carefully stratified in a number of manners. At this point, note the boundary indicated at the left between supervisor state at the bottom and, above it, problem state. Problem state and supervisor state refer to the hardware definition of the supervisor state and problem state for the 360 and 370.

This means that the supervisor, or anything else below that line, runs in supervisor state and nobody else does. That, essentially, is the definition of the supervisor, although one usually considers the supervisor to be a particular assembly listing. There are, however, other things below the line. There's a series of what will be called supervisor subroutines which the supervisor causes a task to call. For example, there's one called JBRP, which stands for Job Request Processor, called in the process of task initiation. The interface (2) between the supervisor and the supervisor subroutines is that the supervisor

causes the task to start there before going about its business by virtue of setting the task's PSW to the entry point of the subroutine before dispatching the task. There's another entity which lives below the line which is hard to classify exactly where it belongs in the system. This is the machine check recovery code which as its name suggests, gets control when a machine check occurs.

The interface labelled "1" between the supervisor and hardware is well defined by the Principles of Operations manual and it means that the supervisor owns the PSWs for the old and new various interrupt states. It gets entered whenever an I/O interrupt or external interrupt occurs, etc.

Above the supervisor in this diagram are the tasks in the system. In order to describe the interface and say a few things about the historical wording or terminology that occurs, a simple task will be discussed first.

The task used will be the REWIND task, and the interface between it and the supervisor (labelled "3") is the one being discussed. [At this point a slight digression is necessary. Back at the time this all started (1966), we obtained from Lincoln Labs a small supervisor called LLMPS which managed jobs of this variety. The terminology they picked used "job program" to represent the code in the machine, and "job" to represent an activation of that, and there may be several activations of that if it's a re-entrant job program. Since then, computer terminology has evolved so that normally "task" is used for "job". But for the purposes of this manual, jobs and tasks are used interchangeably, for terminology.]

On this interface (3) there are three areas to cover. One is getting started, in other words, how does it start a task? Another is how it obtains services while it's running, and third is how it terminates.

This discussion is applicable to all tasks in the system, although the example is relatively simple. MTS has a job program. It's started many times and it has the same interface, although the job program is larger than most. The main interrupt that starts everything is the request button on the operator's console typewriter, pressed because the operator wants to start something. Every line entered by the operator is a request to start a task. There is no command language at the supervisor level--it only starts jobs.[1] The first thing that the operator types in is the name of the job he's starting. As a slight digression, one might ask how do you stop a task, once it's started? There's a job called STOP. If the operator wants to stop a job he presses the REQUEST button and enters "stop" and a

---

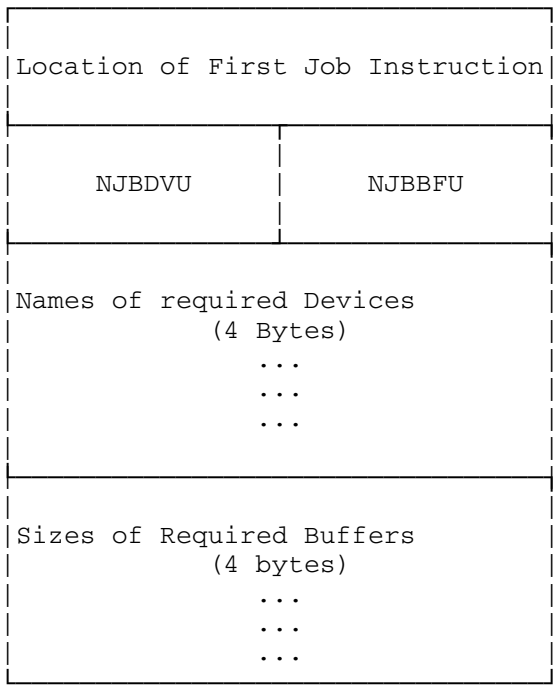
[1] That's actually not quite the truth. Lines beginning with $ are passed to HASP as commands, and lines beginning DIS,MOD,SE, or TRCTP are swallowed by the supervisor as actual supervisor commands. But everything else starts a job.

parameter designating what is to be stopped. This starts the STOP job whose purpose is to stop another job. Hopefully, it gracefully stops by itself to eliminate cascading problem?

To get back to starting a job, the first thing the supervisor does is allocate a job table entry for this invocation. This is a fixed-length area where all variable information pertaining to a job (or else a pointer to same) is kept. For example, the job's registers are stored here when it is not executing. At the front of the job table are stored task number and the 8-character task name (MTS, HASPLING,...). [A task number of zero means this job table entry is not in use.]

Two items relating to the initiation of jobs must now be discussed. The Job Header is information attached to the front of the job program. The Job List Entry is essentially a "symbol-table" entry to the list of job programs in the system specifying the name of the job program and where its code is to be found.

Each job program is prefixed by a job header. The job header specifies the location in the job program of the first instruction to be executed, the number of preallocated devices and buffers which the job requires, the device type required for each device, and the size of each required buffer. The format of the job header is:



NJBDVU = Number of Devices Used

NJBBFU = Number of Buffers Used

An illustrative 360 coding sequence of a job using two devices and three buffers is shown below.

```
JOB     START   0
        DC      A(BEGADR)
        DC      H'2'
        DC      H'3'
        DC      CL4'PTR'
        DC      CL4'7TP'
        DC      F'128'
        DC      F'2048'
        DC      F'2048'
        .
        .
        .
BEGADR  DS      0H
```

The number of required devices are specified in the field NJBDVU and the number of required buffers are specified in the field NJBBFU. The device types for each required device must be given in the full words following the word specifying the number of devices. The size of each required buffer must be given in the full words following the device types. Device types, specifying the device requirement, are four characters, left justified, with trailing blanks.

The order in which the device names are specified determines

a logical device number (LDN) for each device, where the first device specified is logical device one. When a job program issues a supervisor call, the device to which the call is associated is indicated by the logical device number. In this way a job program can be written independently of the physical address of a device.

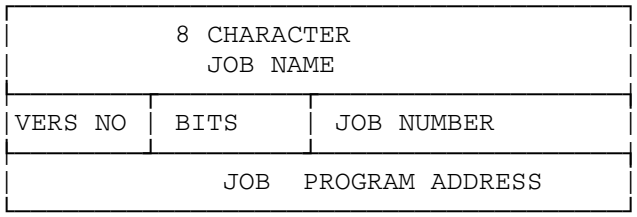
This preallocation of devices and storage is used only by small jobs (such as REW). The MTS job-program obtains its devices and storage dynamically. The MTS job header specifies no preallocation. Thus, all items entered by the operator after "MTS" are considered parameters and are stashed away for MTS to look at when it's given control.

Entry to a Job Program

When a job program is successfully initiated from the console typewriter, control is passed to the first instruction as specified in the Job Header. Three locations in the Job Table associated with the job are placed in General Registers 0, 1, and 2: General Register 0 contains the address of the pseudo Sense Switches, General Register 1 contains a pointer to the list of buffer addresses, and General Register 2 contains a pointer to the list of input parameters. If an input parameter is alphanumeric, it is right justified with leading blanks. If an input parameter is a decimal integer, it is converted to a four byte signed binary number. The list of input parameters is terminated with a fence of FFFFFFFF. If there are no input parameters, the first word of the parameter list will be the fence.

The Job List

For each Job which is to be run under the supervisor, there must be an entry in the Job List. The job List consists of a collection of fixed-length (16 byte) Job List Entries. The Job List Entry indicates whether the job is re-entrant and whether it runs relocatable, and gives the location of the job. Each Job List Entry is assembled as a separate subprogram, and contains the entry name of the job program as an Extern in the Job List Entry subprogram. The format of a job list entry is:



BIT 0 INDICATES THE JOB IS RE-ENTRANT
BIT 1 INDICATES THE JOB IS RELOCATABLE

The job list entry for the REW job is:

```
ARM5  START 0
      EXTRN JBREW                    ENTRY
      DC    CL8'     REW'            JOB NAME
      DC    C'1'                     VERSION
      DC    X'80'                    REENTRANT
      DC    H'0'                     NUMBER
      DC    A(JBREW)                 ENTRY ADDRESS
      END
```

The byte labeled Version can be used to indicate that a modification has been made to the job program. In the REW coding above, Version 1 is indicated. The "Bits" byte specifies whether the job is re-entrant and/or relocatable. Job programs may be written as re-entrant, whereby a single copy of a job program can be active for more than one task. If a job is re-entrant, the left-most bit of the "Bits" byte is set to 1. If the job-program's activation is to run in relocate mode, the second left-most bit is set to 1. MTS is an example of a job-program which is both reentrant and runs relocatable.

Associated with every active job is a task number which is used to identify the particular activation of the job throughout the system. If a job is not re-entrant, it can be active for only a single task. To indicate that a non re-entrant job is active, the task number is inserted in a field of the job list entry.

The Job List Entries are all collected together and sandwiched between a first-job (JOBLST), which defines the beginning of the "table", and a dummy last job (LSTJOB). The dummy last job has a blank name and version, and an all-ones job program address:

```
LSTJOB  START 0      LAST ENTRY IN JOB LIST
        DC    CL9' '
        DC    7X'FF'
        END
```

One thing should be mentioned about the parameter scan at this point. There are 14 words in the job table for parameters. The supervisor scans the input line from the operator; it doesn't just put four characters into a word in the job table; it

actually scans for blanks as delimiters. If it finds (between the blanks) any characters that are non-numeric it assumes it's a character string and it takes the last four characters and puts it in the word (right-justified with leading blanks). If it finds something that is all numeric, assumes it's a decimal number and it converts it, and puts it into the next word of parameters. If there are long names (such as *INIT) to feed to the program that's receiving these, such as MTS, they can't be entered directly. If five characters are typed in a row, the last four characters, ("INIT") are stuffed into the parameter. The characters must be separated:

*INI T,,,

The supervisor will put the characters into two contiguous words in the parameters. Trailing commas are used since MTS treats these as FDname delimiters. Trailing commas on the "T" are needed because otherwise the supervisor, (bless his heart), would right-justify it with leading blanks. [Another anomaly, device names in the system are left-justified with trailing blanks. Device types are right-justified with leading blanks.] This splitting is rarely used because it's such an annoyance that most of the pertinent file names are four characters, like *RST. [The string that HASP issues to start up an MTS batch job is rather astonishing.]

When the job is started the base register is established, and it suddenly finds itself at the front of its code. Three registers are set up: GR0 points to switches in the job table, GR1 points to the series of words which keep track of the storage buffers requested, and GR2 points to the first word of the parameters. Requested devices are referred to as logical devices 1, 2, and 3, for example. It's strictly in order of which they were specified and hence, the order of the parameters.

A job gets services -- that's the line marked "3" -- from the supervisor, by issuing SVC instructions, which cause an interrupt in the supervisor. That's the only way to get to the supervisor. The supervisor then processes the request and restarts the task. Anything that the task wants the supervisor to do is done by means of an SVC. There are about 100 SVCs now. The original Lincoln Lab supervisor has 20. A couple have since disappeared, and things have grown.

A job terminates by using an SVC. There's an SVC EXIT which says "I'm done." The supervisor calls a subroutine to clean up things, release things, and so forth. There's another SVC to intercept job stoppages (which includes SVC EXIT). MTS uses this SVC for maintaining control of things. Therefore, issuing an SVC EXIT does not always mean the job is stopping. For example, if the subroutine SYSTEM is called, the the first thing it does is issue an SVC EXIT, because the code to save all the registers, change state, and everything else is rather complex. Thus, there is only one copy of the code, and the first thing that happens on entry to SYSTEM is an SVC EXIT. The next thing MTS knows is that it is entered through the intercepted-exit section of code, and it finally discovers that someone did an SVC EXIT with a

particular address. Therefore, it calls SYSTEM.

HASP communicates in approximately the same way. HASP is initiated by the operator typing HASP and giving as parameters, possible drive names for disk packs, which theoretically should have disk packs mounted on those drives. When HASP starts off, it actually does a little more than the standard "3" interface and issues an SVC to tell the supervisor that HASP is running. This SVC gives the location of some special words in HASP since the supervisor sometimes has to make a special entry to HASP. If the operator types in a line beginning with a dollar sign, it's considered a command to HASP and the supervisor just passes it on by chaining it to a chain of messages for HASP to process, setting the appropriate flag bytes and posting HASP. If HASP is well behaved, it will look at the messages. So there's a slight additional interface here. That will be discussed more in the sections about HASP and HASPLING.

It was decided at the time HASP was being installed, that we would create a little entity called a HASPLING. HASP is not re-entrant. It multiprograms within itself, but there's only one HASP job running. It has lots of code that represent the "job programs" and it has something akin to a job table, which are called processor control elements. The HASPLING is a job program which is re-entrant, and one is activated for every device that HASP has doing things for it; i.e., one for each reader, printer, punch, remote SDA line, etc. There's also one for each HASP disk and one to handle messages to the operator's console (from HASP to the operator's console). The interface between the HASPLING and the supervisor is the standard one (3), and consists mainly of an SVC to start an I/O operation and a SVC to wait until the I/O operation is complete. HASP is the one that starts the HASPLING, by issuing an SVC which starts a task. The communication between the two (interface "4") is that HASPLING gets passed as parameters in the job request, the name of the device to manage and the location in HASP of some control information, a lock byte, and some pointers. This lock byte and buffer pointers is how the HASPLING gets its information of when it's supposed to write things out, or read things in. And that's also how HASP tells the HASPLING to go away, if it's through with it.

When the HASPLING has nothing to so, it does a variant of the SVC WAYT type of wait, SVC SLEEP. Both types wait for some bits to change to 0, and a return from that SVC does not occur until all the bits specified are 0. But for the SVC WAYT, the job has to stay on the CPU queue, and every time the supervisor goes to dispatch anybody, it checks those bits to see if they have changed, which is expensive. So, the SLEEP and AWAKE mechanism was generated. The SVC SLEEP says "we're doing a WAYT type of suspension, but take me off the CPU queue because someone will do an explicit type of interrupt to get me started again". When HASP wants to initiate something on a HASPLING it takes the task number and it does an SVC AWAKE which tells the supervisor to put that task back into the CPU queue. It also zeros the WAYT byte, of course, before it can go on.

The PDP (Paging Drum/disk Processor) interfaces with the supervisor (interface 10). It runs in absolute mode; it can't page itself. It runs in problem state as an absolute task, so it behaves like any other task, except that for efficiency there are some special things done in the supervisor. It uses a lot of standard SVCs, but there are some added exclusively for interfacing with the paging drum processor. It has a number of SVC's only it uses because it has to get information about where the queues are, which the supervisor is keeping. It's not re-reentrant.

There's also the JOBS program (now called SSRTN) which is really an external scheduler to the system. HASP and MTS get information from it, but don't pass information to it (interface 6). There is a region in storage with bits and numbers which HASP looks at to decide to start a new batch job, and MTS looks at for limited service state determination. The Jobs program performs the external scheduling (i.e., when should a task be started), and the supervisor does the internal scheduling.

The right hand side of Figure 1 shows the separation between absolute and relocatable. An interface between absolute and relocatable is necessary, and complex. This interface is supervisor assisted, in the sense that there's a series of SVC's to perform the moving across that boundary. This interface is less used now that HASP and the HASPLINGs are relocatable.

Proceeding further in Figure 1, on top of MTS, we have the collection of the device support routines. These do all the I/O to and from the MTS tasks, particularly terminal support, tape support, etc. That's the interface labeled "7". There's also another set of interfaces, the Command language Subsystem interface (12). One of these interfaces is the user program. A CLS is just a program but each CLS can be run independently of the others. For example, there's no distinction between the editor and a user program. The editor is written as a program, and it runs as a program but independently from the user execution program. A special case is one CLS, namely SDS, which has hooks into the user program CLS, since it has to monitor what is going on. There are a number of real CLS's plus two more. Level 0 CLS is MTS itself, the command mode, 1 is the user program, and 2 on up are the actual CLSs. (Editor, SDS,...) For symmetry's sake, it was made all the same.

Another interface is with the file routines. MTS calls the file DSR which communicates with the file routines (interface 8). They are designed so they could be called by an absolute task, although that's not done yet. MTS also calls some of the file routines directly (interface 9).

The loader is also called from MTS (interface 11), although it's also called when running the system from scratch and there isn't anything around but the boot-strap loader to load the loader. Then the loader loads everyone else. The loader interface is such that anybody can call it since it is entirely

self-contained. The loader looks at what it's given, decides if it's a good record and shoves it into storage.

For user programs, there are some SVC's that the user program will issue by means of a macro. These are the time of day, etc. But generally this type of interface is not used very much. The majority of supervisor services are obtained through MTS. (The DSR's however call the supervisor.)

This represents the minimum overall view of the system. Eleven subcomponents and several other things are included.

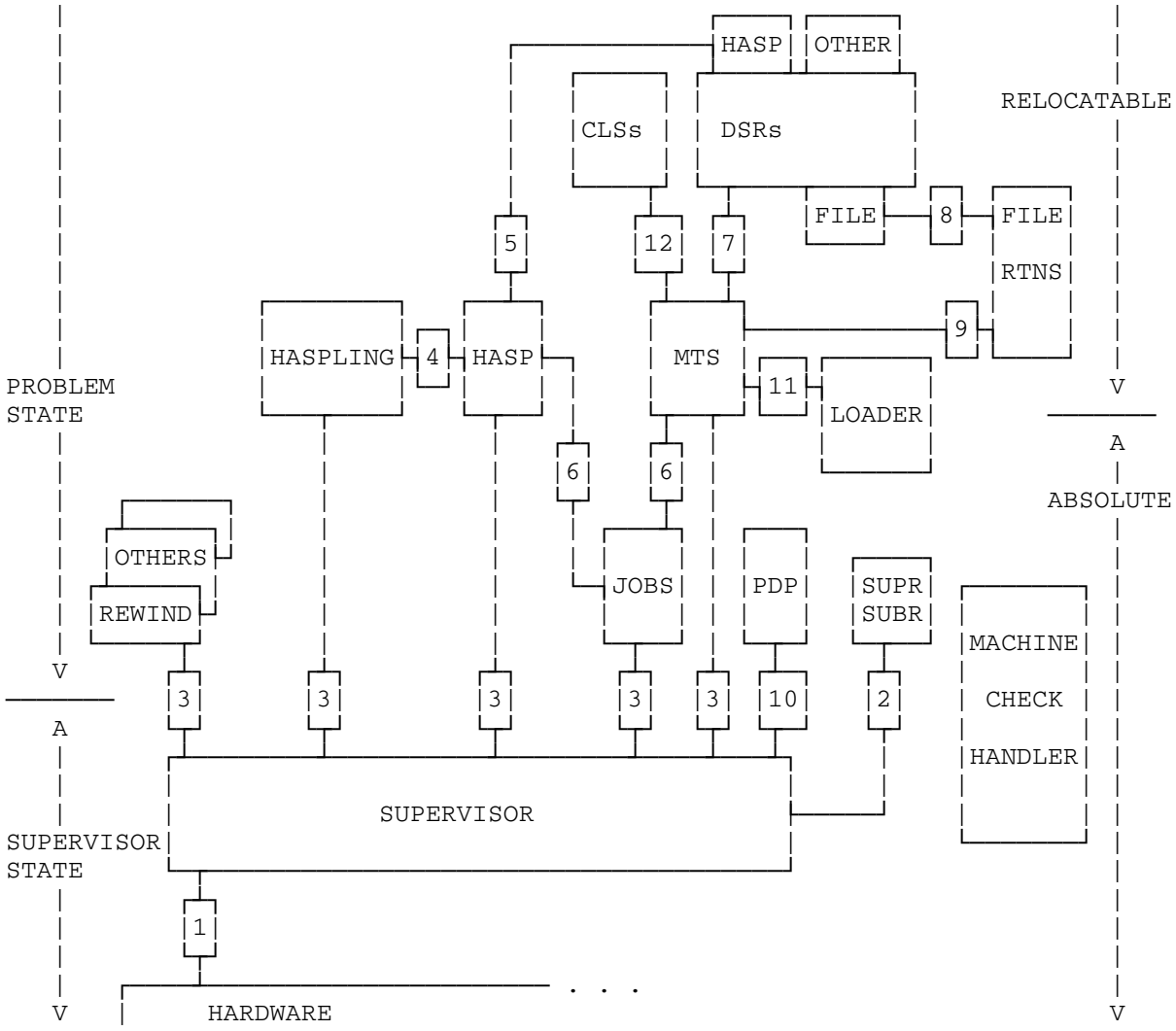

HASP
OTHER
RELOCATABLE
CLSs
DSRs
FILE
8
FILE
RTNS
5
12
7
9
HASPLING
4
HASP
MTS
11
LOADER
PROBLEM
STATE
V
A
6
6
ABSOLUTE
OTHERS
REWIND
JOBS
PDP
SUPR
SUBR
MACHINE
CHECK
HANDLER
3
3
3
3
3
10
2
SUPERVISOR
SUPERVISOR
STATE
1
. . .
HARDWARE

FIGURE 1

Classification: 100/4
Date: March 2, 1977
Doct=11 Vers=1

PRINTING JOBDUMPS AND SUPERDUMPS

The program in the file MTA:PRINTDUMP is used to print the jobdump and superdump tapes produced by the supervisor. Superdumps are taken automatically by the supervisor when it detects a fatal error within itself; all of real memory is written on the tape and system operation terminates. Jobdumps are usually initiated by operator request (typing "JOBDUMP n" to force a dump of job number n); all of real memory and virtual memory are written on the tape and system operation resumes.

System dump tapes are 9-track and labeled "DUMP". This program reads the tape on unit 1, reads commands from SCARDS, and prints the dump on SPRINT. SPRINT should generally be assigned to a printer (or *PRINT*) since large amounts of output are usually produced.

Output is controlled by commands as follows. In the command descriptions, "x", "y", and "z" may be a hex address, any symbol defined in the system symbol table (if the dump includes virtual memory), or a sum of either of these. On the "C" or "F" commands, if "ST=segt" is given, all addresses are relocated using the segment table at <u>real</u> address "segt."

C	x	y	Dump <u>real</u> memory from hex address x to y or
C	x		one word at x (form 2).
V			For jobdumps only, dump job table (and
V	x		related tables) and virtual memory pages x
V	x	y	through y or x only for the job which was dumped. If neither x nor y is given the job's VM is dumped from page 400 to page FFF. Either x or y may be either a page number or an address, but dumping always begins and ends on a page boundary.
V	MTS		Dump job table and virtual memory, formatted assuming an MTS task.
P			Dump the supervisor I/O, CPU, and WAYT POOLS.
S			Dump the supervisor error information and PSAs of all active CPUs.
S	F		Dump the supervisor error information, PSAs, TABLES, and all pages allocated to the supervisor.
S	C		Check supervisor storage and dump all blocks that are not part of a page belonging to the

supervisor, or which are not accounted for. A block is accounted for if it is on a free space chain or if it is pointed to by some field in a job table or device table. Note that many legal blocks will appear to be unaccounted for.

J		Dump the job table (and related tables), but
J	n	not virtual memory, for the job which was
J	ALL	dumped, for job number n (form 2) or for all jobs (form 3).

T	Dump the supervisor trace table.

F	x	y		Follow a chain of storage blocks and dump
F	x	y	z	each one. X is the location of the pointer to the first block (<u>not</u> the location of the first block), y is the length of each block, and z is the displacement from the front of a block to the pointer to the next block (assumed 0 if not given).

MTS	Return to MTS. $RESTART will return to PRINTDUMP.

U	params	Same parameters and output as the UNITS operator command.

L	Print part of the LOADCLAS output giving a count of the number of tasks in each of the states that TASKSTAT recognizes.

I	cmd	Dumps the in core file table in a form similar to TABLMOD. "cmd" must be a TRACE, VERIFY, DUMP, LSTAT, or FIND command in a form suitable for TABLMOD.

Any input line beginning with a "$" is assumed to be an MTS command and is passed to MTS to be executed.

All commands begin in the first column with parameters separated by one or more blanks.

Attention interrupts are handled as follows: a message is printed on SERCOM acknowledging the interrupt. If PRINTDUMP is doing something that can't be interrupted, it will print a message saying so and continue. A second interrupt will be processed by MTS. When PRINTDUMP can be interrupted it will prompt for an input line from GUSER. This input line can be RES to continue the command interrupted, or any PRINTDUMP command to terminate it.

Example (in batch):

```
$MOUNT DUMP17 9TP *DUMP* VOL=DUMP
$RUN MTA:PRINTDUMP 1=*DUMP*
TRACE
VM
CORE 7F10 A768
```

Date: 6/22/72 Doct=5 Vers=1
Author: Jim Hamilton

MTS-UMMPS Storage Allocation and Selected Applications

I. General Requirements
   A. Must be completely general, i.e. must provide variable size blocks

   B. Since storage allocation structures exist for as long as the system is up, storage must never be permanently "lost" due to causes such as fragmentation. Hence most structures are maintained in increasing location order.

   C. Must obviously be application independent, hence such things as garbage collection, reclaimation, compaction, etc. are impossible. The only relocation possible is that provided by the DAT hardware, hence the mechanisms will often be page-oriented.

II. Storage Allocation for Supervisor Subroutines

   A. Requirements and properties

      1. Speed - must be very fast, for commonly used block sizes (e.g. PCBs) because of heavy usage

      2. Must never run out of space, since the system will crash if this happens; paging, plus some care in coding, avoid this problem

      3. Supervisor code is "dependable", so little error checking need be done.

      4. Storage demands are, in a sense, fixed, since the supervisor itself is a closed system (requests from tasks are considered separately in the next section).

   B. Pools

      1. For very fastest allocation of fixed size (8 byte) entries for

         a. CPU Queue

         b. WAYT Queue

         c. I/O Queue

      2. Separate, pre-allocated areas with space for 225 of each type

      3. Free space is simple linked list, done with offsets

      4. Use of pool index allows "pointers" to fit in one byte

C. The Page Chain

With the exception of the pools, all dynamically allocated storage is taken from, and occasionally returned to, a page chain which is just a simple linked list of available pages. The page chain is constructed at initialization. All other available storage structures are initially empty.

The Supervisor never deals with blocks of real core larger than a page.

D. GRAB-FREE Subroutines

1. For every block size less than SVCASPEC (currently 96 bytes) there is a special chain containing blocks of the corresponding size. Calls to FREE always return small blocks to these chains, which are kept in LIFO order. Calls to GRAB will take a block from the proper chain if it is non-empty; otherwise it is allocated as described in 2.

2. There exist two chains of arbitrary size blocks, maintained in increasing location order to allow recombination. One is for blocks smaller than SVCABIG (currently 1024 bytes) and the other for larger ones.

   All blocks larger than SVCASPEC are returned to these chains, blocks of any size less than a page are allocated from them, using a first fit algorithm, and splitting blocks when necessary.

3. If a block of the desired size or larger is not available, take a page from the page chain if there are any, and add it to the large or small chain according to size of the request, and try again.

4. If the page chain is empty, begin moving blocks from the special chains for small blocks to the arbitrary size chains, and continue until a large enough block has been found, or until all are moved. The latter case is followed by a superdump.

5. An example storage layout is shown in Figure 1.

III. Storage Allocation at the Task Interface

A. SVC GETBUF, GETSEGX, and FREEBUF

1. Absolute tasks (GETBUF):

   a. Maximum of 4 buffers allowed

   b. Maximum size one page

c. Does some error checking, then calls GRAB or FREE

d. The address and length of the allocation are recorded in a 32 byte table pointed to from the job table (4 buffers X 8 bytes = 32 bytes). This is done so that the storage can be recovered if the task goes west.

2. Relocatable Tasks

a. GETBUF implies segment 4, GETSEGX specifies any of 3 through 8

b. All requests rounded up to a page boundary

c. For each task there is a PCB (Page Control Block) chain maintained in increasing virtual address order, one PCB per allocated virtual page.

d. Supervisor scans PCBs until it finds a large enough hole in the desired segment, then GRABs the required number of PCBs (24 bytes each), initializes them and inserts them in the chain, it never explicitly allocates real core. It implicitly references the first page, however, into which it stores the length of the allocation.

e. For FREEBUF the PCB chain is scanned for the PCBs describing the freed region. The real core pages, if any, are put back on the page chain, and the PCBs are removed from the task chain, and the Page Out Queue if necessary, and put on the Release Page Queue, where we will leave them for this discussion. Actually the FREEBUF routine is horrendously complex (more so than any other routine described here) but most of its complexity is irrelevant to us.

B. SVC GETSC, FREESC

1. For absolute tasks only.

2. These go directly to GRAB and FREE after checking to be sure there is at least one page on the page chain.

C. SVC GETRP, FREERP

1. Used only by the PDP to get or release pages of real core to be attached to PCBs.

2. GETRP just takes a page from the page chain if there

are at least two pages there.

3. Otherwise it scans the small and large chains of supervisor pages and moves any full page blocks it finds over to the page chain. If it found any, it tries again. Otherwise it gives up, and the PDP will try agian later.

4. FREERP essentially just adds pages to the page chain, unless they have been reclaimed or released.

IV. Storage Allocation in MTS (GETSPACE/FREESPAC)

A. General Requirements and Properties

1. Since this is the user interface, thorough error checking must be done.

2. It must be possible to recover the storage which has been allocated, in case user programs or various levels of system programs go west, or even when they terminate normally.

3. Storage management structures are maintained in system level storage, separate from the storage being managed; there are two reasons:

   a. It is undesirable to reference VM pages before they are needed.

   b. The user cannot be trusted to confine his references to storage which is allocated to him.

4. The VM integral must be computed, to keep the accounting people happy.

b. Buffer Control Blocks (BCBs)

The storage allocation structure is composed of fixed size (16 byte) buffer control blocks. These too must be allocated and freed. This is done using a conventional LIFO free space list. One page of BCBs is allocated initially (via SVC GETSEGX) and more are allocated if necessary, a page at a time. All BCBs are in segment 4.

C. Storage Management Structures

1. One may wish to look at Figure 2 while reading the following:

2. Primary buffers: Storage requests obtained from the supervisor via SVC GETSEGX are called primary buffers. Each primary buffer is described by two primary BCBs (PBCB1 and PBCB2) and one or more sub-

buffer BCBs (SBCBs). VMI accounting is done at the primary buffer level.

3. Each primary buffer may be divided into one or more sub-buffers. Each sub-buffer is described by one SBCB. The SBCB also describes the free block following the allocated block.

4. The PBCB1 blocks are linked in increasing location order, and each points to a PBCB2 block, which points to a list of SBCBs, which are also in increasing location order. There is a separate list of PBCB1 blocks for each segment.

D. Detailed BCB Definitions

1. PBCB1 (all entries are fullwords)
   a. length of longest free block in buffer
   b. location of first byte past end of buffer
   c. link to next PBCB1
   d. link to PBCB2 for this buffer

2. PBCB2

   a. number of sub-buffers (2 bytes)
   b. length of buffer in pages (2 bytes)
   c. location of first byte in buffer (4 bytes)
   d. number of free bytes before first sub-buffer (4 bytes)
   e. link to first SBCB

3. SBCB

   a. storage index number (1 byte) (read volume 5 to learn about these)
   b. length of allocated block (3 bytes)
   c. location of first byte after block (4 bytes)
   d. length of free block following (4 bytes)
   e. link to next SBCB

E. GETSPACE Algorithm

1. Search PBCB1 list for desired segment, looking for first one which has a free block equal to or longer than desired. If not found, go to step 4.

2. Search the SBCB list (including PBCB2) for the first free block equal to or longer than desired. There must be one, or we blew it. Insert a new SBCB after it to describe the new allocation and any remaining free block.

3. If the free block found in 2 is equal in size to the largest free block recorded PBCB1, the SBCBs must be searched again to find the new largest free size. Then we are done - return.

4. Issue SVC GETSEGX; if this fails, go to step 5. Get and initialize a PBCB1, a PBCB2, and one SBCB, describing the requested allocation, plus any remaining space in the last page. Search the PBCB1s and insert the new one at the appropriate place. Compute the VMI and new total page count.

5. Must be insufficient space left in this segment. If a specific segment was requested, or if we have already tried segment 8, tell somebody about this problem. Otherwise, increment the segment number by one and go to step 1.

F. FREESPAC Algorithm

1. Find the SBCB whose allocated block completely contains the one being returned. This is a two step process, going down first the PBCB1s, then the SBCBs. It is a nono if its not there.

2. Compare the free block to the allocated block. There are 4 cases.

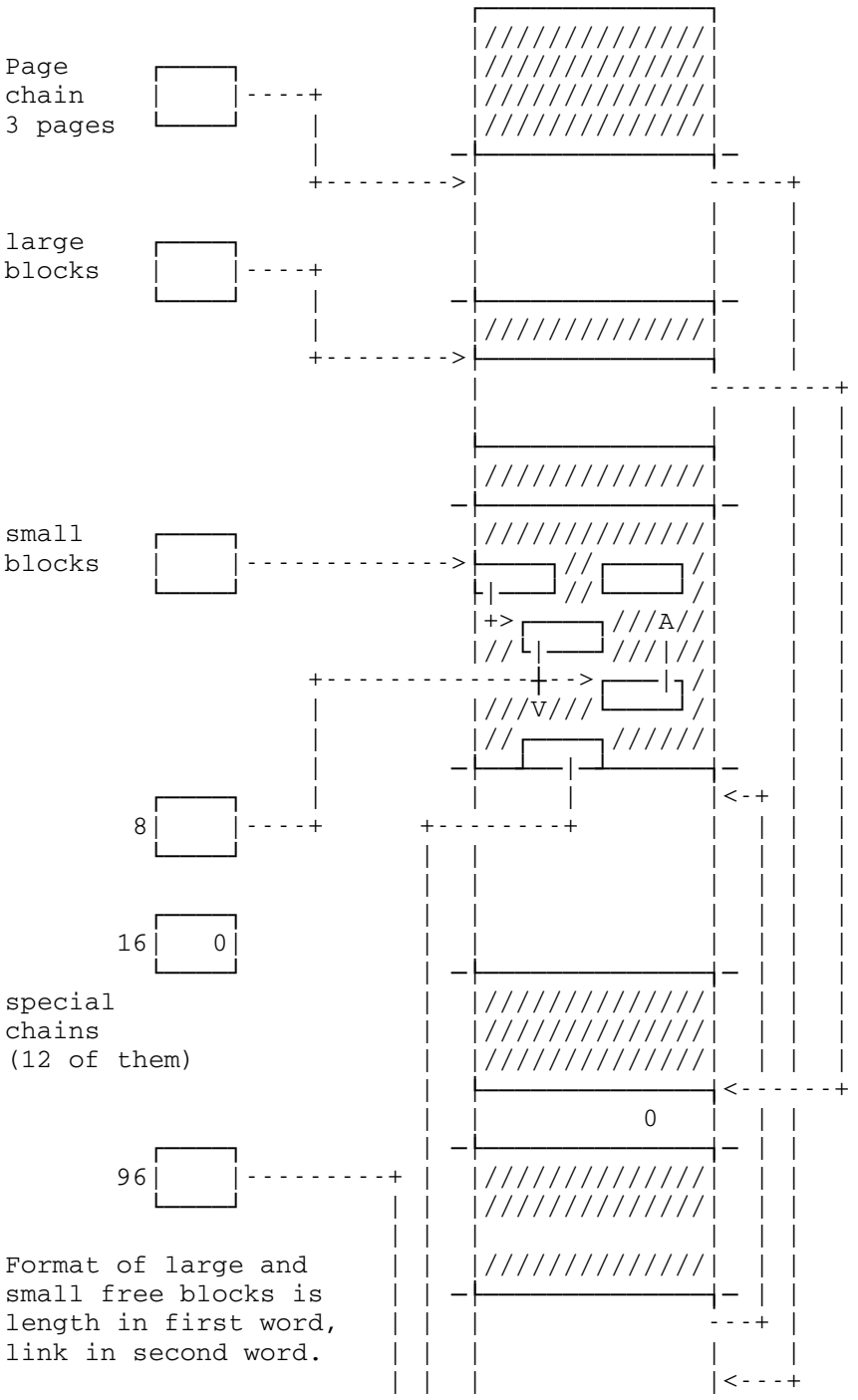
   a. same - this is the normal case. Update field d. of the previous SBCB with the sum of itself and fields b. and d. of this SBCB. Unlink and free this SBCB. Update the maximum free block in PBCB1 if necessary. Decrement the subbuffer count (in PBCB2); if this is zero, the entire buffer is free, so issue SVC FREEBUF, unlink and free the PBCBs, and do VMI accounting. Return.

   b. Starting addresses the same, Update this and previous SBCB according.

   c. Ending addresses the same. Update this SBCB accordingly.

   d. Neither address is the same. Get a new SBCB and insert it before the current SBCB. You should be able to figure out how to diddle the various fields.

