NAME

prof – display profile data

SYNOPSIS

prof [**–a** | **c** | **n** | **t**] [**–o** | **x**] [**–g** | **l**] [**–C**] [**–h**] [**–m** *mdata*] [**–s**] [**–V** *prog*] [**–z**]

DESCRIPTION

The **prof** command interprets a profile file produced by the **monitor** function. The symbol table in the object file *prog* (**a.out** by default) is read and correlated with a profile file (**mon.out** by default). For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

OPTIONS

The mutually exclusive options **–a**, **–c**, **–n**, and **–t** determine the type of sorting of the output lines:

–a Sort by increasing symbol address.

–c Sort by decreasing number of calls.

–n Sort lexically by symbol name.

–t Sort by decreasing percentage of total time (default).

The mutually exclusive options **–o** and **–x** specify the printing of the address of each symbol monitored:

–o Print each symbol address (in octal) along with the symbol name.

–x Print each symbol address (in hexadecimal) along with the symbol name.

The mutually exclusive options **–g** and **–l** control the type of symbols to be reported. The **–l** option must be used with care; it applies the time spent in a static function to the preceding (in memory) global function, instead of giving the static function a separate entry in the report. If all static functions are properly located (see example below), this feature can be very useful. If not, the resulting report may be misleading.

Assume that **A** and **B** are global functions and only **A** calls static function **S**. If **S** is located immediately after A in the source code (that is, if **S** is properly located), then, with the **–l** option, the amount of time spent in **A** can easily be determined, including the time spent in **S**. If, however, both **A** and **B** call **S**, then, if the **–l** option is used, the report will be misleading; the time spent during **B**'s call to **S** will be attributed to **A**, making it appear as if more time had been spent in **A** than really had. In this case, function **S** cannot be properly located.

–g Include static (non-global) functions.

–l Do not include static (non-global) functions (default).

The following options may be used in any combination:

–C Demangle C++ symbol names before printing them out.

–h Suppress the heading normally printed on the report. This is useful if the report is to be processed further.

−m *mdata*
Use file *mdata* instead of **mon.out** as the input profile file.

−s Print a summary of several of the monitoring parameters and statistics on the standard error output.

−V Print **prof** version information on the standard error output.

−z Include all symbols in the profile range, even if associated with zero number of calls and zero time.

A program creates a profile file if it has been link edited with the **−p** option of **cc**(1B). This option to the **cc**(1B) command arranges for calls to **monitor** at the beginning and end of execution. It is the call to **monitor** at the end of execution that causes the system to write a profile file. The number of calls to a function is tallied if the **−p** option was used when the file containing the function was compiled.

A single function may be split into subfunctions for profiling by means of the **MARK** macro (see **prof**(5)).

ENVIRONMENT

PROFDIR The name of the file created by a profiled program is controlled by the environment variable **PROFDIR**. If **PROFDIR** is not set, **mon.out** is produced in the directory current when the program terminates. If **PROFDIR=***string*, *string***/***pid***.***progname* is produced, where *progname* consists of **argv[0]** with any path prefix removed, and *pid* is the process ID of the program. If **PROFDIR** is set, but null, no profiling output is produced.

FILES

mon.out default profile file
a.out default namelist (object) file

SEE ALSO

cc(1B), **exit**(2), **profil**(2), **malloc**(3C), **malloc**(3X), **monitor**(3C), **prof**(5)

The **lprof** section in *Programming Utilities Guide*

NOTES

The times reported in successive identical runs may show variances because of varying cache-hit ratios that result from sharing the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data. In rare cases, the clock ticks initiating recording of the program counter may ‘‘beat’’ with loops in a program, grossly distorting measurements. Call counts are always recorded precisely, however.

Only programs that call **exit** or return from **main** are guaranteed to produce a profile file, unless a final call to **monitor** is explicitly coded.

The times for static functions are attributed to the preceding external text symbol if the **−g** option is not used. However, the call counts for the preceding function are still correct; that is, the static function call counts are not added to the call counts of the external function.

If more than one of the options **−t**, **−c**, **−a**, and **−n** is specified, the last option specified is used and the user is warned.

Profiling may be used with dynamically linked executables, but care must be applied. Currently, shared objects cannot be profiled with **prof**. Thus, when a profiled, dynamically linked program is executed, only the ''main'' portion of the image is sampled. This means that all time spent outside of the ''main'' object, that is, time spent in a shared object, will not be included in the profile summary; the total time reported for the program may be less than the total time used by the program.

Because the time spent in a shared object cannot be accounted for, the use of shared objects should be minimized whenever a program is profiled with **prof**. If desired, the program should be linked to the profiled version of a library (or to the standard archive version if no profiling version is available), instead of the shared object to get profile information on the functions of a library. Versions of profiled libraries may be supplied with the system in the **/usr/lib/libp** directory. Refer to compiler driver documentation on profiling.

Consider an extreme case. A profiled program dynamically linked with the shared C library spends 100 units of time in some **libc** routine, say, **malloc( )**. Suppose **malloc( )** is called only from routine **B and B** consumes only 1 unit of time. Suppose further that routine **A** consumes 10 units of time, more than any other routine in the ''main'' (profiled) portion of the image. In this case, **prof** will conclude that most of the time is being spent in **A** and almost no time is being spent in **B**. From this it will be almost impossible to tell that the greatest improvement can be made by looking at routine **B** and not routine **A**. The value of the profiler in this case is severely degraded; the solution is to use archives as much as possible for profiling.