NAME

Makefile − making an Exodus Storage Manager

DESCRIPTION

This manual page describes the content and use of the storage manager makefiles. The storage manager source spans many directories. Each directory contains a **Makefile**. There is a single **makefile.common**, which is included by each **Makefile**. Other makefiles included by every **Makefile**are **makefile.install**, **makefile.options**, and **makefile.tags**. Superdirectories may contain a **makefile.subdir**, which each *subdir/***Makefile** includes.

MAKE OPTIONS

Silent Makes: You have several ways to limit the output of the makes. You can use the -s option, to prevent Make from printing each command it executes. (See make(1).)

You can set your environment variable LOCK to "-q" and use the "-e" option with Make, to make the rcs functions like co and ci quiet. (See co(1), ci(1), rcs(1).)

You can comment out the .DEFAULT rule in **workspace/makefile.common**, to prevent the printing of "Making *target*in **directory**. " for each target made.

Partial Makes: You can work on a portion of the storage manager code without having to make the entire storage manager, or having to copy the source for the entire storage manager. This is useful when you are working on a workstation that has little available disk space, and want to mount most of the source from another machine. It is also useful when several people are working on the code at once. A stable "baseline" copy is kept on a server machine. Each programmer has his own "workspace". The Makefiles have macros that specify which modules of the storage manager are in you workspace, and which are taken from the baseline. Modules taken from the baseline are taken in object or executable form. Thing in your workspace are made from a copy of the source. See MOD_WORKSPACE and MOD_BASELINE, under "MACROS", below.

CONFIGURATION OPTIONS

Debugging: The storage manager can be made with several debugging options, which can be combined.

If the DEBUG C preprocessor flag is defined (with -DDEBUG), the storage manager includes myriad internal checks. Some of the checks are CPU-intensive.

The -g compiler flag can be used to generate symbols for use with dbx(1). (See cc(1), dbx(1).)

Optimization: The -O compiler flag can be used, in the absence of -g, to generate optimized code. -O cannot be used with -g; -g3 is used with -g to create optimized code with symbols.

Profiling: The storage manager can be configured to profile itself by using compiler flags -DPROFIL -p. (See prof(1).) When compiled with -DPROFIL, the storage manager responds to the cl_MonDump() client request by writing its profiling information to a file, and passing along the request to all its disk processes, who do the same. Use the environment variable PROFDIR, so that each process writes its profiling information in a different file.

IPC configuration options:

The server and disk processes communicate in one of two ways: with Unix-domain sockets (see socket(2)), or with shared-memory queues. When the storage manager is made with -DDISK_QUEUES, the queues are used. When DISK_QUEUES is not defined, the processes use Unix-domain sockets.

MACROS

These three macros are defined in **makefile.install**:

BINDIR	The directory where executables are installed.
INCLUDEDIR	The directory where include files are installed.
LIBDIR	The directory where libraries are installed.

The following targets are defined in **makefile.options**. If you want to make a server with a different configuration from the default, you edit **makefile.options**, which contains comments describing each of the targets. All executables must be made with the same options, since the executables represent processes that communicate and the processes must agree on both their style and content of communication, which are controlled by these flags. A suffix can be attached to the executable names so that different versions of the executables can be made and installed.

DEBUGFLAGS	This macro takes the combined debugging, profiling, and optimization configuration options (listed in **CONFIGURATION OPTIONS**, above.
DISKPROCNAME	The name of the disk process to be exec-ed after the server forks.
DISKRW	The name of the disk process to be exec-ed after the server forks.
FORMATVOL	The name of the formatvol program.
SERVER	The name of the server program.
SUFFIX	The suffix attached to each of formatvol, diskrw, and sm_server to make FORMATVOL, DISKRW, SERVER, respectively.
QUEUES	Is either -DDISK_QUEUES or null. If -DDISK_QUEUES, shared memory queues will be used for communication between the server and its subordinate processes (diskrw). If null, Unix-domain sockets are used.

Each Makefile (or the environment) must define each of these macros. Null values are acceptable for most; those that cannot be null are marked with "!". Those marked with "*" are defined in **makefile.subdir.common**; those marked with "+" are define in **makefile.subdirs**.

+EXECS	The executables produced in this directory.
FILESC	The C or C++ source files in this directory.
FILESH	The header files in this directory.
FILESO	The object files created in this directory.
+LITTLEMAKEFILE	When non-null, "make workspace" treats this directory as a super-directory (rather than a leaf directory), one in which you might choose to take parts from the baseline and parts from the workspace. See the target *workspace.1level*, below.
LOCALCLEANFILES	The target *clean:* removes garbage files that are common to all directories, and those listed in common macros, such as $(FILESO), $(EXECS), makefile.depend, makefile.depend.bak, tags*, and so on. Any garbage files that are local to a directory will be removed if they are listed in this macro.
LOCALRCSFILES	Files that are to be checked in and out with RCS, other than FILESH, FILESC, and MAKEFILES.
+LOCAL_FLAGS	For server modules this must include -DSERVER_MAKE to distinguish the client and server versions of object files that are created from common sources. For disk-process files, LOCAL_FLAGS must also include -DDISKPROC_MAKE, to distinguish server and disk-process versions of object files that are created from common sources.
LOCK	The RCS locks used for checking sources in and out.
!+MAKEFILES	A list of the makefiles in this directory. All directories use **Makefile**. Some directories also contain **makefile.subdir**.

