CFS.DOC

Dave Braithwaite
July 5, 83

This contains two CFS-20 related documents:

```
CFSFS.MEM   --- Dan Murphy
CFSDOC.MEM  --- Arnold Miller
```

The first document is the functional specification and is still accurate despite its age (good design). The glossy flier on CFS-20 reflects this document and is a good document to read. The second document is an internals document which is still in the process of being updated. It provides good insight into the internal mechanisms of CFS.

Document:

Date: 28 October 1981

Project: TOPS20 Common File System

Charge Number: E20-02131

Product ID:

Functional Specification for

TOPS20 COMMON FILE SYSTEM

Issued By:	Dan Murphy	(Developer)
	Fred Engel	(Supervisor)
	Peter Hurley	(Development Manager)

TABLE OF CONTENTS

1.0 PRODUCT OVERVIEW

1.1 Product Description

This project is to develop a "Common File System" for TOPS20. The Common File System capabilities are applicable to configurations of two or more 36-bit processors, each with its own main memory, interconnected by a high speed bus ("CI").

The objective of the Common File System ("CFS") is that disk structures and files within such a system are available to jobs on all processors, regardless of the physical connections of the disk devices.

1.2 Markets

The Common File System is a general operating system capability and is applicable to all present DECSYSTEM-20 markets.

1.3 Competitive Analysis

This project provides more and larger configuration alternatives. This project is not closely related to "distributed processing" in that it is only applicable to configurations on a CI and therefore within the 100 meter limit of the CI.

VAX/VMS is developing means for multiple processors to reference files on a single disk system; however, the basic difference in filesystem architecture between TOPS20 and VMS makes the projects somewhat different.

Related capabilities include "Network File Access" or other techniques for moving files among nodes of a network. CFS is a more powerful and transparent form of file access because it implements all monitor file primitives visible to the user program and operates over a high speed bus.

"Multiple Processors" (implying shared memory as with TOPS10 SMP) is a related capability. SMP is a more powerful approach to the use of multiple processors in that it provides greater transparency and better dynamic load leveling. Jupiter will not support shared main memory however, so an SMP implementation is not possible. There are compensating advantages of CFS over SMP in the area of failsoft and isolation of failures, and in the maximum size of configurations which can be supported.

1.4 Product Audience

There are two primary types of customers that will require CFS:

1. Existing customers with KL10-based configurations who wish to upgrade to Jupiter and retain their massbus peripherals. These customers may add a Jupiter to the existing configuration, and both processors will be able to access the massbus disks. If additional disk storage is provided via HSC-50, the KL10 as well as the Jupiter processor will have access to it.

2. Customers who require multiple processors for capacity or reliability reasons. The CFS allows multiple processors to operate with a single logical disk file system such that the jobs may be run on one processor or another as dictated by load or other considerations. Processors may be added to or taken from the configuration (subject to hardware restrictions) with the remaining processors continuing to use the file system.

2.0 PRODUCT GOALS

2.1 Performance

1. All unprivileged monitor calls which affect disk files on present one-processor TOPS20 systems will work and will have the intended effect on any disk structure within the configuration.

2. The overhead associated with maintaining the common file data base on multiple processors will cause an increase of not more than 10% in execution time of file primitives and operations.

3. A processor referencing files on a disk not directly connected will incur no additional overhead in transferring data.

We expect to use MSCP (Mass Storage Control Protocol) for the data transfers to support file operations. This will exist on the CI along with DECNET and possibly other protocols supporting other functions. MSCP should achieve efficient use of the CI, low-overhead operation of the monitors, and high-bandwidth file interchange. File structure information such as directories and index blocks is passed exactly as read from disk. By passing TOPS20 file data directly, we avoid the overhead of copying and conversion incurred with other protocols.

Obviously, a processor acting as a file server for another processor will incur overhead for this activity not relating to jobs running on it. This overhead will involve primarily instructions executed at interrupt level and main memory space to buffer data being transferred.

CFS supports shared-writable pages (simultaneous write file access) on multiple processors. This is used for various internal mechanisms (e.g. directory lookup, disk allocation tables) as well as user program functions. This type of access generates IO activity and overhead not present on single-processor systems. Because data cannot actually be referenced simultaneously by two processors, it must be moved from one to another by the operating system. Users will be advised of this and should arrange applications so as to avoid frequent write references to the same data from different processors.

Since the monitor itself uses this facility, we conducted a study of monitor reference patterns to ensure that this activity will not be a significant bottleneck. We recorded monitor reference patters to directories and disk allocation tables under actual and simulated loads. This was done by using the SNOOP facility to detect and record references where the job making the reference is different than the job which made the most recent previous reference. This provides worst-case data on the frequency of moving monitor data between processors. We determined that only a few (3 or less) directories were referenced sufficiently often to be of interest. These were all common system directories (e.g. <SUBSYS>), and the frequency was not so high as to suggest a problem. Even so, a further efficiency is possible by duplicating the directory on a separate structure for each processor.

2.2 Environments

Minimum configuration requires two processors, either KL10, Jupiter, or mixed, and a CI. Each processor must have a connection to the CI. (The question has been raised as to whether CFS might be operable over an NI connection. This will not be supported in first release. There should be no logical reason that NI couldn't be used, but additional study and experience is necessary to understand the performance implications. Additional implementation work would also be needed.)

