

Dynamic Tracing with DTrace SystemTap

Sergey Klyaus

Copyright © 2011-2016 Sergey Klyaus

This work is licensed under the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 License. To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Table of contents

Introduction	5
Foreword	5
Typographic conventions	7
TSLoad workload generator	10
Operating system Kernel	11
Module 1: Dynamic tracing tools. dtrace and stap tools	13
Tracing	13
Dynamic tracing	14
DTrace	15
SystemTap	17
Safety and errors	20
Stability	21
Module 2: Dynamic tracing languages	23
Introduction	23
Probes	25
Arguments	31
Context	32
Predicates	33
Types and Variables	35
Pointers	38
Strings and Structures	41
Exercise 1	42
Associative arrays and aggregations	42
Time	45
Printing and speculations	46
Tapsets translators	48
Exercise 2	49
Module 3: Principles of dynamic tracing	50
Applying tracing	50
Dynamic code analysis	51
Profiling	56
Performance analysis	59
Pre- and post-processing	61
Vizualization	63
Module 4: Operating system kernel tracing	67
Process management	67

Exercise 3	73
Process scheduler	74
Virtual Memory	83
Exercise 4	91
Virtual File System	91
Block Input-Output	97
Asynchronicity in kernel	102
Exercise 5	103
Network Stack	105
Synchronization primitives	109
Interrupt handling and deferred execution	113
Module 5: Application tracing	115
Userspace process tracing	115
Unix C library	118
Exercise 6	120
Java Virtual Machine	120
Non-native languages	126
Web applications	128
Exercise 7	132
Appendix A. Exercise hints and solutions	134
Exercise 1	134
Exercise 2	135
Exercise 3	135
Exercise 4	137
Exercise 5	139
Exercise 6	140
Exercise 7	140
Appendix B. Lab setup	142
Setting up Operating Systems	142
iSCSI	143
Web application stack	144
Appendix C. Cheatsheet	149
Cheatsheet	149

Introduction

Foreword

While I was working on my bachelor thesis, I discovered that code analysis task is a key step on the path towards solving software problems: aborts and coredumps, excessive (or unreasonably small) resource consumptions, etc. It was devoted to microkernel architecture, and when I found it inadequately documented, I had to dive deep down to their sources.

After that, I started to apply code reading on my work, because sources always have most actual and full information than user documentation. Sources better explain origin of an error than documentation. For example, take a look at [UFS documentation for Solaris 10](#):

`-b bsize` The logical block size of the file system in bytes, either 4096 or 8192. The default is 8192. The sun4u architecture does not support the 4096 block size.

Real condition that describes block size limits in UFS is a bit more complex:

```
928     if (fsp->fs_bsize > MAXBSIZE || fsp->fs_frag > MAXFRAG ||
929         fsp->fs_bsize fs_bsize
```

So, more accurate condition that applies to all architectures may be described as: block size should be greater or equal page size, but not exceed 8192 bytes (MAXBSIZE macro) and also be larger than superblock. I have to admit, that sometimes I was too hasty to look into code and ignored clues that documentation provides, but in most cases source code analysis approach paid off, especially in hard ones.

Information extraction from source code alone is called *static code analysis*. This method is not sufficient, because you cannot look into source of highly-universal system like Linux Kernel without having in mind what requests it will process. Otherwise, we would have to process all code branches, but that dramatically increases complexity of code analysis. Because of that, you have to run experiments sometimes and perform *dynamic code analysis*. Through dynamic analysis you will cut out unused code paths and improve your understanding of a program.

While I was working on my thesis, I used Bochs simulator which can generate giant traces: one line per assembly instructions. Fortunately, modern operating systems have much more flexible tools: *dynamic instrumentation tools*, and that is the topic of this book.

I wrote first useful DTrace script for the request in which customer encountered the following panic:

```
unix: [ID 428783 kern.notice] vn_rele: vnode ref count 0
```

As you can see from the message, *reference counter* decreases one more time (for example, if you closed file twice). Of course, if you call `close()` twice, that won't cause system panic, so we have to deal with more specific *race condition* or a simple bug when `vn_rele()` is called twice. To unveil that issue, I had to write DTrace script, that traced `close()` and `vn_rele()` calls (and also some socket stuff).

While I was getting familiar with Linux Kernel, I used DTrace competitor from Linux World — SystemTap. I began preparation of small workshop about DTrace and SystemTap in 2011, but I decided to add comments to each slide for my workshop. The amount of comments was growing: I prepared introduction, chapter about script languages in DTrace and SystemTap and description of process management in Linux and Solaris with *dumptask* scripts. But amount of time that I spend to prepare "process management" topic had scared me, and I decided that I couldn't write all topics about OS kernel architecture that I planned in the beginning, so I stopped writing this book.

I returned to it in 2013. At the time, I was actively deconstructing CFS scheduler in Linux, so it made easier to write next architecture topic: "process scheduler". I had some experience with ZFS internals, so writing topics about block input-output was easy too. Eventually, I got interest in web application performance — that gave ground to fifth chapter of this book. In the end of 2013 draft of this book was prepared. Unfortunately, editing took more than year, and another year — translation.

Two specialists in the area of Solaris internals and DTrace: Jim Mauro and Brendan Gregg, had published a book "DTrace Dynamic Tracing in Oracle® Solaris, Mac OS X, and FreeBSD" in 2011. It has huge volume (more than thousand pages), and excellent description of basic performance and computer architecture principles and how they reflected in DTrace tracing capabilities. That book has a lot of one-liners that can be copied to the terminal and start collecting data immediately. In our book we will concentrate our efforts in diving into applications and kernel code and how it can be traced.

Book goals changed while it was written too that lead to inconsistencies. Originally it was just with comments, I put everything looking like documentation as a links, but modules 4 and 5 have tables with probe names and its arguments. While book was written, SystemTap was rapidly growing, Linux kernel is changing fast and Solaris became proprietary so it is hard to maintain example compatibilities for several versions simultaneously. I've updated examples for CentOS 7 and Solaris 11.2, but it'll probably break compatibility with older versions.

Send me your feedback to myautneko+dtrace_stap@gmail.com.

Acknowledgements

I want to thank my advisor, Boris Timchenko, who gave me direction in the world of Computer Science and whose influence was probably highest motivation to write this book. He is probably first man who said word *trace* in my life. Thanks to Sergey Klimenkov and Dmitry Sheshukov, my former supervisors at [Tune-IT](#) who were very supportive during preparation of that book. It will also won't happen without Tune-IT demo equipment which were used to try examples and lab assignments. Sergey is also an expert in Solaris architecture, he is teaching Solaris Internals at Tune-IT education centre, and I was one of his students there.

Thanks DTrace SystemTap community for creating such great instruments, especially Brendan Gregg, who is first man who tamed power of these tools. Nan Xiao is a great person who edited this book.

And finally, this book won't happen without my parents, who inspired my to always learn new.

Typographic conventions

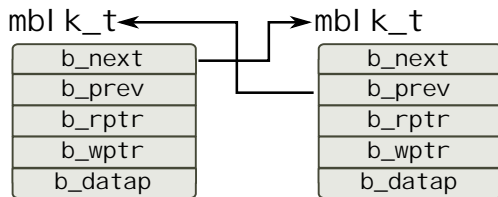
This is a book published on the web and so it doesn't have any "typography", but certain parts of text are decorated with certain styles, thus we describe them in this section of the book.

<i>Meaning</i>	<i>Example</i>
First appearance of new terms	<i>Central processing unit (CPU)</i> executes program code.
Multiple terms linked with each other	CPU consists of execution units , cache and memory controller .
Definition of a term	<div>Definition</div> <p>According to this book,</p> <p><i>A central processing unit (CPU)</i> executes program code.</p>
Additional information about OS or hardware internals	<div>Information</div> <p>Do not read me if you already know me the answer</p>
Notes and some additional information	<div>Note</div> <p>I am note and I am providing external information about the implementation</p> <div>Warning</div> <p>I will warn you about some implementation quirks</p>
Information that some of the examples or code in the section is not suitable for production use	<div>DANGER!</div> <p>Never try to do <code>rm -rf /</code> on your home computer.</p>
Function name, or name of the probe, any other entity that exist in source code	If you want to print a line on standard output in pure C, use <code>puts()</code>
Chunk of the code that has to be used or command to be executed	<pre>int main() { puts("Hello, world"); } \$ perl -se ' print "Hello, \$who" . "\n" ' -- -who=world</pre>
Placeholders in program examples are covered in <i>italic</i>	<code>puts(<i>output-string</i>)</code>

Meaning	Example
Large portions of example outputs may have some output outlined with bold:	<pre>\$ gcc hello.c -o hello \$./hello Hello, world</pre>
Large program listing (if you want to show it, press on "+" button)	Source file: scripts/src/hellouser.py

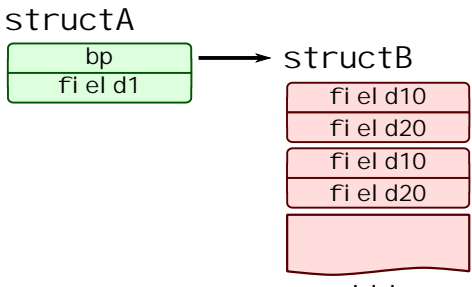
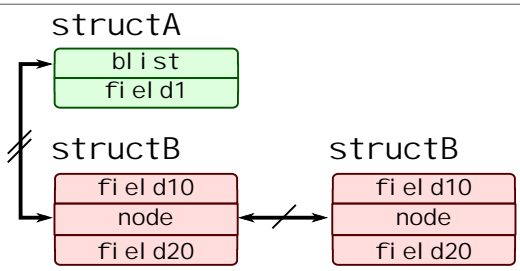
Structural diagrams

Many kernel-related topics will contain structural diagrams which will represent kernel data structures like this:



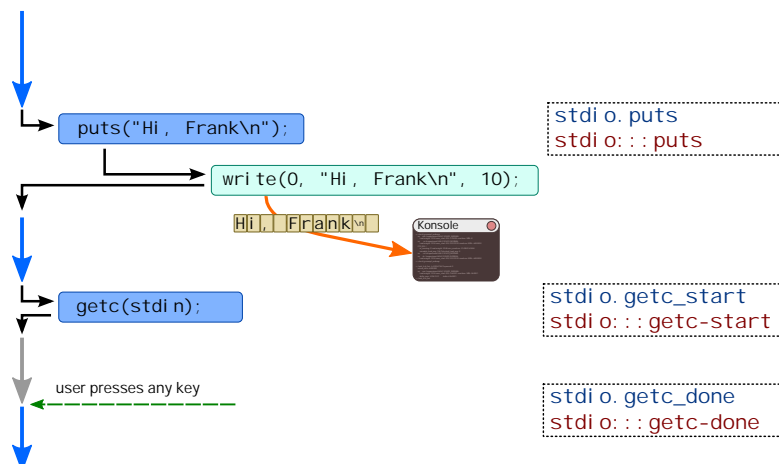
In this example two instances of `mbl k_t` structure (which is typedef alias) are shown which are linked together through pair of `mbl k_t` pointers `b_next` and `b_prev`. Not all fields are shown on this diagram, types are omitted, while order of fields may not match real one. Following conventions are used in this type of diagrams:

Example code	Diagram and explanation
<pre>struct structB { int field10; char field20; }; struct structA { struct structB* bp; int field1; };</pre>	<p>structA</p> <p>structure structA points to instance of structB</p>
<pre>struct structA; struct structB { int field10; struct structA* ap; }; struct structA { struct structB* bp; int field1; };</pre>	<p>structA</p> <p>structure structB contains backward pointer to structure structA</p>
<pre>struct structA { struct structB* bp; int field1; struct { char c1; char c2; } cobj; };</pre>	<p>structA</p> <p>structure structA has embedded structure (not necessarily to be anonymous)</p>

Example code	Diagram and explanation
<pre> struct structB { int field10; char field20; }; struct structA { struct structB* bp; int field1; }; </pre>	 <p>structure structA points to a dynamic array of structures structB</p>
<pre> struct structB { int field10; char field20; struct list_head node; }; struct structA { struct list_head blist; int field1; }; </pre>	 <p>structure structA contains head of linked list of structB instances</p> <p>Various structure relations can be shown with this type of arrows:</p> <ul style="list-style-type: none"> • Single solid glyph shows node-to-node relations in linked list • Double solid glyphs shows head-to-node relations in linked list • Double dashed glyphs shows various tree-like relations like RB tree in Linux

Timeline diagrams

Timeline diagrams are used to show various processes that exist in traced system and chain of events or operations happening with them and at the same time contains names of probes:



This diagram should be read like this:

- Thick **colored** arrows represent flow of some processes — usually they are threads or processes in a system. **Gray** arrows represent processes that are inactive for some reason (usually, blocked and thus cannot be executed on CPU). Arrows corresponding to the same

process share same baseline.

- Thin black arrows demonstrate transfer of control between several operations. In this example `puts()` call triggers `write` system call. Some of them may be omitted.
- Thin colored lines demonstrate subsequent chain of events unrelated to the process. In this example phrase `Hi, Frank` arrives in Konsole window (graphic terminal).
- Text in dashed rectangles contains name of **SystemTap** and **DTrace** probes corresponding to the operations or events happening with the process.

Virtual time axis beginning at the top and it is vertical.

DANGER!

Probes shown in this example are purely fictional.

TSLoad workload generator

During this course we will need to demonstrate created scripts on a real system. We will use version 0.2 of TSLoad workload generator to do so. Its [documentation](#) and [source code](#) are available on GitHub.

Experiment configuration files are kept in JSON format: each experiment starts with directory with `experiment.json` file in it (it can also be accompanied by traces and timeseries). This file contains description of threadpools and workloads: their types and parameters.

Source file: [book/intro/experiment.json](#)

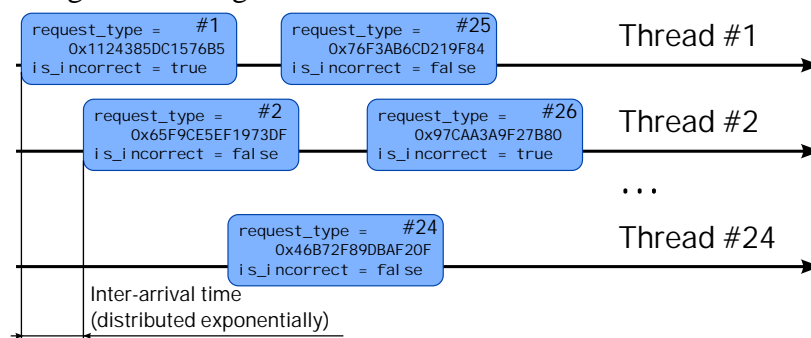
For example, defines an experiment called `jump_table`. `workloads` section defines workload `jt` which type is also `jt`. That workload have the following parameters:

- `num_request_types` - set globally for entire workload - number of "request types" that will be generated;
- `request_types` - generated for each request with linear congruential PRNG;
- `is_incorrect` - boolean value which will be set to true for 20% requests.

It also defines *request scheduler* — inter arrival time will be generated using exponential distribution. `steps` section defines number of requests which will be generated for this workload: 100 steps with 2000 requests in each.

`threadpools` section defines threadpools which will perform our workloads. It defines pool `tp_jt` which contains 24 threads with step period set to 2 second (as parameter `quantum` sets in nanoseconds). *Threadpool dispatcher* describes how requests will be distributed across threads and it is set to round-robin.

If we try to draw a timing diagram of the requests generated by this configuration file we will get something like .



`j t` workload type is defined in a separate loadable module which contains code for simulating requests. During our book we meet similar modules in exercises: `proc_starter` which forks processes, `file_opener` which randomly opens files and other modules.

Experiment is started with `tsexperiment` command:

```
# tsexperiment -e /path/to/experiment run
```

In this command `/path/to/experiment` is a directory which contains file `experiment.json`. That directory will also contain experiment results which can be listed with `list` subcommand of `tsexperiment`:

```
# tsexperiment -e /path/to/experiment list
```

Results may be exported to CSV format with `export` subcommand or some statistics may be shown with `report` subcommand.

It is not necessary to edit configuration file each time parameter have to be altered: `run` subcommand has `-s` option. To provide its argument, check flattened names of configuration parameters with `-l` option of subcommand `show`:

```
# tsexperiment -e /opt/tload/var/tload/mbench/j t show -l
name=j ump_table
steps:j t: num_steps=100
steps:j t: num_requests=2000
...
```

So, to change number of per-step requests to 500, you should call `tsexperiment` with following options:

```
# tsexperiment -e /opt/tload/var/tload/mbench/j t run \
-s steps:j t: num_requests=500
```

In some cases we will need to use hardware device names in experiment configuration, i.e. to bind threads to CPU cores. To get their names, run `tshostinfo` command:

```
tshostinfo -x
```

Operating System Kernel

Definition

According to Wikipedia, [Operating System Kernel](#) is

a computer program that manages I/O (input/output) requests from software, and translates them into data processing instructions for the central processing unit and other electron components of a computer. The kernel is a fundamental part of a modern computer's operating system.

We will refer to operating system kernel as *kernel* in the rest of the book. Applications are using *system call* mechanism to access various kernel functions, and by doing that they transfer control to kernel routines. The current state of application including all variables and current *program counter* is called *context*. C is a programming language which is vastly used for writing Unix-like operating systems kernels such as Solaris, FreeBSD and Linux. C supports only procedural programming, but kernel developers adopted object-oriented and even functional programming.

Where can we get information on kernel? Like I said, the most reliable source of such information is source codes which contain comments. You can use cross-reference tools to navigate source codes as easy as click a hyperlink. Some of them are publicly available: like lxr.linux.no which contains Linux source and src.illumos.org which contains sources for Illumos (FOSS fork of OpenSolaris) in project illumos-gate. You can create your own

cross-reference with OpenGrok tool: <https://github.com/OpenGrok/OpenGrok>.

Of course we have to mention textual sources of information. For Linux it is:

- Documentati on/ directory in kernel sources
- [Linux Kernel Mailing List](#)
- [Linux info from source](#)
- Robert Love book "Linux Kernel Development"
- [Linux Device Drivers Book](#)

Some sources about Solaris:

- solaris.java.net — remnants of old OpenSolaris site
- Richard McDougall and Jim Mauro book "Solaris(TM) Internals: Solaris 10 and OpenSolaris Kernel Architecture"
- Oracle course "Solaris 10 Operating System Internals"

Warning

Solaris sources was closed after Oracle acquisition of Sun in 2009 and some information on Solaris may be outdated.

Module 1: Dynamic tracing tools.

dtrace and stap tools

Tracing

Operating system and application are crucial parts of a computer system, but due to their colossal complexity, there are situations related to software bugs, incorrect system setup that lead to incorrect behavior. To address these issues, system administrator should perform *instrumentation* which depends on the issue arisen: it could be performance statistics collection and their analysis, debug or system audit. Two common approaches to *instrumentation* are *sampling* when you collect state of the system: values of some variables, stacks of threads, etc. at unspecified moments of time and *tracing* when you install probes at specified places of software. *Profiling* is a most famous example of *sampling*.

Sampling is very helpful when you do not know where issue happens, but it hardly help when you try to know why it happened. I.e. profiling revealed that some function, say `foo()` that processes lists of elements, consumes 80% of the time, but doesn't say why: whether some lists are too long, or they should be pre-sorted, or list is inappropriate data structure for `foo()`, or whatever. With *tracing* we can install a probe to that function, gather information on lists (say their length) and collect cumulative execution of function `foo()`, and then cross-reference them, searching for a pattern in lists whose processing costs too much CPU time.

Over time operating system kernels have grown different methods of tracing. First one and a simplest one is **counters** — each time probe fires (say, major page fault), they increase some counter. Counters may be read through `kstat` interface in Solaris:

```
# kstat -p |grep maj_fault
cpu: 0: vm: maj_fault      7588
```

Linux usually provides counters through `procfs` or `sysfs`:

```
# cat /proc/vmstat | grep pgmaj_fault
pgmaj_fault 489268
```

This approach is limited: you can't add counter for every event without losing performance, and they are usually system-wide (i.e. you can't know what process causing major-faults), or process/thread-wide.

More complex approach is **debug printing**: add a `printk()` or `cmn_err()` statement as a probe, but this approach is quite limited, because you need recompile kernel each time you need new set of probes. But if all debug printing will be enabled, you will get excessive system load. By default, most of debug printing in Solaris are disabled unless you compile a DEBUG-build, which is not publicly available. Modern Linux kernels however developed a dynamic debugging facility available via `pr_debug()`. There are several **static probes** which are deactivated on systems start, but can be activated externally: *ftrace* and *kprobes* in Linux and *TNF* on Solaris, but amount of information provided by them is still limited, and *ftrace/kprobes* are requiring writing kernel modules which is not convenient and dangerous.

So, generally speaking, that approaches provide very limited set of data at very limited set of tracing points. The only approach that widens that limits is **kernel debugger**, but because each breakpoint halts system, **it cannot be used on production systems**. The answer to them are **dynamic tracing** which is the topic of this book.


Dynamic tracing




Unlike other approaches, dynamic tracing tool embeds tracing code into working user program or kernel, without need of recompilation or reboot. Since any processor instruction may be patched, it can virtually access any information you need at any place.

Solaris DTrace development was begun in 1999, and it became part of Solaris 10 release. Along with other revolutionary Solaris 10 features, it shaken world of operating systems, and still evolve. You may found more information about DTrace history here: [Happy 5th Birthday, DTrace!](#).

Here are some DTrace information sources:

- [Oracle Wiki](#)
- [DTrace at SolarisInternals wiki](#)
- «Solaris Performance and Tools» book
- «DTrace - Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD» book
- [Solaris Dynamic Tracing Guide](#)

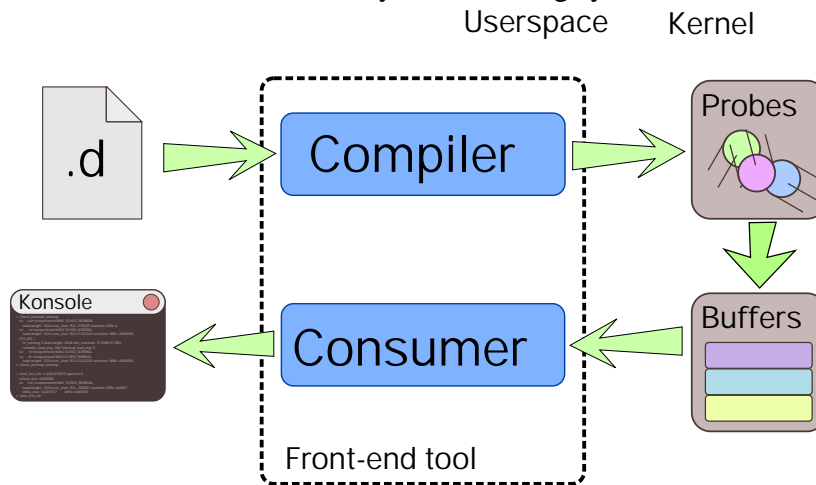
During course we will refer to *Solaris Dynamic Tracing Guide* with the following sign: 

DTrace was open-sourced as a part of OpenSolaris, but due to license incompatibility, it can't be merged with Linux Kernel. Several ports existed, but they lacked of proper support. The only stable port is provided in Unbreakable Enterprise Kernel by Oracle for their own Linux distribution which is not wide-spread. There were attempt to develop another clone of DTrace — DProbes, but it was a failure. Over time three major Linux players: Red Hat, Hitachi, IBM presented dynamic tracing system for Linux called [SystemTap](#). It has two primary sources of information: [SystemTap Language Reference](#) to which we will reference with icon  [SystemTap Tapset Reference Manual](#) to which we will reference with icon . Of course, there is a Unix manual pages, to which we will refer with icon .

SystemTap has to generate native module for each script it runs, which is huge performance penalty, so as alternative to it, [Ktap](#) is developing. Its language syntax shares some features with SystemTap, but it uses Lua and LuaJIT internally which makes it faster than SystemTap. Modern kernel versions has eBPF integrated, and there is experiment on using it as a platform for generating probe code, but it is far from final stage as of kernel version 4.1. Finally, there is a [sysdig](#) which is scriptless. Another implementation of Linux tracing is [LTTng](#). It had used static tracing and required kernel recompilation until version 2.0, but currently utilizes *ftrace* and *kprobe* subsystems in Linux kernel. As name of the book states, it describes

SystemTap and DTrace.

Here are the workflow of dynamic tracing systems:

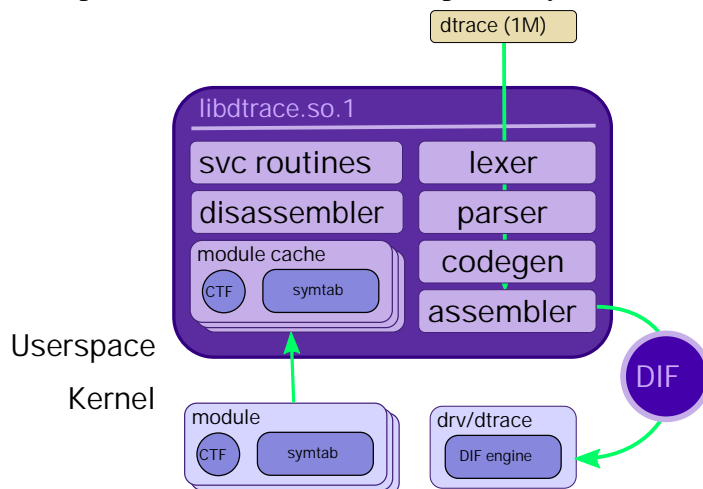


Dynamic tracing system logic is quite simple: you create a script in C-like language which is translated to a probe code by a *compiler*. DTrace scripts are written in D (do not disambiguate with D language from digital mars) have extension .d, while SystemTap scripts have extension .stp and written in SystemTap Language (it doesn't have special name). That *probe code* is loaded into kernel address space by a kernel module and patches current binary code of kernel. Probes are writing collected data to intermediate *buffers* that are usually lock-free, so they have small effect on kernels performance and doesn't need to switch context to a tracing program. Separate entity called *consumer* reads that buffers and writes gathered data into terminal, pipe or to a file.

DTrace

DTrace is shipped with Solaris from version 10, no additional actions needed to install it. It also doesn't need any changes to kernel code: it relies on CTF sections, symbol tables and static tracing points that are included into Solaris Kernel binaries.

The heart of DTrace is `libdtrace.so.1` library which contains compiler that translates script in D language to a *DTrace Intermediate Format* (DIF). That format is machine codes of simplified RISC which are interpreted by `drv/dtrace` driver:



DTrace primary front-end tool is `dtrace(1M)` which act both as compiler and consumer and uses `libdtrace.so.1` facilities to do that. There are other front-ends: `trapstat(1M)` and `lockstat(1M)`, but `libdtrace.so.1` APIs are open, so you can create your own front end for that (i.e. for Java using JNI). We will refer to `dtrace(1M)` as DTrace further in a book.

DTrace tool

DTrace supports three launch modes:

- Script is passed as command line argument: `# dtrace -n 'syscall::write:entry { trace(arg0); }'`
- Script is located in separate file: `# dtrace -s syscall.s.d [arguments]` In that case you may pass arguments, for example user ID for traced processes or disk name for which you trace block input-output. In this case arguments will be accesible in variables `$1`, `$2`, ... `$n`. Note that because there is no special handling for string arguments, you may need duplicate quotes (double-quotes needed by DTrace): `# dtrace -s syscall.s.d '"mysql d"'`
- Explicitly passing name of probe: `dtrace [-P provider] [-m module] [-f function] [-n name]`

Here are some useful command line options:

- `-l` — lists all available probes. Can be filtered using options `-P`, `-m`, `-f` or `-n` or using `grep`. I.e.:

```
# dtrace -l -P io
ID      PROVIDER      MODULE      FUNCTION NAME
800          io      genunix      bi odone done
801          io      genunix      bi owai t wai t-done
802          io      genunix      bi owai t wai t-start
```

- `-q` — enables quiet mode. By default DTrace prints probe id, its name and CPU number when probe fires. `-q` disables that.
- `-w` — allows *destructive* actions, for example system panics or breakpointing applications. That actions may be forbidden globally by setting kernel tunable `dtrace_destructive_disable`.
- `-o FILE` — redirects output to a file. If file already exists, it **appends** to it.
- `-x OPTION[=VALUE]` — sets one of DTrace tunables. Here are some useful tunables:
 - `bufsize` — size of consumer buffer (same as `-b`). Note that consumer buffers are per-cpu.
 - `cpu` — processor on which tracing is enabled (same as `-c`)
 - `dynvarsize` — size of buffers for dynamic variables (associative arrays in particular)
 - `quiet` — quiet mode (same as `-q`)
 - `flowindent` — print probes in tree mode with indentation. See more in [Dynamic code analysis](#).
 - `destructive` — enables destructive mode (same as `-w`). These options may be set inside script using pragma directive: `#pragma D option bufsize=64m`
- `-C` — call C preprocessor `cpp(1)` before script compilation. That allows handling C preprocessor directives such as `#include`, `#define`, `#ifdef` and so on. There are some extra preprocessor-related options:

- `-D MACRO[=SUBSTITUTION]` — defines preprocessor macro. `-U` undefines it.
- `-I PATH` — adds a path to include files
- `-H` — prints included files
- `-A` and `-a` — enable anonymous tracing which is used to trace system's boot and allows early loading of `drv/dtrace`
- `-c COMMAND` and `-p PID` — attaches tracing to a running command or starts new one

DTrace example

Let's create script `test.d` with following contents:

```
#!/usr/sbin/dtrace -qs
#pragma D option flowindent
#pragma D option dynvarsize=64m

syscall::write:entry
/pid == $target/
{
    printf("Written %d bytes\n", arg2);
}
```




Launch it with following options:

```
root@host# chmod +x /root/test.d
root@host# /root/test.d -c "dd if=/dev/zero of=/dev/null count=1"
```

Q: One by one, remove options `flowindent` and `-q` from script. What changed?

Q: Calculate number of probes that are provided by `fbt` provider: `# dtrace -l -P fbg | wc -l`

References

-  [dtrace\(1M\)](#)
-  [dtrace\(1M\) Utility](#)
-  [Options and Tunables](#)

SystemTap

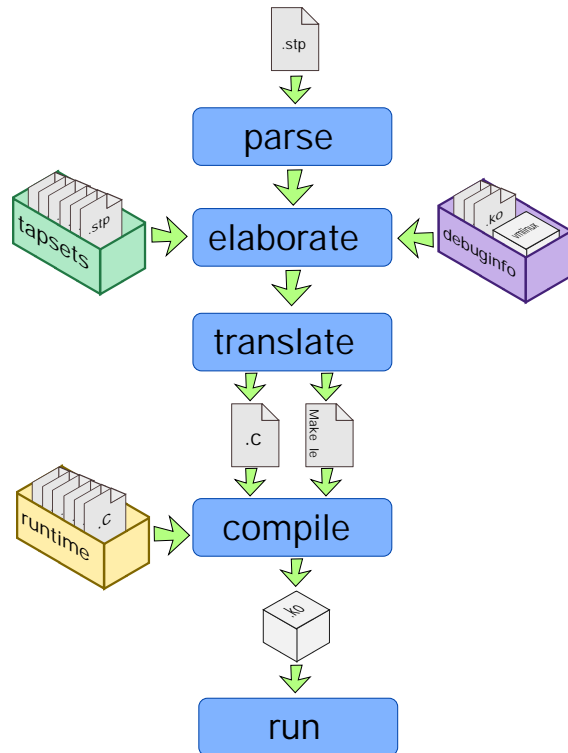
SystemTap is not part of Linux Kernel, so it have to adapt to kernel changes: i.e. sometimes runtime and code-generator have to adapt to new kernel releases. Also, Linux kernels in most distributions are stripped which means that debug information in DWARF format or symbol tables are removed. SystemTap supports *DWARF-less* tracing, but it has very limited capabilities, so we need to provide DWARF information to it.

Many distributions have separate packages with debug information: packages with `-debuginfo` suffix on RPM-based distributions, packages with `-dbg` on Debian-based distributions. They have files that originate from same build the binary came from (it is crucial for SystemTap because it verifies buildid of kernel), but instead of text and data sections they contain debug sections. For example, RHEL need `kernel-devel`, `kernel-debuginfo` and `kernel-debuginfo-common` packages to make SystemTap working. Recent SystemTap versions have `stap-prep` tool that automatically install kernel debuginfo from appropriate repositories with correct versions.

For vanilla kernels you will need to configure `CONFIG_DEBUG_INFO` option so debug information will be linked with kernel. You will also need to set `CONFIG_KPROBES` to allow

SystemTap patching kernel code, `CONFIG_RELAY` and `CONFIG_DEBUG_FS` to allow transfer information between buffers and consumer and `CONFIG_MODULES` with `CONFIG_MODULE_UNLOAD` to provide module facilities. You will also need uncompressed `vmlinux` file and kernel sources located in `/lib/modules/$(uname -r)/build/`.

SystemTap doesn't have VM in-kernel (unlike DTrace and KTap), instead it generates kernel module source written in C than builds it, so you will also need a compiler toolchain (make, gcc and ld). Compilation takes five phases: *parse*, *elaborate* in which tapsets and debuginfo is linked with script, *translate* in which C code is generated, *compile* and *run*:



SystemTap uses two sets of libraries during compilation process to provide kernel-version independent API for accessing. *Tapsets* are a helpers that are written in SystemTap language (but some parts may be written in C) and they are plugged during *elaborate* stage. *Runtime* is written in C and used during *compile* stage. Because of high complexity of preparing source code and compiling that, SystemTap is slower than a DTrace. To mitigate that issue, it can cache compiled modules, or even use compile servers.

Unlike DTrace, SystemTap has several front-end tools with different capabilities:

- `stapio` is a consumer which runs module and prints information from its buffer to a file or stdout. It is never used directly, but called by `stap` and `staprun` tools.
- `stap(1)` includes all five stages and allow to stop at any of them. I.e. combining options `-k` and `-p 4` allow you to create pre-compiled `.ko` kernel module. Note that SystemTap is very strict about version of kernel it was compiled for.
- `staprun(1)` allows you to reuse precompiled module, instead of start compilation from scratch.

Warning

If `stap` parent is exited, than `killall -9 stap` won't finish `stapio` daemon. You have to signal it with `SIGTERM`: `killall -15 stap`

stap

Like many other scripting tools, SystemTap accepts script as command line option or external file, for example:

- Command-line script is passed with `-e` option: `# stap -e 'probe syscall.write { printf("%dn", $fd); }' [arguments]`
- External file as first argument: `# stap syscall.s. [arguments]` SystemTap command line arguments may be passed to a script, but it distinguishes their types: numerical arguments are accessible with `$` prefix: `$1, $2 ... $n` while string arguments have `@` prefix: `@1, @2 ... @n`

Here are some useful `stap(1)` options:

- `-l PROBESPEC` accepts probe specifier without `probe` keyword (but with wildcards) and prints all matching probe names (more on wildcards in [Probes](#)). `-L` will also print probe arguments and their types. For example: `# stap -l 'scsi.*'`
- `-v` — increases verbosity of SystemTap. The more letters you passed, the more diagnostic information will be printed. If only one `-v` was passed, `stap` will report only finishing of each stage.
- `-p STAGE` — ends `stap` process after *STAGE*, represented with a number starting with 1 (*parse*).
- `-k` — `stap` tool won't delete SystemTap temporary files created during compilation (sources and kernel modules kept in `/tmp/stapXXXX` directory),
- `-g` — enables Guru-mode, that allows to bind to blacklisted probes and write into kernel memory along with using Embedded C in your scripts. Generally speaking, it allows dangerous actions.
- `-c COMMAND` and `-x PID` — like those in `DTrace`, they allow to bind SystemTap to a specific process
- `-o FILE` — redirects output to a file. If it already exists, SystemTap **rewrites** it.
- `-m NAME` — when compiling a module, give it meaningful name instead of `stap_`.

When SystemTap needs to resolve address into a symbol (for example, instruction pointer to a corresponding function name), it doesn't look into libraries or kernel modules. Here are some useful command-line options that enable that:

- `-d MODULEPATH` — enables symbol resolving for a specific library or kernel module. Note that in case it is not provided, `stap` will print a warning with corresponding `-d` option.
- `--l dd` — for tracing process — use `l dd` to add all linked libraries for a resolving.
- `--all -modules` — enable resolving for all kernel modules

SystemTap example

Here is sample SystemTap script:

```
#!/usr/sbin/stap
```

```
probe syscall.write
{
    if(pid() == target())
        printf("Written %d bytes", $count);
}
```

Save it to `test.stp` and run like this:





```
root@host# stap /root/test.stp -c "dd if=/dev/zero of=/dev/null count=1"
```

Q: Run SystemTap with following options: `# stap -vv -k -p4 /root/test.stp` , find generated directory in `/tmp` and look into created C source.

Q: Calculate number of probes in a `syscall` provider and number of variables provided by `syscall.write` probe:

```
# stap -l 'syscall.*' | wc -l
# stap -L 'syscall.write'
```

References

-  [STAP](#)
-  [STAPRUN](#)
-  [The stap command](#)
-  [Literals passed in from the stap command line](#)

Safety and errors

Like we said, dynamic tracing is intended to be safely used in production systems, but since it is intrusive to an OS kernel, there is a room for unsafe actions:

- Fatal actions inside kernel like reading from invalid pointer (like `NULL`) or division by zero will cause a panic following by a reboot.
- If probes are executed for too much time (or too often), it will induce performance degradation in a production system, or at least give results that are very different than from a non-traced system (i.e. making racing condition that you debug a very rare).
- Dynamic tracing systems allocate memory for their internal memory which should be limited.

That leads to a common principle for all dynamic tracing systems: **add some checks before executing actual tracing**. For example, DTrace has *Deadman Mechanism* that detects system unresponsiveness induced by DTrace and aborts tracing, while SystemTap monitors time spent in each tracing probe. The common error messages you'll see due to that are processing aborted: `Abort due to systemic unresponsiveness in DTrace` and `SystemTap probe overhead exceeded threshold`.

Unfortunately, SystemTap is not that affective as DTrace, so probe overhead error message is a common thing. To overcome this error in SystemTap you can recompile your script with `-t` option to see what probes are causing overload and try to optimize them. You may also increase threshold by setting compile macro (with `-D` option) `STP_OVERLOAD_THRESHOLD` in percent of overall CPU time or completely disable it with `STP_NO_OVERLOAD` macro (latest SystemTap versions support it via `-g --suppress-time-limits`).

Another resource that is limited is memory. Memory limitations are implemented pretty simple: all allocations should be performed when script is launched and with a fixed size. For associative arrays, SystemTap limits number of entries it can hold (changeable by setting macro `MAXMAPENTRIES`), and `ERROR: Array overflow, check MAXMAPENTRIES near identifier 't' at 1:30`, while DTrace limits overall space for them via `dynvarsize` tunable and it will print it as `dynamic variable drops error`. Note that SystemTap still can exhaust memory if you create too many associative arrays, but this will be handled by OOM which will simply kill `stap` tool. Both DTrace and SystemTap limit size of strings used in scripts.

Transport buffer between probes and consumer is also limited, so if you will print in probes faster than consumer can take, you will see `There were NN transport failures error in`

SystemTap or DTrace drops on CPU X error on DTrace. The answer to that problem is simple: be less verbose, take data from buffer more frequently (regulated by `cleanrate` tunable in DTrace) or increase buffer size (`-b` option and `bufsize` tunable in DTrace and `-s` option in SystemTap).

Both DTrace and SystemTap are also using special handlers for in-kernel pagefaults, that will disable panic and handle fault if it was caused by tracing. For example DTrace will complain with error on enabled probe ID 1 (ID 78: `syscall::read:entry`): `invalid alignment (0x197) in action #1 at DIF offset 24` and continue execution, while SystemTap will print `ERROR: read fault [man error::fault] at 0x00000000000024a8 (addr) near operator '@cast' at :1:45` and stop tracing. Note that SystemTap provides more context than DTrace. That is because error-checking is performed in generated C code, not by RISC-VM inside driver.

Demonstration scripts



These scripts have errors which cause error messages described above. For associative arrays we will use timestamp to flood array with unrepeated data:

```
# dtrace -n 'int t[int];
    tick-1ms {
        t[timestamp] = timestamp }'
# stap -e 'global t;
    probe timer.ms(1) {
        t[local_clock_ns()] = local_clock_ns(); }'
```

To demonstrate segmentation violation, you can interpret wrong integral argument (fd for Solaris and file position in Linux) as pointer to a thread structure and try to access its field.

```
# dtrace -n 'syscall::read:entry {
    trace(((kthread_t*) arg0)->t_procp); }' -c 'cat /etc/passwd'
# stap -e 'probe kernel.function("vfs_read") {
    println(@cast($count, "task_struct")->pid); }' -c "cat /etc/passwd"
```

References

-  [Safety and security](#)
-  [Performance Considerations](#)
- SystemTap Wiki: [Exhausted resources](#)

Stability

Another problem to which dynamic tracing systems face is stability of in-kernel interfaces. While system calls never change their interface due to backwards compatibility (if something need to be changed, new system call is introduced[†]), internal kernel function often do that especially if they not a public API for a drivers. Dynamic tracing languages provide mechanisms to avoid direct use of in-kernel interface by hiding them in abstractions:

Stability	Data access	
	<i>DTrace</i>	<i>SystemTap</i>
High	<i>translators</i> , i.e. <code>fileinfo_t</code>	tapset variables
Lowest	Global variables and raw arguments like <code>args[0]</code> or <code>(struct_t*) arg0</code>	Raw arguments like <code>\$task</code> or <code>@cast(\$task, "task_struct")</code>

<i>Stability</i>	Tracepoints	
	<i>DTrace</i>	<i>SystemTap</i>
High	statically defined tracing providers (like <code>io</code> and many others)	tapset aliases, i.e. <code>vm.kfree</code>
Mediocre	static tracepoints with <code>sdt</code> provider	statically defined ftrace probes like <code>kernel.trace("kfree")</code>
Lowest	<code>fbt</code> and <code>pid\$</code> providers	DWARF probes like <code>kernel.function("kfree")</code>

To achieve maximum script portability, you should pick highest stability options wherever possible. Downside of that approach is that it provides fewer information than you could access with other approaches. These options will be described in [Translators and tapsets](#) section of next module.

Linux kernel is changing faster: it has stable releases each 2-3 months, and moreover, its builds are configurable, so some features present in one kernel may be disabled in another and vice versa which makes stability is much more fragile. To overcome that, SystemTap Language has conditional compilation statements which like in C allow to disable certain paths in code. Simplest conditional compilation statements are `@defined` which evaluates to true if variable passed to it is present in debug information and `@choose_defined` which chooses from several variables. It also support ternary conditional expression:

```
%( kernel_v >= "2.6.30"
    %? count = kernel_long($cnt)
    %: count = $cnt
%)
```

Here, `kernel_v` is numerical version of kernel without suffix (for version with suffix, use `kernel_vr`). SystemTap also defines `arch` variable and `CONFIG_*` tokens similar to configuration options. These options are not available in Embedded C, use traditional preprocessor there.

Finally, if some probe is missing from kernel, script compilation will fail. DTrace allow to ignore such errors by passing `-Z` command line option. In SystemTap you may add `?` at the end of probe name to make this probe optional.

Notes

† — unless you are running Solaris 11 which was deprecated and obsoleted many of its system calls..

References

-  [Conditional compilation](#)
-  [Stability](#)

Module 2: Dynamic tracing languages

Introduction

Both DTrace and SystemTap languages have C-like syntax for dynamic tracing scripts. Every script is a set of probes, and each of them binds to a certain event in kernel or application, for example dispatching of a process, parsing SQL query, etc. Each probe may have a predicate which acts as a filter of unnecessary probes, i.e. if you want to trace specific process or specific kind of query.

Each script consists of global variables declarations followed by probes, and possibly function declarations. In SystemTap each declaration is preceded by `global`, `function` or `probe` keyword:

```
global counter;
function inc_counter() {
    ++counter;
}
probe timer.s(1) {
    inc_counter();
    println(counter);
}
```

Note

Trailing semicolons may be omitted in SystemTap Language, but we will use them in our demonstration scripts to improve readability.

Same works for DTrace, but the syntax of definitions is different:

```
int xcounter;
tick-1s {
    ++xcounter;
    trace(xcounter);
}
```

DTrace language is limited due to safety reasons, so it doesn't support loops and conditional statements. Conditional branch in DTrace may be emulated using predicates, and also a limited support of ternary operator `?:` is available. SystemTap, on the other hand, supports wider subset of C language: it has `for`, `while`, `if/else`, `foreach` statements, and

break/continue for controlling loop behavior.

SystemTap supports declaration of functions:

```
function dentry_name: string(dentry: long) {
    len = @cast(dentry, "dentry")->d_name->len;
    return kernel_string_n(@cast(dentry, "dentry")->d_name->name, len);
}
```

In this example, function `dentry_name()` accepts `dentry` argument of type `long` (in this case, `long` is equivalent to a missing pointer type) and returns a string. It converts received pointer to a type `struct dentry`, extracts string from it and returns it.

DTrace doesn't have a functions, but you may use C macro in simple cases:

```
#define CLOCK_T0_MS(clock)      (clock) * (`nsec_per_tick / 1000000)
```

SystemTap language supports try/catch statement to handle tracing errors which were described in [Safety and errors](#) section:

```
try {
    /* Errorneous expression: read integer on address 4 */
    println(kernel_int(4));
}
catch(msg) {
    /* Ignore errors or print message `msg` */
}
```

There is a hackish way of building loops in DTrace using timer probes:

```
int i;
BEGIN {
    i = 10;
}
tick-1ms
/--i >= 0/ {
    printf("Hello, world!\n");
}
```

This script prints "Hello, world" phrase 10 times. Note that there is a delay of 1 millisecond between loop cycles, but it won't be noticed due to larger buffer switching intervals.

