FILE CACHE

Introduction

The File Cache package implements an automatic, fully transparent file-caching mechanism. Files from remote servers are automatically copied to a specified cache directory when they are first opened, and all future accessing of these files will use the cached version. This allows random access to files even if the files are stored on a server that does not implement random access. Creating a remote file is also done locally, and when the completed file is closed it is (eventually) written out to the remote host.

The file cacher can also speed up many common interactions with Lisp, especially if you access files on heavily loaded file servers. Editing functions are quicker (once the source file is cached, loading the source for a function does not need to go out to the server), and TEdit response is improved.

The file cacher also attempts to deal intelligently with file servers that are not responding. If a server is down, you can still use cached files from that server.

Before using the file cacher package, you should read this document carefully. It contains a number of important warnings that can save you much grief.

Note: The file cacher changes the operation of all file operations from all packages in Lisp. In some cases, the file cachers' actions are not transparent, and inappropriate recovery methods may lead to loss of work. Before using the file cacher package, please read this document carefully. Before attempting to use the scavenger or recover from a crash, please review the relevant sections.

How Does It Work?

The system uses the function \GETDEVICEFROMNAME to determine what device to use to perform I/O operations for a given file. The cacher provides a modified version of this function that returns a specialized cache device when appropriate.

The import of this is that the cacher is transparent. Once you start it running, you shouldn't need to do anything special.

Where Are Files Cached?

The location of the file cache is determined by a "cache prefix." When the file cacher is started up, it looks for the file {DSK}FCache.Pointer;1. If this file is found, a single expression is read from the file and is used to set the cache prefix. If there is no such file, the user is prompted for the cache prefix in a special prompt window. The response will be written out to the appropriate file so that later systems will find this pointer. The default value depends on the machine type: on an 1108 or 1185/86 workstation, the default is {DSK}<LispFiles>Cache>, on 1132s the default is {DSK1}. Setting the cache prefix to NIL disables the cache.

It is possible to use {CORE} as the cache prefix. This results in fast access time to files at the cost of expanding your virtual memory size and slowing down your system due to increased swapping. If you will be making repeated access to certain files, have gobs of relatively unused physical memory, and don't mind losing your cache across different sysouts, you may want to use {CORE} as your cache prefix.

The cached copy of a remote file is given the same name and extension as the original file. For example, if the cache prefix is {DSK}<LispFiles>Cache>, the file {Phylum}<Lanning>Lisp>Init.DCOM will be cached to the file {DSK}<LispFiles>Cache>Init.DCOM. Characters in the file name that are not legal in file names for the local disk are replaced by $'s.

Maintaining the File Cache Across Sysouts

The directory of the cache is maintained in a number of special files on the cache. The file FCache.Listing in the cache contains a complete description of those files in the cache that are copies of files from a remote host. This file is updated before each logout so that other sysouts can determine the cache contents. While a system is running, a log file (FCache.Log;1 on the cache prefix), is continually updated with the changes to the cache. In the event of a system failure, this file contains enough information to reconstruct the state of the cache, including descriptions of files that need to be dumped to a remote host. Starting the cacher in a new sysout will automatically load the FCache.Listing;1 and FCache.Log;1 files, thereby restoring the state of the cache. Dirty files, of course, will still need to be written out.

Restoring the cache can take a while, especially if there is a large number of files. To let you know what it is doing, the cacher will print a "." in the prompt window for each entry that it has successfully restored. If a cache entry is bad (say the local file has been deleted behind the cacher's back), the cacher will print a "?." This printing is controlled by the SILENT cache property described below.

Starting and Stopping the File Cacher

When the file cacher is loaded, it notes the latest cache prefix (from the file {DSK}FCache.pointer;1) and restores the cache listing from that prefix. File caching then needs to be enabled for each file server. This can be accomplished in either of two ways.