Each processor must have its own main memory, swapping device, boot device, and console.

Each processor must have direct access to its public disk structure. In a future release, it may be possible to eliminate this requirement if desired, however, there are other requirements for a directly connected disk (e.g. swapping) which will also have to be addressed.

The maximum configuration for first release of CFS is four processors, however there shall be no CFS-specific software limitation on a larger number. This limit is based on our current knowledge of the CI and the lack of experience with this architecture. The practical limit may be higher. A maximum of one (dual rail) CI will be supported for first release.

2.3 Reliability Goals

1. A customer should be able to improve net system availability of his configuration by use of multiple processors and the CFS.

2. The CFS should cause no significant decrease in the reliability of each single processor.

3. Failure of one processor will have no effect on other processors except for file data which is in the memory of the failing processor.

KL-Jupiter configurations such as this provide some additional redundancy over single-processor systems. The fail-soft characteristics are not symmetric however. The Jupiter processor and/or its attached disk can fail without affecting jobs on the KL processor or files on the KL disks. However, failure of the KL may have more impact depending on what peripherals are connected to the Jupiter.

2.4 Non Goals

1. CFS will not support other than disk structures.

2. CFS is not intended to work with operating systems other than TOPS20 or with machine architectures other than 36-bit.

3. At first release, CFS will not support passing file data through one processor to another except for the case of a KL10 processor acting as file server for massbus disks. That is, a Jupiter processor will never act as a pass-through for file data. Handling pass through data increases overhead and raises the possibility of multiple paths between a processor and a disk which would require routing decisions. CFS may be used in configurations with multiple CI's, but such configurations may have processors that are unable to communicate with some disks. These restrictions may be removed in a subsequent release.

4. CFS does not provide any automatic balancing of job load among processors in a configuration as does SMP. However, users should find it convenient to login to the less loaded processor and/or to switch processors (by logging out and back in again) if the load becomes unbalanced.

2.5 Future Development

In a future release, it may be desirable to allow Jupiter processors to pass file data so that configurations may be built in which some processors do not have direct access to some disks. In general, this will also require routing decisions to be made since multiple paths can easily arise once there are several CI's in a configuration. It would still be avisable to minimize the amount of pass-through traffic in order to minimize overhead, but pass-through offers sizable configuration flexibility and some failsoft features.

3.0 FUNCTIONAL DEFINITION

3.1 Operational Description

Two or more processors are interconnected via a high-speed bus ("CI") having a bandwidth at least comparable to disk transfer rates. A disk which is to be used by a processor must have a direct path to that processor, i.e. be on the same CI. The only supported exception to this will be to allow files to pass through the KL10 to reach Jupiter so that files on Massbus devices may be used by Jupiter processors.

```
                  ------------        --------------
                  !    HSC    !       !    HSC     !
                  !    DISK   !       !    DISK    !
                  !           !       !            !
                  ------------        --------------
                       !!                  !!
                       !!                  !!
         CI ====================================================
                 !!                  !!                     !!
                 !!                  !!                     !!
             ------------       --------------         -------------
             !   KL10   !       !            !         !           !
             !  CPU&MEM !       !   CPU&MEM  !         !  CPU&MEM  !
             ------------       --------------         -------------
                   !!
                   !! MASSBUS
                -------------
                !           !
                ! MASSBUS   !
                !      DISKS!
                -------------
```

One or more logical structures exist on the set of disks. All of these structures are visible to jobs on all of the processors unless the system administrator specifically declares particular structures as 'private' to particular processors.

In order to provide access to Massbus disks connected to a KL10, the KL10 will act as a logical disk controller on the CI for the Massbus disks. There is no visible distinction between a disk structure directly connected to a processor and one which is accessed via another processor. The usual monitor calls are used to access files and structures, and all file open modes are allowed with the exceptions listed below. Shared file access is permitted, and programs need not be aware that other jobs sharing a file are on different processors; however, it may is advisable for reasons of efficiency to avoid simultaneous modification of a file on different processors.

In future releases, it may be desirable to support configurations which include multiple CI's, each with a subset of the processors connected. In such cases, it may be desirable for Jupiter CPUs to act as pass-through servers in order to provide access to a disk from another processor. This will not be supported at first release.

File facilities specifically include:

1. File naming and lookup conventions (GTJFN) - File names on the common file system include structure, directory and subdirectories, file name, extension, generation number, and attributes. Full recognition and wild-carding is available; name stepping (GNJFN); normal access to FDB.

2. Usual open and close modes (OPENF, CLOSF, CLZFF).

3. Usual data transfer primitives, both sequential and random (BIN, BOUT, SIN, SOUT, SINR, SOUTR, RIN, ROUT, DUMPI, DUMPO).

4. File-to-process mapping (PMAP) including all modes (shared read, copy-on-write, shared write, unrestricted read).

5. The device type associated with files on the common file system is the same as that presently used for disk.

6. Privileged operations MSTR (mount structure) and DSKOP.

The above includes all file system primitives relating to accessing files and transferring data but does not include other primitives which may use certain file system entities but which are considered separate and distinct facilities (e.g. ENQ/DEQ).

See Appendix 1 for implementation considerations.

3.2 Restrictions

A file open with OF%DUD (don't update disk) on one processor may not be opened on any other processor.