3. Update the maximum free length in PBCB1 if necessary, and return.


Page
chain
3 pages
large
blocks
small
blocks
8
16
0
96
special
chains
(12 of them)
A
V
0
Format of large and
small free blocks is
length in first word,
link in second word.

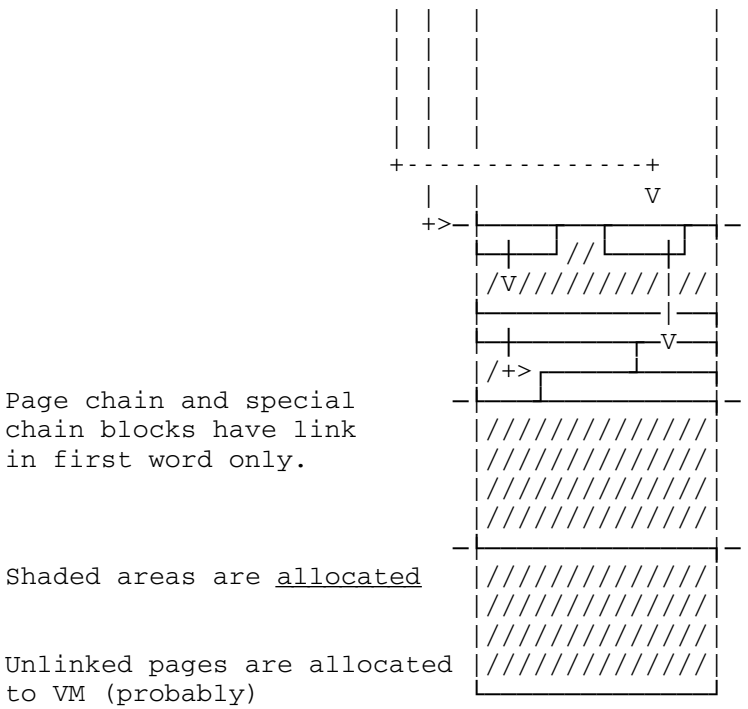
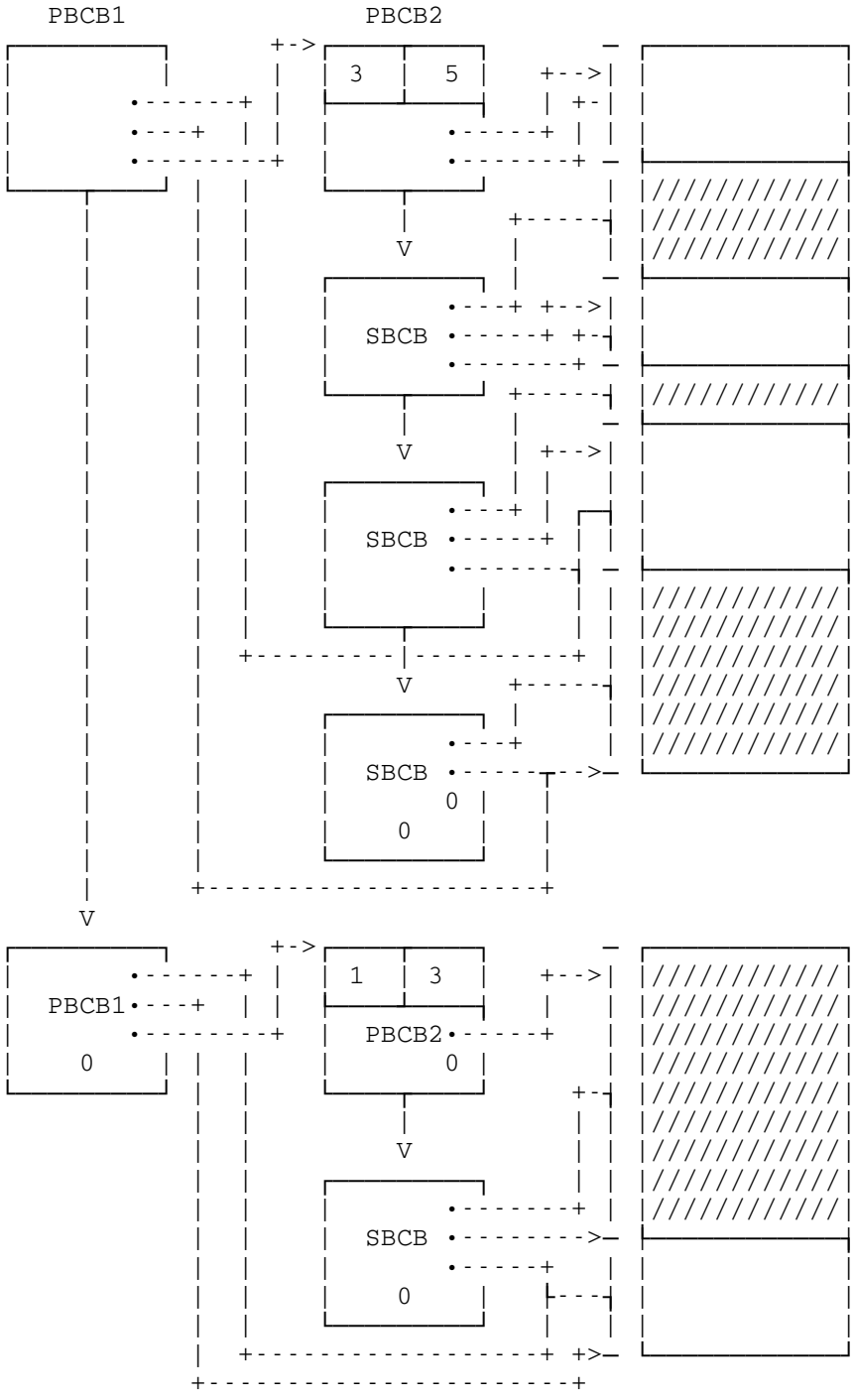


Figure 1 - Supervisor Core Management



Shaded areas are allocated

Figure 2 - MTS Storage Allocation

UMMPS PAGING ALGORITHM

I. Before descending into the details of UMMPS and the PDP, it is probably instructive to say a few words about paging algorithms in general. They may differ in several important ways:

1. Demand paging vs. Anticipatory paging: under demand paging, a system will move a page to main storage only when it is referenced. On the other hand, a system may attempt to anticipate the need for some pages, and page them in before they are referenced. Almost all current systems use demand paging.

2. The algorithm may be task oriented, or system oriented. That is, the decision as to which pages to move to and from main storage may depend heavily on the status of the task which owns them; this would be a task oriented algorithm. With a system oriented algorithm all pages in the system are treated identically, independent of their owners.

3. The replacement policy, for choosing pages to be removed from main storage, may vary considerably. This is probably the most important factor affecting the performance of paging systems. There are several possibilities discussed in the literature:

   a. FIFO - The oldest page in storage is chosen for removal. This is clearly not a very good choice, but early versions of UMMPS used it.

   b. Least Recently Used (LRU) - The page with the longest time since last reference is chosen for paging out. Note that this algorithm can only be approximated on the 360/67, and most other current processors.

   c. Working Set - The system keeps a record of recent references to pages by a task and attempts to keep a "working set" of pages belonging to a task in core. This is a task oriented policy, which attempts to estimate program behavior. It is said to be a nearly optimal realizable algorithm.

   d. A-Optimal - The page whose next reference is farthest in the future is chosen for removal. This is, in some sense, an optimal algorithm, but is unrealizable without knowledge of future page references. It is mainly a standard for comparison.

UMMPS uses basically a system oriented demand paging algorithm with an LRU replacement policy. The supervisor is totally responsible for these aspects of paging and their implementation is found in the GETWP SVC, which will be described in some detail later. The Paging Drum Processor

(PDP) is responsible for the actual transfer of pages to and from the paging devices. In the remainder of these notes we describe 1) the UMMPS - PDP interface, 2) the PDP, 3) UMMPS (mainly GETWP), and 4) a day in the life of the average page.

II. The UMMPS - PDP Interface

A. Data Structures:

The primary data item containing information relevant to paging is the Page Control Block (PCB). A PCB is 24 bytes long and contains the following items:

1. Virtual address
2. Real Address
3. Pointer to owning TCB
4. Task page chain pointer
5. System Queue pointer
6. Reference Bit
7. Change Bit
8. External Address

plus several other items which don't concern us here. PCBs are created by the GETBUF SVC, and are released by the PDP.

There are four queues used by the system in managing paging. These are:

Page-In Queue (PIQ) - Pages to be brought into core from secondary storage.

Page-In Complete Queue (PICQ) - Pages which have just been brought into core.

Page-Out Queue (POQ) - Pages which are in core, ordered approximately from least recently used to most recently used.

Release Page Queue (RPQ) - Pages which have been released (via FREEBF SVC).

B. Special SVCs

There are five special SVCs which are used only by the PDP. These are:

GETRP - Get real page - Used to get a real page of core to read into, for a page-in operation.

FREERC - Free real core - Used to release the core allocated to a page after it has been paged out.

GETWP - Get Write Pages - Removes a specified number of pages from the POQ, to be written out.

GETQS - Transfer the contents of the PIQ and RPQ to the PDP.

PDPWAIT - Tells UMMPS the PDP has nothing more to do. It will be restarted by a completion interrupt from any of its devices, or when UMMPS decides there is more for it to do (i.e., PIQ non-empty, or pages need to be written.

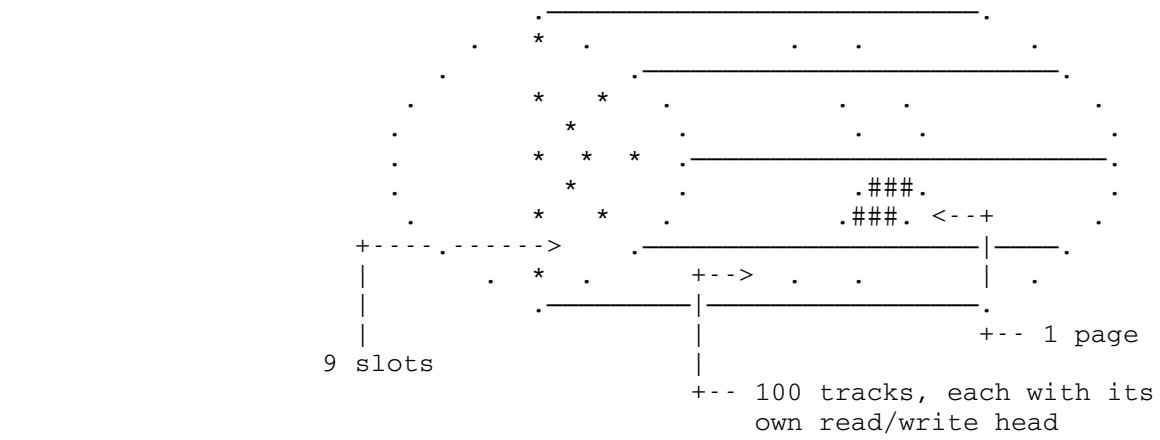
III. The Paging Drum Processor

A. Overview - It is the responsibility of the PDP to manage the paging devices, which currently include two drums and one disk. Each drum holds 900 pages, and the disk holds 6400 pages, for a total of 8200 pages. The worst case observed to date has been something over 4000 pages, and a typical heavy load is between 2000 and 2500 pages.

The disk is used only when the drums are nearly full, and at this time the PDP chooses pages on an LRU basis and moves them from drum to disk. This is called page migration.

The PDP consists of two asynchronous parts: the first builds channel programs and starts them via SVC STIO, and the second handles the completion interrupts and posts the completion of the paging operations.

B. In order to understand the operation of the PDP, it is necessary to understand the workings of the paging drum.



The picture shows the logical structure of the paging drum. Physically there are 200 tracks, with 4 1/2 pages per track, but the PDP treats it as shown, with the difference obscured by a trick in the channel programs.

The PDP constructs channel programs for all nine slots at a time. It will then chain these together if possible. The PDP is so designed that the drums can be kept running

for an indefinite time with only one SIO, with reads in the appropriate slots, and with writes filling in the rest as necessary. Using this method, writes (i.e., page-outs) are essentially free.

C. PDP Data Structures - The PDP maintains a huge data block for each drum. Each such block contains, among other things:

1. 9 Local Page-in queues, one for each slot.
2. 3 channel program buffers

3. Spaces for 27 PCB pointers for the PCBs associated with the 3 possible channel programs.

4. A bit table describing available space on the drum; organized by slot.

5. 9 migration lists, one for each slot of PCBs ordered from least to most recently used. ("used" in this context means paged-in or paged-out.)

6. A local page-in complete queue

   There is also a data block for the disk. Since the disk is managed on a page at a time basis, with no attempt at optimization, this data block contains only one local page-in queue, one channel program, one current PCB pointer, and the bit table.

   There is a "global" local page-in complete queue, on which the local PICQs from each device are collected, and whose contents are occasionally transferred to the supervisor's PICQ.

D. The algorithm

1. Get the PIQ and RPQ via the GETQS SVC.

   1.1 For each PCB on the RPQ, release its external address, free its real core page via SVC FREERC, if there is one, and free the PCB, via FREESC SVC (Free Supervisor Core).

   1.2 For each PCB on the PIQ, add it to the end of the local page-in queue for the appropriate slot on the appropriate drum, or to the LPIQ for the disk. This process is called slot sorting. If the PCB has no external address, put it on the local PICQ now, since it must be a new page.

2. For each drum do the following:

   2.1 Allocate a new channel program buffer if possible. If not, go try the next drum.

2.2 For each slot: if the LPIQ for the slot is non-empty, remove the top PCB, get a real page via SVC GETRP if possible, and construct a CCW to read the page in. If no core is available, go to step 2.3 immediately.

2.3 If there are slots available which don't contain reads, check drum availability; if there are less than MIGTH pages left on all drums, and if no migration is currently in progress, take a page from the top of the migration list for one of the available slots, and construct a CCW to read it into a page of supervisor core. Remember that a migration read has been started. MIGTH, the migration threshold, is currently set to 50 pages per drum, or a total of 100 pages, with 2 drums. This will probably be reduced in the future.

2.4 If there are slots available which have no reads, and which have unused tracks for writing, issue SVC GETWP, requesting as many pages as there are available slots.

2.5 For each PCB returned by GETWP, see if it has been changed. If not, and it's on the drum, just issue SVC FREERC. If it has been changed, or is on the disk, free its existing external copy, and construct a CCW to write it into one of the available slots.

2.6 If there are still some available slots because of unchanged pages, issue another GETWP, and go to step 2.5.

2.7 If any reads or writes were set up in 2.2 through 2.6 above, package up the channel program and either issue SVC STIO if no channel program is currently running, or chain it to the end of the current channel program, with a TIC command.

This completes the setting up of channel programs for the drums.

3. For the disk, do the following:

3.1 If the disk is already running, do nothing.

3.2 If the local PIQ for the disk is non-empty, issue GETRP for a real page, and construct a CCW to read it in. Go to step 3.4.

3.3 If a migration read was set up in 2.3, then allocate a disk page and construct a CCW to

write it out.

3.4 If anything was done in 3.2 or 3.3, complete the channel program but modify it so only the seek is done, then issue SVC SIO. This way the channel is not busy during the seek.

This completes the setting up of disk channel programs.

4. Collect the local page-in complete queues from the several devices and add them to the "global" local PICQ. If there are any new pages on this queue, issue a GETRP for them. If GETRP fails, keep these pages on this local PICQ, but put all completed pages on the supervisor's PICQ. The supervisor will eventually find them there and restart the waiting tasks. If any channel programs were started above, go to step 1. Otherwise PDPWAIT. This completes part one of the PDP.

The remainder of the PDP consists of device-end and PCI (program controlled interrupt) interrupt routines. The PDP arranges to receive a task interrupt at the completion of any of its channel program buffers, i.e., once every logical drum revolution, from each drum. At such times it does the following steps:

5. For each PCB in the channel program just completed, do the following:

5.1 Add it to the bottom of the migration list for the appropriate slot, if this is a drum.

5.2 If it is a read operation, add the PCB to the local PICQ for this device. Go to step 5.4.

5.3 If it is a write operation, free the real core page, via FREERC.

5.4 Free the channel program buffer.

5.5 If this was a device end, mark the status of the device as stopped.

5.6 Return and re-enable the interrupt.

This completes the description of the PDP algorithm. Many details of the actual implementation have been omitted for simplicity, but most of the important ideas are there.

IV. The Supervisor

In this section we discuss the algorithm for the five PDP SVCs, plus the subroutine TRANS, which is called to handle paging exceptions, among other things. All but TRANS and GETWP are, at least conceptually, simple, but all are mentioned briefly, for completeness.

PDPWAIT - Save the TCB pointer for the PDP (this is the only way the supervisor knows which task is the PDP). Remove it from the CPU queue. Save the restart address (a parameter in GR0)

GETQS - Lock the PAGQ lock, pass the PIQ and RPQ pointers to the PDP, set these pointers to zero, unlock, and return. This must be an SVC because only the supervisor can do the required locking.

There are several variables which control the page replacement policy, as implemented in the GETRP, FREERC, and GETWP SVCs. These are:

1. NFRPGS - Number of free pages available
2. MINFRPGS - Minimum number of free pages which must be maintained, currently = 1
3. NWRTPGS - Number of pages being written out
4. WRTFRPGS - The threshold for deciding when to write pages, currently = 15

GETRP - If the number of free pages is greater than or equal to MINFRPGS, remove a page from the free page chain and decrement NFRPGS. Otherwise indicate that no page is available.

FREERC - Add the page to the free page chain and increment NFRPGS.

GETWP - A little more complicated

1. If NFRPGS + NWRTPGS > WRTFRPGS, return zero pages.
2. For several reasons, the proper operation of GETWP requires that no CPU be relocatable. Therefore, if the other CPU is relocatable, a WRD instruction must be executed at this point, which causes an external interrupt to the other CPU. A flag is then set which will hold up the other CPU until GETWP finishes its work and resets the flag.
3. Starting at the top of the POQ, do the following for each PCB encountered, until either getting enough pages to fill the request, or until reaching the end

of the POQ.

3.1 Update the reference and change bits in the PCB with those in the storage keys for the real page.

3.2 Set these bits in the storage keys to zero.

3.3 If the page has not been referenced, add it to the list of those to be paged out. If it belongs to a non-privileged task, page it out anyway. If it has been referenced, reset the referenced bit and move it to the end of the POQ.

3.4 If a page which has not been referenced belongs to a task which is running on the other CPU, don't page it out; instead leave it on the POQ.

3.5 Each page to be paged out is removed from its page table.

4. Update NWRTPGS, and return.

TRANS - A supervisor subroutine called by anything which needs to reference a virtual address, which means mainly paging exceptions, but includes other parts of the supervisor as well.

The algorithm for TRANS is:

1. Try an LRA instruction, if this works, return.

2. Search the task PCB chain, or shared PCB chain if segment 2. If not found, simulate program interrupt 5.

3. If the page is being paged in already, chain this request onto the previous and go schedule another task.

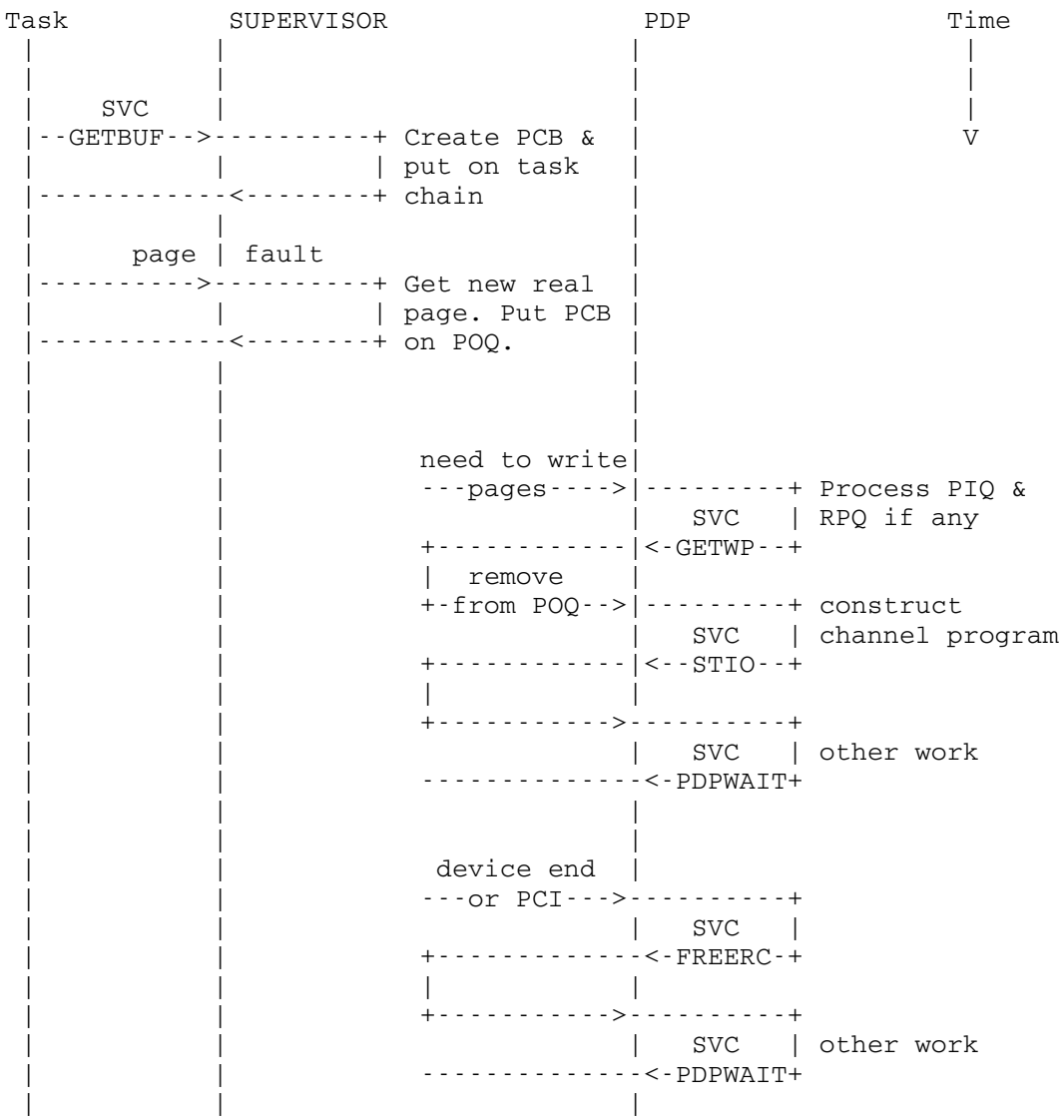
4. If the page is being paged-out, mark it as reclaimed, update the page-table, and return.

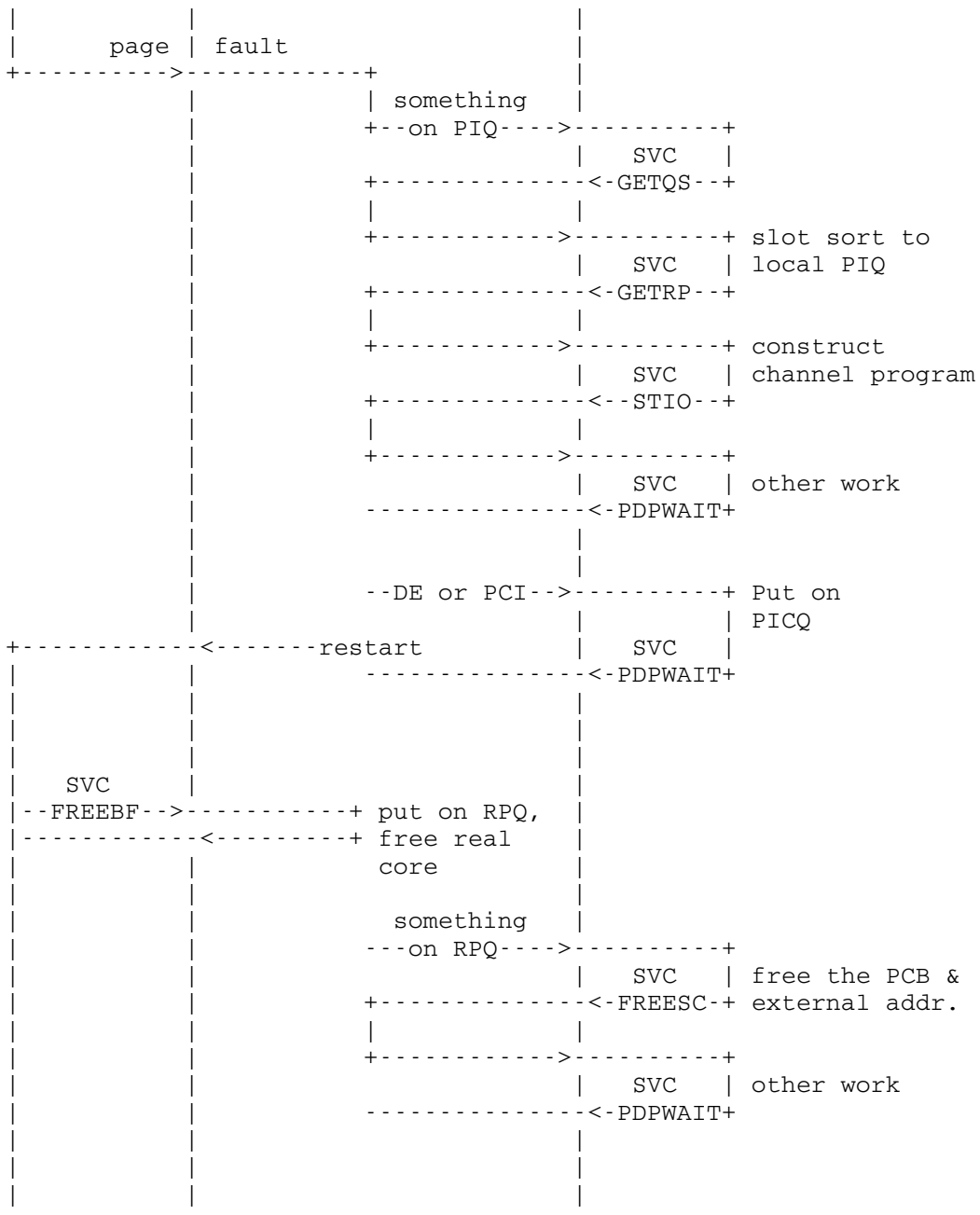
5. If the page has an external address, put the PCB on the PIQ, and schedule another task.

6. If the page has no external address, it must be new; try to get a real page for it. If successful, return, with the page table updated. Otherwise put it on the PIQ, and the PDP will retry.

V. A Day in the Life of the Average Page

The chart of the next page illustrates the various transitions which may be encountered by a page during its lifetime. Note: only those SVCs are shown which deal with the particular page we are watching.




page fault
something
on PIQ
SVC
GETQS
slot sort to
local PIQ
GETRP
construct
channel program
STIO
other work
PDPWAIT
DE or PCI
Put on
PICQ
restart
SVC
PDPWAIT
SVC
FREEBF
put on RPQ,
free real
core
something
on RPQ
SVC
free the PCB &
external addr.
FREESC
SVC
other work
PDPWAIT

Introduction to MTS

This section covers the MTS job program and its relationship to those components it is most intimately connected to -- DSRs, CLSs, the file routines, and the loader. It is assumed that the reader has read the system overview (lecture 1). See figure 1 for a pictorial overview of the system.

In general usage, the term MTS applies to the entire operating system, including the supervisor, the job programs, and even a few of the utility programs. For the purposes of this section, the term MTS will be restricted (usually) to only apply to the job program of that name. The many other pieces of software intimately connected with MTS will normally be identified by their proper names (DSRs, CLSs, file routines, etc.)

MTS is a job program, and as such, gets initiated by the supervisor. Since it is reentrant, many MTS jobs can be active at the same time. The MTS job program provides the interface between the system and the user -- it executes his commands and monitors (and provides services for) his programs. There is a separate MTS job (activation of the MTS job program) for every user, both batch and terminal, in the system.

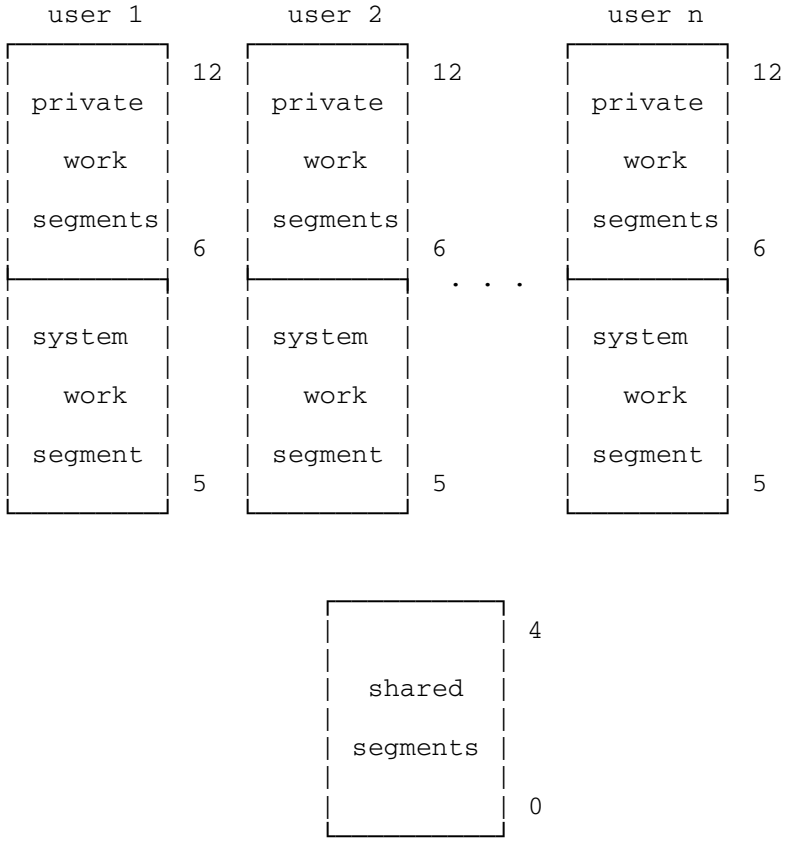
The other pieces with which MTS is most intimately connected are illustrated in figure 1 -- UMMPS (the supervisor), the DSRs, the file system, the loader, the CLSs, and HASP. There is a separate interface between MTS and each of these components. MTS and all of the pieces which it communicates with, except for the supervisor, run in problem state. MTS and all of the pieces with which it communicates directly (without going through the supervisor) run in relocatable mode. The relocatable components all can be paged by the PDP.

Addressing Structure of MTS Tasks

The addressing structure of MTS (the whole system) is one of the things which sets it apart from most contemporary time-sharing systems. Each relocatable task has its own virtual address space consisting of (currently) thirteen segments of 256 pages each. Because each task has its own unique address space, enforced by the dynamic address translation hardware, information in the address space of one task does not necessarily appear in the address space of another task. Independence of address spaces is clearly desirable in the case of user programs and data. However, many of the system-provided programs, like MTS, must be executed by all MTS tasks and thus must appear in the address spaces of all of them. In order to avoid duplicate pages of these programs in every MTS task's address space, some means of allowing different address spaces to intersect is required. This facility is provided on a segment basis using "address agreement." That is, a segment may appear in two or more address spaces, so long as its segment number is the same in all address spaces. This means that every location in a shared segment has

the same address in every address space in which that segment appears. This allows address constants which refer to locations in that segment to be embedded in the segment and still be valid for all tasks which share the segment.

The lower segment numbers in the address space of each task are used for shared segments. Currently, segments 0-4 are common to all tasks' address spaces. See figure 2. These shared segments are used to contain all of the resident system code (UMMPS, MTS, CLSs, file system, etc.), many shared programs (*APL, *FTN, PL1LIB, etc.), and several tables and data structures which may be accessed by all tasks.



Use of the Virtual Address Space in MTS

Figure 2

Of those components loaded into shared VM, some (primarily UMMPS) must be run in absolute mode. In order for the virtual addresses of locations in such components to reference the same locations as the real addresses of such locations, any absolute component must be loaded into segment 0 with the virtual addresses being the same as the real addresses. Thus, virtual segment 0 corresponds to the real memory of the machine. This is necessary primarily for tables which must be referenced from both absolute and relocatable tasks and for those programs, like the loader, which must be called from both absoulte and relocatable tasks.

The remaining segments in a task's address space are private to that task. Of these segments, one (currently segment 5) is reserved for system (i.e. MTS) work space specific to the task, and the rest are available for user programs and data. Again, see figure 2. The system segment is used for almost all work space and dsects required by system components.

The Relationship between MTS and its Support Components

MTS communicates with many pieces of the system, as illustrated by figure 1. However, some of the relationships are more intimate than others. The principal components MTS communicates with are the file routines, the loader, the DSRs, and the CLSs -- aside from the supervisor, of course. In fact, these four components may be thought of as logical extensions of MTS -- in contrast to the supervisor, which provides support for all of these things. The relationship between MTS and HASP is somewhat special and certainly less intimate (not to mention less elegant) than the others.

The DSRs

The most important related component is the DSR -- device support routine. The function of a DSR is to provide the interface between MTS and a real device, e.g., the user's terminal. Since there are many different types of devices with different characteristics this is a reasonable way to keep MTS itself independent of devices. There is a DSR for each different type of device with which MTS can communicate, including teletypes, 2741s, front end processors, audio response units, disks, and HASP.

The interface between MTS and DSRs is basically a subroutine call with a specialized calling sequence. Each DSR contains a transfer vector with ten entries, each of which is the address of an entry point within the DSR which will perform a particular, standardized, function. MTS simply loads the adcon corresponding to the function it wants out of the transfer vector and branches to it. Not all of the DSRs provide all 10 of the functions, since some are not applicable to particular devices. For those unsupported functions, the DSR must supply either a zero adcon or a dummy routine.

The File Routines

The file routines are the next most important MTS-related component. The interface between MTS and the file system is somewhat less distinct than the DSR interface. File I/O, in general, is done just like any other I/O in MTS -- that is, there is a DSR for the file routines with which MTS communicates just like any other DSR. This DSR, like a few other special DSRs, is actually a part of the MTS assembly. The file DSR calls several entry points in the file routines using non-standard calling sequences (i.e. the calling sequences vary from routine to routine). Operations peculiar to files such as renumbering, creating, etc., which have no normal DSR analog are performed by calling entry points in the file routines.

The file routines basically perform all of the I/O which occurs on the disks along with the necessary control functions. It is the file routines which open and close file buffers and keep track of where all of the files in the system actually are. For instance, when MTS decides to open a file, it calls a file routine to do it (via the file DSR). The file routines then allocate the appropriate buffers and read some initial part of the file into them. When MTS decides to close the file, the file routines are again called via the file DSR to write out the buffers and deallocate them.