Some file devices are created when you first reference them. These devices are created by a "generic" device that knows how to create the appropriate device. For example, the LEAF device is responsible for creating the file devices for Phylum and other PARC "interim file servers," and the NSFILING device creates devices for Phylex:PARC:Xerox and other NS hosts. The file cacher provides a way to create a cache device for these file devices.

(FCACHE.GENERIC.START *deviceType*) [Function]

Enables caching for any device of type *deviceType*. When a new file device of type *deviceType* is first referenced, caching will be enabled for that device by a call to FCACHE.START (but see the cache property BADDEVICES described below). FCACHE.GENERIC.START is UNDOable. The "standard" device types are LEAF and NSFILING.

(FCACHE.GENERIC.STOP *deviceType*) [Function]

Undoes the effect of a call to (FCACHE.GENERIC.START *deviceType*). FCACHE.GENERIC.STOP does not turn off caching for devices that were automatically FCACHE.STARTed.

FCACHE.GENERIC.DEVICE.TYPES [Global Variable]

When the cacher is first loaded, the function FCACHE.GENERIC.START is mapped over the value of FCACHE.GENERIC.DEVICE.TYPES.

FCACHE.GENERIC.DEVICE.TYPES is defined by an INITVARS expression in the file, so you can override the default value before loading the file cacher. The default value is (LEAF NSFILING). This effectively turns on caching for all LEAF and NS hosts.

Some devices are part of the system, so FCACHE.GENERIC.START cannot be used to start caching on them. Instead, the following

function can be used to enable caching on a device-by-device basis.

(FCACHE.START *hostName*) [Function]

Turns on caching for the specified host device. The host does not have to be "up" to turn on caching. FCACHE.START is UNDOable. As a side effect, FCACHE.START will remove *hostName* from the cache property BADDEVICES defined below. FCACHE.START returns NIL if caching could not be turned on for the device, otherwise it returns something non-NIL. NOTE: The file cacher does not support caching from UNIX hosts, so FCACHE.START of a UNIX host will do nothing.

(FCACHE.STOP *hostName*) [Function]

Turns off caching for the specified host device. FCACHE.STOP is UNDOable. As a side effect, FCACHE.STOP will add *hostName* to the cache property BADDEVICES defined below.

(FCACHE.VANQUISH) [Function]

Removes the cacher from the system, destroying all evidence that it ever existed (well, almost all).

Cacher Tuning

The functioning of the cache is influenced by a number of properties. These property values can be accessed by the following two functions.

(FCACHE.GETPROP *name*) [Function]

Returns the current value of the cache property *name*. FCACHE.GETPROP has the appropriate SETFN property, so CHANGETRAN expressions like (push (FCACHE.GETPROP 'BADDEVICES) 'ERIS) will work.

(FCACHE.PUTPROP *name value*) [Function]

Sets the value of the cache property *name* to *value*.

PREFIX [Cache Property]

The PREFIX property determines where files are cached. Changing this property will dump all dirty files, update the FCache.Listing and FCache.Log files, update the FCache.Pointer file, set up the new prefix as the cache, and load its FCache.Listing and FCache.Log files. Setting the prefix to NIL will completely disable the file cacher. It is initially set when the package is loaded, and is updated automatically (using the file {DSK}FCache.Pointer) after each logout.

ENTRIES [Cache Property]

The number of files currently cached. This property is read-only.

SIZE [Cache Property]

The SIZE property is the current number of disk pages used by the cache. This property is read-only.

FASTDIR [Cache Property]

Directory enumeration is complicated by the cache: files that are present in the cache but are not yet dumped to the server should be included in the directory. This is accomplished by merging the list of matching cached files with the directory list from the remote server. Generating this list of matching local files can be rather slow, especially if you have many files in the cache. The FASTDIR cache property lets you get around this problem. If FASTDIR is NIL then the directory merging is done, otherwise the merging is disabled. Disabling the merging of cache file listings and remote directory listings can result in a significant speedup (depending on the number of files in your cache), but can produce directory listings that miss some files (those files that have not yet been written to the server). Initial value is NIL. Note that FASTDIR has no effect on INFILEP or OUTFILEP.