Other devices such as magtapes and line printers are not part of CFS and may not be open simultaneously on multiple processors.

Use of simultaneous write access with active writing of file data by jobs on different processors requires the system to move pages among the processors and hence will be much slower than on a single processor. The write token is maintained on a per-OFN basis. This means that a program requiring write access to any one or more pages must have exclusive access to the entire OFN. Each OFN represents 256K words of the file. For large files, programs on different processors could be executing simultaneous write references with no delay if they were referencing data in different 256K sections of the file.

A structure must be "mounted" on any processor which is to access files on it. To be physically removed from a drive, a structure must be dismounted by all processors. The relevant Galaxy components should be modified to provide mount information from processors other than the one on which they are running, but this in not planned for Jupiter CFS. Hence an operator will have to query the OPR program on each processor to find out what users have the structure mounted. Each processor will know, however, which other processors have the structure mounted so that the operator can quickly determine if the structure can be removed.

Dual port operation of a single drive from a single KL10 is not currently supported. If it is implemented, the fact that there are two paths to a disk will not be visible to CFS and will have no impact on the exclusion of routing capabilities in the first release.

4.0 COMPATIBILITY

4.1 DEC Products

All program and user interfaces are compatible with previous versions of TOPS20.

Mountable disk structures are compatible with previous versions of TOPS20.

4.2 DEC Standards

The CFS will use the corporate SCA protocol on the ICCS bus, and will use MSCP for disk data transfer operations.

The CFS will not use DECNET.

4.3 External Standards

None applicable.

5.0 EXTERNAL INTERACTIONS AND IMPACT

5.1 Users

All users of the disk file system are potential users of CFS, however most users will not be aware of or affected by CFS. Some applications developers will rely on CFS to allow applications to exist on multiple processors and communicate through files.

Other projects within the Coexistence and Distributed Processing group will use CFS to provide layered functionality on multiple homogeneous processors.

5.2 Software Products

5.3 Products That Use This Product

The following may use CFS: RMS, DIF, language OTS's.

5.4 Products That This Product Uses

The following hardware components are required:

KLIPA - Interface between KL10 and ICCS bus.

Jupiter CI Port - Interface between Jupiter and ICCS bus.

KL10 Microcode - modifications to support "write access in CST".

The following software modules are required:

MSCP driver and server in TOPS20.

Systems Communications Services (SCA/SCS).

5.5 Other Systems (Networks)

The CFS is not visible to other network hosts; the files in the CFS disk structures may be accessable by remote nodes as provided by other facilities (DAP, NFT, etc.) Each processor in a CFS configuration is a separate network node with its own node name.

The CFS itself does not use node names to reference files and hence is independent of any constraints or requirements of network node naming.

5.6 Data Storage, File Structures, Data Formats and Retrieval Subsystem

The CFS requires an open file data base which is resident in each processor of a configuration. 2-4 words per OFN are required. Other resident storage requirements are one page (512 words) or less. As a side effect of allowing all processors access to all mounted structures, it may be desirable to build standard monitors with a larger number of mountable structures than at present.

The file structure will be identical with previous releases of TOPS20.

Files may be saved and restored with DUMPER without regard to which processor DUMPER is run on, except that DUMPER must be running on the processor which has direct connection to the required tape drive.

5.7 Protocols

CFS will use the corporate SCA protocol on the ICSS bus and will use MSCP for data transfer.

CFS will use a private protocol for control of file openings, structure mounts, file state transitions, etc. There is no present corporate protocol which supports these functions.

6.0 RELIABILITY/AVAILABILITY/SERVICEABILITY (RAS)

6.1 Failures Within The Product

Failures within the CFS-specific software will most likely cause a crash of one processor in a multi-processor environment. Such failures may include loss of recently modified file data. Failures which affect inactive files or file directories are possible, but should be no more frequent than at present.

A processor may be brought on line without restarting other processors in the configuration.

Any disks which are available only via a failed processor will be unavailable so long as that processor is inoperative. If such disks are dual-ported to a different processor, they may be mounted via that processor and remain in use although all open files must be re-opened.

With HSC50, most disk errors will not be seen by the processor(s). All recovery and logging will be handled by the HSC50. Any disk errors that are reported to the processor will be logged in the system error file for that processor. Disk errors occuring on pages that are being "passed through" a processor (e.g. a KL10 servicing a request for a Massbus disk) will be logged on the processor to which the disk is directly connected. If a hard failure occurs such that the server processor must inform the requesting processor that the request could not be completed, then the requesting processor will also log the failure.

7.0 PACKAGING AND SYSTEM GENERATION

7.1 Distribution Media

CFS will be an integral part of TOPS20 and will require no additional packaging.

7.2 Sysgen Procedures

CFS code will be in all monitors. No specific monitor builds are required.

8.0 DOCUMENTATION

Installation Guide - Describe possible CFS configurations; describe installation of CFS. See also "Installation Procedure" Functional Spec.

System Administrator's Guide - Describe characteristics of CFS; assignment of jobs to processors.

Monitor Calls Reference Manual: JSYS to control configuration (small change to MSTR%); error codes.

SYSERR Manual - Specific error cases.

Operator's Guide - Bringing individual processors up and down.