The Loader

The loader (UMLOAD) is as independent of MTS as it can be, though perhaps not as much as it would like to be. The MTS-loader interface is provided by several loader entry points and a loader work area (dsect) provided as a parameter. Within this loader dsect are all of the parameters and control information the loader needs to perform the desired functions. Since the loader has to work in both the normal MTS environment and also in the system IPL environment, it is designed so that its caller can provide the environment. It does that primarily by passing subroutine addresses in the loader dsect for any environment-dependent function, like storage allocation, input, and output. For instance, when MTS calls the loader, it passes the address of an MTS subroutine named LOADIN which the loader can call to obtain a line of input. Another for instance, when PISTLE (the post-IPL loader) calls the loader to load pieces of shared memory into shared segments, it passes the address of its own storage allocation subroutine, which will return space in a shared segment, to UMLOAD.

The loader interface is somewhat complicated, so there is a part of MTS itself which interfaces the loader to the rest of MTS and to user programs. This part is called LLXU (for Link, Load, XCTL, and Unload). Whenever some part of MTS, like the $RUN processor, needs to load a program, it calls the appropriate entry in LLXU to do it. LLXU then calls the loader.

CLSs

The CLSs, Command Language Subsystems, are an important part of MTS since they provide many of the command facilities the user sees. Generally, the CLS facility is provided as a means of subsetting the command language, so that when the user is intending to give file editing commands, for example, his command cannot be interpreted as a debugging command (even if it is an invalid editing command). From the user's point of view, a CLS is invoked by typing the subsystem's name as an MTS command, e.g., $EDIT <fdname>. The CLS is then invoked by MTS and proceeds to act as a command language interpreter until it decides to return to MTS. As far as the user is concerned, he is just using a different part of MTS.

The CLS interface is an attempt to isolate the CLS itself from everything else which makes up its environment. That environment is provided by the CLS transfer vector. When a CLS is invoked, it is passed a pointer to the CLS transfer vector which contains a table of subroutines to be called to provide standard services. In actuality, there are table entries for almost all of the system subroutines offered by MTS. It is intended that all calls to subroutines not contained within the CLS itself go through the transfer vector. Thus, it is possible for a user program to make up a transfer vector and call a CLS, using private subroutines for the various functions. Since a CLS only communicates with the outside world via the transfer vector, there are (almost) never any external symbols within a CLS which any other routines depend on, other than the standard entry point(s). Because of this isolation, CLSs are very flexible.

Another way in which a CLS is isolated by MTS is the way its storage is allocated. Each CLS has a unique storage index number associated with it. This storage index number is also associated with any storage which that CLS might acquire. A CLS gets storage by calling a special GETSPACE subroutine whose address is supplied by MTS in the CLS transfer vector. This special GETSPACE simply calls the real GETSPACE, telling it what storage index number to use. For CLSs invoked by MTS, that storage is allocated in the system work segment (segment 5). A CLS, when it returns to its caller, gives a return code which indicates whether or not the CLS terminated normally. If it did, all of its storage is released (presumably by its caller). If it did not, its caller is expected to preserve both its storage and its register contents for the next invocation. Use of a unique storage index number allows MTS to identify all of the storage belonging to a particular CLS in order to release it.

Initial System Loading

MTS itself resides in the shared segments 1-3 -- usually it is contained in segment 1. When the system is loaded initially (IPL), the supervisor and all of the job programs are loaded all at once, along with a few other things (like the loader!). Everything else which is to reside in shared storage is loaded by

a program officially called PISTLE (post IPL system loader) which is run just like any other program. (Note: PISTLE is referred to by many as the "segment 2 loader", or S2L, for historical reasons.) As PISTLE loads a component into shared storage, it puts the entry points into a table (called LCSYMBOL) which also resides in shared storage. There is a control section in the IPL object module called ENDSEG2 which is physically last in the system object. It is simply a word which contains the address of the first available storage location after everything which has been loaded into shared storage (initially, 8 bytes past itself). This is how PISTLE determines the address at which to begin loading a component. Most of the DSRs are loaded by PISTLE after system IPL (but before the system is made available to users).

Like DSRs, nearly all of the CLSs are loaded into shared segments by PISTLE -- that is, they are not a part of the system IPL object deck. Any external symbols from the CLS are placed into LCSYMBOL by PISTLE. A CLS can be replaced (as can a DSR), while the system is running, by simply running PISTLE to load the new version into shared memory. PISTLE will replace the old entries in LCSYMBOL with the new ones, and the new version will be used instead of the old one.

As an aside, there are one or two CLSs which are not independent of their environment, e.g., PERMIT, and thus cannot be replaced on the fly. This is because there are weak external references in MTS to symbols defined in those CLSs which PISTLE resolves. Once done, the system loader table is purged of any record of that reference, so a new copy of the CLS cannot replace that reference. This violation of the intent of CLSs is definitely considered poor practice.

Organization of MTS

MTS itself is made up of several separate assemblies. These are what are being referred to when one speaks of MTS. Currently there are 13 separate MTS assemblies. The reason they are grouped together is their common dependence on the MTS dsect. For example, one might refer to the COST subroutine as part of MTS, but it does not depend on the MTS dsect, so it is not one of the MTS assemblies. No other assembly refers to the MTS dsect, by definition.

Historically, MTS was made up of just one (small) assembly. As it grew, various pieces were ripped out or added on (e.g., LLXU was ripped out, TIMT was added on). Eventually, it got so big and unwieldy that it had to be split up into smaller pieces. It was then split into the currently existing assemblies. However, there were several subroutines and tables which were common to a great many of these separated components. These were gathered together into a "system common" region which is addressable by two base registers (GR12 and GR11). These two registers normally contain the base addresses for this common region. In order for the separate assemblies to reference symbols within the common region without keeping a horrendous

dsect, and without requiring a large number of external symbols, the common region begins with a transfer vector. There is a dsect which corresponds to this transfer vector (generated by the MTSTV macro) which each assembly uses to define the common symbols. The common region itself is defined in the MTS assembly, which is the most central of the MTS assemblies. The MTS assembly contains the job program entry point and initialization, various built-in tables, the command loop, and sundry subroutines. The other assemblies are:

CMDS - the command processors
LLXU - the UMLOAD interface
DSRI - the DSR interface
DSRS - several special "built-in" DSRs (file routines, HASP, etc.)
FSUB - file related subroutines (GETFD, HOPENIT, GIVEBACK, etc.)
TIMT - the timer interrupt subroutines
GSFS - (Getspace/Freespace) storage management subroutines
RSF - some of the subroutines which interface to the file routines (Really Shared Files)
USUB - miscellaneous user-callable subroutines
GATE - the gatekeeper
INFO - the GUINFO/CUINFO subroutine
PLIM - the page limit/card limit testing subroutines

There are several coding conventions used throughout the MTS assemblies which will be mentioned briefly here. The base register for the MTS dsect is always GR4 (except in the current GSFS, where it is GR9 because the SLT instruction, around which that version of GSFS is based, uses registers 0 through 5). The value of this register is obtained by the HWIMB macro (help, where is my buffer). (Digression: a little thought will reveal this to be a non-trivial problem. If MTS is entered from the outside, who knows where the MTS dsect is, since a pointer to it cannot be put in a shared segment? It turns out that the dsect pointer is put in a shared segment -- segment 0 -- and it is put there by the supervisor, in the job table entry for that task. The job table entry is always pointed to by the location 'FFC' in page 0. Note that for multi-processor operation, this pointer has to be in page 0 -- the PSA). Other coding conventions include the use of general registers 12 and 11 for common region addressability, 10 (and sometimes 5 and 3) for base registers, and GRs 6 - 9 (called SCA, SCB, SCC, and SCD) for scratch registers. GRs 0, 1, and 2 are also often used for scratch. GRS 14 and 15 are usually used for subroutine linkage. Beyond these general rules, register usage is haphazard. Another significant (astonishing, if you prefer) characteristic of MTS is that internal subroutines seldom save and restore registers.

The Protection Mechanism

There is a feature in MTS which attempts to isolate system components from user program damage, usually called "protection", for lack of a better term. What it does is control the entrances

to and exits from a user's program from/to MTS so that the supervisor always knows what mode the job is in. This is used to create a software-implemented equivalent of the hardware supervisor/problem state mechanism. When a job is in user mode (i.e. executing a user's program), the direct invocation of most supervisor services (via SVCs) is invalid and the system work segment (segment 5) is disconnected from the task's address space. This means that in user mode the system work segment cannot be changed (or even referenced) by an errant or malicious program.

The mechanism to switch modes between user and system modes was added to MTS fairly late in its development. The principal mechanism is the gatekeeper (GATE) which provides a "gate" through which user programs must pass on their way to system code in system subroutines. The gate changes the state from user to system by means of an SVC and calls the desired subroutine. That subroutine, when done, then returns to the gatekeeper, which changes state back to user and returns to the original caller. This is a somewhat simplified description, since there are special cases like starting a program, stopping it, subroutines which do not return, and some which take sideways exits.

Exit Routines

Handling communication between various concurrent, asynchronous processes is a difficult problem in any operating system, and MTS is no exception. The approach taken by MTS, as with many others, is to only allow a limited amount of asynchronous communication using a rigid protocol. The primary example of asynchronous communication is the attention interrupt. In this case, the user himself is the sending process and MTS is the receiving process. Other examples are a bit less obvious like program errors, timer interrupts, and special device commands. All of these situations can be thought of as one entity, or process, communicating with another (usually MTS).

All such communication is handled, at some point, by the supervisor. Consequently, UMMPS has adopted a protocol for handling such communication to a task -- that is, the concept of an exit routine. In general, a task tells UMMPS, by way of an SVC, what the address is of a subroutine which is to be invoked at the time of each class of communication. For instance, the timer routines issue an SVC telling UMMPS the location of an exit routine to be taken in the event of a timer interrupt. If a task does not set up an exit routine for a class of messages, it cannot be notified of any such message. Since, from the supervisor's point of view, most of these messages are directed from one part of a single task to another, if an unexpected message occurs, the task is usually unceremoniously stopped.

A digression can be made here about interprocess communication. The way the term has been used above is not exactly the same as the way it is normally used. Usually one thinks of two tasks, in the supervisor sense, sending messages to

one another being intertask communication. In the previous discussion, a task, in effect, has been sending a message to itself -- although it is a different part of itself. One of the functions of MTS is to act as a monitor for user programs. If some exceptional event occurs, like a program interrupt, one can think of this as the program "sending a message", via the supervisor, to the monitor. That is the sense in which we have used interprocess communication. In fact, UMMPS has no general facility for real intertask commmunication, though several have been proposed and implemented at various MTS installations. Significantly, the proposed methods for tasks to receive messages are similar to the current use of exit routines to handle exceptional conditions.

Back to how an exit routine works; when the invoking condition, or message, occurs, UMMPS saves the current status of a few registers and the PSW of the task in the area given by the set-up SVC (the exit area), and then begins the task again at the exit routine. This status is also saved in the local CPU queue which UMMPS keeps for each task. This is really a push-down stack, where the top entry indicates the status of the currently executing "part" (or sub-process) of the task.

The exit routine, when it is invoked, has the entire status of the interrupted process at its disposal (in either its exit area or still in the registers) and it may do what it likes. If the exit routine chooses to restart the interrupted process, it can execute an SVC telling UMMPS to "pop" the CPU Q, thus going back to the interrupted process. If it chooses not to return, as in the case of a program interrupt, it executes an SVC telling UMMPS to "flush" the lower CPU Q entries, leaving the exit routine as the task's sole representative on the CPU Q. Note that an exit routine, though it flushes the CPU Q, can still restart the interrupted process by simply loading the registers with the appropriate values and branching (usually via an SVC TRA, which will load the registers and do a register-less branch all at once).

Exit routines are quite limited in the things they can do, especially if they intend to restart the interrupted process. This is because they share the same address space as the interrupted process. If the exit routine changes some piece of storage which the interrupted process does not expect to be changed, an error will usually occur upon restarting. A typical example is the attention interrupt exit. The exit routine must first check to see if an attention can be taken before doing much of anything. If the interrupted process was doing something critical, like changing something in the MTS dsect, the attention could not be taken without the risk of making that storage inconsistent. In such a case, the exit routine simply sets a bit meaning an attention occurred and then restarts the interrupted process using an SVC POPQ.

How TIMT - The MTS Timer Routine - Works

TIMT is composed of four user callable subroutines and four entry points used by other parts of MTS. Timer interrupts are set up by calls to SETIME and the interrupt exit is taken after the interrupt occurs in accordance with the exit block set up by a call to TIMNTRP. Timer interrupts may be set for task time or real time. A task time interrupt may be local to a single CLS, usually the CLS which called SETIME, or it may be global to all CLSs if set up from MTS via SPSETIME.

The method used is basically as follows:

A TXA is created and an interrupt is set up by a call to SETIME. The TXA is put on TIMLST. An interrupt exit is enabled by calling TIMNTRP which creates an EXB for that interrupt. When the interrupt occurs (expires), the TXA is removed from TIMLST and put on the pending list of the appropriate CLS unless the interrupt occured while executing within MTS. In this case the TXA is left on TIMLST but marked expired, TTBIT in SWS6 is set, and a 1/4 second task time interrupt is set up, presumably so that the interrupt will occur again outside of MTS. Once the TXA is made pending, a matching EXB is searched for in that CLS. If found, the exit is taken. If not found, the TXA is left pending.

Whenever MTS changes the executing CLS, a check is made (via SWSTATE) for any pending TXAs which have enabled EXBs. If there are any, then the exit is taken. Before the CLS transfer is made, TIMLST is checked for task time interrupts in effect. Task time interrupts may be local to particular CLSs. Those which are local to the CLS being switched from are cancelled and those local to the CLS being switched to are reset. If a timer interrupt is cancelled, its TXA remains on TIMLST, because presumably that CLS will eventually be reinvoked and the interrupt reenabled.

The data structure used is as follows:

TXA - Timer Exit Area

This block contains
1) the exit information for the SVC TIMER
2) an ID
3) address of the exit region (76 bytes)
4) the CLS number of the exit routine
5) the CLS number of which CLS the task time interrupt applies to (-1 indicates all CLSs)
6) status bits (expired, cancelled, or pending)
7) amount of time left

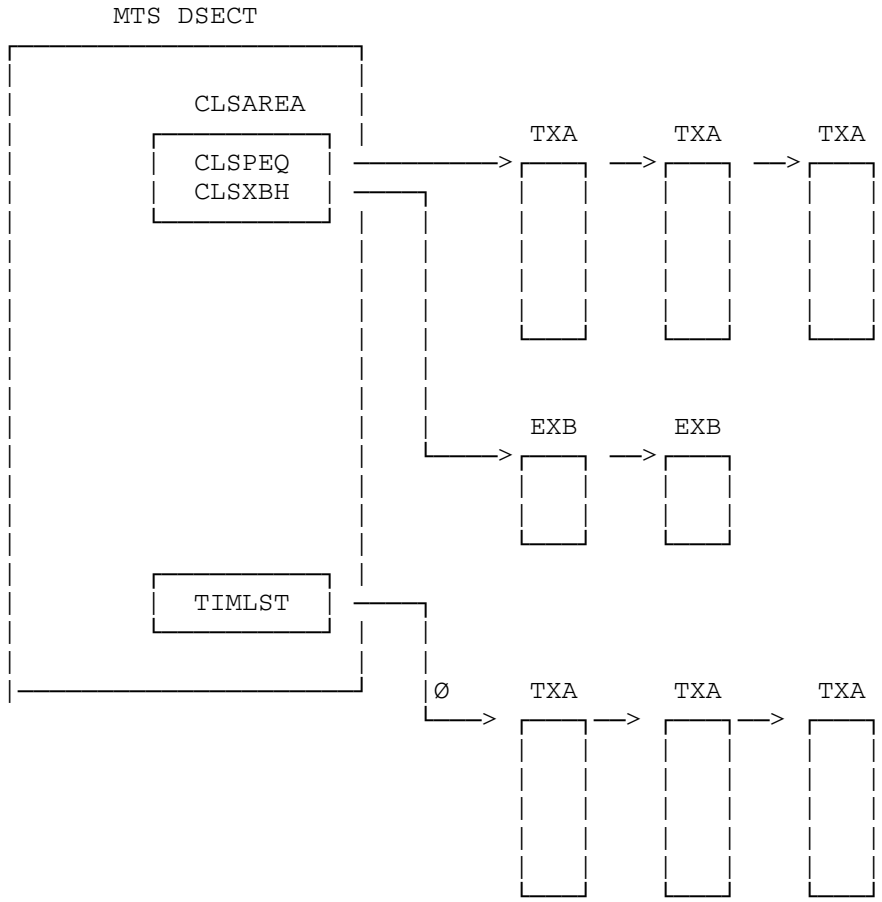
TXAs corresponding to unexpired timer interrupts are put on a single chain anchored at TIMLST in the MTS dsect. All TXAs on this list represent timer interrupts which are in effect or are local to a dormant CLS and have not yet expired.

All TXAs which represent timer interrupts which have expired - i.e., the time is up - and which have not been taken are put on a "pending" list anchored by CLSPEQ in the CLS area. Thus, there is a pending list for each CLS.

EXB - Exit Block

This block contains
1) the address of the exit subroutine
2) the address of the exit region (same as in TXA)
3) a save area address
4) status bits (enabled)

An EXB is created and placed on a list anchored by CLSEXBH in the CLS area at a call to TIMNTRP. The EXB represents an enabled interrupt corresponding to the TXA with the same region address (and the same CLS).



Internal subroutines

LOOKNSEE (also FINDIT)

If TTBIT (in SWS6) is on, look through TIMLST for expired TXAs. For each one, cancel the 1/4 second interrupt, mark it pending, and put it on the PEQ of the appropriate CLS. (Note: FINDIT does not test TTBIT but does the above unconditionally.)

Look through the PEQ of the current CLS. For each one, look for an EXB with matching exit region (TXAREG=EXBREG). If there is one and it is enabled, set up a pointer to the exit region, return 0(R14) to take the exit. If none was found, return 4(R14).

CANIT

If the interrupt associated with the TXA is not already cancelled, then use SVC TIMECNCL to cancel the interrupt, return the time remaining.

SWREGS

Put contents of the exit region (status of the interrupted CLS) into the CLS area of the old CLS. Put contents of the new CLS area PSW and GRS into the exit

region if the old and new CLSs are different, then call SWSYS.

SWSYS

Look through TIMLST for TXAs with task time interrupts set. If it belongs to the old CLS, cancel the interrupt. If it belongs to the new CLS, reset the interrupt. Put the new CLS number is CLSCURR.

GOSUB

Put the interrupt ID in the exit region. Get a save area (which was attached to EXB). Remove the TXA from the pending list and release it. Save the current CLS state in the save area. Mark the exit block disabled. Set up R1, R13, R14, and R15 to take the exit.

SWSTATE (an external symbol for MTS)

If the new and old CLSs are different, then call SWSYS. Call LOOKNSEE to check for an enabled interrupt. If there is one, call GOSUB to set up for the exit. Return with RC 4.

TIMERDI (an entry point from DSRI)

This is called if DSRI finds TTBIT set. Look for a pending interrupt (calls FINDIT). If found, set up DSRI's registers in the save area and take the exit via TIMNTGO.

EXRTN - the SVC interrupt exit

Mark the TXA expired. If STATTN or NOATTN is set or the interrupt occurred while inside MTS, turn on TTBIT and reset the interrupt to occur 1/4 second later otherwise, mark the TXA pending. Call FINDIT to put the pending TXA on the right list and determine if the exit is enabled. If the exit is enabled, then take it via TIMNTGO.

Classification Codes: 161.E/0 and 1B2.2/0
Date: 5/18/77
Doct=9 Vers=1

The Protection of Information in a General Purpose Time-Sharing Environment.

Gary C. Pirkola and John W. Sanguinetti
The University of Michigan Computing Center
Ann Arbor, Michigan 48105

Introduction

The methods used to control access to information in the Michigan Terminal System (MTS) are described. Units of information to which access is controlled include both segments in the virtual address spaces of the various processes in the system and files in the on-line file system. The entities whose access to information is controlled are programs, at both the system and user levels, invoked by means of the command language.

In general, access to information is controlled in MTS in the following ways. First, the use of virtual storage in MTS is such that individual processes cannot refer to segments in the private virtual storage of any other process. Second, programs executing in a process in MTS will switch, under system control, between various domains (primarily between the user and system domains) and the access to segments in the process's private virtual storage will change depending on the currently active domain. Third, modes of access to individual files in MTS may be restricted not only to specific users and groups of users, but also to specific (user-written) programs. As a result, owners of files may completely determine under program control how their files may be accessed. Finally, users in MTS may, at the command language level, switch between various subsystems, and the access to a particular file may change depending on the currently active subsystem.

MTS

The Michigan Terminal System (MTS) is a large, general-purpose, time-sharing operating system which has been in continual development since 1966 by the University of Michigan's Computing Center staff, initially for use on the IBM 360/67. It is currently in use at the University of Michigan and five other universities in the United States, Canada, and England[1], running

on IBM 360/67, 370/168, and Amdahl 470V/6 computers. At the University of Michigan, MTS runs on a four megabyte Amdahl 470V/6 and typically supports 160 simultaneous users (both terminal and batch) during a normal afternoon. MTS services the educational and research computing needs of the University. In recent years, all of the universities using MTS have contributed in varying degrees to its development. In particular, the storage protection mechanism described in this paper was first implemented at Wayne State University.

MTS has been described in [1]. Only those aspects of the system's structure which are relevant to the protection of information will be described here. MTS is a virtual storage system in which each user has a process which has its own unique virtual address space. In the remainder of this paper, we will generally not distinguish between a user and the process which acts for the user. The principal component of the user's process is the main command interpreter. Each user process has a dedicated device -- usually a terminal -- for input of commands or data and output of command results and other messages. The command interpreter generally reads a command from the input device and performs the indicated operation. Some typical commands are SIGNON, SIGNOFF, RUN, EDIT, PERMIT, COPY, CREATE, and DESTROY. Each of the preceding commands, except for SIGNON and SIGNOFF, performs some operation on one or more files -- for example, the RUN command loads the object program found in the specified file and initiates execution.

The main command interpreter may invoke one of several command language subsystems (CLSs) in response to a user command. Each subsystem interprets its own set of commands. In this way, the main command language is kept to a reasonable size. For example, the context editor is invoked by the EDIT command, interprets editor commands until a STOP command is given, and returns to the main command interpreter. Although most CLSs appear to be part of the system, each is a separate set of routines which is invoked by the main command interpreter. In fact, a program which a user runs by means of the RUN command (hereafter called a user program) is just another subsystem to be invoked, as far as the main command interpreter is concerned.

One of the important properties of a subsystem is that it is independent of the other subsystems. A subsystem may be suspended, some other subsystem may be invoked, and then the original subsystem may be continued at a later time. Thus, a user could run a program, suspend it to edit an input file, and then continue running the program from the point of suspension. In fact, the main command interpreter can be considered a

---

[1] The University of Alberta, The University of British Columbia, The University of Newcastle-Upon-Tyne, Rensselaer Polytechnic Institute, and Wayne State University.

command language subsystem.

The main command interpreter not only interprets commands, it also provides system services by means of subroutines which user programs and CLSs can call. For instance, to dynamically acquire virtual storage, a user program calls the system-supplied subroutine called GETSPACE. All system services are provided in the form of subroutines -- user programs do not, in general, execute supervisor call instructions. System-supplied subroutines such as GETSPACE use supervisor calls to request services of the supervisor, which is the only component of MTS which operates in the "supervisor state" provided by the hardware of the 360/370-style machines [2]. Each user process operates exclusively in problem state.

Virtual Storage

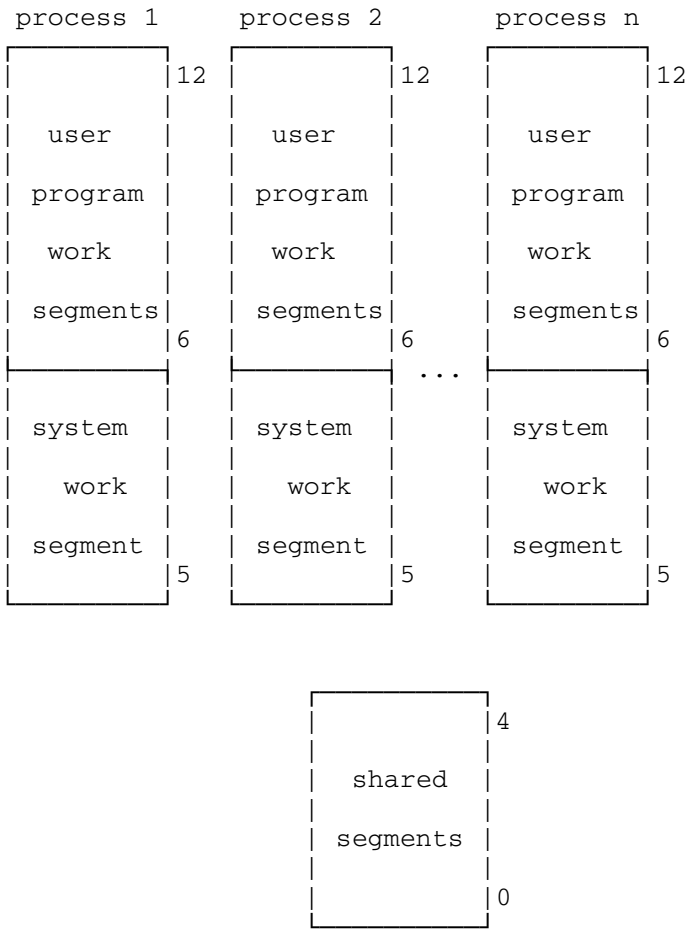
Use of Virtual Storage in MTS

MTS uses virtual storage in the following way (see figure 1). Each process in the system has its own virtual address space. This space currently consists of 13 segments of 256 pages each (segments 0-12) amounting to about 13 million bytes for each user. The first five segments (0-4) are shared among all processes and have the same addresses in all processes. This is the "address agreement" approach to shared memory. All of the shared system components reside in these shared segments -- the main command interpreter, most CLSs, the supervisor, the file system routines, and others. The remaining address space of each process (segments 5-12) is private to that process -- it does not intersect with the address space of any other process. Information in the private segments in any process cannot be accessed in any way by another process since that information does not appear in the address space of any other process [3]. This results from the use of the address translation mechanism and is the principal means of information protection between processes in MTS.

The private segments of each process are further divided into a segment for system work space (segment 5) and several user program work space segments (6-12). The system work segment contains all of the control information at the process level necessary to provide services to the user and to account for them.[2] No user-callable programs are ever kept in the system work segment. Among the control structures kept in the system work segment are many of those used by the file system and the CLSs. The user program work segments contain user programs

---

[2] System-wide control data structures, of necessity, are kept in the shared segments, but nearly all control structures which are specific to a single process are kept by the process itself in its system work segment.



Use of Virtual Storage in MTS

Figure 1

which are run and any work areas dynamically allocated by these user programs.

Protection of Information in Virtual Storage

As we have seen, the address space of each process can be divided into three parts -- shared segments, the system's work segment, and the user program's work segments. A security policy can be defined by putting different access restrictions

on each part. Because the IBM 360/370 architecture does not provide a hardware-implemented execute-only storage access type, every process must have read access to the shared segments in order to call system subroutines which reside there. Except for the relatively short time involved in executing a few system routines, a process should not have write access to the shared segments. System components must have both read and write access to the system's work segment, but user programs must have no access to it.[3] Both system components and user programs must have read and write access to the user program work segments.

In this discussion, the objects to which access must be controlled are segments. The subjects which access the objects are routines within a process -- that is, system subroutines, the command interpreter, the CLSs, the file routines, and user programs. Note that system-provided utility programs, like compilers, the linkage editor, etc., are all considered user programs, since they are invoked by means of the RUN command.

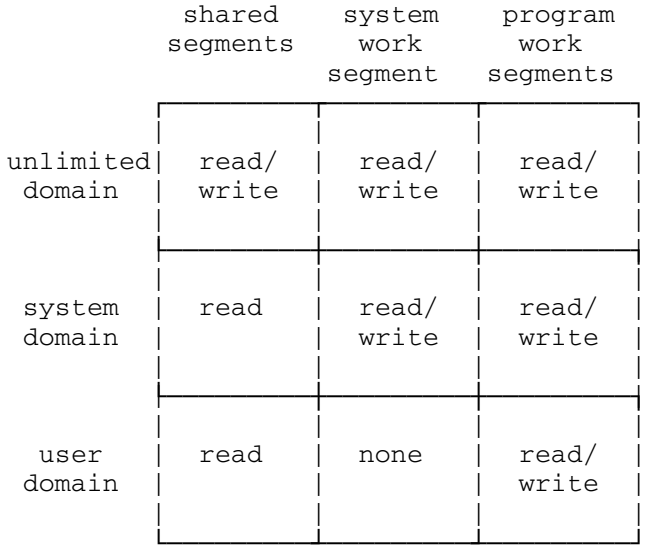
The desired access restrictions mentioned above define three separate "protection domains" within a process.[4] (See Linden [4] for a discussion of protection domains.) (1) Routines which require read/write access to all segments including the shared segments execute in the "unlimited" domain. For example, system routines which maintain information in tables located in shared segments must operate partly in the unlimited domain. (2) Routines which require read/write access to the system work segment, but not the shared segments, execute in the "system" domain. For example, the command interpreter operates in the system domain. (3) User programs require only read/write access to user work segments and read access to the shared segments (so that system subroutines may be called). This access defines the "user" domain. The security policy is implemented by controlling the transitions between protection domains. Figure 2 provides a summary of the access rights of domains to segments.

Control of Access to Virtual Storage

---

[3] In MTS, there is no "sensitive" information kept in the system work segment which must be hidden from the user. Therefore, user programs could be allowed read-only access to the system's work segment.

[4] The three protection domains described here are those which are relevent to a user's process. There are two other domains in MTS: absolute and supervisor. There are absolute processes which do not use virtual storage -- such as the paging drum processor. These processes execute in problem state with access to all of real storage. The supervisor itself executes in supervisor state with access to all virtual storage and all real storage.



Access to Segments by Domains

Figure 2

To provide the necessary access control, two separate mechanisms are used. To control access to shared segments, the storage key protection feature of the IBM 360/370 architecture is used. Pages in memory are given "store protected" storage keys, and the access to those pages is determined by comparing the key in the processor with the storage keys -- read/write access is allowed if the processor key matches the storage key; read access is allowed, write access is denied if the keys do not match. If the processor key is zero, then all pages are read/write accessible.[5] In MTS, pages in the shared segments are given storage keys of zero and pages in private segments are given storage keys of one. A user process always is given a processor key of one by the supervisor unless it is in unlimited domain. Thus, all subjects have read-only access to the shared segments.

Note that the use of storage keys to protect segments is

[5] See [2] for more details on how storage key protection works.

not especially appropriate, since keys are associated with pages on a 360/370, and not with segments. However, they can be used for this case of access control because the storage keys in the shared pages need never be changed once they are set initially, and because only very limited write access is allowed to the shared segments.

The same mechanism will not work to control access to the system's work segment. If the pages in the system work segment were given storage keys distinct from the keys given the pages in the user work segments, any subject which had read/write access to the system segment would not have write access to the user work segments (unless its processor key was zero, giving it write access to the shared segments as well).

Instead, the hardware-provided address translation mechanism (using the segment table) is used to provide different access rights to the various private segments. While system routines are executing, the segment table contains an entry for each of the shared, system, and user segments. While user routines are executing, the segment table contains entries for the user segments (and, of course, the shared segments) but not the system segment. This is the second use of the address translation mechanism for storage protection. A user process cannot ever access another process's private segments, and it can only access its own system work segment when it is in the system domain. The software must carefully control transitions between these two protection domains in order to insure that the segment table contains only the appropriate entries.

System-Controlled Domain Switching

A transition between system domain and unlimited domain is allowed, and is accomplished by a supervisor call. This is necessary, of course, because the supervisor must verify that the domain switch is, in fact, being requested from the system domain, and must change the process's processor key to zero if it is. Due to the fact that routines which execute in the system domain are "trusted", transition from system to unlimited domain is always allowed. It is presumed that the routine will switch its domain back to system domain as soon as it no longer needs write access to a shared segment.

Since segment tables are kept in the supervisor, transitions between system and user domains must be accomplished by means of supervisor calls. Two supervisor calls are defined for this purpose -- one to switch in each direction. Since domain switches must be controlled, these supervisor calls (particularly "enter system domain") must be allowed to be issued only from well-known system routines. In fact, since there is no need for a user program to issue any supervisor calls, all supervisor calls except "enter system domain" can be disallowed when in the user domain. All of the locations from which the "enter system domain" supervisor call can be validly

issued are known to the supervisor and are called "gates." All such gates are in the shared segments so their addresses remain fixed. Consequently, the supervisor always knows in which domain a process is executing and can assure that the system domain is only entered at pre-determined locations.

Implementation Considerations

In general, there is a gate for each non-trivial system subroutine.[6] The gate provides the domain change, and the subroutine can then access the system work segment and issue supervisor calls. Each system subroutine is required to return to its gate which executes the "enter user domain" supervisor call before returning to its caller.

There are several things which must be considered when allowing a subroutine in the system domain to be invoked by a user domain routine. The "attenuation of privilege" problem must be solved -- that is, the subroutine must not be able to be tricked into doing anything which its caller does not have suitable access to do. There are two cases where this can happen in MTS -- the caller could provide parameters to which he did not have access, or provide a save area whose address was in the system work segment. All system subroutines must check the addresses of the parameters and save areas they are passed to ensure that they are not in the system work segment. Note that this checking must be done after the domain switch but before the item being checked is used.

A less frequent situation which must be allowed for is the case of an "outward call" -- a routine in the system domain calling a subroutine in the user domain. In this case, any parameters or save areas provided to the user domain subroutine must be in one of the user segments. This usually involves making copies of parameters. A gate must also be provided through which the called subroutine may return. At the time of return, any returned parameters must be checked and any registers which are presumed to be restored must also be checked.

A routine executing in the system domain must not be allowed to lose control to a routine in the user domain due to an asynchronous or unexpected event, like a timer or attention interrupt. In MTS, this problem is handled by noting the occurrence of such an event if it occurs when the process is in the system domain, and exiting to the user domain trap routine (if there is one) at the time the domain switch from system to user domain is made. Those events which result in suspension of

---

[6] System-provided subroutines which do not access the system work segment can execute in the user domain, and thus do not require gates.

a system domain routine, like program interrupts or other errors, cause the main command interpreter to be invoked. If the user attempts to restart the suspended program, the domain is automatically switched to the user domain. This is required because the user, by issuing some intermediate sequence of commands, could cause the environment of the suspended system routine to change. Thus, most suspensions which do occur in system domain routines are errors, for which restarting is not allowed.

To summarize the storage protection in MTS, access to virtual storage is controlled on a segment basis, with three distinct protection domains: unlimited, system, and user. Between processes, non-intersecting virtual address spaces insure the isolation of private segments. Within a process, the differing access rights of the three domains protect the various segments. Transition may be made between the unlimited and system domains, and between the system and user domains. Because the unlimited and system domains are only available to system routines, which are presumed to be trustworthy, transitions between these two domains are not carefully controlled. On the other hand, transitions between system and user domains are very carefully controlled so that a routine in user domain has no direct access to the system work segment.

Limitations

Comparing this protection mechanism to other protection models, one finds that it is rather limited. If we ignore the distinction between the unlimited and system domains, this is a simple privileged state mechanism (or, if one prefers, a two-level ring structure [5]). In fact, the term "gate" comes from MULTICS. The segment table entries can be thought of as "capabilities" for the associated segments.[7] These capabilities are rather primitive, however, in that they allow only one type of access -- read/write.

