tuneit

Инструменты динамической трассировки

DTrace и SystemTap

Автор курса: Кляус С.М.

Оглавление

Список листингов

Скрипты SystemTap

Скрипты DTrace

Вспомогательные скрипты

Примеры программ

Введение

От автора

Мое отношение к задаче анализа кода как ключевого шага на пути к решению программных проблем: зависаний программ, их аварийному завершению, чрезмерному (или неоправданно маленькому) потреблению вычислительных ресурсов сформировалось вовремя работы над выпускной квалификационной работой. Она была посвящена архитектуре микроядерных операционных систем, и столкнувшись с недостаточной ее документированностью, мне пришлось обратиться к исходным кодам ядер.

После этого, я стал применять чтение кода и в своей профессиональной деятельности, так как код всегда содержит более актуальную и полноценную информацию чем документация, или лучше объяснит причину ошибки, чем сообщение о ней. Так, обратимся к документации, посвященной файловой системе UFS в операционной системе Solaris:
-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.
http://docs.oracle.com/cd/E23823_01/html/816-5166/newfs-1m.html

На самом деле условие, которое описывает ограничения на размер блока в файловой системе UFS немного сложнее:

```
928    if (fsp->fs_bsize > MAXBSIZE || fsp->fs_frag > MAXFRAG ||
929        fsp->fs_bsize < sizeof (struct fs) ||
                       fsp->fs_bsize < PAGESIZE) {
930           error = EINVAL; /* also needs translation */
931           goto out;
932    }
```

http://fxr.watson.org/fxr/source/common/fs/ufs/ufs_vfsops.c?v=OPENSOLARIS#L928

Итак, более точное условие, распространяющееся на все архитектуры можно записать так: размер блока должен быть больше или равен размеру страницы, но не превышать 8192 байта (макрос MAXBSIZE), и также вмещать суперблок. Вынужден признать, что иногда, я слишком тороплюсь обращаться к исходным кодам, даже когда документация достаточно подробно и качественно описывает проблему, но в большинстве случаев мой подход окупался.

Метод, при котором информация извлекается из исходного кода программы называется *статическим анализом кода*. Однако этот метод не самодостаточен, так как невозможно рассматривать высокоуниверсальную систему, такую как операционную систему Solaris, в отрыве от того, какие запросы она обрабатывает – иначе бы мы были вынуждены рассматривать все возможные случаи (например, все ветвления в коде), и трудоемкость *статического анализа кода* неимоверно бы возросла. По этой причине, необходимо периодически проводить

экспериментальные запуски системы и выполнять *динамический анализ кода*, отсекая таким образом незадействованные ветки программного кода и также уточная структуру программы.

Во время работы над выпускной работой, я использовал эмулятор Bochs, который мог генерировать гигантские трассы: по одной строке на каждую выполненную инструкцию. По счастью, современные операционные системы имеют куда более гибкие инструменты: *среды динамической трассировки*, о которых и пойдет речь в этом пособии.

Первый полноценный скрипт DTrace я написал для заявки, связанной со следующей паникой ядра Solaris:

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

Как видно из сообщения, счетчик ссылок уменьшился лишний раз (например, из-за попытки закрыть файл дважды). Конечно, если вы попытаетесь вызвать close() последовательно в рамках одного процесса, это не приведет к панике системы, а значит мы здесь имеем дело с каким-то специфичным состоянием гонок, или ошибкой, допущенной разработчиками ядра. Выявить эту проблему можно, собрав трассу для системного вызова close(), попутно отслеживая все вызовы vn_rele, уменьшающие счетчик ссылок, для чего я и написал DTrace-скрипт.

Постепенно знакомясь с Linux, я добавил к знанию DTrace аналогичную среду для этой операционной системы: SystemTap и в 2011-м году начал готовить небольшой семинар по этим двум системам. Кроме этого, я дополнял слайды для семинара комментариями. Количество информации разрасталось, и к июню были уже готовы вводный раздел, разделы, посвященный языкам динамической трассировки, и описание управления процессами в Linux и Solaris, а также соответствующие скрипты dumptask.* Высокие временные затраты как на чтение кода, так и на подготову скриптов отпугнули меня, и мне показалось невозможным качественно раскрыть остальные темы, связанные с внутренней архитектурой ядер операционных систем Linux и Solaris, поэтому проект был положен в долгий ящик.

Вернулся я к написанию пособия в 2013-м году: в это время я активно занимался изучением планировщика CFS в Linux, и следующая за темой “Управление процессами”: “Планировщик ядра” была подготовлена куда быстрее. К тому же я накопил немало опыта, связанного с дисковой подсистемой, когда работал с файловой системой ZFS. Наконец, моя заинтересованность в производительности веб-приложений помогла мне при написании пятого раздела, и в конце 2013-го года была готова черновая версия пособия. Внесение правок в нее, и определение дальнейшей судьбы пособия, к сожалению, растянулось еще более чем на год.

В 2011-м году вышла книга двух больших специалистов в области внутренней архитектуры Solaris – Джима Мауро (Jim Mauro) – и DTrace – Брендана Грегга (Brendan Gregg) “DTrace Dynamic Tracing in Oracle® Solaris, Mac OS X, and FreeBSD”. Несмотря на несравнимо более высокий уровень авторов, чем мой, и куда большее количество информации, содержащееся в ней (ее объем превышает

тысячу страниц), я надеюсь, что не только дополнение в виде Linux и SystemTap привлечет вас к ознакомлению с моим пособием. В отличие от этой книги, в которой вы найдете общие сведения о принципах работы ядра и ряд готовых скриптов-однострочников (one-liners) или вызовов скриптов из DTraceToolkit, в данном пособии большее внимание уделялось внутренней архитектуре ядер и приложений.

Из-за непростого пути подготовки пособия, в него закралось немало нелогичностей. Так, изначально не задуманное, как документация или полноценное пособие, первые два раздела имели лишь краткие комментарии к слайдам, тогда как некоторые из пунктов из разделов 4 и 5 содержат подробные списки проб с их аргументами. К тому же, за 4 года, в течении которых готовилось пособие среда SystemTap претерпела значительные изменения, а Solaris превратился в операционную систему с закрытыми исходными кодами, поэтому некоторые нюансы могут быть пропущены в пособии. Я постарался скрасить эти недостатки при подготовке финальной версии, но даже под самой красивой штукатуркой можно разглядеть кривые стены.

Присылайте пожалуйста свои отзывы, замеченные ошибки и неточности на myautneko+dtrace_stap@gmail.com

Принятые обозначения

Книга представляет из себя слайды к одноименному мастер-классу, каждый из которых дополнен большим количеством комментариев, справочной информации и примеров. Гиперссылки в тексте помечены подчеркиванием: http://www.tune-it.ru/. Рисунки в тексте не подписаны и следуют сразу за ссылающимся на них абзацем текста.

Определение какого-либо термина выделено в тексте символом горящей лампочки, а сам термин - курсивом.

Вспомогательные сведения выдена значком «информация» в кружочке. Они содержат базовую информацию о каком-то принципе организации вычислительных систем и могут быть опущены

Замечание предупреждает об особенностях применения того или иного функционала DTrace и SystemTap или дополняет уже имеющуюся информацию и выделено восклицательным знаком в треугольнике.

Предупреждение сообщает о том, что некоторые действия могут иметь фатальные последствия для корректного функционирования системы.

