NAME

dyntrace — Dynamic execution tracing utility

SYNOPSIS

dyntrace [**-vz**] [**-c** *seconds*] [**-f** *opcodefile*] [**-o** *outputfile*] *command ...*
dyntrace [**-vz**] [**-c** *seconds*] [**-f** *opcodefile*] [**-o** *outputfile*] **-p** *pid*

DESCRIPTION

The **dyntrace** utility traces the specified process building an execution profile of which hardware instructions are executed and the time spent by the processor to execute them. Tracing continues until either the target process exits or **dyntrace** is terminated. At the end of the trace, a summary of the executed instructions and their timings is written to an output trace file. The instructions are grouped in the output based on the region of memory they were executed in (e.g. text, data, or stack). The trace file is in XML format described by the document type definition file `/usr/local/share/dyntrace/dyntrace.dtd`.

The options are as follows:

-v Increase verbosity. May used multiple times to increase the amount of information **dyntrace** reports about its own operation during the trace. Beware: messages are written on `stderr` and may be intermixed with the traced process' own output.

-z Include hardware instructions with zero execution counts in the output trace file. By default, only instructions with non-zero counts are recorded.

-c *seconds*
Checkpoint the execution profile every *seconds* seconds. This causes the current state of the execution profile to be written to the trace file periodically during a trace, allowing partial data to recovered in the event the trace gets interrupted. By default, checkpoints are performed every 15 minutes (900 seconds). A *seconds* value of 0 disables checkpointing; the execution profile will only be recorded to the output file when the trace terminates.

Checkpoints can also be requested interactively by the user by sending the `USR1` or `INFO` signal to the **dyntrace** process. See `kill`(1).

-f *opcodefile*
Specify an alternate file to load descriptions of the hardware instructions from.

-o *outputfile*
Specify the trace output file. By the default, **dyntrace** records the execution profile in a file named *procname*.trace in the current directory where *procname* is the name of the process being traced.

-p *pid*
Begin tracing the execution of the indicated process id (only one **-p** flag is permitted). The trace will start with the next instruction executed by the specified process. Note that if the specified process is currently performing a system call, the trace will not begin until the system call returns which may occur after an indeterminate time has elapsed.

The trace can be terminated by sending **dyntrace** a `HUP`, `QUIT`, or `TERM` signal. When **dyntrace** receives one of these signals it attempts to detach from the traced process before exiting, allowing the traced process to continue running untraced. However, if the traced process is currently performing a system call, the detach may not occur until that system call returns.

Which processes **dyntrace** can attach to are determined by the security policy of the host operating system.

command ...
Execute *command* and trace its execution. The remainder of the command line arguments are passed to the specified command as its arguments.

The **-p** and *command* options are mutually exclusive.

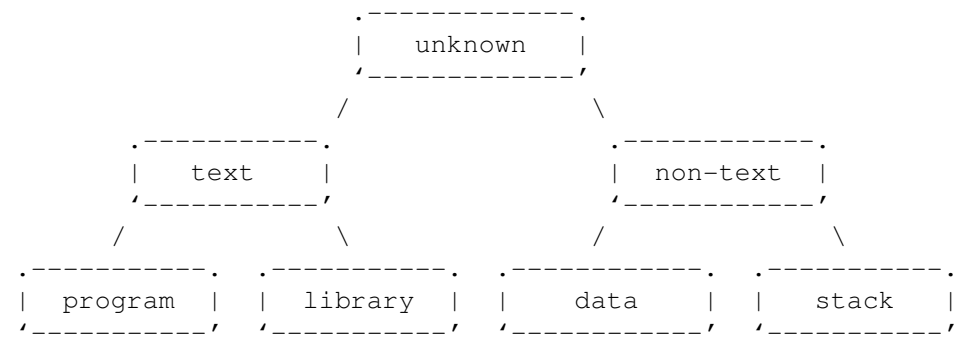
IMPLEMENTATION NOTES

The **dyntrace** utility encapsulates three main functions:

- Instruction counting. This is the most basic function of **dyntrace** and is always available. The target process is single stepped through its execution with the **dyntrace** utility updating the instruction count histogram before each instruction is executed.
- Instruction timing. Some platforms provide per-process performance counters that can be utilized to get the number of processor cycles elapsed in the target process between trace steps. Since the target process is stopped after each instruction, this cycle count represents the number of cycles elapsed executing that single instruction within the limits of the accuracy of the platform's performance counters.