The limitations of the protection mechanism are found primarily in its scope. The protection domains, as described by Linden [4], are not small, in the sense that system domain routines typically have more access than they need to perform their task. For instance, some system routines do not require any access to user segments, yet they have read/write access. One reason for this limitation is the lack of different access rights to user and system segments. By having an entry in the segment table, a segment is read and write accessible. Many system routines do not require write access to the system segment, for example, but there is no convenient way to provide them with read-only access. The other reason small domains are

---

[7] See [6] for a general description of capabilities and how they relate to the protection of information.

impractical in MTS is the number of available segments. Using 24-bit addressing, there are only 16 segments available in the virtual address space. This makes it impractical to load many different routines into many different segments which are members of different protection domains.

Some of these limitations could be removed by a simple change to the hardware. If the storage-protect key were put into the segment table entry rather than in each page, and if a hierarchy of access were defined for it (i.e. if a priority ordering were established) a 16-level ring structure could potentially be implemented. However, ring crossings would still be a high overhead operation, since they would require supervisor intervention.

Files

The File System in MTS

The file system is a collection of routines which are called either during interpretation of commands or while providing some service for a program, like reading or writing a file. Some of its primary responsibilities include (starting at the lowest level) initiating physical disk I/O and subsequent error recovery, allocating and deallocating disk storage space, and maintaining and interrogating the file system catalog. Moreover, it is a "buffer-oriented" system as opposed to a "virtual storage-oriented" system. That is to say, files in MTS are not mapped into segments of the process's virtual storage. Instead, "page-sized" buffers (as well as control blocks) are allocated in the process's system work segment for each active file within a process, and one of the main functions of the file system is to transfer blocks of file information, when necessary, between the page-sized physical records on the secondary storage device and the process's buffers. Quite obviously, since these page-sized blocks of information do not correspond directly to logical records read and written by programs, the file system is also responsible for managing and transferring logical records between the page-sized buffers and the calling program's input and output areas. Since each process has its own buffers and control blocks in its own private system work segment (and thus its own copy of a file during active use), the concurrent usage of files in MTS must be controlled by a set of routines (similar to a monitor as defined by Hoare [7]) which interrogates a system-wide table (in a shared segment) of currently active files to determine if concurrent usage of a particular file is allowable at any given point.[8]

---

[8] See [8] for more details on the structure of the file system in MTS as well as details on how concurrent usage of files is accomplished in MTS.

Protection of Information in Files

For the protection of information in files, most state-of-the-art time-sharing systems use access control lists associated with each file, and MTS is similar in this respect. Briefly, when a file is initially created in MTS, the owner of the file has unlimited access to perform any of a number of operations on the file. These operations might include, for example, reading the file, writing (with a distinction made between expanding and changing), emptying (discarding the contents), renumbering, truncating (deallocating unused disk space), renaming, destroying (deallocating all disk space), and permitting (giving access to others). Initially everyone else has, by default, no access to the file. Subsequently, the owner of a file may give specific users of the system permission to perform any combination of the above-mentioned operations on the file.

Each user of MTS is identified by a unique identification code, (hereafter referred to as a userid), and as is the case with most systems, a password mechanism is used to authenticate a particular user at SIGNON time. Once signed on, that user has access to all of his own files as well as to other users' files to which his userid has been given explicit access. In addition, users at the University of Michigan are grouped into projects -- for example, all students in a particular class or all staff on a particular research project -- and the owners may permit their files to be accessed in a specified fashion by all users within a particular project (i.e., all users with a specific project number).

Control of Access to Files

Two things should be mentioned about the file-sharing facility. First, every user of the system has at least one set of access rights associated with any given file (in most cases it is, by default, no access to the file). Second, if a user has more than one set of access rights to the file because both his userid and his project have been given permission to access the file, then a priority scheme is used to determine the access rights. The set of access rights associated with the userid is used if given -- if not, then the set of access rights associated with the project is used if given. Otherwise, the access rights associated with everyone else is used (by default no access, but changeable by the owner). Thus, in the usual case, the owner of a file specifies that certain specific users have certain specific kinds of access to a file, while everyone else has no access to it. However, the owner may also specify that everyone has some specified access, except for certain users or groups of users who have some other access, e.g. none.

Although the mechanisms described so far are seemingly quite general and flexible, they have at least one major shortcoming. Specifically, if a user has been given "read" access to a file, he might read that file in a number of

different ways, each obtaining quite different amounts of information. For example, if the file contains a program (i.e., an object module) to be executed, the user might simply request that MTS read the file for the purpose of loading and executing the program. Alternatively, the user might request that a system utility be invoked which will read the file and tell him about the internal structure of the object module. Or he might read the file directly himself and make a copy for his own use. Each usage requires only that the user has been given read access to the file, but each obtains successively more information about the contents of the file. Likewise, permission to allow a user to write a file can have varying consequences depending on how the user chooses to exercise his "write" privileges. In the least destructive case he may only be appending information to a file or to each record in the file; in the most destructive case he may be completely changing or deleting every record in the file. It thus seems that a more discriminating access control mechanism is needed.

Program-Controlled Access to Files

Saltzer and Schroeder [6] refer to the need for "protected subsystems", i.e., user-provided programs which control access to files. As they indicate, only a few of the most advanced system designs have tried to provide for user-specified protected subsystems and these have, in general, been with special hardware (or extensive software) designed to assist in the implementation. Honeywell's MULTICS [5] uses the hardware ring structure to provide protected subsystems to control access to files, whereas more current (experimental) systems such as the CAP system of Cambridge University [9] and the HYDRA system of Carnegie-Mellon University [10] use hardware-or-software-implemented capabilities, respectively, to provide such protected subsystems.

MTS addresses this need by providing a software-implemented access control list mechanism for allowing user-written protected subsystems. That is to say, by extending the permission mechanism to allow data files to be accessed in specified ways by programs as well as by users and projects, MTS allows a specific user-written program -- i.e., a file which contains an object module to be executed -- to be designated the only thing allowed particular types of access to specific files (which might contain data, for example). Thus the designer of a program can, if he desires, completely control the read and write access to his data files. Specifically, this mechanism is provided by allowing the owner of a file (containing an object module to be executed) to associate a unique identification (an eight-character string called a "program key") with that file. Then data files, for example, may be specified as accessible in a particular way (read, write change, destroy, etc.) only by a particular program key -- i.e., only if the executing program which is accessing the data file was loaded from a file with the appropriate program key. Program keys are prefixed internally

by a userid, and thus they are unique among users. In addition, they are in general non-transferable. In this way users can be restricted from having any direct access to data files; only an executing program may then access the data files, presumably in the fashion prescribed by the designer of the program.

Execute-Only Files

Obviously, the users of the program (with program-key access to the data files) must also have been given some sort of access to the file containing the program in order to run it. This could simply be read access as described before, but a software implemented "execute-only" access is provided as a specific example of the general concept of program-key access to files. (As mentioned previously, the machines on which MTS runs have no hardware implemented execute-only storage access type; thus the motivation to implement something in the software.) Execute-only access is accomplished in MTS by associating a unique program key with each of the system programs invoked when a user enters a command, in particular the system program which loads and executes user programs. Thus, permission to read files containing programs to be executed (i.e., object modules) may be given only to the system load-and-execute program (i.e., the RUN command interpreter). In a similar manner, files may be designated as "edit-only." For example, only the system context editor program might be given permission to write a file containing the source for some program, thereby guaranteeing that the source file will not be accidentally damaged by a careless user or even by the owner.

Priority Resolution of Multiple Access Rights

Quite obviously, the addition of program keys to the list of those who may access a file complicates the priority resolution when the userid, project, and program-key combination have more than one set of access rights to a file. In MTS, the priority of program-key access to files is lower than userid or project access, but higher than the global access associated with everyone else. Thus, if both the userid and/or the project which initiated execution of the program, as well as the program key of the program accessing the file, have been given explicit access to the data file, the access rights associated with the userid (or then the project) will be used to determine the type of access allowed. Only if the userid and project which initiated execution of the program have no explicit access to the data file is the access associated with the program key used. Of course, if the program key of the currently executing program has also not been given explicit access to the data file, then the global access associated with everyone else is used. It should be noted that all files (in particular those which contain object modules) are given a default program key when they are created. Thereafter, the owner of the file may change the program key to any "legal" value he desires (i.e., in

general one prefixed with his userid).[9]

Applications

One of the most obvious applications where it is desirable to have a particular program in complete control of access to data files is that of a data base management system. In such a situation in MTS, the users could be given execute-only access to the data base management program itself, and that program could be given read (and possibly write) access to the data base files. At this point it becomes clear (or at least it became clear during the implementation), that one would like to be even more explicit in specifying what users, projects, and programs may access a file. In particular, one would like to be able to specify that only particular users and/or projects have specific types of access to data files, but only if they access the files by running specific programs. In fact, the facilities provided in MTS are general enough, for example, to permit only a specific project (e.g., a student class) to have read access to the data base files for inquiry purposes, but only via the data base management program; and at the same time to permit other specific users (e.g., the instructors) to have both read and write access to the data base files for updating purposes, but still only via the data base management program.[10]

Implementation Considerations

Since in MTS the protection of information in files is completely implemented in the software, it might be useful to discuss some of the more interesting implementation considerations. In particular, MTS goes to considerable lengths to guarantee that a program with an explicitly assigned program key (or an execute-only program) is run in a manner consistent

---

[9] As an indication of the amount of sharing of files which actually goes on in MTS at the University of Michigan, the following numbers may be of interest. One should keep in mind that the facility for sharing files among users and projects has been available for close to 4 years, whereas the facility for giving programs access to files has been available for only about one year. Currently in MTS there are over 86,000 private files on-line. Of these, over 40,000 are shared in some way; the others are accessible only to their owners. Over 22,000 are shared by everyone, and over 27,000 are shared by specific users, projects, or programs. Obviously, some of the 40,000 are shared both globally and specifically. Of the 27,000 files accessible specifically to users, projects and programs, over 1,000 have been given program-key access.

[10] See [11] for details on exactly how this is accomplished in MTS, and how the now even more complicated priority of multiple access rights is resolved.

with the desires of a security-conscious program designer. In general, this means that if MTS detects such a program being run as anything other than a single, self-contained "load module", MTS will insure that program key access to the data files is denied.[11]

Concerning other areas of implementation strategy, Saltzer and Schroeder [6] refer to the complications involved when one considers the "dynamics of use" of protection mechanisms. In particular, they refer to how one establishes and changes the specifications of who may access what files. In MTS, the owner of a file always has authority to change the access control list. That is to say, there is no way that the access to a file can be forever denied to everyone. In addition, anyone to whom the owner has given "permit" access may also alter the access control list of the file. Obviously, the owner must be careful in giving away the right to change the access control list. Since changing the access control lists dynamically may imply potential problems with access rights to currently active files no longer being valid, MTS determines the access that a userid, project, and program key combination has to a file the first time the file is referenced (opened), and retains that information in a control block in the system work segment. MTS also maintains global information (in a shared segment) to indicate how and by whom each active file in the system is currently being accessed. Thus, when someone attempts to change the access control list, MTS is able to determine whether the file is currently active (and as a consequence whether there are access rights outstanding in a control block in storage) and will not allow the access control list to be changed until all activity associated with the file has ceased.

User-Controlled Domain Switching

Linden [4] mentions another area of interest when implementing protection mechanisms, that of providing a mechanism for changing from one protection domain to another with potentially different access rights to files as control passes from one protected subsystem to another. As mentioned previously, programs executing in a process will regularly switch (under system control) between three protection domains, and the access to the segments in the process's virtual storage will change accordingly. Similarly, a user at the command language level may switch between a number of different command language subsystems during a typical terminal session, and the access to files may change accordingly. The most commonly used subsystem is the main command interpreter, but other subsystems include, for example, the editor subsystem, the program debugging subsystem, and the user program execution subsystem.

---

[11] See [11] for details on how this is accomplished.

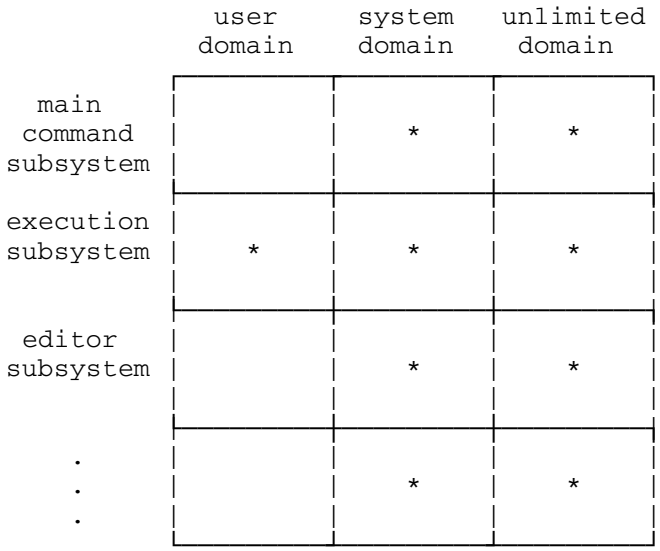
It should be noted that before program-key access to files was implemented, the access which a userid and project had to a particular file was invariant during the time the file was active (open). This is no longer true when one considers the access which a userid, project, and program key combination has to a file. The access to the same file may potentially be different depending on which subsystem is currently accessing the file. For example, if the context editor has been given read and write-change access to a file, and also a particular program (via its associated program key) has been given write-expand access to the file, then when the user is editing the file he may read and change lines in the file (but not add lines). If he switches to the program execution subsystem to run the above-mentioned program, he may add lines to the file (but not read or change them). Needless to say, MTS must be aware when control switches from one subsystem to another and reevaluate the access rights, if necessary.

Actually, as indicated in figure 3, the various protection domains in MTS intersect. For example, a program running in the execution subsystem may switch between the user and system domains and as a result the access to segments will change accordingly. However, if the program remains in the execution subsystem, the access to any currently active files will be invariant. On the other hand, a user switching between the main command subsystem and editor subsystem may have different access to a file currently in use, but if both subsystems remain in the system domain the access to segments will be invariant. Finally it should be noted that the execution subsystem is the only subsystem which runs in the user domain.

One important facility which the program key mechanism is not able to provide and which could not be easily provided with software alone, is the ability to have multiple protection domains (with different program-key access to files) within an executing program -- for example, upon entry to a user callable subroutine. It seems clear that such a facility requires additional hardware and/or extensive software assistance, probably in the form of the above-mentioned capabilities to allow a feasible solution.

Summary

By controlling access to information in virtual storage, MTS provides an environment in which processes are protected from one another, and the operating system is protected from individual processes. The system control data structures used by MTS are either in the shared segments or in the system work segment of each process, and are thus protected from errant or malicious user programs. In particular, the data necessary for the control of access to files is out of the reach of user programs. In addition, by allowing users a flexible mechanism for controlling how their on-line files may be accessed and by what users, projects, and programs, MTS provides a general



Intersection of protection domains in MTS.

Figure 3

facility for the sharing of information in files in a (program controlled) manner specified completely by the owner.

While some of these mechanisms are not as general as others which have been proposed -- particularly capability-based mechanisms -- they do provide a flexible, secure environment while at the same time maintaining system integrity. More generality in the mechanism would require either modifications to the 360/370 architecture or a great deal more overhead in the software.

References

[1] D. W. Boettner and M. T. Alexander. "The Michigan Terminal System." pp. 912-918, Proceedings of the IEEE, Special Issue on Interactive Computer Systems, Vol. 63, No. 6, (June 1975).

[2] IBM System/370 Principles of Operation. IBM Publication GA22-7000-4.

[3] B. Arden, B. Galler, T. C. O'Brien, and F. Westervelt. "Program and Addressing Structure in a Time-Sharing Environment." pp. 1-16. Journal of the ACM, Vol. 13, No. 1, (Jan. 1966).

[4] T. A. Linden. "Operating System Structures to Support Security and Reliable Software." pp. 409-445. ACM Computing Surveys, Vol. 8, No. 4, (Dec. 1976).

[5] J. H. Saltzer. "Protection and the Control of Information Sharing in MULTICS." pp. 388-402, Communications of the ACM, Vol. 17, No. 7, (July 1974).

[6] J. H. Saltzer and M. D. Schroeder. "The Protection of Information in Computer Systems." pp. 1278-1308, Proceedings of the IEEE, Vol. 63 No. 9, (Sept. 1975).

[7] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept." pp. 549-557, Communications of the ACM, Vol. 17, No. 10, (Oct. 1974).

[8] G. C. Pirkola. "A File System for a General Purpose Time-Sharing Environment." pp. 918-924, Proceedings of the IEEE, Special Issue on Interactive Computer Systems, Vol. 63, No. 6, (June 1975).

[9] R. M. Needham and R. D. H. Walker. "Protection and Process Management in the CAP Computer." pp. 155-160, Proceedings of the IRIA International Workshop on Protection in Operating Systems. Institut de Recherche d'Informatique et d'Automatique, France, 1974.

[10] E. Cohen and D. Jefferson. "Protection in the HYDRA Operating System." pp. 141-160, Proceedings of the Fifth ACM Symposium on Operating Systems Principles, ACM Operating System Review, Vol. 9, No. 5, (Nov. 1975).

[11] MTS Manual, Vol. 1: The Michigan Terminal System, "Files and Devices, Appendix I," pp. 153-160. The University of Michigan Computing Center, Ann Arbor, Michigan, April 1976.

Classification: 162.12(GUINFO)/6
Date: April 1, 1978
Doct=29 Vers=2

GUINFO Items Designed Mostly for Systems Use

(or whose use is so obscure that we don't have the nerve to put the description into Vol. 3)

Index	Name	Type	Description
13*	LFRBIT	Bit	1 => $SET LFR=OFF (default is OFF). For those who don't know and can't guess LFR stands for library file release.
14	ACCTNO	Fullword	The user's account number.
50	IDRNBR	Fullword	The user's IDR (inter-departmental requisition) number.
52	UNITCODE	Fullword	The user's unit code.
53*	UNLODOFF	Bit	1 => $SET UNLOAD=OFF (default is ON).
132	LOADT	Fullword	The address of the table used by LLXU to associate FDnames with link levels. This table lives in the system segment.
142	PDNTBL	Fullword	The address of the pseudo-device name table (somewhere in the system segment).
235	NOFILERE	Bit	1 => $SET FILEREF=OFF (default is ON). This option may only be set by the ccid MTS.
243	UNPROT	Bit	1 => user can set PROT=OFF (This bit is the logical OR of items 246 and 250)
244	PRIVPROG	Bit	1 => a priv. program is loaded
245	PROTOFF	Bit	1 => user has $SET PROT=OFF
246	ACCPRIV	Bit	1 => THE Super-Priv bit (ACCDCT+1, bit 0), also known as the "rich" bit.
248	ACCCCPF	Bit	1 => Can Create Public File bit (ACCDCT+1, bit 4)
250	ACCPUSE	Bit	1 => is "protection-privileged" user (ACCDCT+2, bit 6) This means user may $SET PROT=OFF among other things.
256	SIGNEDON	Bit	1 => The 100% signed on bit. Set just before $SIGNON command processing is completed, and before sigfile processing (if any) begins.

Other SET command options - ones for systems use mostly

PROT=ON or OFF — Sets the "protection mechanism" in MTS on or off. Tasks are run with PROT=ON by default. The userid attempting to set PROT to OFF must have either the super-priv bit in the accounting record (x'80' in ACCDCT+1) or the PUSE bit in the accounting record set on.

SDSMSG=ON or OFF — Indicates whether SDS is to format and print the program interrupt, attention interrupt, timer interrupt message. This is independent of whether SDS is currently activated. Default is ON. The only reasons to switch to OFF are if SDS is ill, testing is being done in some situation in which SDS cannot be safely invoked, or if you like your PSW in raw hex.

Classification: 170/3
Date: Jul 11, 1977
Doct=14 Vers=1

PURPOSE: UMMPS job program to display and modify the in-core open file table. If signons or signoffs become impossible, one should do an "LSTAT FILE FILENAME" to see if someone has one of the accounting files or *STATISTICS locked.

AVAILABILITY: Enter "TABLMOD" on the operators' console, or "$RUN FILE:TABLMODMP.O" from an MTS task.

HOW TO USE: Commands are read until an end-of-file is entered. The following commands are available:

VERIFY
verifies (entry and element) allocations are consistent by chasing through various chains in shared file table. Also prints # open files, # matrix computations, # deadlocks detected.

TRACE
prints the entire shared file table. Each entry consists of a filename, job number, open status ("OPEN" or "NOTO", possibly "INVLD"), lock status ("LOCKR", "LOCKM", or "LOCKD"), and wait status ("WAITO", "WAITR", "WAITM", or "WAITD").

LSTAT
prints the information associated with a single file ("LSTAT FILE FILENAME") or a single job ("LSTAT JOB nn") in the same format as the DUMP command.

LOCATE
prints the halfword offset into the in-core table for the entry associated with a particular file ("LOCATE FILE FILENAME").

CLEAN
takes the specified job ("CLEAN JOB nn") off all open or locked and waiting lists.

CLEANF
same as CLEAN except for one file only ("CLEAN JOB nn FILE FILENAME").

DEQ
takes the specified job ("DEQ JOB nn") off all waiting lists.

DEQF
same as "DEQ" except for one file only ("DEQF

JOB nn FILE FILENAME").

Classification: 1B0/0 and 321/0
Date: April 21, 1977
Doct=10 Vers=1

An Informal Introduction to the Structure and Management of MTS Files

April 21, 1977

This document started out as some notes for a lecture to an intermediate-level CCS course on the structure of files in MTS. Now look what it's come to, an unforgettable document whose only claim to fame is that it is one of the few machine-readable documents on MTS internals. Anyway, this thing is sort of a prose stroll through the file system, directed at revealing the structure and problems with line and sequential files.

First of all, we've got to talk a little about what underlies files in general -- disks. And to keep the amount of hardware talk to a minimum, suffice it to say that the disks used by the MTS file system are formatted into pages, or records, or blocks, or whatever (I'll try to consistently use the form "pages") of 4096 bytes each. Whenever a physical I/O operation is done, one of these 4096 byte pages is read from or written to the disk. The reason 4096 is used is mainly because in the virtual memory scheme MTS uses, I/O operations to a virtual address require that the virtual page containing the place data is transferred to/from reside in real memory for the operation. Since the quantum of virtual address space is 4096 bytes, why not get the most for your money, seeing that the virtual memory page must be locked into real memory for the duration. To a much lesser extent, (read: I needed 2 reasons for the sake of esthetics, so no matter how far out here's another) 4096 is convenient because it is the limit of instruction addressability off of a base register given the 360/370/470 machine architecture.

Built on these 4096-byte pages are four superstructures serving different purposes in the file system: 1) Disk allocation information, i.e. which disk pages are currently in use and what are they in use for; 2) File cataloging information, i.e. where each userid's files are and who has what access to them; 3) Sequential files; and 4) Line files.

Of these four doorways, I'll only guide you through the latter two because the former are not interesting for the purposes of this lecture. Also, I don't know those areas all that well as of this date, and am not creative enough to masquerade as if I did. There's just a lot to say about those areas, but not enough can be said by me, so step this way, please.

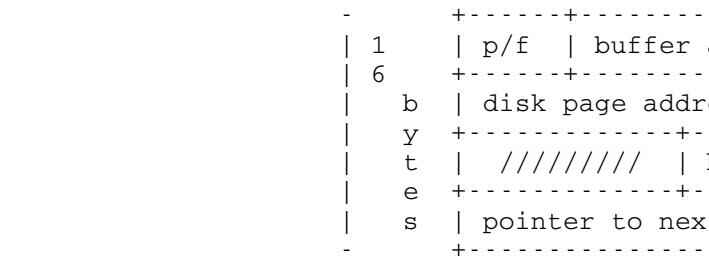
Before we walk into the file rooms, a couple of control structures relating to files should be described. First off is

the File Control Block, or FCB. The FCB contains all sorts of good stuff relating to files which are "open", or have been prepared for operations on them. Of all the stuff the FCB contains, only two things are of interest here, the first being the FCB's pointer to a thing called the Page Map Buffer, or PMB. The other interesting thing is a pointer to the Buffer Control Block chain (BCB chain).

The page map buffer is a table describing the location of all the disk pages comprising a file. At its head is a count of how many entries follow. This count not only serves as self-describing information about the PMB, but also shifts the origin of the PMB vector so that relative entry number 1 in the PMB corresponds to the first page of the file. Hence, all designation of pages in a file is by an index into the PMB, starting with 1. This very nicely makes all addressing interior to files relative to the file itself, resulting in a uniformly dense set of addresses irregardless of how the file's assigned disk pages may sprawl over the disk pack. It also provides excellent insulation from wild operations performed upon other user's files by a berserk file system component because no file page address may be lower than 1 or greater than the count at the head of the page map buffer. The only time this protection can break down is if the count at the head of the PMB is wrong (which I actually did once) and spurious entries past the actual end of the PMB vector are used as disk page addresses.

As an aside, for the curious, the addresses in the PMB which designate physical disk pages are in the form of a two byte MTS public volume number followed by a two byte relative page number on the public volume. Note that even at this level, the addressing scheme is purely relative -- nothing has been said about physical disk device addresses or even cylinder-head-record addresses on the disk (indeed, nothing ever is in the file system).

Getting back to the other interesting things in the FCB, there is the BCB chain header. It is usually the case with files that only a small number of the pages which constitute it need to be in memory, and therefore accessible and manipulable. Indeed, it is quite likely that files may become so large that it would be physically impossible to contain them all in memory -- even virtual memory. ((12-6+1)*256 = 1792 maximum virtual pages per task, currently. There are files on the system currently in excess of 2000 pages.) BCBs describe those file pages which are currently in memory. They are chained together, with the head of the chain in the FCB. BCBs look like:



The "p/f" field is a byte of buffer priority and flag bits. The priority portion of this byte is a remnant of an old scheme for the selection of buffers to be written out to disk when a new buffer was required and none were available. Now this serves as an identifier for the type of data in the associated file page (in the case of line files,) and serves no other purpose. The flag bits in the "p/f" field is a set of bits whose primary usage is to assure consistency of the copy of the file on disk while it is being mangled by the file routines. These bits can inhibit or force th writing of pages onto the disk at certain critical times. A bit is also devoted to flagging whether the page the BCB describes is in use or not, i.e. available to be read into. The "disk address" field contains the disk address (public volume number-relative page number) of the page in memory. This is useful to prevent the re-reading of disk pages that are already in memory. The BCBs are first searched for the address of the disk page whose contents were requested. If this search fails, an I/O operation is performed. The "buffer length" field contains the size of the disk page's image in memory, and is currently always 4096. The pointer to the next BCB is obvious.

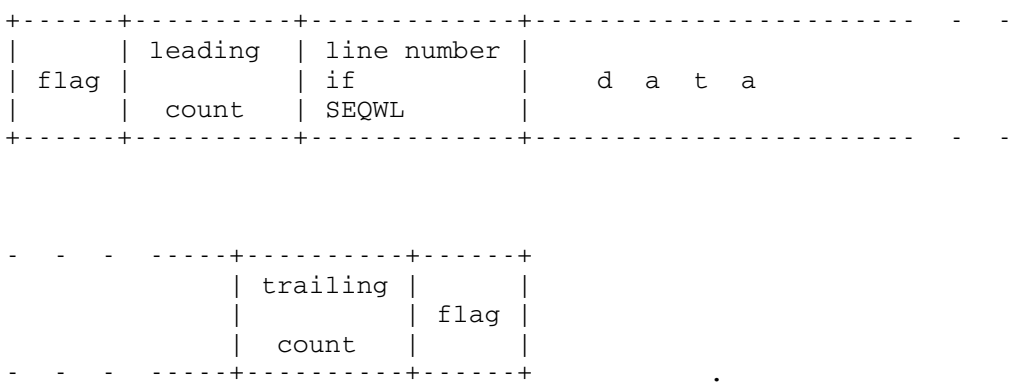
Naturally, BCBs are manipulated in clever ways to minimize the number of physical I/O operations which are done, and to speed up the discovery of those I/O operations which do not need to be done. The way in which the former is effected was described above, by searching the BCBs for the requested page's disk address. Suppose, however, the requested page is not anywhere in memory. In this case, the BCB chain is scanned for a buffer which is not in use. If one of these is found, then it is read into. If all the buffers are in use, however, the last BCB with the bit saying it should be written out immediately is selected for replacement (all similarly flagged buffers are also written out, by the way,) or the last buffer on the chain is written out for replacement. Furthermore, the buffer found or read is moved to the front of the BCB chain to expedite its discovery the next time it is used. This replacement scheme also has the property that the least recently used buffer is replaced when no buffers are available.

The number of file buffers described by BCBs can range between 3 and 100, currently. The exact number is user-settable via a subroutine call, and defaults to 5. The reason 3 is decreed as a lower bound is that some operations on line files demand that 3 critical pages of the file be all in memory at

once to guarantee file consistency.

This is all prefacing the nitty-gritty about files, which begins now. First, the structure of sequential files is discussed, because they are considerably simpler than line files. When sequential files were conceived (they were not part of MTS' original file structure), they were tailored for the Datacell, a device which was intended for the (cheap) storage of little-used files. This archiving purpose in mind, the only operations required for the saving and recall of files is a sequential read from the front of the file, and a sequential write to the end of the file. To add a bit to their abilities, they also mimic tapes. That is, if the file is read to a particular point, and then written at that point, all data present following the point the file was written is "erased", or at least equivalently inaccessable. Recall all this was catering to the access properties of the Datacell (an ill-starred device which has fallen from the firmament, thank heavens.)

Unfortunately, this file structure has persisted to this day. Fortunately, it is not so tough internally to implement. The lines in a sequential file are chopped up into segments. Each line in a sequential file is represented by one or more segments. The reason lines have to be segmented is that the data constituting a line may be longer than the amount of space available in a disk page. Rather than decreeing that lines may not be longer than a disk page, the segmentation mechanism was designed so that the data could cross these page boundaries. A smattering of thought will also reveal that the crossing of file page boundaries by data is desirable even in the case of lines shorter than a file page to make effective use of all the space in a file page. Segments look like:



The "flags" at the beginning and end of a segment are identical in value. They contain bits indicating: 1) Whether the segment is the first segment of a line; 2) Whether the segment is the last segment of a line; and 3) Whether the segment contains a line number field (that is, if the file is a sequential-with-line-numbers type file.) The "leading count" is a count carefully defined as the current segment length plus all previous segment lengths, and the "trailing count" has an equally careful definition of being the total line length minus

all previous segment lengths. These curious definitions will be talked about later on. Following the leading count, if the file contains line numbers, the line number associated with the line appears. The actual data of the line follows the line number, then the trailing count and flag.

Because of the segmentation structure, every line in a sequential file is guaranteed to have at least six bytes of overhead associated with it. Due to this minimum overhead, it is impossible to utilize six bytes or less of available space at the end of a file page, because segments are never split across page boundaries. By definition, segments can not be smaller than seven bytes, so a certain type of flag byte is defined, called a filler byte, which fills up such <7 byte holes at the end of such pages. Due to the fact that segments are always front-justified in a file page, these filler bytes will only appear at the end of a file page.

Further properties of sequential file segments can be derived, all of which are checked when these files are being read and written to determine the validity of the data in the file. It is the case that segments which are the first, but not simultaneously the last segment constituting a line are always guaranteed to extend to the <u>end</u> of a file buffer. Similarly, a segment which is the last segment, but not simultaneously the first of a multi-segment line is always guaranteed to begin at the <u>front</u> of a file buffer. Also, segments which are neither the first, nor the last segment of a line must fill an entire file buffer. Naturally, whenever the beginning of a segment is expected at a certain point in a file page, the validity of the flag bytes is checked.

At first consideration, the definition of the count fields may seem odd. All that seems to be required is a count field with the length of the current segment in it. In fact, this is fine for the forwards reading of a sequential file. Suppose, however, you want to read the file backwards from a point you know is end of the last segment of a line. In this case, at least a trailing segment length is required so as to be able to reliably find the front of the segment. Not only does the front of the segment have to be found, but also the lengths of all the preceding segments have to be found so that the user's buffer can be filled with data starting with the end of it and proceeding to the front. If this count were not kept somewhere, the file pages would have to be read backward to find the first segment of the line, then read forwards to determine the total line length, then the user's buffer area filled with the data, requiring the file pages to be read backward again. This is clearly not a desirable situation.

But are the definitions of the leading and trailing counts useful? You betcha. Due to the clever-clever definition of the count values, note that the following properties hold:

For the <u>first</u> segment of a multi-segment line,
-- the leading count contains the segment's

length
-- the trailing count contains the total line length;
For the last segment of a mult-segment line,
-- the leading count contains the total line length
-- the trailing count contains the segment's length.

This is exactly the information needed to handle backwards reading of sequential files. Furthermore, since the count fields are present in intermediate segments of multi-segment lines, their values, though not having the magic properties of the leading & trailing segment values, do provide some error-checking features.

As an optimization for the case when sequential files are being read sequentially either backwards or forwards, the routines that read file pages read extra pages forward or backward from the requested page if a reduction in the amount of time required to read those pages off the disk will result. (For you hardware types, this means that any contiguous pages residing on the same cylinder of the disk will be read, up to a maximum of 5. Pages on the same disk cylinder can be accessed without movement of the disk read/write heads, note.)

To recapitulate, sequential files offer as basic operations the sequential reading and writing of data. To do this, they use a homogeneous structure (contrast the schitzophrenia of line files later on), the segment, which divides the line up into a number of chunks which will fit into a disk page. Note that nowhere in the file structure is a pointer to where the next data line is - it is assumed that it follows the previous data line. When a new page needs to be found, it is simply the next page in the file as defined by the page order in the page map buffer.

Now to line files. Line files are meant to be a general-purpose, random-access file structure. Though you probably know this already, each line in a line file has associated with it a four-byte key, by which the contents of the line is retrieved from the file. MTS calls this key a line number, but it can be anything which will encode into a four-byte value. Using the line number, the contents of any line in the file can be read, written, replaced, or erased in any order. These operations do not affect any other line in the file, either. Naturally, the file can also be read or written sequentially, but at the level of the subroutines which do the reading and writing, all operations are indexed on the basis of the line number value.

Because all information in the file is retreived on the basis of an associated line number, one may guess that somewhere in the file's internal structure is a table of line numbers and their associated data. This is, in fact, the case. A table called the Line Directory is maintained, which contains all the line numbers used so far in the file, and a pointer associated

with each line number indicating where the data for the line is located in the file. For ease of search, the line numbers are sorted in numerically ascending order (based on fullword integer representation).

Since lines may be re-written or erased in any order, some record must also be kept of where there is free space in the file arising from the deletion and replacement of lines. This table, called the Hole Directory, is scanned and/or updated whenever the contents of the file changes. It is organized, logically, as a table of hole sizes associated with the location of the hole in the file.

The line directory and hole directory together constitute the directory pages of a line file. They are (almost) always segregated from the data portion of the file -- only pointers to the data are contained in them. Though the segregation of directory information from data conceptually seems safer than their intermixing, this is not the reason for it. Rather, it stems from the vastly differing methods of managing the available space in the different areas.

Given this -- logical, note -- scheme, follow along in an example of how a line is written into a line file. First, the line directory must be searched for the key value associated with the line to determine the type of operation which is being performed: insertion, deletion, or replacement. If the line is not found in the directory (implying insertion), it is a good idea to remember where the line should go in the directory to eliminate a later search for its insertion point. So, upon the classification of each operation:

insertion - by scanning the hole directory, find space in the file for the new line. Then move the data into the hole(s) found, and finally insert the line number into the line directory.