Текст, выделенный моноширинным шрифтом, представляет из себя простой

пример, как правило сводящийся к вызову одной команды или ее ожидаемого вывода на терминал:

```
# dd if=/dev/zero of=/dev/null &
```

Абзац, набранный мелким моноширинным шрифтом, и отделенный двумя горизонтальными линиями — это полноценный листинг программы:

Листинг 0. Скрипт hellouser.py

```
import os
print "Hello, " + os.getenv('LOGNAME', 'anonymous') + '!'
```

Нагрузчик TSLoad

В процессе курса нам потребуется продемонстрировать разработанные скрипты на реальной системе. Для этого, в ходе выполнения упражнений или запуска примеров, мы воспользуемся нагрузчиком TSLoad версии 0.2. Документация по генератору нагрузки доступна здесь: http://myaut.github.io/tsload/, а исходные коды — в соответствующем GitHub-репозитории: https://github.com/myaut/tsload.

Чтобы создать новый эксперимент в нагрузчике, нужно создать файл в формате JSON, назвать его experiment.json и поместить в пустую директорию. Этот файл содержит описание нагрузок: тип нагрузки и ее параметры, а также пулов потоков.

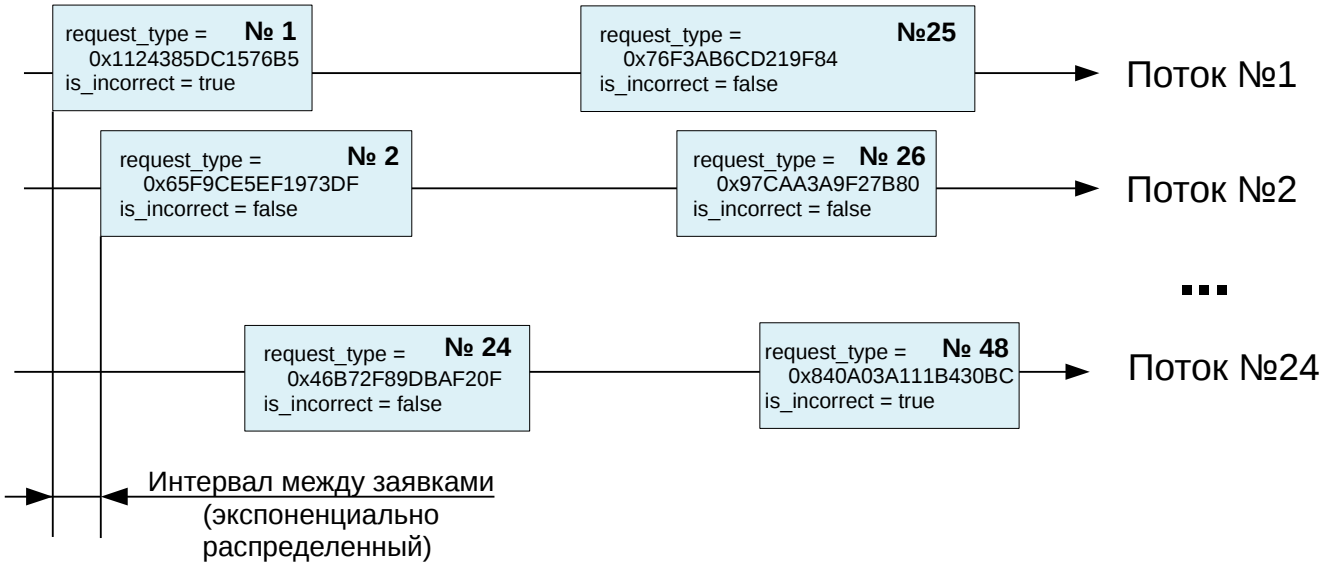
Например в листинге 1 определяется эксперимент, который называется "jump_table". В эксперименте в разделе "workloads" определяется нагрузка jt, тип которой также jt, для которой установлены следующие параметры:

- num_request_types — установлено глобально для всей нагрузки — количество типов запросов, которые будут генерироваться;
- request_types — для каждого запроса генерируется линейно-конгруэнтным генератором случайных чисел;
- is_incorrect — булевское значение, для 20% запросов будет установлено в true, для остальных в false.

Планировщик запросов этой нагрузки — время прихода между запросами (iat, inter-arrival time), и генерироваться оно будет по экспоненциальному закону. В разделе "step" указывается количество запросов, которое будет сгенерировано для этой нагрузки: 100 шагов по 2000 запросов.

В разделе "threadpools" определяются пулы потоков, которые будут исполнять тестовые нагрузки. В нашем примере определяется пул tp_jt (к нему же и привязывается нагрузка jt) содержащий 24 потока с периодом шага в 2 секунды (задается параметром quantum в наносекундах). Диспетчер потока — round-robin, определяет как будут распределяться запросы между потоками.

76Если представить полученную нагрузку, можно получить приблизительно следующую картину:



Листинг 1. Файл experiment.json

```
{   "name": "jump_table",
    "steps": {
        "jt": {
            "num_steps":    100,
            "num_requests": 2000 }
    },
    "threadpools": {
        "tp_jt" : {
            "num_threads": 24,
            "quantum": 2000000000,
            "disp": { "type": "round-robin" } }
    },
    "workloads" : {
        "jt" : {
            "wltype": "jt",
            "threadpool": "tp_jt",
            "params": {
                "num_request_types": 5000,
                "request_type": {
                    "randgen": { "class": "lcg" } },
                "is_incorrect": {
                    "randgen": { "class": "lcg" },
                    "pmap": [
                        { "probability": 0.2, "value": true },
                        { "probability": 0.8, "value": false }
                    ] } },
            "rqsched": {
                "type": "iat",
                "distribution": "exponential"
            }
```

```
        }
    }
}
```

Тип нагрузки jt определен в отдельном модуле — он и содержит код, который симулирует выполнение запросов. В ходе курса мы столкнемся с другими такими модулями и типами нагрузок: proc_starter, в каждом запросе запускающий новый процесс, file_opener, открывающий файлы, и некоторые другие.

Чтобы запустить эксперимент, нужно выполнить команду tsexperiment:

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

где /path/to/experiment — директория, содержащая созданный ранее файл experiment.json.

Внутри этой директории также будут располагаться результаты эксперимента, посмотреть список которых можно с помощью команды ls или подкоманды list команды tsexperiment:

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

Вывести некоторые статистические результаты эксперимента можно с помощью подкоманды report, а экспортировать их в csv или json с помощью export.

Чтобы сделать несколько запусков эксперимента с разными значениями параметров но не редактировать конфигурацию можно указать опцию -s в подкоманде run. Для этого сначала вызовите show с опцией -l чтобы получить список всех доступных опций:

```
# tsexperiment -e /opt/tsload/var/tsload/mbench/jt show -l
name=jump_table
steps:jt:num_steps=100
steps:jt:num_requests=2000
...
```

Тогда чтобы выполнить эксперимент в котором бы запускалось 500 запросов на шаг нужно вызвать команду tsexperiment так:

```
# tsexperiment -e /opt/tsload/var/tsload/mbench/jt run      \
     -s steps:jt:num_requests=500
```

В некоторых случаях нам потребуется указывать объекты операционной системы в конфигурации эксперимента: например привязывать потоки к процессорным ядрам. Чтобы узнать их имена, используйте команду tshostinfo:

```
# tshostinfo -x
```

Ядро операционной системы

Ядро операционной системы

- Что такое ядро операционной системы, системный вызов?
- Что такое контекст исполнения?
- Какова структура программы на C?
  - Что такое аргументы и переменные?

2

Ядро (Wikipedia) — центральная часть операционной системы (ОС), обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации, переводя команды языка приложений на язык двоичных кодов, которые понимает компьютер. Также обычно ядро предоставляет сервисы файловой системы и сетевых протоколов.

