Monitoring Object Library Usage and Changes

R.K. Owen, Sterling Software/NASA Ames Research Center

ABSTRACT: The NASA Ames Numerical Aerodynamic Simulation program / Aeronautics Consolidated Supercomputing Facility (NAS/ACSF) supercomputing center services over 1600 users, and has numerous analysts with root access. Several tools have been developed to monitor object library usage and changes. Some of the tools do "non-invasive" monitoring and other tools implement run-time logging even for object-only libraries. The run-time logging identifies who, when, and what is being used. The benefits are that real usage can be measured, unused libraries can be discontinued, training and optimization efforts can be focused at those numerical methods that are actually used. An overview of the tools will be given and the results will be discussed.

1 Introduction

A large site with hundreds of users and numerous analysts with root access poses a unique challenge to tracking and maintaining third party libraries. The combined NASA Ames Numerical Aerodynamic Simulation program / Aeronautics Consolidated Supercomputing Facility (NAS/ACSF) supercomputing center services over 1600 users locally and across the North American continent. This site has two Cray C90s running the UNICOS 8.0.3 operating system, one with 16 processors and one gigaword of memory, the other with 8 processors and 256 megawords of memory. The environment is maintained by approximately 50 analysts each of whom have root access. With this many analysts with root access there is the possibility of some confusion particularly when dealing with the math or graphic libraries. Non-prime analysts may make good intentioned object library changes at the urging of panicked users calling during off-hours, which may result in confusion and problems days or weeks later. Libraries once installed become entrenched and nearly impossible to remove for fear that some users may need the object library for their codes.

In such an environment, several tools have been developed in the last three years that monitor object library changes, load usage, run-time usage, and help create online documentation. The tools follow the UNIX toolkit approach of performing a single function and relying on other tools for scheduling, parsing, *etc.* This paper describes the underlining concepts and design issues for each of the tools in some detail.

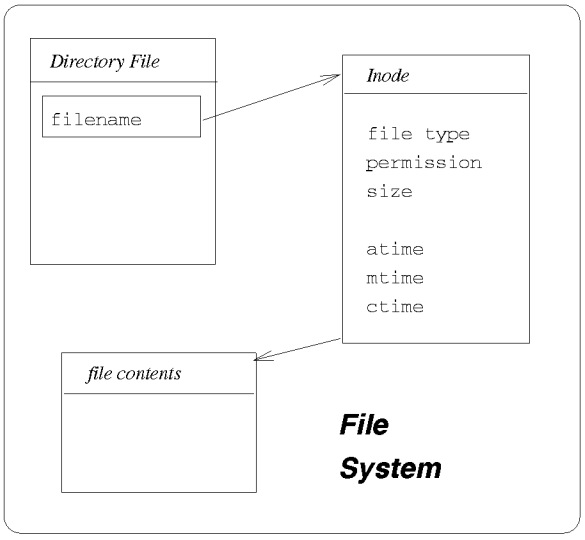

Figure 1: Symbolic representation of UNIX files and inodes.

2 UNICOS/UNIX File System Internals

The UNICOS/UNIX operating system provides some unique abilities to monitor access. The inode for each file contains several bits of information some of which is

(refer to figure 1) the file type, access permissions, owner and group id numbers, file size, number of links, and several times - *atime*, *mtime*, *ctime*. The entry *atime* refers to the last time the file contents were accessed or read, *mtime* is the last time the file contents were modified, and finally *ctime* is the last time the inode for the file was modified. Incidentally, a long directory listing reports the *mtime* value for each file. This inode information, particularly the *atime* and *mtime* forms the basis for a collection of tools to perform non-invasive relocatable object library monitoring.