9.0 REFERENCES

1. Functional Specification for Loosely Coupled Systems (LCS) - Fred Engel, 30 April 1980

2. Jupiter Configuration Recommendation (Memo) - Peter Hurley, 1 May 1981

3. LCS and the Common File System (Memo) - Dan Murphy, 15 Jan 1980

10.0 APPENDIX

10.1 Implementation Details

All transfers will be in page (512-word) units. The PAGEM-PHYSIO interface will recognize a request that must be handled via the CI and another processor instead of a direct disk transfer. That is, the access to the common file system disks will be exclusively through address mapping as it is now. If the full CI hardware were available, a disk transfer could always be requested directly and moved to or from memory without the involvement of any other processor. In actual systems (including KL10s), a particular processor may not have direct access to a particular disk and so must forward a request to a processor which does. The processor making the request acts in exactly the same way as if the request had been made to disk; when the page is ultimately received from the other processor, the usual transfer completion action is invoked. Some processors must be 'servers' in such a system. A server must be prepared to receive requests for disk transfers from other processors at any time and respond to them quickly. This is reasonably simple; the server processor places the disk request in its own disk transfer queues (including allocating some memory) and when the transfer is complete, starts another transfer of the data over the processor-to-processor link to the destination processor.

We intend to provide this by implementing an MSCP server module. Thus, a host processor will make disk requests in the same way whether the disk is directly connected via the CI or indirectly accessed via another processor.

Open File Data Base

A major new element of the common file system is the common open file data base. This is somewhat equivalent to the present OFN data base in TOPS20 but is maintained jointly on all processors. Interlocks and updating mechanisms serve to let each processor make necessary accesses to its own copy of the data, to keep all copies updated, and to handle a processor crashing and coming back on line.

Files that are open exclusively on one processor or on multiple processors for read-only (including copy-on-write) can be handled with no delays except for the data base lock. Simultaneous write on multiple processors requires some additional algorithms, primarily the passing of the "write token" from processor to processor. There is a separate write token for each open file. The write token logically exists on only one processor at a time and allows processes running on that processor to read and write data in the file to which the write token applies. If a processor does not have the write token for a file, no accesses, read or write, are allowed. The write token for a file which is shared-writable and is being actively used by multiple processors will be rotated among the processors. If only one processor is demanding it, it will move to that processor after a short time interval to prevent thrashing. Also, a maximum time limit will be used to prevent any one processor from hogging in the presence of demand from others.

In order to give up the write token, a processor must make inaccessible all of the pages of the file to which it applies. This is done by modifying the index block or SPT for pages then in use and writing any modified pages to their home addresses on disk.

Since all processors are running the same operating system, we assume a consistent algorithm for requesting and giving up the write token will be present on all. The CST will be used to detect the first write reference to a page so that the processor can request the write token for the file if it is not already owned.

We have considered having the write token exist on a single-page basis, however the data base required to support this seems to be too great. On the other hand, it is possible to leave file pages readable when the file write token passes to another processor and then only invalidate pages when the processor with the write token actually modifies them. This incurs a delay each time a page is first accessed for write however, since the writing process must wait until the invalidate is acknowledged by all other processors.

There is a file opening mode, OF%RDU, which always is allowed but does not guarantee that the file will be seen in a consistent state. A file open in this state on one processor may continue to be read when another processor has the write token.

Directories

Once we assume the basic file mapping mechanism as above, most of the remainder of the TOPS20 file system works without significant modification. Directories are mapped into the process address space and referenced by the process as at present. A common inter-processor lock mechanism must be implemented to handle directory locks. The write-token mechanism will serve to allow each processor to modify the directory as necessary. Most directories are referenced only by a single job and so should present no problem of excess movement of the write token. Some common directories (e.g. <SUBSYS>) are referenced frequently by many jobs but only for read; so again delays for write-token latency should not be a problem.

The disk allocation tables are mapped in the same way as directories, and the basic page mechanism will support this. There is already a explicit lock on the use of these tables; making it global will prevent any improper simultaneous reference by multiple processors. We are planning a study of the patterns of use of the allocation tables to determine if there will be excessive traffic between processors.

Structure Data Base

The structure data base will remain generally as it is now. All structures which are accessable ("spinning") will be included, whether or not any users on the system have mounted them. When the number of users having a structure mounted on a particular processor changes from 0 to non-0 or vice versa, a message is broadcast to all other processors. Each processor, on receipt of such a message, adjusts its own data to reflect which other processors have the structure mounted. This can be shown as a "funny" user name, e.g. *KL2102* so that MOUNTR will know whether the structure can be physically removed from the drive.

The structure data base will also record if a structure has been declared "private" to one processor and will limit access accordingly.

MAJOR AREAS OF DEVELOPMENT - Affected modules

I. File System Initialization (FILINI, MEXEC)

Exchange configuration information; each processor knows of all others.

Exchange mounted structure info.

II. Mount Structure (MSTR)

III. Handle disk allocation tables. (DSKALC)

Global (inter-processor) lock set while modifications in progress. Make present lock global. Allow bit table routines to run OKSKED.

Write-token mechanism handles concurrent modification by multiple processors.

IV. OFN (DISC, PAGEM)

Write-token handled on per-OFN basis (equivalent to per-file for non-long files).

Finite state model to represent movement of write-token, updating of pages, etc.