Для того, чтобы обратиться к различным функциям ядра приложения используют механизм системных вызовов — передавая тем самым управление подпрограммам операционной системы. Текущее состояние приложения при этом составляет его контекст.

C — язык программирования, лежащий в основе ядер современных UNIX-подобных операционных систем, таких как Solaris, FreeBSD и Linux и реализует парадигму структурного программирования.

Ядро операционной системы

- Cross-Reference
  - Linux: http://lxr.linux.no/
  - OpenSolaris: http://src.opensolaris.org/
- Литература
  - Роберт Лав — Разработка ядра Linux
  - Richard McDougall, Jim Mauro - Solaris Internals
  - SI-365-S10 — Solaris Internals

3

Где же получить информацию по ядру? В первую очередь — это его исходные коды, содержащие многочисленные комментарии. В этом хорошо помогают т. н. Cross Reference, позволяющие быстро переходить между определениями в коде с помощью гиперссылок. Для Linux — это сайт http://lxr.linux.no/, для Illumos (открытый форк проекта OpenSolaris, закрытого Oracle) — http://src.illumos.org/ (имя проекта illumos-gate).

При желании можно создать свой Cross-Reference, используя сервер OpenGrok: https://github.com/OpenGrok/OpenGrok

Второй способ — это литература и текстовая документация. По Linux:

- Директория Documentation/ ядра
- Списки рассылок разработчиков ядра Linux http://lkml.org/
- Linux info from source: http://lwn.net/
- Книга Роберта Лава «Разработка ядра Linux»
- Linux Device Drivers Book: https://lwn.net/Kernel/LDD3/

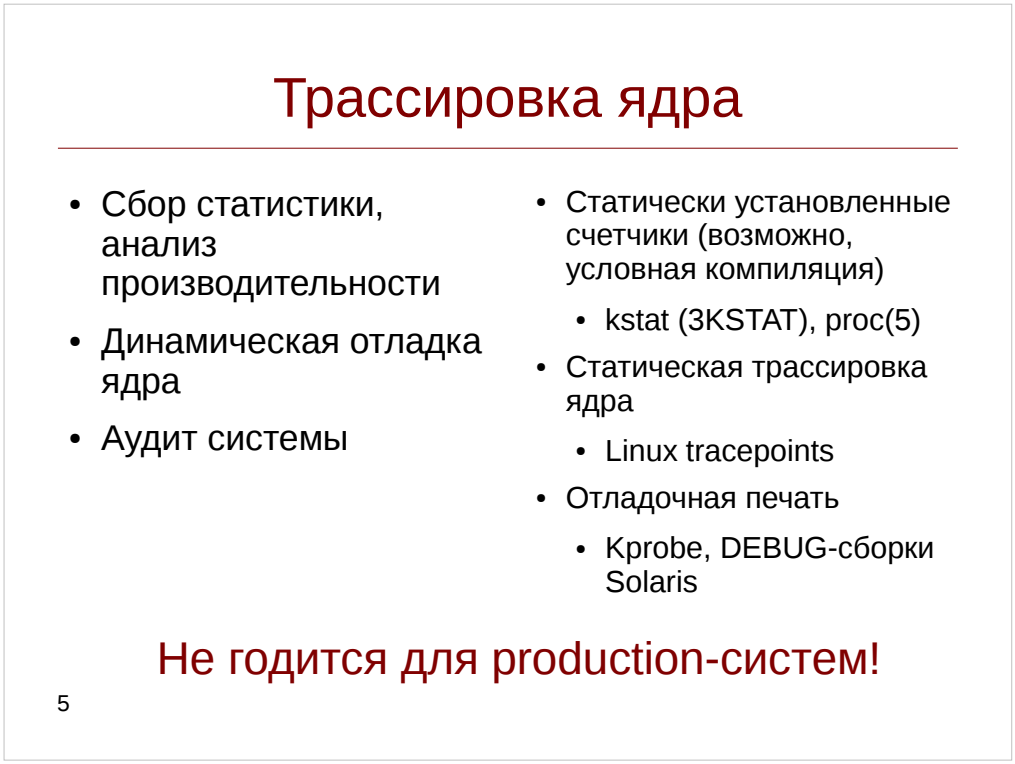
По Solaris:

- http://solaris.java.net/
- Книга «Richard McDougall, Jim Mauro Solaris(TM) Internals: Solaris 10 and OpenSolaris Kernel Architecture»
- Курс Solaris 10 Operating System Internals (D61832GC20)

⚠ *Замечание*: следует признать, что после приобретения компании Sun корпорацией Oracle большая часть информации о Solaris стала закрытой, что затрудняет его анализ.

Модуль 1. Инструменты динамической трассировки. Утилиты dtrace и stap

Трассировка ядра



Трассировка (Wikipedia) - пошаговое выполнение программы с остановками на каждой команде или строке.

Значительная часть методов анализа производительности систем основана на т. н. статических счетчиках: при возникновении определенного события в ядре (например, страничном, page fault) увеличивается соответствующий счетчик, например с помощью интерфейса kstat в ядре Solaris:

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

или посредством procfs/sysfs в linux

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

Этот подход имеет несколько недостатков: во-первых большое количество счетчиков негативно сказывается на производительности системы, во-вторых такие данные крайне неполны (например невозможно выяснить количество событий по процессам)

Во-вторых, осложнен аудит и отладка ядра, так как каждый вызов отладчика требует остановки работы системы (в случае классических точек останова, т. е. breakpoints), ввода дополнительного отладочного вывода (что в случае Solaris вообще невозможно, так как т. н. DEBUG-сборки являются закрытыми).

Динамическая трассировка

Основное отличие динамической трассировки от других методов анализа ядра — возможность встраивать пользовательский код в работающее ядро, что позволяет предоставить большое количество событий и большую гибкость при их обработке.

Один из первых и наиболее известных инструментов динамической трассировки — Solaris DTrace, разрабатывавшийся с 1999 года. Презентации, посвященные ранней разработке находятся здесь:
https://blogs.oracle.com/bmc/entry/happy_5th_birthday_dtrace

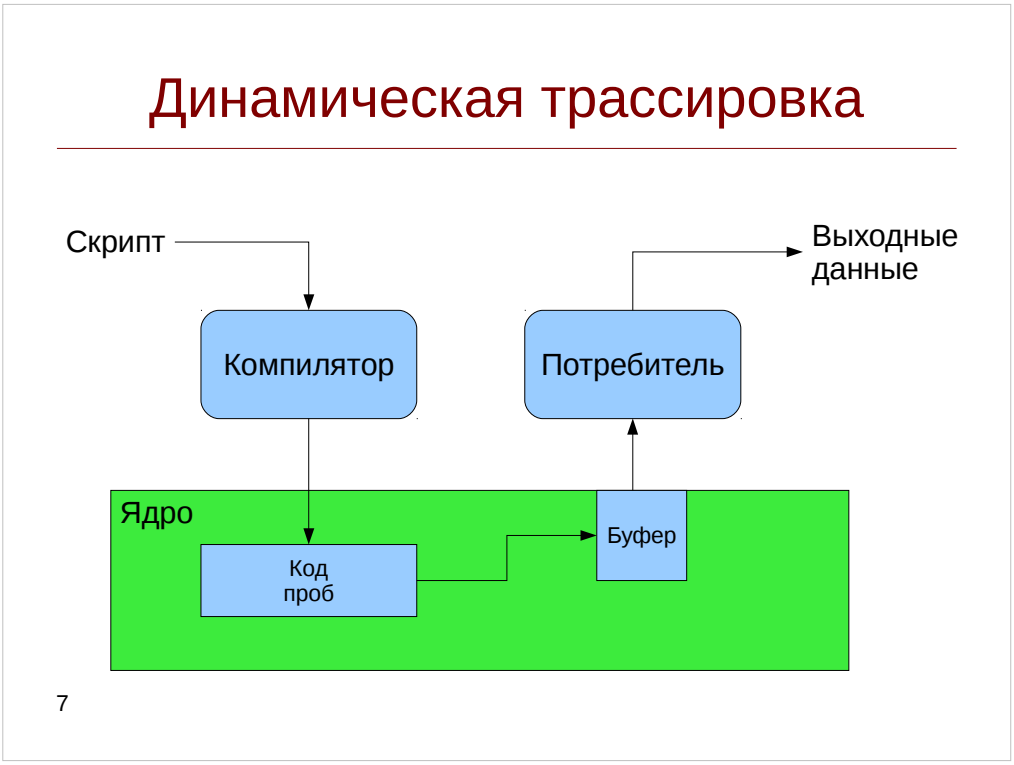
Более подробная информация по DTrace:
https://wikis.oracle.com/display/DTrace/DTrace
http://www.solarisinternals.com/wiki/index.php/DTrace_Topics
А также книги «Solaris Perfomance and Tools» и «DTrace - Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD»
Официальная документация - «Solaris Dynamic Tracing Guide»: http://download.oracle.com/docs/cd/E19253-01/817-6223/