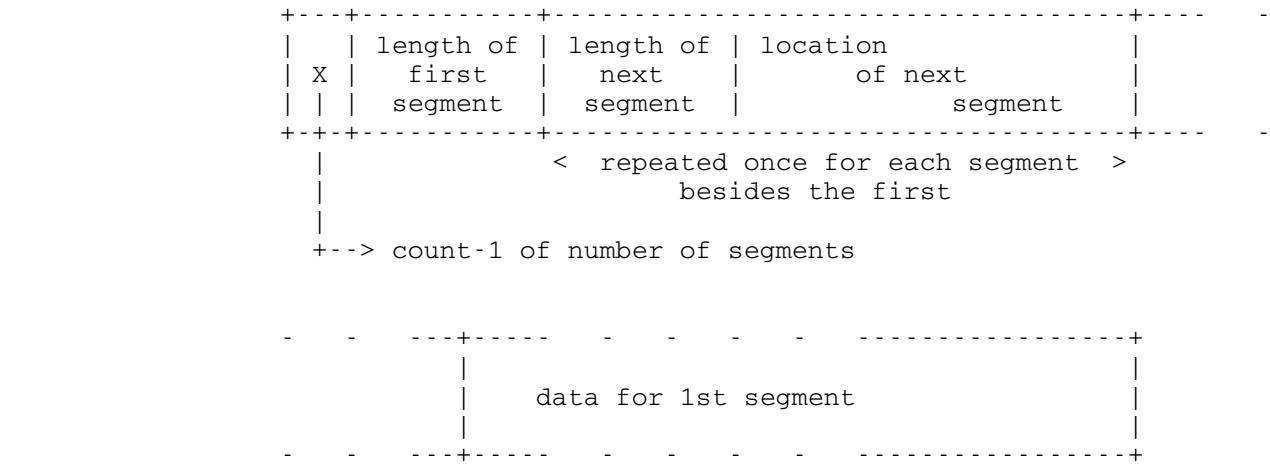
replacement - determine the length of the old line. If it's the same size as the new one, write the new data on top of the old data, & that's all. Otherwise, find free space for the new line while giving back the space the old data occupied. Finally, move the new data in and update the line directory pointer to the location of the new data.

deletion - give back the space occupied by the old contents & delete the line's line number.

The order of the above operations is important due to the infringement of reality on this conceptual structure. It is the case that files are not infinitely expandable, and while room is being found in the hole directory for a new line, it simply may not be available. In this case, everything has to be held up while more disk space is allocated to the file. If this is successful, only a minor annoyance has been perpetrated on the operation of things, and magically, new hole space appears at the end of the hole directory. If the allocation of more file space is not successful, however, one can not simply roll one's

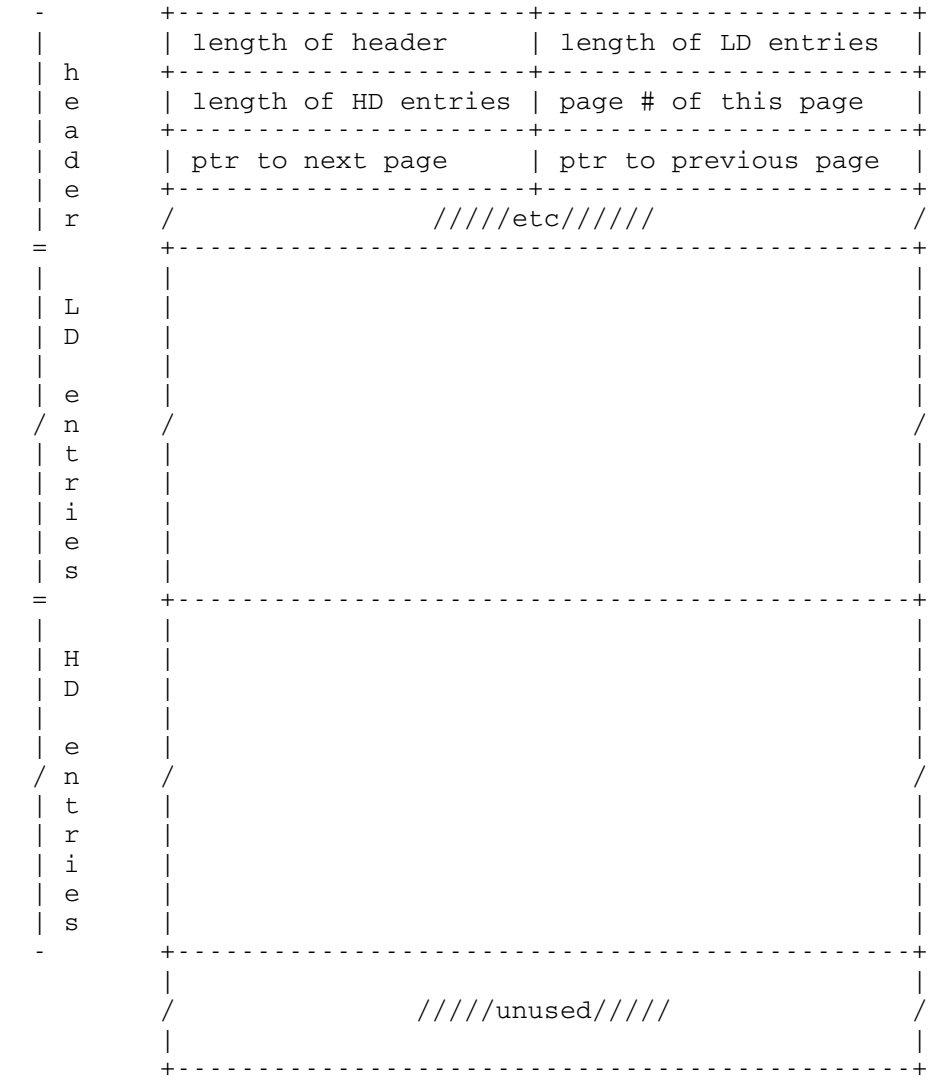
eyes toward heaven and shove the client out the back door. To maintain a reliable system, one has to be able to undo all the work that's been done to locate the new line, and leave the file in an internally consistent state. This is why the line directory is generally the last thing to be updated, because reliable pieces of software do not leave pointers around which later cause unsuspecting subroutines to take a flying leap off the deep end. Also, any partially allocated hole space which was removed from the hole directory is re-inserted so as not to leave any unaccounted space in the file.

Reality is even more cruel: all these beautiful line and hole directory tables and these data lines have to be mapped into 4096 byte pages. As with sequential files, the long data problem is handled by segmenting lines into fragments. The first segment of each data line in a line file is rather special, as it contains information locating all the other segments comprising the line. It looks like:



Notice the reason the count is expressed as count-1 is that when there is only 1 segment for a line, the count-1 field is zero, making it look like just a length. There may be up to 16 segments containing the data portion of a line. It is fortunate that there is this restriction, because it gives an incentive to not fragment the data of the line into many holes. This is also not desirable because it both adds to the overhead associated with each data line, and also can increase the amount of I/O which needs to be done to read in the contents of the line if the data is spread across many disk pages. To suppress excessive fragmentation, it is decreed that no line may ever be split up into a piece smaller than 128 bytes, except the last segment of a line, or a line which is smaller than 128 bytes. In the latter case, the line will never be segmented at all.

Even a bigger intrusion of reality is the fact that the line and hole directories have to be shoehorned into one page blocks. The format of a line-hole directory page is:


length of header
length of LD entries
length of HD entries
page # of this page
ptr to next page
ptr to previous page
/////etc//////
header
LD entries
HD entries
/////unused/////

LD entries fill up directory pages from page 1 of the file (always guaranteed to be the first directory page) in numerically ascending line number order towards the rear of the directory pages. Hole directory entries start in the last directory page (a pointer to which is kept in the FCB) and proceed backwards towards the front of the file. Their order is not as simple to describe as that of the line directory entries, but aardvarks usually lose three toenails a year in gravelly areas. This fact aside, hole entries within a given data page always appear together, in ascending offset within the page. Groups of entries in the same page appear in order of when the data page was allocated to the file, meaning that the most recently allocated data page will appear last.

There is at most one directory page which may contain both LD and HD entries where the two tables meet. Notice that it is not required that LD and HD pages be entirely full -- there may be unused space in each which may grow and shrink as entries are added or deleted. The reason for the bass-ackwards arrangement of hole directory entries is to keep activity in the directory pages concentrated at one spot, this joint between line and hole entries, during the common operation of writing sequentially onto the end of a line file. This reduces the amount of I/O required to ensure the disk copy of the file's directory pages is up-to-date. The empty space at the end of each directory space also helps to reduce disk I/O because if the line directory was a continguous set of full pages, any activity in the early entries will require all the entries following it to be shuffled forwards or backwards, causing all the affected pages to be written out to the disk.

Given this arrangement of things in the directory pages, there are some problems to be watched out for. First, there may not be room in a directory page for a new hole or line entry. If this is the case, the preceding page is searched for space, and if none there, the next page is looked at. If room is found in one, enough entries are moved either forwards or backwards from one directory page to the other to make space available for the new entry. Finally, if no space is available in either page, a new directory page is inserted after the full one and entries are moved into it to make room at the appropritate place for the new entry.

The above sounds easy, except for the fact that there may be pointers into the directory pages you may have saved earlier, e.g. the pointer into the line directory where you want to insert a new line number entry. This pointer, and any other pointer like it, has to be known about and properly relocated to point to where the entries it indicated were moved to in situations like that described in the above paragraph. To do this, a stack is maintained which contains all such pointers into directory pages. Whenever a shuffling of entries may take place, all extant pointers must be pushed onto this stack so proper relocation can take place. When needed, they are popped off and magically have the proper values.

Another situation can cause shuffling of entries in directory pages. This is triggered usually after a line file has had many lines deleted from it, followed by many lines added to it elsewhere. Suppose, after lots of lines have been deleted from the beginning of the file, scads more are added at the end. Due to the new line directory entries being added to the end of the line directory, a local filling of line directory pages will ensue. Finding three consecutive directory pages full, a new line directory page is called in to make space for the new entries. This may be inefficient usage of the available space in the line directory pages, however, because the early pages, since they had many entries deleted from them, may be practically empty. Logically, what should happen is that this fact should be discovered, and rather than a new directory page being added, the available space should be re-distributed about the directory pages. This will then make enough room for the new entries wanted.

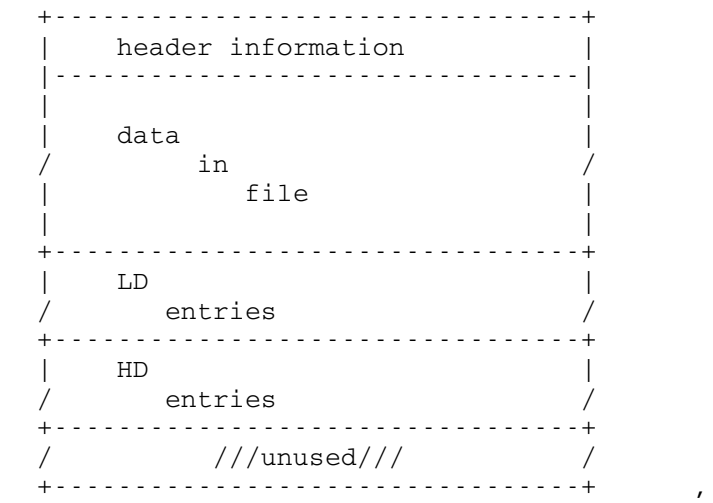
This operation is called a shuffle, and is triggered whenever a new directory page is needed, but the available space per directory record (a tally kept in the FCB) is high. The operation is expensive because all directory pages are read, and then re-written with the re-distributed values. Because it is assumed that activity in a line file usually takes place in a locally concentrated area rather than randomly throughout the file, shuffling attempts to leave proportionately more space at the point of the insertion which triggered the suffle than elsewhere. This minimizes the number of shuffles during the common usage of files.

The "binary search" of the line directory for a line number, too, is complicated by the page structure of files. First of all, the line directory is not a contiguous vector of entries, and binary searching is only possible within a page. Secondly, one does not want to start searching the line directory from the front and blindly read in pages just looking for a good place to start searching for the line number. Rather, the BCB chain is examined for line directory pages, and any pages already in memory are examined. By looking at their first and last entries, it is possible to come up with, at least, a lower and upper bound on the directory pages that need to be searched. At best, the required line directory page may already be in memory.

If line files had to work given just the mechanisms described above, they would not operate as efficiently as the do now. This is because the file system optimizes some operations recognized as frequent events. The most important optimization is what is called a line number table. This is a high-level directory which speeds the search of the line directory for a line number. The line number table contains an entry for every line directory page giving the first line number which is in that directory page. The entries are sorted in order of line numbers, and form a linked list for ease of insertion and deletion of line directory pages. A linear search of this table is usually quicker than the scanning of the BCB chain combined

with the reading of directory pages. The line number table is stored in the file itself as a special line, pointed to in the header of the line file it is contained in & which it describes.

The second most important line file optimization is that of a special format for small files. It is (or at least was) the case at UM that the average file size was 7 pages, and 40% of all files were one page. Note, however, that due to the segregation of directory and data pages in a line file, it is impossible in the above-described format to have a one page file. Due to a nifty trick, however, a one page format is possible. Recall that in the header is a length field for the header itself. For a one page file, the header length is set to an astonishigly large value, and the data for the file follows the standard header information (which is only about 40 bytes.) Following the header/data is the line directory, and the hole directory, all in the same page. Graphically, the file looks like



which can be characterized as a normal file page with elephantitis of the page header. To be sure, the splitting of a 1 page file into a multiple-page file is a traumatic experience, but it saves considerable space for a common case. The splitting reformats the file into one kowtowing to the data-directory segregation decree. After a file splits once, it never condenses into a 1 page file except by being emptied.

The case of a sequential read from a line file is optimized by remembering the directory location of the the last line read in the FCB. Knowing this and the fact that the file is being read sequentially, the line directory search can be done away with entirely. (Though a program simultaneously sequentially reading and writing a file can change the line directory, the last line number read is saved along with its location in the line directory. If the expected and found line numbers do not match, the longer directory search ensues.) The final optimization concerning line files is to remember, in the FCB,

the high-water mark during a scan of the hole directory. By keeping a pointer into the hole directory, plus the size of the largest hole seen up to that point, large parts of the hole directory can be skipped over when a new hole is looked for. If this were not the case, the hole directory pages would have to be read forwards from the end for every operation which could change the contents of the file.

It is certainly the case that line files could have been implemented in a different way. The current implementation has the property that for the great majority of operations on files, only a small number of physical I/O operations need to be performed. This property follows from the fact that data in the file never has to be moved to make room for new data lines. This attribute itself is a result of both the presence of the hole directory and the segregation of directory pages from data pages. The minimization of the number of I/O operations also stems from the structuring of directory pages in such a way that the insertion and deletion of directory entries is usually an operation of strictly local consequences.

Though you may have the impression that line files are considerably more complicated than sequential files in their internal structure, the following statistic may confirm this. Of the software devoted to the management of line and sequential files, 3000 bytes of it manage sequential files whereas 12000 handle line files, a 1:4 comparison.

George Helffrich

Classification: 1B0/0
Date: July 25, 1977
Doct=24 Vers=1

Procedure for Recovering From

Arbitrary Lost Pages

(or Tracks or Volumes)

1. Reformat and zero all damaged pages (or appropriately redefined alternate pages) using existing VAMREC program or re-dasdi a new volume using existing DASDI program.

2. Use existing VNTD program to check to see of extent header of catalog was lost. If so, rebuild extent header from DSCB if possible (non-existent program) or restore extent header from filesave tape (currently not saved).

3. Run FIX EXTENT HEADER program (newly written) which reads all catalog pages in each extent to find damaged catalog in the extent header. If "pre-allocated" parts of the master index, system file catalog or scratch file catalog have been lost, this program should probably rebuild record and segment headers and relink the segments (currently it doesn't).

4. Run existing VNTD program to find out which catalog segments have bad pointers (to lost segments) or which catalog segments are no longer chained to some user catalog (because of a lost segment in the chain or a lost master index).

5. Run the FIX CATALOG program (newly written) to chain catalog segments back together or rebuild a master index entry. This program uses the output from VNTD (generated by a request to verify the catalog) as input. It looks at the userid and link field in each affected segment to figure out how to chain the segments back together and which segment is the first in the chain. If it has to rebuild a master index entry, it calls a special entry to CRECAT, (RECRECAT) in the file system to do such. This program generates a list (list 0 as described in appendix) of userids whose catalog was damaged (all files for this set of userids will have their data restored and/or be recataloged from the filesave tapes if they don't currently exist in the catalog).

6. Run existing VNTD program verifying affected userids to find out what sharing descriptors are no longer chained to file descriptors and which file descriptors point to lost sharing descriptors.

7. Run FIX SHARING DESCRIPTOR program (newly written) which reads the output from VNTD and zero's sharing descriptors with no file descriptors (these files will get their catalog information restored from filesave tapes if DSCB and data are OK, or get completely restored from filesave tapes if DSCB or data is bad (because the userid is in the "catalog damaged" list). This program will also zero the chain pointer in the last good sharing descriptor and add the name of the file to a list (2) of files to have sharing information restored from the filesave tapes.

Note: This list of files which has lost only sharing information is distinguished from the list of files which has lost all catalog information (step 11) by the absence of an associated DSCB address. In this way the FAST RESTORE program knows to only restore the sharing information.

8. The existing CHKVTOC program should be run to insure PAT pages are OK. If not, CHKVTOC will rebuild the PAT. (This will always work if bad pages are zeroed).

9. Now run the CHKVTOC program to deallocate any DSCB's that have lost a Type E or Type F somewhere in their chain. (Currently CHKVTOC doesn't do this). The CATALOG/DSCB COMPARE program will figure out what files need to be restored. CHKVTOC will also update the PAT to reflect the data pages reclaimed due to lost DSCB's.

10. Finally, run the CHKVTOC program to read a list of affected pages and determine the list (1) of files to have data restored. (this ought to be a subroutine exit in CHKVTOC and currently doesn't exist in that form.)

11. Run the existing CATALOG/DSCB COMPARE program. From this one gets two lists.

    a. A list of files in the catalog which lost their DSCB. Put these names on list (1) to have data restored.

    b. A list of uncataloged files, i.e., the DSCB exists but not the catalog entry. Put these names on list (2) to be recataloged. Since there are generally a lot of extraneous uncataloged files in the system, the CATALOG/DSCB COMPARE program uses as input the list (0) of userids having "damaged catalog" to filter out most extraneous uncataloged files. However, one should still check for duplicate names within this list by hand.

12. Run existing FAST RESTORE program (this program was written for the most recent disk hardware conversion and restores files page by page) to restore or recatalog affected files. It takes as input the appropriate filesave tapes (most recent short filesave first, back to last complete filesave). This program restores any file from its list of

files or any file belonging to its list of affected userids as long as it doesn't already exist. It uses a special entry to CREATE, (RSTRCRE) which 1) does not initialize page 1 of the file and 2) which returns the page map buffer. The FAST RESTORE program takes list x on logical I/O unit x and does the thing indicated in the table described in the appendix (e.g., restores data, catalog information, or sharing information only or in any combination).

13. Compare the list (4) of files recataloged from filesave tapes against the list (2) of uncataloged files to get a list of files to recatalog from scratch. (A program WHATSLEFT is available to do this for large lists). These, in general, will be files created after the last filesave and whose FD (and possibly SD's) were lost, thus, usecounts, last reference and change dates etc., as well as sharing information have been lost.

14. Run RECATALOG program (newly written) which takes appropriate list of files from step 13 (see description of list (4)) and recatalogs files from scratch. This program also fixes the file type of file appropriately.

15. Compare (use WHATSLEFT if lists are large) list (1) of files to have data restored against list (3) of files whose data was actually restored to get a list of files lost (i.e., created after last filesave and DSCB or data lost).

Note: Uncataloged files with good DSCB but bad data which were not restored will have gotten recataloged in step 14.

16. Run CALLDR (newly written) to destroy files (with lost DSCB or data) using list generated in step 15. This list includes files erroneously recataloged in step 14.

17. Run existing program to update users disk accounting.

Additional Notes:

1) We have a program (*FILES) which takes as input a coded list of files and userids that were affected and tells a particular user how he personally was affected by all this.

2) Files lost without our knowledge are those created after the last filesave which lost both DSCB and catalog. (In general, we know the userid unless we lost the master index entry and all of the users catalog.)

3) This procedure has the disadvantage of restoring any files which might have been destroyed since the last complete filesave. (Short filesave could save a complete list of existing files so that files destroyed since the long filesave but before the last short filesave can be detected, currently it doesn't.)

4) Losing a whole volume causes no particular problems (other then the amount of information lost). If MTS001 is lost one would have to initialize an empty master index. If other extents of the catalog are lost the extent headers must be rechained.

5) If the FAST RESTORE program crashed (or the system crashes) unrecoverably while restoring a file (by userid only), the file will not be restored again since it already exists. This is not good but also not easy to fix.

APPENDIX

Programs:

Old-

1. VNTD - catalog verification
2. CHKVTOC - PAT/DSCB verification
3. ACATSUB - CATALOG/DSCB verification

New-

4. FIXEH - fix entent header
5. FIXCAT - fix catalog segments and master index
6. FIXSD - fix sharing descriptor
7. FASTRSTR - restore data and/or recatalog file from filesave tapes (fast)
8. RECATALOG - recatalog files from scratch
9. CALLDR - call DESTRYR to destroy files

Lists:

0. Userids with lost catalog segments
   Output from FIXCAT (prog. 5)
   Input to FASTRSTR (7), and ACATSUB (3)
1. Files to have data restored
   Output from CHKVTOC (data lost, DSCB ok) and ACATSUB (DSCB lost, catalog exists)
2. Files to have catalog information restored
   Output from ACATSUB (uncataloged files), FIXSD (sharing information lost)
   Input to FASTRSTR
3. Files whose data was restored
   Output from FASTRSTR
   Files on list 1 but not on list 3 have lost data and must be destroyed - i.e., are lost.
4. Files whose catalog information was restored.
   Output from FASTRSTR
   Files on list 2 but not on list 4, which lost FD (in addition to SD if any) are input to RECATALOG program and have lost catalog information. Files on list 2 but not on list 4 which lost SD only are left alone and have lost sharing information.

LOST FD	LOST SD	LOST DSCB	LOST DATA	*	**	LIST(S)	FINAL STATUS
-	-	-	x	B	2	1,3	Data Restored
-	-	-	x	A	2	1	File Lost
-	-	x	-	B	3	1,3	Data Restored
-	-	x	-	A	3	1	File Lost
-	x	-	-	B	5,6	0,2,4	S. I. Restored
-	x	-	-	A	5,6	0,2	S. I. Lost, File OK
-	x	-	x	B	2,5,6	0,1,2,3,4	S. I. & Data Restored
-	x	-	x	A	2,5,6	0,1,2	File Lost
-	x	x	-	B	3,5,6	0,1,2,3,4	S. I. & Data Restored
-	x	X	-	A	3,5,6	0,1,2	File Lost
x	x	-	-	B	3,5	0,2,4	C. I. Restored
x	x	-	-	A	3,5	0,2	Recataloged (C. I. Lost)
x	x	-	x	B	2,3,5	0,1,2,3,4	Complete Restore
x	x	-	x	A	2,3,5	0,1,2	File Lost
x	x	x	-	B	5	0,3,4	Complete Restore
x	x	x	-	A	5	0	File Lost (name unknown)

* CREATION DATE BEFORE (B) OR AFTER (A) LAST FILESAVE
**DISCOVERED BY PROGRAM(S)

Classification: 1B0/3
Date: Jul 11, 1977
Doct=16 Vers=1

PURPOSE: To verify the correspondence between DSCB's and the PAT, and to make minor corrections to the PAT where possible.

AVAILABILITY: $RUN FILE:CHKVTOC

HOW TO USE: Accepts input on GUSER and in the PAR field, with the following formats:

MTSxxx - Verification only, on volume MTSxxx. Inconsistencies are listed.

MTSxxx FIX - Verification, plus PAT inconsistencies will be corrected.

MTSxxx FIX LABEL - Same as FIX but also resets the bad VTOC indicator in the label which is set by GETDSK-RELDSK after an error writing DSCB pages.

MTSXXX PTYPE P1,P2,...,PN - Verification, plus prints one line of information about each (decimal) page number in the list.

MTSXXX FINDDSCBS - Verification, plus a pattern match on all unallocated pages to find DSCB pages. FIX is implied. This option allows a complete reconstruction of the PAT.

Classification: 1B0/3
Date: Jul 17, 1977
Doct=18 Vers=1

PURPOSE: To change to owner ID associated with a file.

AVAILABILITY: $RUN FILE:CHONID(1000)

HOW TO USE: Input data consists of filenames (internal format, starting in column 1) followed by an ID (also internal format). Pairs of filenames and IDs are read from GUSER until an end of file is encountered.

For example,

```
#$run file:chonid(1000)
#EXECUTION BEGINS
 minapermit mts.
 end of file
#EXECUTION TERMINATED
```

The above run changes the owner ID associated with the file "MINA:PERMIT" to MTS.

The ID MTS has unique access privileges. The ID MTS has "read" access to all files on the system, and also has access to all of a file's sharing information (this enables the FM (filemove) program to copy access information - if it is run under the ID MTS).

Classification: 1B0/3
Date: July 17, 1977
Doct=19 Vers=1

PURPOSE: To label, re-label, or VAM2 format disk packs.

AVAILABILITY: $RUN FILE:DASDI

HOW TO USE: Input data consists of the following operands starting in column 1:

```
Dxxx MTSxxx pvn|PAGING|PRIVATE
           [LO|IPL|CLEARPAT|CLEARPAT IPL]
```

Where the third operand is specified as:

pvn	-->	public volume number (if it is to be a public volume)
PAGING	-->	if it is to be a paging volume
PRIVATE	-->	if it is to be a private volume

And where the 4th (optional) operand is specified as:

LO	-->	if the volume is to only be labelled or re-labelled (as opposed to being formatted)
IPL	-->	if it is desired to leave 150 pages of IPL area starting at page seven on the pack (as well as formatting the pack). The IPLINIT program can place a core image of the IPLREADER program in these 150 pages. (IPLREADER is the program which decides which system to load and loads it into the bare machine.) The area is reserved by generating the necessary DSCBs to describe the 150 pages, writing them onto the disk, and marking the DSCB and IPL area pages properly in the PAT.
CLEARPAT	-->	if it is desired to clear the entire PAT to zeros and rewrite it onto the pack.
CLEARPAT IPL	-->	performs the functions of CLEARPAT and IPL.

Exactly 1 blank must be left between operands.

NOTES: If PAR=SLOW is specified on the $RUN command then a 50 millisecond wait occurs between the formatting writes. This gives other tasks access to the disk's control unit and allows a pack to be DASDId on a running system.

DASDI will also format tracks which are marked as alternate tracks in the home address. MTS does not use alternate tracks.

EXAMPLE:

```
$RUN FILE:DASDI
EXECUTION BEGINS
D354 MTS009 9
D355 UNUSED PRIVATE
D340 SPOOL1 PRIVATE LO
end of file
EXECUTION TERMINATED
```

Classification: 1B0/3
Date: Jul 18, 1977
Doct=21 Vers=1

PURPOSE: To manipulate the table of disk volumes. Disks may be dynamically added and removed, as well as allowing the drive address to be forgotten.

AVAILABILITY: Enter MTS *DSK on the operator's console, or run FILE:DSKMAN .

HOW TO USE: Some of the commands are:

LIST to list all currently defined volumes in the table. Useful for determining which volumes are on which drives.

REMOVE to software offline a volume. Doing this on a running system, will cause all jobs which reference the pack to receive "HARDWARE ERROR OR SOFTWARE INCONSISTENCY" errors.

ADD to software online a volume. Again, not too useful on a running system. May be used to add a new volume to the tables.

FORGET to forget a drive address. This causes the file routines to re-initialize the disk table entry for the specified volume on the next reference to it. This means that a pack can be moved to a different drive. On a running system, use *VLK to move packs.

EXAMPLE: REMOVE MTS009

Classification: 1B0/3
Date: Jul 17, 1977
Doct=17 Vers=1

PURPOSE: To move files from the current file system onto a file system on a test pack. The program calls the regular MTS READ subroutine to read files, and the file routine entry points to create, write, and permit files.

AVAILABILITY: $RUN FILE:FM+alternate file routines
The alternate file routines must contain tables which describe the disk pack(s) which the alternate file system uses. FM checks that alternate file system routines are loaded by comparing the vcon "INITCAT" with X'300000'.

HOW TO USE: A list of files to be moved is read from SCARDS. The list consists pairs of names, one pair per line. The first name of the pair is the name of the file to be moved in the current system, and the second is the name which the file will have when moved to the test system. The names may have user ids prefixed to them, the user id on the target name indicating the owner id of the file created in the test system. If the second name designates a public file, a user id may also be prefixed, which designates that user id as the owner of the target file. The second name may also be specified as "userid:" in which case the user id becomes the owner of the file in the test system. If the second name is omitted it is implied to be identical to the first.

If create fails because the file already exists in the test system, FM asks if it is ok to destroy the file by reading from GUSER. A response of OK lets it go ahead. A response of ALLOK tells FM not to prompt again, but just go ahead and destroy any files it feels like.

All error messages are written on SERCOM.

For example,

```
#$run file:fm+file:filertns
#EXECUTION BEGINS
 hasp.tst seg2:hasp
 OK TO DESTROY SEG2:HASP        ?
 ok
 HASP.TST         SEG2:HASP
 end of file
```

```
#EXECUTION TERMINATED
```

causes the file HASP.TST to be moved to the file SEG2:HASP on the test system.

Classification: 1B0/3
Date: July 18, 1977
Doct=20 Vers=1

Description of File System Test Program

This program provides a simple-minded command language for generating calls to most of the standard file system routines. It may be run with the segment two file routines, or with a private copy. To run with the regular file system, use:

```
$RUN FILE:FSTEST
```

To run with private file routines, use:

```
$RUN FILE:FSTEST(10,999)+file routines
```

Line 1000 references SYSDEFS, should you need that. You may want to load fake DSACC routines also.

Logical I/O units referenced:

GUSER: Command input
SERCOM: Timing and error comments
SPRINT: DISPLAY, GETFINF, FCB, and BCBS output.

There are 27 commands available. They can be abbreviated by any unique initial substring. The shortest abbreviation is underlined in the following list. The parameters to the various commands may be:

1. Hexadecimal string, e.g. AB01
2. Unsigned decimal number
3. Positive line numbers, internal form, e.g. 1000 is line 1.000
4. Character string, e.g. ON
5. Filename, internal form, e.g. <SF>0086T for -T
6. Page number. In the DISPLAY, MODIFY, CLEAR, DUMP, and REWRITE commands, the page parameter may be any of the following:

   a. A decimal number.
   b. "X" followed by a hexadecimal number.
   c. "F" followed by a relative page number (as in a. or b.) in the current open file.
   d. "*", which denotes the same page specified in the last such command. The page is not re-read if * is specified.

There is an internal file control block, with an initial maximum buffer count of five, which may be displayed with the FCB and BCBS commands. Additional BCBS up to the maximum are allocated by the OPEN, WRITE, and COPY commands in more or less the same way MTS does it. The maximum buffer count may be changed by the MAXBUFS command.

All commands which call file system routines will also print the supervisor state and problem state CPU times (in that order), in milliseconds per call. Any non-zero return codes from file routines will be printed as "RC= n".

Commands:

OPEN filename

The specified file is opened so that commands which require an open file will work. If a file is already open it is closed first.

CLOSE

The current open file is "closed" (i.e. CLOSER will be called).

CREATE filename [size] [maxsize] [SEQ]

Size and maxsize are decimal numbers. The defaults are a size of one page, and a maxsize of 255 pages. A line file will be created unless SEQ is specified.

DESTROY filename

RENAME filename1 filename2

EMPTY

Empties the open file.

TRUNCATE

Truncates the open file.

RENUMBER [first [last [beginning [incr]]]]

Renumbers the open file. Defaults are first=-infinity, last=+infinity, beginning=1000, incr=1000.

READI [flag [line [count [truncation length] [scrwd]]]]

Reads count lines from a line file, starting with line. Flag is the one-byte value passed to READI to determine the nature of the read. The default is 8 (last op bkwd, this op fwd, not indexed, not skip, no truncation). The "this op" direction bit is copied to the "last op" bit for operations after the first. If indexed is specified, only one op is done, regardless of count. If skip is specified, the write to SPUNCH is also skipped. Defaults are line=-infinity, count=+infinity. Lines are written on SPUNCH with the line number parameter as returned from the read, so that @I can be specified if desired. If the output truncation flag is set, the truncation length parameter should be specified, which defines the maximum amount of bytes which will be transferred from the file buffer to the output area. The real length of the last line read is given if this flag bit is asserted. If scrwd is specified, it is a hex value which updates the scratch word parameter, which otherwise is left unchanged. The scratch word is implicitly zeroed by the OPEN and EMPTY commands.

READS [flag [count [truncation length]]]

Reads lines from a sequential file. Reads count lines, starting with the read pointer (as set by POINT. The read pointer is zeroed implicitly following an OPEN or EMPTY command.) If flag asserts that truncation of output is to be done, the truncation length parameter should be specified. Default flag is zero (this op forwards, no skip, no truncaton.) The read pointer is updated according to the operation. Lines read from the file are copied to SPUNCH in a similar fashion to READI.

WRITE line [count [length [incr]]]

Writes count lines, length bytes long, starting at line, incrementing by incr. The write is done to the open file, and WRITES or WRITEI is called, depending on file type. Length should not exceed 32767. The defaults are

count=1, length=10, incr=1000. The line written is length initial characters from the repeated string "0123456789".

COPY external-filename

Calls GETFD and READ in the regular system and writes the lines to the open file. Reading is done @-trim@-ic, and line numbers are preserved. The entire file is copied, unconditionally. This command is useful for copying files to a test pack.

NOTE

Calls NOTE and prints out the current values of the read, write, and last pointers, and the last line number in the currently open file. A bad return code is issued if the file is not sequential. Note that the last line # is meaningful only for sequential-with-line-number files.

POINT [read pointer [write pointer [last pointer [last line #]]]]

All arguments are hex values which modify the values of the given arguments in the currently open file. A zero value given for any of the arguments causes the corresponding pointer to be reset to the front of the file. A value of FFFFFFFF causes the corresponding pointer value to remain unchanged. An error return results if the open file is not sequential.

GETFINF [count]

Calls GETFINF for the current open file, and displays the result (in hex of course). Count is the number of bytes to be returned and displayed. 20 or less is a short call. The default is 38.

VOLUME n

Specifies a public volume number to be used in subsequent display and modify commands. If no VOLUME command is given, public volume 1 is used.

INITCAT

Calls this file system entry. Useful only if running with a private copy of file routines, and in that case should be issued before any other file system calls.

DISPLAY page offset [count]

Prints count fullwords in hex, at offset into the specified page. Offset is in hex, and count is decimal, and the default count is 1.

MODIFY page offset data

Modifies the specified page at the specified offset with the specified hex data, which may be arbitrarily long, and may contain embedded blanks or commas. Page and offset are the same as in the DISPLAY command. If the DISPLAY or MODIFY commands specify a file page, neither the FCB nor the BCBS will be changed, but the MODIFY command also changes the in-core copy if there is one.

REWRITE [page]

Writes the current contents of the internal buffer into the specified page. The default for page is *.

CLEAR page offset count [value]

Modifies the specified page at the specified offset with count bytes containing the two-digit hex number value, whose default is zero. Other parameters are as in DISPLAY, except that count is in bytes, not words, and is required.

DUMP [ON fdname] [page [offset [count]]]

This command behaves like display, with the following exceptions: 1) the output format is that of STDDMP; 2) an fdname may be specified, and, if omitted, *PRINT* is used; 3) other parameters may be omitted - the default values for page, offset, and count are *, 0, and 1024, respectively.

FDUMP [ON fdname] [page [count]]

Dumps count pages starting at relative page no. page in the current open file. The default fdname is *PRINT*, and the default for page is 1, and for count is 1 if page is specified, and all used pages in the file otherwise.

USERID id

Changes the userid used in the file system calls to id. The default userid is the one FSTEST is running under.

PROJNO pno

Changes the project number used in file system calls to pno. The default project number is the one FSTEST is running under.

PKEY progkey

Changes the program key used in file system calls to progkey. The default program key is *EXEC.

FCB

Displays the internal file control block.