  The accuracy of instruction timing is further limited by the fact that tracing the process perterbs its execution. For example, modern processor optimizations such as branch prediction and speculative execution are undermined by virtue of trapping and transfering control to **dyntrace** for every instruction. As such, collected instruction timings should not be taken as absolute times but rather an indicator of the relative timing of instructions.
- Memory region differentiation. Identifies the type of memory regions instructions are executed in and groups them in the output according to type. Whether regions of the traced process' virtual memory can be categorized and at what level of detail they can be categorized depends on the platform. For example, if the operating system makes no distinction between regions of the processes' virtual memory, then all instructions are reported as having executed in an "unknown" region.

  The levels of detail in memory region differentiation from least specific at the top to most specific at the bottom:



  Levels of detail are not mutually exclusive. It is possible for one region to be of unspecified "non-text" type while another region is clearly identifiable as "stack".

Some functions may not be available on some platforms or may require elevated privileges (e.g. root user) to use; **dyntrace** will issue a warning whenever a function is not available.

FILES

`/usr/local/share/dyntrace/dyntrace.dtd`
Document type definition of the XML-format execution profile file output by **dyntrace**.

`/usr/local/share/dyntrace/oplist.dtd`
: Document type definition of the XML-format instruction description file used by **dyntrace** to identify instructions.

`/usr/local/share/dyntrace/oplist-x86.xml`
: Instruction description file for Intel(R) 8086 and later processor lines and their clones.

EXAMPLES

execute and trace the command "df -i"

```
$ dyntrace df -i
```

execute and trace the command "javavm -green HelloWorld"

```
$ dyntrace javavm -green HelloWorld
```

execute and trace the command "/bin/sh", write profile to "test.trace"

```
$ dyntrace -o test.trace /bin/sh
```

begin tracing the execution of process id 1024, disable checkpointing

```
$ dyntrace -c 0 -p 1024
```

DIAGNOSTICS

On error, **dyntrace** exits with one of the exit codes defined in the host operating system's `<sysexits.h>`.

COMPATIBILITY

The **dyntrace** tracer runs on the following platforms:

FreeBSD/i386
: Instruction counting and region differentiation are implemented on all versions of FreeBSD 4.0 and greater. Region differentiation is only available if `procfs`(5) is mounted and accessable to the user. Instruction timing is implemented on FreeBSD 5.4 and later via the `pmc`(4) API. The `pmc`(4) API is only available if the kernel has been compiled with the **PMC_HOOKS** option and the pmc module is loaded into the kernel.

  Note: versions of FreeBSD released prior to December 12th, 2004 have a bug which causes child processes of the traced process to terminate immediately after the first executed instruction. **dyntrace** does not trace the child processes (see **BUGS**) but their premature deaths will presumably alter the flow of control in the traced process itself. For this reason, it is recommended to use a version of FreeBSD released more recently than December 12th, 2004. The **dyntrace** distribution includes a patch that can be applied by the system adminstrator to FreeBSD versions 5.0 through 5.3 to correct the bug (patches/trace-fix.diff in the **dyntrace** source distribution).

more to come...

AUTHORS

Kelly Yancey ⟨kbyanc@posi.net, kbyanc@FreeBSD.org⟩

BUGS

dyntrace does not yet support tracing multithreaded processes utilizing multiple light-weight processes. A workaround for tracing Java programs is to force the virtual machine to use its "green threads" implementation by specifying **-green** on the java command-line (see **EXAMPLES**).

There is currently no way to include children of the specified process in the trace. This precludes collecting useful execution profiles from programs such as apache which fork child processes to perform a portion of their work. Conversely, tracing children should never be the default as that would preclude tracing debuggers (or another instance of **dyntrace**) which need to control their children themselves.

Some processes are really the agreggation of multiple programs loaded in succession using `execl`(3) or a similar system call. For example, the **`javavm`** program on FreeBSD is actually a shell script which execs another shell script which in turn execs the real Java VM. In this example, there are three programs all of which were run as a single process, one program after the other. The output trace file format should be extended to report instruction counts from each program separately if possible.