Портирование DTrace в Linux затруднено из-за лицензионных ограничений, поэтому он поставляется только в Oracle Enterprise Linux. Существует более ранний фреймворк DProbes, однако наиболее развитый фреймворк на сегодняшний день — SystemTap, разработанный Red Hat, Hitachi, IBM и Oracle (http://sourceware.org/systemtap/).

Официальная документация - «SystemTap Language Reference» http://sourceware.org/systemtap/langref/ и «SystemTap Tapset Reference Manual» http://sourceware.org/systemtap/tapsets/

В качестве альтернативы громоздкой системе SystemTap, также создана более леговесная система Ktap, основное отличие которой — использование Lua и LuaJIT для исполнения кода внутри ядра (SystemTap компилирует нативные модули ядра). Также, существует инструмент Sysdig, который с помощью специального модуля ядра позволяет трассировать события, возникающие внутри ядра, однако он не имеет скриптового языка.

Как следует из названия курса, в центре нашего внимания окажутся системы DTrace и SystemTap.



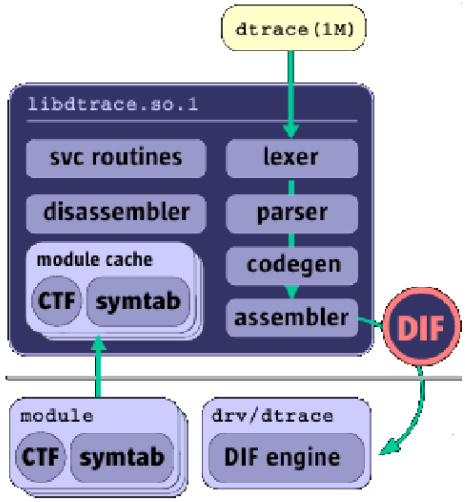
Логика работы динамического трассировщика проста: создается скрипт на C-подобном языке (в DTrace скрипты имеют расширение .d, в SystemTap - .stp) и преобразуется в бинарный код целевой архитектуры, в DTrace это делается посредством компилятора языка D (не путать с языком D от Digital Mars), в SystemTap скрипт транслируется в исходник на C.

Затем полученный код отображается в адресное пространство ядра — в Solaris это делает модуль ядра DTrace, в SystemTap каждый скрипт компилируется в отдельный модуль ядра. Вывод данных осуществляется через промежуточный буфер, откуда потребитель забирает данные. Промежуточный буфер нужен, чтобы потребитель мог работать асинхронно, не блокируя работу ядра и чтобы избежать огромного количества переключений контекста. Потребитель выводит полученную информацию на терминал, в конвейер или в файл.

DTrace

DTrace

- По-умолчанию входит в Solaris.
- Использует Dtrace Intermediate Format (компактная RISC-VM)



8

DTrace впервые появился в Solaris 10 и не требует никаких специальных изменений ядра — таблица символов и информация о типах CTF по умолчанию поставляется в бинарных образах ядра Solaris.

Сердце DTrace — библиотека libdtrace.so.1, содержащая в себе компилятор, преобразующий текст на языке D в DTrace Intermediate Format. Этот формат по сути представляет собой машинные коды компактной виртуальной машины, имеющей крайне упрощенную RISC систему команд.

В качестве потребителей выступают утилиты dtrace(1M), по сути являющейся центральным фронт-ендом, и trapstat(1M) с lockstat(1M). Исполнением кода со стороны ядра занимается драйвер drv/dtrace.

dtrace(1M)

- dtrace
  - `# dtrace -n 'syscall::write:entry { trace(arg0); }'`
  - `# dtrace -s syscalls.d [аргументы]`
  - `# dtrace [-P провайдер] [-m модуль] [-f функция] [-n имя]`
- Параметры
  - -l — вывести список проб (вместо выполнения скрипта)
  - -q — тихий режим
  - -w — деструктивный режим
  - -o — вывод данных в файл
  - -c — привязаться к выполнению программы

9

dtrace


[14]
[15]

DTrace поддерживает три режима запуска:

- Скрипт явно указывается в командной строке
  `# dtrace -n 'syscall::write:entry { trace(arg0); }'`
- Скрипт располагается в отдельном файле
  `# dtrace -s syscalls.d [аргументы]`

В данном случае возможно передавать аргументы скрипта, например pid наблюдаемого процесса или имя файла, для которого наблюдается ввод-вывод. При этом аргументы будут располагаться как и в любом shell-скрипте в переменных $1, $2 … $n

Так как строки в DTrace заключаются в двойные кавычки, то соответствующий аргумент следует передавать следующим образом:

```
# dtrace -s syscalls.d '"mysqld"'
```

- Явным образом указав пробу привязки
  `# dtrace [-P провайдер] [-m модуль] [-f функция] [-n имя]`

Описание параметров коммандной строки приведено ниже:

- `-l` позволяет вывести список всех доступных проб. Фильтровать можно, используя параметры -P, -m, -f, -n (см. выше). Например:

```
# dtrace -l -P io
  ID    PROVIDER            MODULE          FUNCTION NAME
  800         io           genunix           biodone done
  801         io           genunix           biowait wait-done
  802         io           genunix           biowait wait-start
<cut>
```

- `-q` — вызывает тихий режим. По-умолчанию при срабатывании пробы DTrace выводит ее номер, имя и номер процессора, -q отключает это.
- `-w` разрешает использование деструктивных действий (actions), таких как например actions, таких как например паника и приостановка выполнения процесса.

- `-o ФАЙЛ` перенаправляет вывод в файл, добавляя новый вывод DTrace в него
- `-x ОПЦИЯ[=ЗНАЧЕНИЕ]` указывает одну из настроек DTrace. Список опций представлен ниже:
  - bufsize — размер промежуточного буфера ввода-вывода (то же, что и -b)
  - cpu — процессор, на котором ведется трассировка;
  - dynvarsize — размер буфера под динамические переменные (в частности, ассоциативные массивы);
  - queit — тихий режим (то же, что и -q);
  - flowindent — выводить в режиме дерева вызовов. Подробнее этот режим описан в разделе «Динамический анализ кода» на с. 66;
  - destructive — деструктивный режим (то же, что и -w).