!MODULE	The name of this directory (without its path).
+MOD_BASELINE	A list of the immediate subdirectories that will come from the baseline.
+MOD_WORKSPACE	A list of the immediate subdirectories that are in the workspace.
OTHERTAGS	Tags files that should be incorporated with tags in this directory. Tags depends on $(OTHERTAGS) so be careful not to create circular references.
!+PARENT	The name of **..** without its path.
!+PATHTOROOT	The path from this dirctory to the root of the workspace. For example, in **workspace/client/bf**, PATHTOROOT is "../..".
+SPECIALINCLUDE	Any include directory other than and that's needed in this directory.

TARGETS

These targets are available at all levels.

ci:	Checks in everything listed in MAKEFILES, FILESC, FILESH, and LOCALRCSFILES. Checks in MAKEFILES with -u $(LOCK), and the rest with $(LOCK). Recursive.
clean:	Cleans out everything but the MAKEFILES and sources. Removes object files, executables (EXEC) and LOCALCLEANFILES. Recursive.
co:	Checks out everything listed in MAKEFILES, FILESC, FILESH, and LOCALRCSFILES. Checks out with $(LOCK). Recursive.
copies:	Copies files listed in COPIES into the working directory (.). If the source file does not exist, the error is ignored.
default:	Alias for workspace.mod; appears at the beginning. Recursive.
depend:	Makes dependencies for everything listed in FILESC. Recursive.
dummy*:	Fake targets that are necessary only to keep the makefile syntax error-free when certain macros are undefined.
getbaselineparts:	Gets the products of the submodules listed in MOD_BASELINE (rather than the product of ".") from the baseline. Used when parts of the module are from the baseline and parts are in the workspace. Recursive.
getmakefiles :	Checks out only the MAKEFILES in each directory. Recursive.
install :	Installs the products in the directories listed in **makefile.install**. Recursive.
links :	Makes symbolic links from . to each of the files listed in LINKS. See also target *$(LINKS)*, below.
makefile :	Makes a writable copy of Makefile, in **makefile** if **makefile** is out-of-date or non-existent.
makercs :	Makes links named RCS to the RCS directories in $BASELINE at each level. Recursive.
rcsdiff:	Recursive.
saber_obj:	Not used.
saber_src:	Not used.
touch:	Like "make -t", except that each directory touches its own products, whose names differ. Recursive.
workspace:	Creates the directory structure and, for the first level (only), gets from the baseline the modules that are listed in MOD_BASELINE, and gets source for the (entire) modules that are in MOD_WORKSPACE. Recursive, but keep in mind that when you first create a workspace, the default Makefiles indicate that everything is in

MOD_WORKSPACE. If you want to get any parts from the baseline, you should use the target *workspace.1level*: edit the resulting **makefile**s as needed, before doing "make workspace".

workspace.1level: Creates the subdirectories of "." and checks out the makefiles in each subdirectory. If the macro LITTLEMAKEFILE in any **Makefile**is defined and non-null, and if **makefile** is not up-to-date, this rule also makes a writable copy of **Makefile** in **makefile**. This target is meant to be used when "." contains modules that could be taken from the baseline. Directories that do not have parts that could be taken from the baseline should not define LITTLEMAKEFILE in their Makefiles.

workspace.recursive: Recursive part of the target *workspace*:.

workspace.mod.recursive:
The recursive part of the target *workspace.mod*:.

$(COPIES) Copies the target source file to ".". If the source file does not exist, the resulting error is ignored. See target copies.

$(FILESO) Compiles object files listed in FILESO.

$(LIBS) If the product depends on any libraries, the libraries are listed in LIBS, with full path names. This target goes to the directory in which the libraries are made, and makes them there.

$(LINKS) If the target file does not exist, it makes the file in its home directory. Makes a symbolic link from this directory to the target. To make the symbolic links without making the target file, use the target *links*:.

$(MOD_WORKSPACE) $(MOD_BASELINE)
Useful when you want to add a module to the workspace (from the baseline) or vice versa. Remove the module's directory and its contents, and then "make <module>".

$(RCSFILES) Compiles object files listed in FILESO, checks out source files listed in FILESC and FILESH, MAKEFILES, and LOCALRCSFILES.

The following targets are available at the root of the workspace:

install_directories: Makes the bin/, lib/, and h/ directories named in **makefile.install**. Invoked by the target *workspace*:.

tags.clean: Removes duplicate tags files in the subdirectories.

The following targets are defined in each directory's Makefile:

getbaseline: Gets the module's product from the baseline. Does not get source or intermediate results such as object files (unless the product of the module is an object file).

workspace.mod: This is the default target. It makes all object files and final products in each directory. It must be define if the product of the directory is anything other than that defined in FILESO and EXECS (such as as library). Recursive.

RESTRICTIONS

The client and server have some function names in common, such as bf_ReadPage, so it is not possible to have a global tags file for the entire storage manager. Instead, there are separate tags files for client modules and for server modules. The tags files you use depend on your working directory. Since not all of the source is in a workspace, you also need to use tags from the baseline. The unfortunate result is that all the tags in the tags files have absolute path names. This may create problems for people who mount a baseline with NFS.

SEE ALSO

Makefile − making an Exodus Storage Manager workspace