Support of simultaneous write for OF%DUD requires extra work - may be deferred. (I.e., limitation is that all openers of OF%DUD file must be on same processor.)

V. Page Management (APRSRV, CISRV, LCSSRV, PAGEM, SCHED)

Detect first write to page (bit in CST - ucode requirement).

Force out all pages in OFN, assuming none modified.

Detect reference to page in OFN not currently available.

Transfer page to/from other systems.

VI. Misc (BUGS, GLOBS, STG)

OFN STATES

1. Open, any processor read
10. Exclusive request sent, waiting for response.
2. Open, exclusive not self (i.e. access prohibited)
20. Share request sent, waiting for response.
21. Wait before repeat send of share request.
3. Not in use
30. Status request sent, waiting for response.
4. Exclusive, self

Request conflict resolution:

While waiting for response to a sent request, a conflicting request may be received. Based on the types of the two requests, one of two things is done:

1. Deny the received request, continue to wait for response to the sent request.

2. Grant the received request, continue to wait for denial of sent request.

The priority of requests are:

1. Exclusive
2. Share

Each processor is assigned an arbitrary priority which is assumed to have no significant affect on system performance.

The rules are:

If received request is higher priority than sent request, then do 2;

If received request is lower priority, then do 1;

Otherwise, if this processor has higher priority, do 1; Otherwise, do 2.

Any received request which is compatible with the current state is granted, even if no change of state occurs in the local OFN. Hence, processors do not have to change states of OFNs not being actively used.

Note the above is only to resolve race conditions. Once the race is resolved, the "losing" processor will wait for a small time and then resend the request. Exclusive or shared access must be given up upon request if the access has been available for longer than a specified period of time. (This period may be dynamically determined based on number of pages in use or modified, i.e. work necessary to give up access.)

CFSSRV is a lock manager. The locks it manages represent resources in the system, but CFSSRV is not aware of the mapping of lock to resource. The mapping, or meaning, is made by the creator of the resource.

Files are a resource with CFS locks. Each file has the following CFS locks:

- open type
- write access
- ENQ/DEQ lock

In addition, each of the sections of the file, represented by an OFN, has an access token. Therefore a file has up to 512 access tokens.

When a file is opened, the "open type" and "write access" lock are acquired. The "open type" is either"

- shared read (frozen)
- shared read/write (thawed)
- exclusive (restricted)
- promiscuous (unrestricted)

The word in parentheses represents the argument to OPENF%.

If the opener requests "frozen write" access, then if the "open type" lock is successfully locked, i.e. no one has the file open in a conflicting mode, the "write access" lock is acquired. This is an exclusive lock that represents the single "frozen write" user of the file. The lock is held by the system that has the file opened "frozen write".

Each of the locks described above apply to a file, that is something described by an FDB. In addition to these, each file has some number of OFNs, one for each file section that is in use. Therefore, a file may have up to 512 OFNs or file sections.

Each active OFN has an "access token" lock. The access token represents the ability of the system to access the data described by the OFN. The access token may be held in one of the following modes:

- place-holder
- read-only
- exclusive (read or write)

A read-only access token may be held by any number of systems simultaneously. An exclusive token is held by only one system. A "place-holder" access token is an artifact that permits the CFS systems to agree on the end-of-file correctly. It also has some ramifications for bit table access tokens that will be described later. Place-holder tokens are also an optimization to avoid reallocating tokens that have been "lost" to another system.

The file access token is the most fundamental CFS lock in that it is used not only to control simultaneous access to user files, but also to manage directories and bit tables.

The access token state transition table is given below, with the action required to make the designated state change

\ new / old \	read	exclusive	place-holder
read	nothing	vote	DDMP*
exclusive	DDMP**	nothing	DDMP*
place-holder	vote	vote	nothing

Where:

vote	means that the other CFS systems must be asked for permission to make the state transition. Voting is a fundamental operation of CFSSRV and is done by a software implemented broadcast.
DDMP*	means that DDMP must run and remove all of the OFN's pages from memory and update the disk copy of any modified pages.
DDMP**	means that DDMP must run to update to disk any modified pages and any in memory pages must be set to "read only". This latter operation is performed by clearing the CST write bit. The CST write bit has been implemented in KL paging explicitly to support loosely-coupled multi-processors.

While DDMP is performing a CFS-directed operation, all pages of the OFN are inaccessible to any other process. This is achieved by a bit, SPTFO, set in SPTO2 by DDMP.

Access permission to a file moves among the CFS systems on demand. Each system must remember its state of the token so it may respond to requests for the access permission.

The token consists of:

- The structure name
- the OFN disk address
- a flag bit to indicate this is the access token
- state
- end-of-file pointer
- end-of-file transaction number

- fairness timer
- the OFN this token is for

and, if this is a token for a bit table:

- structure free count
- structure free count transaction number

The fairness timer is a CFS service that allows a resource to be held on a node for a guaranteed interval. Therefore, the owner need not lock the resource and arrange to unlock it later. Rather it simply places the guarantee interval in the resource block and the CFS protocol takes care of the rest.

Place-holder tokens exist principally to hold the values associated with the end-of-file pointer and with the structure free count. It is important that these be held by each system, because the owner of the OFN token may crash and therefore the last known state of these quantities must be remembered so that the remaining nodes may have the best possible value for them. The transaction count is intended to determine whose value is the most recent should the owner not be present to contribute the current value. During the voting for acquiring a token, these values are passed among the CFS nodes, and the node conducting the vote retains the values associated with the largest transaction number.