  Внутри скрипта определить опцию можно посредством директивы #pragma, например:
  ```
  #pragma D option bufsize=64m
  ```
- `-C` — вызывать препроцессор C cpp(1) перед выполнением скрипта. Необходимо для обработки директив #include, #define/#ifdef/#ifndef и т. д. Опции препроцессора:
  - `-D МАКРОС[=ПОДСТАНОВКА]` определяет макроподстановку, `-U` снимает макроопределение
  - `-I` добавляет путь ко включаемым директивой #include файлам
  - `-H` печатает включенные скриптом файлы
- `-A/-a` — анонимная трассировка (anonymous tracing). Используется для трассировки процесса загрузки системы

Полная информация: https://wikis.oracle.com/display/DTrace/dtrace%281M%29+Utility и https://wikis.oracle.com/display/DTrace/Options+and+Tunables

dtrace(1M)

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

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

```
root@host# chmod +x /root/test.d
root@host# /root/test.d -c "dd if=/dev/zero of=/dev/null
count=1"
10
```

Предлагается создать файл-скрипт test.d и запустить его, как указано на примере.

По очереди уберите опции flowindent и q из скрипта test.d и снова запустите DTrace-скрипт. Что изменилось?

_____

_____

_____

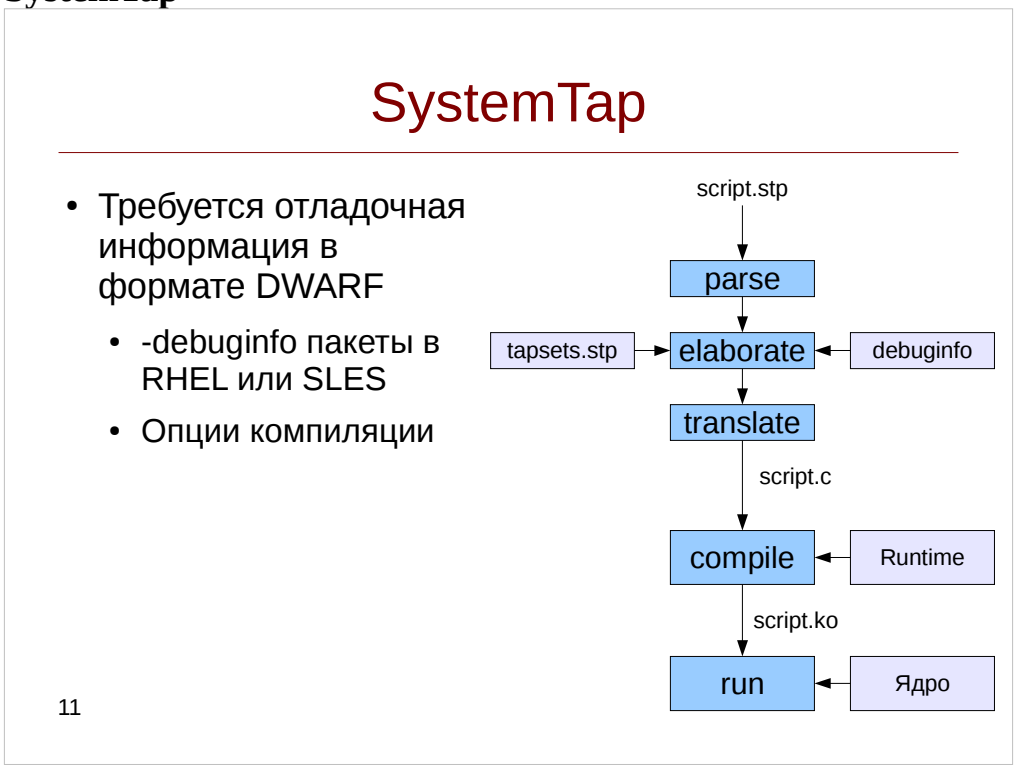
_____

Подсчитайте количество функций, предоставляемых провайдером fbt:

```
# dtrace -l -P fbt | wc -l
```

_____

SystemTap



Для работы SystemTap требуется отладочная информацией в формате DWARF: в противном случае набор предоставляемых проб будет ограничен.

- На штатных ядрах требуется пакеты kernel-devel, kernel-debuginfo и kernel-debuginfo-common (RHEL) или аналоги, которые содержат как заголовочные файлы, так и DWARF-секции бинарных файлов, вынесенные в отдельные файлы. В новых версиях SystemTap нужные пакеты можно доустановить используя команду stap-prep.
- На самосборных ядрах: ядро с опциями CONFIG_DEBUG_INFO, CONFIG_KPROBES, CONFIG_RELAY, CONFIG_DEBUG_FS, CONFIG_MODULES, CONFIG_MODULE_UNLOAD. Также требуется бинарный образ vmlinux (несжатый) и исходники ядра, расположенные в /lib/modules/<версия ядра>/build/

Утилиты:

- stap(1) компилирует скрипт в модуль ядра и запускает его
- staprun(1) запускает заранее скомпилированный модуль
- stapio — потребитель (получает информацию из временного буфера и выводит ее на stdout или в файл).

З*амечание*: если родитель процесса stap завершился, то вызов killall -9 stap не завершит демон stapio. Необходимо вызывать killall -15 stap (сигнал SIGTERM).

Запуск скрипта SystemTap выполняется в 5 фаз: синтаксический разбор (parse), отображение на отладочную информацию (elaborate), трансляция в код на C (translate), компиляция модуля ядра (compile) и собственно запуск (run). Скомпилированные модули кешируются.

stap(1)

- stap:
  - `# stap -e '[код пробы]' [аргументы]`
  - `# stap syscalls.d [аргументы]`
- Параметры
  - -l/-L — вывести список проб (вместо выполнения скрипта)
  - -q — тихий режим
  - -g — режим Guru
  - -c / -x — привязаться к выполнению программы
  - -o — вывод данных в файл

12

stap
staprun

1.5
5.7

Также как и DTrace, SystemTap поддерживает два основных режима запуска, синтаксис вызова:

- Скрипт может явно указываться в коммандной строке:
  `# stap -e 'probe syscall.write { printf("%d\n", $fd); }' [аргументы]`
- Скрипт указывается во внешнем файле (или в stdin, тогда в качестве имени файла передается дефис):
  `# stap syscalls.d [аргументы]`

Аргументы при этом, также как в DTrace передаются в качестве переменных $1, $2 … $n, однако отстутсвует проблема со строковыми переменными — строковые переменные явно содержаться в @1, @2 … @n

Наиболее часто используемые опции SystemTap:

- `-l` выводит список проб вместо исполнения скрипта, -L также печатает и имена аргументов. В качестве аргумента этой опции указывается спецификатор пробы, например:
  `# stap -l 'scsi.*'`
- `-v` - в зависимости от количества букв 'v' увеличивает подробность вывода диагностической информации. Если указана однократно — информирует об окончании фаз запуска скрипта
- `-p X` - заставляет stap останавливаться после выполнения стадии X. Например, остановившись на 4й стадии мы не запустим модуль, но скомпилируем его.