2.1 Library Monitoring - libmon

The first tool for monitoring described here, *libmon*, searches several standard library paths (`/lib`, `/usr/lib`, `/usr/local/lib`, `/usr/unsupported/lib`, and `/usr/nas/lib`, where the library path is specific to the given machine and is customized via the Makefile. It maintains information with log files in `/usr/spool/libmon`. The log file tracks when it was created or notes when the specified library is modified. For example, here are the contents of `libm.log`

```
Mon_Mar_06_1995_07:00 Wed_Feb_08_1995_09:39 LOG_CREATED
Tue_Aug_22_1995_17:00 Tue_Aug_22_1995_16:04 LIB_MODIFIED
```

The first date and time note when the entry was made, the second date and time is the library *mtime*. The *libmon* tool is called from a *cron* job once every hour and tracks the library *mtime* by changing the log file *mtime* such that they match. If a subsequent scan finds a difference then *libmon* flags the change, logs it, and resets the log file *mtime*. Note that the date and time is in a special format to be used with another tool, *libq*, that parses through and selects lines according to the date and time.

2.2 Non-invasive Library Usage Tracking - libuse

An approximate measure of library usage can be determined by periodically checking the object library *atime*. The tool, *libuse*, is invoked every 10 minutes from a *cron* job and checks whether the object library was read since the last time *libuse* was executed. Many standard libraries are read whether they are needed or not, such as `/lib/libm.a`, hence the list of object library directories should only include those containing optional third-party object libraries specific to that machine. For example, one of the machines at this site has the paths `/usr/local/lib`, `/usr/unsupported/lib`, and `/usr/nas/lib`. The log files are stored in the directory `/usr/spool/libuse`, one for each library, where it writes a line with the date and time. It may also contain additional information noting whether the object library was modified. So the *libuse* files reproduce the same information as *libmon*, but due to the volume of entries is less easy to use. The following abridged example, `libnag.log`, is of the NAG (Numerical Algorithms Group) mathematical library

```
Mon_Mar_06_1995_17:00 LOG_CREATED
Tue_Mar_21_1995_16:30
Tue_Mar_21_1995_16:40
. . .
Fri_May_26_1995_16:00
Mon_Jun_05_1995_11:30 LIB_MODIFIED
Tue_Jun_06_1995_09:40
Tue_Jun_06_1995_13:10
. . .
```

Notice that the *libuse* log files use the same peculiar date and time format that can be parsed by *libq*.

One of the disadvantages of *libmon*/*libuse* is that there is no easy way to determine who made the changes or who used the library. In particular, *libuse* can not determine whether the object library was read once or multiple times in the last 10 minutes and by whom. The information only approximately measures how often the object library is compiled with and can not determine whether the library routines are used or not. Alternately, an executable can be compiled once with one of the numeric object libraries and the program may be executed often.

2.3 Parsing Log Files - libq

Hundreds of log files can be generated and it becomes problematic to determine which of the object libraries were modified or used within a given period of time. The tool, *libq* is a *grep*-like tool designed to easily search through the *libmon* and *libuse* log files and is flexible enough to search any file that uses the same format. All options are given on the command line along with the range of date and times to select for. There are defaults for each data and time range end point. One option specifies to list only file names for those log files which have entries within the specified date and time range. Another option lists only the lines that fall within the date and time range, which is useful for doing a line count of 'hits'.

Yet another option will allow the passing of all lines that do not fall within the date and time range. Thus, *libq* forms the basis for several scripts that measure library usage metrics. Typically, first *libq* is used to find which log files have entries within the specified time interval, and secondly then to output the selected lines within the given time and date range which are then counted.

2.4 Additional Script Tools - libdate

libdate is a very simple tool that just outputs the date and time in a format that can be parsed by *libq*. A typical use of *libdate* is in an application front-end script where the script tracks by whom and when the application was executed. The following example appends the date, time, and username to a log file.

```
echo `libdate` `logname` >> /usr/spool/logs/logfile
```

