



Linux Ubuntu 26.04 Manual Pages on command 'sprof.1'

\$ man sprof.1

sprof(1) General Commands Manual sprof(1)

NAME

sprof - read and display shared object profiling data

SYNOPSIS

sprof [option ...] shared-object-path [profile-data-path]

DESCRIPTION

The sprof command displays a profiling summary for the shared object (shared library) specified as its first command-line argument. The profiling summary is created using previously generated profiling data in the (optional) second command-line argument. If the profiling data pathname is omitted, then sprof will attempt to deduce it using the soname of the shared object, looking for a file with the name <soname>.profile in the current directory.

OPTIONS

The following command-line options specify the profile output to be produced:

--call-pairs

-c Print a list of pairs of call paths for the interfaces exported by the shared object, along with the number of times each path is used.

--flat-profile

-p Generate a flat profile of all of the functions in the monitored object, with counts and ticks.

--graph

-q Generate a call graph.

If none of the above options is specified, then the default behavior is to display a flat profile and a call graph.

The following additional command-line options are available:

--help

-? Display a summary of command-line options and arguments and exit.

--usage

Display a short usage message and exit.

--version

-V Display the program version and exit.

STANDARDS

GNU.

EXAMPLES

The following example demonstrates the use of sprof. The example consists of a main program that calls two functions in a shared object. First, the code of the main program:

```
$ cat prog.c;

#include <stdlib.h>

void x1(void);

void x2(void);

int

main(int argc, char *argv[])

{

    x1();

    x2();

    exit(EXIT_SUCCESS);

}
```

The functions x1() and x2() are defined in the following source file that is used to con?

struct the shared object:

```
$ cat libdemo.c;

#include <unistd.h>

void

consumeCpu1(int lim)

{

    for (unsigned int j = 0; j < lim; j++)

        getppid();

}
```

```
void
x1(void) {
    for (unsigned int j = 0; j < 100; j++)
        consumeCpu1(200000);
}
```

```
void
consumeCpu2(int lim)
{
    for (unsigned int j = 0; j < lim; j++)
        getppid();
}
```

```
void
x2(void)
{
    for (unsigned int j = 0; j < 1000; j++)
        consumeCpu2(10000);
}
```

Now we construct the shared object with the real name libdemo.so.1.0.1, and the soname lib?

demo.so.1:

```
$ cc -g -fPIC -shared -Wl,-soname,libdemo.so.1 \
    -o libdemo.so.1.0.1 libdemo.c;
```

Then we construct symbolic links for the library soname and the library linker name:

```
$ ln -sf libdemo.so.1.0.1 libdemo.so.1;
$ ln -sf libdemo.so.1 libdemo.so;
```

Next, we compile the main program, linking it against the shared object, and then list the dynamic dependencies of the program:

```
$ cc -g -o prog prog.c -L. -ldemo;
$ ldd prog;
    linux-vdso.so.1 => (0x00007fff86d66000)
    libdemo.so.1 => not found
    libc.so.6 => /lib64/libc.so.6 (0x00007fd4dc138000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fd4dc51f000)
```

In order to get profiling information for the shared object, we define the environment vari?

able LD_PROFILE with the soname of the library:

```
$ export LD_PROFILE=libdemo.so.1;
```

We then define the environment variable LD_PROFILE_OUTPUT with the pathname of the directory where profile output should be written, and create that directory if it does not exist already:

```
$ export LD_PROFILE_OUTPUT=$(pwd)/prof_data;
```

```
$ mkdir -p $LD_PROFILE_OUTPUT;
```

LD_PROFILE causes profiling output to be appended to the output file if it already exists,

so we ensure that there is no preexisting profiling data:

```
$ rm -f $LD_PROFILE_OUTPUT/$LD_PROFILE.profile;
```

We then run the program to produce the profiling output, which is written to a file in the directory specified in LD_PROFILE_OUTPUT:

```
$ LD_LIBRARY_PATH=. ./prog;
```

```
$ ls prof_data;
```

```
libdemo.so.1.profile
```

We then use the sprof -p option to generate a flat profile with counts and ticks:

```
$ sprof -p libdemo.so.1 $LD_PROFILE_OUTPUT/libdemo.so.1.profile;
```

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	us/call	us/call	name
60.00	0.06	0.06	100	600.00		consumeCpu1
40.00	0.10	0.04	1000	40.00		consumeCpu2
0.00	0.10	0.00	1	0.00		x1
0.00	0.10	0.00	1	0.00		x2

The sprof -q option generates a call graph:

```
$ sprof -q libdemo.so.1 $LD_PROFILE_OUTPUT/libdemo.so.1.profile;
```

index	% time	self	children	called	name
	0.00	0.00	100/100		x1 [1]
[0]	100.0	0.00	0.00	100	consumeCpu1 [0]

	0.00	0.00	1/1		<UNKNOWN>
[1]	0.0	0.00	0.00	1	x1 [1]

	0.00	0.00	100/100	consumeCpu1 [0]

	0.00	0.00	1000/1000	x2 [3]
[2]	0.0	0.00	0.00	1000 consumeCpu2 [2]

	0.00	0.00	1/1	<UNKNOWN>
[3]	0.0	0.00	0.00	1 x2 [3]
	0.00	0.00	1000/1000	consumeCpu2 [2]

Above and below, the "<UNKNOWN>" strings represent identifiers that are outside of the pro?
filed object (in this example, these are instances of main()).

The sprof -c option generates a list of call pairs and the number of their occurrences:

```
$ sprof -c libdemo.so.1 $LD_PROFILE_OUTPUT/libdemo.so.1.profile;
<UNKNOWN>          x1          1
x1          consumeCpu1      100
<UNKNOWN>          x2          1
x2          consumeCpu2     1000
```

SEE ALSO

gprof(1), ldd(1), ld.so(8)