SILENT [Cache Property]

Normally, the cache writes informative messages to the prompt window whenever files are being loaded into the cache or dumped from the cache to a remote host (and a few other situations). If the SILENT property is non-NIL, this reporting is not done. The default value is NIL, i.e., print out the messages.

Dealing With Unresponsive Hosts

The file cacher tries to deal in an intelligent way with hosts that are not responding.

UNSAFE [Cache Property]

If the File Cacher believes a remote host is not available (according to a heuristic algorithim), it uses the UNSAFE property to determine how to proceed. If the value is NIL, the cacher just keeps going (probably generating some sort of error). If the value is ASK, you will be asked if you want to trust the cache. If the value is NOTIFY, the cacher will automatically trust the cache, but will notify you of its action in the main prompt window. If the value is T, the cacher will also trust the cache, but will give no indication of this. The default value of the UNSAFE property is ASK.

"Trust" is dependent on what you are trying to do with the host. If you are generating a directory listing or performing an INFILEP or OUTFILEP, "trust" means to use only those matching files that have been cached. If you are attempting to verify that a given cache entry is a valid copy of a remote file, it means to believe that it is.

Making Room in the Cache

The local file cache consumes a finite resource, space on the local disk. When there is insufficient room on the disk for a new cache file, old cache files must be deleted to make room. Files are selected for deletion based on a modified Least Recently Used (LRU) algorithm. These modifications are described below.

MAXSIZE [Cache Property]

The MAXSIZE property is the maximum number of disk pages to be used by the cache. When this limit is reached, old cache files are deleted to make room for new files, even if there is room left on the disk. Note that the file cacher also checks DISKFREEPAGES to determine if there is enough room for a file (on those devices where DISKFREEPAGES makes sense).

Setting MAXSIZE to a very large number will cause the cache to consume as much space as available in the area defined by the cache prefix. This may be the desired functionality when an entire partition or logical volume is devoted to the cache. MAXSIZE is intended for situations where the cache is a local disk volume or partition which is shared with other file uses.

The value of the MAXSIZE cache property is an attribute of the cache prefix, so it is saved on the FCache.Listing file and restored when the cache is restored. The default value is 10,000.

KEEPVERSIONS [Cache Property]

When you add a new version of a file to the cache, older versions of the file become less valuable, and can be deleted from the cache. KEEPVERSIONS specifies the number of old versions of a file that should be kept in the cache. If KEEPVERSIONS is zero or NIL, all old versions are kept. Otherwise, versions that are KEEPVERSIONS older than the new file are deleted. The default value of KEEPVERSIONS is two.

Actually, old versions are not deleted right away. Instead, they are moved to the tail end of the cache list. This will cause the old files to be removed from the cache as soon as the cacher needs the space.

Some users think that they know better than the cacher what files should be deleted from the cache. The following function is documented to provide a hook for the strong of will. It can be advised or redefined as you like.

(\FCACHE.MAKEROOM.DELETEABLE? *fileName*) [Function]

Is used to determine if a file can be deleted from the cache to make room for a new file. \FCACHE.MAKEROOM.DELETEABLE? always returns T.

Forcing Direct Access to Remote Files

There are a number of ways that you can bypass the cache.

DON'TCACHE [OPENFILE Parameter]

If the atom DON'TCACHE is present in the PARAMETERS argument to OPENFILE (e.g., if the atom DON'TCACHE is in the PARAMETERS list, or if the list contains the list (DON'TCACHE T)), the file will not be opened on the cache. In the Koto and later releases of Interlisp, the COPYFILE function uses this to prevent caching files when you are copying them.

BADDEVICES [Cache Property]

Provides a way to prevent the cacher from being automatically started for specific hosts. BADDEVICES is a list of device names that will not be FCACHE.STARTed automatically. See FCACHE.GLOBAL.START and FCACHE.STOP above.