Finally, SystemTap have Embedded C extension (enabled only in Guru-Mode or in tapsets), which allow to write raw C code compiled directly to module's code without passing first three stages of translation:

```
function task_val_id_file_handle: long (task: long, fd: long) %{ /* pure */
    [...]

    rcu_read_lock();
    if ((files = kread(->files))) {
        filp = fcheck_files(files, STAP_ARG_fd);
        STAP_RETVALUE = !!filp;
    }

    CATCH_DEREF_FAULT();
    rcu_read_unlock();
%}
```

This example is taken from *pfiles.stp* sample. It has to grab RCU lock to access file pointer safely, which is done by direct call to `rcu_read_lock()` and `rcu_read_unlock()` functions. Note that to access arguments and return value it has to use names prefixed with `STAP` (in early versions of SystemTap there were magic pointers `THIS` and `CONTEXT` for this).

To read pointer safely it uses `kread()` function.

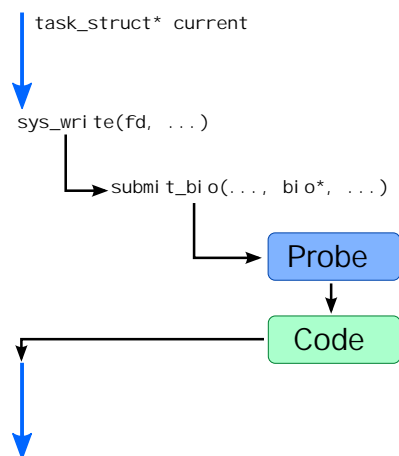
Embedded C part starts with `%{` and ends with `%}` and may be used as function body, and in global scope if you need extra includes.

Probes

Definition

Probe — is a handler of kernel or application event. When probe is installed into kernel or application, so it can handle such event, we will call it *attaching a probe* or *binding a probe*. When event occurs and probe code is executing, we will say *probe is firing*.

For example, let's see how synchronous writing to a disk is performed in Linux and what can be traced:



When process wants to start synchronous write, it issues `write()` system call and by doing that it transfers control to a kernel code, to a `sys_write()` function in particular. This function eventually calls a `submit_bio()` function which pushes data from user process to a queue of corresponding disk device. If we attach probes to these functions, we can gather the following information:

- Process and thread which started input/output which is accessible via global current pointer.
- File descriptor number which is passed as first argument of `sys_write` and called `fd`.
- Disk I/O parameters such as size and requested sector from `bio` structure.

To satisfy this requirements, tracing languages provide mechanisms of defining probes. Definition of SystemTap probe begins with `probe` keyword followed by probe name and body of probe handler. Name is a dotted-symbol sequence, where each symbol may have optional parameters in braces. SystemTap supports wildcards in probe names or several probe names in probe clause if you need to use same handler for multiple probes. For example:

```
probe kernel.function("vfs_*") {
    // Actions
}
```

```
probe timer.ms(100) {
    // Actions
}
```

```
probe scheduler.cpu_on {
    // Actions
}
```

```
}
```

Probe names in DTrace are four identifiers separated by colons: Provider: Module: Function: Name[-Parameter].

- *Provider* is a hint to DTrace on how to attach a probe. Different providers usually have different mechanisms of attachment.
- *Function* and *Module* are relate to a code location where probe will be installed.
- *Name* and optional parameters provide meaningful names to a event which will be handled in a probe. For example:

```
fbt::fop_*:entry {  
    // Actions  
}
```

```
profile-100ms {  
    // Actions  
}
```

```
sched:::on-cpu {  
    // Actions  
}
```

DTrace support wildcards, and some parts of probe name may be omitted: `fbt: *: *: entry`, `fbt:::entry` are equivalent, while `fop_read:entry` is shorter form of `fbt:genunix:fop_read:entry`.

Probe names may be combined using comma, and have multiple probes attached to same event, for example in SystemTap:

```
probe syscall.read {  
    /* Preparations */ }  
probe syscall.read, syscall.write {  
    /* Common actions for read and write */ }
```

Or in DTrace:

```
syscall::read:entry {  
    /* Preparations */ }  
syscall::read:entry, syscall::write:entry {  
    /* Common actions for read and write */ }
```

First probe body going in script executes first.

If DTrace or SystemTap fail to find a probe, it will abort script translation. To overcome that, use `-Z` option can be supplied to `dtrace` or question mark has to be added to a probe name in SystemTap:

```
probe kernel.function("unknown_function") ?
```

Function boundary tracing

Function boundary tracing is the largest and most generic class of tracing. Function boundary probes attach to entry point or exit (hence bounds) from a function. Since most functions begin with saving stack and end with `retq` or similar instruction, tracer simply patches that instruction, by simply replacing it to interrupt or call (depending on a platform). That interrupt is intercepted by probe code which after execution returns control to function, like in `submit_bio` case described above. Here are similar example for Solaris and DTrace:

```
bdev_strategy:    pushq    %rbp                €   int    $0x3  
bdev_strategy+1:  movq     %rsp, %rbp          movq    %rsp, %rbp
```

```
bdev_strategy+4:  subq    $0x10,%rsp          subq    $0x10,%rsp
```

Warning

Userspace probes will be covered in [Module 5](#).

SystemTap

SystemTap function probe names have the following syntax:

```
{kernel | module("module-pattern")}.function("function-pattern") [. {call | return | inline}]
```

where *kernel* means that function is statically linked into `vmlinux` binary, while *module* followed by its name pattern seeks inside module. *module-pattern* is usually a name of a kernel module, but may contain wildcards such as `*`, `?`, and character class `[]`. *function-pattern* is a bit more complex: along with direct specifying its name, or using wildcards, it also support at-suffix followed by a source file name and optional source line number:

```
function-name[@source-path[:line-number|:first-line-last-line|+relative-line-number]]
```

Wildcards can be used in *source-path*.

Function probe name ends with suffix defining a point in function where probe should be attached:

- `.call` is used to attach entry point non-inlined function, while `.inline` is used to attach first instruction of inlined function;
- `.return` is used for return points of non-inlined functions;
- empty suffix is treated as combination of `.call` and `.inline` suffixes.

Along with attaching to any line through *relative-line-number* syntax, SystemTap allows to patch any kernel instruction:

```
kernel.statement(function-pattern)
kernel.statement(address).absolute
module(module-pattern).statement(function-pattern)
```

Note

When we will use following syntax for probe names:

- `{x|y|z}` — one of the options
- `[optional]` — optional part of name which can be omitted
- *parameter* — changeable parameter which can have different values described below

Another option is DWARF-less probing which uses kprobes if debug information is not available:

```
kprobe[. module("module-pattern")].function(function-pattern) [. return]
kprobe.statement(address).absolute
```

DTrace

DTrace function tracing is much simpler: it is supported by `fbt` provider which has only two probe names: `entry` for entry point and `return` for exit from function. For example:

```
fbt:e1000g:e1000g_*.entry
```

System call tracing

A simplest variant of function boundary tracing is a system call tracing. In **SystemTap** they are implemented as aliases on top of corresponding functions and accessible in `syscall` tapset:

```
syscall::system-call/-name[. return]
```

DTrace uses different mechanisms for attaching to a system calls: it is implemented through driver `systrace` and patches system call entry point in a `sysent` table. A syntax for probes, however, is similar to `fbt`:

```
syscall::system-call/-name: {entry|return}
```

Note that if you omit provider name, some probes will match both function and system calls, so probe will fire twice.

Statically defined tracing

Sometimes is function boundary tracing is not enough: an event may occur inside function, or may be spread through different functions. In **DTrace** and Solaris, for example, there are two implementations of scheduler functions that are responsible for stealing task from cpu: older `di_sp_getbest` and newer and available in newer versions of Solaris: `di_sp_getkpq`. But they both provide `steal` probe that fires when dispatcher moves a thread to idle CPU: `sdt::steal` or simply `steal`. You can still distinguish these probes by explicitly setting function name: `sdt::di_sp_getbest:steal`.

Another use-case for statically defined probes is long functions that contain multiple steps, like handling TCP flags and advancing FSM of TCP-connection or handling multiple requests at once. For example, Solaris handles task queues like this:

```
static void taskq_thread(void *arg)
{
    /*...*/

    for (;;) {
        /*...*/
        tqe->tqent_func(tqe->tqent_arg);
        /*...*/
    }
}
```

It is impossible to attach probe to a `tqent_func` because it is dynamically set, but Solaris provides `taskq-exec-start` and `taskq-exec-end` probes which are set around `tqent_func` call.

Probes may be added to kernel using `DTRACE_PROBEn` macros, i.e.:

```
DTRACE_PROBE3(steal, kthread_t *, tp, cpu_t *, tcp, cpu_t *, cp);
```

Statically defined probes are extremely useful in **DTrace** because it doesn't provide access to local variables or tracing any instruction of kernel.

In Linux statically defined tracing were added in version 2.6.24, as kernel markers, but it is deprecated now and replaced by *FTrace* subsystem. **SystemTap** supports both:

```
kernel.trace("tracepoint-pattern")
kernel.mark("mark") [. format("format")]
```

Events provided by *FTrace* tracepoints are defined using `TRACE_EVENT` macro and later used by calling `trace_` function. For example:

```
TRACE_EVENT(sched_switch,
            [...])

[...]
static inline void
prepare_task_switch(struct rq *rq, struct task_struct *prev,
                   struct task_struct *next)
{
    trace_sched_switch(prev, next);
    [...]
```

In ideal case, statically defined probe is just a nop instruction or a sequence of them. In Linux, however it involves multiple instructions.

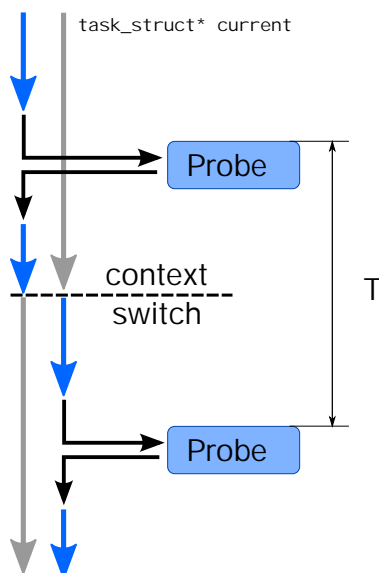
Alias probes

Function boundary probes lack of stability, so dynamic tracing provide intermediate layer that we will refer as *alias probe*. Alias probe is defined in kernel as statically defined probe, like Solaris does, or provided by tapset in SystemTap and converts and extract data from its arguments using variables in SystemTap or translators in DTrace. Creating aliases will be covered by [Translators and tapsets](#) topic.

Timers and service probes

These probes are not related to a kernel events, but to execution of tracing script itself. They may trace starting of script, end of it and occurred error, thus handle initialization of global variables and printing results on end of script execution. Another kind of service probe is timer probe, which is called every ΔT time on one or all system CPUs. Timers are useful for creating stat-like utilities which print data every second or for profiling.

Take for example profiler probe which records task name from current pointer (it always points to task executing on CPU now):



So if we count that timer probe has fired two times, once in context of left process and once in context of right process, we can conclude that they both consume 50% of CPU time, like `prstat` and `top` utilities do. Profiling will be covered in [Profiling](#) section of Module 3.

In **SystemTap** service probes have following syntax:

```
{begin|end}[(priority)]
error
```

Where *priority* is a number which defines an order of executing *begin* and *end* statements. Explicit order is needed because *begin* and *end* probes may be specified by tapsets.

Timers are specified in a following form:

```
timer.uni t(period)[. randomi ze(devi ati on)]
```

Timer probes are executed on single CPU which *id* is undefined. *randomize* allows to make period a uniform distributed random value.

For profiling use *timer.profile* probe which fires on all CPUs and attaches to system timer. You may also use *perf*-probes for profiling.

DTrace has **BEGIN** and **END** probes in *dtrace* providers. Timers are handled by *profile* provider which provide two types of probes: *tick* which fires on any CPU once at a time period, and *profile* which does the same for all CPUs. Probe name is followed by a parameter with number and unit:

```
[profile::]{tick|profile}-period[unit]
```

For example *tick-1s* will fire every second. Note that, not all platforms may provide nanosecond or microsecond resolution, so probe will fire rarely when it should be. Timer probes with period above 1 millisecond are usually safe to use.

SystemTap and DTrace support the following timer units:

Unit		
ns	nsec	nanoseconds
us	usec	microseconds
ms	msec	milliseconds
s	sec	seconds
m	min	minutes (DTrace)
h	hour	hours (DTrace)
d	day	days (DTrace)
hz		
jiffies		

Example

Lets take following C code as an example (assuming it is located in kernel-space) and see how its lines may be probed:





```
1 float tri_area(float a, float b,
2               float angle) {
3     float height;
4
5     if(a = 180.0 || angle or
    trace_tri_angle_height(h);
12
13     return a * height;
14 }
```

Lineno	DTrace	SystemTap
1	fbt::tri_area:entry	kernel.tri_area("tri_area").call

Lineno	DTrace	SystemTap
7	fbt::tri_area:return	kernel.tri_area("tri_area").return kernel.statement("tri_area+6")
9		kernel.statement("tri_area+8")
11	sdt::tri_area:triangle-height	kernel.trace("triangle_height")
13	fbt::tri_area:return	kernel.tri_area("tri_area").return kernel.statement("tri_area+12")

References

DTrace

-  [D Program Structure](#)
-  [fbt Provider](#)
-  [sdt Provider](#)
-  [profile Provider](#)

SystemTap

-  [STAPPROBES](#)
-  [Probe points](#)

Arguments

When you bind a probe, you need to collect some data in it. In C, data is usually passed as arguments to a function, or returned as *return value*. So, when you bind a function boundary tracing probe, you may need to gather them. Argument extraction relies on calling conventions, and extracts data directly from registers or stack.

For example, let's look at Solaris kernel function from ZFS: `void spa_sync(spa_t *spa, uint64_t txg);`. First argument is ZFS representation of a pool, second is 64-bit unsigned integer which is transaction group number. So when we bind a probe to a `spa_sync`, we can print both of them:

```
# dtrace -qn '
::spa_sync:entry {
    printf("synced txg=%d [%s]\n",
        args[1], args[0]->spa_name); }'
```

DTrace supports two forms of arguments: `arg0, arg1 ... argN` are `uint64_t` values, while `args[0], args[1] ... args[N]` have actual types if DTrace is able to extract them (i.e. DTrace forbids type hinting for unstable probes). If `args[N]` is unavailable, you can still treat `argN` as pointer and covert it as you want:

```
# dtrace -qn '
::spa_sync:entry {
    printf("synced txg=%ld [%s]\n",
        (long) arg1, ((spa_t*) arg0)->spa_name); }'
```

DTrace supplies two arguments for return probes: `arg0` is an instruction pointer to a caller, and `arg1` or `args[1]` is a return value.

DWARF format used in Linux is richer than CTF from Solaris and saves not only argument types, but their names too. They are provided in SystemTap in separate namespace beginning

with `$` and followed by name of argument. It provides access to locals as well as arguments. However, some of them may be unavailable at the probe, because they are overwritten by other data (which is called *optimized out*). For example, let's look at `vfs_read` function from Linux kernel:

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;

    [...]

    return ret;
}
```

Unfortunately, variable `ret` is inaccessible at the return probe, but you can still get it from `%rax` register on `x86_64` which is used for saving return values. SystemTap supplies return values in `$return` variable:

```
# stap -e '
    probe kernel.function("vfs_read").return {
        printf("VARS: %s\nreturn: %d\n", $$vars, $return);
        exit(); }'
VARS: file=0xcfa79580 buf=0xbf9fa8b8 count=0x2004 pos=0xcf2e9f98 ret=?
return: 12
```

To handle such situations (and many others, i.e. when name of argument was changed in current kernel), you may use `@defined` expression, or `@choose_defined` which works like ternary operator: `@choose_defined($a, $b)` is equivalent to `@defined($a)? $a : $b`. Here is an example of `@defined`:

```
if (@defined($var->field)) {
    println($var->field);
}
```


If you want to print all arguments simultaneously, you should carefully handle each argument. However, SystemTap can do it automatically. Such strings provided in meta-variables:

- `$$parms` contains function arguments with their names
- `$$locals` contains local variables with their names
- `$$vars` contains both `$$parms` and `$$locals`
- `$$return` contains return value. An example of `$$vars` may be found above.

Finally, SystemTap allows to convert arguments to strings, including pretty representation of structure pointers when all fields are read, if trailing dollar sign is added to an argument:

```
# stap -e '
    probe kernel.function("vfs_read") {
        println($file$); }'
```

References

-  [Built-in probe point types \(DWARF probes\)](#)
- [Troublesome Context Variables](#)

Context

Definition

Probe context contains system state related to a fired probe, including:

- Register values
- Thread and process, which caused probe firing, including CPU where thread is running
- Currently executing probe

Context is provided as built-in variables in DTrace such as `execname` or as tapset functions in SystemTap such as `execname()`.

Userspace register values are available in DTrace through built-in variable `uregs`. In SystemTap, they are available through Embedded C and kernel function `task_pt_regs`, or a special Embedded C variable `CONTEXT`, see for example implementation of `uaddr()` and `print_regs()` tapset functions.

Here are some useful context information:

Description	DTrace	SystemTap
Current executing thread	<code>curthread</code>	<code>task_current()</code>
ID of current thread	<code>tid</code>	<code>tid()</code>
ID of current process	<code>pid</code>	<code>pid()</code>
ID of parent of current process	<code>ppid</code>	<code>ppid()</code>
User ID and group ID of current process	<code>uid/gid</code>	<code>uid()/gid()</code> , <code>euid()</code> , <code>egid()</code>
Name of current process executable	<code>execname</code> <code>curpsinfo->ps_fname</code>	<code>execname()</code>
Command Line Arguments	<code>curpsinfo->ps_psargs</code>	<code>cmdline_*</code>
CPU number	<code>cpu</code>	<code>cpu()</code>
Probe names	<code>probeprov</code> , <code>probemod</code> , <code>probefunc</code> , <code>probename</code>	<code>pp()</code> , <code>pn()</code> , <code>ppfunc()</code> , <code>probefunc()</code> , <code>probemod()</code>

References

-  [Built-in Variables](#)
-  [Context Functions](#)

Predicates

Predicates are usually go in the beginning of the probe and allow to exclude unnecessary data from output, thus saving memory and processor time. Usually predicate is a conditional expression, so you can use C comparison operators in there such as `==`, `!=`, `>`, `>=`, `,` and logical operators for logical AND, `||` for logical OR and `!` for logical negation, alas with calling functions or actions.

In DTrace predicate is a separate language construct which is going in slashes / immediately after list of probe names. If it evaluated to true, probe is **executed**:

```
syscall::write:entry
/pid == $target/
```

```
{
    printf("Written %d bytes", args[3]);
}
```

In SystemTap, however, there is no separate predicate language construct, but it supports conditional statement and next statement which exits from the probe, so combining them will give similar effect:

```
probe syscall.write {
    if(pid() != target())
        next;
    printf("Written %d bytes", $count);
}
```

Note that in SystemTap, probe will be **omitted** if condition in `if` statement is evaluated to true thus making this logic inverse to DTrace.

Starting with SystemTap 2.6, it supports mechanism similar to predicates which is called on-the-fly arming/disarming. When it is active, probes will be installed only when certain condition will become true. For example:

```
probe syscall.write if(i > 4) {
    printf("Written %d bytes", $count);
}
```

This probe will be installed when `i` becomes more than four.

`$target` in DTrace (macro-substitution) and `target()` context function in SystemTap have special meaning: they return PID of the process which is traced (command was provided as `-c` option argument or its PID was passed as `-p/-x` option argument). In these examples only `write` syscalls from traced process will be printed.

Warning

Sometimes, SystemTap may trace its consumer. To ignore such probes, compare process ID with `stp_pid()` which returns PID of consumer.

Sometimes, if target process forking and you need to trace its children, like with `-f` option in `truss/strace`, comparing `pid()` and even `ppid()` is not enough. In this case you may use DTrace subroutine `progenyof()` which returns non-zero (treated as true) value if current process is a direct or indirect child of the process which ID was passed as parameter. For example, `progenyof(1)` will be true for all userspace processes because they are all children to the `init`.

`progenyof()` is missing in SystemTap, but it can be simulated with `task_*` functions and the following SystemTap script (these functions are explained in [Process Management](#)):

```
function progenyof(pid:long) {
    parent = task_parent(task_current());
    task = pid2task(pid);

    while(parent & task_pid(parent) > 0) {
        if(task == parent)
            return 1;

        parent = task_parent(parent);
    }
}

probe syscall.open {
    if(progenyof(target()))
```

```

        printf(" ", pid(), execname(), filename);
    }

```

Assume that 2953 is a process ID of bash interactive session, where we open child bash and call cat there:

```

root@lctest: ~# bash
root@lctest: ~# ps
  PID TTY          TIME CMD
 2953 pts/1        00:00:01 bash
 4794 pts/1        00:00:00 bash
 4800 pts/1        00:00:00 ps
root@lctest: ~# cat /etc/passwd
[...]
```

cat is shown by this script even if it is not direct ancestor of bash process that we are tracing:

```

# stap ./progeny.stp -x 2953 | grep passwd
4801 cat /etc/passwd

```

Types and Variables

In this section we will speak about typing in dynamic tracing languages and variable scopes. Details on complex types are covered in further sections.

Variable types may be split in several categories. First and simpler one, is **scalar types** which consist of integral types: `int`, `uint32_t`, etc, floating point types are not supported. Second large group is **pointers**. Unlike C, dynamic tracing languages provide explicit **string** type. SystemTap and DTrace support **associative arrays** and **aggregations** for keeping statistics data. Finally, there is a set of **complex** types such as structures, enumerations, unions and arrays. DTrace supports complex types, their definitions and even aliasing through typedef, while in SystemTap they are implicitly used for DWARF variables, but in scripts they are explicitly available only in Embedded C.

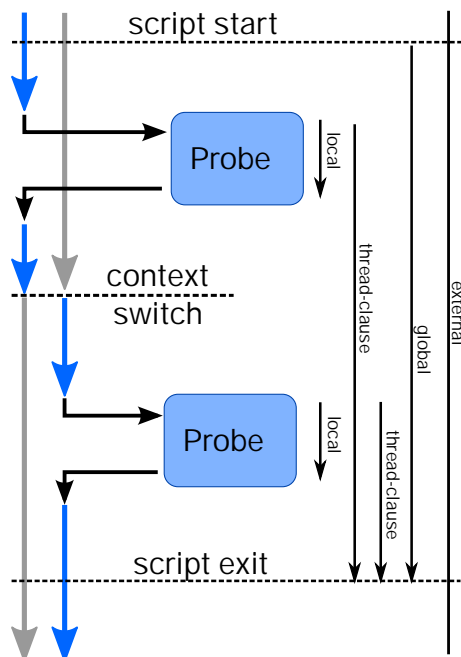
You can explicitly declare variable types in DTrace, thus `long`, `uintptr_t`, `string`, etc. are valid identifiers in it, but it is optional for non-global variables. In SystemTap, there are only two primitive types: `long` for keeping any scalar integral type or pointer, and `string` for strings. Types are explicitly specified only as return values of functions or function arguments. If types are not specified, then they are deduced from first assignment, but dynamic typing is disallowed: in case of type incompatibility error operands have `incompatible types` or `type mismatch` will be printed. DTrace also supports C-style type casting:

```

printf("The time is %lld\n", (unsigned long long) timestamp);

```

There are four variable scopes in DTrace: external, global, local and thread-local. SystemTap doesn't support thread-local variables, but it can be emulated via associative arrays.



In this image variable lifetimes are shown as arrows on the right of the drawing.

External variables

External variables are exported by kernel or application, for example tunable module parameters, thus they have longest lifespan that goes beyond running tracing scripts. In DTrace external variables are kept in separate namespace, and accessible with backtick (`) prefix:

```
# dtrace -qn '
BEGIN {
    printf("Maximum pid is %d\n", `pidmax );
    exit(0); }'
```

In earlier versions of SystemTap they can be only read by using Embedded C capabilities:

```
# stap -g -e '
function get_jiffies:long() %{
    THIS->__retvalue = jiffies;    %}
probe timer.us(400) {
    printf("The time is %d jiffies\n",
        get_jiffies());    %}'
```

Recent versions adopted a @var-expression, which accept name of variable and optionally a path to a source file where it is located like in function probes: @var("j i f f i e s").

Global variables

Global variables are created on script start and destroyed when script finishes their execution. They are often initialized by begin probes and sometimes printed in the end probe. In SystemTap global variables are declared with `global` keyword:

```
global somevar;
```

You can also put an initializer to a global variable, thus it is useful to simulate constants and enumerations:

```
global READ = 1;
```

Global variables in DTrace may be declared with type keyword, but that is optional:

```
uint32_t global var;
```

Aggregations in DTrace are implicitly global.

Global variables in probes are accessible by their names: `global var += 1;`

Local variables

Local (or *clause-local* in terms of DTrace) variables lifespan are the shortest of all which last only for single probe, or for a probe-prologue followed by probe in SystemTap. There is no need to define them in SystemTap, they may be used after first assignment:

```
probe kernel::function("vfs_write") {  
    pos = $file->f_pos;  
}
```

In DTrace, their types may be optionally defined with `this` keyword, and later used with `this->` prefix:

```
this uint32_t local var;
```

```
::write:entry {  
    this->local var = (uint32_t) arg0;  
}
```

Warning

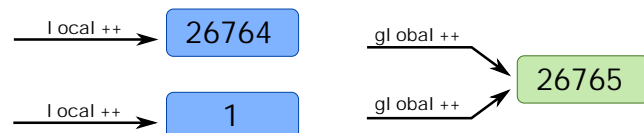
DTrace doesn't check scopes for local variables nor initialize it with zero, thus allowing racing conditions. Take the following script as an example, which counts the number of `read()` system calls:

```
int global;  
this int local;  
  
syscall::read:entry {  
    this->local++;  
    global++;  
}  
syscall::read:return {  
    printf("local: %d global: %d\n", this->local, global);  
}
```

If you run this script in parallel with single `dd` process, everything will look fine:

```
# dtrace -qs clauselocal.d -c "dd if=/dev/zero of=/dev/null"  
[...]  
local: 26765 global: 26765  
[...]
```

But when you run multiple `dd` processes, local and global numbers will eventually differ, because in case of race condition, new space will be allocated:



Thread-local variables

Thread local variables are created in a context of a thread, and after thread is switched, you will access a new instance of variable. Their syntax is similar to local DTrace variables, but use `self` keyword instead of `this`. They are extremely useful in passing data between distinct probes:

```
self int readfd; // Optional

syscall::read:entry {
    self->readfd = arg0;
}
syscall::read:return {
    printf("read %d --> %d\n", self->readfd, arg1);
}
```

Thread-local variables are not supported by SystemTap but may be easily simulated with associative array whose key is a thread ID:



```
global readfd;
probe syscall.read {
    readfd[tid()] = fd;
}
probe syscall.read.return {
    printf("read %d --> %d\n", readfd[tid()], $return);
}
```

In this case thread-local variable `readfd` is used to pass value from entry (call) probe to return probe. Same effect can be achieved with `@entry` expression in SystemTap, however it is limited to DWARF probes and its arguments, so prologue variable `fd` is not accessible with it:

```
probe syscall.read.return {
    printf("read %d --> %d\n", @entry($fd), $return);
}
```

Other use-case for thread-local variables is when you want to trace only processes that did certain actions, and filter others. In this case, you will introduce a kind of thread-local `do_trace` flag, which will be set to 1 if action was done (and probably, reset later), and later check this flag in predicate. If value is not set in associative array in SystemTap or as thread-local variable in DTrace, it defaults to 0, which by default disables probes. This approach is idiomatic, and for example used in [Dynamic code analysis](#) for building code graphs.

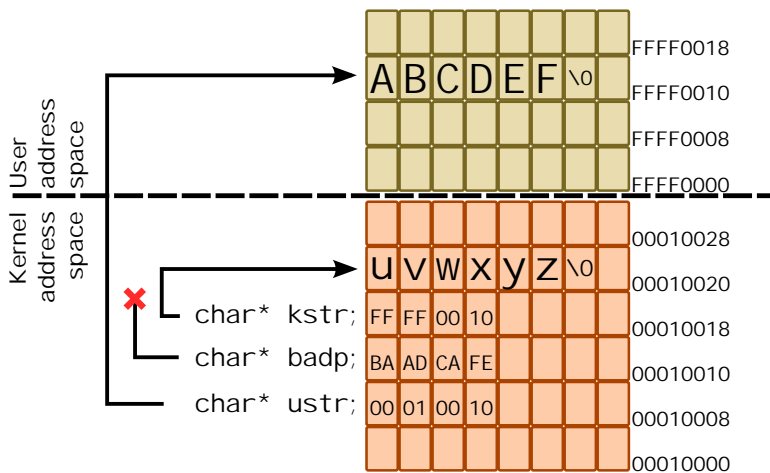
References

-  [Variables](#)
-  [Data types](#)
-  [Variables](#)

Pointers

Pointer is a special variable in C language that points (references) to a some data in memory, thus pointer usually contains address of that data. It is a common way to keep complex data structures in dynamically allocated memory, and pass a pointer between functions or share data among them by using same pointers at all consumers. SystemTap supports pointers in DWARF variables, but for locals it treats them as long. DTrace simulates full support of pointers, arrays and even dynamic allocation of them. To create a pointer you can use operator like you do in C.

Things in kernel get complicated because some pointers point to a *user address space* which is not trivially accessible, so instead of dereferencing it special function is called to copy data in or out. For example, when application issues `open()` system call, it keeps `pathname` argument as a string located in user address space, and passes only pointer to an argument. Moreover, some pointers may be invalid, and dereferencing them may cause system fault. So instead of working with raw pointers, dynamic tracing languages provide set of interfaces. In the following example, `badp` is bad pointer, which points nowhere, `kstr` points to a data in *kernel address space*, while `ustr` references string in *user address space*:



Accessing a data in kernel address space in DTrace is performed by simple dereferencing it in C-style. For example, `fop_open()` function accepts pointer to pointer to `vnode_t`, so to get actual address of `vnode_t`, you need to dereference it:

```
# dtrace -n '
    fbt::fop_open:entry {
        printf("0x%p", (uintptr_t) *args[0]); }'
```

User address space may be read in DTrace by using `copyin`, `copyinstr` or `copyinstr` subroutines, or be overwritten with `copyout`/`copyoutstr` (requires destructive actions to be allowed). For example, `poll` system call accepts array of `fds`, which are located in userspace and should be copied into address space of script before being printed:

```
# dtrace -n '
    this struct pollfd* fd0;

    syscall::pollsys:entry
    /arg1 != 0/
    {
        this->fd0 = copyin(arg0, sizeof(struct pollfd));
        printf("%s: poll %d\n", execname, this->fd0->fd); }'
```

SystemTap allows to access kernel and user memory through set of functions which are implemented in `tapsets conversions.stp` and `conversions-guru.stp`. They also allow to

specify different types such as `ulong` or `int16`, but they silently convert their result to `long` or `string`

- `kernel_` reads kernel memory. For example, `vfs_write` call changes file position, thus it gets position as pointer to a `struct file` member or a stack variable. To trace it, we have to dereference it:

```
# stap -e '
  probe kernel.function("vfs_write").return {
    off = kernel_ulong($pos);
    printf("write: off=%d\n", off);    }'
```

- `set_kernel_` writes kernel memory if Guru-mode is enabled
- `user_` reads userspace memory
- `kread()` used for safely reading kernel space in Embedded C

Summarizing all that, we should use following to read or write first character of strings in example above:

<i>Operation</i>	<i>Pointer</i>	<i>DTrace</i>	<i>SystemTap</i>
read	<code>kptr</code>	<code>*((char*) arg0)</code>	<code>kernel_char(\$kptr)</code>
	<code>badp</code>		<code>kernel_char(\$kptr)</code> with try-catch-block
	<code>uptr</code>	<code>*((char*) copyin(arg0, 1))</code>	<code>user_char(\$uptr)</code>
write	<code>kptr</code>	-	<code>set_kernel_char(\$kptr, 'M')</code>
	<code>badp</code>		<code>set_kernel_char(\$kptr, 'M')</code> with try-catch-block
	<code>uptr</code>	<code>this->c = (char*) alloca(1);</code> <code>*this->c = 'M';</code> <code>copyout(this->c, arg0, 1);</code>	-

Safety notes

To avoid system panicking, before actually accessing memory through raw pointer, DTrace and SystemTap have to:





- Check correctness of userspace pointer by comparing it with base address
- Check correctness of address by comparing it to a forbidden segments (such as OpenFirmware locations in SPARC).
- Add extra checks to page fault interrupt handlers (in case of DTrace) or temporarily disable pagefaults (SystemTap)

If you access to incorrect address, DTrace will warn you, but continue execution: `dtrace: error on enabled probe ID 1 (ID 1: dtrace::BEGIN): invalid address (0x4) in action #1 at DIF offset 16` SystemTap prints similar message and then fail: `ERROR: kernel string copy fault at 0x0000000000000001 near identifier 'kernel_string' at /usr/share/systemtap/tapset/conversions.stp: 18: 10`

Warning

Sometimes even correct addresses cause faults if data they point to is not in memory.

References

-  [Pointers and arrays](#)
-  [Actions and Subroutines](#)
-  [String and data retrieving functions Tapset](#)
-  [String and data writing functions Tapset](#)

Strings





Strings in dynamic tracing languages are wrappers around C-style null-terminated `char*` string, but they behave differently. In SystemTap it is simple alias, while DTrace add extra limitations, for example, you can't access single character to a string. String operations are listed in following table:

Operation	DTrace	SystemTap
Get kernel string	<code>stringof (expr) or (string) expr</code>	<code>kernel_string*()</code>
Convert a scalar type to a string		<code>sprint()</code> and <code>sprintf()</code>
Get userspace string	<code>copyinstr()</code>	<code>user_string*()</code>
Compare strings	<code>==, !=, >, >=, ,</code> •- semantically equivalent to <code>strcmp</code>	
Concatenate two strings	<code>strjoin(str1, str2)</code>	<code>str1 . str2</code>
Get string length	<code>strlen(str)</code>	
Check if substring is in string	<code>strstr(haystack, needle)</code>	<code>isinstr(haystack, needle)</code>

Note that this operations may be used in DTrace predicates, for example:

```
syscall::write:entry
/strstr(execname, "sh") != 0/
{}
```

References

-  [Strings](#)
-  [Actions and Subroutines](#)
-  [Strings](#)
-  [A collection of standard string functions](#)

Structures

Many subsystems in Linux and Solaris have to represent their data as C structures. For example, path to file corresponds from file-related structure `dentry` and filesystem-related structure `vfsmnt`:

```
struct path {
    struct vfsmount *mnt;
    struct dentry *dentry;
};
```

Structure fields are accessed same way it is done in C: in DTrace depending on what you are getting you need to use `->` for pointers and `.` for structures. In SystemTap you should always use `->` which will be contextually converted to `.` where needed. Information about structures is read from CTF sections in Solaris and DWARF sections in Linux, including



field names. To get C structure you may need to cast a generic pointer (void* in most cases) to a needed structures. In DTrace it is done using C-style syntax:

```
(struct vnode *)((vfs_t *)this->vfsp)->vfs_vnodecovered
```

Conversion in SystemTap is used more often, because in many places, typed pointers are coerced to generic long type. It is performed with @cast expression which accepts address, name of structure as string (struct keyword is optional), and an optional third parameter which contains name of include file, for example:

```
function get_netdev_name: string (addr: long) {  
    return kernel_string(@cast(addr, "net_device")->name)  
}
```

References

-  [DTrace Structs and Unions](#)
-  [SystemTap Expressions](#)

Exercise 1

Write opentrace.d and opentrace.stp scripts which are tracing open() system calls. They should print following information in one line:

- Call context: name of executable file, process ID, user and group IDs of user and group which are executing process.
- Path to file which should be opened.
- A string containing open() flags O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT
- Return value of system call

For example:

```
tee[939(0:0)] open("/tmp/test", O_WRONLY|O_APPEND|O_CREAT) = 3
```

Bit flags values are presented in following table:

Flag	Solaris	Linux (x86)
O_RDONLY	bits 0-1 are not set	
O_WRONLY	1	1
O_RDWR	2	2
O_APPEND	8	1024
O_CREAT	256	64

Test script that your created by experimenting with redirection to file or a pipe with tee tool:

```
# cat /etc/inittab > /tmp/test  
# cat /etc/inittab >> /tmp/test  
# cat /etc/inittab | tee /tmp/test  
# cat /etc/inittab | tee -a /tmp/test
```

Warning

In Solaris 11 open() system call was replaced with more generic openat().

Optional: Modify your scripts so only files that have "/etc" in their path will be shown.

Associative arrays

Definition

Associative array is a sequence of values which are accessible through one or more keys. Any types may be used for hashing, but they have to be comparable, and in some cases hashable.

Associative arrays are useful for saving last observable state related to a some object, so it can be reused in subsequent probes. For example, let's save last read or write operation performed on file. You will need to define keys and value types in DTrace:

```
string last_fop[int, int];
syscall::read:entry, syscall::write:entry {
    last_fop[pid, (int) arg0] = probefunc;
}
```

In SystemTap, however, they are deduced from the assignment:

```
global last_fop;
syscall.read, syscall.write {
    last_fop[pid(), $fd] = pn();
}
```

To delete entry from an associative array, it should be assigned to 0 in DTrace or deleted using `delete array[key1];` expression in SystemTap. If value does not exist, both DTrace and SystemTap will return 0 as a default value.

In DTrace you only can access value in associative array knowing its key, in SystemTap along with that you can walk entire array with `foreach` statement:

```
foreach([pid+, fd] in last_fop limit 100) {
    printf("%d\t%d\t%s\n", pid, fd, last_fop[pid, fd]);
}
```

Variables for keys are listed in square braces. If variable name ends with `+` or `-`, than keys will be sorted in ascend or descend order correspondingly (only one key may be used for sorting). Optional `limit N` part allows to limit amount of entries.


Maximum amount of entries that associative array can keep is limited by `dynvarsize` tunable in DTrace or `MAXMAPENTRIES` in SystemTap. Additionally, you may explicitly specify maximum number of entries in array:

```
global array[SIZE];
```

Warning

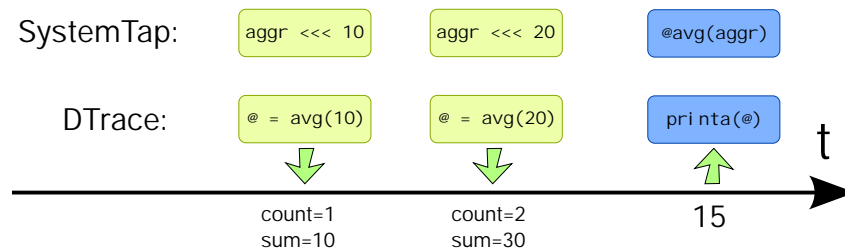
Starting with SystemTap 2.1 it allocates `MAXMAPENTRIES` entries for associative array on per-cpu basis (using not only online, but possible CPUs too) at start (to avoid further allocation faults). Also, it allocates memory for strings statically too. So to keep associative array with string key you will need at least `NR_CPUS * MAXMAPENTRIES * MAP_STRING_LENGTH` which gives 128 megabytes of memory on CentOS 7.0 x86_64.

References

-  [Variables/Associative arrays](#)
-  [Associative arrays](#)

Aggregations

Aggregations are most useful for evaluating system performance (they are called *statistics* in SystemTap). Aggregation will update intermediate set of parameters when new value is added. Overall value is calculated from that intermediate set when its printing is requested. Let's for example see how it works for mean value — dynamic tracing system saves count of added values and their sum, and when values need to be printed, sum is divided to a count:



Aggregations in DTrace reside in separate namespace: each name of aggregation begins with at-symbol @. Single at-symbol @ is an alias to @_ and is a shorter possible aggregation name which is useful for one-liners. Moreover, if it was not printed in the END probe, or timer probe, DTrace will automatically print it for you. There is no need to declare aggregation, and it support key access same way associative array does. When value is added to a aggregation, it is "assigned" to a return value of aggregating function, i.e. `@fds = avg(arg0);` will create an aggregation which calculates mean value of `arg0`.

SystemTap have a statistics. They are do not support indexing like associative arrays (but they may be a values in associative arrays), thus they are special kind a variable. To create a statistic you need to use *aggregate operator* instead of assignment operator =, for example: `fds .` Aggregating function is used when result is printed, and begins with @, i.e. `@avg(fds)` will return mean value of statistic `fds`. This allows to use single statistic for multiple functions wherever possible.

Here are list of aggregating functions (note that in SystemTap they have to be preceded with @):

- `count` — counts number of values added
- `sum` — sums added value
- `min/max/avg` — minimum, maximum and mean value, respectively
- `stddev` — standard deviation (only in DTrace)
- `linearize` — prints linear histogram (`hist_linear` in SystemTap)
- `quantize` — prints logarithmic histogram (`hist_log` in SystemTap)

The following actions may be performed on aggregations:

Action	DTrace	SystemTap
Add a value	<code>@aggr[keys] = func(value);</code>	<code>aggr[keys]</code>
Print	<code>printa(@aggr)</code> or <code>printa("format string", @aggr1, @aggr2, ...)</code>	<code>println(@func(aggr))</code> (use <code>foreach</code> in case of associative arrays).
Flush values and keys	<code>clear(@aggr)</code> (only values) or <code>trunc(@aggr)</code> (both keys and values)	<code>delete aggr</code> or <code>delete aggr[keys]</code>
Normalize	<code>normalize(@aggr, value);</code> and <code>denormalize(@aggr);</code>	Use division / and multiplication * on results of aggregating functions
Limit number of values	<code>trunc(@aggr, num)</code>	Use <code>limit</code> clause in <code>foreach</code>

Warning

Aggregations may be sorted in DTrace using `aggsortkey`, `aggsortpos`, `aggsortkeypos` and `aggsortrev` tunables.

Aggregations are extremely useful for writing stat-like utilities. For example, let's write utilities that count number of write system calls and amount of kilobytes they written.

Source file: [scripts/dtrace/wstat.d](#)

Note that aggregations are follow after keys in `printa` format string, and they are going in the same order they are passed as `printa` parameters. Format fields for aggregations use `@` character. Sorting will be performed according to a PID (due to `aggsortkey` tunable), not by number of operations or amount of bytes written. Option `aggsortkeypos` is redundant here, because 0 is default value if `aggsortkey` is set.



SystemTap has similar code, but `printa` is implemented via our own `foreach` cycle. On the other hand, we will keep only one associative array here:

Source file: [scripts/stap/wstat.stp](#)

Output will be similar for DTrace and SystemTap and will look like:

PID	EXECNAME	FD	OPS	KBYTES
15881	sshd	3	1	0
16170	stapi o	1	1	0
16176	python	8	8052	32208
16176	python	7	8045	32180
16176	python	10	8007	32028
16176	python	9	8055	32220

References

-  [Aggregations](#)
-  [Statistics \(aggregates\)](#)

Time

A man used to live with a calendar and 24-hour representation of time. Coordinated Universal Time (UTC) is used for that now. These details are not needed for most kernel or application processes, so there is multiple time sources available for tracing tools:

Time source	DTrace	SystemTap
<i>System timer</i> is responsible for handling periodical events in kernel such as context switch. System timer usually ticks at constant frequency (but ticks may be omitted in <i>tickless kernels</i>). Interval between firing timer is usually referred as special unit of time: <i>tick</i> , <i>lbolt</i> in Solaris or <i>jiffy</i> in Linux. Timer frequency in Linux can be get using <code>HZ()</code> function.	<code>^lbolt</code> or <code>^lbolt64</code>	<code>jiffies()</code>
<i>Processor cycles counter</i> is a special CPU register which act as a counter which increases on each cycle, such as <code>TSC</code> in x86 or <code>%tick</code> in SPARC. It may not be monotonic.		<code>get_cycles()</code>
<i>Monotonic time</i> . Starts at unspecified moment of time (usually at system boot), but ticks with constant intervals. May use high-resolution time source such as HPET on x86, but may impose some jitter between CPU cores or CPUs.	<code>timestamp</code>	<code>local_clock_()</code> or <code>cpu_clock_()</code>

Time source	DTrace	SystemTap
<i>Virtual monotonic time of thread.</i> Similar to previous time source, but only accounts when thread is on CPU, which is useful to calculate CPU usage of a thread	<code>vtimestamp</code>	
<i>Real time or Wall-clock time.</i> Monotonic time source which starting point is an UNIX Epoch (00:00:00 UTC, Thursday, 1 January 1970). May use extra locks, access RTC, so it generally slower than previous time sources	<code>walltimestamp</code>	<code>gettimeofday_()</code>

In this examples is one of (s -- seconds, ms -- milliseconds, us -- microseconds and ns -- nanoseconds). DTrace time sources always have nanosecond resolution.

Generally speaking, monotonic time sources are better for measurement relative time intervals, while real time is used if you need precise timestamp of an event (i.e. for cross-referencing with logs). To print a real timestamp, use `ctime()` function in SystemTap which converts time to string, or use `%Y` format specifier in DTrace print functions.

References

-  [Built-in Variables](#)
-  [Timestamp Functions](#)

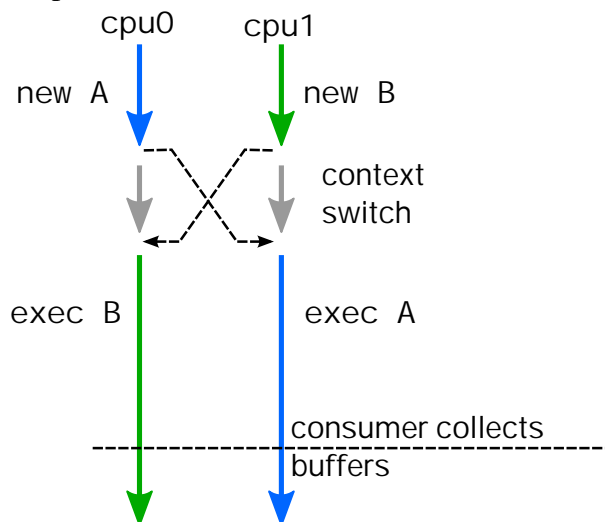
Printing

As we mentioned earlier, DTrace and SystemTap are printing to a special channel established between probe handlers and consumer process. SystemTap maintains multiple channels, and some of them support prioritized printing through `log()` and `warn()` functions.

A simplest printing doesn't allow any formatting. It is performed via `trace()` action in DTrace which accepts only single argument. In SystemTap it is performed through `print[d][ln]` functions which accept unspecified number of arguments, and may use first argument as delimiter (should be constant) if `d` suffix is present, and add a newline if `ln` suffix is present. I.e. `printdln(", ", $fd, $pathname)` will print comma-separated line with two parameters. Formatting output is supported through using `printf()` function which accepts format string and unspecified number of arguments. Rules for creating format strings are similiar to C standard `printf()` function including support for various such as `%p` stands for pointer, setting width and alignment of field, etc. Dynamic tracing languages are strict about types of arguments and format strings.

DTrace allows to print a memory dump using `tracemem()` action which accepts address and number of bytes to be printed (should be constant). There is no such function in SystemTap, but it can be simulated using for-cycle, `kernel_int()` functions and `printf()` or use `%m` format specifier with width modifier which is mandatory in this case (and sets length of memory area to be printed). Also, in some cases pretty conversion to a strings is allowed, i.e. `inet_ntop()` in DTrace and `ip_ntop()` in SystemTap allow to convert IP-address to a string.

To reduce competition for output buffers in multicore systems, SystemTap and DTrace allocate buffers on per-cpu basis. Then they need to extract data from them, they switch buffers and walk over it. Consider following example: process A starts on CPU 0 while process B starts on CPU 1, than context switch occurs and both processes migrating on opposite CPU (this is unrealistic situation for scheduler, so it is only an example) as shown on picture:






In this example you will get the following output:

```
new A
exec B
new B
exec A
```

This makes interpretation of output is extremely complicated especially in case of dozens events (such as tracing ZIO pipeline in ZFS filesystem). This problem can be solved only by adding extra key related to a request (such as process ID, like A and B in this example) to a each line and group events in [post-processing](#).

References

-  [Actions and Subroutines](#)
-  [Output Formatting](#)
-  [Formatted output](#)

Speculations



Predicates is one form to get rid to useless event, but they only allow to decide when probe is firing. What if there are several probes and decision can be made only in the last one? To answer that problem, dynamic tracing languages support *speculations*. For example you may want to trace only requests which are finished with an error code.

Speculations allow to create independent output buffer for each request using `speculation()` function which returns id of that buffer. You may put it to an associative array using some vital request information as a key, for example pointer to a structure. While tracing you may either print data from the buffer to a main buffer using `commit()` function or reject it using `discard()` function. Maximum number of speculations in DTrace is regulated by `nspec` tunable.

To add an output to a speculation in DTrace, call `speculate()` function which accepts single argument — speculation id. After that call, all subsequent print statements in current probe body will be redirected to a speculation buffer. In SystemTap `speculate()` accepts two parameters: one for speculation id and second for string to be put into speculation, so you should use `spri ntf()` instead of `pri ntf()` to print to a buffer.

Speculations are used in [Block Input-Output](#) scripts.

References

-  [Speculative Tracing](#)
-  [Speculation](#)

Tapsets translators

We already discussed problem with probe stability. Some issues may be related to changing data structures in kernel, or several variants may exist in kernel, for example for 32- and 64-bit calls. Let's see how access to fields of that structure may be unified.

DTrace has a *translators* for doing that:

Source file: [scripts/dtrace/stat.d](#)

In this example translator describes rules of converting source structure `stat64_32` to a structure with known format defined in DTrace `stat_info`. After that, `xlate` operator is called which receives pointer to `stat64_32` structure to a `stat_info`. Note that our translator also responsible for copying data from userspace to kernel. Built-in DTrace translators are located in `/usr/lib/dtrace`.

SystemTap doesn't have translators, but you can create *prologue* or *epilogue alias* which performs necessary conversions before (or after, respectively) probe is called. These aliases are grouped into script libraries called *tapsets* and put into `/usr/share/systemtap/tapset` directory. Many probes that we will use in following modules are implemented in such tapsets.

Linux has several variants for `stat` structure in `stat()` system call, some of them deprecated, some are intended to support 64-bit sizes for 32-bit callers. By using following tapset we will remove such differences and make them universally available through `fil ename` and `si ze` variables:

Source file: [scripts/stap/tapset/lstat.stp](#)

This example is unrealistic: it is easier to attach to `vfs_l stat` function which has universal representation of `stat` structure and doesn't involve copying from userspace. Summarizing the syntax of creating aliases:

```
probe alias-name {=|+=} probe-name-1 [?] [, probe-name-2 [?] ...] probe-body
```

Here `=` is used for creating prologue aliases and `+=` is for epilogue aliases. Question mark `?` suffix is optional and used if some functions are not present in kernel — it allows to choose probe from multiple possibilities.

Warning

Note that this tapset only checks for 64-bit Intel architecture. You will need additional checks for PowerPC, AArch64 and S/390 architectures.

After we created this tapset, it can be used very easy:

Source file: [scripts/stap/lstat.stp](#)

Also, sometimes we have to define constants in dynamic tracing scripts that match corresponding kernel or application constants. You can use enumerations for that in DTrace, or define a constant variable with `inline` keyword:

```
inline int TS_RUN = 2;
```

You may use initializer for global variable to do that in SystemTap:

```
global TASK_RUNNING = 0;
```

If you have enabled preprocessor with `-C` option, you may use `#define` to create macro as well.

References

-  [Translators](#)
-  [Probe aliases](#)

Exercise 2

Modify scripts from Exercise 1 so they count following statistics for processes that are running in a system:

- number of attempts to open existing file;
- number of attempts to create a file;
- number of successful attempts.

At a period that is defined as command line arguments (specified in seconds) script should print:

- Current time and day in human-readable format.
- Table that contains gathered statistics per process along with that process name and PID.

Numbers should be cleared during each iteration.

You can use module `file_opener` to demonstrate your scripts. This module uses working directory which is passed as `root_dir` parameter, fills it with some files that are created preliminary (their number is set by `created_files` parameter). While executing request, it uses `file_random` variable (which range is cut to `[1; max_files)`) and either tries to create a file or open it depending on `create` parameter.

Run several experiments using `TSLoad` workload generator varying `created_files` parameter and compare the results:

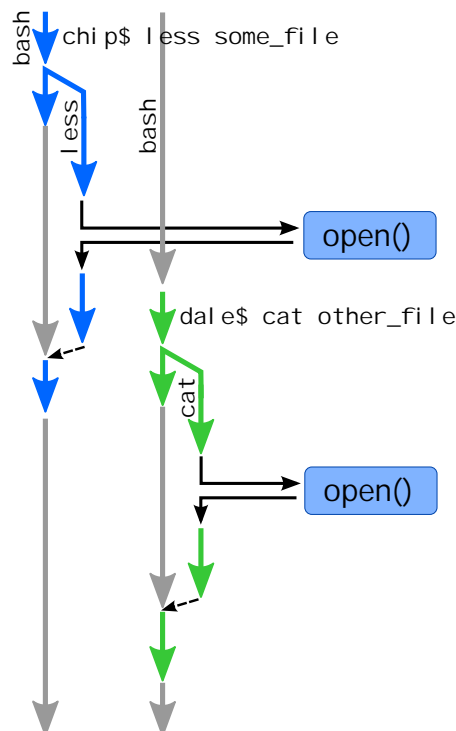
```
# EXPDIR=/opt/tsload/var/tsload/file_opener
# for i in 1 2 3; do
    mkdir /tmp/fopen$i
    tsexperiment -e $EXPDIR run \
        -s workloads: open: params: root_dir=/tmp/fopen$i \
        -s workloads: open: params: created_files=$((i * 160))
done
```

Try to explain differences you get from the nature of `file_opener` workload generator module.

Module 3: Principles of dynamic tracing

Applying tracing

As we mentioned in [Tracing](#), it is used for statistics collection and performance analysis, dynamic kernel or application debug, system audit. Imagine the situation in which various processes running by two different users are opening files:



What problems can occur and how they are solved by dynamic tracing? Users can complain to very slow opening of a file, so we need to do *performance analysis*. First of all, we have to confirm user complaints by measuring time spent in `open()`, `read()` and `write()` system calls. We can also try to cross-reference slow calls and filesystems on which they occur (by gathering mount paths), if problems are caused by bad NAS or disk. If the problem still exists, then you will need to go down VFS stack, i.e. by measuring time spent in block I/O or

in lookup operations.

If user encounters errors while opening files, then you will need to trace `errno` values. These values are usually returned by system call functions in Linux, or saved into `errno` variable in DTrace. To determine why system call returns an error, you will need *dynamically debug* it by checking return values of callees. We will demonstrate it in following section. If users try to attempt files they do not have permissions, we can record `errno` along with paths and user ids, so by doing that we will perform *system audit*.

To demonstrate it on real example, we will use following examples and run `cat /etc/shadow` from some non-root user:

```
# dtrace -qn '  
    syscall::open*:entry {  
        printf("=> uid: %d pid: %d open: %s %lld\n",  
            uid, pid, copyinstr(arg1), (long long) timestamp);  
    }  
    syscall::open*:return {  
        printf("
```

SystemTap version:

```
# stap -e '  
    probe syscall.open {  
        printf("=> uid: %d pid: %d open: %s %d\n",  
            uid(), pid(), filename, local_clock_ns());  
    }  
    probe syscall.open.return {  
        printf("
```

Here is sample output:

```
=> uid: 60004 pid: 1456 open: /etc/shadow 16208212467213
```

First of all, we measured time spent for `open()` system call: 16208212482430 , 16208212467213 = 15217 = 15.2 us. We can also see that user received an error (return code is -1, while in case of correct call it would be positive) and now we may try to seek for a source of a problem. Finally, we have audited attempt to open critical system file `/etc/shadow` which is forbidden for users. So now we should find user name with id 60004 and politely ask him why he tried to open `/etc/shadow` file.

We will discuss how trace data may be analysed and what conclusion can be made from it in this module. However, we will not introduce useful kernel or application probes as we will discuss them in modules 4, 5. On the other hand, all examples in following modules will be pure tracers, so you will need to add additional processing of results which will be discussed in this module.

Dynamic code analysis

Definition

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor.

We will refer to program analysis as code analysis since program is a product of code compilation. On contrary, static code analysis is performed without actually running the program. Code analysis helps to match program behaviour such as opening files, sending messages over network to their code.

Backtraces (stacks)

Simplest way to perform code analysis is to print a backtrace.

Information

Code uses registers while function is executed most of the time. However, when function is called it will use same registers (unless CPU supports register windows like SPARC processor do, however even they are limited), so it has to save registers somewhere in memory including program counter, which will be used when we return from function. Usually *stack* is used as memory for function locals and saved registers. It has special rule of allocation: stack always grow to a increasing or decreasing address, each call allocate a memory beyond current stack pointer (which is also a special register), and increases/decreases it, while each return resets value of stack pointer to a previous, thus deallocating *stack frame*.

If we extract program counter and register values from a stack, we may be able to recover history of calling functions and their arguments. For example, once I encountered panic in Solaris kernel. Printing stack (or, more correctly *backtrace*) from crash dump uncovered this:

```
fzap_cursor_retrieve+0xc4(6001ceb7d00, 2a100d350d8, 2a100d34fc0, 0, 0, €
2a100d34dc8)
[...]
zfsvfs_setup+0x80(6001ceb2000, 1, 400, 0, 6001a8c5400, 0)
zfs_domount+0x20c(60012b4a240, 600187a64c0, 8, 0, 18e0400, 20000)
zfs_mount+0x20c(60012b4a240, 6001ce86e80, 2a100d359d8, 600104231f8, 100, 0)
domount+0x9d0(2a100d358b0, 2a100d359d8, 6001ce86e80, 60012b4a240, 1, 0)
mount+0x108(600107da8f0, 2a100d35ad8, 0, 0, ff3474f4, 100)
[...]
```

Name prefix of top-level function implies that problem is in ZAP subsystem, and bottom function says that problem occur while mounting file system. Second argument to `zfs_domount` function is the name of mounting dataset. By reading string from it we were able to determine its name, make it readonly and boot the system.

In DTrace stack functions may be used as a keys to an associative arrays, or as separate function calls (in that case they will just print the stack). Stack of kernel functions is available by using `stack()` subroutine, while userspace application stack is available using `ustack()` subroutine. Both of them have optional constant integer argument which specifies how many stack frames should be printed. For example:

```
# dtrace -c 'cat /etc/passwd' -n '
    syscall::read:entry
    /pid == $target/
    { stack(); ustack(); }'
```

There are multiple SystemTap functions that are responsible for printing stack:

- `backtrace()` and `ubacktrace()` returns a string containing a list of addresses in hexadecimal format;
- `print_stack()` and `print_ustack()` get stack from string returned by `backtrace` functions, convert addresses to symbols wherever possible and print it;
- `print_backtrace()` and `print_ubacktrace()` gets stack and prints it immediately, thus no arguments accepted and no return values supplied;
- `task_backtrace()` accepts pointer to process/thread `task_struct` as a parameter and returns its kernel stack wherever possible. Functions which have `u` in their names print userspace backtrace, functions which do not have it, print kernel backtrace. For example:

```
# stap -c 'cat /etc/passwd' -e '
    probe kernel.function("sys_read") {
        if(pid() == target())
            print_stack(backtrace());
    } '
# stap -c 'cat /etc/passwd' -e '
    probe process("cat").function("read")
```

```
{ print_ubacktrace(); } '
```

Printing backtraces involves getting a symbol which matches some memory address which involves digging into symtab or similar section of binary files. Dynamic tracing systems can do that and print (DTrace) or return a symbol as a string (SystemTap) with following functions:

Userspace	DTrace	SystemTap
Symbol	usym(addr) or ufunc(addr)	usymname(addr)
Symbol + offset	uaddr(addr)	usymdata(addr)
Library	umod(addr)	umodname(addr)
Kernel	DTrace	SystemTap
Symbol	sym(addr) or func(addr)	symname(addr)
Symbol + offset		symdata(addr)
Library	mod(addr)	modname(addr)
Module, symbol + offset	printf("%a", addr)	

For example, some kernel interfaces like VFS are polymorphic, so they have a function pointer table. You may extract these pointers and resolve them to a function name:

```
# stap -c 'cat /etc/passwd' --all-modules -e '
    probe kernel.function("do_filp_open").return {
        if(_IS_ERR($return)) next;
        addr = $return->f_op->open;
        printf("name: %s, addr: %s, mod: %s\n",
            symname(addr), symdata(addr), modname(addr)); }'
```

Similar example for DTrace:

```
# dtrace -c 'cat /etc/passwd' -n '
    fop_open:entry {
        this->vop_open =
            (uintptr_t)(*args[0])->v_op->vop_open;
        sym(this->vop_open); mod(this->vop_open); }'
```

In this example when cat will try to open file, tracing script catch this event and show name of filesystem driver and function from it implementing open() call (unless it is generic function from kernel).

Warning

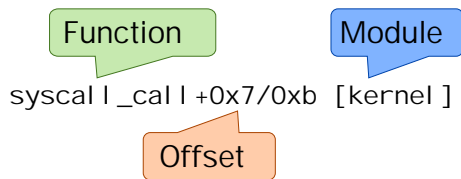
By default SystemTap seeks only in vmlinux or binary executable files. To search over libraries and modules, use -d, --all-modules and --ltd options as stated in [SystemTap](#).

Warning

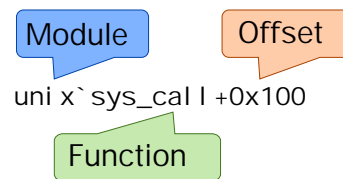
In DTrace symbols are resolved by consumer, not by DIF interpreter — that is why they do not return strings, and not usable in string functions. Moreover, if when buffers are switched module was unloaded or process is finished, DTrace will fail to resolve symbols and print raw address in stack or symbol functions.

SystemTap and DTrace have different formats when printing backtrace symbol names:

SystemTap:



DTrace:



Call trees

Note that backtraces show only a stack, a linear structure of functions that lead to event, they do not include callees that were previously called, but already exited. For example in following code (which obviously causes a segmentation fault):

```
char* foo() {  
    return NULL; }  
void bar() {  
    char* str = foo();  
    puts(str); }
```

we will see functions `bar()` and `puts()` on stack, but the problem is caused by `foo()` function. To trace it along with other functions, we will need a *call tree*, which is close to *call graph* which is collected during static calling analysis.

Global or thread-local flag (say `traceme`) is used to gather call tree. It is set when we enter some function (which is considered a bound for tracing), and reset when we leave it. In following example, we, for example, may limit tracing by using `bar()` as our bound. Without using such boundary functions, tracing will have too much performance penalties. Probes are attached to **all** functions, but predicate is used to check if `traceme` flag, so only useful probes will be printed. Such probe only print names of functions preceded with indent whether indent is set according to a depth of call, so output look like a tree.

DANGER!

To attach to all kernel functions in SystemTap you have to use `kernel.function("**")` construct. But even with blacklisted probes, it will most likely panic system, or cause a serious system slowdown. To keep that from happen, limit number of attached probes to a module or at least subsystem by using `@path/to/files.c` construct like we do in following examples. In DTrace, however `fbt:::` is pretty safe and may only cause a temporary small freeze (while probes are attached).

For example, let's see how this approach helps find a source of fault in system call. We will try to execute `cat not_exists` file which will set `errno` to `ENOENT` as expected. Let's find a kernel function that actually reports `ENOENT`. Usually, negative integer return value used for that, so we will print return values in function return values. This approach is also useful when you have no permission to open file and want to find a security hardening module that stops you from doing that.

In SystemTap call tree indentation is performed through `indent()` and `thread_indent()` which are maintaining internal buffer, and increase or decrease number of space characters in it according to a number passed as argument to that functions. It is used in following script:

Source file: [scripts/stap/callgraph.stp](#)

Output will be following:

```
# ./callgraph.stp -c "cat not_exists"  
cat: not_exists: No such file or directory
```

```
=> syscall.open [cat]
0 : -> do_sys_open
[...]
11020 : -> do_fi l p_open
[...]
11982 : -> do_path_l ookup
12277 : -> path_i ni t
12378 : path_wal k
12581 : -> _l i nk_path_wal k
[...]
14284 :
14339 : -> path_put
[...]
14655 : 14732 :
14755 :
[...]
15449 : 15851 :
```

So, now we can say that problem is in `__link_path_walk` function. This is output from CentOS 6, in modern kernels `__link_path_walk` is deleted and responsible function would be `path_openat`.

`indent()` function also prints time delta in microseconds since first call. `thread_indent()` also prints information about execution thread and maintains separate buffer for each thread.

DTrace consumer supports automatic indentation of output if `flowindent` tunable is set:

Source file: [scripts/dtrace/callgraph.d](#)

Script output for ZFS filesystem will be similar and reveal that ENOENT error was raised by ZFS module:

```
# dtrace -s ./callgraph.d -c "cat not_exists"
dtrace: script './callgraph.d' matched 69098 probes
cat: not_exists: No such file or directory
CPU FUNCTION
  0  -> open64
  0  openat64
  0  copy
[... ]
  0  -> vn_openat
  0  -> lookupnameat
  0  -> lookupnameatcred
[... ]
  0  -> fop_lookup
  0  -> crgetmapped
  0  zfs_lookup
[... ]
  0
  0
  0  -> vn_rele
  0
  0
  0
  0
  0
[... ]
  0  -> set_errno
  0
```

More backtraces

You may also need to track state of kernel data structures or passing parameters during tracing, if you wish to extend your knowledge about kernel or application. Now we know from our traces that Linux uses `__link_path_walk()` and Solaris has `lookupnpvp()` functions to lookup file on filesystems. Let's see, how they handle symbolic links. Let's create one first:

```
# touch file
# ln -s file symlink
```

As you can see, Linux calls `__link_path_walk` recursively:

```
# stap -e '
    probe kernel.function(%( kernel_v >= "2.6.32"
                          %? "link_path_walk"
                          %: "__link_path_walk" %) ) {
        println(kernel_string($name));
        print_backtrace(); }' -c 'cat symlink'
symlink
0xfffffffff811ad470 : __link_path_walk+0x0/0x840 [kernel]
0xfffffffff811ae39a : path_walk+0x6a/0xe0 [kernel]
0xfffffffff811ae56b : do_path_lookup+0x5b/0xa0 [kernel]
[...]
file
0xfffffffff811ad470 : __link_path_walk+0x0/0x840 [kernel]
0xfffffffff811ade31 : do_follow_link+0x181/0x450 [kernel]
0xfffffffff811adc1b : __link_path_walk+0x7ab/0x840 [kernel]
0xfffffffff811ae39a : path_walk+0x6a/0xe0 [kernel]
[...]
```

Since this function was removed in recent kernels, this behaviour is not reproducible in them.

In Solaris, however, this function is called only once — for symbolical link:

```
# dtrace -n '
    lookupnpvp:entry {
        trace(stringof(args[0]->pn_path));
        stack(); }' -c 'cat symlink'
1 19799          lookupnpvp:entry symlink
                genunix`lookupnpathcred+0x119
                genunix`lookupnameatcred+0x97
                genunix`lookupnameat+0x6b
[...]
```

Profiling

Consider the following task: you need to know which functions are called more often than others or spend most time when executing because it makes them perfect targets for code optimization. You may do it by attaching to every function entry and exit point the following script:

```
fbt:::entry {
    self->start = timestamp;
}
```



```

fbt::: return
/self->start/
{
    @fc[probefunc] = count();
    @ft[probefunc] = avg(timestamp - self->start);
}
tick-1s {
    printa("%s %d %d", @fc, @ft);
    trunc(@fc); trunc(@ft) }

```

DANGER!

This script is conceptual! Do not run it on real system.

If you were able to collect data with this script, you'll got *population*, but you couldn't do that. Usually function call takes several processor cycles and a single instruction, but when you run it, you'll need hundreds of instructions (for getting timestamp and writing to a aggregation), which causes colossal *overhead*. Statistics theory, however, provides a solution to that: instead of gathering entire population, you may reduce it to a *sample*, which is representative (reproduces significant properties of a population). Collecting a sample is called *sampling*, while sampling function calls is usually referred as *profiling*.

Definition

In software engineering, *profiling* is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.

Modern operating systems provide builtin profilers, such as **OProfile** and **SysProf** in Linux which were replaced with **perf** subsystem since 2.6.31 kernel or **er_kernel** from Solaris Studio. However, Dynamic tracing languages allow to build custom profilers.

A simplest profiler records process ID to see which processes or threads consume CPU resources more than others, as we discussed about [timer probes](#). They may be implemented with following DTrace script:

```

# dtrace -qn '
    profile-997hz {
        @load[pid, execname] = count();
    }
    tick-20s { exit(0); }'

```

Or in SystemTap:

```

# stap -e 'global load;
    probe timer.profile {
        load[pid(), execname()]
    }

```

If we want to go down to a function level, we need to access *program counter* register (or *instruction pointer* in x86 terminology) each time profiling probe fires. We will refer to program counter as PC later in this book. In DTrace these values are explicitly provided in `arg0` — PC in kernel mode and `arg1` — PC in userspace mode in `profiling` probes. Depending on if process was in kernel mode when profiling probe fired or not, `arg0` or `arg1` will be set to 0. Moreover, you may always get current userspace program counter using `uregs` array: `uregs[REG_PC]`. There is also `caller` and `ucaller` built-in variables.

You can use `addr()` tapset function in SystemTap which returns userspace PC or kernel PC depending on where probe were fired (some probes do not allow that, so 0 will be returned). To get userspace address explicitly, use `uaddr()` function.

Warning

Note that we were used `profile-997hz` probe to avoid "phasing": if we'd used `profile-1000hz` probe, there were a chance, that all probes were fired while system timer handler is working, thus making profiling useless (we will see that 100% of time kernel spends in system timer). In `SystemTap timer.profile` uses system timer for profiling, but `addr()` and `uaddr()` return correct values.

CPU performance measurement

Even if you collect program counter values, you will get what functions use CPU the most, but that doesn't mean that utilize processor resources effectively. For example, it can spend most of the time waiting for memory or cache or reset pipeline due to branch misprediction instead of utilizing ALU for actual computations. Such wasted cycles are referred as *stalled* in Intel processor documentation.

Modern processors allow to measure influence of such performance penalties through CPU performance counters. Each time such event happens, CPU increments value of the counter. When counter exceeds threshold, exception is arisen which may be handled by dynamic tracing system. Or, counter may be read from userspace application, for example with `rdpmc` assembly instruction on Intel CPUs.

You may use `cpustat` tool to get list of available CPU events in Solaris:

```
# cpustat -h
[... ]
event0:  cpu_clk_unhalted.thread_pinst_retired.any_p
```

Description of such events may be found in CPU's documentation. SPARC counters are described in the book "Solaris Application Programming", but it lacks description of newer CPUs (SPARC T3 and later). However, documentation on SPARC T4 and T5 may be found here: [Systems Documentation](#). Solaris also provides CPU-independent generic counters which names start with `PAPI` prefix.

Linux have separate subsystem that is responsible for providing access to CPU performance counters: `perf`. It has userspace utility `perf`, which can show you list of supported events:

```
# perf list
List of pre-defined events (to be used in -e):
    cpu-cycles OR cycles                [Hardware event]
    instructions                        [Hardware event]
```

You can use userspace tools `perf` in Linux or `cpustat/cputrack` in Solaris to gather CPU counters.

DTrace provides CPU counters through `cpc` provider (which is implemented through separate kernel module). Its probe names consists from multiple parameters:

`EventName`-{kernel|user|all}[-Mask]-Number

EventName is a name of event taken from `cpustat` output (and matches documentation name in case of Intel CPUs). Following parameter defines a mode: *kernel* probes only account kernel instructions, *user* only work for userspace, and *all* will profile both. *Number* is a threshold for a counter after which probe will fire. Do not set *Number* to a small values to avoid overheads and system lockup, 10000 provides relatively accurate readings. *Mask* is an optional parameter which allows to filter devices which accounted in performance counters (such as memory controllers or cores) and should be a hexadecimal number.

For example, you may use probe `PAPI_L3_tcm-user-10000` to measure number of userspace misses to last-level cache which is L3 cache in our case:

```
# dtrace -n '
  cpc::PAPI_13_tcm-user-10000
  /arg1 != 0/ {
    @[usym(arg1)] = count(); }
  END {
    trunc(@, 20);
    printa(@);
  }'
```

SystemTap provides access to CPU counter using perf tapset:

```
# stap -l 'perf.*.*'
perf.hw.branch_instructions
[...]
# stap -l 'perf.*.*.*.*'
perf.hw_cache.bpu.read.access
```

These probes are actually aliases for the following probes:

```
perf.type(type).config(config)[.sample(sample)][.process("process-name")][.counter("counter-name")]
```

type and *config* are numbers used in `perf_event_attr` — their values may be found in header `linux/perf_event.h`. *sample* is a number of events after which probe firing. *process-name* allows to monitor only certain processes instead of system-wide sampling and contains name of the process (path to executable). *counter-name* allows to set an alias for performance counter which will be later used for `@perf` expression (see below).

To measure last userspace level cache misses in SystemTap, you may use following script:

```
# stap -v -e '
  global misses;
  probe perf.hw_cache.ll.read.miss {
    if(!user_mode()) next;
    misses[probefunc()]
```

Warning

These examples were tested on Intel Xeon E5-2420 processor. Like we mentioned before, performance counters are CPU-specific.

SystemTap allows to create per-processor counter which can be read later:

```
# stap -v -t -e '
  probe perf.hw.instructions
    .process("/bin/bash").counter("insns") { }

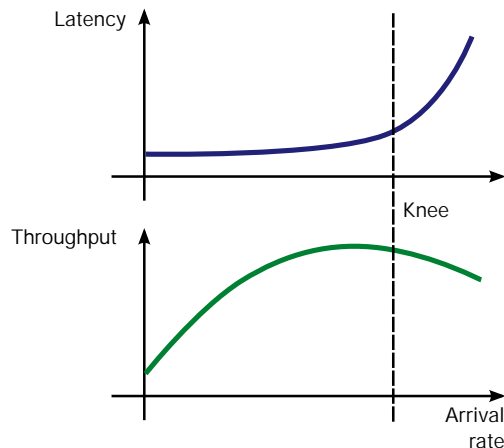
  probe process("/bin/bash").function("cd_builtin") {
    printf("insns = %d\n", @perf("insns"));
  }'
```

Warning

There is a bug [PR-17660](#) which can cause `BUG()` in kernel when you use `@perf` in userspace. It seem to be resolved in current SystemTap/Kernel.

Performance analysis

Computer users, system administrators and developers are interested in improving of performance of computer systems and dynamic tracing languages are very handfull in analysing soft spots of computer systems. We will use two characteristics of computer system to evaluate its performance most of the time: *throughput* and time spent for servicing request (usually referred as *latency*). These two characteristics depend on each other as following picture shows:



Information

For example, let's imagine a newspaper kiosk. Than number of customers per hour will be its *arrival rate*. Sometimes, when clerk is busy while servicing customer, other customers will form a queue, which can is measurable to *queue length*. Growing queues is a sign of the system's *saturation*. *Throughput* of the kiosk is the number of customers which bought a newspaper per hour. However, if number of customers is too large, kiosk couldn't service them all, and some of them will leave after waiting in line — they are treated as *errors*. When kiosk reaches its saturation point or the *knee*, *throughput* of the kiosk will fall, and number of errors will increase, because clerk will be tired.

Latency consists of *service time* which depends on many factors: i.e. if customer need change or clerk can't find copy of newspaper it will grow, and *waiting time* — time spent by a customer waiting in queue. *Utilization* is defined by a fraction time that clerk spends servicing their customers. I.e. if clerk spends 15 minutes to sell a magazines or newspapers per hour, utilization is 25%.

These definitions are part of *queueing theory* which was applied to telephone exchange, but it is also applicable to computer systems. Either network packet or block input-output operations may be considered as *request*, while corresponding driver and device are considered as *servers*. In our kiosk example, customer were the requests while clerk at the kiosk was the server.

To measure throughput we have to attach a probe to one of the functions responsible for handling requests, and use `count()` aggregation in it. It is preferable to use the last function responsible for that, because it will improve data robustness. Using a timer, we will print the aggregation value and clear it. For example, throughput of disk subsystem may be measured using following SystemTap script:

```
# stap -e ' global i o;
    probe i o block. end {
        size = 0
        for(vi = 0; vi < bi_vcnt; ++vi)
            size += $bi o->bi_i o_vec[vi ]->bv_l en;
        i o[devname]
```

Or with DTrace:

```
# dtrace -n '
    i o:::done {
```

```

        @[args[1]->dev_statname] = sum(args[0]->b_bcount);
    }
    tick-1s {
        printa(@);
        clear(@);
    }
}'

```

To measure *arrival rate*, on contrary, we need first functions which handle request "arrival" which are in our case `io_block.request` and `io::start` correspondingly. These probes will be covered in [Block Input-Output](#) section.

Latency measurement is a bit more complicated. We will need to add probes to request arrival and final handler and calculate time difference between these two moments. So we need to save a timestamp of a request arrival and retrieve it at the final handler probe. The easiest way to do that is thread-local variables, but it is not guaranteed that final handler will be called from same context request was created from. For example, final handler may be called from IRQ handler thread. In such cases we will need associative arrays and a unique request key retrievable on both sides, which is usually an address of requests descriptor in memory. For block input-output is `struct buf` in Solaris and `struct bio` in Linux. So let's calculate mean latency in SystemTap:

```

# stap -e ' global start, times;
    probe io_block.request {
        start[$bio] = gettimeofday_us();
    }
    probe io_block.end {
        if(start[$bio] != 0)
            times[devname]

```

Similar script is for DTrace:

```

# dtrace -qn '
    io:::start {
        iostart[arg0] = timestamp;
    }
    io:::done {
        @rq_svc_t[args[1]->dev_statname] = avg(timestamp - iostart[arg0]);
    }
    tick-1s {
        printf("%12s %8s %Y\n", "DEVICE", "ASVC_T", walltimestamp);
        printa("%12s %8d\n", @rq_svc_t);
        clear(@rq_svc_t);
    }
'

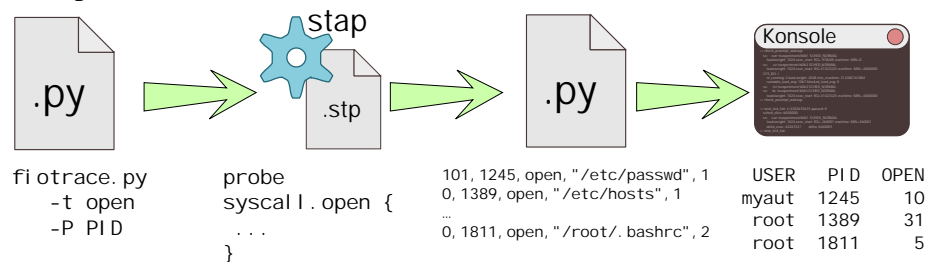
```

Utilization may be measured similar to a profiling: high-resolution timer determines if server is busy or not, so utilization will be busy ticks to all ticks ratio. Queue length may be modelled from arrival rate and dispatch rate, but in many cases it is explicitly accessible from kernel or application data.

Pre- and post-processing

Despite the flexibility of the dynamic tracing languages, it lacks of common tools to create user-friendly interfaces like command line options to generate different filtering with predicates, sorting and omitting columns, making scripts are hard to reuse. For example, `iosnoop` from DTraceToolkit allows to generate user-printable timestamps or not with `-v` option, filter device or PID with `-d` and `-p` options, and a series of options that enable or disable showing various columns.

In such cases we can use general purpose scripting language such as Python, Perl or even shell-script to generate dynamic tracing on-the fly, run it, read its output in some form and than print it in human-readable form:



For example, let's add the following capabilities to our `open()` system call tracer: customizable per-pid and per-user filters, and also make it universal — capable running in DTrace and SystemTap.

Source file: [scripts/src/opentrace.py](#)

First half of this script is an option parser implemented with `OptionParser` Python class and intended to parse command-line arguments, provide help for them and check their correctness — i.e. mutually-exclusive options, etc. Second half of the script is a `run_tracer()` function that accepts multiple arguments and `if-else` statement that depending on chosen dynamic tracing system, generates parameters for `run_tracer()` as follows:

Parameter	Description	SystemTap	DTrace
entry	entry probe name and body	<code>syscall::open</code>	<code>syscall::open*: entry</code> or <code>syscall::openat*: entry</code> depending on Solaris version
ret	return probe name and body	<code>syscall::open.return</code>	Similar to entry probe, but with name return
cond_proc	predicate for picking a process	<code>pid() != target()</code>	<code>pid == \$target</code>
cond_user	predicate template for per-user tracing	<code>uid() != %d</code>	<code>uid == %d</code>
cond_default	always-true predicate	0	1
env_bin_var	environment option used to override path to DTrace/SystemTap binary	STAP_PATH	DTRACE_PATH
env_bin_path	default path to DTrace/SystemTap binary	/usr/bin/stap	/usr/sbin/dtrace
opt_pid	option for tracing tool accepting PID	-x	-p

Parameter	Description	SystemTap	DTrace
opt_pid	option for tracing tool accepting new command	-C	-C
args	arguments to read script from stdin	-	-q -s /dev/fd/0
fmt_probe	format string for constructing probes		

So this script generate predicate condition `uid == 100` for the following command-line:
`# python opentrace.py -D -u 100`

Post-processing is intended to analyse already collected trace file, but it might be run in parallel with tracing process. However, it allows to defer trace analysis — i.e. collect maximum data as we can, and then cut out irrelevant data, showing only useful. This can be performed using either Python, Perl, or other scripting languages or even use static analysis languages like R. Moreover, post-processing allows to reorder or sort tracing output which can also help to avoid data mixing caused by per-process buffers.

The next script will read `opentrace.py` output, merge information from entry and return probes, and convert user-ids and time intervals to a convenient form. Like in dynamic tracing languages we will use an associative array `states` which is implemented as `dict` type in Python to save data from entry probes and use process ID as a key.

Source file: [scripts/src/openproc.py](#)

If we pipe `opentrace.py` output to this script, we can get similar data:

```
# python opentrace.py -c 'cat /tmp/not_exists' |
    python openproc.py
cat: cannot open /tmp/not_exists: No such file or directory
[...]
OPEN root 3584 /tmp/not_exists => ERROR -1 [10.17 us]
[...]
```

Warning

This is only a demonstration script, and many of their features may be implemented using SystemTap or DTrace. Moreover, they allow to use `system()` calls an external program, for example to parse `/etc/passwd` and get user name. However, it will cost much more, and if this call will introduce more `open()` calls (which it will obviously do), we will get more traced calls and a eternal loop.

Visualization

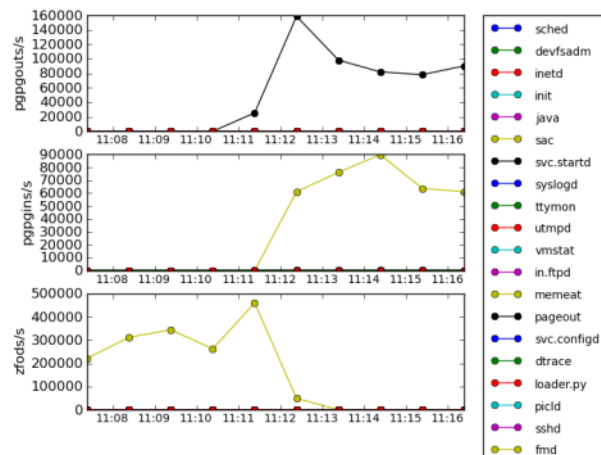
Reading trace files is exhausting, so the most popular scenario for the post-processing is visualization. There are multiple standard ways to do that:

- Use GNU Plot as shown here: [System utilization graphing with Gnuplot](#). Tracing script directly generates commands which are passed to GNU Plot.
- [DTrace Chime](#) plugin for the NetBeans
- [SystemTap GUI](#)
- Writing you own visualization script. For example, following examples were generated using Python library `matplotlib`.

Which types diagrams are mostly useful? Let's find out.

Linear diagram

The simplest one is a *linear diagram*. X axis in that diagram is the time, so it allows to see changes in system's behaviour over time. These diagrams may be combined together (but the time axis should be same on all plots), which allows to reveal correlations between characteristics, as shown on following image:



These three characteristics are names of the probes from vmi nfo provider in DTrace: zfod stands for *zero-filled on-demand* which is page allocation, while ppggi n and ppggout are events related to reading/writing pages to a backing store, such as disk swap partition. In this case, memeat process (which name is self-explanatory — it allocates all available RAM) allocates plenty of memory, so number of zfod events is high, causing to some pages being read or written to a disk swap.

Tracing a scheduler

Now let's run following loop in a shell which periodically eats lots of CPU than sleeps for 5 seconds:

```
# while :
do
    for I in {0..4000}
    do
        echo '1' > /dev/null
    done
    sleep 5
done
```

And gather scheduler trace: each time it dispatches a new process we will trace process name, cpu and timestamp:

```
# dtrace -n '
sched::on-cpu {
    printf("%d %d %s\n", timestamp,
    cpu, (curthread ==
    curthread->t_cpu->cpu_idle_thread)
    ? "idle"
    : execname ); }'
```


Histograms

In many cases of performance analysis we rely on average values, which are not very representative.

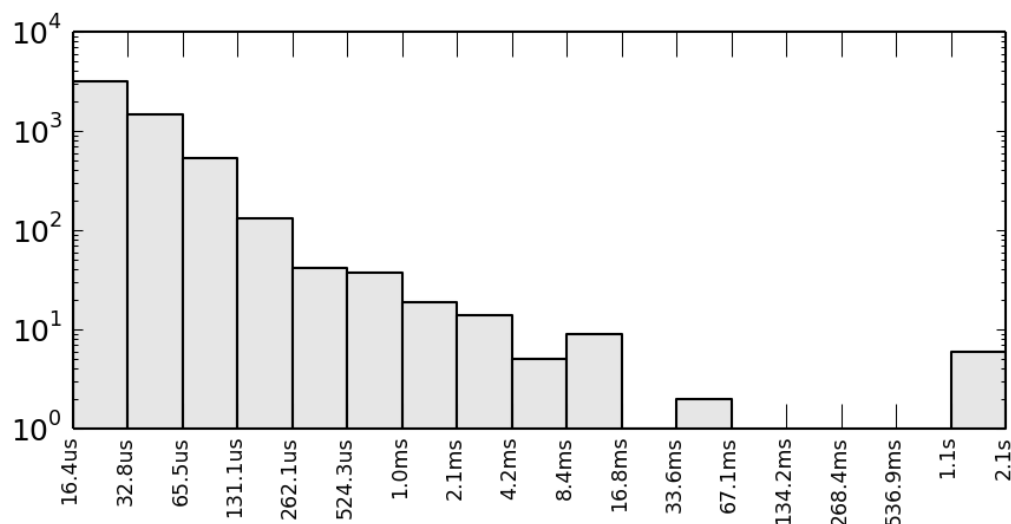
Information

Consider the following example: you apply to a job in some company which has 100 employees and *average* salary is about 30k roubles. This data can have many interpretations:

CEO salary	Senior staff salary	Junior staff salary
100k roubles	48k roubles	25k roubles
2 million roubles	23k roubles	7k roubles

Would you want to work there if salary is distributed according row? Doubtful. Like with employment, you cannot rely on average readings in performance analysis: average latency 10ms doesn't mean that all users are satisfied — some of them may had to wait seconds for web-page to render.

If we calculate per-process difference between scheduler timestamps and build a logarithmic histogram plot, we'll see several requests which lasts for seconds:



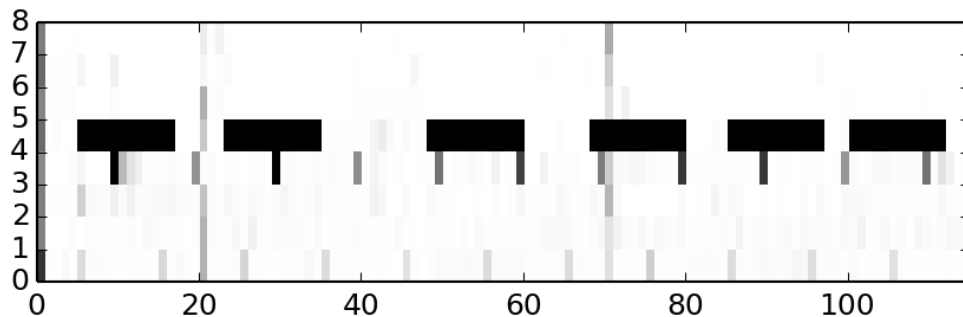
Y axis is logarithmic and represents a number of observed intervals when CPU was busy for time period shown on X axis. If we normalize this characteristic, we will get probability density function.

Warning

Aggregations `quantile()/hist_linear()` and `quantile()/hist_log()` might do the same, but in text terminal.

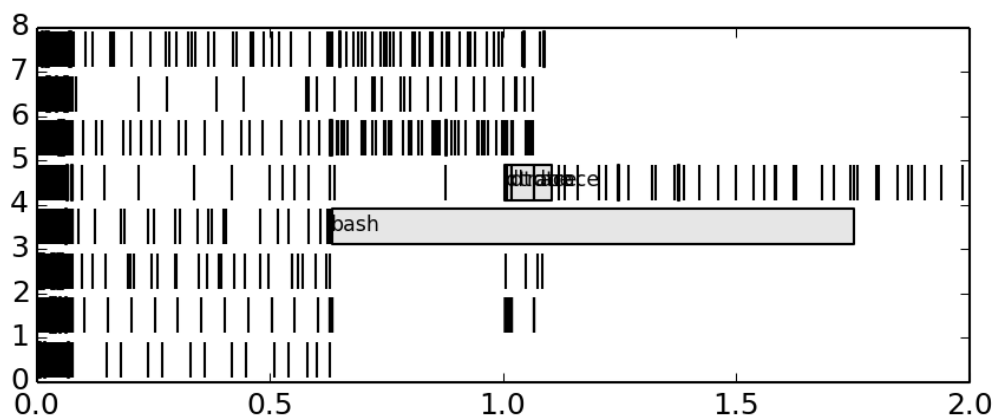
Heat maps

When two axes is not enough for your graph, you may also use a colour intensity of each pixel too. Let's see, how CPU usage is distributed across CPUs. To do so we need pick a step for the observation interval, say $T=100\text{ms}$, accumulate all intervals when non-idle thread were on that CPU, say t , than pixel's intensity will be $1.0 - t/T$ so 1.0 (white) will say that CPU was idle all the time, while 0.0 (black) will be evidence that CPU is very busy. For our example, we will see, that CPU 4 is periodically runs CPU-bound tasks:



Gantt charts

Generally speaking, gantt charts help to understand state of the system across the timeline. For example they are helpful in planning projects: what job needs to be done by whom and when, so the jobs are placed on X axis while Y axis is a timeline, and color is used to distinguish teams responsible for jobs. In our case we may be interested in how load distributed across CPUs, and what's causing it, so we here are a gantt chart:



We added process name to the longest bars, and it seems that bash process causing trouble. We could discover it before, adding tags on histogram.

Module 4: Operating system kernel tracing

Process management

Definition

According to Andrew Tanenbaum's book "Modern Operating Systems",

All the runnable software on the computer, sometimes including the operating system, is organized into a number of sequential processes, or just *processes* for short. A process is just an instance of an executing program, including the current values of the program counter, registers, and variables.

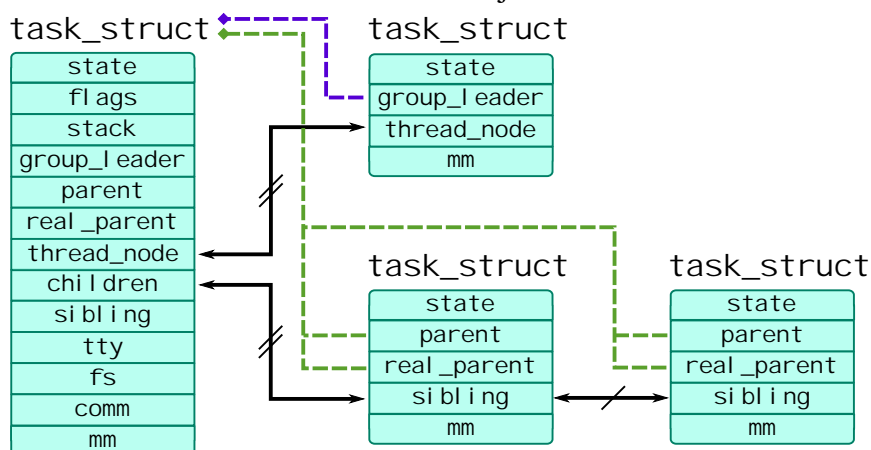
Information

Each process has its own address space — in modern processors it is implemented as a set of pages which map virtual addresses to a physical memory. When another process has to be executed on CPU, *context switch* occurs: after it processor special registers point to a new set of page tables, thus new virtual address space is used. Virtual address space also contains all binaries and libraries and saved process counter value, so another process will be executed after context switch. Processes may also have multiple *threads*. Each thread has independent state, including program counter and stack, thus threads may be executed in parallel, but they all threads share same address space.

Process tree in Linux

Processes and threads are implemented through universal `task_struct` structure (defined in `include/linux/sched.h`), so we will refer in our book as *tasks*. The first thread in process is called *task group leader* and all other threads are linked through list node `thread_node` and contain pointer `group_leader` which references `task_struct` of their process, that is, the `task_struct` of *task group leader*. Children processes refer to parent process through `parent` pointer and link through `sibling` list node. Parent process is linked with its children using `children` list head.

Relations between `task_struct` objects are shown in the following picture:



Task which is currently executed on CPU is accessible through current macro which actually calls function to get task from run-queue of CPU where it is called. To get current pointer in SystemTap, use `task_current()`. You can also get pointer to a `task_struct` using `pid2task()` function which accepts PID as its first argument. Task tapset provides several functions similar for functions used as [Probe Context](#). They all get pointer to a `task_struct` as their argument:

- `task_pid()` and `task_tid()` — ID of the process ID (stored in `tgid` field) and thread (stored in `pid` field) respectively. Note that kernel most of the kernel code doesn't check cached `pid` and `tgid` but use namespace wrappers.
- `task_parent()` — returns pointer to a parent process, stored in `parent/real_parent` fields
- `task_state()` — returns state bitmask stored in `state`, such as `TASK_RUNNING` (0), `TASK_INTERRUPTIBLE` (1), `TASK_UNINTERRUPTIBLE` (2). Last 2 values are for sleeping or waiting tasks — the difference that only interruptible tasks may receive signals.
- `task_execname()` — reads executable name from `comm` field, which stores base name of executable path. Note that `comm` respects symbolic links.
- `task_cpu()` — returns CPU to which task belongs

There are several other useful fields in `task_struct`:

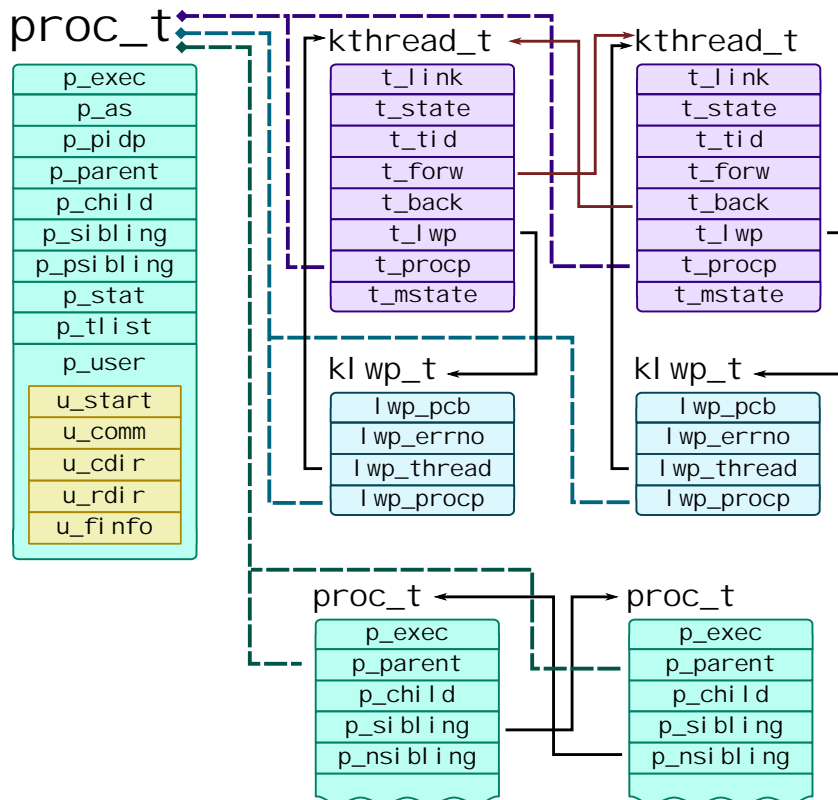
- `mm` (pointer to struct `mm_struct`) refers to a address space of a process. For example, `exe_file` (pointer to struct `file`) refers to executable file, while `arg_start` and `arg_end` are addresses of first and last byte of `argv` passed to a process respectively
- `fs` (pointer to struct `fs_struct`) contains filesystem information: `path` contains working directory of a task, `root` contains root directory (alterable using `chroot` system call)
- `start_time` and `real_start_time` (represented as struct `timespec` until 3.17, replaced with `u64` nanosecond timestamps) — *monotonic* and *real* start time of a process.
- `files` (pointer to struct `files_struct`) contains table of files opened by process
- `utime` and `stime` (`cputime_t`) contain amount of time spent by CPU in userspace and kernel respectively. They can be accessed through Task Time tapset.

Script `dumtask.stp` demonstrates how these fields may be useful to get information about current process.

Source file: [scripts/stap/dumtask.stp](#)

Process tree in Solaris

Solaris Kernel distinguishes threads and processes: on low level all threads represented by `kthread_t`, which are presented to userspace as *Light-Weight Processes* (or *LWPs*) defined as `klwp_t`. One or multiple LWPs constitute a process `proc_t`. They all have references to each other, as shown on the following picture:



Current thread is passed as `curthread` built-in variable to probes. Solaris `proc` provider has `lwpsinfo_t` and `psinfo_t` providers that extract useful information from corresponding thread, process and LWP structures.

		Description
Process		
<code>psinfo_t</code> field	<code>proc_t</code> field	
<code>p_exec</code>		vnode of executable file
<code>p_as</code>		process address space
<code>pr_pid</code>	In <code>p_pid</code> of type <code>struct pid</code>	Information about process ID
<code>pr_uid</code> , <code>pr_gid</code> , <code>pr_euid</code> , <code>pr_egid</code>	In <code>p_cred</code> of type <code>struct cred</code>	User and group ID of a process
	<code>p_stat</code>	Process state
<code>pr_dmodel</code>	<code>p_model</code>	Data model of a process (32- or 64- bits)
<code>pr_start</code>	<code>p_user.u_start</code> , <code>p_mstart</code>	Start time of process, from epoch
<code>pr_fname</code>	<code>p_user.u_comm</code>	Executable name
	<code>p_user.p_cdir</code>	vnode of current process directory

		<i>Description</i>
	<code>p_user.p_rdir</code>	vnode of root process directory
For current process -- <code>fds</code> pseudo-array	<code>p_user.u_finfo</code>	Open file table
<i>Thread / LWP</i>		
<i>lwpsinfo_t field</i>	<i>kthread_t field</i>	<i>Description</i>
<code>pr_lwpid</code>	<code>t_tid</code>	ID of thread/LWP
<code>pr_state</code> (enumeration) or <code>pr_sname</code> (letter)	<code>t_state</code>	State of the thread -- one of SSLEEP for sleeping, SRUN for runnable thread, SONPROC for thread that is currently on process, SZOMB for zombie threads, SSTOP for stopped threads and SWAIT for threads that are waiting to be runnable.
<code>pr_stype</code>		If process is sleeping on synchronization object identifiable as <i>wait channel</i> (<code>pr_wchan</code>), this field contains type of that object, i.e.: SOBJ_Mutex for mutexes and SOBJ_CV for condition variables
	<code>t_mstate</code>	micro-state of thread (see also <code>prstat -m</code>)

Parent process has `p_child` pointer that refers its first child, while list of children is doubly-linked list with `p_sibling` pointer (next) and `p_prev` (previous) pointers. Each child contains `p_parent` pointer and `p_ppid` process ID which refers his parent. Threads of the process is also a doubly-linked list with `t_forw` (next) and `t_prev` pointers. Thread references corresponding LWP with `t_lwp` pointer and its process with `t_procp` pointer. LWP refers to a thread through `lwp_thread` pointer, and to a process through `lwp_procp` pointer.

The following script dumps information about current thread and process. Because DTrace doesn't support loops and conditions, it can read only first 9 files and 9 arguments and does that by generating multiple probes with preprocessor.

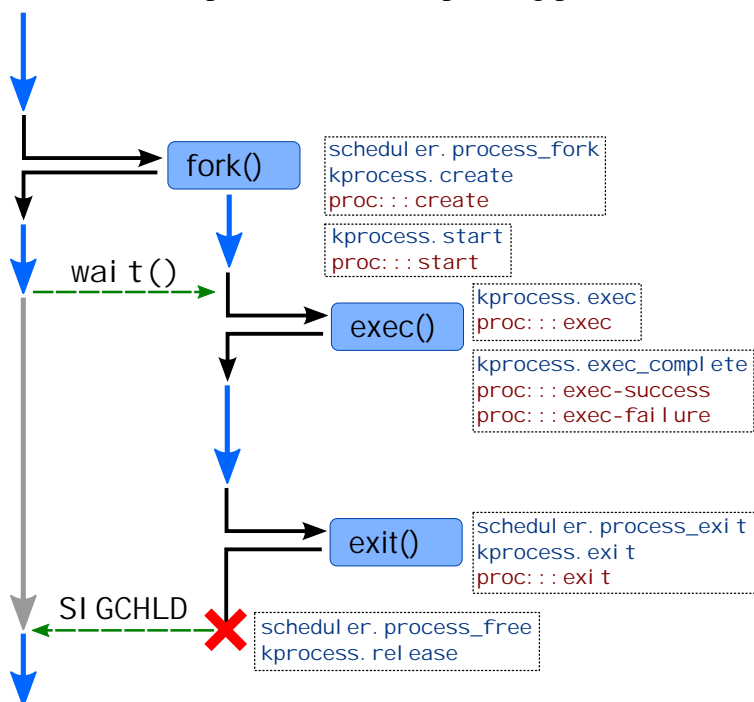
Source file: [scripts/dtrace/dumptask.d](#)

Warning

`psinfo_t` provider features `pr_psargs` field that contains first 80 characters of process arguments. This script uses direct extraction of arguments only for demonstration purposes and to be conformant with `dumptask.stp`. Like in SystemTap case, this approach doesn't allow to read non-current process arguments.

Lifetime of a process

Lifetime of a process and corresponding probes are shown in the following image:



Unlike Windows, in Unix process is spawned in two stages:

- Parent process calls `fork()` system call. Kernel creates exact copy of a parent process including address space (which is available in copy-on-write mode) and open files, and gives it a new PID. If `fork()` is successful, it will return in the context of two processes (parent and child), with the same instruction pointer. Following code usually closes files in child, resets signals, etc.
- Child process calls `execve()` system call, which replaces address space of a process with a new one based on binary which is passed to `execve()` call.

Warning

There is a simpler call, `vfork()`, which will not cause copying of an address space and make it a bit more efficient. Linux features universal `clone()` call which allows to choose which features of a process should be cloned, but in the end, all these calls are wrappers for `do_fork()` function.

When child process finishes its job, it will call `exit()` system call. However, process may be killed by a kernel due to incorrect condition (like triggering kernel oops) or machine fault. If parent wants to wait until child process finishes, it will call `wait()` system call (or `waitid()` and similar calls), which will stop parent from executing until child exits. `wait()` call also receives process exit code, so only after that corresponding `task_struct` will be destroyed. If no process waits on a child, and child is exited, it will be treated as *zombie* process. Parent process may be also notified by kernel with `SIGCHLD` signal.

Processes may be traced with `kprocess` and `scheduler` tapsets in SystemTap, or DTrace proc provider. System calls may be traced with appropriate probes too. Here are some useful probes:

Action	DTrace	SystemTap
Process creation	<code>proc::create</code>	<ul style="list-style-type: none"> <code>kprocess.create</code> <code>scheduler.process_fork</code>

Action	DTrace	SystemTap
Forked process begins its execution	<ul style="list-style-type: none"> • <code>proc:::start</code> — called in new process context 	<ul style="list-style-type: none"> • <code>kprocess.start</code> — called in a new process context • <code>schedul er.wakeup_new</code> — process has been dispatched onto CPU first time
<code>execve()</code>	<ul style="list-style-type: none"> • <code>proc:::exec</code> — entering <code>execve()</code> • <code>proc:::exec-success</code> — <code>execve()</code> finished successfully • <code>proc:::exec-failure</code> — <code>execve()</code> has failed, <code>args[0]</code> contains <code>errno</code> 	<ul style="list-style-type: none"> • <code>kprocess.exec</code> — entering <code>execve()</code> • <code>kprocess.exec_complete</code> — <code>execve()</code> has been completed, <code>success</code> variable has true-value if completed successfully, <code>errno</code> variable has error number in case of error
Process finished	<ul style="list-style-type: none"> • <code>process:::exit</code> — process exited normally via <code>exit()</code> syscall • <code>process:::fault</code> — process has been terminated due to fault 	<ul style="list-style-type: none"> • <code>kprocess.exit</code> • <code>schedul er.process_exit</code>
Process structures deallocated due to <code>wait()/SIGCHLD</code>	-	<ul style="list-style-type: none"> • <code>kprocess.release</code> • <code>schedul er.process_free</code>
LWP management	<ul style="list-style-type: none"> • <code>proc:::lwp-create</code> • <code>proc:::lwp-start</code> • <code>proc:::lwp-exit</code> 	LWPs are not supported in Linux

These probes are demonstrated in the following scripts.

Source file: [scripts/stap/proc.stp](#)

Running this script for `uname` program called in foreground of `bash` shell gives following output:

```

2578[  bash]/ 2576[  sshd] syscall.fork
2578[  bash]/ 2576[  sshd] kprocess.create
2578[  bash]/ 2576[  sshd] schedul er.process_fork
    PID: 2578 -> 3342
2578[  bash]/ 2576[  sshd] schedul er.wakeup_new
3342[  bash]/ 2578[  bash] kprocess.start
2578[  bash]/ 2576[  sshd] syscall.wait4
2578[  bash]/ 2576[  sshd] schedul er.process_wai t
    filename: /bin/uname
3342[  bash]/ 2578[  bash] kprocess.exec
3342[  bash]/ 2578[  bash] syscall.execve
3342[  uname]/ 2578[  bash] kprocess.exec_compl ete
    return code: 0
3342[  uname]/ 2578[  bash] kprocess.exit
3342[  uname]/ 2578[  bash] syscall.exit
3342[  uname]/ 2578[  bash] schedul er.process_exit
2578[  bash]/ 2576[  sshd] kprocess.release

```

Source file: [scripts/dtrace/proc.d](#)

DTrace will give similar outputs, but also will reveal LWP creation/destruction:

```

16729[  bash]/ 16728[  sshd] syscall::forksys:entry
16729[  bash]/ 16728[  sshd] proc:::lwp_create:lwp-create
16729[  bash]/ 16728[  sshd] proc:::cfork:create
    PID: 16729 -> 17156
16729[  bash]/ 16728[  sshd] syscall::waitsys:entry
17156[  bash]/ 16729[  bash] proc:::lwp_rtt_i ni ti al:start

```








```

17156[    bash]/ 16729[    bash] proc: : lwp_rtt_initial: lwp-start
17156[    bash]/ 16729[    bash] syscall: : exece: entry
17156[    bash]/ 16729[    bash] proc: : exec_common: exec
      filename: /usr/sbin/uname
17156[    uname]/ 16729[    bash] proc: : exec_common: exec-success
17156[    uname]/ 16729[    bash] proc: : proc_exit: lwp-exit
17156[    uname]/ 16729[    bash] proc: : proc_exit: exit
      return code: 1
      0[    sched]/      0[    ???] proc: : sigtoproc: signal-send

```

References

-  [Context Functions](#)
-  [Task Time Tapset](#)
-  [Kernel Process Tapset](#)
-  [Scheduler Tapset](#)
-  [proc Provider](#)

Exercise 3

Part 1

Modify `dumtask.stp` and `dumtask.d` so it will print information on successful binary load by `execve()` and before process exit. Write a simple program `lab3.c`:

Source file: [scripts/src/lab3.c](#)

Compile it with GCC:

```
# gcc lab3.c -o lab3
```

Run changed scripts and run your program in different ways:

- Run it with argument:

```
# ./lab3 arg1
```

- Create a symbolic link and run a program through it:

```
# ln -s lab3 lab3-1
# ./lab3-1
```

- Created chrooted environment and run `lab3` inside it:

```

# mkdir -p /tmp/chroot/bin /tmp/chroot/lib64 /tmp/chroot/lib
# mount --bind /lib64 /tmp/chroot/lib64                (in Linux)
# mount -F lofs /lib /tmp/chroot/lib                    (in Solaris)
# cp lab3 /tmp/chroot/bin
# chroot /tmp/chroot/ /bin/lab3

```

Q: What data output has been changed? Try to explain these changes.

Part 2

Shell scripts have overhead caused by need to spawn new processes for basic operations, and thus calling `fork()` and `execve()`. Write SystemTap and DTrace scripts that measure following characteristics:

- time, spent for `fork()` and `execve()` system calls;

- Current task leaves CPU. This event is traceable as `sched:::off-cpu` in DTrace or `schedul er. cpu_off` in SystemTap. This may be caused by many reasons:

- Task was blocked on kernel synchronisation object (6), for example due to call to `poll()` and waiting network data. In this case task is put into *sleep queue* related to that synchronisation object. It would later be unblocked by another task (7), thus being put back to run-queue.

- Task is voluntary gives control to a scheduler by calling `yield()` call (3)

- Task has been exhausted its timeslice or task with higher priority has been added to a run queue (or a new, forked process added to queue), which is called *preemptiveness*. Usually timeslice is checked by system timer, which is traceable as `sched:::tick` probe in DTrace or `schedul er. tick`. (3) Some system calls and interrupts may also trigger context switch.

- New task is picked to be run on CPU (2). When CPU resumes from kernel mode, interrupt or system call, it changes context to a new task, like in `resume()` low-level routine of Solaris. Context switch may be traced in SystemTap with `schedul er. ctxswi tch` probe.

OS creates at least one run-queue per CPU in SMP systems. When some CPU prepares to become idle, it may check run-queues of other CPUs and *steal* task from it, thus task *migrates* (5) between CPUs. This allows to balance tasks across CPUs, but other factors like NUMA locality of process memory, cache hotness should be taken into account. Migration may be tracked by with `schedul er. mi grate` probe in SystemTap. DTrace doesn't provide special probe for that, but it may be tracked comparing CPU ids in `on-cpu` and `off-cpu` probes:

```
# dtrace -n '
    sched:::off-cpu {
        sel f->cpu = cpu; }
    sched:::on-cpu
    /sel f->cpu != cpu/
    {
        /* Mi grati on */    } '
```

Usually task is blocked on various synchronisation objects waiting for data available for processing, i.e. `accept()` will block until client will connect and `recv()` will block until client will send new data. There is no need to use a processor when no data is available, so task simply leaves CPU and being put to a special *sleep queue* related to that object. Speaking of `accept()` call, it would be `so_acceptq_cv` condition variable in kernel socket object (sonode) in Solaris and `sk_wq` wait queue wrapper in Linux object `sock`. We will cover synchronisation objects in detail later in section [Synchronisation objects](#).

Solaris has dedicated probes for sleeping and awoken processes: `sched:::sleep` and `sched:::wakeup` correspondingly which may be used like this:

```
# dtrace -n '
    sched:::sleep {
        pri ntf("%s[%d] sl eepi ng", execname, pi d);
    }
    sched:::wakeup {
        pri ntf("%s[%d] wakes up %s[%d]", execname, pi d,
            args[1]->pr_fname, args[1]->pr_pi d); }' | grep cat
```

Note that `wakeup` process is called in context of process which initiates task unblocking.

SystemTap provides `schedul er. wakeup` probe for process that return to a run-queue, but has no special probe for sleeping process. The most correct way to do that is to trace `schedul e()` calls and task state transitions: task should change its state from `TASK_RUNNI NG` to a `TASK_I NTERRUPTI BLE` or `TASK_UNI NTERRUPTI BLE`. In following example, however, we

will use much simpler approach: most sleep queues are implemented as *wait queues* in Linux, so we will trace corresponding call, `add_wai t_queue()` that puts task onto queue:

```
# stap -e '
  probe kernel.function("add_wai t_queue") {
    printf("%s[%d] sleeping\n", execname(), pid());
  }
  probe scheduler.wakeup {
    printf("%s[%d] wakes up %s[%d]\n", execname(), pid(),
      pid2execname(task_pid), task_pid); }' | grep cat
```

These examples may be tested with the following simple one-liner:

```
# bash -c 'while : ; do echo e ; sleep 0.5 ; done ' | cat
```

When dispatcher puts task onto queue, it is called *enqueueing* (4), when it removes it from queue, it is called *dequeueing* (1). DTrace has probes `sched::enqueue` and `sched::dequeue`. SystemTap doesn't have these probes, but you may trace `enqueue_task()` and `dequeue_task()` for that.

As we mentioned before, purpose of the scheduler is to distribute time between tasks. To do that, it prioritizes tasks, so to pick a task for execution it may create multiple queues (one for each priority level), than walk over these queues and pick *first* task with top-level priority. Such approach is called *cyclic planning*. *Fair planning*, on contrary, is concerned about time consumption by difference threads, processes, users and even services (which are all considered as *scheduling entities*), and try to balance available processor time fairly.

Scheduling in Solaris

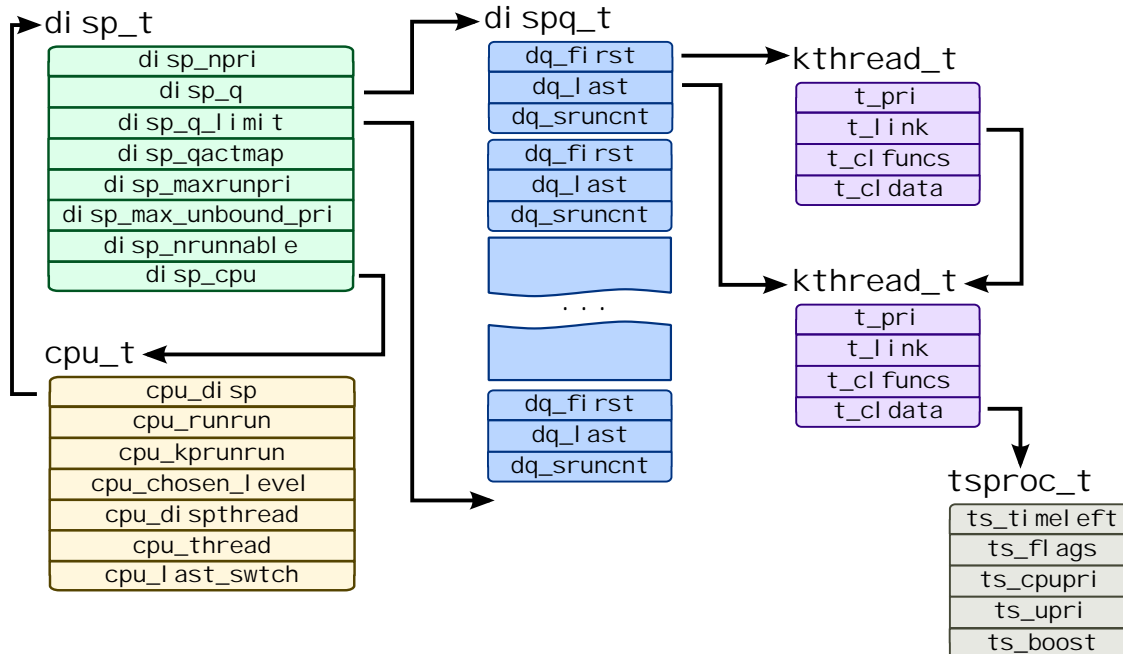
Solaris implements cyclic scheduling, but it support fair scheduling algorithms via FSS (*fair share scheduling*) class. Each thread in Solaris may have priority from 0 to 170 which is saved in `t_pri` field of `kthread_t`. Each thread has its own scheduler class, which may have different algorithms for allocating timeslices, prioritizing threads. Generic API for scheduler class is provided as `t_cl_funcs` field of thread where each field is a pointer to a corresponding function, while specific scheduler data is kept under `t_cl_data` field.

The following table shows scheduler classes implemented in Solaris.

Class	Priority range	Description
-	160-169	Interrupt threads (they are not handled by scheduler explicitly).
RT	100-159	<i>RealTime</i> processes and threads.
SYS	60-99	<i>SYStem</i> — for kernel threads which always have precedence over user processes. Also, timeslices are allocated to them, they consume as much processor time as they can.
SYSDC	0-99	<i>SYStem Duty Cycles</i> — for CPU-bound kernel threads like ZFS pipeline (which involves encryption and compression). This class is implemented to prevent userspace starvation under heavy I/O load.
TS and IA	0-59	<i>Time Sharing</i> and <i>InterActive</i> — default classes for userspace processes. TS class has dynamic priorities: if thread had consumed entire timeslice, its priority is reduced to give room to other threads. IA is a modification of TS class which also adds small "boost" (usually, 10 priorities) for processes which have focused window (useful for graphics stations).
FX	0-59	<i>FiXed</i> — unlike TS/IA, such processes never change their priorities unless it is done by user themself.

Class	Priority range	Description
FSS	0-59	<i>Fair Share Scheduler</i> — allows to distribute processor time proportionally between groups of processes such as zones or projects.

Solaris dispatcher control structures are shown on the following picture:



Each processor has corresponding `cpu_t` structure, which includes two pointers to threads: `cpu_di_spthread` — a thread chosen by scheduler to be the next process after resume, and `cpu_thread` — process which is currently is on CPU. `cpu_last_swch` contains time of last context switch in lbolts (changed into high-resolution time in nanoseconds in Solaris 11). Each `cpu_t` has dispatcher queue represented by `di_sp_t` structure and corresponding array of queue heads of type `di_spq_t`. Each queue head has links to first and last thread in queue (`kthread_t` objects are linked through `t_link` pointer) and `dq_sruncnt` — number of threads in this queue.

`di_sp_t` refers queues through `di_sp_q` pointer which refers first queue with priority 0 and `di_sp_q_limit` which points one entry beyond array of dispatcher queues. `di_sp_qactmap` contains bitmap of queues that have active processes at the moment. `di_sp_npri` is the number of priorities serviced by this dispatcher object — it should be 160. `di_sp_maxrunpri` contains maximum priority of a thread in this dispatcher object — it will be top-most queue which has active processes and when thread will be switched, this queue will be checked first. `di_sp_max_unbound_pri` also contains maximum priority of a thread, but only for a thread that is not bound to a corresponding processor and thus may be considered a candidate for task-stealing by free CPUs. Finally, `di_sp_nrunnable` has total number of runnable threads which is serviced by this dispatcher object.

Warning

Newer Solaris 11 versions use `hrtime_t` type for `cpu_last_swch` (high-resolution unscaled time).

By default Solaris userspace processes use TS scheduler, so let's look into it. Key parameter that used in it is `ts_timeleft` which keeps remaining thread timeslice. Initial value of `ts_timeleft` is taken from table `ts_dptbl` from field `ts_quantum`. Each row in that table matches priority level, so processes with lower priorities will have larger quantum (because

they will get CPU rarely). You can check that table and override its values with `di spadmi n` command, for example:

```
# di spadmi n -c TS -g
```

Priority is also set dynamically in TS scheduler: if thread will exhaust its timeslice, its priority will be lowered according to `ts_tqexp` field, and if it will be awoken after sleep, it will get `ts_slpret` priority. Modern Solaris systems were replaced `ts_timeleft` with `ts_timer` (for non-kernel threads those have `TSKPRI` flag is set).

Tracer for TS scheduler is available in the following listing:

Source file: [scripts/dtrace/tstrace.d](#)

Let's demonstrate some of TS features live. To do that we will conduct two TSLoad experiments: *duality* and *concurrency*. In first experiment, *duality*, we will create two different types of threads: *workers* which will occupy all available processor resources, while *manager* will rarely wakeup (i.e. to queue some work possibly and report to user), so it should be immediately dispatched. In our example manager had LWPID=7 while worker had LWPID=5. Experiment configuration is shown in the following file:

Source file: [experiments/duality/experiment.json](#)

Here is sample output for *duality* experiment (some output was omitted):

```
=> cv_unsleep [wakeup]
    CPU : last switch: T-50073us rr: 0 kpr: 0
    wakeup t: fffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
=> setbackdq
    setbackdq t: fffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
=> setfrontdq
    setfrontdq t: fffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
=> di sp
    CPU : last switch: T-50140us rr: 1 kpr: 0
           current t: fffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
           disp t: fffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
    DISP: nrun: 2 npri: 170 max: 59(65535)
    curthread: t: fffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
    TS: timeleft: 19 flags: cpupri: 0 upri: 0 boost: 0 => 0
    disp t: fffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
    TS: timeleft: 3 flags: cpupri: 59 upri: 0 boost: 0 => 0
=> di sp
    CPU : last switch: T-1804us rr: 0 kpr: 0
           current t: fffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
           disp t: fffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
    DISP: nrun: 1 npri: 170 max: 0(65535)
    curthread: t: fffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
    disp t: fffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
```

Note that when manager wakes up, it has maximum priority: 59 but being put to the queue tail. After that worker thread is being queued because `cpu_runrun` flag is being set (note `rr` change), number of runnable processes increases up to two. After 1.8 ms manager surrenders from CPU, and worker regains control of it.

In the *concurrency* experiment, on the opposite we will have two threads with equal rights: they both will occupy as much CPU as they can get thus being *worker* processes. Experiment configuration is shown in the following file:

Source file: [experiments/concurrency/experiment.json](#)

Here is sample output for *concurrency* experiment (some output was omitted):

```

=> di sp
    CPU : last switch: T-39971us rr: 1 kpr: 0
          current t: fffffc10009711800 tsexperiment[1391]/6 TS pri: 40
          disp    t: fffffc10009711800 tsexperiment[1391]/6 TS pri: 40
    DISP: nrun: 2 npri: 170 max: 40(65535)
    curthread:  t: fffffc10009711800 tsexperiment[1391]/6 TS pri: 40
    TS: timeleft: 4 flags:  cpupri: 40 upri: 0 boost: 0 => 0
    disp t: fffffc10005c07420 tsexperiment[1391]/5 TS pri: 40
    TS: timeleft: 4 flags:  cpupri: 40 upri: 0 boost: 0 => 0
=> clock_tick
    clock_tick t: fffffc10005c07420 tsexperiment[1391]/5 TS pri: 40
    TS: timeleft: 3 flags:  cpupri: 40 upri: 0 boost: 0 => 0
=> clock_tick
    clock_tick t: fffffc10005c07420 tsexperiment[1391]/5 TS pri: 40
    TS: timeleft: 2 flags:  cpupri: 40 upri: 0 boost: 0 => 0
=> clock_tick
    clock_tick t: fffffc10005c07420 tsexperiment[1391]/5 TS pri: 40
    TS: timeleft: 1 flags:  cpupri: 40 upri: 0 boost: 0 => 0
    cpu_surrender t: fffffc10005c07420 tsexperiment[1391]/5 TS pri: 30
=> clock_tick
    clock_tick t: fffffc10005c07420 tsexperiment[1391]/5 TS pri: 30
    TS: timeleft: 0 flags: TSBACKQ| cpupri: 30 upri: 0 boost: 0 => 0
=> setbackdq
    setbackdq t: fffffc10005c07420 tsexperiment[1391]/5 TS pri: 30
    TS: timeleft: 8 flags:  cpupri: 30 upri: 0 boost: 0 => 0
=> di sp
    curthread:  t: fffffc10005c07420 tsexperiment[1391]/5 TS pri: 30
    TS: timeleft: 8 flags:  cpupri: 30 upri: 0 boost: 0 => 0
    disp t: ffffffff80389c00 sched[0]/0 SYS pri: 60
=> di sp
    curthread:  t: ffffffff80389c00 sched[0]/0 SYS pri: 60
    disp t: fffffc10009711800 tsexperiment[1391]/6 TS pri: 40
    TS: timeleft: 4 flags:  cpupri: 40 upri: 0 boost: 0 => 0

```

Note how timeleft field is changing: it is calculated as `ts_timer - ts_lwp->lwp_ac.ac_clock`. After each clock tick latter is incremented thus timeleft is decrementing. When timeleft becomes 0, it means that *worker* has exhausted scheduler quantum, so its priority falls from 40 to 30 and it is being put to the tail of corresponding dispatcher queue. After that sched thread runs for a short time (which is some kernel thread managed by SYS scheduler), and eventually another *worker* thread gets on CPU.

Scheduling in Linux

Cyclic scheduling was implemented in Linux O(1) scheduler, but it was replaced with Completely Fair Scheduler (CFS) scheduler in 2.6.22 kernel. Cyclic scheduling is represented by RT class which is rarely used. There are also some non-default schedulers like BFS which are not available in vanilla kernel but shipped as separate patches. Each `task_struct` has field `policy` which determines which scheduler class will be used for it. Policies are shown in the following table:

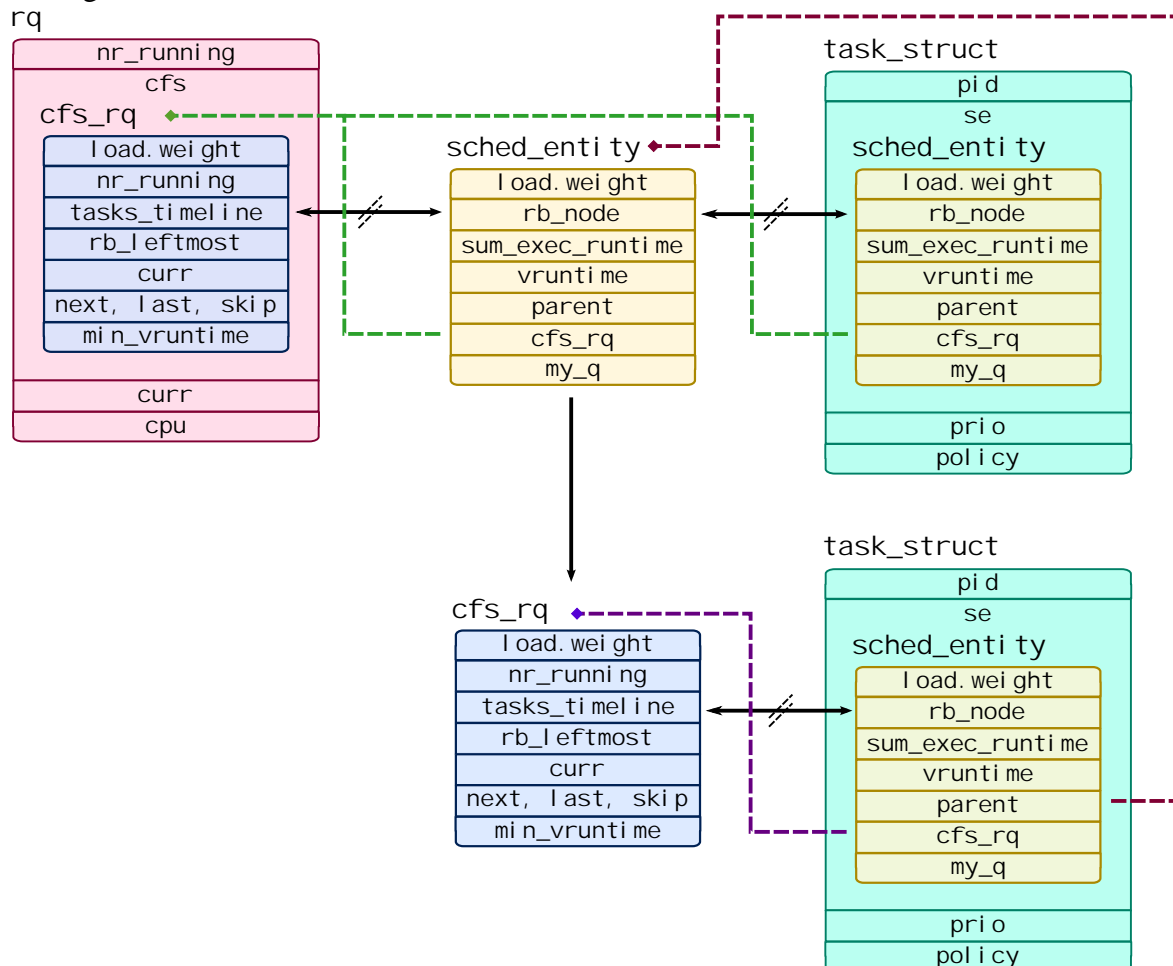
Priority	Class	Policy	Description
1	stop	-	Special class for stopped CPUs. Such CPUs cannot execute any threads.

Priority	Class	Policy	Description
2	rt	SCHED_RR	Implements cyclic scheduling using round-robin or FIFO policies
		SCHED_FIFO	
3	fair (CFS)	SCHED_NORMAL (SCHED_OTHER)	Default policy for most kernel and user threads
		SCHED_BATCH	Similar to SCHED_NORMAL, but process which was recently waken up won't try to dispatch on CPU which is more fitful for batch tasks
4	idle	SCHED_IDLE	Idle threads — picked only when other classes do not have runnable threads.

Information

Consider the following situation: there are currently two users on a 8-CPU host where user dima had run `make -j 8` and another user, say myaut, had run `make -j 4`. To maintain fairness so users dima and myaut will get equal amount of CPU time, you will need renice processes of user dima, but calculating correct priority penalty will be inobvious. Instead, you can create a two CGroups and add one instance of `make` per CGroup. Than all tasks which are spawned by dima's `make` will be accounted in scheduler entity corresponding to dima's CGroup.

Let's look into details of implementation of CFS scheduler. First of all, it doesn't deal with tasks directly, but schedules *scheduler entities* of type `struct sched_entity`. Such entity may represent a task or a queue of entities of type `struct cfs_rq` (which is referenced by field `my_q`), thus allowing to build hierarchies of entities and allocate resources to task groups which are called *CGroups* in Linux. Processor run queue is represented by type `struct rq` contains field `cfs` which is instance of `struct cfs_rq` and contains queue of all high-level entities. Each entity has `cfs_rq` pointer which points to CFS runqueue to which that entity belongs:



In this example processor run queue has two scheduler entities: one CFS queue with single task (which refers top-level `cfs_rq` through `parent` pointer) in it and one top-level task.

CFS doesn't allocate timeslices like TS scheduler from Solaris did. Instead it accounts total time which task had spend on CPU and saves it to `sum_exec_runtime` field. When task is dispatched onto CPU, its `sum_exec_runtime` saved into `prev_sum_exec_runtime`, so calculating their difference will give time period that task spent on CPU since last dispatch. `sum_exec_runtime` is expressed in nanoseconds but it is not directly used to measure task's runtime. To implement priorities, CFS uses task weight (in field `load.weight`) and divides runtime by tasks weight, so tasks with higher weights will advance their runtime meter (saved into `vruntime` field) slower. Tasks are sorted according their `vruntime` in a red-black tree called `tasks_timeline`, while left-most task which has lowest `vruntime` of all tasks and saved into `rb_leftmost`.

CFS has special case for tasks that have been woken up. Because they can be sleeping too long, their `vruntime` may be too low and they will get unfairly high amount of CPU time.

To prevent this, CFS keeps minimum possible `vruntime` of all tasks in `min_vruntime` field, so all waking up tasks will get `min_vruntime` minus a predefined "timeslice" value. CFS also have a *scheduler buddies* — helper pointers for a dispatcher: `next` — task that was recently awoken, `last` — task that recently was evicted from CPU and `skip` — task that called `sched_yield()` giving CPU to other entities.

So, let's implement a tracer for CFS scheduler:

Source file: [scripts/stap/cfstrace.stp](#)

Let's conduct same experiments we performed on Solaris. In "duality" experiment *manager* task (TID=6063) didn't preempt *worker* (TID=6061) immediately, but it was put into `task_timeline` tree. Since it would have minimum `vruntime` of all tasks there (note that CFS scheduler removes task from queue when it is dispatched onto CPU), it becomes left-most task. It picked on a next system tick:

```
=> check_preempt_wakeup:
  se:   curr tsexperiment/6061 SCHED_NORMAL
        load.weight: 1024 exec_start: RQ+-978205 vruntime: MIN+0
  se:   se tsexperiment/6063 SCHED_NORMAL
        load.weight: 1024 exec_start: RQ+41023325 vruntime: MIN+-6000000
  CFS_RQ: /
        nr_running: 2 load.weight: 2048 min_vruntime: 314380161884
        runnable_load_avg: 1067 blocked_load_avg: 0
  se:   first tsexperiment/6063 SCHED_NORMAL
  se:   rb: tsexperiment/6063 SCHED_NORMAL
        load.weight: 1024 exec_start: RQ+41023325 vruntime: MIN+-6000000
check_preempt_wakeup

=> task_tick_fair J=4302675615 queued: 0 curr: tsexperiment/6061 SCHED_NORMAL
      sched_slice: 6000000
      se:   curr tsexperiment/6061 SCHED_NORMAL
            load.weight: 1024 exec_start: RQ+-260001 vruntime: MIN+260001
            delta_exec: 42261531          delta: 6260001
task_tick_fair

=> pick_next_task_fair D=42422710 J=4302675615
      pick_next_entity
      CFS_RQ: /
            nr_running: 2 load.weight: 2048 min_vruntime: 314380161884
            runnable_load_avg: 1067 blocked_load_avg: 0
      se:   first tsexperiment/6063 SCHED_NORMAL
            load.weight: 1024 exec_start: RQ+42261531 vruntime: MIN+-6000000
      se:   rb: tsexperiment/6063 SCHED_NORMAL
            load.weight: 1024 exec_start: RQ+42261531 vruntime: MIN+-6000000
      se:   rb-r: tsexperiment/6061 SCHED_NORMAL
            load.weight: 1024 exec_start: RQ+-131878 vruntime: MIN+391879
pick_next_task_fair
      se:   sched tsexperiment/6063 SCHED_NORMAL

In concurrency experiment thread receives 6ms timeslice (return value of sched_slice() function), so it will be executed until its vruntime won't exceed min_vruntime:

=> pick_next_task_fair D=7015045 J=4302974601
      pick_next_task_fair
      se:   sched t: 0xfffff880015ba0000 tsexperiment/6304 SCHED_NORMAL

=> task_tick_fair J=4302974602 queued: 0 curr: t: 0xfffff880015ba0000 tsexperiment/6304
      SCHED_NORMAL
```

```

    sched_slice: 6000000
    se:      curr tsexperiment/6304 SCHED_NORMAL
           load.weight: 1024 exec_start: RQ+-868810 vruntime: MIN+0
           delta_exec: 868810      delta: -4996961
task_tick_fair
...