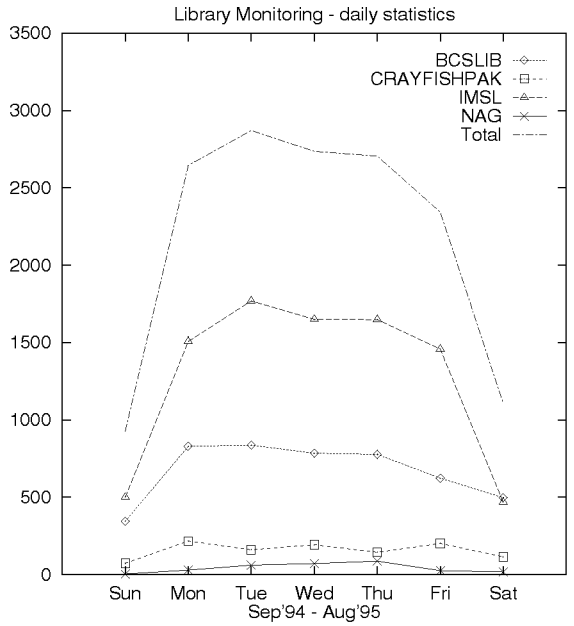

Figure 2: One year's library usage data - daily totals - from Sept. 1, 1994 to Aug. 31, 1995

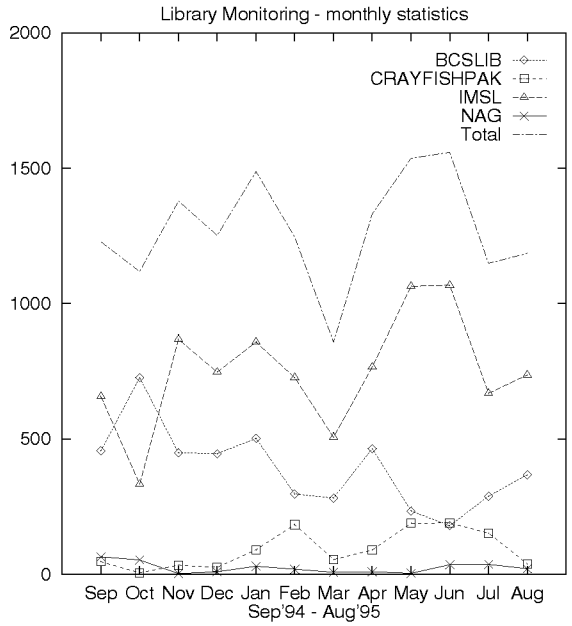

Figure 3: One year's library usage data - monthly totals - from Sept. 1, 1994 to Aug. 31, 1995

3 Library Usage

The tables 1 and 2, and the corresponding figures 2 and 3, represent one year's data from the NAS machine, a C90 with 16 processors and 1 GW of memory. The selection of mathematical libraries range from the general purpose: IMSL, NAG, SLATEC, and the no-longer supported BCSLIB; to the specialized: BCSEXT, and CRAYFISHPAK. LAPACK version 2.0 was added in Jun '95 and has been trimmed down to not conflict with the portion of LAPACK included in the CRI LIBSCI. CRAYFISHPAK is a proprietary library from Green Mountain Software that is the purported successor to the elliptic partial differential equation solver FISHPAK. BCSLIB is from Boeing Computer Services and was once freely available for Crays; however, it is no longer supported and only the extend library BCSEXT is supported which adds many specialized routines for out-of-core solvers and sparse solvers. BCSEXT is not a replacement for the BCSLIB though. This site, which is a national resource for aerodynamics, has many users from Boeing whose codes rely on accessing BCSLIB. The first figure 2 shows that most of the library compilation occurs during the work week, which

is not at all surprising. The second figure 3 shows the usual seasonal variations. The "dip" in March '95 is due to the machine's new operational period (NOP) where the users' allocations are renewed, new users are added, and non-renewed users are disabled. Therefore, it's not unusual for a flurry of activity and lull in the couple of months before the NOP, where users are attempting to use up their allocations and finish their work.

4 Run Time Logging

The non-invasive monitoring that was described starting in section 2 can not determine who is using a library, what specific routines are being used, or even if the library is used at all. If the sources for an object library are available, then calls to some type of logging routine can be added within the sources ... perhaps with some difficulty. However, many of the object libraries (e.g. BCSLIB, BCSEXT, and even the CRI libraries) are available in binary form only and the "luxury" of modifying the sources is not possible. The only solution is to actually modify the relocatable object files and change the entry and external names, and to add front-end code that calls the run-time logging code and the routine.