MAXFILEPAGES [Cache Property]

The MAXFILEPAGES property gives the maximum size (in pages) of a file that should be cached. Files larger then this are not cached. A value of NIL means there is no upper limit on the size of cached files. The default value is 400. Like the MAXSIZE cache property, the MAXFILEPAGES cache property is stored along with the cache listing, so its value is remembered across sysouts. It should be noticed that it is possible for files to grow larger than the MAXFILEPAGES limit as you write to them.

USECACHE [Cache Property]

The USECACHE property specifies what I/O operations will use the cache. Possible values are T, Read, Write, and NIL. A value of T means that the cache will be used for both input and output operations; Read means that the cache will be used for input only; Write means the cache will be used for output only, and NIL means the cache will not be used at all.

(ADD.FILE.TO.CACHE? *fullName*) [Function]

Is used to determine if a file should be cached. If *fullName* matches any of the file specs on the global list DON'T.CACHE.FILES, ADD.FILE.TO.CACHE? returns NIL, otherwise T. File specs on the list DON'T.CACHE.FILES can contain the standard wild card characters. The host name and version number fields default to "*"; the others default to the empty string "". This function is separated out so you can advise or redefine it for specialized applications.

DON'T.CACHE.FILES [Global Variable]

A list of file specs, used by ADD.FILE.TO.CACHE? above. The default value is NIL.

(WITHOUT.FCACHE *form1 ... formN*) [Macro]

Evaluates the (unevaluated) forms in a context where caching is disabled. For example, (WITHOUT.FCACHE (TCOMPL 'Foo)) will

compile the file Foo without going through the cache. Note: WITHOUT.FCACHE operates by changing the cache property USECACHE. This is a global property, so if WITHOUT.FCACHE is being run in one process, all processes are running with caching disabled.

Private Files and the Cache

Private files present a problem to the cacher. Typically, the local disk on an 1108 is a public area, so files left there by the cacher are accessible to the public. The cacher attempts to get around this problem by deleting private files from the cache before each logout.

(FCACHE.PRIVATE.FILE? *fullName*) [Function]

Is used to determine if a file should be deleted from the cache when you log out. This is an attempt to solve the problem of having "private" files copied to a public machine. FCACHE.PRIVATE.FILE? uses the global variable PRIVATE.FILES described below. Files that match any file spec on PRIVATE.FILES will be deleted automatically when you log out. This function is separated out so that you can redefine it if you want. Note that if you exit a sysout by some other method (e.g., a crash), the private files will not be deleted.

PRIVATE.FILES [Global Variable]

A list of file specs, used to determine what files should not be left on the local disk when you log out. The default value is NIL. You may want to set it to (*.MAIL), or the value of (LIST (PACK* LOGINHOST/DIR "*")).

Caching Files for Output

Files that are cached when opened for output present a number of small problems.

- When should dirty files be written out to the host?

Turning off the cacher, changing the cache prefix, or logging out causes all (non-open) dirty files to be dumped out to their appropriate servers. Dirty files are also written out at irregular intervals by the following background process. The file cacher uses the term *dump* to mean write out a dirty file to the file server.

DUMP-FCACHE [Process]

A background process that wakes up every now and then and looks for dirty files. Dirty files that have been left idle for long

enough are dumped to the appropriate hosts. Note that only files that are not open can be written out.

DUMPSLEEP [Cache Property]

The number of seconds that the DUMP-FCACHE process (see above) waits before it checks for new dirty files. The default value is 10 seconds.

DUMPIDLE [Cache Property]

When the DUMP-FCACHE process wakes up, it checks each dirty file to see how long it has been since it was last closed. If this time is greater than DUMPIDLE seconds, the file will be dumped. Note that dirty files that are still open are not dumped. The default value is 20 seconds.

- Should file locking be implemented, and if so, how?