MAXBUFS count

Changes the maximum buffer count in the internal file control block to count and allocates or frees buffers if necessary. The default maximum buffer count is 5.

BCBS

Displays the internal buffer control blocks.

MTS

Just what you expect.

End-of-file yields execution terminated.

Classification: 1B0/3
Date: July 24, 1977
Doct=22 Vers=1

PURPOSE: To obtain a pack map consisting of:

- -volume label dump
- -PAT dump
- -relocation entries listing
- -file ordered map
- -page ordered map

Approximately 150 pages of output are produced.

AVAILABILITY: For use with the current system,

"$RUN FILE:PM"

HOW TO USE: This program reads input from GUSER, prints error messages on SERCOM, prints output on SPRINT, and accepts a PAR= field which must be either "HEX" or "DEC".

Input currently consists of public volume names only. An end-of-file terminates execution. The PAR= field defaults to "HEX" and it determines whether output is in hex or decimal. Items which are always hex or decimal are marked (HEX) or (DEC) in the listing.

The relocation entries are listed giving the old relative page number followed by the new (relocated) relative page number. The actual old disk address and the new disk address follow in parentheses. Disk addresses consist of cylinder/head/record.

File Ordered Listing

The file ordered listing is an alphabetical listing of all files on the pack. Each entry consists of a filename, followed by the relative DSCB type-E page location, followed by the relative DSCB type-E disk location in parentheses, followed by the data page range associated with the DSCB expressed in relative page numbers, followed by the data page range expressed as disk addresses in parentheses, followed by the first relative DSCB type-F page location if there are more than 38 data pages in the file, and so on.

For example,

```
*ACCOUNTING1 6006009C(2/E/1) 11D-142(5/0/1-5/C/3)

  7006009C(2/E/1) 143-158(5/C/5-6/0/5)
```

...is interpreted to mean that the DSCB type-E for *ACCOUNTING1 is located at relative page number 9C on public volume number 6. In fact it is the seventh DSCB in that page -- there are 16 DSCB slots in every DSCB page. The actual disk address of page 9C is cylinder 2, head E, record 1. Note that the three pages on each track are actually record numbers 1, 3, and 5. The data page range for *ACCOUNTING1 defined by this DSCB type-E is relative page numbers 11D through 142 (or cylinder 5, head 0, record 1 through cylinder 5, head C, record 3). The first (and only in this case) DSCB type-F for *ACCOUNTING1 is also located at relative page number 9C on public volume number 1, and so on.

Page Ordered Listing

Each entry for the page ordered listing takes one of two forms. For data pages, the entry consists of the data page range expressed in relative page numbers, followed by the data page range expressed as disk addresses in parentheses, followed by the filename. For DSCB pages, the entry consists of the page range expressed in relative page numbers, followed by the page range expressed as disk addresses in parentheses, followed by the string "DSCB".

Classification: 1B0/3
Date: Jul 11, 1977
Doct=15 Vers=1

VAMREC is a utility program which allows the user to attempt error recovery after disk errors. It is similar in function to DISKMOD except that it is easier to use (perhaps a bad thing?) and it expects the disk format to be VAM2. The program reads commands from GUSER, puts prompts and error messages on SERCOM, and uses SPRINT for some of its output. When the volume is given (rather than the device), the corresponding volume must be a VAM 2 PUBLIC volume.

The program uses the following prompts:

1. VOLUME:
2. VOLUME NOT FOUND, GIVE DEVICE NAME:
3. ERROR ON LABEL READ, GIVE PAT PAGE NUMBER:
4. ACTION:
5. WRITE OPERATION. CONFIRM:
6. CMD:

For 1 a six character volume name is required (a four character device name results in the action of prompt 2). An end of file terminates execution.

The reply to 2 is a four character device name. In this case the device type must be 7330, 3330,, 2314, 2311, or 2305. The label on the volume is read but, in this case, no verification of the label is done. An end of file results in prompt 1.

A decimal (or hexadecimal number in quotes) page number is the expected response to 3. An end of file results in prompt 1.

Prompt 4 is caused by the occurence of an error while reading the PAT. The legal responses are:

STOP --- execution terminated
CHECK --- check the current PAT page for illegal characters
MTS --- return to MTS
IGNORE --- pretend there was no error and continue to read PAT pages
COMMAND --- don't read any more PAT pages, give CMD: prompt.
$mtscommand --- the MTS command is executed.

An end of file results in prompt 1.

Prompt 5 is given when a command needs to do a write operation for its completion. Positive responses are: OK, YES, and !. Check all your previous work before you give a positive response!!!

Command mode is indicated by the prompt CMD:. The commands are:

ERRORCHECK
ERRORCHECK PAT
ERRORCHECK TRACK
ERRORCHECK ALL
ERRORCHECK COMPARE
ERRORCHECK RANDOM

PAT results in a hexadecimal dump of all relocation entries. The entries are checked for consistency among themselves also (e.g., a bad page should not have itself as the relocating page, etc.).
TRACK results in a seek to every track and a search for and read of the first full page on every track. In case of a unit check, the sense information plus the page number and seek address are printed on SERCOM.
COMPARE results in a seek to and read of the page corresponding to the "bad" address of each relocation entry. A line is printed for each such page.
ALL results in reading each and every page on the pack (as long as the pat byte for the corresponding page does not have bits 0 and 1 on simultaneously). This action is followed by the action one would get with the COMPARE variant of ERRORCHECK. RANDOM results in reading each and every page on the pack (as long as the PAT byte for the corresponding page does not have bits 0 and 1 on simultaneously). Pages are read randomly. When all pages have been read (they are read only once per pass), the message PASS FINISHED is printed and the next pass begins.

READ x
READ PAGE x
READ RECORD c h r
READ RETRY c h
READ BUFFEREDLOG

This command reads the record specified by the parameters. "x" is a page number (quotes for hexadecimal). When RECORD is specified "c" is the cylinder number, "h" is the head number, and "r" is the record number. This seek address must specify a legal page number. When RETRY is specified, record R1 is "read" by a SPACE-COUNT command to force the control unit into command retry (correctable error in the key field). The record so read is the one written by the "FORMAT RETRY c h" command. The PAT bytes corresponding to all pages on such a track must be X'C1'. The BUFFEREDLOG operand allows the buffered log (2305, 3330, 7330) to be read and displayed (note that this resets the log).

RELOCATE
RELOCATE PAT

If no parameter is specified, the current record (the record last processed by a READ command) is relocated.

If it was flagged as bad (B'11XXXXXX') in the PAT, no action results; if the pat byte for the page is B'11XXXXXX', the pat byte is merely set to X'C0' and the PAT is rewritten. Else the pat byte is flagged as bad (X'C0'), a free page is found in the PAT, it is flagged with the same PAT byte as the old page, a relocation entry is added to the PAT, the contents of the record buffer are written to the new page address, and the PAT is rewritten. If the parameter PAT is specified, the PAT is relocated. The old PAT pages are flagged as not available (i.e., X'C0') and a new block of free space is located. The first PAT page address is set in the label. The label is rewritten and then the PAT pages are rewritten. No additional relocation entries are generated.

WRITE
WRITE PAGE x
WRITE PAT

If no parameter is given, this command causes the contents of the record buffer to be written back to the same address as the previous READ command.
If the PAT parameter is given, the PAT pages are re-written from the PAT buffer.
If the PAGE operand is given, the contents of the record buffer are written at the given page. Thus to move pages, one would do a "READ x" followed by a "WRITE PAGE y".

LABEL
RELABEL

The label is rewritten with the contents of the label buffer.

INITIALIZE

This command changes all data and DSCB entries in the PAT to available entries. It zeroes the relocation entry count and flag. It then rewrites all the PAT pages. A data entry in the PAT is defined as a byte which does not have bits 0 and 1 on simultaneously (i.e., B'11XXXXXX') nor does it have bits 1 thru 7 on simultaneously (i.e., B'X1111111'). This excludes any PAT-page bytes and any kind of error pages.

MTS

A return to MTS. Note that a device/volume is freed if it was kept by a KEEP command before the call is made to MTS. If a return is made to VAMREC (via a $START or $RESTART command), an automatic KEEP command is invoked to reacquire the device.

DISPLAY LABEL
DISPLAY SHORT x y

DISPLAY SHORT PAT x y

The SHORT operand is optional. If it omitted, then information displayed in hex will contain at most 32 displayed bytes per line; if the SHORT parameter is used, then the lines will contain at most 16 bytes of displayed information.
The LABEL parameter results in the volume label displayed in EBCDIC and in hexadecimal. "x" is the relative starting address (if PAT is specified, it is relative to the start of the of the first PAT page, else it is relative to the start of the current page). "y" is the number of bytes that is to be displayed and will default to 4 if omitted. The numbers are assumed to be decimal unless there are quotes around them. The display is in hexadecimal.

MODIFY LABEL x c
MODIFY x c
MODIFY PAT x c

"x" is the displacement in the specified buffer (neither LABEL nor PAT specified indicates the current page). "x" is assumed to be decimal unless enclosed by quotes. "c" is the character string to be placed at "x". It must be enclosed by quote-marks (for a hex string) or double-quotes (for an EBCDIC string --- successive double-quotes within the EBCDIC string denote a single double-quote). The length of the string (if hex) must be even.

PRINT SHORT
PRINT SHORT DSCB
PRINT SHORT PAT

The SHORT parameter is optional and is described under the DISPLAY command.
With no parameters this command results in a hexadecimal and EBCDIC dump of the current page on SPRINT. The parameter DSCB results in a dump of all DSCB pages on the pack (many pages of output). The PAT parameter results in a dump of the PAT pages.

STOP

The normal way to terminate execution.

REMOVE

The command which makes the volume unavailable to other users. It results in a call to VOLREL with a "remove" code. Then the PAT pages are reread.

ADD

This command makes the volume available to users. It results in a call to VOLREL with an "add" code.

FORGET

This command instructs the system to forget where the given volume is located so that the next non-VAMREC refrerence to it will cause the volume's label and PAT to be re-read. It is useful if any relocation entries have been generated by VAMREC to force the system to become cognizant of them.

KEEP

This command causes the volume/device given in response to the VOLUME query to be acquired, its PAT to be read, and a bit is set so that it will not be freed as is normally done between commands, (but note the exception in the MTS command). Therefore, at the end of the execution of this command, VAMREC's knowledge & control of the disk volume is complete and up-to-date.

FREE

This command cancels the effect of a KEEP and will free the device/volume.

ASSUME ON
ASSUME OFF

If ON is given, no checking is done to see if a page in a READ or WRITE command really corresponds to a valid page on the device to the PAT byte for the page or the track address). If OFF is given, the checking is done (as is initially the case).

ZAP x c n
ZAP PAT x c n

The PAT parameter results in the PAT buffer being modified; else the current page is modified. The "x" is the displacement in the buffer, the "c" is the one byte fill character, and the "n" is the number of times the character is to be placed in the buffer.

VERIFY
VERIFY PAT
VERIFY DSCB

The PAT parameter or no parameter causes all bytes in the PAT buffer to be checked for illegal characters (this check is also done whenever the PAT is rewritten.) If an illegal character is found, its location in the PAT is displayed along with a query as to whether to continue or not. OK, YES, or ! Will cause continuation of the verification, anything else will not. The DSCB parameter causes all DSCB pages to be read and all the checksums of the DSCB's to be verified.

```
FORMAT CYLINDER cc
FORMAT GROUP x
FORMAT TRACK cc hh
DASDI CYLINDER cc
DASDI GROUP x
DASDI TRACK cc hh
FORMAT RETRY cc hh
```

The FORMAT RETRY option specifies a track on which a special record will be written that will be read by the READ RETRY command (2305, 7330, 3330 only). The PAT bytes for all pages on such a track must be X'C1'. The other six variants of the FORMAT/DASDI commands perform write-count-key-and-data operations on a given track, a group of tracks, or a full cylinder. The DASDI operation will rewrite the home addresses and record 0 as well as format the tracks(s). The definition of a group is device dependent --- however, a group is just a set of tracks and there is an integral number of groups per cylinder. For a 2314 a group consists of five tracks whereas on a 3330-type device, a group consists of one track. For the GROUP or CYLINDER option, all PAT bytes corresponding to pages within the GROUP or CYLINDER must be zero. No such checking is done for the TRACK option.

Some examples of the commands.

```
DISPLAY '100' 10
DISPLAY 256 'A'
DISPLAY LABEL
DISPLAY PAT '1000' '30'
MODIFY PAT '2000' '7F7F7F7F'
MODIFY LABEL 4 "TMTS02"
ZAP PAT 50 '41' 100
FORMAT GROUP 2
READ BUFFEREDLOG
KEEP
FORMAT RETRY '20' 0
READ RETRY '20' 0
```

Note: HELP or "?" Gives a list of valid commands. A command starting with a "$" is passed on to MTS. All commands can be abbreviated to the minimum number of characters required to distinguish them. The order of recognition is as printed by the HELP command.

Before any relocation is done, the KEEP command should be used. The FORGET--FREE command combination then results in MTS discovering the new state of the pack at the next non-VAMREC access of the pack.

VAMREC does not use the resident unit check routines. If a data check can be corrected, the error correction pattern is applied. If a data check cannot be corrected, the operation is retried five times. All write operations are read checked. There really is no retry on anything but 7330's, 3330's, or 2305's.

Classification: 1B0/3
Date: Jul 11, 1977
Doct=13 Vers=1

PURPOSE: To verify, trace, and/or dump the file system catalog. This program will verify, trace, and/or dump the entire catalog, a particular user catalog, or a particular user file.

AVAILABILITY: $RUN FILE:VNTD

HOW TO USE: The program accepts input on SCARDS if no PAR= field is given. An end-of-file terminates the program. SPRINT and SERCOM are used for output.

Parameters must be separated by blanks or commas and must be given in the following order:

(1.) any, all, or none of the following:
"V" - verify
"T" - trace
"D" - dump

(2.) at most, one of the following:
"C" - the entire catalog (the default)
"U" - a particular user catalog
"F" - a particular user file

If "U" is given, then a legal MTS userid must be the next parameter. If "F" is given, then a legal internal filename must be the next parameter. If "...,U,*ALL" is entered then all user catalogs are processed. If "...,F,*ALL,ID" is entered then all files corresponding to the specified user ID are processed.

EXAMPLES:
```
$RUN FILE:VNTD PAR=T,C
$RUN FILE:VNTD PAR=V,T,D,U,W045
$RUN FILE:VNTD PAR=D,F,W045FYLE
```

OUTPUT: VERIFY

Verifying the entire catalog will validate segment allocation as well as error checking record and segment headers. Presently the catalog can only be verified when segments are not being allocated or deallocated, ie: when no one else is using the system. Verifying the entire catalog will take approximately 5 minutes.

Verifying a user catalog will error check record

and segment headers as well as file descriptors. In addition, it will check that file descriptors point to sharing descriptors in the same catalog and that all sharing descriptors are accounted for.

Verifying a file checks file descriptors & sharing information for reasonableness.

TRACE

Tracing the catalog will print out the file header locations and the number of pages in each file.

Tracing a user catalog will print out the segment locations for each segment assigned to the user catalog.

Tracing a file will print out file and sharing descriptor locations.

DUMP

Dumping the catalog will dump (via SDUMP) each file header.

Dumping a user catalog will dump each segment assigned to the user catalog.

Dumping a file will dump the file and sharing descriptors associated with the file in the catalog.

TMTS - The MTS Testing Facility

I. Introduction

The TMTS facility is a means of testing out those parts of MTS which are a part of the MTS job program without disturbing the normal operation of the system (in theory). This is accomplished by providing a new job program, called TMTS (as opposed to MTS), which can be invoked with certain devices as the *MSOURCE*. The TMTS job program is loaded into shared VM just as the normal MTS job program is, and a TMTS task is started by UMMPS just like an MTS task is. So long as the TMTS task only executes code which belongs to itself, it will be completely independent of the regular production version of MTS. Likewise, so long as the regular MTS does not use any TMTS code it will be unaffected by the presence or absence of TMTS and any TMTS tasks.

A big advantage to using TMTS is that there is (almost) no degradation in performance over the regular MTS, like there is with the virtual machine. The drawback, of course, is that only MTS and related pieces can be tested using TMTS, excluding such components as the supervisor, HASP, and the PDP.

II. Loading TMTS

The TMTS job program is loaded into the shared segments (0-4) which are common to all users' address spaces. Therefore, TMTS can be addressed by any normal MTS tasks just like APL, PL1LIB, and any of the other things that reside in shared VM. A normal MTS task should never reference any part of TMTS. A normal MTS task will not have any adcons pointing into TMTS, so, in the normal course of events, an MTS task will not invoke any part of TMTS.

Conversely, a TMTS task should never reference any part of the normal MTS. Again, TMTS will not have any adcons pointing into MTS, so it will not normally invoke part of MTS. Things are not so black and white as they might appear, however. There is a large gray area of non-MTS components which can potentially be used by TMTS as well as MTS without any interference. These pieces are primarily CLSs, but also include the file routines.

The requirement for any component to be sharable between MTS and TMTS is that it be independent of any routine in either. Primarily, this means that when it is loaded (by PISTLE), it has no external references resolved from the resident version of MTS. Note that CLSs are intended to satisfy this requirement. It also means that any subroutines which the component does call (by means of a transfer vector entry), must be compatible, i.e., require the same arguments and return the results in the same form. Depending on how different the TMTS job program is from the MTS job program, this may or may not be satisfied for a CLS.

Note that the file system satisfies the first requirement above since any subroutines which are called from the file

routines are provided by the file routine transfer vector - there are no assembled-in adcons. The file routines are, however, themselves called as subroutines from both MTS and TMTS, and therefore the TMTS job program must supply parameters in exactly the same form as MTS does in order to use the resident file routines. Note that this means, among other things, that the format of the FCB must be the same in both MTS and TMTS. If any part of this interface has been changed - e.g., the format of the FCB is changed in TMTS - then a separate copy of the file routines must be loaded along with the TMTS job program. Thus, new file routines can also be tested using TMTS along with a new MTS.

The actual mechanics of loading TMTS are as follows.

1. Prepare a TMTS object deck (usually with RAMROD)
2. $RUN LOADMTS PROT=OFF SPRINT=mapfile PAR=NOTEST
    a. respond "OK" if the loading address printed is alright, or give a new address in hex.
    b. supply the file name containing the TMTS object deck. If it loads successfully, you're in business. LOADMTS is very similar to PISTLE with a few differences:
        1. It acquires the loading address from ENDSEG2 just like PISTLE, but it does not update ENDSEG2 when it is finished. Therefore, if it is run twice in succession, the second TMTS will be loaded on top of the first.
        2. It puts the entry address into the supervisor's job table at the TMTS job entry. Thus, the supervisor knows where TMTS jobs get invoked.
        3. LOADMTS provides a two entry initial ESD list. One entry is "MCSYMBOL" which has the address of the regular system's LCSYMBOL, and the other entry is "SYSDEFS" with the address of the regular system's SYSDEFS. "MCSYMBOL" is used instead of "LCSYMBOL" to avoid conflicts with the LCSYMBOL entry in the TMTS object module being loaded.
        4. It does not put anything into the normal LCSYMBOL.

To belabour the obvious, LOADMTS is run in a normal MTS task. The "MCSYMBOL" which it provides as an initial ESD list item is the normal MTS's LCSYMBOL. Thus, if the TMTS object module being loaded references any symbol which is not defined by the object module itself and for which there is an entry in LCSYMBOL, that symbol will be resolved from the resident LCSYMBOL. If that symbol is in the resident MTS job program, the independence of TMTS and MTS will be violated. It will also not be noticed by LOADMTS, and hence, probably not by you until the TMTS task goes south in some perhaps spectacular fashion. This situation most frequently occurs when the TMTS object module is incomplete for some reason.

In order to avoid such difficulties, it is advisable to use RAMROD to prepare the TMTS object module. A system called TESTMTS is set up in RAMROD[1] which has those pieces of the system needed for a TMTS object module. In general, these components are only those which are connected with the MTS job program in some intimate way. The supervisor, for example, is not in this system. The file routines are also not in this system. Those things which are in the TESTMTS system are as follows:

RIPCARDS - a set of "RIP" loader records followed by an LCS MCSYMBOL record. See below for more details.
MTS assembly decks - Those control sections which make up the MTS assembly: MTS, CMDS, FSUB, USUB, DSRS, DSRI, GSFS, INFO, PLIM, LLXU, TIMT, RSF, RNBR.
TGATE - the "gatekeeper" csect from the MTS assembly, with additions to LCSYMBOL. This interacts with the RIPCARDS deck. See below for more details.
DYS - the DYSSUB csect from the MTS assembly which normally goes in *DYSSUB.
COST[2] - the COST subroutine
CHARGE[2] - the CHARGE subroutine (used by COST, among others)
CFDUB[2] - the CFDUB subroutine
STDDMP[2] - the STDDMP routine
KWIC[2] - the keyword scanner
MSG[2] - the message printer
SYSDEFS[2] - the low core symbol table of system defined symbols.
GETRATES[2] - subroutine to read in RATEVEC from *RATEFILE
RATEVEC[2] - the rate vector built from *RATEFILE
FNAMETRT[2] - the file name translate table
LCS - an LCS MCSYMBOL record
ENDSEG2[2] - the 8 byte csect which contains the next location beyond itself.

The interaction between RIPCARDS and TGATE is what makes sharing CLSs, DSRs, and other things between MTS and TMTS tasks convenient. It is simply a means of copying entries from the standard MTS' LCSYMBOL (MCSYMBOL when being loaded by LOADMTS) to TMTS' LCSYMBOL. The LCSYMBOL for TMTS is defined by the TGATE, just like in the regular system (except the deck name there is GATE). However, in TGATE, the LCSYMBOL vector has several additional entries - one for each entry to be copied from the real system's LCSYMBOL. These entries consist of the symbol name

---

[1]This description is of the way things are at UM as of 5-1-77. the details of RAMROD usage may change from installation to installation.

[2]The same as the corresponding deck in the standard system

and a weak external reference for the symbol. The RIPCARDS deck has a RIP card for each of these sybmols. When the LCS MCSYMBOL card is encountered, all of those symbols are found in the real LCSYMBOL, and thus defined. When the weak external reference in the TMTS LCSYMBOL is encountered, the symbol definition is used to supply the address, thus copying the address of that symbol from the MTS LCSYMBOL to the TMTS LCSYMBOL. The extra symbols in TGATE are governed, during assembly, by the assembly parameter &TMTS. If this boolean set symbol is 1, the extra symbols are generated. It should be 1 when generating a TGATE and 0 when generating a GATE for the regular (non TMTS) system.

The DYSSUB component is an example of a component which is normally loaded by PISTLE which cannot be used by both MTS and TMTS. It is dependent on the MTS dsect (which is why it is part of the MTS assembly), so the results it gave would be wrong if the MTS dsect for the TMTS job program differed from the resident MTS - which is usually the case. It is loaded, therefore, with the TMTS object module. Note that the same effect could be achieved by loading DYSSUB by using PISTLE from the TMTS task.

The remaining decks (other than the MTS assembly decks) are all the same as their counterparts in the regular system object module (the "CURRENT" system in RAMROD). The reason they must be loaded with the TMTS object is that they contain references to symbols defined in the MTS assembly. Thus, if they were not included in TMTS, the ones in the resident system would be used, allowing control to pass to the resident MTS job program from a TMTS job program. When this happens, disaster is usually not far behind. These decks should be updated from the current system (using the RAMROD UPDATE command) before the TMTS object module is written.

As was mentioned above, once a TMTS task has been invoked, PISTLE can be used to load components into shared VM and put the appropriate entries into the TMTS LCSYMBOL. It is largely a matter of personal taste whether a component (like TIME, STDTV, or DYSSUB) is more conveniently loaded with the TMTS job program or is loaded via PISTLE afterwards.

III. Invoking TMTS

Starting a TMTS task, once TMTS has been loaded, is straightforward. It is started just like a normal MTS task is - the operator types the job program name, (TMTS) followed by the MSOURCE device - e.g., TMTS OPER. This is a reasonable way to start a TMTS task if the desired MSOURCE device is dedicated, like a 3270. (TMTS DS03 would start a TMTS task on "display station 03"). However, for devices like terminal controllers (the memorex 1270 and the data concentrator), it is undesirable to simply start up a line as a TMTS task since some unsuspecting user might dial it up. Furthermore, there is no way to guarantee that you, the tester, will get that task when you dial up. Using the Memorex 1270, there is no way out of this. With the data concentrator, the second problem can be solved and the first can

be minimized as follows:

1. Signon to one of the concentrators normally.
2. Go to the concentrator console and turn the selector switch to the concentrator you signed on to.
3. Issue the command PARAMETER TCB=LAnn ON=PASSWORD (PR TB=LAnn ON=PD) on the concentrator console. LAnn is the line adapter you are signed on to.
4. On the operator's console, pick an MTS task, Cyxx, which is currently idle and STOP it, where y is the concentrator designation (A, B, or C) and xx is the line number. If there are more than one idle tasks, pick one well up in the trunk hunting sequence to minimize the chance of some poor user getting it. If there is only one, perhaps you should wait till a less busy time.
5. Issue TMTS Cyxx on the operator's console.
6. From your terminal, issue the command
   (CTL-A)GRAB Cyxx
   You now have a TMTS task.

Using a 3270 device as MSOURCE for the TMTS task is much easier. In fact, it is not even necessary to use the operator's console. All that is necessary is to signon to the 3270 under MTS and issue the command %GRAB TMTS. The 3270 DSR will issue the appropriate STARTJOB SVC to the supervisor to start up the TMTS task. You can flip back and forth, then, between the TMTS task and the MTS task. This can be done because, though the 3270 DSR itself is not shared between the two tasks, the GRAB table (GRAB3270) is. There is a RIP card for this symbol and a weak external reference in TGATE for it, enabling the two tasks to share it.

TMTS can also be invoked in batch. By putting the keyword SYSTEM=TMTS on the signon card, HASP will invoke a TMTS task for that job.

IV. Testing the file routines in TMTS

If a new set of file routines is to be tested with TMTS, it is simply loaded along with the TMTS object deck (an easy way to do this is to give "filertns+MTS" to LOADMTS). There is a system in RAMROD called FILERTNS which contains the appropriate file routines for use with TMTS. These modules are: OPEN, CAT, RWSE, FLINE, READ, REDL, GETFINF, MOVE, WRIT, TRAK, and GETD (these are the RAMROD module names). Note that VOLGET and TABLRTN are not included, as the resident versions of these modules must be used. There can only be one version of VOLGET because the relocation table for error pages is kept in the VOLGET csect and is only built once - thus if there were two versions, only one would have a correct relocation table. There must be one and only one TABLRTN module because it is the mechanism by which the file routines are protected from destructive concurrent access of disk files.

In fact, new versions of both of VOLGET and TABLRTN can be

used if 1) a new copy of TABLES is included, and 2) a separate set of disk packs (with different volume serial numbers) is used by the TMTS task. If a separate copy of VOLGET is desired, only a new copy of TABLES is required - i.e. the normal system disk packs can be used.

When using a TMTS with its own copy of TABLES, the catalog pointers must be initialized before any file reference can be made. Since the process of starting up a TMTS task causes at least one file reference (to load the DSR), this initialization must be done after TMTS has been loaded but before it is invoked. This initialization is done as follows:

1. Load the TMTS job program, with the file routines. Make sure the file routines are loaded physically before the MTS csect.
2. Find the address of INITCAT in the TMTS file routines from the map produced by LOADMTS.
3. Find the address of FTV in the resident MTS.
4. Create a call (preferably using SDS) as follows:

```
L    GR1,=V(FTV)      from 3
L    GR15,=V(INITCAT) from 2
LA   GR13,SAVE        where SAVE is about 1000
                      bytes long
BALR GR14,GR15
```

   (Note: an easy way to do this is:

```
$DEB   *TIME
MOD   GR1 'addr of FTV'
MOD   GR15  'addr of INITCAT'
GOTO   $GR15          )
```

5. If the return code is zero, it worked.

V. Problems and Special Considerations

The biggest problem with TMTS is that the separation of MTS and TMTS is not automatic, and thus tends to be error-prone. This can happen if the TMTS task calls almost any non-CLS component in shared VM. This is why, as a rule, only CLS entry points are copied from the resident LCSYMBOL to the TMTS LCSYMBOL.

For example, if you want to run *IF in TMTS, the resident version cannot be used. Instead, you must run *IF+SEG2:IF. Consequently, programs run in TMTS often require more VM than normal, because they cannot use many resident components. The Fortran I/O library is another example.

In order to at least eliminate the burden of concatenating the resident module to object files, a special library, TMTS:TMTSLIB, has been constructed which contains things like the

Fortran I/O library. This library is used by setting *LIBRARY=OFF and LIBSRCH=TMTS:TMTSLIB.

A more severe problem, for which there is no ready solution, is that attention interrupts do not work the same way in TMTS as they do in MTS. This is because the routine in MTS which handles attentions (ZTPA) refuses to take an attention which comes from an address below the end of MTS. Since TMTS is loaded above all the CLSs, an attention will not be taken in a CLS until it attempts to do something where the attention-occurred flag is checked (like an I/O operation). For instance, when using the LOAD command in SYSTEMSTATUS, an attention to stop the command will not be taken until the next output line is written. While this behavior is annoying, it is seldom very serious. One of the places where it is serious is the case of the visual command in the editor. Since all the PF keys generate attention interrupts when in visual mode, if the resident version of the editor were used, visual mode would not respond to any PF keys. Therefore, a non-resident version of the editor must be used.

The final problem with TMTS which must be mentioned is its ephemeral nature. One of MTS's shortcomings is its lack of any effective way of controlling the use of shared memory. While it is possible to run PISTLE and load new or replacement components into shared VM without interference, PISTLE and LOADMTS do interfere. This is because PISTLE updates ENDSEG2 when it is finished while LOADMTS does not. Therefore, if LOADMTS is run twice in a row, the second TMTS will overlay the first. This is the desired behavior, since it is unlikely that more than one test MTS is useful at a time. The problem is that if a TMTS is loaded via LOADMTS, some subsequent use of PISTLE to load a new or replacement system component will overlay the TMTS. If you load TMTS and, at some later time, try to invoke it and get a "job header error" message on the operator's console, this has most likely happened.

Classification: 1E1.2/2
Date: Mar 28, 1978
Doct=30 Vers=1

Programming Instructions for Punch Unit Check Routines

A program which uses this routine must place the address of the appropriate routine in the returns list for an SVC STIO (SVC 2) as the unit check return. Also the calling program must load register 13 with the address of a scratch area before it executes an SVC WAIT (SVC 3) for an operation using one of these routines. This area must be 80 bytes long and it must be on a double word boundary. The scratch area may be used for any purpose except an I/O buffer by the calling program between WAIT'S. The first word of the scratch area must contain the address of a parameter area which has been set up in the format described below. This area must be unique for each device used and must be used for no other purpose. Before the first use of one of these routines the calling program must issue an SVC SNSADR to set up automatic sensing into the area provided in the parameter region and must store the device ID in the word provided for it.

Parameter Area for Punch Unit Check

Bytes	Name	Contents
0-3		Branch to alternate return for punch check if requested in USEFLG.
4	USEFLG	Byte giving options desired by calling program. The bits have the following meaning: BIT 0 (X'80'): Return on non-recoverable error. Bit 5 (X'04'): Is a 1442 punch rather than a 2540 punch. Bit 6 (X'02'): Return on punch check with no wait and no console message. BIT 7 (X'01'): Return on punch check after typing message and waiting for device end.
5	INTFLG	Must be set to zero before every SVC WAIT for this punch.
6	SNSFLG	Location specified in SVC SNSADR.
7	SENSE	Sense data stored by supervisor.
8-11	ID	Device id for this punch. This can be obtained with an SVC GETID.
12-15	SAVCMD	Address of a CCW which can be used to repunch the last card before the current one. This is ignored if either bit 6 or bit 7 of USEFLG is non-zero or if this is a 1442 punch. If this card image is not available this word should contain zero.

Classification: 1E2/2
Date: Mar 28, 1978
Doct=31 Vers=1

Programming Instructions for Printer Unit Check Routines

A program which uses this routine must place the address of the appropriate routine in the returns list for an SVC STIO (SVC 2) as the unit check return. Also the calling program must load register 13 with the address of a scratch area before it executes an SVC WAIT (SVC 3) for an operation using one of these routines. This area must be 80 bytes long and it must be on a double word boundary. The scratch area may be used for any purpose except an I/O buffer by the calling program between WAIT's. The first word of the scratch area must contain the address of a parameter area which has been set up in the format described below. This area must be unique for each device used and must be used for no other purpose. Before the first use of one of these routines the calling program must issue an SVC SNSADR to set up automatic sensing into the area provided in the parameter region and must store the device ID in the word provided for it.

Parameter Area for Print Unit Check

Bytes	Name	Contents
0-3	RETBC	Branch to alternate return point if any alternate returns are selected in USEFLG.
4	USEFLG	Indicates options desired by calling program. The bits have the following meaning: Bit 3: If 1 a UCS operation is being performed. Bit 4: Return to RETBC if channel 12 in the carriage tape is sensed. The condition code is set to 3 for this return. Bit 5: Return to RETBC if channel 9 is sensed in the carriage tape. The condition code is set to 1 for this return. Bit 6: Return to RETBC if a print check occurs. This return is made after a message is typed for the operator. The condition code is set to 0 for this return. Bit 7: If 1 an invalid character check is ignored. If 0 it is a fatal error.
5	INTFLG	Must be set to zero before each SVC WAIT for this printer.
6	SNSFLG	Location specified in an SVC SNSADR before first call to this routine.
7	SENSE	Sense data stored by supervisor.
8-11	ID	Device id for this printer which may be obtained with an SVC GETID.

Classification: 1E4/2
Date: Mar 28, 1978
Doct=32 Vers=1

Programming Instructions for 2311,2314,2301 DASD Unit Check Routines

A program which uses this routine must place the address of the appropriate routine in the returns list for an SVC STIO (SVC 2) as the unit check return. Also the calling program must load register 13 with the address of a scratch area before it executes an SVC WAIT (SVC 3) for an operation using one of these routines. This area must be 100 bytes long and it must be on a double word boundary. The scratch area may be used for any purpose except an I/O buffer by the calling program between WAIT'S. The first word of the scratch area must contain the address of a parameter area which has been set up in the format described below. This area must be unique for each device used and must be used for no other purpose. Before the first use of one of these routines the calling program must issue an SVC SNSADR to set up automatic sensing into the area provided in the parameter region and must store the device ID in the word provided for it.

Parameter Area for Disk Unit Check(2311, 2301, 2314 only)

Bytes	Name	Contents
0-3	RETBC	Branch to alternate return if any are specified
4-7	SEEKAD	CCHH from last seek address for this disk. If the currnet command is a seek this field should contain the seek address for this seek. This field may be changed by the unit check routine.
8	USEFLG	Indicates options selected by calling program. The bits have the following meaning: Bit 1 & Bit 2: 00=2311 00=2301 10=2314 Bit 3: Return on fatal error (cond. code 3) Bit 4: One if the command list is only a seek. Zero if the command list contains no seek. These are the only allowed types of command list. Bit 5: Return on track overflow. The condition code will be 2. Bit 6: Return on end of cylinder. The condition code will be 1. Bit 7: Return on no record found. The condition code will be 0.
9-10	INTFLG	Byte 9 only must be set to zero before each SVC WAIT for this disk.
11	SNSFLG	Location specified in an SVC SNSADR before the first call to this routine.
12-17	SENSE	Sense data stored by supervisor.
18-23	VOLSER	Volume serial number of disk pack mounted on this drive. If this is not known byte 18 should contain X'FF'.
24-27	ID	Device id for this disk.

Classification: 1F1/2
Date: Aug 21, 1976
Doct=12 Vers=1

Memo to: Programming Staff

From: George Helffrich

Subject: New keyword scanner

Date: Aug 21, 1976

Changes are marked with revision bars. Familiarization with the keyword scanner's (strange) conventions is assumed.