- `-k` - stap не удаляет сгенерированные файлы (обычно располагаются в /tmp/stapXXXX)
- `-g` - включает guru-mode, активирующий дополнительные пробы и позволяющий изменять состояние ядра
- `-c КОММАНДА / -x PID` - при запуске stap запускает одновременно команду или цеплятся к уже запущенному процессу по pid. Этот pid можно узнать, используя встроенную функцию systemtap target()
- `-o ФАЙЛ` - перенаправляет вывод systemtap в файл
- `-m МОДУЛЬ` - указывает имя модуля вместо stap_YYYY. В сочетании с опцией -k это полезно в случае, если мы хотим использовать модуль повторно, не перекомпилируя его. Для запуска чистого модуля stap используется команда staprun. Но учитывайте, что собранный модуль подойдет только для того ядра, для которого он собран
- `--all-modules` - разыменовывает адреса для всех модулей (по-умолчанию stap делает это только в рамках одного модуля)
- `-d МОДУЛЬ` путь до модуля ядра, исполняемого файла или приложения для которого должен выполняться поиск имен
- `--ldd` — для трассировки процесса — использовать ldd, чтобы определить задействованные модули. Это позволяет избежать явного задания опций -d.

stap(1)

```
#!/usr/bin/stap

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

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

13

Также, запустите SystemTap с опциями:

```
# stap -vv -k -p4 /root/test.stp
```

Откройте полученный C-файл (будет иметь имя вида /tmp/stapXXXXXX/stap_yyyyy.c) и найдите сгенерированный код пробы.

Подсчитайте количество системных вызовов, предоставляемых tapsetom syscall и выясните имена аргументов и переменных syscall.write:

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

Безопасность

Как уже говорилось, динамическая трассировка — превосходный способ наблюдения за production-системами, однако не стоит забывать, что код проб располагается в ядре, а значит выполнение проб сопряжено с определенными рисками: фатальные действия, такие как разыменовывание NULL-указателя могут привести к панике ядра, а слишком большое время выполнения проб — к серьезному замедлению работы системы.

Поэтому трассировщики вынуждены вставлять дополнительный код для

- Мониторинга за временем выполнения проб
- Отслеживания использования ресурсов а также предоставлять обертки для работы с сырой памятью ядра.

Наиболее часто встречающиеся ошибки, связанные с нехваткой ресурсов:

- *DTrace drops on CPU #* - связано с переполнением промежуточного асинхронного буфера — требуется уменьшить switchrate или увеличить bufsize
- *SystemTap probe overhead exceeded threshold* — слишком большое время выполнения проб (более 50% процессорного времени). Опция -t позволяет собрать информацию о времени выполнения проб. Можно изменить допустимый предел (переменная STP_OVERLOAD_THRESHOLD) или отключить его (STP_NO_OVERLOAD). Эти переменные можно задать через опцию -D утилиты stap.

Подробная информация по ошибкам, типичным для SystemTap http://sourceware.org/systemtap/wiki/TipExhaustedResourceErrors

См. также DTrace Deadman Mechanism

Стабильность

Стабильность

Проблема:

- Переменчивость интерфейсов внутри ядра

Решение:

- DTrace — классификация интерфейсов по уровню стабильности, трансляторы
- SystemTap — условная компиляция

```
%( условие %? код [ %: код ] %)
```

15 SL 5.8 D [39]

Другая проблема — стабильность интерфейсов внутри ядра. Если системные вызовы редко (практически никогда) не меняют свой интерфейс, то для функций и структур внутри ядра это неверно, поэтому при написании скриптов следует делать поправку на стабильность интерфейсов ядра и следовать простым правилам:

- Использовать по возможности пробы-алиасы вместо непосредственной привязки к функциям ядра
- Использовать функции из tapset или трансляторы в DTrace, извлекающие данные из структур

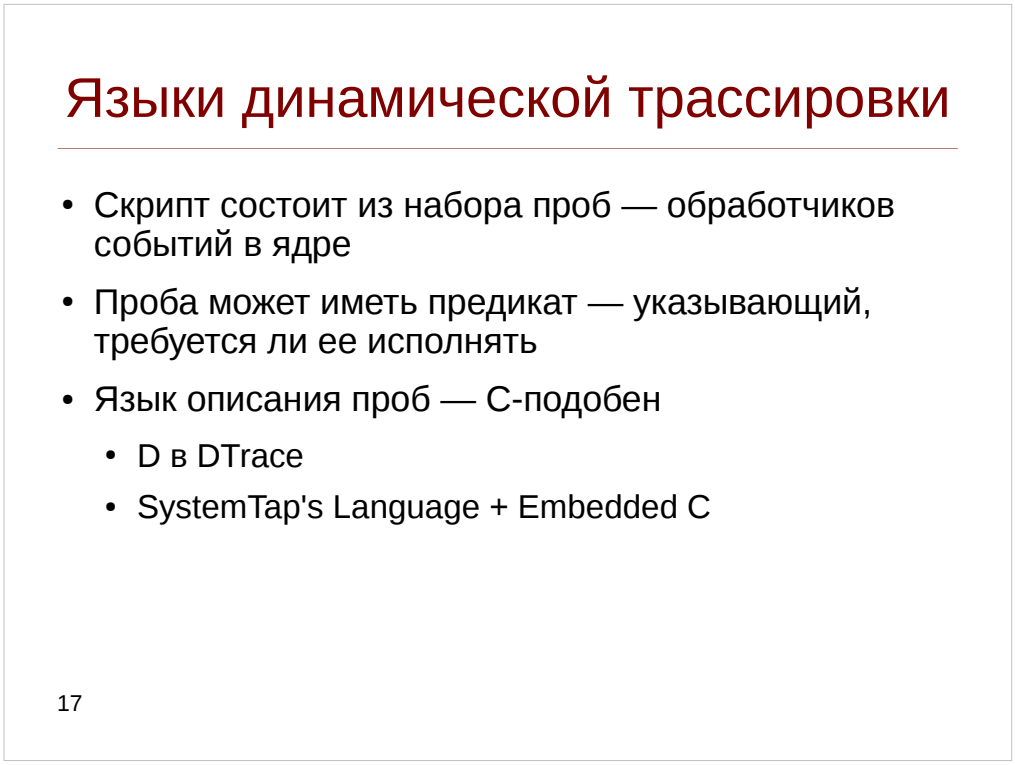
В DTrace все провайдеры классифицируются по уровню их стабильности (подробнее — в Solaris Dynamic Tracing Guide [39]), в SystemTap единственный способ — это условная компиляция:

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

Среди доступных переменных для условной компиляции — arch (архитектура процессора, для которой собрано ядро), kernel_v и kernel_vr — версии ядра (_vr также включает буквенный суффикс), опции конфигурации ядра CONFIG_*. В Embedded C условная компиляция выполняется также как и в языке C — через директивы препроцессора.

Если какая-нибудь функция отсутствует в ядре, то DTrace и SystemTap выведут ошибку. Чтобы отбросить такие функции, нужно указать опцию -Z DTrace или добавить к пробе суффикс ? в SystemTap.

Модуль 2. Языки динамической трассировки



Языки DTrace и SystemTap имеют C-подобный синтаксис, позволяющий описывать скрипты динамической трассировки. Скрипт состоит из набора проб, каждая из которых однозначно привязывается к определенному событию в ядре или приложению, например диспетчеризации процесса на исполнение процессором или разбор SQL-запроса, предикат пробы определяет, следует ли исполнить код пробы или отбросить его, соответственно это может быть имя исполнимого файла или тип запроса.

SystemTap в отличие от DTrace реализует более широкое подмножество языка C (фактически — это обертка над языком C), а также позволяет интегрировать код на C в скрипты (т. н. Embedded C). В частности, SystemTap поддерживает инструкции для организации циклов (for, foreach, while, break и continue) и условную инструкцию if/else.

SystemTap поддерживает создание функций. Они определяются следующим образом:

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

В этом примере функция dentry_name получает аргумент dentry типа long (на самом деле это указатель на структуру данных) и возвращает соответствующую этой структуре строку.

В DTrace такой возможности нет, но можно создавать макросы препроцессора

C:

```
#define CLOCK_TO_MS(clk)     (clk) * (`nsec_per_tick / 1000000)
```

Кроме того, SystemTap имеет оператор try-catch, который позволяет обрабатывать внутренние ошибки:

```
try {
     println(kernel_int(4));
}
catch(msg) {
     /*Игнорировать ошибки*/
}
```

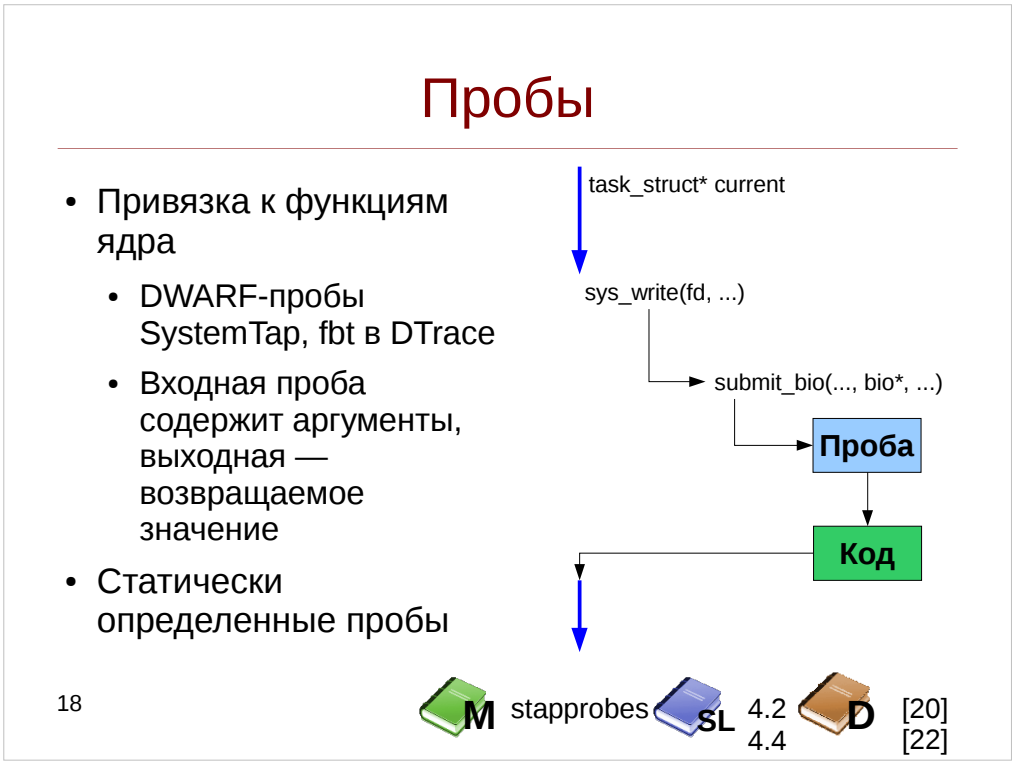
В DTrace доступен только тернарный оператор ?: для организации условий, но и он содержит ограничения. Построить циклы (и, естественно условия) на предикатах проб например так:

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

Код на Embedded C в SystemTap заключается в %{ ... %} и требует запуска утилиты stap в режиме Guru. В этом случае код не транслируется, и таким образом доступны все интерфейсы ядра, например интерфейсы read-copy-update:

```
function task_valid_file_handle:long (task:long, fd:long) % {  /*
pure */
   <cut>

   rcu_read_lock();
   if ((files = kread(&p->files))) {
           filp = fcheck_files(files, THIS->fd);
           THIS->__retvalue = !!filp;
   }

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

Пробы



Проба — обработчик события в ядре.

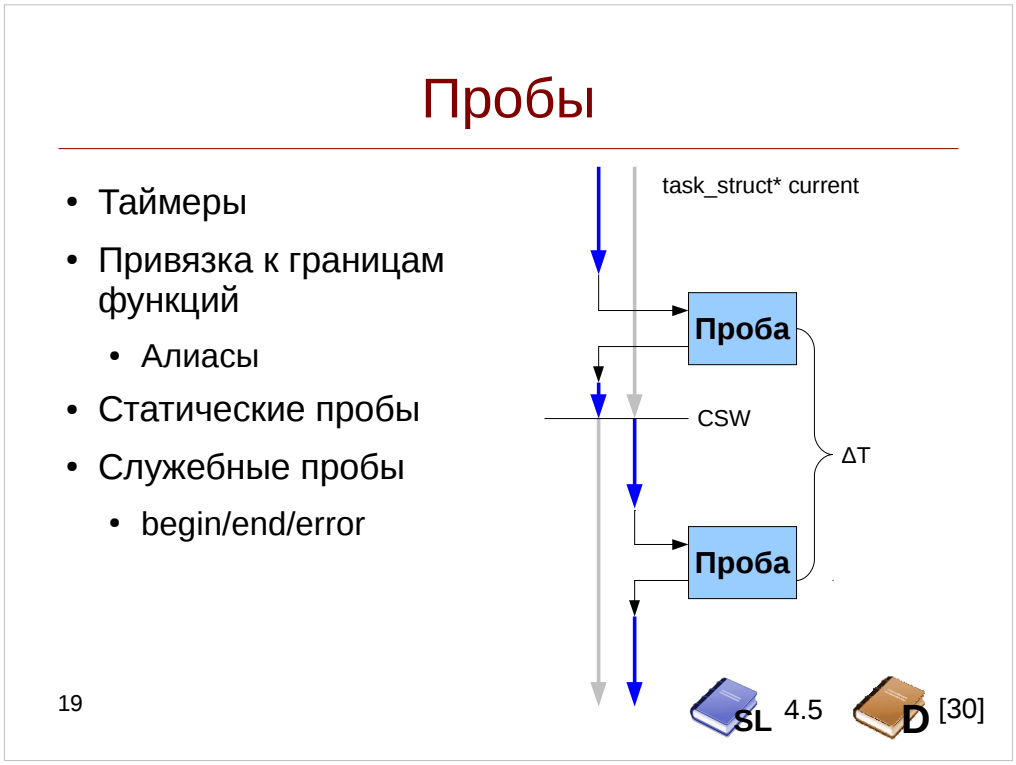
- Наиболее распространены пробы, привязываемые к вызову или возврату для той или иной функции ядра. Вызов большинства функций традиционно начинается с сохранения регистров и установки указателя стека. Трассировщик перехватывает этот вызов, например:

```
bdev_strategy:   pushq  %rbp          →  int    $0x3
bdev_strategy+1: movq   %rsp,%rbp        movq   %rsp,%rbp
bdev_strategy+4: subq   $0x10,%rsp       subq   $0x10,%rsp
```

  После этого при вызове функции bdev_strategy вызов будет перехвачен и управление передано коду соответствующей пробе-обработчику.
- Статически определенные пробы работают аналогичным образом, однако реализованы поверх установленных меток внутри самих функций. Без активной пробы они в идеальном случае содержат лишь инструкцию nop, после ее активации инструкция соответственно подменяется вызовом соответствующей пробы.

Например рассмотрим процесс, осуществляющий синхронную запись в некий файл. При старте записи он передаст управление ядру, осуществив системный вызов write, что приводит к вызову функции sys_write, которая в свою очередь так или иначе вызывает функцию submit_bio. Привязавшись к этим двум функциям можно получить информацию:

- Процесс, инициировавший ввод-вывод (фактически указатель current)
- Файловый дескриптор, в который осуществляется вывод (fd в sys_write)
- Параметры дискового ввода-вывода (например номер блока из bio)



Пробы-таймеры, срабатывают один раз в период ΔT, на одном или на всех процессорах системы. Используя все тот же указатель current, можно при каждом срабатывании узнать текущий процесс и таким образом получить аналог утилит top или prstat, который бы вычислял процент использования процессора тем или иным процессом. Этот метод называется профилирование.