=> task_tick_fair J=4302974608 queued: 0 curr: t: 0xfffff880015ba0000 tsexperiment/6304
    SCHED_NORMAL
    sched_slice: 6000000
    se:      curr tsexperiment/6304 SCHED_NORMAL
           load.weight: 1024 exec_start: RQ+-1007610 vruntime: MIN+1008440
           delta_exec: 6874211      delta: 1008440
task_tick_fair

=> pick_next_task_fair D=7040772 J=4302974608
    pick_next_entity
    CFS_RQ: /
           nr_running: 2 load.weight: 2048 min_vruntime: 337102568062
           runnable_load_avg: 2046 blocked_load_avg: 0
    se:      first tsexperiment/6305 SCHED_NORMAL
           load.weight: 1024 exec_start: RQ+6874211 vruntime: MIN+0
    se:      rb: tsexperiment/6305 SCHED_NORMAL
           load.weight: 1024 exec_start: RQ+6874211 vruntime: MIN+0
    se:      rb-r: tsexperiment/6304 SCHED_NORMAL
           load.weight: 1024 exec_start: RQ+-160403 vruntime: MIN+1168843
pick_next_task_fair
    se:      sched tsexperiment/6305 SCHED_NORMAL

```

You can conduct these experiments on your own.

Virtual memory

Consider the following C program which will be translated into assembler:

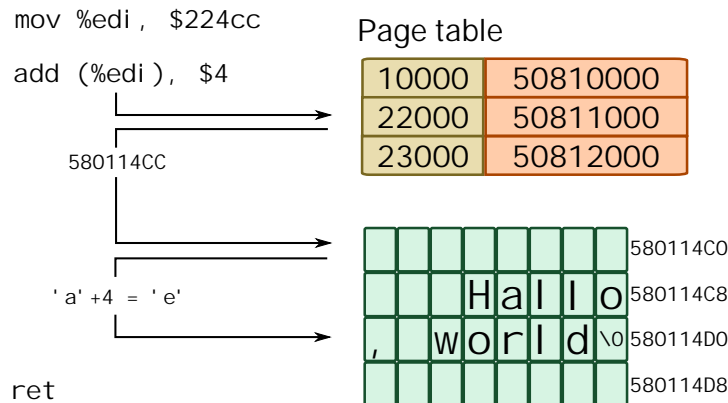
```

char msg[] = "Hallo, world";    // mov %edi, $224cc
msg[1] += 4;                    // add (%edi), $4

```

When single instance of that program is running, it will work as expected, message will become "Hello, world". But what will happen if two instances of program will be run simultaneously? Since compiler have used absolute addressing, second program may have been overwritten data of first instance of a program, making it "Hillo, world!" (actually, before that, program loader should load original message "Hallo, world" back). So multiprocessing creates two problems: same addresses of different processes shouldn't point to same physical memory cell and processes should be disallowed to write to memory that doesn't belong to them. *Virtual memory* is the answer to these problems.

Modern virtual memory mechanisms are based on *page addressing*: all physical memory is divided to a pages of a small size (4 kb in x86). Processes exist in a virtual address space where each subset of addresses, say [BASE; BASE+PAGESIZE), maps to a single page. List of such mappings is maintained as *page table*. Modern CPUs also provide support for *huge pages* (Linux) or *large pages* (Solaris) which may be megabytes or even gigabyte in size. Speaking of our previous example, kernel *binary format loader* will set up a virtual address space for our program, copying all data to new locations in physical memory:



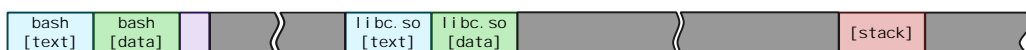
When second instance of a program will start, new process with separate address space will be created, thus making independent copy of process data, including "Hallo, world" message, but with same addresses. When process (actually its thread or task) is dispatched onto CPU, address of its page table is written to a special register (like CR3 on x86), so only it may access its data. All address translations are performed by *Memory Management Unit* in CPU and are transparent for it.

From the process point of view, pages are grouped into *segments* which constitute *address space*:

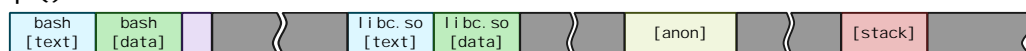
execve()



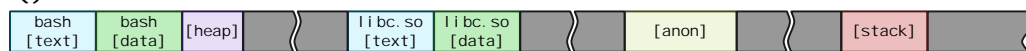
ld.so



mmap()



brk()



New address spaces are created as a result of `execve()` system call. When it is finished, new address space constitutes from four segments: *text* segment contains program code, *data* segment contains program data. Binary loader also creates two special segments: *heap* for dynamically allocated memory and *stack* for program stack. Process arguments and environment are also initially put onto stack. Then, kernel runs process interpreter `ld.so`, which actually a dynamic linker. That linker searches for libraries for a process such as standard C library `libc.so` and calls `mmap()` to load text and data sections of that libraries.

When you try to allocate memory using `malloc()`, standard C library may increase heap using `brk()` or `sbrk()` system call. Program may also use `mmap()` calls to map files into memory. If no file is passed to `mmap()` call, then it will create special memory segment called an *anonymous memory*. Such memory segment may be used for memory allocators, independent from main process heap.

You can check address space of a process with `pmap` program or by viewing `/proc/PID/mapping` file on Linux.

Let's for example see, how memory is dynamically allocated by calling `malloc()` with relatively large value. I used Python 2 `range(10000)` built-in which creates list with 10000 numbers.

SystemTap provides corresponding syscalls via tapset `vm`:

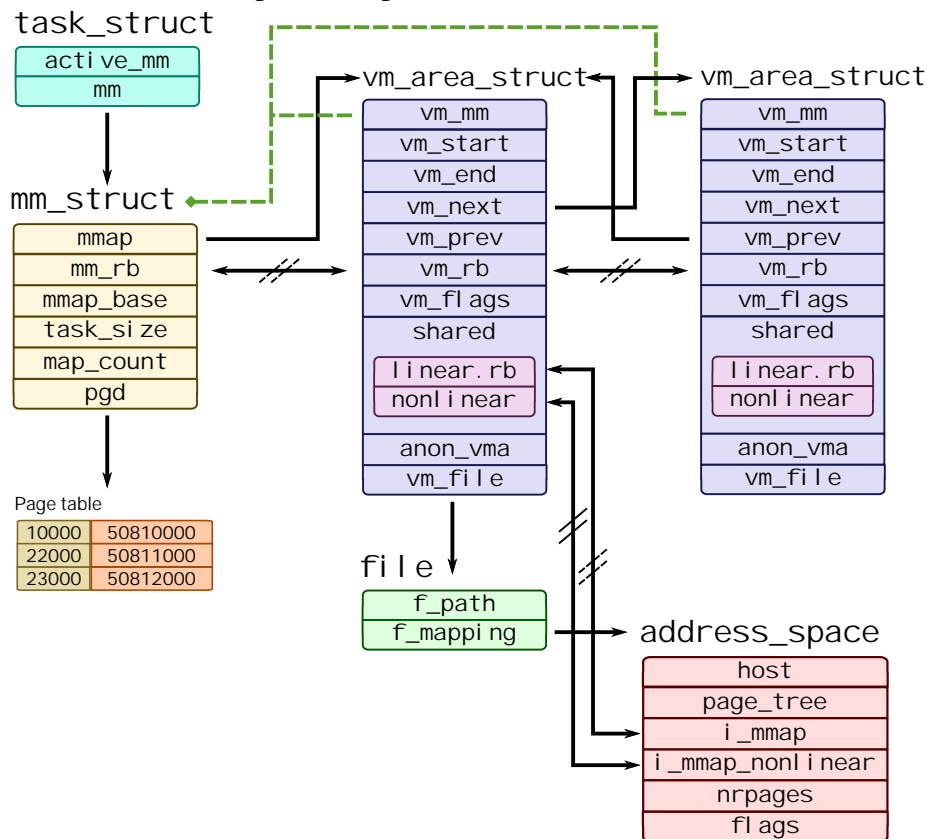
```
# stap -d $(which python) --ldd -e '
    probe vm.brk, vm.mmap, vm.munmap {
        printf("%8s %s/%d %p %d\n",
            name, execname(), pid(), address, length);
        print_ubacktrace();
    } -c 'python -c "range(10000)''
```

Solaris doesn't have such tapset, but these operations are performed using `as_map()` and `as_unmap()` kernel functions:

```
# dtrace -qn '
    as_map:entry, as_unmap:entry {
        printf("%8s %s/%d %p %d\n",
            probefunc, execname, pid, arg1, arg2);
        ustack();
    }'
# python -c "import time; range(10000); time.sleep(2)"
```

After running both of these scripts, you will see, that lot's of `brk()` calls are caused by `builtin_range()` function in Python.

Process address space is kept in `mm_struct` in Linux and in `as_t` structure in Solaris:



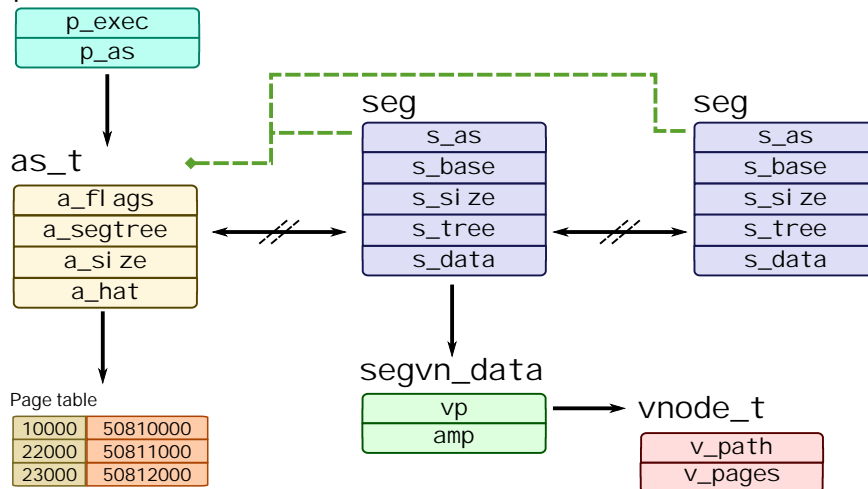
Each memory segment is represented by instance of `vm_area_struct` structure which has two addresses: `vm_start` which points to the beginning of a segment and `vm_end` which points to the end of the segment. Kernel maintains two lists of segments: linear double-linked list of segments (sorted by their addresses) starting with `mmap` pointer in `mm_struct` with `vm_next` and `vm_prev` pointers, another list is a red-black tree built with `mm_rb` as root and `vm_rb` as node.

Segments may be mapped files, so they have non-NULL value of `vm_file` pointing to a `file`. Each `file` has an `address_space` which contains all pages of a file in a `page_tree` in a `address_space` object. This object also references `host` inode of a file and all mappings corresponding to that file through linear and non-linear lists, thus making all mappings of a file shared. Another option for mapping is anonymous memory — its data is kept in `anon_vma` structure. Every segment has a `vm_mm` pointer which refers `mm_struct` to which it belongs.

`mm_struct` alone contains other useful information, such as base addresses of entire address space `mmap_base`, addresses of a stack, heap, data and text segments, etc. Linux also caches memory statistics for a process in `rss_stat` field of `mm_struct` which can be pretty-printed with `proc_mem*` functions in SystemTap:

```
# stap -e '
  probe vm.brk, vm.mmap {
    printf("%8s %d %s\n", name, pid(), proc_mem_string());
  } -c 'python -c "range(10000)''
```

In Solaris `as_t` structure accessible through `p_as` field of process and keeps all segments in AVL tree where `a_segtree` is a root node and `s_tree` is a nodes embedded to a segment:



Each segment has backward link to address space `s_as`, `s_base` as base address of a segment and `s_size` as its size. Solaris uses so-called *segment drivers* to distinguish one type of a segment to another, so it provides table of operations through `s_ops` field and private data through `s_data` field. One of the segment drivers is *segvn* driver which handles mmapped segments of memory both from files and anonymous, which keep their data in `segvn_data` structure which holds two pointers: `vp` to file's vnode and `amp` for a map of anonymous memory.

Some memory will be consumed by a process indirectly. For example, when application transfers a packet through the network or writes data to a file on `/tmp` filesystem, data is buffered by Kernel, but that memory is not mapped to a process. To do so, Kernel uses various in-kernel memory allocators and maintains *kernel address space*.

Page fault

As we mentioned before, when program accesses memory, memory management unit takes address, finds an entry in a page table and gets physical address. That entry, however, may not exist — in that case CPU will raise an exception called a *page fault*. There are three types of page faults that may happen:

- *Minor* page fault occurs when page table entry should exist, but corresponding page wasn't allocated or page table entry wasn't created. For example, Linux and Solaris do not allocate mmapped pages immediately, but wait until first page access which causes minor page faults.
- *Major* page fault requires reading from disk. It may be caused by accessing memory-mapped file or when process memory was paged-out onto disk swap.
- *Invalid* page fault occur when application access memory at invalid address or when segment permissions forbid such access (for example writing into text segment, which is usually disallowed). In this case operating system may raise `SI GSEGV` signal. A special case of invalid page faults is *copy-on-write* fault which happens when forked process tries to write to a parent's memory. In this case, OS copies page and sets up a new mapping for forked process.

Page faults are considered harmful because they interrupt normal process execution, so there are various system calls such as `mlock()`, `madvise()` which allow to flag memory

areas to reduce memory faults. I.e. `mlock()` should guarantee page allocation, so minor fault won't occur for that memory area. If page faults occurs in a kernel address space, it will lead to kernel oops or panic.

You can trace page faults in Linux by attaching to `vm.pagefault.return` probe. It has `fault_type` variable which is a bitmask of a fault type. RedHat-like kernels also have `mm_anon_*` and `mm_filemap_*` probes. Page faults is also presented to a perf subsystem. In Solaris all virtual memory events including page faults are available in `vminfo` provider:

Type	DTrace	SystemTap
<i>Any</i>	<code>vminfo::as_fault</code>	<code>perf.sw.page_faults</code>
<i>Minor</i>		<code>perf.sw.page_faults_min</code>
<i>Major</i>	<code>vminfo::maj_fault</code> usually followed by <code>vminfo::pgin</code>	<code>perf.sw.page_faults_maj</code>
<i>Invalid</i>	<code>vminfo::cow_fault</code> for copy-on-write faults <code>vminfo::prot_fault</code> for invalid permissions or address	See notes below

Note

Linux doesn't have distinct probe for invalid page fault -- these situations are handled by architecture-specific function `do_page_fault()`. They are handled by family of `bad_area*`() functions on x86 architecture, so you can attach to them:

```
# stap -e '
    probe kernel.function("bad_area*") {
        printf("%s pid: %d error_code: %d addr: %p\n",
            probefunc(), pid(), $error_code, $address);
    }
```

Note

By default perf probe fires after multiple events, because it is sampler. To alter that behaviour, you should use `.sample(1)` which will fire on any event, but that requires to pass perf probes in raw form, i.e.:

```
perf.type(1).config(2).sample(1)
```

You can check actual values for type and config in `/usr/share/systemtap/linux/perf.stp` tapset. See also: [perf syntax](#) in Profiling section of this book.

Page fault is handled by `as_fault()` function in Solaris:

```
faultcode_t as_fault(struct hat *hat, struct as *as,
                    caddr_t addr, size_t size,
                    enum fault_type type, enum seg_rw rw);
```

This function calls `as_segat` to determine segment to which fault address belongs, providing `struct seg*` as a return value. When no segment may be found due to invalid fault, it returns NULL.

Let's write simple tracer for these two functions. It also prints `amp` address and path of `vnode` for `segvn` driver:

Source file: [scripts/dtrace/pagefault.d](#)

Here is an example of a page fault traced by this script:

```
as_fault pid: 3408 as: 30003d2dd00
  addr: d2000 size: 1 flags: F_PROT|S_WRITE
 [c0000:d4000] rwxu
  vn: /usr/bin/bash
  amp: 30008ae4f78:0
```


It was most likely a data segment of a `/usr/bin/bash` binary (because it has rights `rwXu`), while type of the fault is `F_PROT` which means invalid access right which makes it copy-on-write fault.

If you run a script for a process which allocates and initializes large amount of memory, you'll see lots of minor faults (identifiable by `F_INVAL`) with addresses which are go sequentially:

```
as_fault pid: 987 as: fffffc10008fc6110
  addr: 81f8000 size: 1 flags: F_INVAL|S_WRITE
  [8062000:a782000] rw-u
as_fault pid: 987 as: fffffc10008fc6110
  addr: 81f9000 size: 1 flags: F_INVAL|S_WRITE
  [8062000:a782000] rw-u
as_fault pid: 987 as: fffffc10008fc6110
  addr: 81fa000 size: 1 flags: F_INVAL|S_WRITE
  [8062000:a782000] rw-u
```

Like we mentioned before, when application allocates memory, pages are not necessarily created. So when process touches that memory first time, page fault occurs and actual page allocation is performed.

Similarly, all pagefaults are handled by `handle_mm_fault()` function in Linux:

```
int handle_mm_fault(struct mm_struct *mm,
                    struct vm_area_struct *vma,
                    unsigned long address, unsigned int flags);
```

SystemTap provides a wrapper for it: `vm.pagefault` which we will use to write pagefault tracer script for Linux:

Source file: <scripts/stap/pagefault.stp>

Here is an example of its output:

```
vm.pagefault pid: 1247 mm: 0xdf8bcc80
  addr: 0xb7703000 flags: WRITE
  VMA [0xb7703000:0xb7709000]
  prot: rw-- may: rwx- flags: VM_ACCOUNT
  amp: 0xdc62ca54
  => pid: 1247 pf: MINOR
```

Warning

`vm_flags` are not stable and change from version to version. This script contains values according to CentOS 7. Check `include/linux/mm.h` for details.

Kernel allocator

Virtual memory is distributed between applications and kernel by a subsystem which called *kernel allocator*. It may be used both for applications and for internal kernel buffers such as ethernet packets, block input-output buffers, etc.

Lower layer of the kernel allocator is a *page allocator*. It maintains lists of *free pages* which are immediately available to consumers, *cache pages* which are cached filesystem data and may be easily evicted and *used pages* that has to be reclaimed thus being writing on disk swap device. Page allocation is performed by `page_create_vma()` function in Solaris which provides `page-get` and `page-get-page` static probes:

```
# dtrace -qn '
  page-get* {
```

```

    printf("PAGE lgrp: %p mnode: %d bin: %x flags: %x\n",
           arg0, arg1, arg2, arg3);
}'

```

Warning

Solaris 11.1 introduced new allocator infrastructure called VM2. Information about it is not publicly available, so it is out of scope of our book.

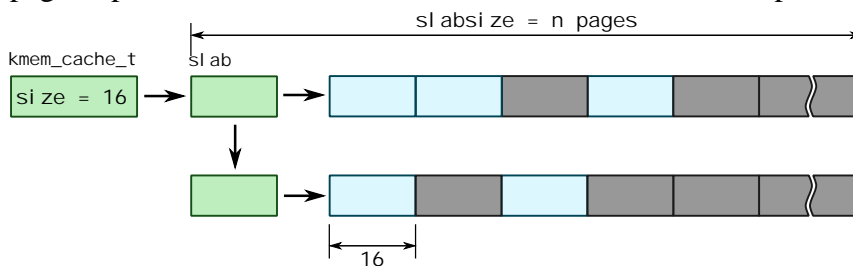
Linux page allocator interface consists of `alloc_pages*()` family of functions and `__get_free_pages()` helper. They have `mm_page_alloc` tracepoint which allows us to trace it:

```

# stap -e '
    probe kernel.trace("mm_page_alloc") {
        printf("PAGE %p order: %x flags: %x migrate: %d\n",
               $page, $order, $gfp_flags, $migratetype);
    }'

```

For most kernel objects granularity of a single page (4 or 8 kilobytes usually) is too high, because most structures have varying size. On the other hand, implementing a classical heap allocator is not very effective considering the fact, that kernel performs many allocations for an object of same size. To solve that problem, a *SLAB allocator* (which we sometimes will refer to as *kmem allocator*) was implemented in Solaris. SLAB allocator takes one or more pages, splits it into a buffers of a smaller sizes as shown on picture:



Modern SLAB allocators may have various enhancements like per-cpu slabs, SLUB allocator in Linux. Moreover, cache object is not necessarily created in SLAB allocators — objects of generic sizes may be allocated through function like `kmallocc()` in Linux or `kmem_alloc` in Solaris which will pick cache based on a size, such as `size-32` cache in Linux or `kmem_magazine_32` in Solaris. You can check overall SLAB statistics with `/proc/slabinfo` file in Linux, `::kostat mdb` command in Solaris or by using `KStat`: `kstat -m unix -c kmem_cache`.

Here are list of the probes related to kernel allocator:

Object	Action	DTrace	SystemTap
Block of an unspecified size	alloc	fbt::kmem_alloc:entry and fbt::kmem_zalloc:entry <ul style="list-style-type: none"> arg0 — size of the block arg1 — flags 	vm.kmallocc and vm.kmallocc_node <ul style="list-style-type: none"> caller_function — address of caller function bytes_req — requested amount of bytes bytes_alloc — size of allocated buffer gfp_flags and gfp_flags_str — allocation flags ptr — pointer to an allocated block

Object	Action	DTrace	SystemTap
Block of an unspecified size	free	fbt::kmem_free:entry <ul style="list-style-type: none"> arg0 — pointer to the block arg1 — size of the block 	vm.kfree <ul style="list-style-type: none"> caller_function — address of caller function ptr — pointer to an allocated block
Block from pre-defined cache	alloc	fbt::kmem_cache_alloc:entry <ul style="list-style-type: none"> arg0 — pointer to kmem_cache_t arg1 — flags 	vm.kmem_cache_alloc and vm.kmem_cache_alloc_node Same params as vm.kmalloc
Block from pre-defined cache	free	fbt::kmem_cache_free:entry <ul style="list-style-type: none"> arg0 — pointer to kmem_cache_t arg1 — pointer to a buffer 	vm.kmem_cache_free Same params as vm.kfree

Note that SystemTap probes are based on a tracepoints and provided by `vm tapset`.

On the other hand, when kernel needs to perform large allocations which are performed rarely, different subsystems are used: *vmalloc* in Linux, or *vmem* in Solaris (which is used by *kmem* SLAB allocator). Solaris also have segment drivers such as *segkmem*, *segkpm*, etc.

Exercise 4

Part 1

Implement scripts `pfstat.d` and `pfstat.stp` which will print count of pagefaults grouping by a mmaped file name (if it reachable). Print statistics once per second. Use `proc_starter` experiment from [exercise 3](#) to demonstrate it and try to explain results you are getting (you may need to include additional outputs for that).

Part 2

Implement scripts `kmemstatp.stp` and `kmemstat.d` which will gather stats on allocations and frees on per-cache basis for a *SLAB allocator*. Use an `file_opener` experiment from [exercise 1](#) to demonstrate your script and find a correlation between number of requests per-second generated by an experiment and cache allocations. What caches are primarily used while file is opened?

Virtual File System

One of the defining principles of Unix design was "*Everything is a file*". Files are organized into filesystems of different nature. Some like FAT are pretty simple, some like ZFS and btrfs are complex and incorporate volume manager into them. Some filesystems doesn't require locally attached storage — networked filesystems such as NFS and CIFS keep data on remote node, while special filesystems do not keep data at all and just representation of kernel structures: for example pipes are files on *pipefs* in Linux or *fifofs* in Solaris.

Despite this diversity of filesystem designs, they all share same API and conform same call semantics, so working with local or remote file is transparent for userspace application. To maintain these abstractions, Unix-like systems use **Virtual File System** (VFS) layer. Each *filesystem driver* exports table of supported operations to VFS and when system call is issued, VFS performs some pre-liminary actions, finds a filesystem-specific function in that table

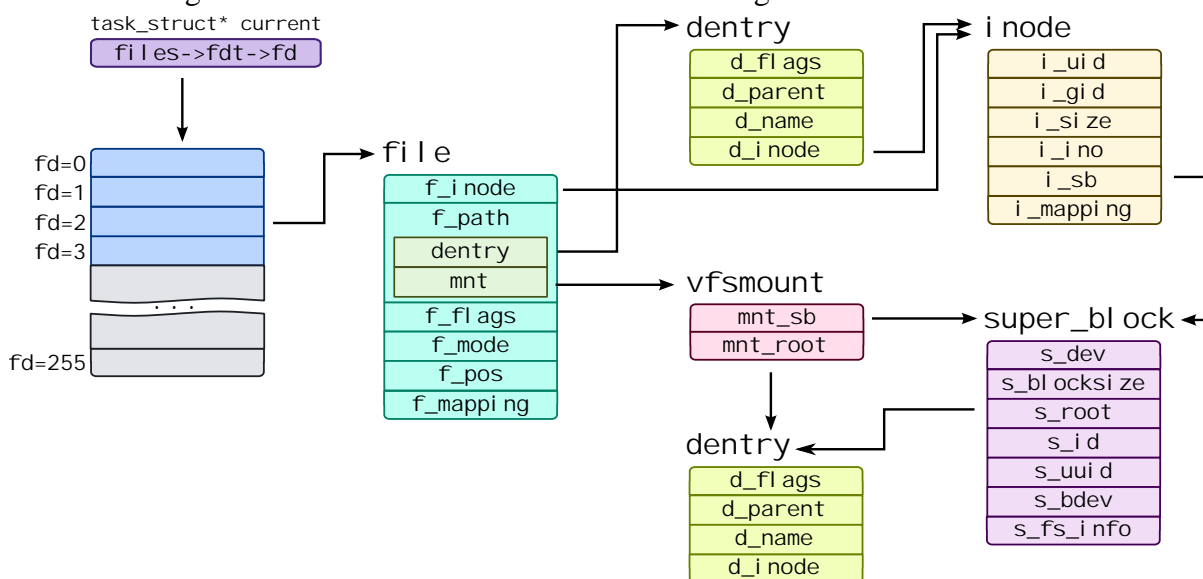
and calls it.

Each filesystem object has a corresponding data structure as shown in the following table:

Description	Solaris	Linux
Open file entry	uf_entry_t and file	file
Mounted filesystem	vfs_t	vfsmount -- for the mount point super_block -- for the filesystem
Table of filesystem operations	vfsops_t	super_operations
File or directory	vnode_t	dentry -- for entry in directory for file itself
Table of file/directory operations	vnodeops_t	file_operations -- for opened file inode_operations -- for inode operations address_space_operations -- for working data and page cache

Each process keeps table of opened files as an array of corresponding structures. When process opens a file, `open()` system call returns index in that array which is usually referred to as *file descriptor*. Following calls such as `read()` or `lseek()` will get this index as first argument, get corresponding entry from array, get `file` structure and use it in VFS calls.

Linux management structures are shown on the following schematic:



Open file table is indirectly accessible through `files` field of `task_struct`. We used 256 entries as an example, actual amount of entries may vary. Each entry in this table is an `file` object which contains information individual for a specific file descriptor such as open mode `f_mode` and position in file `f_pos`. For example, single process can open same file twice (one in `O_RDONLY` mode another in `O_RDWR` mode) -- in that case `f_mode` and `f_pos` for that file will differ, but `inode` and possibly `dentry` objects will be the same. Note that last 2 bits of `file` pointer are used internally by kernel code.

Each file is identifiable by two objects: `inode` represents service information for file itself like owner information in fields `i_uid` and `i_gid`, while `dentry` represents file in directory hierarchy (`dentry` is literally a directory entry). `d_parent` points to a parent `dentry` -- a `dentry` of directory where file is located, `d_name` is a `qstr` structure which keeps name of the file or directory (to get it use `d_name` function in `SystemTap`).

dentry and inode identify a file within filesystem, but systems have multiple filesystems mounted at different locations. That "location" is referred to as *mountpoint* and tracked through `vfsmount` structure in Linux which has `mnt_root` field which points to a directory which acts as mountpoint. Each filesystem has corresponding `super_block` object which has `s_bdev` pointer which points to a block device where filesystem data resides, `s_blocksize` for a block size within filesystem. Short device name is kept in `s_id` field, while unique id of filesystem is saved into `s_uuid` field of super block.

Note the `i_mapping` and `f_mapping` fields. They point to `address_space` structures which we have been discussed in section [Virtual Memory](#).

Let's get information on a file used in `read()` system call:

```
stap -e '
    probe syscall.read {
        file = @cast(task_current(), "task_struct")->
            files->fdt->fd[fd] - 3;
        if(!file)
            next;
        dentry = @cast(file, "file")->f_path->dentry;
        inode = @cast(dentry, "dentry")->d_inode;

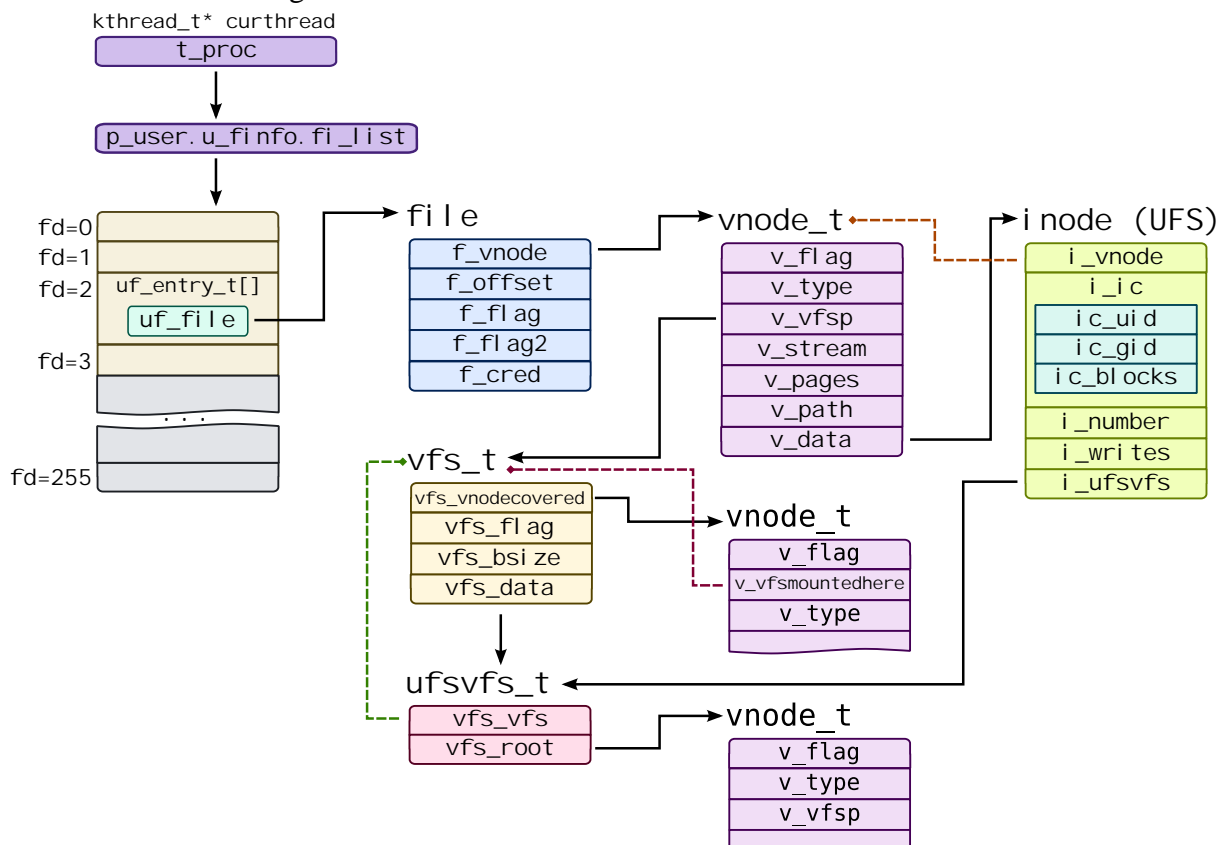
        printf("READ %d: file '%s' of size '%d' on device %s\n",
            fd, d_name(dentry), @cast(inode, "inode")->i_size,
            kernel_string(@cast(inode, "inode")->i_sb->s_id));
    }' -c 'cat /etc/passwd > /dev/null'
```

You may use `task_dentry_path()` function from `dentry tapset` instead of `d_name()` to get full path of opened file.

Warning

`fdt` array is protected through special RCU lock, so we should lock it before accessing it like [pfiles.stp](#) authors do. We have omitted that part in purpose of simplicity.

Solaris structures organization is much more clear:



Like Linux, each process keep an array of `uf_entry_t` entries while entry in this array points to an open file through `uf_file` pointer. Each file on filesystem is represented by `vnode_t` structure (literally, *node on virtual file system*). When file is opened, Solaris creates new `file` object and saves open file mode in flag fields `f_flag` and `f_flag2`, current file position in `f_offset` and pointer to a `vnode_t` in `f_vnode`.

`vnode_t` caches absolute path to a file in `v_path` field. Type of vnode is saved in `v_type` field: it could be `VREG` for regular files, `VDIR` for directories or `VFIFO` for pipes. VFS will keep `v_stream` pointing to a stream corresponding to FIFO for pipes, and list of pages `v_pages` for vnodes that actually keep data. Each filesystem may save its private data in `v_data` field. For UFS, for example, it is `i_node` structure (UDF also uses different `i_node` structure, so we named it `i_node (UFS)` to distinguish them). UFS keeps id of inode in `i_number` field, number of outstanding writes in `i_writes` and `i_ic` field which is physical representation of inode on disk, including uid and gid of owner, size of file, pointers to blocks, etc.

Like in case of vnode, Solaris keeps representation of filesystem in two structures: generic filesystem information like block size `vfs_bsize` is kept in `vfs_t` structure, while filesystem-specific information is kept in filesystem structure like `ufsvfs_t` for UFS. First structure to specific structure through `vfs_data` pointer. `vfs_t` refers to its mount point (which is a vnode) through `vfs_vnodecovered` field, while it refers to filesystem object through `v_vfspmountedhere` field.

DTrace provides array-translator `fds` for accessing file information through file descriptor — it is an array of `fileinfo_t` structures:

```
# dtrace -q -n '
syscall::read:entry {
    printf("READ %d: file '%s' on filesystem '%s'\n",
```

```

        arg0, fds[arg0].fi_name, fds[arg0].fi_mount);
}' -c 'cat /etc/passwd > /dev/null'
```

However, if you need to access `vnode_t` structure directly, you may use schematic above:

```

# dtrace -q -n '
syscall::read:entry {
    this->fi_list = curthread->t_procp->p_user.u_finfo.fi_list;
    this->vn = this->fi_list[arg0].uf_file->f_vnode;
    this->mntpt = this->vn->v_vfsp->vfs_vnodecovered;

    printf("READ %d: file '%s' on filesystem '%s'\n",
        arg0, stringof(this->vn->v_path),
        (this->mntpt)
        ? stringof(this->mntpt->v_path)
        : "/");
}' -c 'cat /etc/passwd'
```

Note that root filesystem have NULL `vfs_vnodecovered`, because there is no upper-layer filesystem on which it mounted.

Solaris provides stable set of probes which are tracing VFS through `fsinfo` provider. It provides `vnode` information as `fileinfo_t` structures just like `fds` array:

```

# dtrace -n '
fsinfo::mkdir {
    trace(args[0]->fi_pathname);
    trace(args[0]->fi_mount);
}' -c 'mkdir /tmp/test2'
```

Note that DTrace prints "unknown" for `fi_pathname` because when `mkdir` probe fires, `v_path` is not filled yet.

VFS interface consists of `fop_*` functions like `fop_mkdir` which is callable through macro `VOP_MKDIR` and, on the other side, call `vop_mkdir` hook implemented by filesystem through `vnodeops_t` table. So to trace raw VFS operations you may attach probes directly to that `fop_*` functions:

```

# dtrace -n '
fop_mkdir:entry {
    trace(stringof(args[1]));
}' -c 'mkdir /tmp/test1'
```

Now string name should be correctly printed.

There is no unified way to trace VFS in **Linux**. You can use `vfs_*` functions the same way you did with `fop_*`, but not all filesystem operations are implemented with them:

```

# stap -e '
probe kernel.function("vfs_mkdir") {
    println(d_name($entry));
}' -c 'mkdir /tmp/test4'
```

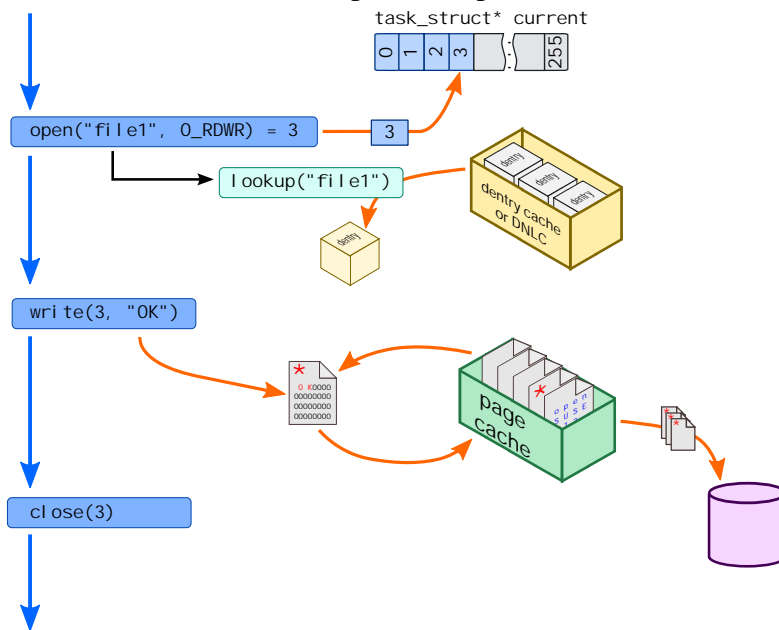
You may however use *inotify* subsystem to track filesystem operations (if `CONFIG_FSNOTIFY` is set in kernel's configuration):

```

# stap -e '
probe kernel.function("fsnotify") {
    if(!($mask == 0x40000100))
        next;
    println(kernel_string2($file_name, "???"));
}' -c 'mkdir /tmp/test3'
```

In this example `0x40000100` bitmask consists of flags `FS_CREATE` and `FS_ISDIR`.

Now let's see how VFS operations performed on files:



Application uses `open()` system call to open file. At this moment, new `file` object is created and free entry in open files table is filled with a pointer to that object. Kernel, however needs to find corresponding `vnode/dentry` object — it will also need to check some preliminary checks here. I.e. if `uid` of opening process is not equal to `i_uid` provided by operating system and file mode is `0600`, access should be forbidden.

To perform such mapping between file name passed to `open()` system call and `dentry` object, kernel performs a kind of *lookup* call which searches needed file over directory and returns object. Such operation may be slow (i.e. for file `/path/to/file` it needs `readdir` path than do the same with `to`, and only then seek for file `file`), so operating systems implement caches of such mappings. They are called *dentry cache* in Linux and *Directory Name Lookup Cache* in Solaris.

In Solaris top-level function that performs lookup called `lookupnvp()` (literally, lookup `vnode` pointer by path name). It calls `fop_lookup()` which will call filesystem driver. Most filesystems however will seek needed path name in DNLC cache, by doing `dnlc_lookup()`:

```
# dtrace -n '
  lookupnvp:entry /execname == "cat"/ {
    trace(stringof(args[0]->pn_path));
  }
  fop_lookup:entry /execname == "cat"/ {
    trace(stringof(arg1));
  }
  dnlc_lookup:entry /execname == "cat"/ {
    trace(stringof(args[0]->v_path)); trace(stringof(arg1));
  } -c 'cat /etc/passwd'
```

Linux uses unified system for caching file names called *Directory Entry Cache* or simply, *dentry cache*. When file is opened, one of `d_lookup()` functions are called:

```
# stap -e '
  probe kernel.function("__d_lookup*") {
    if(execname() != "cat") next;
    println(kernel_string($name->name));
  } -c 'cat /etc/passwd > /dev/null'
```


Now, when file is opened, we can read or write its contents. All file data is located on disk (in case of disk-based file systems), but translating every file operation into block operation is expensive, so operating system maintains *page cache*. When data is read from file, it is read from disk to corresponding page and then requested chunk is copied to userspace buffer, so subsequent reads to that file won't need any disk operations — it would be performed on *page cache*. When data is written onto file, corresponding page is updated and page is marked as dirty (red asterisk on image).

At the unspecified moment of time, page writing daemon which is relocated in kernel scans page cache for *dirty pages* and writes them back to disk. Note that `mmap()` operation in this case will simply map pages from page cache to process address space. Not all filesystems use page cache. ZFS, for example, uses its own caching mechanism called *Adaptive Replacement Cache* or ARC which is built on top of kmem allocator.

Let's see how `read()` system call is performed in detail:

Action	Solaris	Linux
Application initiates file reading using system call	<code>read()</code>	<code>sys_read()</code>
Call is passed to VFS stack top layer	<code>fop_read()</code>	<code>vfs_read()</code>
Call is passed to filesystem driver	<code>v_ops->vop_read()</code>	<code>file->f_op->read()</code> or <code>do_sync_read()</code> or <code>new_sync_read()</code>
If file is opened in direct input output mode, appropriate function is called and data is returned	I.e. <code>ufs_directio_read()</code>	<code>a_ops->direct_io</code>
If page is found in page cache, data is returned	<code>vpm_data_copy()</code> or <code>segmap_getmap_fit()</code>	<code>file_get_page()</code>
If page was not found in page cache, it is read from filesystem	<code>v_ops->vop_getpage()</code>	<code>a_ops->readpage()</code>
VFS stack creates block input-output request	<code>bdev_strategy()</code>	<code>submit_bio()</code>

Warning

This table is very simplistic and doesn't cover many filesystem types like non-disk or journalling filesystems.

We used names `v_ops` for table of vnode operations in Solaris, `f_op` for file_operations and `a_ops` for address_space_operations in Linux. Note that in Linux filesystems usually implement calls like `aio_read` or `read_iter` while read operation calls function like `new_sync_read()` which converts semantics of `read()` call to semantics of `f_op->read_iter()` call. Such "generic" functions are available in generic and `vfs` tapsets.

Block Input-Output

When request is handled by Virtual File System, and if it needs to be handled by underlying block device, VFS creates a request to Block Input-Output subsystem. Operating system in this case either fetches new page from a disk to a page cache or writes dirty page onto disk. Disks are usually referred to as *block devices* because you can access them by using blocks of fixed size: 512 bytes which is disk sector (not to mention disks with advanced format or SSDs). On the other hand, *character devices* like terminal emulator pass data byte by byte while *network devices* might have any length of network packet.

BIO top layer is traceable through `io` provider in DTrace:

```
# dtrace -qn '
  io:::start
  /args[0]->b_flags B_READ/ {
    printf("io dev: %s file: %s blkno: %u count: %d \n",
      args[1]->dev_pathname, args[2]->fi_pathname,
      args[0]->b_lblkno, args[0]->b_bcount);
  } -c "dd if=/dev/dsk/c2t0d1p0 of=/dev/null count=10"
```

If you check function name of that probe, you may see that it is handled by `bdev_strategy()` kernel function which has only one argument of type `struct buf`. That buffer represents a single request to a block subsystem and passed as `arg0` to `io:::start` probe and then translated to a `bufinfo_t` structure which is considered stable. DTrace also supplies information about block device and file name in `args[1]` and `args[2]`.

Linux has similar architecture: it has `struct bio` which represents single request to block subsystem and `generic_make_request()` function (which, however, has alternatives) which passes `bio` structure to device queues. SystemTap tapset `io_block` provides access to BIO probes:

```
# stap -e '
  probe io_block.request {
    if(bio_rw_num(rw) != BIO_READ)
      next;
    printf("io dev: %s inode: %d blkno: %u count: %d \n",
      devname, ino, sector, size);
  } -c "dd if=/dev/sda of=/dev/null count=10"
```

In these examples we have traced only read requests.

Here are description of `buf` structure from Solaris and `bio` structure from Linux:

Field description	<code>bufinfo_t</code> translator or <code>struct buf</code>	<code>struct bio</code>
Request flags	<code>b_flags</code>	<code>bi_flags</code>
Read or write	flags <code>B_WRITE</code> , <code>B_READ</code> in <code>b_flags</code>	<code>bi_rw</code> , see also functions <code>bio_rw_num()</code> and <code>bio_rw_str()</code>
Number of bytes	<code>b_bcount</code>	<code>bi_size</code>
Id of block	<code>b_blkno</code> , <code>b_lblkno</code>	<code>bi_sector</code>
Request finish callback	<code>b_iDONE</code>	<code>bi_end_io</code>
Device identifiers	<code>b_edev</code> , <code>b_dip</code>	<code>bi_bdev</code>
Pointer to data	<code>b_addr</code> or <code>b_pages</code> (only in <code>buf</code> when <code>B_PAGEIO</code> flag is set)	See note below
Pointer to file descriptor	<code>b_file</code> (only in <code>buf</code>)	

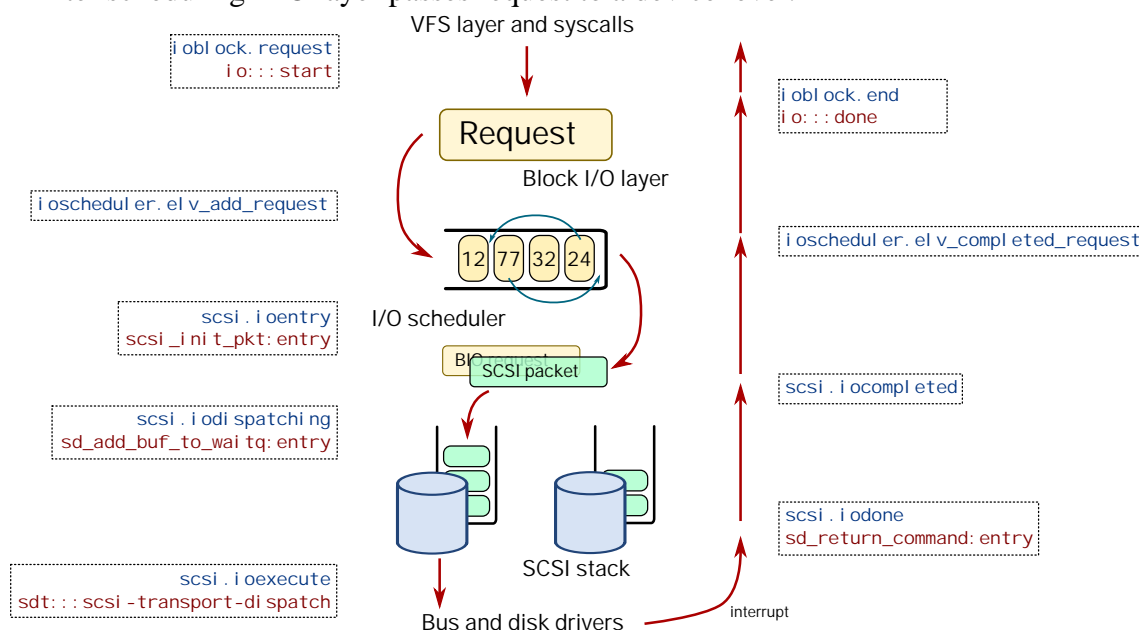
Note

struct `bio` in Linux contains table `bi_io_vec`, where each element contains pointer to a page `bv_page`, length of data `bv_len` and offset inside page `bv_offset`. Field `bi_vcnt` shows how many structures of that type is in vector while current index is kept in `bi_idx`.

Every `bio` structure can contain many files related to it (i.e. when I/O scheduler merges requests for adjacent pages). You can find file inode by accessing `bv_page` which points to a page-cache page, which will refer `inode` through its mapping.

When *BIO request* is created it is passed to scheduler which re-orders requests in a way which will require fewer movement of disk heads (this improves HDD access time). This subsystem plays important role in Linux which implements a lot of different schedulers, including *CFQ* (used by default in many cases), *Deadline* and *NOOP* (doesn't perform scheduling at all). They are traceable with `ioscheduler tapset`. Solaris doesn't have centralized place for that: ZFS uses *VDEV queue* mechanism, while the only unifying algorithm is *lift sorter* which is implemented in `sd_add_buf_to_wai tq()`.

After scheduling BIO layer passes request to a device level:



Both Solaris and Linux use SCSI protocol as unified way to represent low-level device access. SCSI devices can be stacked, i.e. with *device mapper* in Linux or *MPxIO* in Solaris, but we will have only single layer in our examples. In any case, this subsystem is called *SCSI stack*. All requests in SCSI stack are translated to SCSI packets (which can be translated to ATA commands or passed as is to SAS devices). SCSI packet is handled in a several steps:

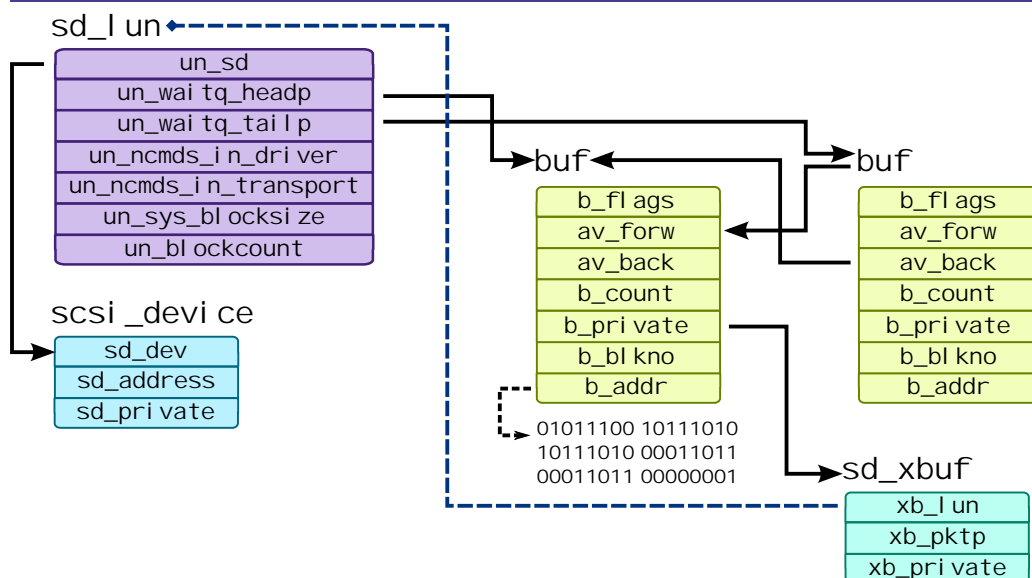
Action	Solaris	Linux
New instance of SCSI packet is created	<code>scsi_init_pkt()</code>	<code>scsi.ioentry</code>
SCSI packet is dispatched on queue	<code>sd_add_buf_to_wai tq()</code>	<code>scsi.iopatched</code>
SCSI packet is passed to low-level driver	<code>sdt::scsi-transport-dispatch</code> <code>scsi_transport()</code>	<code>scsi.ioexecute</code>
Low-level driver generates interrupt when SCSI packet is finished	<code>sd_return_command()</code>	<code>scsi.iocompleted</code> <code>scsi.iocompleted</code>

Warning

Probe `scsi . i oexecute` can be not fired for all SCSI packets: usually bus or disk driver puts request to internal queue and processes it independently from SCSI stack.

Note

We have used Solaris functions starting from `sd` prefix in this example. They are from `sd` driver which represents SCSI disk. There is also `ssd` driver which is used for FC disks — it is based on `sd` driver, but all functions in it are using `ssd` prefix, i.e. `ssd_return_command`.



In Solaris each SCSI *LUN* has a corresponding `sd lun` structure which keeps queue of buffers in doubly-linked list referenced by `un_waitq_headp` and `un_waitq_tailp` pointers. When new command is passed to SCSI stack, `un_ncmds_in_driver` is increased and when packet is dispatched to transport, `un_ncmds_in_transport` is increased. They are decreased when SCSI packet is discarded or when it was successfully processed and interrupt is fired to notify OS about that. SCSI stack uses `b_private` field to keep `sd_xbuf` structure that keeps reference to SCSI packet through `xb_pkt` pointer.

Following script traces block I/O layer and SCSI stack (`sd` driver in particular) in Solaris:

Source file: [scripts/dtrace/sdtrace.d](#)

It saves history of all I/O stages into a [speculation](#) which is committed when operation is finished. Note that due to the fact that speculation has one buffer per processor output may be garbled when interrupt was delivered to a processor other than processor that initiated request and `sdi ntr` is called on it.

Here is an example output for script:

```

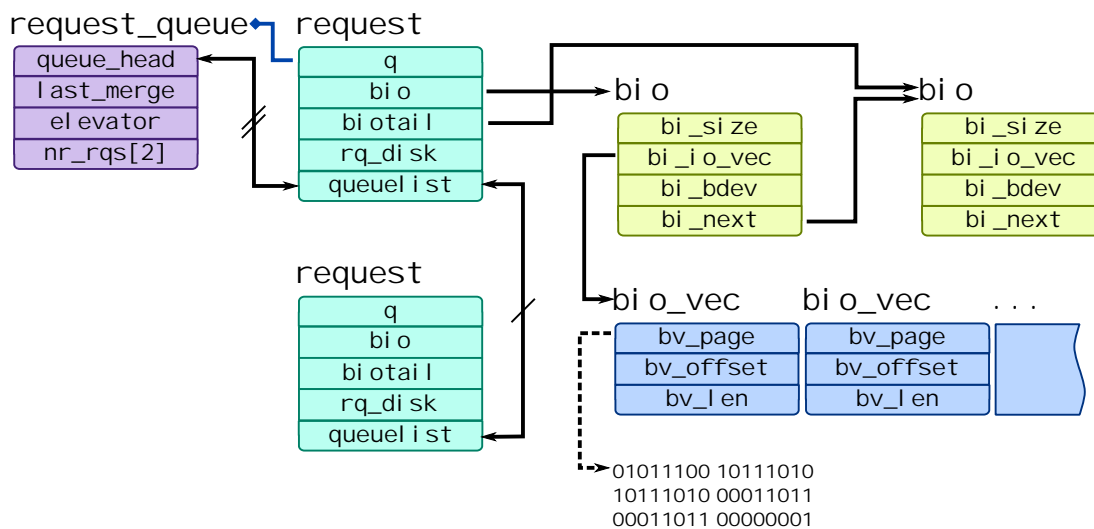
i o-start                                ffffc100040c4300 cpu0 2261
    PROC: 1215/1 dd
    BUF flags: R 200061 count: 512 blkno: 0 comp: 0x0
    DEV 208,192 sd
    FILE +-1
sd_add_buf_to_waitq                       ffffc100040c4300 cpu0 11549
scsi -transport-dispatch                  ffffc100040c4300 cpu0 18332
scsi_transport                            ffffc100040c4300 cpu0 21136
    SCSI PKT flags: 14000 state: 0 comp: sd`sdintr
sdi ntr                                   ffffc100040c4300 cpu0 565121
    SCSI PKT flags: 14000 state: 1f comp: sd`sdintr
i o-done                                   ffffc100040c4300 cpu0 597642
    
```

BUF flags: R 2200061 count: 512 blkno: 0 comp: 0x0

Each stage of request (marked bold) contains its name, address of buf pointer and time since request creation in nanoseconds. In our case largest difference is between `scsi_transport` and `sdintr` which is about half a second. It can be simply explained: actual I/O was performed between these stages, and it is slower than processor operations.

SCSI stack also uses callback mechanism to notify request initiators when it is finished. In our case lower-level driver had used `sdi_ntr` callback while `b_i odone` field wasn't filled. It is more likely that caller used `bi owai t()` routine to wait for request completion.

Like we said before, Linux has intermediate layer called a scheduler which can re-order requests. Due to that, BIO layer maintains generic layer of block device queues which are represented by struct `request_queue` which holds requests as struct `request` instances:



Each request may have multiple bio requests which are kept as linked list. New requests are submitted through `blk_queue_bio()` kernel function which will either create a new request using `get_request()` function for it or merge it with already existing request.

Here are example script for Linux which traces BIO layer and SCSI stack:

Source file: [scripts/stap/scsitrace.stp](#)

Script example outputs are shown below:

```
kernel.function("get_request@block/block-core.c:1074").return 0xffff880039ff1500 0xffff88
```

PROC: 16668/16674 tsexperiment

BUF flags: R f0000000000000001 count: 4096 bl kno: 779728 comp: €

end_b i o_bh_i o_sync

DEV 8, 0 I NO 0

```
i oschedul er. el v_add_request 0xffff880039ff1500 0xffff88001d8fea00 cpu0 15830
```

DEV 8, 0

```
scsi . i oentry 0xffff880039ff1500 0xffff88001d8fea00 cpu0 19847
```

```
scsi . iodi spatchi ng 0xffff880039ff1500 0xffff88001d8fea00 cpu0 25744
```

```
SCSI DEV 2: 0: 0: 0 RUNNING
```

```
SCSI PKT flags: 122c8000 comp: 0x0
```

```
scsi . iodi spatchi ng 0xffff880039ff1500 0xffff88001d8fea00 cpu0 29882
```

```
scsi . i odone 0xffff880039ff1500 0xffff88001d8fea00 cpu1 4368018
```

```
scsi . i o completed 0xffff880039ff1500 0xffff88001d8fea00 cpu0 4458073
```

```
i obl ock. end 0xffff880039ff1500 cpu0 1431980041275998676
```

Unlike Solaris, it shows to pointers for each stage: one for `bio` structure and one for `request`. Note that we didn't use `io_block.request` in our example. That is because we

wanted to distinguish merged and alone requests which can be done only with function boundary tracing.

Note

Linux 3.13 introduced a new mechanism for block device queues called *blk-mq* (*Multi-Queue Block IO*). It is not covered in this book.

Asynchronicity in kernel

Let's return to our scripts, `scsi trace.stp` and `sdtrace.d` which we had introduced in previous section, [Block Input-Output](#). We tested it with block I/O. If we create a filesystem on block device and try to write to it, we will see some interesting names of processes which actually perform the write. I.e. on Solaris:

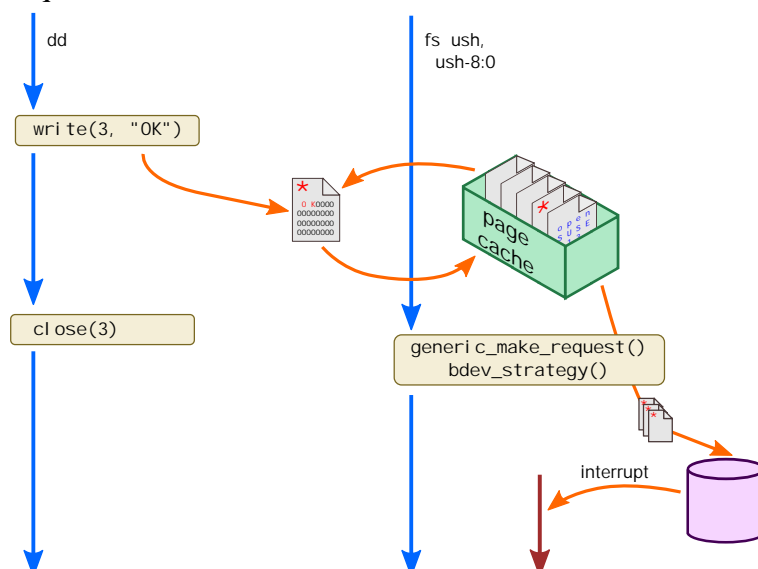
```
PROC: 5/15 zpool -t i ger
```

or on Linux:

```
PROC: 643/643 fl ush-8:0
```

This is internal system process which is not related to a process which is actually initiated `wri te()` call.

This is called an *asynchronicity*, a common kernel idiom when requests are kept in queues and being flushed to receiver at an unspecified moment of time (i.e. when device is ready, when we collected batch of adjacent requests, etc.). That *flush* is performed by separate kernel process (note how it is named in Linux) which has its own context. In our example `wri te()` system call won't actually start block input-output but updates page in page cache making it dirty (incorrect ZFS as it actually queues ZIO operation). Then *writeback* mechanism in Linux or *fsflush* daemon in Solaris is awoken. They walk dirty pages in page cache and write them back to stable storage. Finally, when disk finishes requested operation, bus driver will generate an interrupt which will create three independent contexts where request was handled:



This makes thread-local variables useless for such cases.

To overcome this situation we can use associative arrays, but instead of using process ID or thread ID we have to use something stable for a whole request execution. We can save PID using that key and then access it from interrupt probe by using same key and print it.

In our example it would be address of page in Linux:

```
# stap -e '
    global pids;

    probe module("ext4").function("ext4__write_end"),
        module("xfs").function("xfs_vm_writepage") {
        page = $page;
        pids[page] = pid();

        printf("I/O initiated pid: %d page: %p %d\n",
            pid(), $page, gettimeofday_ns());
    }

    probe ioblock.request {
        if($bio == 0 || $bio->bi_io_vec == 0)
            next;
        page = $bio->bi_io_vec[0]->bv_page;

        printf("I/O started pid: %d real pid: %d page: %p %d\n",
            pid(), pids[page], page, gettimeofday_ns());
    }
'
f
I/O initiated pid: 2975 page: 0xffffea00010d4d40 1376926650907810430
I/O initiated pid: 2975 page: 0xffffea00010d1888 1376926650908267664
I/O started pid: 665 real pid: 2975 page: 0xffffea00010d4d40 1376926681933631892
```

Note that despite the fact that process IDs in filesystem and block I/O probe are different, address of page structure is stable here.

Same works for Solaris — we can rely on `dbuf` pointer which represents a *dnode buffer*:

```
# dtrace -n '
    dbuf_dirty:entry {
        pids[(uintptr_t) arg0] = pid;
        printf("I/O initiated pid: %d dbuf: %p %d",
            pid, arg0, timestamp);
    }

    dbuf_write:entry {
        this->db = args[0]->dr_dbuf;
        printf("I/O started pid: %d real pid: %d dbuf: %p %d",
            pid, pids[(uintptr_t) this->db], this->db,
            timestamp);
    }
'
```

This technique is usually used *request extraction* — in ideal case we could observe all request handling from user clicking in a browser through network I/O, processing in web server, accessing database and, eventually, block I/O caused by that.

Modern kernels have low-level primitives for building such asynchronous mechanisms. We will discuss some of them later.

Exercise 4

Part 1

Create two scripts: `deblock.d` and `deblock.stp` which would demonstrate effects of unaligned input output for a synchronous writes. To do so, use aggregations to gather throughput (amount of data written) on VFS and BIO layers. Dump aggregations on terminal periodically using timer probes. Results should be grouped using name of disk device or/and mount point path.

Create filesystems to conduct experiment. For example, ext4 in CentOS:

```
# mkdir /tiger
# mkfs.ext4 /dev/sda
# mount /dev/sda /tiger
```

and ZFS in Solaris:

```
# zpool create tiger c3t0d0
```

Warning

`/dev/sda` and `c3t0d0` are the names in our lab environment. Replace them with correct ones. `/tiger` mount point, however will be used in experiment configuration files.

Note

You can use other filesystem types, however scripts from hints and solutions section will use ext4 and ZFS. Filesystem should also have prefetching mechanisms for part 2.

Run `deblock` experiment to evaluate your scripts. 1Mb file is created in this example, and blocks of random size (which is uniformly distributed value from 512 bytes to 16 kilobytes) are written at random offsets to it. Filesystem driver has to align block size to filesystem blocks which will induce additional overheads on block layer and even extra reads.

You can avoid effects of unaligned I/O by changing block size variator parameters like this:

```
# /opt/tload/bin/tsexperiment -e deblock/experiment.json run \
    -s workloads:fileio: params: block_size: randvar: min=4096 \
    -s workloads:fileio: params: block_size: randvar: max=4096
```

Block size becomes uniformly distributed in interval [4096;4096] which will be simply a constant value of 4 kilobytes. Re-run your script and see how workload characteristics are changed.

Part 2

In second part of this exercise we will evaluate filesystem prefetching or readahead mechanisms. It will significantly improve synchronous sequential read performance as operating systems will read blocks following by requested block asynchronously making it available in page cache when application will request it.

Write `readahead.stp` and `readahead.d` scripts to see which layer is responsible for prefetching. To do so, count number of operations on three levels: Virtual File System, Block Input-Output and SCSI stack. Grouping and output are the same as in part 1.

Note that test file will be already in page cache (or ARC cache in case of ZFS) after we create them, so we will need to flush it before running an experiment. The simplest way to do that is to unmount filesystem and mount it again. I.e in Linux:


```
# umount /tiger/ ; mount /dev/sda /tiger/
```

Entire ZFS pools have to be exported-imported to destroy ARC cache.

```
# zpool export tiger ; zpool import tiger
```

You may also use `drop_caches` tunable in Linux. This should be done before each experiment run.

SimpleIO module of TSLoad workload generator will immediately start experiment after writing the file. We should create that file manually before starting it:

```
# dd if=/dev/zero of=/tiger/READAHEAD count=40960
```

And set `overwrite` option to true in experiment configuration.

Use `readahead` experiment to demonstrate your script. You may change sequential operations to random by changing random generator type from sequential to linear congruential generator and see how effects of `readahead` is changed:

```
# /opt/tsload/bin/tsexperiment -e readahead/experiment.json run \
-s workloads:fileio:params:offset:randgen:class=lcg
```

Network stack

One of the largest kernel subsystem is a network stack. It is called a *stack* because it consists from multiple protocols where each of them works on top of the more primitive protocol. That hierarchy is defined by different models such as *OSI model* or *TCP/IP stack*. When user data is passed through network, it is encapsulated into packets of that protocols: when data is passed to a protocol driver it puts some service data to the packet header and tail so operating system on receiver host can recognize them and build original message even when some data was lost or order of packets had changed during transmission.

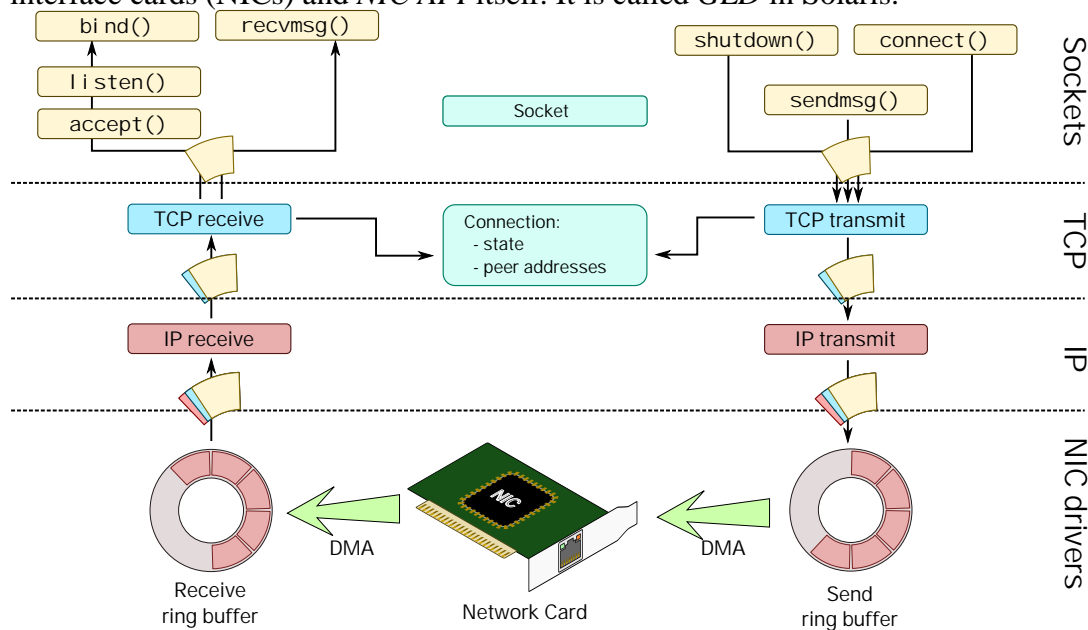
Each layer of network stack has its responsibilities so they are not of concern of higher-layer protocols. For example, IP allows to send datagrams through multiple routers and networks, can reassemble packets but doesn't guarantee reliability when some data is lost — it is implemented in TCP protocol. Both of them can only transmit raw data, encoding or compression is implemented on higher layer like HTTP.

Network subsystem (which transmits data between hosts) has a major difference over block input-output (which stores data): It is very sensitive to *latency*, so writing or reading data cannot be deferred. Due to that, sending and receiving is usually performed in the same thread context.

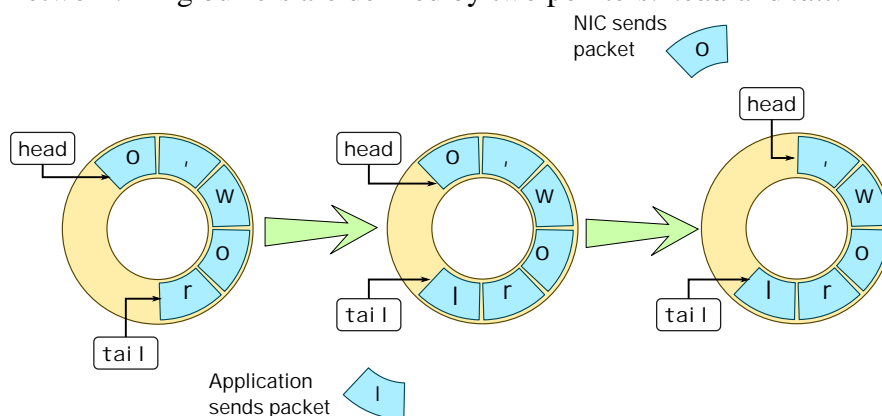
Network stack in Unix systems can be split into three generic layers:

- *Socket layer* which implements BSD sockets through series of system calls.
- Intermediate protocol drivers such as *ip*, *udp* and *tcp* and packet filters.

- Media Access Control (MAC) layer on the bottom which providing access to network interface cards (NICs) and *NIC API* itself. It is called *GLD* in Solaris.

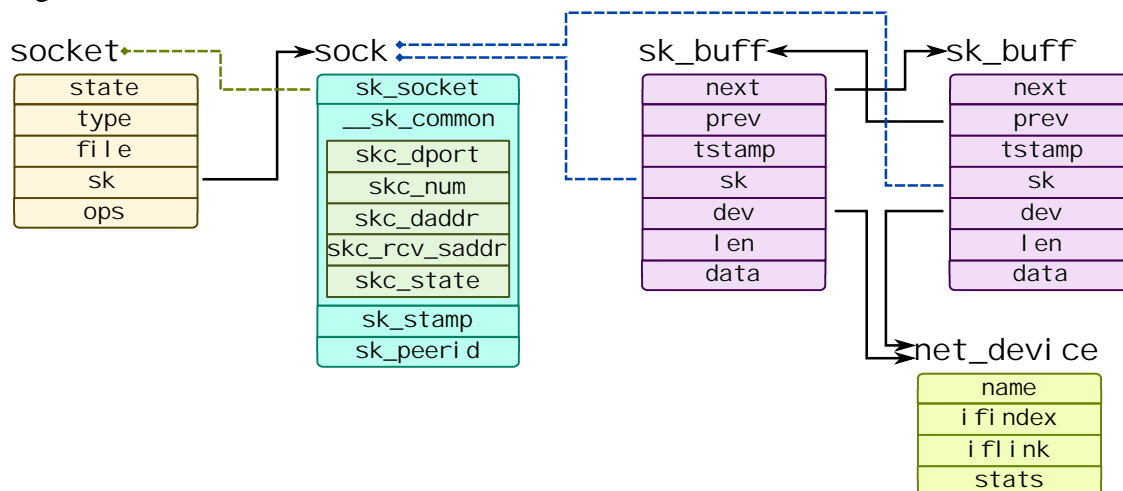


Network input-output can require transferring huge amounts of data, so it may be ineffective to explicitly send write commands for each packet. Instead of handling each packet individually, NIC and its driver maintain shared *ring buffer* where driver puts data while card uses *DMA* (direct memory access) mechanisms to read data and send it over network. Ring buffers are defined by two pointers: *head* and *tail*:



When driver wants to queue packet for transmission it puts it into memory area of designated ring buffer and updates *tail* pointer appropriately. When NIC transfers data over a network it will update *head* pointer.

Data structures are usually shared between stack layers. In Linux packets are represented by a generic `sk_buff` structure:



That structure keeps two pointers: head and data and includes offsets for protocol headers. Data length is kept in `len` field, time stamp of packet in `tstamp` field. `sk_buff` structures form a doubly-linked list through `next` and `prev` structures. They refer network device descriptor which is represented by `net_device` structure and a socket which is represented by pair of structures: `socket` which holds generic socket data including file pointer which points to VFS node (sockets in Linux and Solaris are managed by special filesystems) and `sock` which keeps more network-related data including local address which is kept in `skc_rcv_saddr` and `skc_num` and peer address in `skc_daddr` and `skc_dport` correspondingly.

Note that CPU byte order may differ from network byte order, so you should use conversion functions to work with addresses such as `ntohs`, `ntohl` or `ntohl` to convert to host byte order and `htons`, `htonl` and `htonl` for reverse conversions. They are provided both by SystemTap and DTrace and have same behaviour as their C ancestors.

Here are sample script for tracing message receiving in Linux 3.9:

```
# stap -e '
  probe kernel.function("tcp_v4_rcv") {
    printf("[%4s] %11s: %-5d -> %11s: %-5d len: %d\n",
      kernel_string($skb->dev->name),

      ip_ntop($skb->sk->__sk_common->skc_daddr),
      ntohs($skb->sk->__sk_common->skc_dport),

      ip_ntop($skb->sk->__sk_common->skc_rcv_saddr),
      $skb->sk->__sk_common->skc_num,

      $skb->len);
  }'
```

Earlier versions of Linux (2.6.32 in this example) use different structure called `inet_sock`:

```
# stap -e '
  probe kernel.function("tcp_v4_do_rcv") {
    printf("%11s: %-5d -> %11s: %-5d len: %d\n",
      ip_ntop(@cast($sk, "inet_sock")->daddr),
      ntohs(@cast($sk, "inet_sock")->dport),

      ip_ntop(@cast($sk, "inet_sock")->saddr),
```

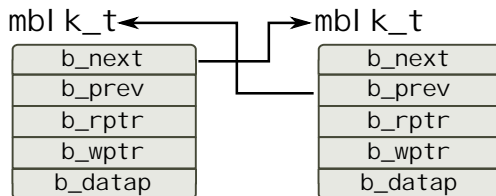
```

        ntohs(@cast($sk, "inet_sock")->sport),

        $skb->len);
}'

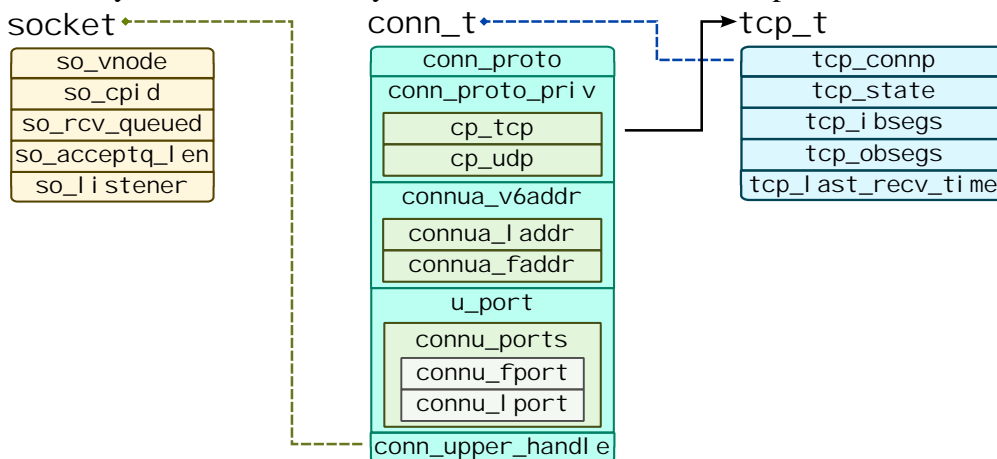
```

Solaris has derived *STREAMS* subsystem from System V which is intended to provide API for passing messages between multiple architectural layers which is perfectly fits to how network stack look like. Each message is represented by an `mbik_t` structure:



Consumer reads data referred by `b_rptr` pointer while producer puts it under `b_wptr` pointer if there is enough space in allocated buffer (it is referred by `b_datap`) or allocates a new message and sets up forward and backward pointers `b_next` and `b_prev` so these messages form a doubly-linked list.

Note that unlike `sk_buff` from Linux, these messages do not contain pointers to the management structure. Instead of doing that, functions pass pointer to them as a separate argument which is usually first argument of the function (`arg0` in DTrace): `mac_impl_t` for MAC layer, `ill_t` for IP layer and `conn_t` for TCP/UDP protocols:



Solaris wraps sockets into `sonode` structure which are handled by virtual file system called *sockfs*. `so_vnode` field in that structure points to VFS node. Like we mentioned before, TCP and UDP connection are managed by `conn_t` structure. It keeps addresses in `connua_laddr` and `connu_lport` fields for local address and uses `connua_faddr` and `connu_lport` for remote ports. Note that these names are different in Solaris 10.

Here are example DTrace script for tracing message receiving in Solaris 11:

```

# dtrace -n '
tcp_input_data:entry {
    this->conn = (conn_t*) arg0;
    this->mp = (mbik_t*) arg1;

    printf("%11s: %-5d -> %11s: %-5d len: %d\n",
        inet_ntoa((ipaddr_t*) (this->conn->connua_v6addr.
                                connua_faddr._S6_un._S6_u32[3])),
        ntohs(this->conn->u_port.connu_ports.connu_fport),

        inet_ntoa((ipaddr_t*) (this->conn->connua_v6addr.

```

```

                                connu_l_addr._S6_un._S6_u32[3])),
    ntohs(thi s->conn->u_port.connu_ports.connu_l_port),

    (thi s->mp->b_wptr - thi s->mp->b_rptr));
}'

```

Solaris 11 introduced new providers for tracing network: tcp, udp and ip. Here are probes that are provided by them and their siblings from Linux and SystemTap:

Action	DTrace	SystemTap
TCP		
Connection to remote node	tcp::connect-request tcp::connect-established tcp::connect-refused	kernel.function("tcp_v4_connect")
Accepting remote connection	tcp::accept-established tcp::accept-refused	kernel.function("tcp_v4_hnd_req")
Disconnecting	fbt::tcp_disconnect	tcp.disconnect
State change	tcp::state-change	-
Transmission	tcp::send	tcp.sendmsg
Receiving	tcp::receive	tcp.receive tcp.recvmsg
IP		
Transmission	ip::send	kernel.function("ip_output")
Receiving	ip::receive	kernel.function("ip_rcv")
Network device		
Transmission	mac_tx: entry, or function from NIC driver like e1000g_send: entry	netdev.transmit netdev.hard_transmit
Receiving	mac_rx_common: entry, or function from NIC driver like e1000g_receive: entry	netdev.rx

Sockets can be traced using syscall tracing. SystemTap provides special tapset socket for that.

Both Linux and Solaris provide various network statistics which are provided by SNMP and accessible through netstat -s command. Many events registered by these counters are implemented using mi b provider from DTrace or tcpmb, ipmb and linuxmb tapsets in SystemTap, but they do not have connection-specific data.

Synchronization primitives

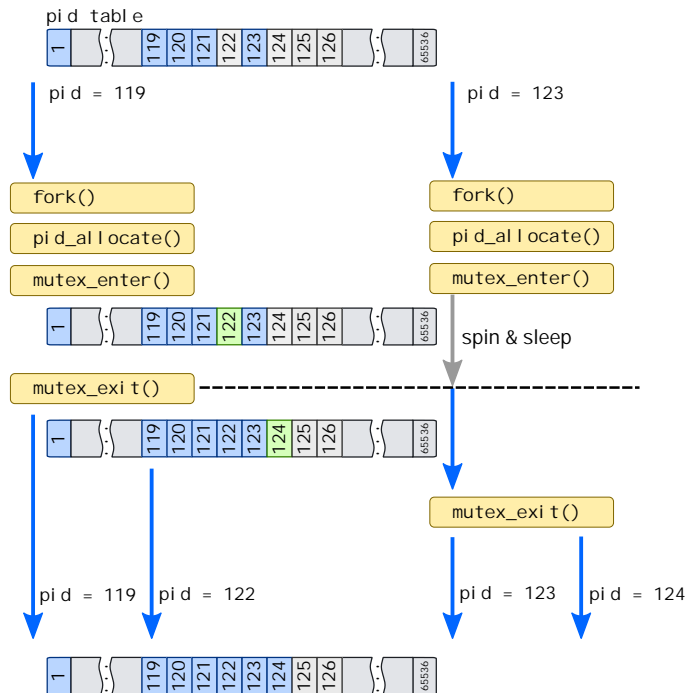
Locks

Modern operating system kernels are multi-threaded thus they allow parallel execution of service routines and userspace threads. However, they share some common pools of resources and objects for which these threads compete and may cause conflicts if two threads request access simultaneously. If operating system won't resolve such conflict, object data may be corrupted which may cause incorrect system behaviour and eventually, system panic.

Information

For example, we have two processes in a system with pids 119 and 123 which are simultaneously executed on different CPUs (for simplicity of the example, strictly speaking processes can compete for same resource even on uni-processor system if scheduler will perform context switch while process is accessing resource or object). So those processes are simultaneously called `fork()`, so operating system has to create clones of these processes and assign new pids to them. If we didn't provide mechanism that regulates accesses to pid table, they both acquire first free pid 122 which leads to creation two processes with equal process ids which makes no sense.

To prevent this, operating systems may use *mutex* (like it is done in Solaris) or combination of *spin-lock* and *atomic operation* (Linux). `mutex_enter()` call in Solaris guarantees that only one thread can be executed while holding mutex. Other threads will execute busy loop (which is called *spin*) or will be blocked on a sleep queue and will be removed from run queue when they call `mutex_enter()` for mutex that already held:



When process 119 acquires new process id, it leaves mutex by calling `mutex_exit()`. This function activates other process, 123, which may now access pid table, but it can't get process id 122 because process 123 is already sees changes made by process 119. So it takes next available pid which is 124.

Mechanisms that prevent such conflicts from happening like processes with equal pids in the example above are called *synchronization primitives*. They synchronize (even serialize) accesses to shared resources and objects, but their implementation is independent of nature of resource or object they are protecting.

Simplest primitive is an *atomic*. Atomics rely on processor ability to lock the system bus and prevent other processor accesses to the memory (i.e. with `lock` instruction prefix in x86 command set) for a single instruction thus guarantee that no other thread will perform another operation with the cell at the moment. They are widely used in Linux (but can be emulated on some architectures), and almost not used in Solaris.

Atomics allow only single machine instruction to be performed on data atomically. If more actions has to be done with guarantee that no other thread will intervene, *critical section* has to be implemented. First concern is how many threads are allowed in the critical section. *Mutexes* (mutual exclusion) allow only single thread, *semaphores*, which are generic variant of mutex, allow limited amount of threads, *read-write locks* allow multiple reader threads which do not change object but only single writer thread which exclusively modifies object data.

The second question is how to handle thread that failed competition for accessing synchronization object: it could either *spin* in busy loop or being put to a scheduler's *sleep* queue. Not all synchronization objects are suitable for sleeping: i.e. it couldn't be used in a interrupt context. Both approaches are also can be wasteful: spinning for a long time can occupy processor while dequeuing and enqueueing threads to a scheduler queue can be wasteful for short operations.

Linux prefers *spin locks*, but uses blocking mutexes in some places, Solaris uses universal mutex interface which provides both spinning and adaptive mutexes (adaptive mutex spins for short time and then goes to sleep phase). There are also sequential locks and Read-Copy-Upgrade synchronization policy in Linux which would be outside of our short review.

DANGER!

Linux and SystemTap doesn't provide probes for tracing locks (that are used in critical sections), and, moreover, critical sections may be used in modules generated by SystemTap. Due to that, some spinlock functions are blacklisted from tracing and require Guru mode to be enabled. We will provide name of the function for tracing locks, but do not recommend to use them in production tracing.

Action	DTrace	SystemTap
Adaptive locks (which support blocking)		
Acquire	lockstat::adaptive-acquire lockstat::adaptive-block lockstat::adaptive-spin	kernel.function("mutex_lock") kernel.function("debug_mutex_add_waiter")
Release	lockstat::adaptive-release	kernel.function("mutex_unlock") kernel.function("debug_mutex_unlock") kernel.function("debug_mutex_wake_waiter")
Spin locks		
Acquire	lockstat::spin-acquire lockstat::spin-spin lockstat::thread-spin	kernel.function("spin_lock") kernel.function("debug_spin_lock_before") kernel.function("debug_spin_lock_after")
Release	lockstat::spin-release	kernel.function("spin_unlock") kernel.function("debug_spin_unlock")
Read-write locks		
Acquire	lockstat::rw-acquire lockstat::rw-block	kernel.function("_raw_read_lock") kernel.function("_raw_write_lock") kernel.function("do_raw_read_lock") kernel.function("debug_write_lock_before") kernel.function("debug_write_lock_after")
Release	lockstat::rw-release	kernel.function("_raw_read_unlock") kernel.function("_raw_write_unlock") kernel.function("do_raw_read_unlock") kernel.function("debug_write_unlock")
Reader to writer promotion	lockstat::rw-upgrade	-
Writer to reader downgrade	lockstat::rw-downgrade	-

Note

Probes ending with `-spin` and `-block` in DTrace fire at the same time as `-acquire` but provide information about a time spent in sleep queue or spinning.

Note

SystemTap probes that are shown with font are only available when kernel built with debug configuration options such as `CONFIG_DEBUG_MUTEXES`.

There is a separate consumer for lockstat provider in Solaris: `lockstat` is a separate utility which doesn't require a script to be written.

Events

Another type of synchronization primitives is event notifications. For example, command that expects input on tty should be queued into corresponding queue of tty device so then user puts data into it they will be activated and can begin processing of user input. Linux provides *wait queues* to implement such behaviour with a simplified interface to them called *completion variable*.

They can be traced with following script:

Source file: [scripts/stap/wqtrace.stp](#)

Here is example command which periodically awakens `cat` process on pipe input:

```
$ bash -c 'for l in a b c d e f g h;
do echo $l;
sleep 0.1; done' | cat > /dev/null
```

If we'd run that command, we will see similar output:

```
[11704]bash pipe_wri te: __wake_up_sync_key
      wq head: 0xfffff8800371f6628 state: 0x1 nr: 1
[11704]bash do_wai t: add_wai t_queue
      wq head: 0xfffff880039e11220 wq: 0xfffff88001f89ff20
      tsd: 0xfffff88003971a220 state: ffffffff func: €
      chld_wai t_cal l back
[11705]cat pipe_wai t: fi ni sh_wai t
      wq head: 0xfffff8800371f6628 wq: 0xfffff88001ee47d40
[11705]cat pipe_read: __wake_up_sync_key
      wq head: 0xfffff8800371f6628 state: 0x1 nr: 1
[11705]cat pipe_wai t: prepare_to_wai t
      wq head: 0xfffff8800371f6628 wq: 0xfffff88001ee47d40
      tsd: 0xfffff8800397196c0 state: 1 func: autoremove_wake_functi on
```

As you can see, bash triggers `pipe_wri te()` function to write a character to a pipe with a `cat` process. After that `cat` process awakens from queue with head `0xfffff8800371f6628` and goes through `fi ni sh_wai t()` function. It reads data from pipe, notifies writers that there is free space in pipe buffer that can be written into and after putting letter to `/dev/null` sleeps again in `prepare_to_wai t()` function. If we hadn't redirected `cat` output to `/dev/null`, than we would see longer chain of activated processes, maybe including SSH daemon process which host `pty` and network activity.

In Solaris kernel event notification is performed with pair of mutex and *condition variable* of type `kcondvar_t`. It has pretty simple interface: `cv_wai t` family of functions waits on condition variable (adds process to sleep queue) with optional timeout parameter and allowing signal handling, `cv_si gnal` notifies single thread and wakes up it, `cv_broadcast` wakes up all threads:

Source file: [scripts/dtrace/cvtrace.d](#)

An example with cat utility which will used above will induce following output:

```
[15087] cat cv_wai t_si g_swap_core cv: ffffc1000c9dde9c mutex: ffffc1000c9dde40 €
        timeout: 0
        genuni x` cv_wai t_si g_swap+0x18
        fi fofs` fi fo_read+0xc7
        genuni x` fop_read+0xaa
        genuni x` read+0x30c

[15086] bash cv_broadcast cv: ffffc1000c9dde9c
        fi fofs` fi fo_wakereader+0x2f
        fi fofs` fi fo_wri te+0x316
        genuni x` fop_wri te+0xa7
        genuni x` wri te+0x309
```

Kernel also provide tools for implementing thread blocking in user space. Solaris provides set of syscalls for doing that, such as `lwmutex_timedlock` while Linux supplies universal system call called *futex* (fast userspace mutex).

Interrupt handling and deferred execution

Single instance of processor (core or hardware thread) can execute only single flow of instructions and won't switch to another flow of instructions unless it is explicitly specified with branch instruction. This model, however, prevents operating system from implementing illusion of multiprocessing by periodically switching active threads (which represent flow of instructions). To implement multiprocessing and many other concepts, processor provide mechanism of *interrupts*.

When device, another processor or internal processor unit has to notify current processor about some event: arrival of data in ring buffer of NIC, process exit leading to killing all of its threads or integer division by zero, they send an *interrupt request (IRQ)*. Multiprocessing itself is handled through virtual device, called *system timer* which can send interrupt after pre-defined time. In response to interrupt request, processor saves context on current stack and switches its execution to pre-defined *interrupt service routine (ISR)* or *interrupt handler*. The closest userspace analogue of interrupts is signals.

Interrupts are generally considered bad for performance as their handling "steals" time from actual program, leads to cache cooldown, etc., so a lot of effort in operating system development is put to reduce their negative effect. It is done through better balancing interrupts across processors and deferring execution of the interrupt service routine. Due to that only high-priority interrupts such as *Non-Maskable Interrupt (NMI)* are handled directly in a interrupt service routine.

Most interrupts use *interrupt threads* - separate threads that can handle interrupt but in a same time can be interrupted or de-scheduled (but they will have highest priorities in Solaris). These threads are created by device drivers in Linux and then saved into `handler` field of `irqaction` structure and are being activated if interrupt handler has returned `IRQ_WAKE_THREAD` code. In Solaris, on contrary, each processor has its own interrupt handling thread which is saved into `cpu_intr_thread` of `cpu_t`.

SystemTap provides `irq` tapset which contains probes for tracing interrupt handlers:

```
# stap --all-modules -e '
    probe irq_handler.entry, irq_handler.exit {
        printf("%-16s %s irq %d dev %s\n", pn(), symname(handler),
```

```
irq, kernel_string(dev_name)); } '
```

Solaris has several SDT probes that can be used to trace interrupt handlers too:

```
# dtrace -n '
  av_dispatch_autovect:entry {
    self->vec = arg0; }
  sdt:::interrupt* {
    printf("%s irq %d dev %s", probename, self->vec,
      (arg0) ? stringof(((struct dev_info*) arg0)->devi_addr) : "??");
    sym(arg1); } '
```

Interrupt threads is not the only method to defer execution of certain kernel code: kernel provides a lot of other facilities to do so. They are referred to as *bottom halves* of interrupt, while interrupt handler itself is a *top half* which responsibility is to activate bottom half. An example of them is deferred interrupt handlers **softirqs** and **tasklets** in Linux which are executed in the context of ksoftirqd-N kernel threads. They are prioritized where TASKLET_SOFTIRQ has the lesser priority and serves for execution of tasklets (they are more lightweight than softirqs). They could be traced using softirq.entry and softirq.exit probes.

If bottom half (or any other in-kernel job) has to be executed at specified moment of time, Linux provides **timers** while Solaris has *cyclic subsystem* which can be accessed through **callouts** or timeout()/untimeout() calls.

To simplify execution of small chunks of work Linux provide **workqueue** mechanism which has closest analogue in Solaris — **task queues**. They provide a pool of worker threads whose extract function and data pointers from a queue and call that function. Many drivers may implement their own work queues. Solaris provides static probes to trace task queues:

```
# dtrace -n '
  taskq-exec-start, taskq-enqueue {
    this->tqe = (taskq_ent_t*) arg1;
    printf("%-16s %s ", probename, stringof(((taskq_t*) arg0)->tq_name));
    sym((uintptr_t) this->tqe->tqent_func);
    printf("(%p)", this->tqe->tqent_arg); }'
```

SystemTap provide corresponding probes in irq tapset, but they do not work in modern kernels:

```
# stap --all-modules -e '
  probe workqueue.insert, workqueue.execute {
    printf("%-16s %s %s(%p)\n",
      pn(), task_execname(wq_thread), symname(work_func), work);
  } '
```

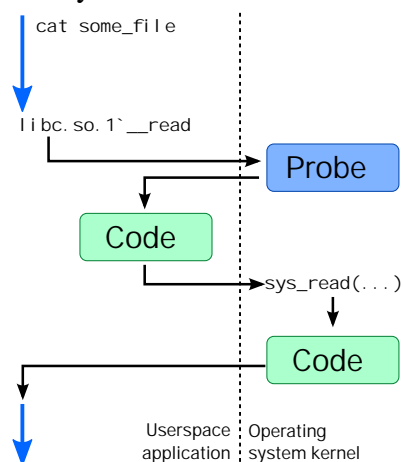
Linux kernel 2.6.36 got new workqueue implementation, and, eventually new set of tracepoints which can be traced like this (SystemTap >=2.5 required):

```
# stap --all-modules -e '
  probe kernel.trace("workqueue_execute_end"),
    kernel.trace("workqueue_execute_start") {
    printf("%s %s(%p)\n",
      pn(), symname($work->func), $work); } '
```

Module 5: Application tracing

Userspace process tracing

We had covered kernel organization in detail in previous chapter, but it would be useless without userspace application that services end-user requests. It can be either simple `cat` program which we used in many previous examples to complex web application which uses web server and relational database. Like with the kernel, DTrace and SystemTap allow to set a probe to any instruction in it, however it will require additional switch to kernel space to execute the code. For example, let's install probe on a `read()` call on the side of standard C library:



In DTrace userspace tracing is performed through `pid` provider:

```
pid1449:libc:__read:entry
```

In this example entry point of `__read()` function from standard C library is patched for process with PID=1449. You may use `return` as name for return probes, or hexadecimal number — in this case it will represent an instruction offset inside that function.

If you need to trace binary file of application itself, you may use `a.out` as module name in probe specification. To make specifying PID of tracing process easier, DTrace provides special macro `$target` which is replaced with PID passed from `-p` option or with PID of command which was run with `-c` option:

```
# dtrace -n '
    pid$target:a.out:main:entry {
```

```
        ustack();
    }' -c cat
```

Userspace probes are created with `process().function()` syntax in SystemTap, where `process` contains path of shared library or executable binary which should be traced. This syntax is similar to kernel syntax (as described in [Probes](#)): it supports specifying line numbers, source file names, `.statement()` and `.return` probes:

```
# stap -e '
    probe process("/lib64/libc.so.6").function("*readir*") {
        print_ubacktrace();
    }' -c ls -d /usr/bin/ls
```

Unlike DTrace, in SystemTap any process which invokes `readir()` call from standard C library will be traced. Note that we used `-d` option so SystemTap will recognize symbols inside `ls` binary. If binary or library is searchable over `PATH` environment variable, you may omit path and use only library name:

```
# export PATH=$PATH:/lib64/
# stap -e '
    probe process("libc.so.6").function("*readir*") {
        [...] }' ...
```

SystemTap uses *uprobes* subsystem to trace userspace processes, so `CONFIG_UPROBES` should be turned on. It was introduced in Linux 3.5. Before that, some kernels (mostly RedHat derivatives) were shipped with *utrace* which wasn't supported by vanilla kernels. It is also worth mentioning that like with kernel tracing, you will need debug information for processes you want to trace that is shipped in `-debuginfo` or `-dbg` packages.

Like with kernel probes, you may access probe arguments using `arg0-argN` syntax in DTrace and `$arg_name` syntax in SystemTap. Probe context is also available. Accessing data through pointers however, would require using `copyin()` functions in DTrace and `user_()` functions in SystemTap as described in [Pointers](#) section.

Warning

Tracing multiple processes in DTrace is hard — there is no `-f` option like in *truss*. It is also may fail if dynamic library is loaded through `dl_open()`. This limitations, however, may be bypassed by using destructive DTrace actions. Just track required processes through process creation probes or `dl_open()` probes, use `stop()` to pause process execution and start required DTrace script. `dtrace_helper.d` from JDK uses such approach.

User Statically Defined Tracing

Like in Kernel mode, DTrace and SystemTap allow to add statically defined probes to a user space program. It is usually referred to as *User Statically Defined Tracing* or *USDT*. As we discovered for other userspace probes, DTrace is not capable of tracking userspace processes and automatically register probes (as you need explicitly specify PID for `pid$` provider). Same works for USDT — program code needs special post-processing that will add code which will register USDT probes inside DTrace.

SystemTap, on contrary, like in case of ordinary userspace probes, uses its *task finder* subsystem to find any process that provides a userspace probe. Probes, however are kept in separate ELF section, so it also requires altering build process. Build process involves `dtrace` tool which is wrapped in SystemTap as Python script, so you can use same build process for DTrace and SystemTap. Building simple program with USDT requires six steps:

- You will need to create a definition of tracing provider (and use `.d` suffix to save it). For example:

```

provider my_prog {
    probe input__val (int);
    probe process__val (int);
};

```

- Here, provider `my_prog` defines two probes `input__val` and `process__val`. These probes take single integer argument.

- (optional) Then you need to create a header for this file:

```
# dtrace -C -h -s provider.d -o provider.h
```

- Now you need to insert probes into your program code. You may use generic `DTRACE_PROBE` macros (in `DTrace`, supported by `SystemTap`) or `STAP_PROBE` macros (in `SystemTap`) from header:

```
DTRACE_PROBE(provider-name, probe-name, arg1, ...);
```

- Or you may use macros from generated header:

```
MY_PROG_INPUT_VAL(arg1);
```

If probe argument requires additional computation, you may use *enabled*-macro, to check if probe was enabled by dynamic tracing system:

```

if(MY_PROG_INPUT_VAL_ENABLED()) {
    int arg1 = abs(val);
    MY_PROG_INPUT_VAL(arg1);
}

```

In our example, program code will look like this:

```
#include
```

```

int main() {
    int val;
    scanf("%d", &val);
    DTRACE_PROBE1(my_prog, input__val, val);
    val *= 2;
    DTRACE_PROBE1(my_prog, process__val, val);
    return 0;
}

```

- Compile your source file:

```
# gcc -c myprog.c -o myprog.o
```

- You will also need to generate stub code for probe points or additional ELF sections, which is also performed by `dtrace` tool. Now it has to be called with `-G` option:

```
# dtrace -C -G -s provider.d -o provider.o myprog.o
```

- Finally, you may link your program. Do not forget to include object file from previous step:

```
# gcc -o myprog myprog.o provider.o
```

Name of a probe would be enough to attach an USDT probe with `DTrace`:

```

# dtrace -n '
    input-val {
        printf("%d", arg0);
    }'

```

Full name of the probe in this case will look like this: `my_prog10678:myprog:main:input-val`. Module would be name of the executable file or shared library, function is the name of C function, name of probe matches name specified in

provider except that double underscores `__` was replaced with single dash `-`. Name of the provider has PID in it like `pid$$` provider does, but unlike it you can attach probes to multiple instances of the program even before they are running.

USDT probes are available via process `tapset`:

```
# stap -e '
    probe process("./myprog").mark("input__val") {
        println($arg1);
    }'
```

Full name of the probe will use following naming schema:

```
process("path-to-program").provider("name-of-provider").mark("name-of-probe")
```

Note that unlike DTrace, SystemTap won't replace underscores with dashes

To implement probe registration, Solaris keeps it in special ELF section called `.SUNW_dof`:

```
# elfdump ./myprog | grep -A 4 SUNW_dof
Section Header[19]: sh_name: .SUNW_dof
sh_addr: 0x8051708 sh_flags: [ SHF_ALLOC ]
sh_size: 0x7a9 sh_type: [ SHT_SUNW_dof ]
sh_offset: 0x1708 sh_entsize: 0
sh_link: 0 sh_info: 0
sh_addralign: 0x8
```

Linux uses ELF *notes* capability to save probes information:

```
# readelf -n ./myprog | grep stapsdt
stapsdt 0x000000033 Unknown note type: (0x00000003)
stapsdt 0x000000035 Unknown note type: (0x00000003)
```

Because of the nature of DTrace probes which are registered dynamically, they could be generated dynamically. We will see it in [JSDT](#). Another implementation of dynamic DTrace probes is [libusdt](#) library.

References

- SystemTap Wiki: [Adding User Space Probing to an Application](#)
-  [Statically Defined Tracing for User Applications](#)

Unix C library

libc is a C library shipped with Unix which provides access to most of its facilities like system calls in a portable manner. Linux *glibc* (one of the implementations of *libc* which is most popular) and *libc* shipped with Solaris contain some USDT probes. We will discuss them in this section.

On Solaris USDT probes are limited to userspace mutexes and read-write locks which available as `plockstat` provider (similar to [lockstat](#) provider we discussed earlier). *glibc*, however implements wider set of probes: along with various *pthread* operations which include not only mutexes and rwlocks but also condition variables and threads operations, it supports tracing of `setjmp/longjmp` and dynamic linker `ld.so`.

Lets see how mutexes are traced in Solaris and Linux (in this section we will assume *glibc* by saying "Linux"). Solaris provides them through `plockstat` provider:

```
plockstatpid::probe-name
```

SystemTap will use standard USDT notation for it:

```
probe process("libpthread.so.0").mark("probe-name")
```

Note that `libpthread.so.0` will vary in different distributions. We will use macro-definitions for paths in our scripts.

Userspace programs have to explicitly ask kernel to block thread that is waiting on condition variable or mutex. Linux provides `futex` system call for it which is wrapped into so-called *low-level-locks* in glibc (they are seen by probes those name start with `lll`). Solaris provides multiple `lwp_*` system calls for it like `lwp_park` which "parks" thread (stops its execution).

Here are list of probes available for userspace mutexes (use them as probe name). First argument (`arg0` in DTrace or `$arg1` in SystemTap) would be address of pthread mutex. Some probes can contain more arguments, i.e. DTrace will pass number of spinning loops to `mutex-spun` probe. Check documentation for them.

Action	DTrace	SystemTap
Creation	-	<code>mutex_init</code>
Destruction	-	<code>mutex_destroy</code>
Attempt to acquire	-	<code>mutex_entry</code>
Busy waiting (spinning)	<code>mutex-spin</code> <code>mutex-spun</code>	-
Attempt to block	<code>mutex-block</code>	<code>lll_lock_wait</code>
Acquired mutex	<code>mutex-locked</code> <code>mutex-acquired</code> <code>mutex-error</code>	<code>mutex_release</code> <code>lll_futex_wake</code>

Here are an example of pthread tracer in SystemTap:

Source file: [scripts/stap/pthread.stp](#)

If we set `tsexperiment` process as a target, we can see how request is passed from control thread to a worker thread (some output is omitted):

```
[8972] process("/lib64/libpthread.so.0").mark("mutex_entry") 0xe1a218
0x7fbcf890fa27 : tpd_wqueue_put+0x26/0x6a [/opt/tsl oad/lib/libtsl oad.so]
[8972] process("/lib64/libpthread.so.0").mark("mutex_acquired") 0xe1a218
0x7fbcf890fa27 : tpd_wqueue_put+0x26/0x6a [/opt/tsl oad/lib/libtsl oad.so]
[8972] process("/lib64/libpthread.so.0").mark("cond_broadcast") 0xe1a240
[8972] process("/lib64/libpthread.so.0").mark("mutex_release") 0xe1a218
0x7fbcf890fa27 : tpd_wqueue_put+0x26/0x6a [/opt/tsl oad/lib/libtsl oad.so]
[8971] process("/lib64/libpthread.so.0").mark("mutex_entry") 0xe1a628
0x7fbcf9148fed : cv_wait+0x2d/0x2f [/opt/tsl oad/lib/libtscommon.so]
0x7fbcf890f93f : tpd_wqueue_pick+0x44/0xbc [/opt/tsl oad/lib/libtsl oad.so]
[8971] process("/lib64/libpthread.so.0").mark("mutex_acquired") 0xe1a628
```

Note that thread with TID=8972 will acquire mutex in `tpd_wqueue_put` function and then send a broadcast message to all workers. One of them (one with TID=8971) wakes up, re-acquires mutex and gets request through `tpd_wqueue_pick`.

`plockstat` doesn't support many probes that glibc do, but we can easily replace them with `pid` provider and function boundary tracing:


Source file: [scripts/dtrace/pthread.d](#)

That script yields similar results on Solaris:

```
[7] mutex_lock_impl: mutex-acquired 0x46d4a0 [46d4a0]
libtsl oad.so`tpd_wqueue_put+0x26
[7] cond_signal: entry 0x46d4e0 [46d4e0]
[7] mutex_unlock_queue: mutex-release 0x46d4a0 [46d4a0]
```

```
[7] mutex_unlock_queue: mutex-release    0x46d4a0    [46d4a0]
[6] mutex_lock_impl: mutex-acquire    0x46d4a0    [46d4a0]
      libtsload.so`tpd_wqueue_pick+0xb6
[6] pthread_cond_wait: return    0x15    [15]
```

References

- [glibc documentation on SystemTap probes](#)
-  [plockstat Provider](#)

Exercise 6

Implement two scripts: `mtxtime.d` and `mtxtime.stp` that would compute delay between attempt to acquire a userspace mutex and a moment when mutex is acquired. Group times by user stacks and print data as logarithmic histograms.

Use `pthread_experiment` to demonstrate your scripts, like in previous section, `TSLoad` workload generator itself would be an object in the experiment. Try to identify mutexes that show delays larger than 1 ms.

Warning

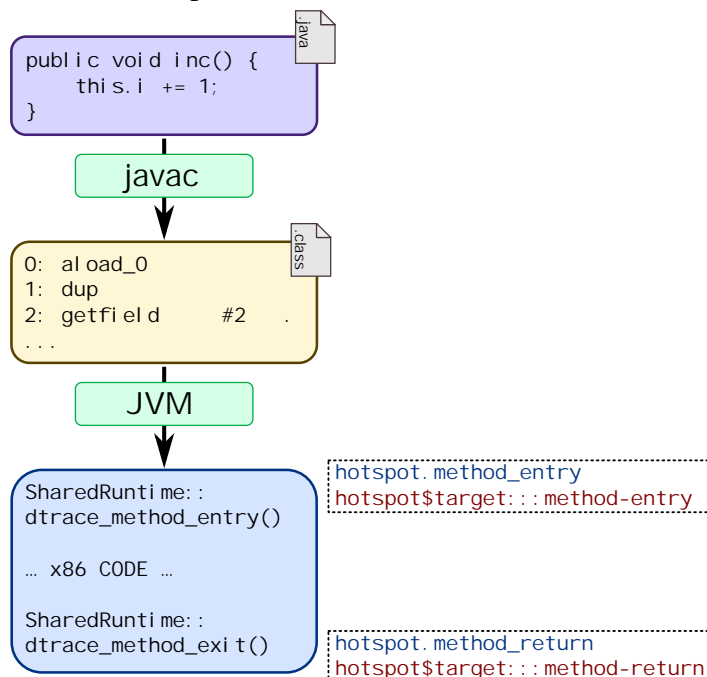
To prevent problems with symbol resolving in DTrace after tracing process finishes, you can attach to a function `experiment_unconfigure()` from `tsexperiment` to print gathered data.

Java Virtual Machine

DTrace and SystemTap are intended to trace applications written in native languages like C or C++ and dependent on compiler ABIs. If application requires virtual machine (in case it is interpreted or translated on-the-fly), virtual machine has to implement USDT probes so it can be traceable by DTrace or SystemTap. For example Zend PHP keeps call arguments in a global object, so you have to access that object to get arguments values instead of using `arg0-argN` syntax.

Same works for *Java Virtual Machine*. Oracle's implementation of JVM, called *Hotspot* and *OpenJDK*, which based on Hotspot support DTrace since version 1.6. It is available through `hotspot` and `hotspot_jni` providers. Latter is intended for tracing of Java Native Interface, so we leave it out of our scope.

By default hotspot provider allows to trace only rare events, such as class loading, starting and stopping threads, VM-wide and GC-wide events and JIT compiler events. That was done to reduce overheads of USDT probes. For example, to trace methods, compiler has to inject two calls into produced code:



To enable additional probes, use following `java` command line options:

- `-XX: +DTraceMethodProbes` — enables function boundary tracing for methods
- `-XX: +DTraceAllLocProbes` — enables tracing of object allocation and deallocation
- `-XX: +DTraceMonitorProbes` — enables object monitors tracing
- `-XX: +ExtendedDTraceProbes` — enables all of events listed above

These options can be set dynamically for running virtual machine using `jinfo` tool.

Tracing provider is implemented in shared library `libjvm.so` which is dynamically loaded using `dlopen()` call. Due to limitations of `pid$` provider we mentioned before, `dtrace` cannot use `hotspot$target` syntax directly:

```
# dtrace -c 'java Test' -n 'hotspot$target:::'
dtrace: invalid probe specifier hotspot$target::: probe description €
hotspot3662:: does not match any probes
```

To launch tracing, use helper script `dtrace_helper.d`: it stops execution of JVM (using `stop()` destructive action) when it loads `libjvm.so` through `dlopen()` and restarts execution of JVM only when tracing script is up and running. Moreover, starting with JDK7, Solaris builds of JDK will use `-xlazyload` linker flag. Due to that, JVM won't register probes automatically until `dtrace` is attached to it explicitly with `-p` options so hotspot probes will be missing from `dtrace -l` outputs. `-p` option will work as expected:

```
# dtrace -p 3682 -n 'hotspot$target:::'
dtrace: description 'hotspot$target:::' matched 66 probes
```

`-Z` option can also be helpful.

JDK shipped in `openjdk` packages in RHEL-like operating systems supports hotspot tracing too. `SystemTap` provides its probes through `hotspot_tapset` which doesn't have such limitations and can be used directly:

```
# stap -e 'probe hotspot.* { println(pn()); }' -c 'java Test'
```

Let's write small program, called *Greeter* which will write "Hello, DTrace" from four threads. Its implementation is based on *Greeter* from [Solaris Internals wiki](#) with small difference: `Greeting.greet()` method uses `synchronized` keyword so it will use monitor.

Source file: [scripts/src/java/Greeting.java](#)

Source file: [scripts/src/java/GreetingThread.java](#)

Source file: [scripts/src/java/Greeter.java](#)

Here are small tracer for it implemented with SystemTap:

Source file: [scripts/stap/hotspot.stp](#)

Similar script on DTrace will look like this and should use `dtrace_helper.d` or called with `-Z` option:

Source file: [scripts/dtrace/hotspot.d](#)

Note that we are using `copyin` function to copy strings from userspace instead of `copyinstr`. That is because hotspot probes pass strings as non-null-terminated. Due to that, it will use additional argument to pass string length.

Here are examples of this script outputs:

```
class-loaded [??] Test
...
class-loaded [??] Greeting
...
class-loaded [??] GreetingThread
...
thread-start [ 14] Thread-1
method-entry [ 9] GreetingThread.run
method-entry [ 9] Greeting.greet
...
monitor-contended-exit [ 8] Greeting
method-return [ 8] Greeting.greet
method-entry [ 8] java/lang/Thread.sleep
method-return [ 9] java/lang/Thread.sleep
monitor-contended-enter [ 9] Greeting
method-entry [ 9] Greeting.greet
method-entry [ 9] java/io/PrintStream.println
```

You can see that when thread leaves `Thread.sleep()` method, it acquires monitor of `Greeting` object, calls `Greeting.greet()` method which will call `PrintStream.println()` method to output line.

Here are list of probes provided by JVM in `hotspot$target` DTrace provider and `hotspot` tapset:

Action	DTrace	SystemTap
JVM		
Start	vm-init-begin vm-init-end	hotspot.vm_init_begin hotspot.vm_init_end
Shutdown	vm-shutdown	hotspot.vm_shutdown
Threads		

<i>Action</i>	<i>DTrace</i>	<i>SystemTap</i>
Start	thread-start <ul style="list-style-type: none"> • arg0:arg1 — thread name • arg2 — internal JVM thread's identifier • arg3 — system's thread identifier (LWP id) • arg4 — is thread a daemon? 	hotspot.thread_start <ul style="list-style-type: none"> • thread_name — thread name • id — internal JVM thread's identifier • native_id — system's thread identifier (task id) • is_daemon — is thread a daemon?
Stop	thread-stop Arguments are same as for thread-start	hotspot.thread_stop Arguments are same as for hotspot.thread_start
Methods		
Call	method-entry <ul style="list-style-type: none"> • arg0 — internal JVM thread's identifier • arg1:arg2 — class name • arg3:arg4 — method name • arg5:arg6 — method signature 	hotspot.method_entry <ul style="list-style-type: none"> • thread_id — internal JVM thread's identifier • class — class name • method — method name • sig — method signature
Return	method-return Arguments are same as for method-entry hotspot.method_return	Arguments are same as for hotspot.method_entry
Class loader		
Load	class-loaded <ul style="list-style-type: none"> • arg0:arg1 — class name • arg2 — class loader id • arg3 — does class originate from shared archive? 	hotspot.class_loaded <ul style="list-style-type: none"> • class — class name • classloader_id — class loader id • is_shared — does class originate from shared archive?
Monitors (locks)		
Attempt to acquire	monitor-contended-enter arg0 — Java thread id arg1 — unique monitor id arg2:arg3 — class name	hotspot.monitor_contended_enter thread_id — Java thread id id — unique monitor id class — class name
Acquire	monitor-contended-entered Arguments are same as for monitor-contended-enter	hotspot.monitor_contended_entered Arguments are same as for monitor_contended_enter
Release	monitor-contended-exit Arguments are same as for monitor-contended-enter	hotspot.monitor_contended_exit Arguments are same as for monitor_contended_enter
Monitors (events)		
Entering .wait()	monitor-wait Arguments are same as for monitor-contended-enter with one addition: arg4 keeps timeout	hotspot.monitor_wait Arguments are same as for monitor_contended_enter with one addition: timeout variable keeps timeout
Leaving .wait()	monitor-waited Arguments are same as for monitor-contended-enter	hotspot.monitor_waited Arguments are same as for monitor_contended_enter
.notify()	monitor-notify Arguments are same as for monitor-contended-enter	hotspot.monitor_notify Arguments are same as for monitor_contended_enter

<i>Action</i>	<i>DTrace</i>	<i>SystemTap</i>
<code>. notifyAll ()</code>	<code>monitor-notifyAll</code> Arguments are same as for <code>monitor-contended-enter</code>	<code>hotspot.monitor_notifyAll</code> Arguments are same as for <code>monitor_contended_enter</code>
Allocator and garbage collector		
GC cycle has been started	<code>gc-begin</code> <ul style="list-style-type: none"> <code>arg0</code> — is this cycle full? 	<code>hotspot.gc_begin</code> <ul style="list-style-type: none"> <code>is_full</code> — is this cycle full
GC cycle has been finished	<code>gc-end</code>	<code>hotspot.gc_end</code>
Garbage collection is initiated for memory pool	<code>mem-pool-gc-begin</code> <ul style="list-style-type: none"> <code>arg0:arg1</code> — name of manager <code>arg2:arg3</code> — name of memory pool <code>arg4</code> — initial pool size <code>arg5</code> — used memory <code>arg6</code> — number of committed pages <code>arg7</code> — maximum usable memory 	<code>hotspot.mem_pool_gc_begin</code> <ul style="list-style-type: none"> <code>manager</code> — name of manager <code>pool</code> — name of memory pool <code>initial</code> — initial pool size <code>used</code> — used memory <code>committed</code> — number of committed pages <code>max</code> — maximum usable memory
Garbage collection is finished in a memory pool	<code>mem-pool-gc-end</code> Arguments are same as for <code>mem-pool-gc-begin</code>	<code>hotspot.mem_pool_gc_end</code> Arguments are same as for <code>hotspot.mem_pool_gc_begin</code>
JIT compiler		
Start of method compilation	<code>method-compile-begin</code> <code>arg0:arg1</code> — compiler name <code>arg2:arg3</code> — class name <code>arg4:arg5</code> — method name <code>arg6:arg7</code> — method signature	<code>hotspot.method_compile_begin</code> <code>compiler</code> — compiler name <code>class</code> — class name <code>method</code> — method name <code>sig</code> — method signature
Ending of method compilation	<code>method-compile-end</code> Arguments are same as for <code>method-compile-begin</code> . with one addition: <code>arg8</code> keeps compilation result	<code>hotspot.method_compile_end</code> Arguments are same as for <code>hotspot.method_compile_begin</code> . with one addition: <code>\$arg9</code> keeps compilation result
Load of compiled method	<code>compiled-method-load</code> <ul style="list-style-type: none"> <code>arg0:arg1</code> — class name <code>arg2:arg3</code> — method name <code>arg4:arg5</code> — method signature <code>arg6</code> — instruction pointer <code>arg7</code> — size of compiled code 	<code>hotspot.compiled_method_load</code> <ul style="list-style-type: none"> <code>class</code> — class name <code>method</code> — method name <code>sig</code> — method signature <code>code</code> — instruction pointer <code>size</code> — size of compiled code
Unload of compiled method	<code>compiled-method-unload</code> <ul style="list-style-type: none"> <code>arg0:arg1</code> — class name <code>arg2:arg3</code> — method name <code>arg4:arg5</code> — method signature 	<code>hotspot.compiled_method_unload</code> <ul style="list-style-type: none"> <code>class</code> — class name <code>method</code> — method name <code>sig</code> — method signature

In addition to provided probe arguments, SystemTap will supply `name` which will contain probe name, and `probeStr` which keeps string with pre-formatted probe arguments. There are also several probes that are not documented: such as `class-initialization-*` and thread probes: `thread-sleep-*`, `thread-yield`.

SystemTap and DTrace can also collect backtraces of a running Java thread. DTrace provide `jstack()` function for that:

```
# dtrace -n '
    syscall::write:entry
    / execname == "java" /
    { jstack(); }'
```

SystemTap needs to gather some information about VM to build stack traces correctly, so it needs to bind to probe hotspot. `vm_init_end`, so `print_jstack()` will work only if you run SystemTap with `-c` option:

```
# stap -e '
    probe syscall::write {
        if(pid() == target())
            print_jstack();
    }' -c 'java Test'
```

However, you can alter source code of `jstack` tapset to use other global events and use `jstack()` on live processes.

Warning

There is a bug in JDK: [JDK-7187999: dtrace jstack action is broken](#). Due to it, `jstack()` won't work for Java7 in Solaris 11. One of workarounds is to try to seek for available probes in a process:

```
# dtrace -P fooJava-PID
```

That attempt will fail, but it will lead DTrace to extract required helper functions from Java process.

JSDT

You could notice that we can't extract method's arguments in method probes like we did it in other places via `args` array. That complicates Java application tracing. As you can remember from USDT description, in DTrace applications can register their probes within DTrace. This is also true for Java applications which can provide *Java Statically Defined Tracing* probes (JSDT). It is supported only in DTrace and only in BSD or Solaris.

JSDT is implemented in packages `com.sun.tracing` and `sun.tracing`. Each provider should be a class which implements `com.sun.tracing.Provider` interface, while each method of this class will be a probe. Reimplement our greeting example with JSDT support:

Source file: [scripts/src/jsdt/Greeting.java](#)

Source file: [scripts/src/jsdt/GreetingProvider.java](#)

Source file: [scripts/src/jsdt/JSDT.java](#)

Package `sun.tracing` is treated as "closed", so you will need to pass an option to `javac` to compile JSDT:

```
$ javac -XDignore.symbol.file JSDT.java
```

You can see that our provider was registered within DTrace when we start JSDT example and we can trace it:

```
root@sol11: ~/java1/hs1# dtrace -l | grep GreetingProvider
69255 GreetingProvider3976      java_tracing      unspeci fi ed €
      greetingStart
69256 GreetingProvider3976      java_tracing      unspeci fi ed €
      greetingEnd
root@sol11: ~/java1/hs1# dtrace -n 'greetingStart { trace(arg0); }'
dtrace: description 'greetingStart' matched 1 probe
CPU      ID      FUNCTION: NAME
0 69255      unspeci fi ed: greetingStart      61
```

P.S.: Of course, DTrace and SystemTap are not the only option to trace Java. It provides JVMTI interface since Java 6 which allows to instrument Java applications as well. Most famous implementation of JVMTI is BTrace.

Non-native languages

As DTrace became popular, many language interpreters got USDT probes. Some of them adopted them in upstream, some probes are provided by the binaries custom packages shipped with operating system. The basic pair of probes provided by most language interpreters are function entry and exit probes which provide name of the function, line number and file name. For example, Perl can be traced that way:

```
# echo 'use Data::Dumper;
    map { Dumper($_, "", "") } ("Hello", "world");' |
    dtrace -n '
        perl$target::sub-entry {
            trace(copyinstr(arg0)); trace(copyinstr(arg1)); trace(arg2);
        } -c 'perl -- -'

# stap -e '
    probe process("/usr/lib64/perl5/CORE/libperl.so").mark("sub__entry") {
        printf(" : ", user_string($arg1), user_string($arg2), $arg3);
    }
    -c '$ perl -e \'use Data::Dumper;
        map { Dumper($_, "", "") } ("Hello", "world");\''
```

Note that we had to use stdin as a script in DTrace example. That happened because DTrace cannot parse `-c` option value in shell-like manner.

Language interpreters provide not only function entry probes, here are other examples of supplied probes:

- Function entry and exit probes.
 - In PHP and Python — `function-entry/function-return`.
 - In Perl — `sub-entry/sub-return`.
 - In Ruby — `method-entry/method-return`.
 - In Tcl — `proc-entry/proc-return`.
- Probes that fire inside function: `line` in Python which corresponds to a interpreted line and `execute-entry/execute-return`, which fire per each Zend interpreter VM operation.
- Probes of file execution and compilation: such as `compile-file-entry/compile-file-return` in PHP and `loading-file/loaded-file` in Perl
- Error and exception probes: `raise` in Ruby and `exception-thrown/exception-caught/error` in PHP
- Object creation probes: `obj-create/obj-free` in Tcl, `instance-new-*/instance-delete-*` in Python, `object-create-start/object-create-done/object-free` in Ruby
- Garbage collector probes: `gc-start/gc-done` in Python, `gc-*/begin/gc-*/end` in Ruby 2.0 or `gc-begin/gc-end` in Ruby 1.8

Here are list of availability of that probes in various interpreters shipped as binary packages. If you lack them, you may want to rebuild interpreters with some configure option like `--enable-dtrace`.

Interpreter	CentOS	Solaris
Python 2	Python has never accepted DTrace patches into upstream. However, it was implemented by Solaris developers for Python 2.4, and being ported to Fedora's and CentOS python. Only function-related probes are supplied: <code>function-entry</code> and <code>function-return</code> .	
Python 3	Like Python 2, Python 3 in CentOS (if installed from EPEL) supports <code>function__entry</code> and <code>function__return</code> probes. In addition to that, SystemTap supplies example <code>python3</code> tapset.	Python 3 is supplied as FOSS (unsupported) package in Solaris 11 and has <code>line</code> probe, instance creation and garbage-collector related probes.
	Starting with Python 3.6, DTrace probes function entry and exit probes, garbage collector probes and <code>line</code> are supported by vanilla interpreter	
PHP5	Doesn't support USDT tracing but can it be enabled via <code>--enable-dtrace</code> switch when it is built from source.	PHP supports tracing functions, exceptions and errors, VM opcodes execution and file compiling from scratch. Its probes will be discussed in the following section, Web applications .
Ruby 2	Supports multiple probes including object creation, method entry and exit points and garbage collector probes in Ruby 2.0 in CentOS or Ruby 2.1 as FOSS package in Solaris 11.	
Perl 5	Supports subroutine probes <code>sub-entry</code> and <code>sub-return</code> (see examples above).	
Go	Go is pretty close to native languages in Linux, so you can attach probes directly to its functions while backtraces show correct function names. Differences in type system between C-based languages and Go, however prevents SystemTap from accessing arguments.	Go has experimental support for Solaris so it is not considered as a target for DTrace.
Erlang	Neither EPEL nor Solaris package feature USDT probes in BEAM virtual machine, but they are supported in sources, so building with <code>--with-dynamic-trace</code> option enables various probes including function-boundary probes.	
Node.JS	Node.JS is not supplied as OS packages, while binaries from official site doesn't have USDT enabled in Linux or simply not working in Oracle Solaris (only in Illumos derivatives). Building from sources, however adds global network-related probes like <code>http-server-request</code> . Function boundary tracing is not supported.	

Most interpreted language virtual machines still rely on `libc` to access basic OS facilities like memory allocation, but some may use their own: i.e. `PyMalloc` in Python, Go runtime is OS-independent in Go language. For example let's see how `malloc` calls may be cross-referenced with Python code in `yum` and `pkg` package managers using SystemTap or DTrace. We will need to attach to function entry and exit points to track "virtual" python backtrace and `malloc` call to track amount of allocated bytes. This approach is implemented in the following couple of scripts:

Source file: [scripts/dtrace/pymalloc.d](#)

Source file: [scripts/stap/pymalloc.stp](#)

Note

We have used non-existent `foo` provider in DTrace example because like JVM, Python is linked with `-xla` linker flag, so we apply same workaround to find probes that we used in [Java Virtual Machine](#)

section.

Arguments and local variables are also inaccessible directly by SystemTap or DTrace when program in non-native language is traced. That happens because they are executed within virtual machine which has its own representation of function frame which is different from CPU representation: languages with dynamic typing are more likely to keep local variables in a dict-like object than in a stack. These frame and dict-like objects, however, are usually implemented in C and available for dynamic tracing. All that you have to do is to provide their layout.

Let's see how this can be done for Python 3 in Solaris and Linux. If you try to get backtrace of program interpreted by Python 3, you will probably see function named `PyEval_EvalCodeEx` which is responsible for evaluation of code object. Code object itself has type `PyCodeObject` and passed as first argument of that function. That structure has fields like `co_firstlineno`, `co_name` and `co_filename`. Last two fields not just C-style strings but kind of `PyUnicodeObject` — an object which represents strings in Python 3. It have multiple layouts, but we rely on the simplest one: compacted ASCII strings. That may not be true for all string objects in the program, but that works fine for objects produced by the interpreter itself like code objects.

DTrace cannot recognize type information from Python libraries, but it supports struct definitions in the code. We will use it to provide `PyCodeObject` and `PyUnicodeObject` layouts in a separate file `pycode.h`. DTrace syntax is pretty much like C syntax, so these definitions are almost copy-and-paste from Python sources. Here is an example of DTrace scripts which trace python program execution:

Source file: [scripts/dtrace/pycode.h](#)

Source file: [scripts/dtrace/pycode.d](#)

Note

Similar mechanism is used in so-called *ustack helpers* in DTrace. That allows to build actual backtraces of Python or Node.JS programs when you use `jstack()` action.

SystemTap can extract type information directly from DWARF section of shared libraries so all we need to do to achieve same effect in it is to use `@cast` expression:

Source file: [scripts/stap/pycode.stp](#)

References

- *Python*: Bugs [#4111](#), [#13405](#) and [#21590](#)
- *Perl*: [perltrace](#)
- *PHP*: [Using PHP and DTrace](#)
- *Ruby*: [DTrace Probes](#)
- *Erlang*: [DTrace and Erlang/OTP](#)

Web applications

Many interpreted languages that are used in Web development like Python, Perl, PHP and Ruby implement USDT probes. Some HTTP servers like Apache HTTP server (there is also an nginx fork, called nginx-dtrace) and databases such as MySQL, PostgreSQL and Berkeley DB provide them too. Let's see how that can be used to trace a real web application like Drupal CMS framework.

Warning

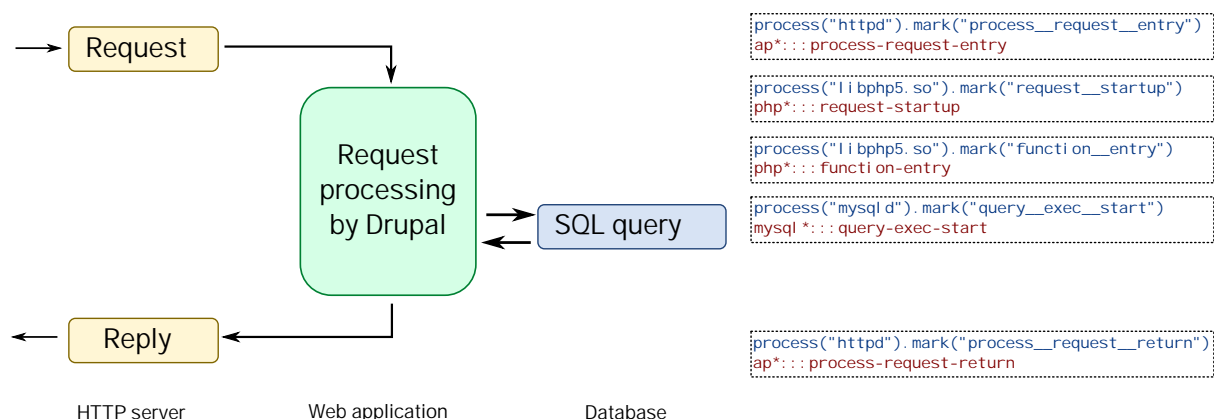
Despite that Apache HTTP server declares support of USDT probes, it is not supported by its build system (as you can remember, you need to perform additional steps and build extra object file). Due to that, when you build it, with `--enable-dtrace` option, you will see error message:

DTrace Support in the build system is not complete. Patches Wel come!

There is a patch written by Theo Schlossnagle that modifies build system properly, but it won't accepted. You can find fresh version of it in a bug [55793](#).

An alternative of that is to use `mod_dtrace` module, but we won't discuss it in our book.

We will use Drupal with MySQL database running under Zend PHP interpreter in Apache web-server in `mod_php` mode. You can also use PHP-FPM, but it makes requests mapping harder as requests would be processed by different processes. In our case, without PHP-FPM, web-application and http-server would be of same context:



You will need to use provider name to access PHP, MySQL and Apache HTTP Server probes. Their naming convention is the same as any other USDT probe:

`php*:::probe-name`

`mysql*:::probe-name`

`ap*:::probe-name`

Same works for SystemTap: provider names are optional, but you will need to specify full path to a binary file or use its name and setup `PATH` environment variable:

```
process("httpd").mark("probe__name")
```

```
process("mysqld").mark("probe__name")
```

```
process("libphp5.so").mark("probe__name")
```

We will use full paths in macros in example scripts.

Here are list of arguments and names of some useful Apache probes:

<i>Action</i>	<i>DTrace</i>	<i>SystemTap</i>
Request is redirected	internal-redirect <ul style="list-style-type: none"> arg0 — old URI arg1 — new URI 	internal__redirect <ul style="list-style-type: none"> \$arg1 — old URI \$arg2 — new URI
Request is read from socket	read-request-entry <ul style="list-style-type: none"> arg0 — request_rec structure arg1 — conn_rec structure 	read__request__entry <ul style="list-style-type: none"> \$arg1 — request_rec structure \$arg2 — conn_rec structure
	read-request-success <ul style="list-style-type: none"> arg0 — request_rec structure arg1 — method (GET/POST/...) arg2 — URI arg3 — server name arg4 — HTTP status 	read__request__success <ul style="list-style-type: none"> \$arg1 — request_rec structure \$arg2 — method (GET/POST/...) \$arg3 — URI \$arg4 — server name \$arg5 — HTTP status
	read-request-failure <ul style="list-style-type: none"> arg0 — request_rec structure 	read__request__failure <ul style="list-style-type: none"> \$arg1 — request_rec structure
Request is processed	process-request-entry <ul style="list-style-type: none"> arg0 — request_rec structure arg1 — URI 	process__request__entry <ul style="list-style-type: none"> \$arg1 — request_rec structure \$arg2 — URI
	process-request-return <ul style="list-style-type: none"> arg0 — request_rec structure arg1 — URI arg2 — HTTP status 	process__request__return <ul style="list-style-type: none"> \$arg1 — request_rec structure \$arg2 — URI \$arg3 — HTTP status

Warning

When read-request-entry/read__request__entry probe is firing, request_rec structure fields is not yet filled.

There are also many Apache Hooks probes, but they are not providing useful arguments.

Following table provides list of useful PHP SAPI probes:

<i>Action</i>	<i>DTrace</i>	<i>SystemTap</i>
Request processing		
Request processing started	request-startup <ul style="list-style-type: none"> arg0 — file name arg1 — request URI arg2 — request method 	request__startup <ul style="list-style-type: none"> \$arg1 — file name \$arg2 — request URI \$arg3 — request method
Request processing finished	request-shutdown Arguments are same as for request-startup	request__shutdown Arguments are same as for request__startup
Compiler		
Compilation	compile-file-entry <ul style="list-style-type: none"> arg0 — source file name arg1 — compiled file name 	compile__file__entry <ul style="list-style-type: none"> \$arg1 — source file name \$arg2 — compiled file name
File is compiled	compile-file-return Arguments are same as for compile-file-entry	compile__file__return Arguments are same as for compile__file__entry
Functions		

<i>Action</i>	<i>DTrace</i>	<i>SystemTap</i>
Function call	functi on-entry <ul style="list-style-type: none"> • arg0 — function name • arg1 — file name • arg2 — line number • arg3 — class name • arg4 — scope operator :: 	functi on__entry <ul style="list-style-type: none"> • \$arg1 — function name • \$arg2 — file name • \$arg3 — line number • \$arg4 — class name • \$arg5 — scope operator ::
Function return	functi on-return Arguments are same as for functi on-entry	functi on__return Arguments are same as for functi on__entry
VM execution		
Beginning of operation execution	execute-entry <ul style="list-style-type: none"> • arg0 — file name • arg1 — line number 	execute__entry \$arg1 — file name \$arg2 — line number
Beginning of operation execution	execute-return <ul style="list-style-type: none"> • Arguments are same as for execute-entry 	execute__return <ul style="list-style-type: none"> • Arguments are same as for execute__entry
Errors and exceptions		
PHP error	error <ul style="list-style-type: none"> • arg0 — error message • arg1 — file name • arg2 — line number 	error <ul style="list-style-type: none"> • \$arg1 — error message • \$arg2 — file name • \$arg3 — line number
Thrown exception	excepti on-thrown arg0 — exception class name	excepti on__thrown arg0 — exception class name
Caught exception	excepti on-caught <ul style="list-style-type: none"> • Arguments are same as for excepti on-thrown 	excepti on__caught <ul style="list-style-type: none"> • Arguments are same as for excepti on__thrown

MySQL has wide set of probes. They are described in MySQL documentation: [5.4.1 mysqld DTrace Probe Reference](#). Here are list of basic probes which allow to trace queries and connections:

<i>Action</i>	<i>DTrace</i>	<i>SystemTap</i>
Connection	connecti on-start <ul style="list-style-type: none"> • arg0 — connection number • arg1 — user name • arg2 — host name 	connecti on__start <ul style="list-style-type: none"> • \$arg1 — connection number • \$arg2 — user name • \$arg3 — host name
	connecti on-done <ul style="list-style-type: none"> • arg0 — connection status • arg1 — connection number 	connecti on__done <ul style="list-style-type: none"> • \$arg1 — connection status • \$arg2 — connection number
Query parsing	query-parse-start <ul style="list-style-type: none"> • arg0 — query text 	query__parse__start <ul style="list-style-type: none"> • \$arg1 — query text
	query-parse-done <ul style="list-style-type: none"> • arg0 — status 	query__parse__done <ul style="list-style-type: none"> • \$arg1 — status

Action	DTrace	SystemTap
Query execution	query-exec-start <ul style="list-style-type: none"> • arg0 — query text • arg1 — connection number • arg2 — database name • arg3 — user name • arg4 — host name • arg5 — source of request (cursor, procedure, etc.) 	query__exec__start <ul style="list-style-type: none"> • \$arg1 — query text • \$arg2 — connection number • \$arg3 — database name • \$arg4 — user name • \$arg5 — host name • \$arg6 — source of request (cursor, procedure, etc.)
	query-exec-done <ul style="list-style-type: none"> • arg0 — status 	query__exec__done <ul style="list-style-type: none"> • \$arg1 — status

Here are simple tracer for PHP web application which is written on SystemTap:

Source file: [scripts/stap/web.stp](#)

If you run it and try to access index page of Drupal CMS with your web-browser, you will see similar traces:

```
[httpd] read-request
[httpd] read-request      GET ??? €
                        /drupal /modul es/contextual /i mages/gear-sel ect. png  [status: 200]
[httpd] process-request  '/drupal /modul es/contextual /i mages/gear-sel ect. png'
[httpd] process-request  '/drupal /modul es/contextual /i mages/gear-sel ect. png' €
                        access-status: 304
[httpd] read-request
[httpd] read-request      GET ??? /drupal /  [status: 200]
[httpd] process-request  '/drupal /'
[ PHP ] request-startup  GET '/drupal /i ndex. php' file: €
                        /usr/l ocal /apache2/htdocs/drupal /i ndex. php
[ PHP ] function-entry   main file: index. php: 19
[ PHP ] function-return  main file: index. php: 19
f
[ PHP ] function-entry   DatabaseStatementBase::execute file: database. inc: 680
[MySQL] query-parse      'SELECT u. *, s. * FROM users u INNER JOIN sessions s ON €
                        u. uid = s. uid WHERE s. si d = 'yI R5hLWScBNAfwOby2R3Fi DfDoki U456ZE-rBDsPfu0' ' stat
[MySQL] query-exec       'SELECT u. *, s. * FROM users u INNER JOIN sessions s ON €
                        u. uid = s. uid WHERE s. si d = 'yI R5hLWScBNAfwOby2R3Fi DfDoki U456ZE-rBDsPfu0' ' stat
...
[ PHP ] request-shutdown      GET '/drupal /i ndex. php' file: €
                        /usr/l ocal /apache2/htdocs/drupal /i ndex. php
[httpd] process-request  '/drupal /i ndex. php' access-status: 200
```

As you can see from this trace, there is a request of a static image gear-sel ect. png which is resulted in status 304 and a dynamic page i ndex. php which eventually accesses database to check user session.

Warning

You will need to restart Apache HTTP server after you start web.stp script.

Due to high amounts of script outputs, you will need to increase buffers in DTrace. The rest of script will look similar to web. stp:

Source file: [scripts/dtrace/web.d](#)

Exercise 7

Create two scripts: `topphp.d` and `topphp.stp` which will measure mean execution time of each PHP function and count number of calls to that function. Group functions by request URI and full function name including class name (if defined). Use `drupal` experiment to demonstrate your script.

Note

It would be reasonable to run workload generator on system other than server (so it won't affect execution of web applications). You can switch roles of virtual machines in lab environment i.e. use Solaris as server and Linux as client and vice versa. To alter server's address, use `-s` option in `tsexperiment` command line:

```
# /opt/tload/bin/tsexperiment -e drupal / run \  
-s workloads: drupal : params: server=192.168.13.102
```

Appendix A. Exercise hints and solutions

Exercise 1

This exercise is intended to learn some features of dynamic tracing languages that was discussed in modules 1 and 2. First of all we need to pick probes that we will use in our tracing script. They would be parts of `syscall` provider/tapset. As you can remember from [stap command options](#), probe parameters can be checked with `-L` option:

```
# stap -L 'syscall.open'
syscall.open name:string filename:string flags:long mode:long argstr:string €
           $filename:long int $flags:long int $mode:long int
# stap -L 'syscall.open.return'
syscall.open.return name:string retstr:string $return:long int $filename:long €
                   int $flags:long int $mode:long int
```

Same can be done for `dtrace` with `-l` option:

```
# dtrace -l -f openat\* -v
   ID    PROVIDER      MODULE          FUNCTION NAME
14167    syscall      openat          entry
...
```

Return value (which would represent file descriptor number) will be saved into `$return` variable in SystemTap and `arg1` argument in DTrace. We will also need flags values: `arg2` in DTrace (because they are going third in `openat()` prototype). In SystemTap you can use either DWARF variable `$flags` or tapset variable `flags`. Latter is more stable.

Similarly, path to opened file will be passed as second `openat()` argument and will be available in DTrace as `arg1` or `$filename/filename` in SystemTap. At the moment of system call, however, file path will be a string which is located in user address space, so to get it in tracing script, you will need to copy it by using `copyinstr()` in DTrace or one of `user_string*()` functions. Tapset variable already uses `user_string_quoted()` to access variable, so we will use it in our scripts.

Note that data that we want gather is available in two different probes: flags and file path are provided by entry probe, while file descriptor number can only be collected in return probe (SystemTap can provide flags and file path in return probe, but as we mentioned, it depends on compiler optimizations). Since both probes will be executed in the same context,

we can use [thread-local variables](#).

Finally, stringifying flags will require usage of ternary operator `?:` in DTrace or `if/else` construct in SystemTap. To concatenate strings, use `strjoin` from DTrace or string concatenation operator `.` in SystemTap.

Here are resulting DTrace script which implements required functionality:

Source file: [scripts/dtrace/opentrace.d](#)

I have used `sprintf()` to concatenate strings in SystemTap version of a script:

Source file: [scripts/stap/opentrace.stp](#)

Finally, you will need to add predicates to compare paths with `/etc` in DTrace by using `strstr()` subroutine and comparing it with 0 and SystemTap's `iniustr()` function.

Exercise 2

We will have to use `count()` [aggregation](#) to count `open()` and `openat()` system calls. It can be cleaned up from outdated data with `trunc()` action in DTrace or `delete` operation in SystemTap. These scripts are roughly based on `wstat.d` and `wstat.stp` from [aggregations example](#).

To print current timestamp we can use `%Y` formatting specifier and `walltimestamp` variable in DTrace. Same can be achieved with `ctime()` and `gettimeofday_s()` functions from SystemTap. To print data periodically, we can use [timer probes](#): `timer.s($1)` in SystemTap or `tick-$1s` from DTrace. `$1` here represents first command line argument.

Finally, we need to determine if `open` operation requests creation of file or not. We should write predicate for that which tests flags passed to `open` for `O_CREAT` flag (we have learned how to access flags in previous exercise).

Here are resulting scripts:

Source file: [scripts/dtrace/openaggr.d](#)

Source file: [scripts/stap/openaggr.stp](#)

Exercise 3

Part 1

In the first part of this exercise we will need to check which fields of `struct task_struct` in Linux or `proc_t` are responsible for which aspects of process functioning. You will need to apply following changes to dump task scripts: timer probe has to be replaced to a pair of probes: `proc::exec-*` and `proc::exit` in DTrace or `kprocess.exec_complete` and `kprocess.exit` in SystemTap. We have used exit probes for `execve()` system call to collect command line arguments: they are not filled in unless `execve()` call finishes.

Here list of expected observations during this exercise:

- When you run program with extra argument, it will be cleared in `main()` function, so you will see original argument in `exec-probe`, but only 'X' letters when program exits.
- When you run program through symbolic link `lab3-1`, VFS node which refer to a binary file will point to a regular file `lab3`. That node is represented by `p_exec` field of `proc_t` in Solaris or `exe_file` of `task_struct` in Linux. `execname`, however, will behave differently

in Solaris and Linux.

- When you run program in chroot environment, root process directory will change from / to /tmp/chroot.

Here are resulting scripts (they are not much different from original):

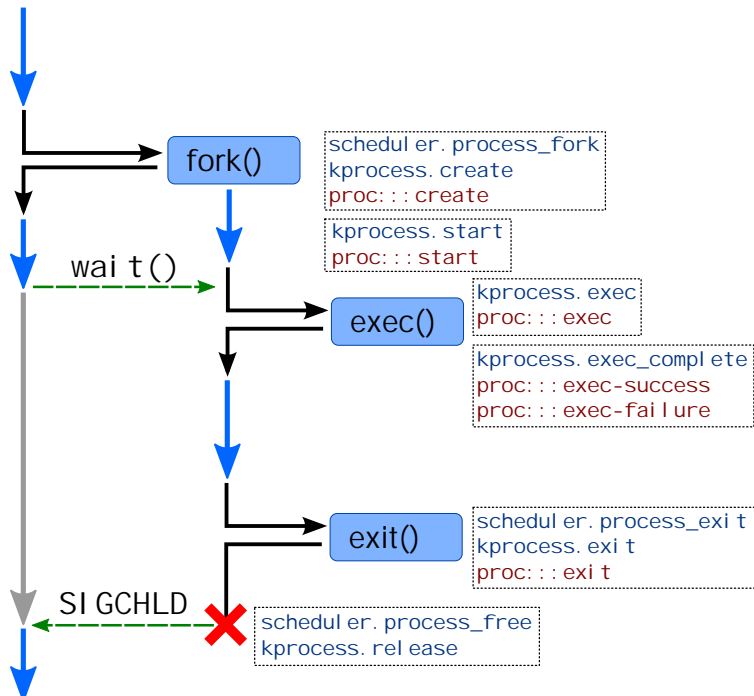
Source file: [scripts/stap/dumptask-lab3.stp](#)

Source file: [scripts/dtrace/dumptask-lab3.d](#)

Part 2

First of all we have to create several associative arrays which will use PID as a key (we can't use thread-local variables here because `exit()` can be called from any of process threads), and timestamp as a value. Final data will be kept in aggregations which we already learned in exercise 2.

We will use probes from the section *Lifetime of a process*. They are shown in the following picture:



However, we do not know PID at the time `fork()` is called so we will use thread-local variable for that. We can check return value of `fork()` in return probe and re-use timestamp saved previously if everything went fine and `fork()` has returned value greater than 1 or throw it away.

We wrote an ugly function `task_args()` to collect process arguments in `dumptask.stp` script. This data is available since SystemTap 2.5: `kprocess. exec` probe provides program's arguments in `argstr` argument. We will use `curpsinfo->pr_psargs` on Solaris as it keeps first 80 characters of command line to get rid of copying userspace arguments too. We will use `timestamp` variable in DTrace as a source of timestamps (again, a tautology). We will use `local_clock_us()` function as we do not care about CPU time skew.

Finally, to reduce memory footprint in SystemTap, we will reduce associative arrays sizes. Here are resulting scripts:

Source file: [scripts/stap/forktime.stp](#)

Exercise 4

Part 1

If you ever traced system calls with `strace` in Linux or `truss` in Solaris, you'll probably noticed that dynamic linker `ld.so` maps shared objects into memory before program is actually run:

```
# strace /bin/ls
...
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\34\2\0\0\0\0\0"... , €
    832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2107600, ...}) = 0
mmap(NULL, 3932736, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = €
    0x7f28fc500000
mprotect(0x7f28fc6b6000, 2097152, PROT_NONE) = 0
mmap(0x7f28fc8b6000, 24576, PROT_READ|PROT_WRITE, €
    MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b6000) = 0x7f28fc8b6000
mmap(0x7f28fc8bc000, 16960, PROT_READ|PROT_WRITE, €
    MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f28fc8bc000
close(3)
```

Note that Linux `uselib` call may also being used for that.

Even if pages are already loaded into *page cache*, they might be mapped with different addresses or with different permissions (which is less likely for shared objects), so operating system need to create new memory segments for them. It also performs it lazily and only maps limited amount of pages. The rest are mapped on demand when *minor fault* occur. That is why they occur often when new processes are spawned.

We will use `@count` aggregation and `vm.pagefault` probe to count pagefaults in SystemTap for Linux. Path to file is represented by `dentry` structure `$vma->vm_file->f_path->dentry` — we will use `d_name` function to access string representation of its name.

Source file: [scripts/stap/pfstat.stp](#)

Speaking of Solaris, we will need to intercept `as_segat()` calls. Segments that are related to files are handled by `segvn` segment driver, so we have to compare pointer to operations table to determine if segment is corresponding to that driver.

Source file: [scripts/dtrace/pfstat.d](#)

When you run these scripts, you may see that most of faults are corresponding to `libc` library.

Part 2

As you can see from exercise text, you'll need to get allocator cache names, but they are not described in documentation to SLAB probes. To find them we will need to dive into Solaris and Linux kernel source. We seek for a name (which would be represented as a string), so we have to find allocator cache structure and fields of type `char[]` or `char*` in it.

Let's check prototype of `kmem_cache_alloc()` function which is mentioned in probe description. First argument of it is a cache structure which we seek for. As you can see from source, it is called `kmem_cache_t`:

```
void *
kmem_cache_alloc(kmem_cache_t *cp, int kmflag)
    (from usr/src/uts/common/os/kmem.c)
```

This type is a typedefed alias for a struct `kmem_cache` which is defined in `usr/src/uts/common/sys/kmem_impl.h`. When you check its definition, you may find `cache_name` field — it is obvious that this field contains name of cache.

```
struct kmem_cache {
    [...]
    char          cache_name[KMEM_CACHE_NAMELEN + 1];
    [...]
}
```

Of course that is not the only way to find that field. If you know how to check cache statistics statically, you should know that is is done with `::kmastat` command in `mdb` debugger. Looking at its source will reveal `kmastat_cache()` helper which prints statistics for a cache. Looking at its source may reveal accesses to `cache_name` field:

```
mdb_printf((dfp++)->fmt, cp->cache_name);
    (from usr/src/cmd/mdb/common/modules/genunix/genunix.c).
```

Considering our findings, we may define `CACHE_NAME` macro and use it in a DTrace script:

Source file: [scripts/dtrace/kmemstat.d](#)

Seeking for a cache name in Linux is not that straightforward. First of all, we deal not with function boundary probes for SLAB caches, but with tapset aliases which do not provide pointers to a cache structures. Moreover, Linux has three implementations for kernel memory allocators: original SLAB, its improved version SLUB and a SLOB, compact allocator for embedded systems. SLOB is not generally used, so we will omit it.

`vm.kmem_cache_alloc` probe is defined in a `vm` tapset, so we will need to look into tapset directory `/usr/share/systemtap/tapset/linux/` to find its definition. As you can see from it, it refers either tracepoint `kmem_cache_alloc` or `kmem_cache_alloc()` kernel function. Here are source code of it:

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
{
    void *ret = slab_alloc(cachep, flags, _RET_IP_);

    trace_kmem_cache_alloc(_RET_IP_, ret,
                          cachep->object_size, cachep->size, flags);

    return ret;
}
    (from mm/slab.c)
```

Note the `trace_kmem_cache_alloc` call which is actually represents invocation of the FTrace tracepoint. The same function is defined in SLUB allocator, but it uses `s` as a name of first functions argument, so we will have to use `@choose_defined` construct. Both allocators use same name for cache's structure: `struct kmem_cache`. The name is defined in that structure in files `include/linux/slab_def.h` and `include/linux/slub_def.h`:

```
struct kmem_cache {
    [...]
    const char *name;
    [...]
}
```

Like in Solaris, Linux shows cache statistics in `/proc/slabinfo` file. Diving into Kernel sources will reveal that `cache_name` function is used to get cache's name and it accesses `name` field.

Considering all this, here are an implementation of SystemTap script:

Source file: [scripts/stap/kmemstat.stp](#)

Exercise 5

We will need to access pointer to `block_device` pointer in Linux to collect block device name and group data by it. As we know, main filesystem structure is called `super_block` which contains `s_bdev` pointer which has `struct block_device*`. SystemTap has two tapset functions, `MINOR()` and `MAJOR()` which allow to extract device number from `bd_dev` field of that structure. There is also an undocumented `bdevname()` function which is more convenient as it returns string, so we will use it.

We will attach probes to `vfs_write()` and `vfs_read()` functions to trace filesystem operations. First argument of that functions is pointer to file of type `struct file*`. Amount of data being written or read is passed through argument `$count`.

BIO level can be traced with `ioblock` tapset. We will do so by using its `ioblock.request` probe. It has following arguments: `bdev` — `block_device` pointer, `size` — amount of data in request, `rw` — read or write flag which can be tested for equality with `BIO_READ` or `BIO_WRITE` constants.

Here is resulting `deblock.stp` script:

Source file: [scripts/stap/deblock.stp](#)

To trace readahead from part 2 we will need to replace `vfs_write` to `vfs_read`, `BIO_WRITE` to `BIO_READ` and get rid from amount of data in request saving into aggregation by replacing it with number of requests (which will be constant 1).

We can use `scsi.ioentry` probe to trace SCSI operations. We can actually detect which command was used by parsing CDB buffer, but we will omit that and will trace all SCSI operations. Getting device name, however is not that easy: `request` structure which is used in SCSI stack refers to `gendisk` and `hd_struct` structures, but they won't refer to `block_device` (on contrary, `block_device` itself refers them). So we will make a small trick: there is a linked list of structures `bio ... biotail` which refer block device structure the same way they do in BIO probes, so we will simply copy approach from `ioblock.request` probe.

We will get `readahead.stp` script after applying all these modifications:

Source file: [scripts/stap/readahead.stp](#)

One can get device name with `ddi_pathname()` action or `devinfo_t` translator (which uses it indirectly) which is applied to `buf` structure. Probes from `io` provider will do it automatically by passing resulting pseudo-structure as `args[1]` argument. Aside from name, it contains minor and major device names.

Getting device name on VFS layer, however is harder: `vfs_t` structure which describes filesystem has only device number `vfs_dev`. ZFS makes things even harder: there is intermediate layer called pool which hides block devices from filesystem layer. So we will use mountpoints as an aggregation key. We will trace filesystem operations by attaching to `fop_read()` and `fop_write()` which accept pointers to `vnode_t` of file as their first argument and pointer to `uio` structure as their second argument (it describes user request and

thus contains amount of data).

Using all this we can get our `deblock.d` script:

Source file: [scripts/dtrace/deblock.d](#)

We will trace SCSI stack by attaching to `scsi-transport-dispatch` probe which will receive pointer to `buf` as first argument. That is very similar to probes from `io` provider except that probe doesn't apply translators on buffer.

Other changes in `readahead.d` are similar to those that was done for SystemTap:

Source file: [scripts/dtrace/readahead.d](#)

Exercise 6

This exercise is not so different than any [latency measurement](#) script where latency is measured as difference between timestamps of two probe firings and saved to an aggregation.

Note that `pthreadstat$` provider doesn't serve a probe for mutex lock attempt, so we had to expand it by using `pthread$` provider. As you may notice from `ustack` outputs in `pthread.d` example, mutex lock attempts are implemented by `mutex_lock_impl()` libc function. We will use `quantize()` aggregation which will be printed with `printf`:

Source file: [scripts/dtrace/mtxtime.d](#)

You will need to bind this script to `tsxperiment` process using `-c` or `-p` option.

We will need to use static probes `mutex_entry` and `mutex_acquired` for a SystemTap version of that script. However, we will need to be careful while working with userspace backtraces. First of all, we should use `-d` option to provide path to SystemTap for resolving symbols or `--ldd` to make it scan library dependencies of traced binary and automatically add them (when some of them are missing, `stap` utility will provide a hint with full paths).

Mutexes are also often used in TSLoad which can cause excessive overheads when we try to trace them, especially when we will use `ubacktrace()` function. You can use `STP_NO_OVERLOAD` macro definition (which can be passed to `stap` with `-D` option) to prevent `stap` from failing when overheads are big, or you can reduce overheads. In our case we will limit amount of traced callers by using `ucallers()` function which accepts depth of backtrace as a first argument like `ustack()` function from backtrace and only collects addresses without resolving them to symbols. We will defer symbol resolving to an aggregation printing.

Here are our script for SystemTap:

Source file: [scripts/stap/mtxtime.stp](#)

Exercise 7

Nature of solution of this exercise depends on your Apache and PHP configuration. In our case (as described in [lab description](#)), PHP was built as Apache HTTPD module (by using `--with-apxs2` option of `configure-script`), so all PHP code will be executed in a context of Apache worker, so we can safely use Thread-Local variables. If PHP was deployed with PHP-FPM, things would be more complicated.

So we will need to use `process-request-entry` probe to get URI of request and pair of probes `function-entry/function-entry` to measure execution time of a function. Since name of a method is passed in multiple probe arguments, we will have to use string concatenation. Like in many other exercises, we will use aggregations to collect statistics.

Note that you can use PHP probe `request-startup` instead of `process-request-entry`.

As you could remember from a [profiling](#) section of tracing principles, generally you shouldn't measure execution time of a function by tracing entry and exit points of it. However, PHP is an interpreted language, so it has lesser relative overheads of tracing because execution of its opcodes is slower than for the real processor (unless you are using some precompiler to machine language like HHVM) and we can afford full-tracing of it.

Here are resulting implementations of scripts for SystemTap and DTrace:

Source file: [scripts/stap/topphp.stp](#)

Source file: [scripts/dtrace/topphp.d](#)

Appendix B. Lab setup

Setting up Operating Systems

Originally we used Virtual Machines for Oracle VirtualBox with Solaris 11.0 and CentOS 6.4. Unfortunately, these versions become stale, while VirtualBox is a second-level hypervisor which complicates performance analysis experiments.

Actual version of this book was modified to support Solaris 11.2 and CentOS 7.0. They were installed in a Xen 4.4 environment in HVM machines. I assume that you were installed these operating systems and already performed basic setup like setting IP address or root password.

Setting up CentOS 7

- You will need debuginfo packages to access debug information. They are located in separate CentOS repository which you will need to turn on:

```
# sed -i 's/^enabled=0/enabled=1/g' /etc/yum.repos.d/CentOS-Debuginfo.repo
```

Warning

CentOS 7.0 contains incorrect GPG key for debuginfo repository like described in [bug 7516](#), so you will also need to update centos-release package:

```
# yum install centos-release
```

- Install SystemTap

```
# yum install systemtap systemtap-sdt-devel systemtap-client
```

- Run stap-prep script. That script will install packages that are needed for building kernel modules and debuginfo packages:

```
# stap-prep
```

Note

kernel-debuginfo may be installed manually using YUM package manager. In that case, however, you should add precise version of kernel to a package name. Otherwise YUM will install newest version that probably wouldn't match kernel you running.

- Install debuginfo-install utility:

```
# yum install yum-utils
```

- Install debug information for userspace programs:

```
# debuginfo-install cat python
```

- Change /tmp mount point to tmpfs. To do that, add following line to /etc/fstab file:

```
tmpfs          /tmp          tmpfs          defaults      0 0
```

- After that clean up /tmp and run mount -a command.
- Building TSLoad workload generator and its modules
 - Install SCons

```
# yum install wget
# cd /tmp
# wget 'http://prdownloads.sourceforge.net/scons/scons-2.3.4-1.noarch.rpm'
# rpm -i scons-2.3.4-1.noarch.rpm
```

- Install development files:

```
# yum install libuuid-devel libcurl-devel
```

- Build a workload generator:

```
# git clone https://github.com/myaut/tsload
# cd tsload/agent
# scons --prefix=/opt/tsload install
```

- Build loadable modules:

```
# git clone https://bitbucket.org/sergey_klyaus/dtrace-stap-book.git
# cd dtrace-stap-book/tsload
# scons --with-tsload=/opt/tsload/share/tsload/devel install
```

- Install OpenJDK7:

```
# yum install java-1.7.0-openjdk-devel.x86_64
```

Setting up Solaris 11.2

- Building TSLoad workload generator and its modules

- Install SCons

```
# wget 'http://prdownloads.sourceforge.net/scons/scons-2.3.4.tar.gz'
# tar xzvf scons-2.3.4.tar.gz
# cd scons-2.3.4/
# python setup.py install
```

- Install development files:

```
# pkg install pkg:/developer/gcc-45
# pkg install pkg:/developer/build/ondld
```

- Build a workload generator:

```
# git clone https://github.com/myaut/tsload
# cd tsload/agent
# scons --prefix=/opt/tsload install
```

- Build loadable modules:

```
# git clone https://bitbucket.org/sergey_klyaus/dtrace-stap-book.git
# cd dtrace-stap-book/tsload
# scons --with-tsload=/opt/tsload/share/tsload/devel install
```

- Install JDK7:

```
# pkg install --accept pkg:/developer/java/jdk-7
```

iSCSI

We will need to use SCSI device so we can fully trace it in [exercise 5](#). Xen hypervisor supports SCSI emulation, but only by emulating outdated *LSI 53c895a* controller which is not supported by Solaris. However, we can create iSCSI devices in Dom0 and supply them to virtual machines. The following guide is created for Debian 7 which uses *iSCSI Enterprise Target*. Recent Linux kernels replaced it with *LIO* stack.

- Install IET packages:

```
# aptitude install iscsi target iscsi target-dkms
```

- Create logical disks for virtual machines. They would be LVM volumes `/dev/mapper/vgmain-sol11--base--l ab` and `/dev/mapper/vgmain-centos7--base--l ab` in our example.

- Create targets in `/etc/iet/ietd.conf` file by adding following lines:

```
Target iqn.2154-04.tdc.r520:storage.l ab5-sol11-base
    Lun 0 Path=/dev/mapper/vgmain-sol11--base--l ab, Type=blockio
```

```
Target iqn.2154-04.tdc.r520:storage.l ab5-centos7-base
    Lun 0 Path=/dev/mapper/vgmain-centos7--base--l ab, Type=blockio
```

- Note that target names should match DNS name of a host which provides them.
- Configure `/etc/iet/initiators.allow` file to forbid Solaris access to disk allocated for CentOS machine and vice versa. Delete or comment out ALL `ALL` line and add lines with target names and IP addresses of corresponding machines:

```
iqn.2154-04.tdc.r520:storage.l ab5-sol11-base      192.168.50.179
iqn.2154-04.tdc.r520:storage.l ab5-centos7-base    192.168.50.171
```

- Restart IET daemon:

```
# /etc/init.d/iscsi target restart
```

- Configure Solaris initiator. `192.168.50.116` is an IP address of our Dom0 system which provides iSCSI targets.

```
# iscsiadm add discovery-address 192.168.50.116
# iscsiadm modify discovery -t enable
# svcadm restart svc:/network/iscsi/initiator:default
```

- Similarly configure CentOS initiator:

```
# yum install iscsi-initiator-utils
# systemctl enable iscsid
# systemctl start iscsid
# iscsiadm -m discovery -t sendtargets -p 192.168.50.116
# iscsiadm -m node --login
```

Web application stack

We will need to setup web stack to complete [exercise 7](#). We will use following applications: web server *Apache HTTPD 2.4*, relational database *MySQL Community Edition 5.6* and PHP interpreter *PHP 5.6*, and content management system *Drupal 7* on top of them. This guide can be used to setup both CentOS 7 and Solaris 11 except for few commands (they will be marked). We will need to build these programs from source codes to enable USDT probes in them.

DANGER!

This guide is intended for lab setup. It could lack security, so do not use it for production systems.

Download programs sources

- Download sources with wget:

```
# wget http://us3.php.net/distributions/php-5.6.10.tar.bz2
# wget http://cdn.mysql.com/Downloads/MySQL-5.6/mysql-5.6.25.tar.gz
# wget http://archive.apache.org/dist/httpd/httpd-2.4.9.tar.gz
# wget http://archive.apache.org/dist/httpd/httpd-2.4.9-deps.tar.gz
```

- We will also need patch for Apache HTTPD build system:

```
# wget -O dtrace.patch https://bz.apache.org/bugzilla/attachment.cgi?id=31665
```

• (*Only CentOS7*) Remove programs that were installed from package repositories and install some dependencies:

```
# yum erase php mysql mysql-server httpd
# yum install libxml2-devel bzip2
```

• (*Only Solaris*) We will have to use GNU Make for all builds, so make an alias to maintain guide uniformity:

```
# alias make=gmake
```

- Unpack downloaded archives

```
# tar xzvf httpd-2.4.9.tar.gz
# tar xzvf mysql-5.6.25.tar.gz
# tar xjvf php-5.6.10.tar.bz2
# tar xzvf httpd-2.4.9-deps.tar.gz
```

Build and install MySQL from sources

- Change current directory to one with sources:

```
# cd mysql-5.6.25
```

- (*Only CentOS7*) Install dependencies:

```
# yum install cmake bison ncurses-devel gcc-c++
```

- (*Only Solaris*) Install dependencies:

```
# pkg install pkg:/developer/buildd/cmake
# pkg install pkg:/developer/parser/bison
```

- Build and install MySQL (it would be installed into `/usr/local/mysql`)

```
# cmake --enable-dtrace .
# make -j 4
# make install
```

Build and install Apache HTTPD from sources

- Change current directory to one with sources:

```
# cd ../httpd-2.4.9
```

- (*Only CentOS7*) Install dependencies:

```
# yum install pcre-devel autoconf flex patch
```

- (*Only Solaris*) Install dependencies:

```
# pkg install pkg: /developer/build/autoconf
# pkg install pkg: /developer/macro/gnu-m4
# pkg install pkg: /developer/lexer/flex
```

- Apply patch to a build system and recreate configure script:

```
# patch -p0
```

- Create file for building MPM:

```
# (cd server/mpm/event/
  echo "$ (pwd)/event.o $(pwd)/fdqueue.o" > libevent.objects)
```

- Fix server/Makefile.in file:

```
# sed -i 's/apache_probes.h/"apache_.*probes.h"/' server/Makefile.in
```

- Build and install Apache HTTPD (it would be installed into /usr/local/apache2):

```
# ./configure --with-included-apr --enable-dtrace
# make -j 4
# make install
```

Build and install PHP interpreter from sources

- Change current directory to one with sources:

```
# cd php-5.6.10
```

- (*Only CentOS7*) Install dependencies:

```
# yum install libjpeg-turbo-devel libpng12-devel
```

- (*Only Solaris*) Install dependencies:

```
# pkg install pkg: /system/library/gcc-45-runtime
```

- Build and install PHP:

```
# ./configure --enable-debug --enable-dtrace --with-mysql \
  --with-apxs2=/usr/local/apache2/bin/apxs --without-sqlite3 \
  --without-pdo-sqlite --with-iconv-dir --with-jpeg-dir \
  --with-gd --with-pdo-mysql
# make -j 4
# make install
```

Setup MySQL database

- Change directory to a MySQL root directory:

```
# cd /usr/local/mysql
```

- (*Only CentOS7*) Create mysql user:

```
# groupadd -g 666 mysql
# useradd -u 666 -g mysql -d /usr/local/mysql/home/ -m mysql
```

- Create data files:

```
# chown -R mysql:mysql data/
# ./scripts/mysql_install_db
```

- Create /etc/my.cnf configuration file:

```
# cat > /etc/my.cnf
```

- Create log file and directory for PID file:

```
# touch /var/log/mysql.log chown mysql:mysql /var/log/mysql.log
# mkdir /var/run/mysql d/ chown mysql:mysql /var/run/mysql d/
```

- Fill system tables:

```
# chown -R mysql:mysql data/
# ./scripts/mysql_install_db --ldata=/usr/local/mysql/data
```

- Start mysqld daemon:

```
# ./support-files/mysql.server start
```

- Set MySQL root password to changeme:

```
# /usr/local/mysql/bin/mysqladmin -u root password changeme
```

- Create drupal user in MySQL:

```
# /usr/local/mysql/bin/mysql --user=root -p mysql -h localhost
Enter password: changeme
mysql> CREATE USER 'drupal'@'localhost' IDENTIFIED BY 'password';
mysql> GRANT ALL PRIVILEGES ON *.* TO 'drupal'@'localhost';
mysql> FLUSH PRIVILEGES;
```

Setup Apache and PHP interpreter

- (*Only CentOS*) Disable firewall:

```
# systemctl disable firewall
# systemctl stop firewall
```

- Change directory to HTTPD root directory:

```
# cd /usr/local/apache2
```

- Make a backup copy of configuration file and add PHP 5 support to it (you will have to use gsed instead of sed in Solaris):

```
# cp conf/httpd.conf conf/httpd.conf.orig
# sed -re 's/^(\\s*DirectoryIndex. *)/\\1 index.php/' conf/httpd.conf.orig > €
    conf/httpd.conf
# cat >> conf/httpd.conf
```

- Start Apache HTTPD:

```
# ./bin/httpd
```

Install Drupal 7

- Change to a directory with web documents:

```
# cd /usr/local/apache2/htdocs
```

- Download and unpack Drupal 7:

```
# cd /tmp
# wget http://ftp.drupal.org/files/projects/drupal-7.38.zip
# cd /usr/local/apache2/htdocs/
# unzip /tmp/drupal-7.38.zip
# mv drupal-7.38/ drupal/
# chown -R daemon:daemon .
```

- Create drupal database:

```
# /usr/local/mysql/bin/mysql --user=drupal -p -h localhost
Enter password: password
mysql> CREATE DATABASE drupal;
```

- Enter `http://SERVER_ADDRESS/drupal/install.php` in web browser and follow Drupal installer instructions. Use following parameters when setting up database:
 - Database name: drupal
 - Database username: drupal

- Database password: password

Install Drupal module devel and setup test data

- Download and install it:

```
# wget -O /tmp/devel-7.x-1.5.tar.gz €  
    http://ftp.drupal.org/files/projects/devel-7.x-1.5.tar.gz  
# cd /usr/local/apache2/htdocs/drupal/modules  
# tar xzvf /tmp/devel-7.x-1.5.tar.gz
```

- Access index page of Drupal, choose *Modules* in top-level menu, and check *Devel* and *Devel generate* in the shown list, then click on *Save Configuration*.

- After enabling modules, choose *Configure* for *Devel generate* module, and choose *Generate content* in a menu, pick option *Article* and click *Generate*. 50 test pages should appear at index page.

Note

To start services, use following commands:

```
# /usr/local/mysql/support-files/mysql.server start  
# /usr/local/apache2/bin/httpd
```

Appendix C. Cheatsheet

Tools

	<i>DTrace</i>	<i>SystemTap</i>
Tool	<code>dtrace(1M)</code>	<code>stap(1)</code>
List probes	<code># dtrace -l</code> <code># dtrace -l -P io</code>	<code># stap -l 'ioblock.*'</code> <code># stap -L 'ioblock.*'</code>
One-liner	<code># dtrace -n 'syscall::read:entry { trace(arg1); } '</code>	<code># stap -e 'probe syscall.read { println(fd); } '</code>
Script	<code># dtrace -s script.d</code> (optionally add <code>-C</code> for preprocessor, <code>-q</code> for quiet mode)	<code># stap script.stp</code>
Custom probe	<code># dtrace -P io -n start</code>	-
Integer arguments	<code># dtrace -n 'syscall::read:entry / cpu == \$1 / ' 0</code>	<code># stap -e 'probe syscall.read { if(cpu() != \$1) next; println(fd); } ' 0</code>
String arguments	<code># dtrace -n 'syscall::read:entry / execname == \$1 / ' '"cat"'</code>	<code># stap -e 'probe syscall.read { if(execname() == @1) println(fd); } ' cat</code>
Guru/destructive mode (!)	<code># dtrace -w ...</code>	<code># stap -g ...</code>
Redirect to file	<code># dtrace -o FILE ...</code> (appends)	<code># stap -o FILE ...</code> (rewrites)
Tracing process	<code># dtrace -n 'syscall::read:entry / pid == \$target / { ... }' -c 'cat /etc/motd'</code> (or <code>-p PID</code>)	<code># stap -e 'probe syscall.read { if(pid() == target()) ... }' -c 'cat /etc/motd'</code> (or <code>-x PID</code>)

Probe names

	<i>DTrace</i>	<i>SystemTap</i>
Begin/end	<code>dtrace::BEGIN, dtrace::END</code>	<code>begin, end</code>
<code>foo()</code> entry	<code>fbt::foo:entry</code>	<code>kernel.function("foo") module("mod").function("foo")</code>
<code>foo()</code> return	<code>fbt::foo:return</code>	<code>kernel.function("foo").return</code>
Wildcards	<code>fbt::foo*:entry</code>	<code>kernel.function("foo*")</code>
Static probe mark	<code>sdt::mark</code>	<code>kernel.trace("mark")</code>
System call	<code>syscall::read:entry</code>	<code>syscall.read</code>
Timer once per second	<code>tick-1s</code>	<code>timer.s(1)</code>
Profiling	<code>profile-997hz</code>	<code>timer.profile(), perf.*</code>
<code>read()</code> from libc	<code>pid\$target:libc:read:entry</code> Traces process with <code>pid == \$target</code>	<code>process("/lib64/libc.so.6").function("read")</code> Traces any process that loads libc

In DTrace parts of probe name may be omitted: `fbt::foo:entry -> foo:entry`
 Units for timer probes: ns, us, ms, s, hz, jiffies (SystemTap), m, h, d (all three - DTrace)

Printing

	<i>DTrace</i>	<i>SystemTap</i>
Value	<code>trace(v)</code>	<code>print(v)</code>
Value + newline	-	<code>println(v)</code>
Delimited values	-	<code>printf(", ", v1, v2)</code> <code>printf(" ", v1, v2)</code>
Memory dump	<code>tracemem(ptr, 16)</code>	<code>printf("%16M", ptr)</code>
Formatted	<code>printf("%s", str)</code>	
Backtrace	<code>ustack(n)</code> <code>ustack()</code>	<code>print_ubacktrace()</code> <code>print_ustack(ubacktrace())</code>
Symbol	<code>usym(addr)</code> <code>ufunc(addr)</code> <code>uaddr(addr)</code>	<code>print(usymname(addr))</code> <code>print(usymdata(addr))</code>

If *u* prefix is specified, userspace symbols and backtraces are printed, if not — kernel symbols are used

String operations

<i>Operation</i>	<i>DTrace</i>	<i>SystemTap_</i>
Get from kernel	<code>stringof(expr)</code> <code>(string) expr</code>	<code>kernel_string*()</code>
Convert scalar		<code>sprint()</code> and <code>sprintf()</code>
Copy from user	<code>copyinstr()</code>	<code>user_string*()</code>
Compare	<code>==, !=, >, >=, ,</code>	
Concat	<code>strjoin(str1, str2)</code>	<code>str1 . str2</code>
Get length	<code>strlen(str)</code>	
Check for substring	<code>strstr(haystack, needle)</code>	<code>isinstr(haystack, needle)</code>

Aggregations

<i>Time source</i>	<i>DTrace</i>	<i>SystemTap</i>
Add value	<code>@aggr[keys] = func(value);</code>	<code>aggr[keys]</code>
Printing	<code>printa(@aggr);</code> <code>printa("format string", @aggr);</code>	<code>foreach([keys] in aggr) {</code> <code>print(keys, @func(aggr[keys]));</code> }
Clear	<code>clear(@aggr);</code> or <code>trunc(@aggr);</code>	<code>delete aggr;</code>
Normalization by 1000	<code>normalize(@aggr, 1000);</code> <code>denormalize(@aggr);</code>	<code>@func(aggr) / 1000</code> in printing
Select 20 values	<code>trunc(@aggr, 20);</code>	<code>foreach([keys] in aggr limit 20) {</code> <code>print(keys, @func(aggr[keys]));</code> }
Histograms (linear in [10;100] with step 5 and logarithmical)	<code>@lin = lquantize(value, 10, 100, 5);</code> <code>@log = quantize(value);</code> ... <code>printa(@lin);</code> <code>printa(@log);</code>	<code>aggr</code>

Where *func* is one of count, sum, min, max, avg, stddev

Context variables

<i>Description</i>	<i>DTrace</i>	<i>SystemTap</i>
Thread	<code>curthread</code>	<code>task_current()</code>
Thread ID	<code>tid</code>	<code>tid()</code>
PID	<code>pid</code>	<code>pid()</code>
Parent PID	<code>ppid</code>	<code>ppid()</code>
User/group ID	<code>uid/gid</code>	<code>uid()/gid()</code> <code>euid()/egid()</code>
Executable name	<code>execname</code> <code>curpsinfo->ps_fname</code>	<code>execname()</code>
Command line	<code>curpsinfo->ps_psargs</code>	<code>cmdline_*</code>
CPU number	<code>cpu</code>	<code>cpu()</code>
Probe names	<code>probedprov</code> <code>probedmod</code> <code>probedfunc</code> <code>probename</code>	<code>pp()</code> <code>pn()</code> <code>ppfunc()</code> <code>probedfunc()</code> <code>probedmod()</code>

Time

<i>Time source</i>	<i>DTrace</i>	<i>SystemTap</i>
System timer	<code>^lbolt</code> <code>^lbolt64</code>	<code>jiffies()</code>
CPU cycles	-	<code>get_cycles()</code>
Monotonic time	<code>timestamp</code>	<code>local_clock_unit()</code> <code>cpu_clock_unit(cpu)</code>
CPU time of thread	<code>vtimestamp</code>	-
Real time	<code>walltimestamp</code>	<code>gettimeofday_unit()</code>

Where *unit* is one of s, ms, us, ns

Process management

SystemTap

Getting task_struct pointers:

- task_current() – current task_struct
- task_parent(t) – parent of task t
- pid2task(pid) – task_struct by pid

Working with task_struct pointers:

- task_pid(t) task_tid(t)
- task_state(t) – 0 (running), 1-2 (blocked)
- task_execname(t)

DTrace

kthread_t* curthread fields:

- t_tid, t_pri, t_start, t_pctcpu

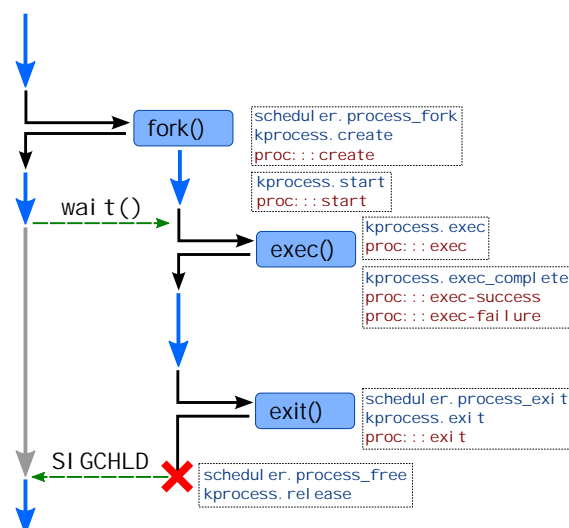
psinfo_t* curpsinfo fields:

- pr_pid, pr_uid, pr_gid, pr_fname, pr_psargs, pr_start

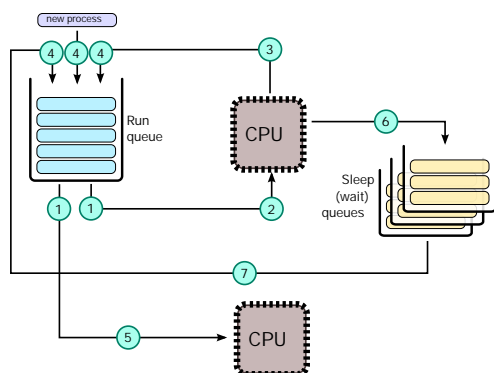
lwpsinfo_t* curlwpsinfo fields:

- pr_lwpid, pr_state/pr_sname

psinfo_t* and lwpsinfo_t* are passed to some proc:: probes



Scheduler



	<i>DTrace</i>	<i>SystemTap</i>
1	sched::dequeue	kernel.function("dequeue_task")
2	sched::on-cpu	scheduler.cpu_on
3	sched::off-cpu	scheduler.cpu_off
4	sched::enqueue	kernel.function("enqueue_task")
5	-	scheduler.migrate
6	sched::sleep	-
7	sched::wakeup	scheduler.wakeup

Virtual memory

Probes

SystemTap

- vm.brk – allocating heap
- vm.mmap – allocating anon memory
- vm.munmap – freeing anon memory

DTrace

- as_map:entry – allocating proc mem
- as_unmap:entry – freeing proc mem

Page faults

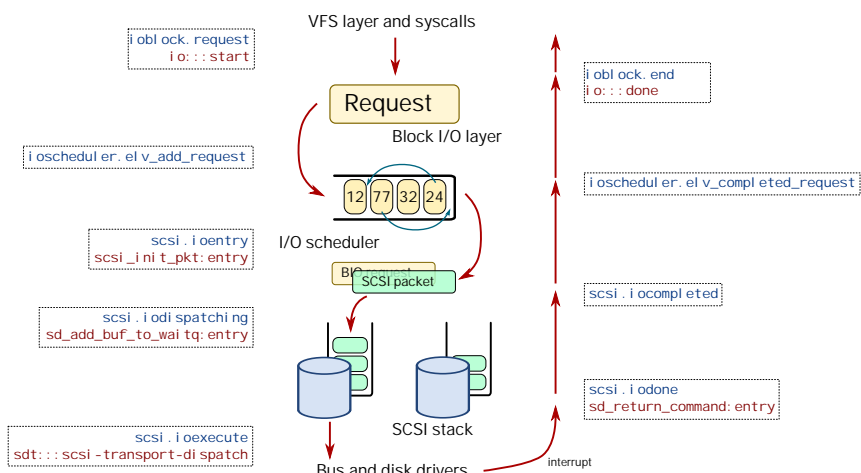
Type	<i>DTrace</i>	<i>SystemTap</i>
Any	vminfo::as_fault	vm.pagefault vm.pagefault.return perf.sw.page_faults
Minor		perf.sw.page_faults_min
Major	vminfo::maj_fault	perf.sw.page_faults_maj
CoW	vminfo::cow_fault	
Protection	vminfo::prot_fault	

Block Input-Output

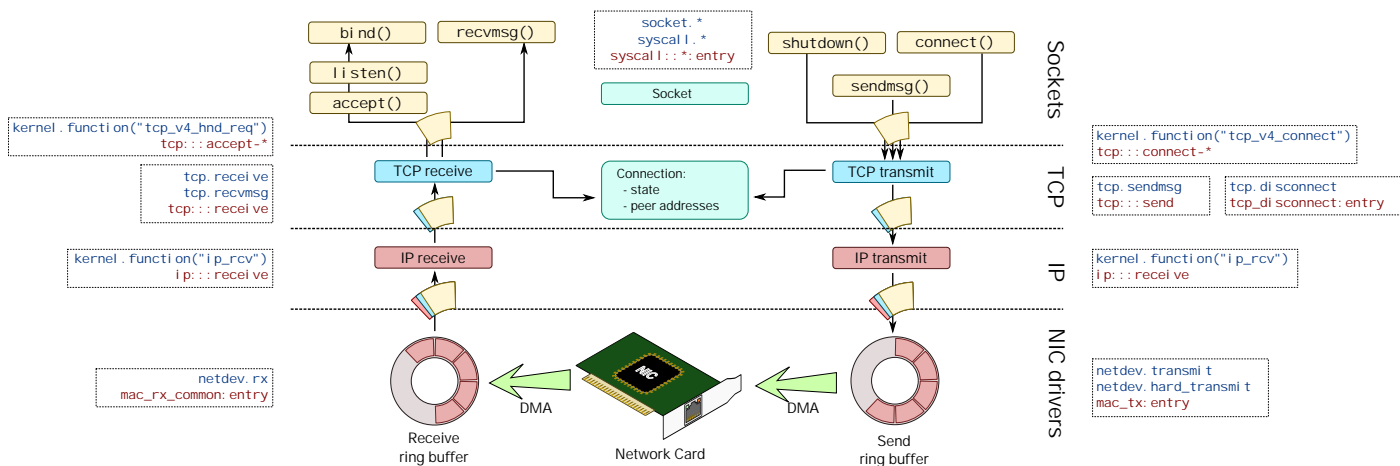
Block request structure fields:

Field	bufinfo_t struct buf	struct bio
Flags	b_flags	bi_flags
R/W	b_flags	bi_rw
Size	b_bcount	bi_size
Block	b_blkno b_lblkno	bi_sector
Callback	b_iodone	bi_end_io
Device	b_edev b_dip	bi_bdev

* flags B_WRITE, B_READ



Network stack



Non-native languages

Function call	DTrace	SystemTap
Java*	method-entry <ul style="list-style-type: none"> arg0 — internal JVM thread's identifier arg1:arg2 — class name arg3:arg4 — method name arg5:arg6 — method signature 	hotspot.method_entry <ul style="list-style-type: none"> thread_id — internal JVM thread's identifier class — class name method — method name sig — method signature
Perl	perl\$target::sub-entry <ul style="list-style-type: none"> arg0 — subroutine name arg1 — source file name arg2 — line number 	process("...").mark("sub__entry") <ul style="list-style-type: none"> \$arg1 — subroutine name \$arg2 — source file name \$arg3 — line number
Python	python\$target::function-entry <ul style="list-style-type: none"> arg0 — source file name arg1 — function name 	python.function.entry <ul style="list-style-type: none"> \$arg1 — source file name \$arg2 — function name
PHP	function-entry <ul style="list-style-type: none"> arg0 — function name arg1 — file name arg2 — line number arg3 — class name arg4 — scope operator :: 	process("...").mark("function__entry") <ul style="list-style-type: none"> \$arg1 — function name \$arg2 — file name \$arg3 — line number \$arg4 — class name \$arg5 — scope operator ::

*requires -XX: +DTraceMethodProbes

Index

A

address space, 39
aggregation, 44
anonymous memory, 85
arguments of probe, 31
associative array, 43
asynchronicity, 102

B

backtrace, 52
block input-output, 98
bottom half, 114
buffer, 15

C

call tree, 54
CGroup (Linux), 80
Completely Fair Scheduler (Linux).
condition variable (Solaris), 112
conditional compilation, 22
consumer, 15
context function, 33
context of probe, 33

context switch, 67
cpc (provider, DTrace), 58
current pointer (Linux), 68
curthread (Solaris), 69

D

Deadman mechanism, 20
debuginfo packages (Linux), 17
dentry cache (Linux), 96
destructive actions, 16
Directory Name Lookup Cache (Solaris), 96
dtrace(1M), 16
dumping memory, 46
dynamic program analysis, 51
dynamic tracing, 15

E

Embedded C (SystemTap), 24
epilogue probe alias, 48

F

file system, 91
fsflush (Solaris), 102
fsinfo (provider, DTrace), 95

function boundary tracing, 26

functions (SystemTap), 24

H

hotspot (provider, DTrace), 122

hotspot (tapset, SystemTap), 122

I

interrupt, 113

io (provider, DTrace), 98

ioblock (tapset, SystemTap), 98

ioscheduler (tapset, SystemTap), 99

ip (provider, Solaris), 109

J

Java Statically Defined Tracing, 125

Java Virtual Machine, 120

K

kernel (external) variables, 36

kprocess (tapset, SystemTap), 71

L

latency, 60

lock, 109

lockstat (provider, DTrace), 111

M

MAXMAPENTRIES (constant), 20

memory allocator, 89

missing probe, 22

monotonic time, 45

mutex, 109

N

network stack, 105

O

open file table, 92

P

page cache, 97

page fault, 87

perf (tapset, SystemTap), 59

performance analysis, 60

pid\$\$ (provider, DTrace), 115

pointer, 39

predicate, 33

printing, 46

probe, 25

probe overhead threshold, 20

proc (provider, DTrace), 71

process, 67
process, spawning, 70
processes, grabbing PID, 34
processor performance counter, 58
profiling, 56
prologue probe alias, 48
provider (DTrace), 26

R

request extraction, 103
return value, 31
ring buffer, 106
runtime (SystemTap), 18

S

sched (provider, DTrace), 74
scheduler, 74
scheduler (tapset, SystemTap), 74
script, 23
scsi (tapset, SystemTap), 99
SCSI stack, 99
segments, 84
socket, 105
speculations, 47

stack, 52
stap(1), 19
statically defined tracing, 28
STREAMS (Solaris), 108
string operations, 41
structure field access, 41
symbols, 53

T

tapset, 22, 48
task (Linux), 67
taskqueue (Solaris), 114
tcp (provider, Solaris), 109
thread-local variables, 38
throughput, 60
timer probes, 29
translators (DTrace), 48
try/catch block (SystemTap), 24
TSLoad workload generator, 10

U

userspace process tracing, 115

V

variable scopes, 36

variable types, 35

virtual memory, 83

vm (tapset, SystemTap), 85

W

wait queue (Linux), 112

wall-clock time, 45

workqueue (Linux), 114

writeback (Linux), 102