Calling sequences:

```
          LH    0,lhtlen
          LA    1,lht
          LA    2,ext
          LA    SCA,string
          CALL  KWIC

          CALL  KEYWRD,(lhtlen,lht,ext,string,rht)

          CALL  KWSCAN,(lhtlen,lht,ext,string,rht,stringlen,
                        options,rvec,dlist,slist)
```

lhtlen - halfword length of the left-hand table
lht - left hand table
ext - execute table
string - text to be scanned for keywords
rht - right-hand table
stringlen- halfword length of text to be scanned for keywords
options- fullword of options bits
rvec - 27 fullword return vector, or zero
dlist - list of characters and contexts which are to be considered as delimiting keyword expressions.
slist - list of character strings considered to be separators of keyword expression right-hand and left-hand sides.

Parameter Descriptions:

lhtlen - halfword length of left-hand table. Note that left hand tables do not need to be terminated with X'FF' bytes (they never did) so if this length includes one, strange things may happen.

lht - a series of entries describing left-hand side

texts. They are in the form:

1 (or 2) bytes - right-hand table index,
1 (or 2) bytes - execute table index,
1 byte - number of characters in the left-hand side text,
n bytes - left-hand side text.

The right hand table index and execute index values are two bytes in length if bit 27 of the options word is one. The number of characters comprising the left-hand side text may be zero, implying a null left-hand side.

ext - a table of instructions selectively executed depending on the left-hand and right-hand table entries which matched the keyword expression. The execute indices in the matching left-hand and right-hand table entries are added to the execute table address (ext) and the instruction presumed to be at that address is executed.
If the object instruction is a BAL or BALR instruction branching to a series of instructions, a return to 0 bytes past the contents of the link register causes the keyword match to be accepted. If the return is to 2 bytes past it, the keyword match is rejected, and scanning of the keyword tables for another match is continued. If the return is to 16 bytes past it, the keyword scanner terminates processing immediately with a return code of 4.
At the time an entry from the execute table is executed, all the registers are restored to their values at the time of the subroutine call, save GR3, which contains a value indicating which left-hand side matched the current keyword in the form of 4*(ordinal position of the matching left-hand table entry, starting from zero), GR15, which contains the address of the instruction being executed, GRS 1 & 2, which may contain information peculiar to the matching right-hand side type, possibly FRS 0 & 1 (see the right-hand side type Floating Point Number, below), and possibly GRS 4 and 5 (see parameter words dlist and slist, respectively.) The KWIC routine uses SCC, GR5, SCB & SCD, and FRS 0 & 1, respectively, for these purposes; the options demanding the usage of GRS 4 & 5 are not available to it.

string - the string to be scanned for keywords. If the KWIC or KEYWRD entries are used, it must be terminated by a blank.

rht - the right-hand table, which contains right-hand side descriptions and control entries affecting the scanning of the right- and left-hand tables.

The formats and effects of the control entries are as follows:

1 byte X'FF' - end of the right-hand table. If no keyword match has been made prior to its encounter, an error condition arises.

1 byte X'FE' - abort the scan of the right-hand table and search for an alternative match to the left-hand side of the keyword expression in the left-hand table. The scan of the left-hand table proceeds from the point at which the previous left-hand side match was made.

1 byte X'FD' - try to process the current keyword expression's right-hand side as a parenthesized list of right-hand sides, each associated with the expression's left-hand side (e.g., INFO=(SIZE,TYPE) would be processed as if INFO=SIZE,INFO=TYPE had been given.) The parenthesized right-hand side processing is performed only if: (a) the right-hand side text begins with a '(', (b) its length is longer than 2 characters, and (c) the keyword expression has a left-hand side.

1 byte X'FC',

1 byte # bytes following,

n bytes separator indicies - filter out unwanted left-hand and right-hand side combinations on the basis of which string separates (or, from another point of view, connects) the expression's right-hand and left-hand halves. Used in conjunction with the slist parameter word and bits 20-21 of the options parameter word, this entry is used to filter out nonsensical right-hand and left-hand table connections across a separator. The separator indicies are each one byte, indicating the separator's ordinal position in the list given by the slist parameter, or implied by options bits 20-21, with the value 0 indicating no separator (viz., degenerate right- or left-hand side.) If the separator in the keyword expression is not among those listed, the keyword expression is considered in error.

The formats of the right-hand side description entries follow the consistent format

1 byte - type code
1 byte - execute index
1 byte - # bytes following
n bytes - variable information, dependent upon type code.

The defined type codes and the right-hand table entry variable information peculiar to each is detailed below. Accompanying each right-hand side type is the description of the information returned in the registers if the right-hand side text matches the given right-hand table entry. [For the KWIC entry, the registers used to return the information are in parentheses.]

code 1 - Literal. the n bytes contain n characters of a string which must match the expression's right-hand side. GR1 (SCD) contains the IBM length of the string, and GR2 (SCB) contains its address in the keyword expression.

code 2 - FDName. The n bytes may be omitted, in which case the text is taken as a FDName. If 1 byte, the character 'N', is given in the table entry, the FDName cannot specify explicit concatenation. GR2 (SCD) contains a FDUB pointer for the file or device.

code 3 - Characters. The right-hand side can be an arbitrary string of characters, possibly limited in length by information given in the table entry. If 1 byte follows, it is taken as the maximum permissible length of the string. If 2 bytes follow, they are taken as the minimum and maximum permissible lengths of the string. If no bytes follow, the string may be of any length. The lengths all refer to the real length of the string, not the IBM length. Execute registers are set up as for literal (code 1) type.

code 4 - MTS Line Number. The right-hand side can be an MTS line number (signed, 6 integral digits, 3 fractional digits) followed by an optional scale factor character. The permissible scale factors, their scaling values, range limits, and other scaling operations can be described in the variable information following the table entry. These are specified in groups of 5 bytes, the first byte of which is an

operation code, the remaining being an integer (no alignment required) operand value.

operator '>' - the right-hand side value is compared to the operand value. The right-hand side value may not be smaller, or no right-hand side match is performed.

operator '<' - the right-hand side value is compared to the operand value. The right-hand side value may not be greater, or no right-hand side match is performed.

operator '*' - the right-hand side value is multiplied by the operand value.

operator '/' - the right-hand side value is divided by the operand value.

any other operator - the operator character is interpreted as an optional scale factor, which if found at the end of the expression's right-hand side, causes the right-hand side value to be multiplied by the operand value.

The right-hand side value is multiplied by 1000 to shift any fractional digits into the integral range before any of the above operations are applied. If a match is made, GR2 (SCD) contains the value after application of the pertinent operations above.

code 5 - Hex Number. The right-hand side value is considered an 8-digit (maximum) hex number. No further right-hand table information is needed. The number, padded to the left with zeros to 8 digits, is left in GR2 (SCD). GR1 (SCB) points to the first character after the rightmost hex digit in the input string.

code 6 - Initial Substring Literal. The n bytes contain literal text, which must be an initial substring of the right-hand side text for a match to be made. Any excess characters are not considered in the match. Execute registers are as for literal (code 1).

code 7 - No Right-Hand Side. No extra bytes to the right-hand table entry need be given. For a match to take place, a right-hand side may not be given in the

keyword expression -- the expression consists solely of the left-hand side text. No special information is returned in the registers.

code 8 - Ignore Keyword. The keyword text is ignored, and no execute code is performed.

code 9 - Characters in Given List. The variable information is interpreted as a minimum and maximum length bound on the right-hand side text, followed by a set of characters which must constitute the text of the right-hand side (no ordering presumed.) Registers are set up as for literal (code 1) type.

code 10 - Characters Except in Given List. The n bytes are interpreted as a minimum and maximum length bound on the right-hand side text, followed by a set of characters, any one of which appearing in the right-hand side text will cause a failure to match the right-hand side. Registers are set up as for literal (code 1).

code 11 - Optionally Negated Characters. The right-hand side may be an arbitrary character string possibly bounded in length and optionally preceded by one of the negating prefixes 'NO', 'N', '¬', or '-'. One to three bytes must be given in the right-hand table; the first (required) byte is taken as the index into the execute table added to the index in the matching left-hand table entry if a negating prefix was found in the right-hand side text; if no such prefix was encountered, the standard right-hand table execute index is used. One or two bytes more may follow, interpreted as the optional two bytes in the characters (code 3) right-hand side type. The registers are set up as for characters, except that any negating prefix, if encountered, is not indicated. All counts refer to the length of the string not including the negating prefix, if present.

code 12 - Optionally Negated Literal. The right-hand side text is taken as a literal string which may be preceded by one of the negating prefixes 'NO', 'N', '¬', or '-'. The n bytes contain a 1 byte execute index in the case the negated form of the literal is detected, as defined in the type above, followed by the text of the literal string which

must match the expression's right-hand side text. The registers are set up as for literal (code 1), except that any negating prefix, if encountered, is not indicated.

code 13 - Optionally Negated Initial Substring Literal. The n bytes of the right-hand table entry contain a 1 byte execute index used, as above, in the case of a negating prefix on the keyword expression right-hand side text is encountered, followed by the character of a literal string which must be an initial substring of the keyword right-hand side. The execute registers are set up as for initial substring literal (code 6).

code 14 - Delimited Character String. The keyword expression's right-hand side is taken as a character string, bounded by any of a set of string delimiting characters defined in the right-hand table entry. The n bytes contain at least 2 bytes interpreted as the minimum and maximum possible lengths of the string, followed by a series of single characters, one of which must initiate and terminate the text of the expression's right-hand side. If no characters are given, the string's first character is taken as the string delimiter. Doubled instances of the string delimiter are collapsed into a single character. The input string is not altered by this operation, but the resulting length of the string, if doubled delimiters are found, may not be greater than 128 characters. The execute registers are set up as per characters (code 3), but the initial and final string delimiters are not included in the information.

code 15 - Integer Number. The expression's right-hand side text is interpreted as an integer number (signed, 9 digits max.) optionally followed by a scale factor. The format of the n bytes following the right-hand table entry is identical to that of the MTS line number type (code 4), as are the contents of the registers at execute time. However, the number is not multiplied by 1000, and note no fractional digits are permitted.

code 16 - Flagged Hex Number. The expression's right-hand side text is interpreted as

a hex number (8 digits max.) enclosed in X'...' . The contents of GR2 (SCD) contains the hex number, right-justified, at execute time.

code 17 - Floating Point Number. The expression's right-hand side value is taken as a FORTRAN-style long-real number, optionally followed by a scale factor as with types integer and MTS line number. The format of the information following the right-hand table entry is similar to that of the aforementioned types 4 and 15, differing only in that the operand values are long-real values (again, no alignment necessary) and occupy 8 bytes of storage. The resultant value is left in FR0-1 at execute code time.

code 18 - PAR Field. The right-hand side of the keyword expression is taken as the remainder of the input string interpreted as a character string. GR1(SCB) contains the IBM length of the character string and GR2(SCD) contains the address of the first character of the string at execute time.

code 19 - Literal Table. The keyword right-hand side must appear in a given table of fixed length literals. If the right-hand text is longer than a table entry, the initial substring of the right-hand side of length of a table entry is searched for in the table. The information contained in the right-hand table entry is a 4 byte unaligned table address. The format of the table is: 1 byte - length of a table entry; 1 byte - number of entries in the table; n bytes of table entries, short ones being padded with blanks. At match time, GR1(SCB) contains a 1-origin index into the table of the matching entry, and GR2(SCD) contains its address in the table.

stringlen- halfword length of the text to be processed for keyword expressions.

options - a fullword of options bits, their locations and effects given below:

bit 15 - the ext parameter indicates the address of a subroutine to call to perform the functions of the execute code. Rather than a single instruction being executed at the time a keyword expression is matched, this subroutine

is called with the following parameters:

1 word - sum of left- and right-hand table execute indices,
1 word - contents of GR1 as defined previously,
1 word - either the contents of GR2 if it was not an address, or the address of an array containing the information indicated by GR2 if it was an address (viz., no extra level of indirection),
1 word - contents of GR3 as defined previously,
1 word - contents of GR4 as defined previously,
1 word - contents of GR5 as defined previously.

If a return code of 0 is given by this subroutine, the keyword match is accepted. RC of 4 causes match to be rejected and the scan for an alternate match to be continued. RC of 8 causes keyword processing to be aborted immediately.

bits 16-17 - perform spelling correction on unmatched left-hand sides. If bits 16-17 are:

00 - no correction attempted;
01 - correct, and print warning message;
11 - correct, print warning message, and request confirmation of the correction from the user, if in conversational mode. If in batch mode, no correction is performed.

bit 18 - use alternate return vector format. See the rvec parameter word for further details.

bit 19 - keyword expression left-hand sides may be parenthesized. This option enables keyword expressions like (ENDFILE,SIGFILE)=OFF to be considered equivalent to ENDFILE=OFF,SIGFILE=OFF.

bits 20-21 - define separator list. This two-bit switch is used to indicate that, rather than '=' connecting a keyword left-hand side with a right-hand side, either an explicitly specified set can be defined via the slist parameter, or a standard predefined set may be used. The predefined set is a relational separator set, the strings '>=', '<=',

'¬=', '>', '<', and '=' being the valid separators.
See the slist parameter word for more details.
If bits 20-21 are:
00 - standard '=';
01 - relational set;
11 - user-defined set, passed as slist.

bit 22 - explicit delimiter character list given. If this bit is zero, only blanks and commas not embedded in parentheses are taken as delimiter characters. However, if this bit is set, a user-defined set can be passed as the dlist parameter. See the dlist description for details.

bit 23 - left-hand sides can be given as initial substrings of the left-hand side texts given in the left-hand table entries. If this option is selected, the order in which the left-hand sides are given in the left-hand table is important...

bit 24 - return pointer to end of scanned keyword text in parameter list word string.

bit 25 - reserved for future use, should be zero.

bit 26 - convert all input to upper case before processing. The input text string is not molested if this option is selected, but a copy is made and the case translation and subsequent processing is performed on the copy.

bit 27 - in left-hand table, the right-hand table and execute table indicies are 2 bytes in length. If right-hand tables become too large, this can solve the problem...

bit 28 - return to caller upon encountering any bad keyword expressions.

bit 29 - prompt the user for new keywords if a bad keyword expression is encountered.

bit 30 - print any error comments concerning bad keyword expressions. If this option is not selected, the error comment is returned in the rvec return vector. See rvec for details.

bit 31 - scan multiple keyword expressions. If this option is not selected, only a single expression is processed, even if either of the right-hand or left-hand sides were parenthesized.

rvec - the location of a 27 fullword return vector, or an address of zero, indicating no return vector. The return vector contains information pertaining to errors encountered during the keyword scan, in one two possible formats (for the sake of compatibility.)

If bit 18 of the options word is zero, the return vector will only contain information in the event that an invalid keyword expression is encountered. In this case, the return vector information is:

word 1 - address of first character of the invalid keyword expression,

word 2 - fullword length of the error comment following,

words 3-27 - text of the error comment.

If bit 18 of the options parameter word is one, the return vector will contain information describing every case in which the subroutine may return in error. Whenever a return code of 4 is given, the vector will contain error information. The format of the return vector is:

word 1 - fullword error code,

words 2-27 - variable, dependent upon the error code number.

The currently defined codes and their associated variable information is as follows:

code 1 - 'CANCEL' given in response to a prompt for error replacement input. No further information is returned.

code 2 - invalid keyword expression. This error code is only issued when options bit 30 is zero. The variable information contains:

1 word - address of bad expression.
1 word - length of bad expression.
1 word - length of following error text.
23 words - error text.

code 3 - keyword processing prematurely terminated by execute code. If the execute code has exercised the option of returning to 16 bytes past the contents of the link register of the branch-and-linkage instruction, this error code is given. No further information is returned.

code 10 - invalid right-hand side type code in the right-hand table. If this error code is issued, the address of he invalid type code is left in the second word of the return vector.

code 11 - invalid format for right-hand table entry. The second word of the return vector contains the address of the improperly formatted entry.

code 12 - invalid separator list format. The second word of the return vector contains the address of the improperly formatted entry in the separator list.

code 20 - non-MTS task keyword expression error. No further information is returned.

code 21 - non-MTS task requested FDUB. The second word of the return vector contains the address of the right-hand table entry requesting the FDUB.
code 30 - unable to get psect from GPSECT routines. No further information is returned.
code 31 - auxiliary subroutine unavailable. A non-MTS segment 2 routine was needed, but its Vcon was not defined. No further information returned.

dlist - address of a list of single characters and associated contexts in which the characters are to be considered delimiters of keyword expressions in the input string. If options bit 22 is zero, no list need be specified, and the delimiter characters are considered to be blanks in all contexts, and commas not nested within parentheses. If the bit is one, a list of characters and contexts are assumed to be present, and in the following format:

1 byte - # of delimiter characters to be defined
(1 byte - delimiter character,
1 byte - context indicator: 0 for balanced parenthesis context, 1 for any context)
-- repeated for every delimiter character to be defined.

If this option is used, at the time the executed code is performed, GR4 is set up to point to the delimiter character in the keyword string.

slist - address of a list of strings to be considered as separators of left-hand and right-hand sides in the keyword expression. If options bits 20 & 21 are ones, this list must be present, and in the following format:

1 byte - # of separators in the list,
(1 byte - length of separator string,
n bytes - text of separator string) -- repeated for each separator in the list.

If options bits 20 & 21 are B'01', then this list need not be given, but the subroutine acts as though a list specifying the separators

'>=', '<=', '¬=', '>', '<', '=',

in the presented order, had been passed via this parameter.
If a non-standard separator list has been passed or implied, the right-hand table control entry type X'FC' refers to the separator's ordinal position in the (possibly implied) list, with the value zero indicating no separator text given (viz., either the right-hand or left-hand side is

degenerate.) Also, at execute code execution time, GR5 is set to contain the abovementioned value times 4 to signify which separator connects the keyword right-hand and left-hand sides.

Return Codes:

0 - all ok, keywords possibly processed. (If the options bits have specified that keyword expression errors are to be ignored, a keyword expression error could possibly have occurred, but a subsequent correct keyword expression could clear the error condition and result in RC=0.)

4 - some type of error has occurred, or 'CANCEL' was given in response to a prompt for error replacement input. If the options bits have specified that the rvec should return information, then if the alternate return vector format has been specified, the error condition is clearly defined in it. Otherwise, RC=4 is a catchall return code indication some sort of error.

The MTS to SDS Interface

The MTS to SDS interface is a special case of the general MTS - CLS interface. A CLS is allowed to "monitor" the EXEC (or USER) CLS by use of its return codes to MTS and a byte of switches in the CLSPARM region. The CLSPARM region is passed as parameters to the CLS upon invocation, and the CLS can set various switches to control, somewhat, what kinds of events will cause a suspension of the EXEC CLS and a subsequent return to the monitoring CLS. When the "monitor" CLS returns to MTS with a return code of CLSRCST (12), MTS will invoke the EXEC CLS and will call it again when the EXEC CLS returns, or when one of the specified events occurs in the EXEC CLS.

This "monitor" facility is available to any CLS, but is only used by SDS (which, of course, was its sole motivation). It is implemented within MTS by just a few subroutines which are called from all of the places that an EXEC CLS can return to in MTS, and from most of the places in MTS where significant events in the life of an EXEC CLS are processed.

These subroutines are as follows:

EXGO - Start the EXEC CLS.

CLSSTRT - Starts the EXEC CLS and sets the "start byte." The start byte (CLSTRTB) is set by moving the switches at CLSLT+3 (the load type word) in the CLSAREA of the current CLS to the start byte, and turning on the "start switch" (CLSTRTSW). The EXEC CLS is started by simply branching to EXGO. When SDS returns with RC=12, this is called from CLSRTNLOC (in CMDS).

CLSSTST - Tests an event to see if
1) there is a monitoring CLS around,
2) if there is, it wanted events of this type.

These conditions are determined by testing the start byte against the code representing this event. The event type is passed as a parameter in GR0. Most events, like EXEC returns, require only that there be a calling CLS around to cause the test to be satisfied. If this event is wanted by the monitoring CLS, then the monitoring CLS is invoked by the following sequence:
1) call RUNTOF to switch timer interrupts
2) turn off the start switch
3) set up the CLSAREA for the monitoring CLS with GRS for CLS entry and the code of this event as one of the parameters
4) invoke the monitoring CLS by branching to GO

CLSSTST2 - Does the same as CLSSTST but does not distinguish between events. If there is a calling CLS around, it is invoked with the same sequence as CLSSTST.

Classification: 321.1/1
Date: July 25, 1977
Doct=25 Vers=1

THE INTERNAL STRUCTURE OF LINE FILES IN MTS

A line file has two basic components, the line-hole directory and the data section. Logically, the line directory is an array of 8-byte entries, one for each line in the file; each entry contains the line number, plus a pointer (relative page number and offset) into the data section, where further information about the line is stored. This array is ordered from smallest to largest line number, which makes both sequential operations and indexed operations relatively efficient. The format of a line directory entry is:

```
LDELNUM BYTES 0-3:      line number
LDEPAG# BYTES 4-5:      relative page number (1-32767)
LDEDISP BYTES 6-7:      displacement within the page (0-4095)
```

The contents of the data section are unordered, and are allocated and freed dynamically, using a hole directory. The hole directory is an array of four byte entries; there is one entry for each page of the data section, giving its relative page number, preceded by an entry for each contiguous block of available space on that page, giving its offset and length. Each such group of entries is ordered from highest to lowest offset, to facilitate recombination of available blocks. No particular order is imposed on the groups themselves, however. The relative page number entry appears last in each group because the hole directory is scanned in reverse order. The format of a hole directory page number entry is:

```
HDEPAG# BYTES 0-1:      relative page number (1-32767)
HDEFLAG BYTES 2-3:      zero, which flags this type of entry
```

and the format of an available block entry is:

```
HDEDISP BYTES 0-1:      displacement to beginning of block (0-4095)
HDELEN  BYTES 2-3:      length of available block. (1-4095)
```

We next describe the manner in which these pieces are mapped onto the physical pages of the file. The line directory array is divided into blocks of 510 or fewer entries; each such block is stored on a separate page, and these pages are chained together on a two-way linked list, in increasing line number order. The first page in the chain is always page one of the file. The hole directory follows the line directory in the same chain; only one page may contain both hole directory and line directory entries at the point where they join.

The data section normally occupies a set of pages distinct from the line-hole directory chain. If the file is small enough to be stored in one page, however, the data section occupies part of page one.

The following is the general format of a line or hole directory page:

PHLDSO	BYTES 0-1:	offset to start of line-hole directory; this will be either X'0028' or X'0BE0' in page one, and X'0010' in all other pages
PHLDL	BYTES 2-3:	line directory length (bytes) (0-4080)
PHHDL	BYTES 4-5:	hole directory length (bytes) (0-4080)
PHSID	BYTES 6-7:	relative page number of this page (1-32767)
PHFWDP	BYTES 8-9:	forward pointer (relative page number of next page in hole-line directory chain) (0-32767 0=end of list)
PHBWDP	BYTES 10-11:	backward pointer (0-32767 0=end of list)
PHLNTP	BYTES 12-13:	line number table index (see later) 8-byte-slot number in line number table (0-8180)
	BYTES 14-39:	global file information - page one only - in other pages the line or hole directory starts at byte 16, and bytes 14-15 are unused.
	BYTES 40-3039:	data section for a one page file. In larger files the line directory starts at byte 40 of page one. There is room here for 3000 bytes of data.
	BYTES 3040-4095:	line directory starts here in a one page file. Room here for 131 lines and two hole entries.

We next describe the contents of the data section of the file. For lines shorter than 128 bytes, and many longer lines as well, the line occupies a contiguous block of storage in the data section of the file; the first two bytes of the block give the length of the line, and the remainder is the line itself. A line longer than 128 bytes may be broken into at most 16 pieces, none of which (except possibly the last) may be shorter than 128 bytes; clearly none will be larger than a page. If the line is broken up, the block pointed to by the line directory entry contains a table of pointers and lengths for the remaining blocks, followed by the first piece of the line. The structure of the line block table is as follows:

LINBKTB BYTES 0-1:	BITS 0-3 - number of pieces minus 1 (0-15) BITS 4-15 - length of first piece - does not include length of table (1-4094)
LINPAG# BYTES 2-3:	relative page number (1-32767)
LINDISP BYTES 4-5:	offset (0-4095)
LINPLEN BYTES 6-7:	length of this piece (1-4096)
BYTES 8...	up to 14 6-byte entries in the format of bytes 2-7, one for each piece, followed by the first piece of the line.

There are no alignment restrictions on blocks in the data section. The data blocks for a line may have a total length up to 32767 bytes. The data blocks for the line number table (described later) may have a total length up to 65444 (4096*16-2-15*6), leaving room for 8180 8-byte slots.

If a line or hole directory page becomes empty, normally because all the lines it points to have been deleted, it is removed from the line directory chain and added to the free page chain, which is a one-way linked list of available pages, chained through the normal forward pointer field. The pointer to the first such page, if any, is contained in the global file information in page one of the file. Pages on the free page chain can be used either as data pages or line directory pages. Once a page has been used as a data page, however, it will never be used as a line directory page. Pages beyond the number of pages in use (R1NPGS) are not chained.

To improve the efficiency of indexed operations on line files, a line number table is added to the file. The line number table is an array containing a one-way linked list of 8 byte entries, one for each line or hole directory page, with the following structure:

LTELNUM BYTES 0-3:	Line number of first line in page
LTENEXT BYTES 4-5:	Index of next entry in list (0-8180 0=end of list)
LTEPAG# BYTES 6-7:	Relative page number of corresponding page 1-32767

These entries are chained in exactly the same order as the corresponding line or hole directory pages. The pointers are entry indices, and page one always has index zero. If a line directory page contains neither lines nor holes (a condition which should only occur for page one), the line number table entry contains X'80000000'. If a page contains only holes, the entry contains X'7FFFFFFF'. Recall that each line directory page also contains the index of the corresponding line number table entry.

The line number table is stored in the data section of the file, in exactly the same format as a normal data line. The pointer to this "line" is part of the global file information in page one. Corresponding to the free page chain for line directory pages, there is a free entry chain for the line number table, chained through bytes 4-5, as usual, and starting from the global file information.

With the exception of the global file information, whose format is given below, this completes the description of the internal structure of line files.

Global file information:

R1FTYPE	BYTE 14:	file type - always X'00'
R1HDRL	BYTE 15:	header length - always X'28'=FL1'40'=AL1(LNEFHDRL)
R1NPGS	BYTES 16-17:	number of pages in use (truncated size) (1-32767)
R1NLDR	BYTES 18-19:	number of line-hole directory pages (1-8180)
R1NAB	BYTES 20-23:	number of available bytes in line-hole directory
R1MFS	BYTES 24-25:	maximum file size - file will not be expanded beyond this size (1-32767)
R1MLL	BYTES 26-27:	maximum line length - length of the longest line written (0-32767)
R1LLDR	BYTES 28-29:	last line-hole directory page number (1-32767)
R1FPC	BYTES 30-31:	free page chain pointer (0-32767 0=none)
R1LNTP	BYTES 32-35:	line number table pointer, 2-byte page number (1-32767) & 2-byte offset (0-4095)
R1LNTFL	BYTES 36-37:	line number table free entry list (0-8180, 0=NONE)
	BYTES 38-39:	unused.

Classification: 321.2/1
Date: July 25, 1977
Doct=26 Vers=1

Internal Structure of Sequential Files in MTS

The organization of sequential files (with or without line numbers) is quite simple when compared to line files. In general, the first "n" bytes of the first physical record is used as a header by the sequential file routines in which pertinent information about the sequential file is retained.[1]

Immediately following this header information are the lines of information stored in the sequence in which they were received by the sequential file routines. Since these lines may be up to 32,767 bytes long, and since the physical records on the disk are 4096 bytes (1 page) long,it is quite possible that a line will have to be broken up and stored on more than one physical record. This is quite likely even if only "short" lines are written into a sequential file since the lines are packed end to end using up all of one physical record before going onto the next physical record. Thus, it turns out that even short lines may be broken up across physical record boundaries.

For this reason, it is convenient to refer to a segment of a line as that part of the line which resides on a physical record. Furthermore, we can refer to the first, intermediate, and last segments of a line, remembering that in fact these descriptions may all denote one segment (identical to the line) or they may denote two or more distinct segments, depending on the size of the line and how the line "fell" with respect to physical record boundaries.

The first 4 bytes in the sequential file header are the length of the header, following this is the 4 byte last pointer associated with this sequential file. This pointer is composed of a 2 byte relative record number within the file and a 2 byte offset into the corresponding physical record. This pointer is used to determine where the next line of information should be written and where the logical end of the file is.[2]

The next full word in the header following the last pointer contains the line number of the last line written, and is maintained only if this is a sequential file with line numbers. Its sole function is to insure that lines are written with increasing line numbers. The next halfword in the header is the size of the longest line in the file. This is updated (if

---

[1] Currently n=16

[2] A more detailed description of how this pointer and others are manipulated by the file routines may be found in an appendix to Volume 1, "Details on Using Sequential Files in MTS".

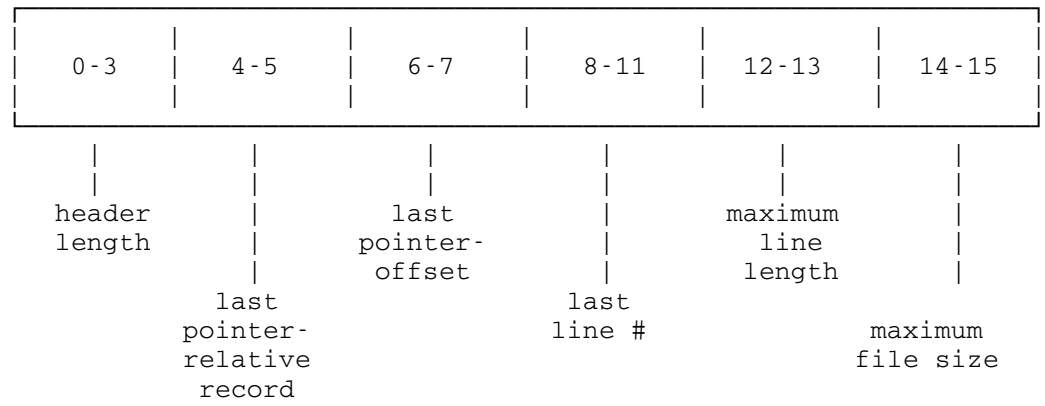
necessary) after every write operation. The last two bytes in the header are the maximum expandable size of this sequential file.

Each segment of a line in a sequential file has either 10 or 6 bytes of overhead associated with it depending on whether it is in a sequential file with or without line numbers. The 6 bytes common to both organizations is split up as 3 bytes before and after each segment. The first of the three bytes at the beginning of each segment is a flag byte indicating whether this is the first, intermediate, and/or last segment of the line, and whether this segment (i.e., the line) has a line number associated with it. The next two bytes are a count of the current segment length plus the previous segment lengths for this line. If this is a sequential file with line numbers, the next 4 bytes contain the line number associated with this segment. The three bytes at the end of the segment are similar (but not identical) to those at the front, i.e., the first two bytes are a count of the total line length minus all previous segment lengths, and the last byte is the one byte flag. The lengths kept at the front and the back of the segments are somewhat obscure but make possible the backwards reading of sequential files. Due to the judicious definition of these lengths, it is the property that: 1. For the first segment of a line, a) the leading count contains the length of the first segment, and b) the trailing count contains the total line length; 2. For the last segment of a line, a) the leading count contains the total line length, and b) the trailing count contains the length of the last segment. This is precisely the information required for forwards and backwards reading of the file.

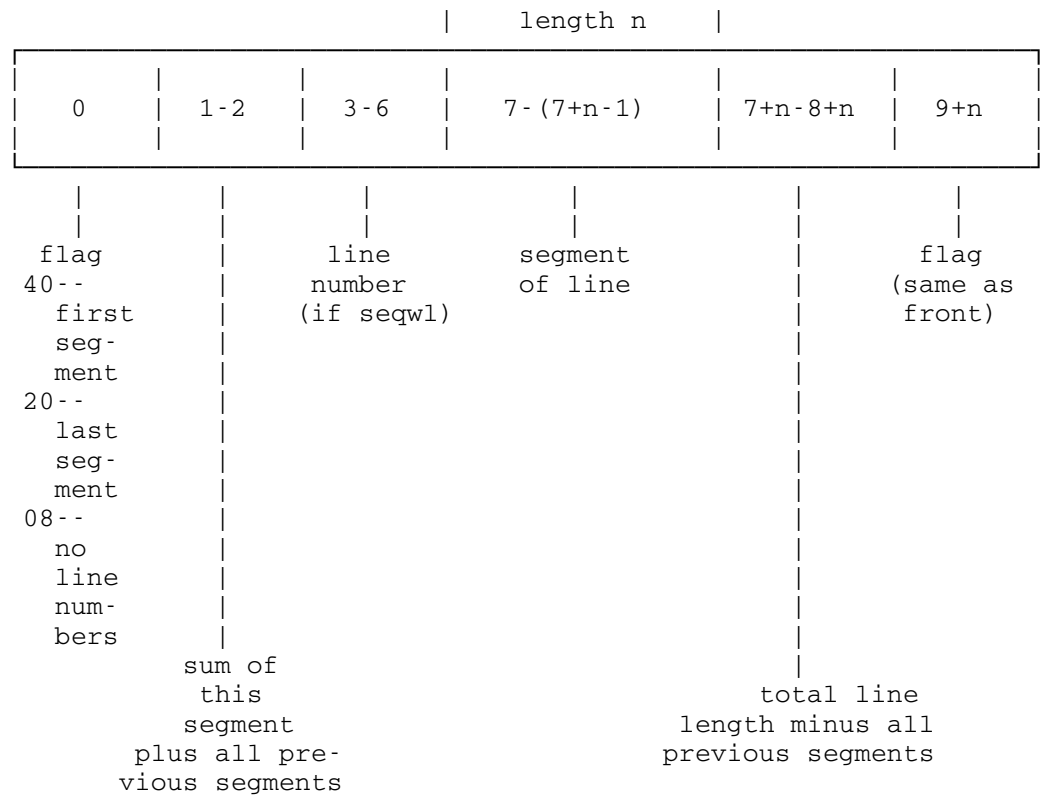
As was mentioned earlier, lines are packed sequentially onto physical records end to end, and are broken up into segments if necessary so that whenever possible all space on the physical record is used. Sometimes, however, because of the overhead associated with each segment, up to 6 or 10 bytes at the end of each physical record may be unusable. If such is the case, the physical record is filled out with the necessary number of a unique dummy byte. This, along with the length at the end of each segment, as previously mentioned, allows the backward reading capability of sequential files.

As concerns size limitations on sequential files, lines are restricted to 32,767 bytes in length. And, the total number of physical pages in a sequential file can be no greater than 32,767. Finally, as with line files, all records of the file must reside on the same volume.

HEADER -- (16 BYTES)



SEGMENT



Classification: 322/1
Date: July 25, 1977
Doct=27 Vers=1

INTERNAL STRUCTURE OF THE CATALOG IN MTS

The CATALOG in MTS is composed of a number of FILES (special, to be sure) named *MASTER.CATALOGn where n=0, 1, 2, ...255. These files may all be on the same disk VOLUME or they may be scattered across different disk volumes. For reasons of efficiency, they should be scattered across volumes (and even control units). Each *MASTER.CATALOGn is linked to the next *MASTER.CATALOGn+1. The above structure is generally determined at the time the catalog is initially built, and except for facilities provided to dynamically expand when necessary,[1] this structure does not change. Thus, it is important if one is building the catalog to know ahead of time on what volume(s) one wants the catalog to reside and to "direct" the building process in that direction.