The advantages of installing run-time logging to a relocatable object library are manyfold. First of all, the detailed information gives library usage, which routines are being used, and who is using it. This knowledge can help focus training efforts, and to guide optimization efforts, and if patterns become apparent then further library acquistions can be more specialized. Second, it's actually easier modifying the relocatable object library than modifying the sources in many cases. Third, only one file needs to be modified ... the relocatable object library instead the hundred or so source routines. Only two additional files need be compiled and added to the object library. One of which contains the front-end code and the other one which handles the run-time logging. However, there are some disadvantages of run-time logging. Primarily, there is the added overhead of performing disk I/O. This is minimized by logging only the first call to any given routine and ignoring subsequent calls. This situation can be further improved by only installing run-time logging to the higher level routine calls that are called infrequently in a code. Avoid run-time logging to low level routines such as the BLAS routines or sorting routines that may be called repeatedly. The call to the run-time logging routine adds the overhead of an additional function call and whatever memory accessing and computational operations involved.

4.1 Object Library Modification - robj

This tool, *robj*, is the most dangerous of all the tools described here. It parses through relocatable object files or *bld* libraries of relocatable object files and identifies the program descriptor table (PDT) entry point name and the externals sections for each relocatable object.

The PDT is the first table for each routine in the relocatable object. It contains information needed to link the module to other modules (such as the entry points and externals used in the routine) along with maintenance information (such as the date and time of compilation, the compiler used, and the operating system level). The four sections of the PDT are the header, block names, entry points, and the external names. The details of the PDT structure and the rest of the relocatable object or *bld* libraries can be found from the *relo(5), bld(5), symbol(5),* and most importantly the header file `relo.h`.

The program, *robj*, departs from the UNIX toolkit approach and provides a comprehensive range of capabilities. In most cases it requires an entry file and an relocatable object file or *bld* library. The entry file contains a list of routines to consider, comments, and optionally the new routine names. The one restriction is that the new routine name must be the same length as the original. This restriction makes the new relocatable object the same length as the original and avoids many other problems. The following is an example of an entry file:

```
# list of entrynames
#   (these are comments)
# change the name of MYSUBR1 to MYSURF1
# the new name must have the same number
# of characters
MYSUBR1 MYSURF1
MYSUBR2
# change MYFUN1 to MYPUN1
MYFUN1 MYPUN1
MYFUN2
MYFUN3
E1
# MINEONLY
# the above is a comment only. Entryname MINEONLY in
# 'mylib' is not affected by any robj processing
```

```
# at all.
```

If no optional secondary name is given then *robj* will "diddle" the routine name. In the above example the routine `MYFUN2` will be changed to `MyFuN2`. Note that Fortran does not specify that case is important and by default the CRI compiler converts all external names to uppercase. The diddled name mixes and alternates uppercase and lowercase letters which makes the Fortran routine inaccessible from Fortran and unlikely to collide with other external names in C. *robj* has options to validate the entry file, compare it to the relocatable object file, and can produce new entry files of matching routines. One example where this capability can be useful for other than run-time logging is with the Numerical Algorithms Group (NAG) math library. NAG only sends out the "single-precision" version of the library for CRI machines. The single-precision routine names end with the letter "E", in contrast to the double precision routines names that end with the letter "F". The following finds all the entry names and can be redirected to a file that needs to be edited

```
robj -v -o libXXX.a | grep Pdtent
```

then using the following

```
robj -x -w -e newXXX.entries -o libXXX.a
```

produces a new library named `LibXXX.a` with modified entry names as specified in `newXXX.entries`. Note that the `-x` option must be added to change all the external references otherwise any internal references within the library will be unsatisfied.