The file access token represents the rights that a system has to access a file section. That is, the token is associated with the file's contents.

However, the owner of a file, i.e. the system holding exclusive rights to access the file, also has the right to modify the file's index block. The owning system may add pages to the file or delete pages from the file.

OFNs are treated specially in TOPS-20. Unlike the file's data pages, an OFN may not be discarded when the system gives up its access to the file and read from its home on the disk when the access is reacquired. An active index block, represented by an OFN, contains paging information that must be retained while the file is opened. For this reason, a system needs to be informed if the index block contents are changed by another system.

This information is disseminated in CFS by a broadcast message. Each time a system writes a changed index block to disk, it informs all of the other CFS systems by a broadcast message. Note that this broadcasting is done only when the changed index block is written to disk, and not each time the index block is modified. A broadcast message is used instead of including this in optional data with the access token for reasons explained in a later section of this document.

When a CFS system receives such a message, it sets a status bit in the appropriate OFN so that the next time a process attempts to reference the OFN the following will happen:

- the disk copy of the index block is examined.
- for each changed entry, update the local OFN

This reconciliation of the index block with the local OFN is accomplished by the routine DDXBI.

VOTING IN CFS

When a node needs to "upgrade" its access to a resource, including acquiring a new resource, it must poll each of the other CFS nodes. This is so because none of the CFS nodes is a master and therefore there is no a priori location for resolving access requests. CFS is not only a democracy, but somewhat of a cacophony.

Voting, then, requires "broadcasting" to each other node the required resource and access. Each node must respond with its permission or denial.

The CI does not support broadcast, and even if it did, it would not support a reliable broadcast. Therefore, CFS implements broadcasting by sending a message to each of the other nodes, one-at-a-time.

A vote request contains:

- function code
- resource "name" (seventy-two bits)
- access desired
- vote number

A reply contains:

- function code
- resource name (seventy-two bits)
- reply (yes, no or "qualified yes")
- vote number
- optional data

The message contains a function code because votes and replies are only one kind of CFS to CFS communication.

The vote number is used to insure that the reply is to the proper request. The requestor may "restart" a vote at any time. It does this be "canceling" the current vote, acquiring a new vote number, and broadcasting the new request. A vote number is a monotonically increasing, thirty-six bit quantity.

A vote will be restarted for one of the following reasons:

- a configuration change is reported by SCA
- a previous vote "times out".

The latter should rarely occur, and is likely indicative of a malfunctioning CFS on some other system. In some cases, a node will not reply if it is unable to acquire the appropriate space for constructing a message. There are a small number of cases where this is legal, and for these cases, the requestor must revote when appropriate.

When a reply is received, the vote number must match the number in the associated resource block.

The replies to a vote are:

- unconditional yes.

- no

- conditional yes.

- cancel yes condition

A conditional yes means that the respondent will approve the request, but it needs to perform a local housekeeping operation first. The most common form of this is voting for an access token where the respondent must first update the disk copy of the file, and perhaps flush all of its local copies of the file data. When the condition has been satisfied, a "condition satisfied" reply is sent.

Each resource has a "delay mask". This mask has a bit for each of the other CFS nodes, and whenever a node replies with "conditional yes", its bit is set in the resource's delay mask. Therefore, a process that is waiting for the conditions to be satisfied, simply examines the delay mask periodically and waits for all of the delay bits to be cleared. While any delay bits are set, the vote is considered to be still in progress, and therefore any configuration change will require restarting the vote.

Conditional yes votes, and the associated delay mask, are provided to eliminate the need for nodes to reply "no" when there are temporary conditions preventing the approval of the request. The overhead required to process such replies, and to wait for them, is offset by the gains in not having to revote in the face of such conditions.

CFS provides the following basic voting services:

- Acquire a resource. If the resource is known on this node, but the current state conflicts with the request, the currently held resource is released and a vote is taken.

  This service is called specifying either "retry until successful", or return after one try.

- Upgrade a resource. This service tries only once. It also guarantees that the currently held resource will not be released. In fact, the resource may be held and "locked" locally when "upgrade" is requested.

- Acquire local resource. This is used for resources not shared by other CFS nodes, but managed by CFS. Examples are directory locks on exclusive structures.

A vote is started by the routine VOTEW. Ordinarily, one does not call this routine directly, but rather one requests a resource, and if necessary, VOTEW will be called to conduct a vote.

VOTEW always waits for the vote results. The results are tallied at interrupt level by noting the number of replies received in the associated resource block. VOTEW periodically examines the resource block testing for:

- all tallies received
- a "no" vote recorded
- a configuration change

The actions taken are as follows:

- configuration change: restart the vote
- a "no" vote: return to the called
- all tallies received:
  - if no "conditional yes" votes, return to caller
  - If one or more "conditional yes" votes, wait for the "condition satisified" replies. While waiting, a configuration change could occur, requiring the vote to be restarted.

RESOURCE ACQUISITION AND UPDATING

CFS resources are acquired and changed in response to requests from other parts of the monitor. Rather than describe each one, it will be instructive to consider how the file related resources are acquired, maintained, and destroyed.