When a file is opened for output on a server (without the cacher being involved), the file cannot be opened by another user. This prevents the user from attempting to read an incomplete or incorrect file. Unfortunately, this functionality cannot be provided when the cache is used. Instead, when a file is dumped the file cacher makes sure that there have been no changes to that file on the host. If there have (for example, if someone else has created a file with the same version number that did not exist when you first created your cached file), the cacher generates a break with an appropriate error message. Generally, you should do a "RETURN '*fileName*," where *fileName* is where the file should be written out to. RETURNing NIL means to use the original file name. See the section on file cache errors below.

- What happens to open files at LOGOUT?

The cacher does not know what to do if you try to log out when there are dirty files in the cache that are open. It punts, generating a break. Open files that are not dirty are handled properly. Again, see the section on file cache errors below.

- What happens if your machine crashes before the completed file is dumped to the remote host?

If your machine crashes for any reason (say a power failure), dirty files will not be written out to remote hosts, and the FCache.Listing file will not be dumped. The FCache.Log files provide enough information to reconstruct the state of your cache. You should start up a new system running the file cache package. This will load the FCache.Log file. The cacher will then dump the dirty files automatically.

- What should be done if the local device runs out of space while attempting to write a cache file?

The cacher catches this error and tries to delete old files from the cache to make room. If it succeeds, everything continues as it was and you should not notice that anything has happened.

- What happens if there is not enough room on the server for the new file?

When the file is created on the cache, no check is made on the server to determine if there is room. Thus, you can (temporarily) exceed your allocated file space. When the cacher gets around to dumping the file, though, you will get the "standard" error. See the section on file cache errors below.

Cache Access From Code

In addition to the cache properties given above, there are a number of semipublic functions provided for access to the cache.

(CACHE.FILE *fileName*) [Function]

Will load the file into the cache. Returns the full file name if the file was successfully loaded into the cache, otherwise NIL.

(FCACHE.DUMP.ALL) [Function]

Writes out all dirty files that are not currently open.

(FCACHE.CACHELIST) [Function]

Returns a list of files that are currently cached. This may include files from hosts where the cache is not currently enabled.

(FCACHE.DIRTY? *fileName*) [Function]

Returns T if the file *fileName* is currently cached and needs to be dumped, otherwise returns NIL.

CACHEDIRTY [File Property]

(GETFILEINFO *fileName* 'CACHEDIRTY) returns the same value as (FCACHE.DIRTY? *fileName*).

CACHEFILE [File Property]

(GETFILEINFO *fileName* 'CACHEFILE) will return the name of the local cache file for *fileName*. If *fileName* is not cached, the value is NIL.

(FCACHE.DUMP.FILE *fileName*) [Function]

Writes out the file *fileName* to the appropriate host if *fileName* is dirty.

(FCACHE.DUMP *onlyIfChanged*) [Function]

Updates the FCache.Listing file and clears the FCache.Log file after writing out all dirty files. The FCache.Listing file will be rewritten if there has been any change to the cache list, or if the argument *onlyIfChanged* is NIL. There is a BackgroundMenu item DumpCache that does an (FCACHE.DUMP).

Cache Scavenging

In case the cacher loses track of files in the cache, you can use the following scavenger function.

(FCACHE.SCAVENGE *options*) [Function]

Returns a list of all files that match the cache prefix, but are not known to the cacher. FCACHE.SCAVENGE attempts to make sure that the internal state of the cacher is correct. The argument *options* is a single option or a list of options to the scavenger. Possible options are described below. Note that the scavenger temporarily turns off caching.

SILENT [Scavenger Option]

If the SILENT option is not present, the scavenger will print out messages to the default window to let you know what it is doing.

EXISTS [Scavenger Option]

Each cache entry is checked to make sure that the local file exists. If it does not, the cache entry is deleted. This option is useful if you have deleted cached files directly from the local disk.

VERIFY [Scavenger Option]

Each cache entry is checked to make sure that it is a valid cache for its remote file. If it is not, the entry is deleted. This may entail a rather long delay as there is a lot of communication with remote servers.