4.2 Front-End Code

One of the principle tasks for *robj* is to produce the front-end code for run-time logging (see figure 4). The front-end code must interface with the established external name, the "diddled" external names, and call a logging routine with the external name passed to it. It must do all of this, while not disturbing the argument list stack. Fortunately, this can be performed using Cray assembly language. The key is the `-a` option in the following example

```
robj -a -x -w -e XXX.entries -o libXXX.a > libXXXfe.s
as libXXXfe.s
bld r LibXXX.a libXXXfe.o
```

where *stdout* is redirected to a file `libXXXfe.s`, which is subsequently compiled and added to the modified library archive.

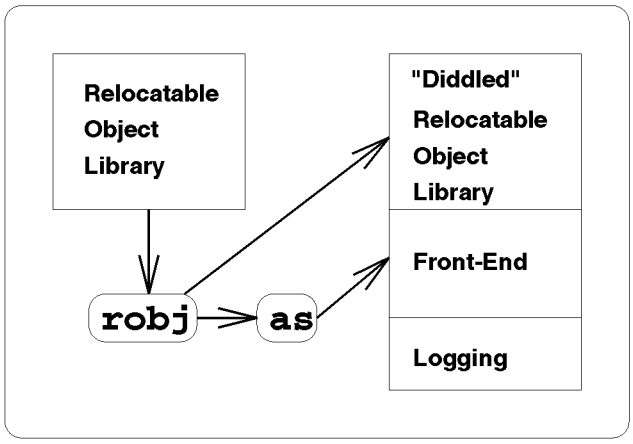

Figure 4: Relocatable object library modification by `robj`.

The following is an assembling language template with explanatory comments that describe the front-end code stubs. The assembly code has the entry name of the library routine which then calls the "diddled" library routine using the argument stack as-is. After the "real" library routine is executed then the assembly code calls another routine that performs the run-time logging.

```
        IDENT           LIBSUBR
* modify the entry point name above
* the following is included in the load module
        COMMENT         'libinfo stub 1.0 09/12/94
, (c) R.K.Owen,Ph.D. 1994'
*******************************************
*      Assemble with Cal Version 2.0        *
*******************************************
* ALLOW UNDERSCORES IN IDENTIFIERS
        EDIT      OFF
        FORMAT    NEW
@DATA   SECTION   DATA,CM
@DATA   =         W.*
        CON       A'LIBSUBR'L    ;Real library rtn
        BSSZ      1              ;name in ascii
                                 ;terminate w/ null
*
SAV1    BSSZ      1              ;2 blank words to
SAV2    BSSZ      1              ;save return value
        SECTION   *
```

```
@CODE   SECTION   CODE
@CODE   =         P.*
*
        MXCALLEN  1             ;max number of args
*
LIBSUBR ENTER
*
* goto subroutine 1st don't mess with registers
* ... pass argument list as-is
*
        R         P.LiBsUbR     ;Real library rtn
*
* save routine/function output (s1/s2) to restore
* later
*
        SAV1,0    S1            ;single return
        SAV2,0    S2            ;double return
*
* log usage
        CALL      LIBINFO,(@DATA),USE=A7
*
* let the return value default to whatever given by
* subroutine/function call
*
        S1        SAV1,0        ;single return
        S2        SAV2,0        ;double return
*
        EXIT
        EXT       LiBsUbR:p     ;Real library rtn
        ENTRY     LIBSUBR       ;stub entry
        END
```

where `LIBSUBR` is the routine name; `LiBsUbR` is the real library routine with a "diddled" name; and `LIBINFO` is the logging routine, which accepts a single argument – a null terminated character string.

The logging routine needs to have a unique external name for each library that has been modified for run-time logging. The utility, *robj*, takes this into account and substitutes a name which is based on the library or relocatable object file name; however, this can be overruled by the `-i` option. Currently, the run-time logging routine is written in C and stores the routine name in a quadratically hashed array. The first invocation of a given routine, the logging routine writes the date and time, the user id, and the routine name to a *world* writable file. This is not desirable! Subsequent versions will use either the *syslogd* daemon, or a customized daemon, to collect the logging information and to write the information in a more secure manner. The following is an example from `/usr/spool/logs/libimsl20.log` (The usernames have been changed)

```
Wed_May_17_1995_08:49 L2TCG     filei
Wed_May_17_1995_08:49 LFSCG     filei
Fri_Jun_02_1995_16:54 QDAGS     mrgoers
Wed_Jun_07_1995_17:26 QDAGS     mrgoers
Wed_Jun_14_1995_15:57 LSGRR     ilu
Thu_Jun_15_1995_08:40 LSGRR     ilu
Thu_Jun_15_1995_08:47 LSGRR     ilu
```