When a file is opened, and the first OFN is created, ASOFN will create the static CFS resources: open type and, if appropriate, the frozen writer token.

Anytime an OFN is created, be it in response to opening the file, or one of the "long file" OFNs, ASOFN will create the access token.

The access token state is verified by various of the file system and memory management routines. The most common place for this is in the page fault handler. The two exceptions to this are for a bit table access token and a long file "super index block". The bit table token is acquired and "locked" when the bit table lock is locked and released only when the bit table is unlocked. The token for a super index block is occasionally acquired in DISC by the routine (NEWLFT) that creates new long file index blocks. In theory, these exception cases need not be exceptions. That is, the code could simply rely on the normal management of the token during page faults to insure data integrity. However, in these cases, the code must perform multiple operations on the file data "atomically". That is, it must modify two or more pages, or it must "test and set" a location with the assurance that no other accesses to the data occur between the steps. On a single system, this is done by a NOSKED to prevent any other process from running. In an LCS environment, NOSKED is not sufficient (although it is necessary!). Another form of interlock must be used to prevent a process on another system from examining or modifying the data. It turns out that the access token satisfies this need quite well.

The above discussion implies that the page fault handler, when it acquires an access token for an OFN, does not "lock" the token on the system. That is, the token is acquired but not "held". This may result in the token being preempted by another system before the process is able to reexecute the instruction that caused the page fault. The "fairness" timer in the token resource is one attempt to minimize such thrashing.

The access token is acquired on the following conditions:

- when an OFN is being created

- when the OFN is locked

- when a page fault occurs because the current access is not correct

The current state of the token is kept in the CFS resource block as well as in the OFN data base. The field, SPTST, is the current OFN state of an OFN. The values are:

0 => no access

.SPSRD => read only

.SPSWR => read/write

SPTST is modified by the routines in CFSSRV that are called to set the state of the file. The values are set here, and not in PAGEM, PAGFIL or PAGUTL because the OFN state must be set while the CFS resource block is interlocked against change.

The routines to modify the state of an OFN token are:

- CFSAWT - acquire token but don't hold it

- CFSAWP - acquire token and hold it

TOKEN MANAGEMENT

Once a token is "owned" on a system, it will remain in that state until it is required on another system. That is, if the token is held for read/write access (exclusive), then all references to the pages of the OFN will succeed without CFSSRV being invoked.

If a token must be revoked because another system needs it, CFSSRV signals DDMP to process the data pages. This is done by:

- Setting bits in the field STPSR in the OFN data base.

- Setting the OFN's bit in the bit mask OFNCFS.

• Waking up DDMP.

The field STPSR is a two-bit quantity indicating the type of access required by the requesting system. DDMP's action is as follows:

read-only needed:

> Write all modified pages to the disk. Clear all of the CST write bits in all in-memory pages.

read/write needed:

> Write all modified pages to disk. Flush all "local" copies of data including any copies on the swapping space. Swap out the OFN page if it is in memory (actually, simply place it on RPLQ).

Once DDMP has performed the necessary operation, it calls CFSFOD. This routine will set the OFN state and the resource state appropriately as follows:

read-only requested:

> set OFN state to .SPSRD and set resource state to "read".

read/write requested:

> set OFN state to 0 and set resource state to "place-holder".

CFSFOD also copies the current end-of-file information from OFNLEN into the resource block and finally it sends the "condition satisfied" message to the requestor.

While DDMP is performing its work on behalf of CFS, it sets the bit SPTFO in the OFN data base. This bit is examined by the page fault handler, and by CFSAWP/CFSAWT to see if the OFN is in a transition state. If SPTFO is set, and the process requiring the OFN is not DDMP, then the process is blocked until SPTFO is cleared by DDMP. In order to facilitate identifying DDMP from all other processes, a new word has been added to the PSB called DDPFRK. If DDPFRK is non-zero, then the current process is indeed DDMP and SPTFO should be ignored.

UNUSED RESOURCES

Whenever a node replies "no" to a request, it remembers in the associated resource block the node(s) that have been rejected. The only reason for unconditionally denying a request is that the resource is "held" locally. If a resource cannot be granted because of the fairness timer, the "no" response includes an optional data word of the time the resource is to be held. Therefore, the requestor knows precisely when to request the resource anew.

When a held resource is "released" (or undeclared), CFS examines the rejection mask for the resource. For each node identified in the mask, a "resource released" message is sent indicating that this is a propitious time to try to acquire the resource. There is no guarantee the new request will be granted as the resource could be held again, or another node could have requested, and been granted, the resource first.

DELETING FILE RESOURCES

The access token is deleted whenever the associated OFN is deassigned.

The static file resources are released when the file is closed. This is performed in RELOFN.

CHANGES TO EXISTING CONCURRENCY CONTROL SCHEMES

As a result of CFS, much of the concurrency control in TOPS-20 has become distributed. In some cases, this has been done by creating a companion resource to an already existing one. As example of this is the file open mode resource described above.

In other cases, existing locks have been replaced by CFS resources.

The decision as to which technique to employ was made on a case-by-case basis. The significant criterion was how easy it was to eliminate the existing concurrency control and replace it with the CFS management. The file resources proved difficult to do. However, there are two important pieces of the monitor's structure that were easily and efficiently replaced: directory locks and directory allocation tables.