REPAIR [Scavenger Option]

For each file that would otherwise be returned by FCACHE.SCAVENGE, tries to find the remote file that it is a copy of. If a file is found, the corresponding entry is added to the cache list. The result of FCACHE.SCAVENGE will not include any files that have been "repaired."

Finding a remote file that is a copy of the local file is a very heuristic process. The following rules guide the search for a matching file:

- If the file is a LISP source file, the first expression on the file is a FILECREATED expression that contains a reference to the file name. Note: this reference may not point to the correct file, because the original file may have been copied someplace else.

- Similarly, if the extension is DCOM, then the file is checked to see if it is a compiled source file. Again, a FILECREATED expression in the beginning of the file contains a reference to the original file. Actually, the reference is to the source file, but this is used to get the directory and base file name to look for.

- If the file is a font file, then the appropriate directories are searched for a matching file.

- If all else fails, the directories on DIRECTORIES are searched for a matching file.

FCACHE.SCAVENGE.IGNORE [Global Variable]

A list that associates (yes, in ASSOC format) machine types with file names that should not be included in the result of (FCACHE.SCAVENGE). The file names include name, extension, and version fields, but do not include host or directory name fields. This variable is provided for users who cache other files (sysouts, for example) on the cache directory. Files matching an entry on FCACHE.SCAVENGE.IGNORE will *not* be repaired.

Cache Inspection

The DUMP-FCACHE process provides a nice way of interactively inspecting the cache. Selecting the DUMP-FCACHE process in the PSW and bugging the Info menu item will bring up an inspect window, viewing the current cache properties. You can use this inspect window to set the cache properties.

The cache property inspect window can also be accessed by a subitem of the DumpCache item from the Background Menu.

You can inspect the list of cache entries by a menu option in the title of cache property inspect window. This new inspect window can be used to selectively dump cache entries or delete unwanted files from the cache.

Hints

You can speed up access to fonts by the following trick: after the cache starts and defines the cache prefix, do a (push DISPLAYFONTDIRECTORIES (FCACHE.GETPROP (QUOTE PREFIX))). Once this is done, any font that is already cached is found directly, without having to perform any directory lookup on the remote host. Font files that are not already cached are not found on the local disk, so they will get cached and be found quickly the next time.

There is an interesting side effect to this: since subsequent access to a font file does not go through the cache, the entry for the font file gradually falls to the bottom of the cache list, and when the cacher needs to make space, it will delete the font files from the cache. Thus, even though you don't go through the cache verification mechanism, the font files can't be out of date for very long.

However, there is a potential problem with this. If you push, say, {DSK18} onto DISPLAYFONTDIRECTORIES and then make a sysout and try to run the sysout on a machine with only five

partitions (or on an 1108), it will die a terrible death when it first looks for a font. Beware.

Known Deficiencies

The local disk may not understand certain file properties (like CREATEDBY) that the remote host does. Attempts to perform a SETFILEINFO on these properties on a dirty file in the cache may not produce the results you would expect. Similarly, GETFILEINFO on a file that has not yet been dumped may return NIL even though the host supports the given file property.

With the file cache, it is possible to create a file whose file name is not legal for that server (e.g., (OPENFILE '{PHYLUM}<Lanning>Foo% bar 'OUTPUT) works). When the cacher attempts to dump the file, however, an error occurs. See the section on cacher errors below.

It is also possible to create files on directories where you do not have write access. Only when the cacher attempts to dump the file will an error occur.

The private file mechanism described above is not foolproof. If you don't go through the standard log out procedure, the private files will not be deleted from the local disk.

As was mentioned above, file locking is not implemented. This can be rather confusing if you switch rapidly between machines.

The file cacher will not work with TCP servers. This is because TCP does not support any file attributes, so the cacher has no way to verify a cache file.

The file cacher will not work with UNIX hosts. This is because of vast amounts of confusion caused by mixed-case file names and UNIX's lack of file version numbers.