The logging routine is also designed to fail gracefully on any type of error, so the user is never inconvenienced in any manner if logging can not be performed (*e.g.* the log file has incorrect permissions or is missing).

5 On-Line Documentation - alldoc / f77head

Finally, the last tool of any use is on-line documentation. The command-line utility, *alldoc*, does a keyword search on a database and lists out the specified documentation from doc files. Keyword searches can be strung together from one call to *alldoc* to another to fine-tune or narrow the search. Once a candidate routine is selected then documentation can be viewed. This documentation is created from the initial comment fields of the library sources. Most library sources are well structured and *awk* scripts can be used to select out these comment fields. For sources which are not consistently structured another tool was created named *f77head* that can select everything from the initial subroutine or function declaration to the first executable statement.

The next version of *alldoc*, which is in progress, will be more user friendly and require less disk space. The document files can be compressed and can invoke other command-line programs (*i.e. man* or other on-line document facilities). This last feature should save even more disk space by eliminating documentation redundancy. The execution of other command-line programs also allows the inclusion of documents, such as man pages, for libraries that are available in object form only.

6 Conclusion and Goals

Several tools have been created in the last few years to monitor, to run-time log, and to document relocatable object libraries. The tools help to track relocatable object library changes in an environment that grants *root* access to many analysts, where changes can be lost in the noise. The tools can also monitor usage, which can be instrumental in determining which object libraries to keep or to discard due to lack of use. The library management tools form a unified collection of utilities that follows the UNIX paradigm of several small well defined tools.

Work continues on modifying and improving the code, which is still considered a *beta* version by the author. The run-time logging needs to be reworked to write the logging information in a more secure way. The library monitoring tools have ported easily to other UNIX systems and the author is exploring the possibility of porting the *robj* capabilities to COFF (common object file format) and ELF (Executable and Linking Format) object formats. All requests should be sent to `rkowen@nas.nasa.gov` and there may be restrictions due to U.S. export control.

7 Acknowledgements

Dr. Ed Hook of Computer Science Corporation wrote the first version of *alldoc*. This project is funded by Sterling Software on behalf of the NAS / ACSF computer center located at NASA Ames Research Center located at Moffett Field, California. Special thanks to Terry Nelson for proof-reading this paper.

Library	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Total
bcsext	5	14	6	13	30	6	2	76
bcslib	345	830	837	785	777	624	498	4696
crayfishpak	73	217	160	193	144	202	115	1104
imsl	500	1507	1769	1650	1648	1458	468	9000
lapack	0	0	1	0	0	5	1	7
nag	4	29	61	72	87	26	17	296
slatec	1	47	36	23	19	22	5	153
Total	928	2644	2870	2736	2705	2343	1106	15332

Table 1: One year's library usage data - daily totals - from Sept. 1, 1994 to Aug. 31, 1995

Library	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Total
bcsext	0	0	2	3	0	4	1	0	3	45	0	18	76
bcslib	457	727	449	446	502	297	282	465	234	180	289	368	4696
crayfishpak	47	6	34	26	91	184	55	90	189	190	153	39	1104
imsl	657	333	868	747	858	727	506	766	1064	1068	669	737	9000
lapack	0	0	0	0	0	0	0	0	0	5	0	2	7
nag	65	53	3	10	30	19	8	10	4	36	37	21	296
slatec	2	0	23	20	7	15	6	0	43	35	1	1	153
Total	1228	1119	1379	1252	1488	1246	858	1331	1537	1559	1149	1186	15332

Table 2: One year's library usage data - monthly totals - from Sept. 1, 1994 to Aug. 31, 1995