Directory locks are now CFS resources. A directory lock resource contains:

- the seventy-two bit identifier
- owning fork
- access type
- share count
- waiting fork bit table

In fact, a directory lock resource is the sole instance of a "CFS long block".

Directory locks are always acquired for exclusive use. However, unlike file access tokens, directory locks are never granted "conditionally". This is because directories are files, and the directory contents are subject to negotiation by the associated file access token. That is, acquiring exclusive use of the directory lock resource is independent of acquiring permission to read or write the directory contents. When some process on the owning system attempts to read or write the directory contents, it must first acquire the file access token in the proper state. Although this sounds somewhat inefficient, i.e. requiring the node to acquire two independent resources, it is in fact a remarkably efficient adaptation of the CFS resource scheme. This is so because a node need not know how the directory contents will be used when it acquires the directory lock. That is the way the lock was handled before CFS, and preserving this convention means that the code to acquire the directory lock under CFS is as efficient as possible. The state of the file access token, and consequently the degree of sharing of the directory contents, is determined by how the contents are referenced and not by how the directory is locked.

This means that a process may lock the directory lock without knowing how it will reference the associated data, and its reference patterns determine what other negotiations are required.

The directory allocation table is a local "cache" for the information normally stored in the directory. Each active OFN is associated with a directory allocation entry. Each entry is for exactly one directory. The entry, before CFS, contained: structure number, directory number, share count, and remaining allocation.

Under CFS, an active allocation entry contains: structure number, directory number, share count, and pointer to the CFS resource block. The CFS resource block contains, besides the normal CFS control information, the remaining allocation for the directory and a transaction number. The transaction number serves the same purpose as the transaction number associated with a file end-of-file pointer.

CFS may have an "unused" resource block for a directory allocation entry. That is, even though there is no active directory allocation entry, there may be a CFS resource block representing the directory. This is because CFS attempts to retain knowledge of resources for as long as possible to avoid having to vote when some process wishes to create the resource anew. However, CFS will destroy any unused resource allocation entry that is requested by another system.

TRANSACTION NUMBER

The optional data items, "end-of-file pointer" and "structure free space", have an associated value called the "transaction number".

One either uses centralized or decentralized control in a "loosely-coupled multiprocessor" system. In a centralized system, control information and updating is coordinated by a master. Transactions are "serialized" by virtue of having a single owner for the resoruce and therefore a single manager of the resource data. In a decentralized system, the various systems share the ownership of resources and use some sort of "concurrency control" technique to manage resources.

CFS is a decentralized system. A resource is not owned or managed by any particular system, but rather the responsibility for the resource is passed from system to system as required. As such, it may not always be possible to uniquely identify a particular system as the owner. This may cause a problem when a system needs to become the owner, and therefore must determine the current status of the resource in question.

There are two possibilites that a nascent owner may encounter:

- The previous owner is present and indentifiable.

- There is no system that is the previous owner

and of this latter case:

- the existing control information is accurate

- the existing control information in not accurate.

Clearly, if the previous owner is present, the new owner has all of the information it needs to proceed with its transaction.

If the previous owner cannot be identified, then the new owner must be able to determine which of the systems has the current control information about the resource. It may be that none of them has, and this is a problem that exists even on a single-processor system. The result of such a problem may be "lost pages", inconsistent data bases and other such phenomena. As in a single-processor system, the problem occurs because the resource control information is lost as an effect of a system crashing.

In order to determine the most up-to-date information about a resource, each system maintains a transaction count along with the information. Whenever it acquires information with a larger transaction count than its own value, it knows that information is more current and it must replace its own copy with the new data and count. Whenever a system unilaterally changes its copy of the control information, it must also increment the associated transaction count. Since a system may perform such an update only when it has write or exclusive access to the resource, the system need change the transaction count only when it must downgrade its access.

Due to the nature of the CFS voting and resource management, it is possible for a system to acquire a resource but to receive a different value for the resource control information from each of the other systems (this will happen only if the owner crashed. If the owner didn't crash, then at least two of the other systems must have the same control information and transaction count). In this case, the transaction counts are used to identify the most up-to-date value.

The transaction count is really a "clock" that is used to "time-stamp" information. When systems communciate with one-another, they synchronize the clocks by sending each other the current counts. Most network concurrency schemes use clocks for similar purposes, and most of the uses and implementations are considerably more exotic than this one. However, since CFS needs the clock only to determine relative ages, and not absolute ages, of information, this simplified clock is adequate.

An alternative to using transaction counts is to "broadcast" changes to resources. This has the disadvantage that it is costly in both processor and communications time and resources. However, CFS does use broadcasting in a few cases where the lack of up-to-date information could result in data being destroyed. The two cases are:

- an OFN being modified and written to disk

- an EOF value being written into the directory copy on the disk

As both of these represent changes in the permanent copy of the resource, it is essential that all of the other systems have current copies or knowledge of the update.

CFS MESSAGE SUMMARY

Items marked with a "*" are sent as broadcast messages.

*1. request resource (vote)

2. reply to request:

a. unconditional yes

    b. unconditional no

    c. no with retry time

    d. conditional yes

3. resource available

4. condition satisfied

*5. OFN updated

*6. EOF changed

In addition, each message type may carry specific optional data items, up to four words of optional data per message.