Dealing With File Cache Errors

- If you should try to log out and the cacher fails in an attempt to write out a dirty file, it will generate a break. This is most commonly caused by the (cached) file already being open. You have two options: you can continue from the break, or you can exit the break with an ↑. If you continue, the logout will proceed, but the file will not be written out. This is safe in that the cacher in the next system will notice that the file needs to be written out. It is potentially confusing in that you may think that you dumped a file out, but it never really got to the server. It is recommend that you close the file (if that is what kept it from being written out), ↑ out of the break, and try to log out again.

• When a file is being written to the cache, the file cacher does not check to make sure that you can really write the remote file. For example, the cacher does not check to see if there is room on the server for the file, or if the file name contains any illegal characters, or if you have write access to the host/directory. Instead, the cacher will go into a break when it tries to dump the file. A message will be printed into the break window telling you how to recover from this error. From this break you can clean up the situation.

If the file name contained an illegal character, or if you don't have access to the host/directory, you can RETURN the "correct" name for the file. The file name you return need not be a complete file name—missing fields in the name will be filled in from the original file name. RETURNing NIL or exiting the break with an OK will cause the cacher to try the original name again.

If instead the problem was lack of space on the server, you can delete some files and continue from the break via OK or RETURN.

• When the cacher is going to dump a file back to a server, it checks to make sure that no new file has appeared if there was no original file on the server. Similarly, if there was an original version of the file, the cacher checks to make sure that the file has not changed since the cached version was created. If the cacher detects a problem, it will go into a break, printing a message telling you how you can recover.

If you think that the cacher is just confused, you can safely RETURN NIL from these errors. If there is a real problem (say between the time you created file Foo.baz;3 and the time the cacher tried to dump it out, somebody else wrote a version of Foo.baz;3), you should RETURN the new file name that you want to write out.

• Sometimes when a file is being written out, it will break with a message that the file is "open in conflicting ways." You can safely ↑ out of this error. If you find that it is happening a lot, try increasing the value of the DUMPIDLE cache property. The cacher attempts to detect this error, and should automatically try again after a short delay.

• There is an occasional situation where a BSP stream operation breaks, calling (SHOULDNT). Maybe it shouldn't, but it does. Anyway, I have found that if you revert to a suitable place up on the stack and try again, things will work. There seems to be a missing monitor lock or something. I dunno.

General Warnings

The cacher cannot dump files that are open. The implications of this are sometimes profound. For example, when you do a Put in TEdit, a new file is opened for output, written, closed, *and then opened for input*. This prevents the file from being dumped.

Further, shrinking the TEdit window to its icon does not close the file. The moral of this is that files that you think are "safe" (written out to the server) are sometimes not. If you often switch between different machines, this can be a real problem. You are warned.

When you log out, the cacher leaves a file, FCache.Listing, on the local cache device that describes the current cache entries. This file is automatically updated each time you do a LOGOUT. *Do not be tempted to alter this file.* Similarly, do not alter the file FCache.Log.

Not all remote-protocol devices are supported. The file cacher does not work with UNIX hosts, or with TCP. (The TCP protocol has no way of finding out the creation date of remote files in order to see if they are in fact the same as local ones.)

There is currently no interlock between files that are accessed through the cache, and those that are accessed directly, e.g., by use of the DON'T.CACHE property in OPENFILEs. This means that two separate processes might get conflicting versions of a file, e.g., if one process is writing a new file with CopyFiles (which uses the DON'T.CACHE property) and another is attempting to write the file into the cache. The result will be that the cacher will complain when it finally goes to write out the version that was written locally.

Finally, a general warning. The file cacher can make your life much easier, but it adds a level of complexity to the file system, and, as it is a new package, it still has a few bugs. As any work you do is likely to depend on the integrity of the file system, failures in the cacher can have disastrous consequences. It is important to carefully follow recovery procedures and proceed cautiously (after reviewing this documentation) if it fails.