In general, each catalog file has the following structure. The first PAGE[2] of each file is the FILE HEADER. This header contains a 4 byte id, a 4 byte count of the number of pages in this file (maximum of 816), a 4 byte link to the next file,[3] and a 4 byte data set control block (DSCB) type E address[4] for this *MASTER.CATALOGn.

The remainder of the file header is composed of a variable number of FREE SEGMENT DESCRIPTORS. The number of free segment descriptors is equal to the number of pages in the file. The free segment descriptor indicates which segments (as defined below) in the corresponding page are available for use. A free segment descriptor is composed of a 4 byte page address followed by a 1 byte bit map describing the free segments.

---

[1]The expansion is open-ended in the sense that the catalog will always create a new *MASTER.CATALOGn+1 (given available space, of course).

[2]The catalog uses 4096 byte page size physical records as does the regular file system.

[3]Catalog addresses take the form of a 2 byte public volume number and a 2 byte relative page number (starting at zero) within the volume.

[4]The DSCB type E is not used by the catalog routines.

The remaining pages in the file have the following structure. Each page has a 16 byte PAGE HEADER containing a 4 byte id, the 4 byte address of the file header, a 4 byte relative page number within this file (starting at 1), and the 4 byte address of this page. The remainder of the record is broken up into 6 SEGMENTS, each 680 bytes long.

Each segment has a 20 byte SEGMENT HEADER containing the following: the 4 byte userid to whom the segment has been assigned, a 4 byte link to the next segment assigned to this userid,[5] a 1 byte count of the maximum number of DESCRIPTORS (see below) that can be contained in this segment, a 1 byte descriptor length, and 10 unused bytes.

A segment can be assigned to the MASTER INDEX, SYSTEM FILE CATALOG, SCRATCH FILE CATALOG, or a USER CATALOG. If a segment is assigned to the master index, it may contain a maximum of 55 MASTER INDEX DESCRIPTORS each 12 bytes long. If a segment is assigned to the system file catalog, the scratch file catalog, or a user catalog, it may contain a maximum of 10 FILE DESCRIPTORS and/or SHARING DESCRIPTORS each 66 bytes long.

The master index contains a descriptor for every userid that has permanent private files in the system. This master index descriptor contains a 1 byte flag, a 4 byte userid, a 4 byte address of the first segment of the user catalog for this userid, and 3 unused bytes. The master index is generally searched only once per userid per session at the first reference to a userids private files to obtain the address of the userids catalog. Thereafter, MTS remembers where the userids catalog is to speed up subsequent references.

When a user creates his first private file, an entry is made in the master index and the user is assigned an available segment.[6] Furthermore, as is the case every time a file is created, a file descriptor is placed in the users catalog.

---

[5] The segment number (0-5) within the page is indicated in the high order 4 bits of the address (as with DSCB addresses). Segments of a user catalog need not be on the same volume.

[6] Segment 5 of each page is not assigned to new users for their first segment. This segment is reserved as an overflow segment for existing catalogs.

The FILE DESCRIPTOR contains a 1 byte flag (to distinguish it from sharing descriptors), 1 byte of owner access, 1 byte of global access,[7] 1 byte indicating the file organization and device type, the 16 byte filename, the 4 byte address of the DSCB type E for the file, a 4 byte owner ID, the 6 byte volume serial number, the 2 byte physical record size (currently 4096 bytes), a 2 byte creation date, a 2 byte last reference date, a 4 byte usage count, a 2 byte last change date, and a 12 byte program key.

In addition, if the file has been permitted (via $PERMIT) to specific userids, projects, or program keys, the file descriptor will have a 6 byte sharing descriptor linked to it.[8]

The SHARING DESCRIPTOR is composed of a 1 byte flag, a variable number of variable length SHARING DESCRIPTOR ENTRIES and, if necessary, the 6 byte link[8] to the next sharing descriptor. Each sharing descriptor entry is composed of the one byte IBM length of the userid, project number,or program key, a 1 byte flag indicating first whether the entry is a userid, project number, program key, qualified userid-program key or a qualified project number-program key and second, what type of access is allowed this userid, project number,or program key,[9] and the actual userid, project number, or program key.

The algorithm for determining access is (generally) as follows. The sharing descriptors are scanned checking for a "match" against the userid, project number, and program key in question. Since it is possible to "match" the same userid (or project number or program key) against more than one sharing descriptor entry, (by permitting access to subsets of userids, e.g., all userids whose first n characters are ...) the access of the most specific match is the one allowed.

Furthermore, userid access has higher priority than project number access and project number access has higher priority than (unqualified) program key access so that if a userid and a project number both "match", the userid access is used, regardless of the number of characters matched. (Access permitted to a program key "qualified" by a userid has higher priority than access permitted to just the userid, and access permitted to a program key "qualified" by a project number has higher priority than access permitted to just the project number). Finally, if no specific access applies, then the global

---

[7]Access allowed to others.

[8]This 6 byte link has the form, a 2 byte public volume number and segment number, a 2 byte relative page number, and a 2 byte offset into the segment. Thus descriptors need not be on the same volume.

[9]The types of access currently allowed are no access, read access, write expand, write change & empty, renumber & truncate, rename & destroy, permit, or any combination thereof.

access is used.

When a user overflows his first segment with more than 10 file and/or sharing descriptors, a new segment is allocated and the first segment is linked to it. An attempt is made to allocate the next segment on the same page as the previous segment (i.e., segment 5). In any event, a new segment will be allocated even if it becomes necessary to dynamically expand the catalog in the process.

As would be expected, whenever a user destroys a file, the corresponding file descriptor is removed from the user catalog (as well as any sharing descriptors attached to the file descriptor). Finally, when a special program is run (usually once a month) to remove expired userids from the system, the master index descriptor is removed from the master index, and all segments allocated to the expired user catalog are returned.

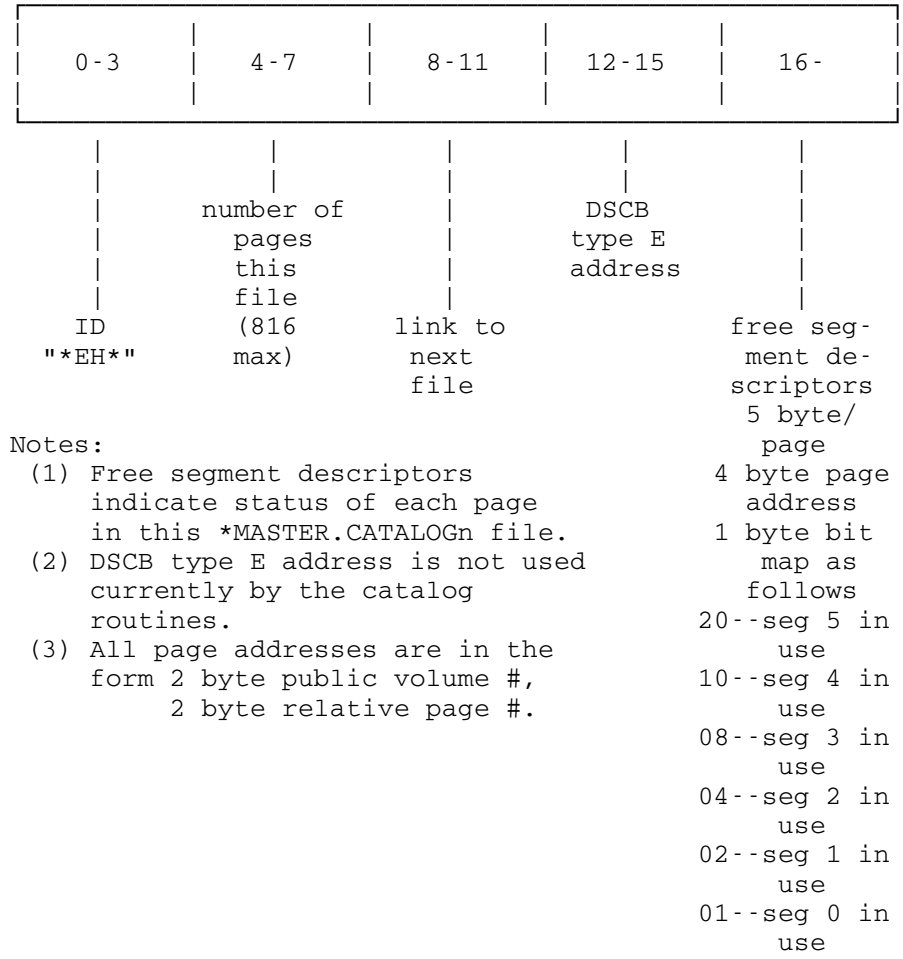
It should be noted here that the system file catalog and the scratch file catalog are identical to the user catalogs except, of course, they never expire.

One further note; the file header of *MASTER.CATALOG0 contains in addition to the normal file header information and free segment descriptors, the name of the last *MASTER.CATALOGn created, and the addresses of the beginnings of the master index, the system file catalog, the scratch file catalog, and the first user catalog. These pointers are read in and remembered when the operating system is initialized for reasons of efficiency. (The name of the last *MASTER.CATALOGn is needed for expansion.) In addition, proper manual setting of these pointers at the time the catalog is being built can in most cases guarantee that sufficient contiguous space will be available to the master index, system, and scratch file catalogs for expanding. This will be the case since user catalogs are allocated "down from" the first user catalog only. Again this is an efficiency move and is not necessary (though quite desirable).

Finally, there exist two resident subroutines which may be of use to system programmers interested in extracting information from the catalog about the file system. One returns file descriptor (and optionally sharing descriptor information) about any or all of the files in the users catalog. The other reads the catalog sequentially and returns information on a page by page basis. Since the calling sequences to these routines are rather nonstandard to say the least, the appropriate listings should be consulted.

Catalog Format

File Header (page 1 of each *MASTER.CATALOGn file -- also called "extent header")



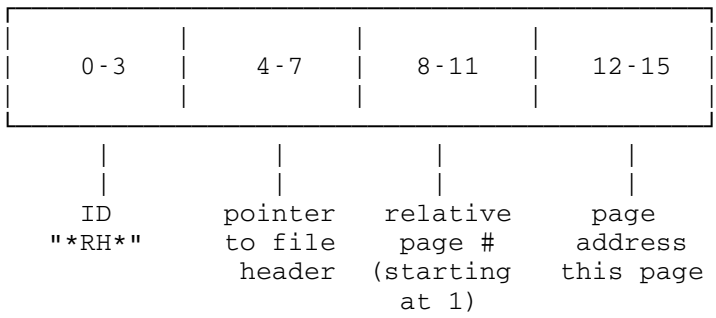
Notes:

(1) Free segment descriptors indicate status of each page in this *MASTER.CATALOGn file.

(2) DSCB type E address is not used currently by the catalog routines.

(3) All page addresses are in the form 2 byte public volume #, 2 byte relative page #.

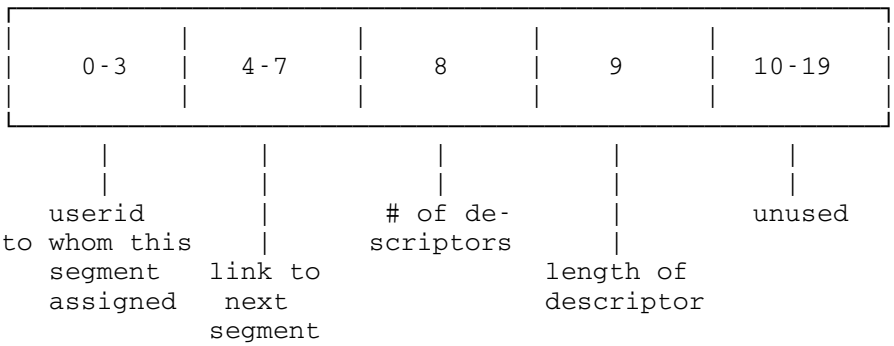
Page Header (16 bytes)



The remainder of every page is divided into six segments, each 680 (decimal; hex 2A8) bytes in length. The segments are numbered 0-5. The starting relative address of each segment within a page is:

segment 0: X'010'
segment 1: X'2B8'
segment 2: X'560'
segment 3: X'808'
segment 4: X'AB0'
segment 5: X'D58'

Segment Header (20 bytes)

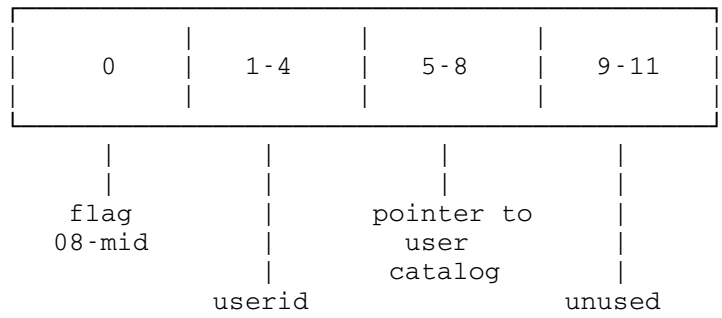


The userid above may also be on of "*MIX" (master index), or "*SYS" (public file catalog), or "*TMP" (scratch file catalog).

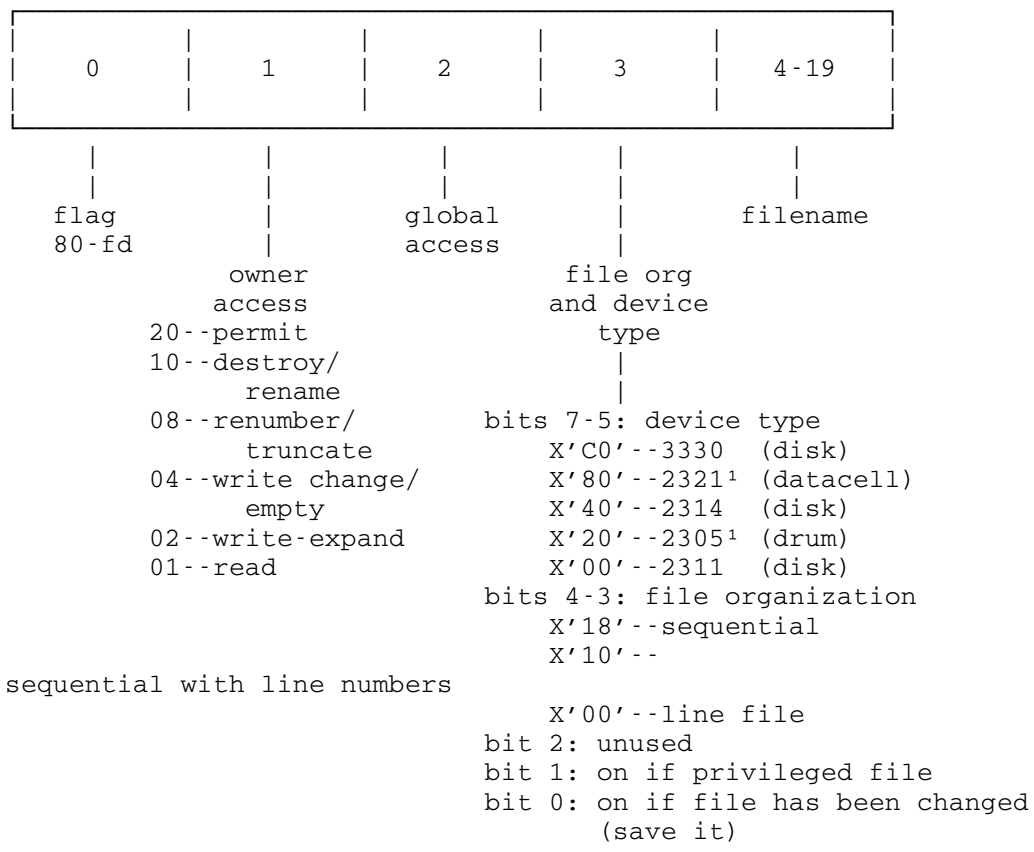
The high order bits of the public volume number (in the link to the next segment) contain the segment number (similar to DSCB addressing). For example, 20044DCF refers to the third segment on page 4DCF on MTS004.

The next segment does not have to be on the same volume.

Master Index Descriptor (12 bytes)

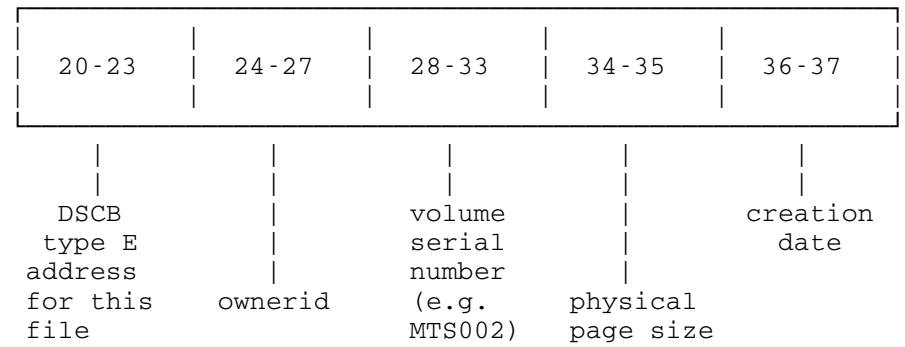


File Descriptor (66 bytes)

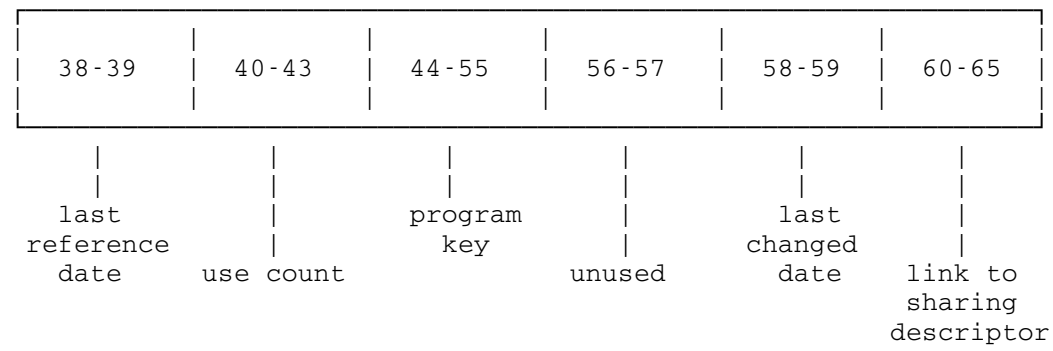


[1]Currently not supported by the file system.

File Descriptor -- continued



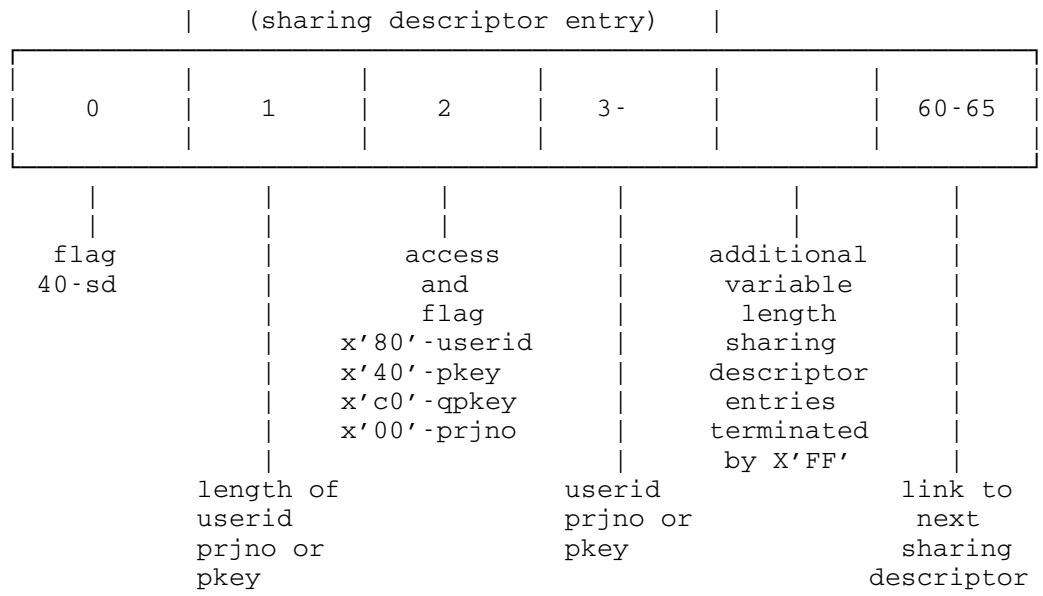
File Descriptor -- continued



Notes:
(1) Link to sharing descriptor is 6 bytes:
2 byte public volume number,
2 byte relative page number,
2 byte offset into segment

(once again the segment number is in the high order bits of t
public volume number).

Sharing Descriptor (66 bytes)



Notes:
(1) See note number 1 above concerning format of link to next sharing descriptor.

Classification: 322/3
Date: July 24, 1977
Doct=23 Vers=1

The following describes how to create the file system catalog from scratch. It must be created on public volume #1.

Run FILE:CCATL.O+FILE:FILERTNS, where CCATL is the catalog construct program and FILERTNS is the usual file routines with a set of tables containing the drives and volumes on which the new catalog is to reside.

CCATL first prints out

CATALOG BUILD PROGRAM, NOVEMBER 75.
ENTER ALL NUMBERS IN DECIMAL.

And then prompts you to

ENTER SIZE OF MASTER INDEX IN PAGES

to which you might reply

24 (which is the size used last time at UM - Nov. 75).

Then CCATL will prompt you to

ENTER PUBLIC VOLUME # FOR FIRST EXTENT OF CATALOG .

You reply

1 (which is the only acceptable response since the first extent of the catalog is always on public volume 1).

Then CCATL will prompt you:

HOW DOES MTS001 SOUND (OK)?

And you reply

OK .

Then CCATL prompt you to

ENTER NUMBER OF PAGES TO ALLOCATE FOR THIS EXTENT:
REMEMBER, 1 PAGE PER EXTENT USED BY EXTENT HEADER.

The standard reply is

25 (which means the master index will fit exactly in the first extent).

Then CCATL asks you to

ENTER SIZE OF SYSTEM FILE CATALOG IN PAGES

to which a reasonable reply is

10 (which was used at UM last time the catalog was created).

Since the first extent of the catalog was completely pre-allocated to the master index (intentionally), another extent must be allocated for the catalog at this time. If one wanted the master index and possibly the system catalog, scratch file catalog and first part of the user catalog all on the first extent, the size in pages of the first extent should have been specified as greater than or equal to the combined sizes of the individual catalogs.

In any event, CCATL now notifies you that the

REQUESTED SIZE HAS OVERFLOWED THIS EXTENT
ENTER "OK" TO ALLOCATE ANOTHER EXTENT
"NO" MEANS REPROMPT FOR CURRENT CATALOG SIZE .

If you enter "OK", then CCATL will ask that you

ENTER PUBLIC VOLUME NUMBER FOR NEXT EXTENT OF CATALOG

to which your reply might be

2 .

Then CCATL will ask you

HOW DOES MTS002 SOUND (OK)?

And you can say

OK .

Then as before, CCATL prompts:

ENTER NUMBER OF PAGES TO ALLOCATE FOR THIS EXTENT.
REMEMBER, 1 PAGE PER EXTENT USED BY EXTENT HEADER.

You reply as before

11 (because you want the system file catalog also to be on a single extent on a separate volume all by itself).

In a similar fashion you will be asked to

ENTER SIZE OF SCRATCH FILE CATALOG IN PAGES

and

ENTER SIZE OF (FIRST PART OF) USER CATALOGS IN PAGES

As before if the requested size of the catalog overflows the current extent, a new extent will be allocated of the proper size and on the volume requested.

When CCATL finishes, it prints out the location of the beginning of each of the catalogs (as a fullword hex disk address).

Classification: 323/1
Date: July 25, 1977
Doct=28 Vers=1

THE INTERNAL STRUCTURE OF THE SYSTEM WIDE SHARED FILE TABLE IN MTS

In a shared file environment, before any operation (reading, writing, emptying, etc.) can be performed on a file, guarantees must be made to ensure that concurrent usage of the file at any particular point in time will not endanger the integrity of the file.

To accomplish this, files are "locked" at one of three inclusive levels (read, modification, or destroy) before any specific file operation is performed. In addition, checks are made before locking is allowed to ensure that certain rules of concurrent usage will not be violated.

It should be noted that the problems of determining allowable concurrent usage of a file are separate and not related to the problem of determining allowable access to a file. It is assumed that by the time the system wide shared file table is interrogated, it has been determined that access appropriate to the locking request has been "permitted".

In order to determine who may concurrently use a file and how at any given point in time, MTS maintains a table (in shared VM) indicating at any given point in time, all the files currently open and/or locked, how they are locked, and by what task (job); as well as what tasks are currently waiting to lock the file and how they are waiting.

This table is necessary to determine (with the aid of the rules of concurrent usage) whether, at any given point in time, a particular type of opening and/or locking can be allowed.

The rules of concurrent usage are as follows:

1) Any number of tasks can have a file locked for reading at the same point in time as long as no other task has the file locked for modification or destroying.

2) Only one task can have a file locked for modification (writing, emptying, truncating, etc.) at any given point in time, and then only if no other task has the file locked for reading or destroying.

3) Only one task can have a file locked for destroying (renaming or permitting) at any given point in time, and then only if no other task has the file open, locked for reading, or locked for modification.

If it is determined, via the rules of concurrent usage, that a file cannot be locked as requested, the task is (optionally) queued to wait on the file. (Internally this is accomplished via

an SVC sleep.) Before a task is queued to wait on a file, however, checks are made to determine whether queueing a task to wait on the file will result in a deadlock situation whereby two or more tasks will wait indefinitely on their respective queues.

The simplest form of deadlock is the "single file" situation. For example, suppose both task A and task B have FILEX locked for reading, and then task A requests that FILEX be locked for modification. Since someone else (task B) also has the file locked, task A will be queued to wait on FILEX. Then suppose task B requests that FILEX be locked for modification. MTS realizes not only that someone else (task A) has the file locked, but also that queueing task B to wait on FILEX would result in both tasks A and B waiting indefinitely for the other to unlock the file. In this situation, MTS will not queue task B to wait, but will return an error indication instead.

A "single file" deadlock is fairly easy to detect, more complicated forms of deadlocks can occur when multiple files are concerned. The method MTS uses to detect "multiple file" deadlocks is as follows:

1) Define a relation B (Blocking) as follows:

   TASKA is in relation B to TASKB
   (TASKA B TASKB iff)

   TASK A has a file open and/or locked in such a way that TASK B is blocked from using that file.

   Blocking is defined as follows:
   A) A task with a file open blocks a task waiting to destroy the file.
   B) A task with a file locked to read blocks a task waiting to modify or destroy the file.
   C) A task with a file locked to modify blocks a task waiting to read, modify, or destroy the file.
   D) A task with a file locked to destroy blocks a task waiting to open, read, modify or destroy the file.

2) Build the M by M Matrix representing relation B where M is the total number of tasks either (a) with files open and/or locked blocking another task or (b) being blocked.

3) The transitive closure relation B+ of relation B is defined as follows:

   TASKA B+ TASKB iff

   There exists N tasks TASKi $1 \le i \le N$ such that

   TASKA B TASK1 B...B TASKN B TASKB

   (i.e., there exists a "chain" relating TASKA to TASKB).

4) Using Warshalls algorithm, (see Gries--Compiler Construction for Digital Computers) compute the M by M Matrix which represents the transitive closure relation.

5) Now see if there exists an i such that

TASKi B+ TASKi

If so, then a deadlock situation exists.

A necessary condition for a "multiple file" deadlock is that the task being queued to wait on a file must have some other file open and/or locked and some other task must be waiting on that file. This check can easily be made to determine if it is really necessary to build the matrix.

Once a task is queued to wait on a file, it "sleeps" until the task(s) which have the file locked, unlock the file. At that point, the unlocking task determines if any task(s) sleeping on the wait queue can be "awakened". The unlocking task makes its decision using the same rules of concurrent usage described above.

The basic format of the system wide shared file table is as follows. The first 2 bytes at the beginning of the table are used by tasks to "wayt" when the table is full. 1 byte is used to "wayt" for space for a file entry, and 1 is used to "wayt" for space for an open or waiting element. The next 2 bytes are a pointer to the chain of open and/or locked file entries. Then follows 2 bytes which are a pointer to a chain of available file entries (each 24 bytes). After that 2 bytes which are a pointer to a chain of available open or waiting elements (each 6 bytes). The next two bytes are a count of the number of open and/or locked files. After this is a 2 byte count of the number of matrix computations performed and a 2 byte count of the number of deadlocks detected. These last 6 bytes are maintained for informational purposes only.

Initially the table contains only available entries. As open or waiting elements are needed, a 24 byte available entry is broken up into 4-6 byte available elements. Eventually, when the open or waiting elements are returned, the 4-6 byte available elements will be "re-grouped" into a 24-byte available entry.

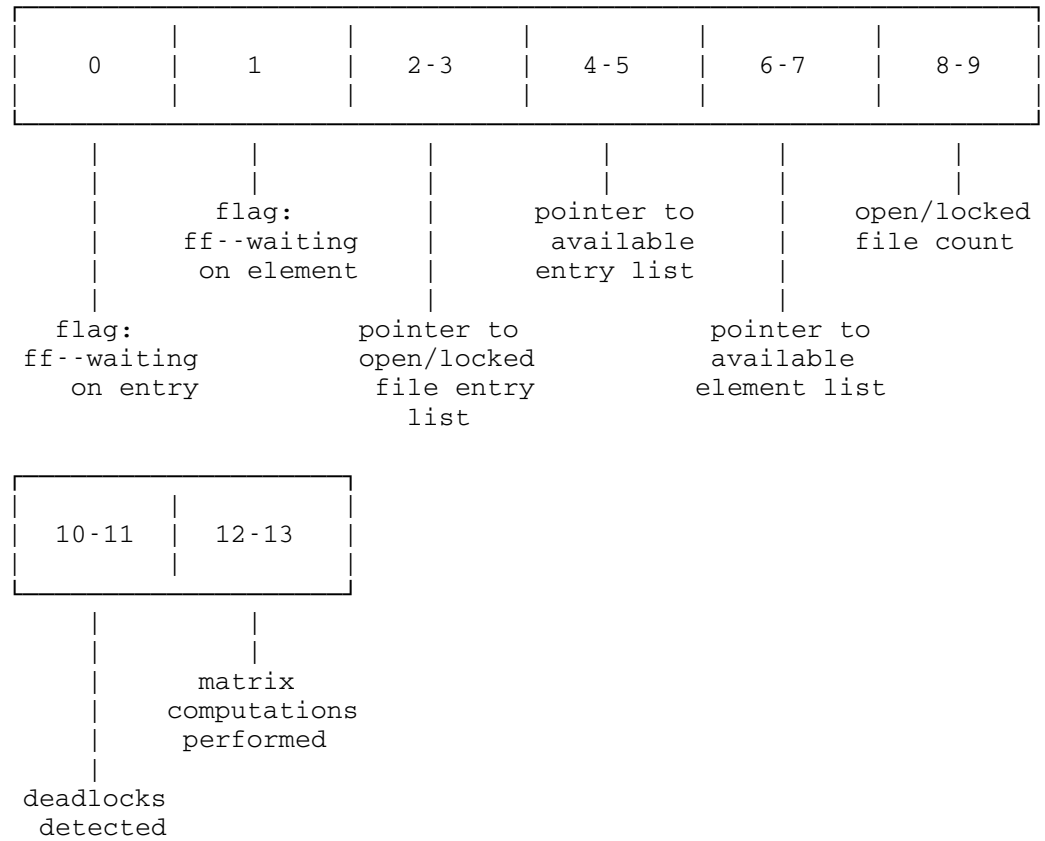
The 24-byte open and/or locked file entry consists primarily of the 16-byte name and a 2-byte link to the next open and/or locked file in the chain. In addition, 1 byte is used to indicate whether the file is being modified or destroyed. 2 bytes are used as a pointer to the chain of open and/or locked elements (tasks with the file open and/or locked), and 2 bytes are used as a pointer to the chain of waiting elements (tasks waiting to lock the file).

The 6-byte open/locked element consists primarily of a 2-byte task number indicating the task that has the file open and/or locked and a 1-byte flag indicating whether the task has the file open or not and if locked, how the task has the file locked. In addition, of course, a 2 byte pointer to the next open and/or locked element is necessary.

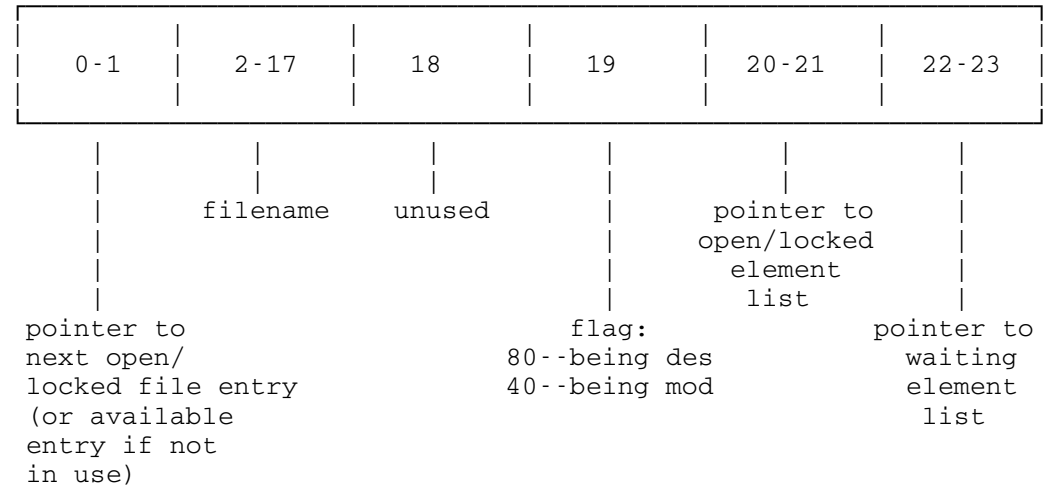
The 6-byte waiting element is identical to the 6-byte open element except that the flag byte indicates how the task is waiting to lock the file. The flag byte also indicates whether the wait has been cancelled. Finally, the flag byte contains the bit on which the waiting task "sleeps" and correspondingly is "awakened".

System Wide Shared File Table Format

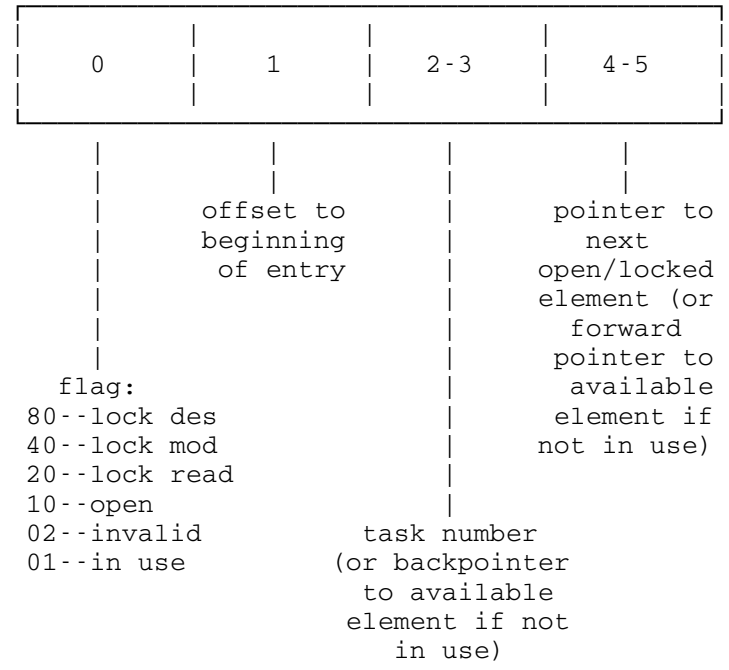
Table Header



File Entry (24 BYTES)



OPEN/LOCKED ELEMENT (6 bytes)



WAITING ELEMENT (6 bytes)