Привязка к границам функций — наиболее часто используемые пробы, которые позволяют получать аргументы и возвращаемые значения функции, измерять ее время выполнения. Статические пробы, как правило подставляются внутри тела функции для отслеживания этапов ее выполнения, там где границ не достаточно, например — конечный автомат TCP-соединения, или получения дополнительных диагностических данных, вычисляемых в ходе выполнения функции.

Пробы-алисы предоставляют более удобный доступ к пробам, привязываемым к функциям. В DTrace они реализованы внутри логики самого языка D, в SystemTap представляют собой внешние скрипты (т. н. tapsets, расположенные в /usr/share/systemtap/tapset). Алиасы в SystemTap можно задавать самостоятельно.

Специальный класс представляют из себя служебные пробы — это прежде всего пробы запуска и остановки скрипта: обычно при запуске инициализируются глобальные переменные, при остановке — выводится результат трассировки. В SystemTap они называются begin, end и error, в DTrace пробы BEGIN и END предоставляются провайдером DTrace.

Пробы

SystemTap

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

probe timer.ms(100) {
    // Действия
}

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

20

DTrace

Провайдер:Модуль:Функция:Имя

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

profile-100ms {
    // Действия
}

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

4.1


[4]

Рассмотрим, каким образом описывается проба. В SystemTap имя пробы указывается через точку и начинается с ключевого слова probe:

- Пробы начала begin(N) и окончания end(N) выполняются в порядке очередности параметра N. Также, SystemTap имеет пробу error, срабатывающую при возникновении ошибки
- Таймерная проба срабатывает через определенный интервал времени:

  `timer.единица_измерения(N)[.randomize(M)]`

  например:

  `timer.ms(100)`

  Дополнительный параметр randomize позволяет менять временной промежуток в интервале [N-M..N+M] Список единиц измерений:
  - s/sec — секунды
  - ms/msec — миллисекунды
  - us/usec — микросекунды
  - ns/nsec — наносекунды
  - hz — герцы
  - jiffies (внутренний тик ядра)

  Для профилирования существует специальная проба timer.profile. Она имеет фиксированный интервал срабатывания, но зато выполняется на всех процессорах.
- DWARF-пробы привязываются к вызову одной из функций ядра и имеют следующий синтаксис:

  `{kernel|module(паттерн)}.function(паттерн).{call|return}`

  kernel — привязка к образу ядра vmunix, а module — к одному из модулей, например: module("*fs").function("*block*"). Возможна подстановка символов в имени модуля - * и ?

  - function определяет имя функции, к которой осуществляется привязка. Синтаксис паттерна имени функции более сложный: он также может включать в себя имя исходного файла, указанный через символ @ например:
    ```
    kernel.function("*@fs/*.c")
    ```

- Модификаторы
  - .call используется для проб не-inline функций
  - .return — для проб возврата (имеются только у не-inline функций)
  - Если ни один модификатор не указан, то это расценивается как объединение модификаторов .call и .inline

Также существует несколько экзотических вариантов DWARF-проб, в частности привязка к адресу в памяти или номеру строки:

```
kernel.statement(паттерн)
kernel.statement(адрес).absolute
module(паттерн).statement(паттерн)
```

Кроме DWARF-проб, поддерживаются также KProbes, Linux Tracepoints, Procfs, и т. д. Нас больше будут интересовать пробы из Tapset'ов, представляющие из себя обертки над DWARF-пробами.

Системные вызовы реализуются в SystemTap специальным набором алиасов (tapset'ом), так как в Linux может быть несколько функций, семантически представляющих один и тот же системный вызов. Помимо этого, эти пробы обрабатывают аргументы, например копируя передаваемые строки в пространство ядра. Этот tapset носит название syscall:

```
syscall.имя_вызова
```

Более подробная информация о пробах в SystemTap — в разделе справочных страниц stapprobes(3stap)

DTrace имеет более простой синтаксис именования проб: имя каждой пробы состоит из четырех частей, разделенных двоеточиями:

```
Провайдер:Модуль:Функция:Имя_пробы[-Параметр]
```

Провайдер определяет группу проб, к которой она относится

- fbt — пробы, привязываемые к вызовам функций ядра
- dtrace — системные пробы (BEGIN/END)
- tick и profile — таймеры. tick выполняется на одном из процессоров, тогда как profile выполняется на всех процессорах
- sdt — статически определенные пробы ядра

Модуль и имя функции конкретизируют привязку для провайдеров fbt и sdt, различие лишь в том, что fbt предоставляет для всех функций пробы entry и return (соответственно точка входа и возврата), например:

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

Тогда как sdt предоставляет специфичные пробы, например:

```
sdt:unix:page_get_freelist:page-get
```

Их необходимость обусловлена тем, что DTrace не позволяет осуществлять привязку к любой строчке кода, а провайдер fbt не позволил бы обработать например такие ситуации:

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

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

Для этого вызов функции окружен пробами taskq-exec-start и taskq-exec-end.

Также, как и SystemTap, DTrace разрешает подстановку символов в именах проб, функций и модулей (отстутсвие одного из параметров означает подстановку всех возможных имен, то есть «*»), например проба, устанавливаемая на вызовы всех функций выглядит так:

```
fbt:::entry
```

Как и в SystemTap, в DTrace есть специальный провайдер для системных вызовов syscall. Он однако реализован через специальный драйвер systrace, который подменяет вызовы в таблице системных вызовов. Обращаться к нему можно также, как и к провайдеру fbt:

```
syscall::openat64:entry
```

Итак, для рассмотрим такой код на языке C:

```
 1 float tri_area(float a, float b,
 2                 float angle) {
 3         float height;
 4
 5         if(a < 0.0 || b < 0.0 ||
 6                   angle >= 180.0 || angle < 0.0)
 7              return NAN;
 8
 9         height = b * sin(angle);
10
11         DTRACE_PROBE1(triangle__height, h);
           или
           trace_triangle_height(h);
12
13         return a * height;
14 }
```

Тогда привязаться к разным строкам кода можно с помощью таких проб:

Строка кода	*DTrace*	*SystemTap*
1	fbt::tri_area:entry	kernel.tri_area("tri_area").call
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("static_mark")
13	fbt::tri_area:return	kernel.tri_area("tri_area").return kernel.statement("tri_area+12")

В случае если в SystemTap или DTrace -скрипте может быть указана проба, реально не существующая в ядре, то компиляция скрипта завершится с ошибкой. Чтобы игнорировать такие ошибки, необходимо вызывать dtrace с опцией -Z, в SystemTap в описателе пробы указывать вопросительный знак:

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

Также отметим, что описания проб можно комбинировать, описывая их через запятую. К одной точке можно привязать несколько проб — в этом случае их код выполняется в порядке их следования в скрипте:

```
probe syscall.read {
     /* Подготовительные действия */
}
probe syscall.read, syscall.write {
     /* Общие для read и write действия */
}
```

или

```
syscall::read:entry {
     /* Подготовительные действия */
}
syscall::read:entry, syscall::write:entry {
     /* Общие для read и write действия */
}
```

Аргументы пробы

Аргументы пробы

- Аргументы содержат параметры вызова пробы
  - DTrace
    - args[0], args[1] … args[n] — типизованные аргументы вызова пробы
    - arg0, arg1 … argn — преобразованные в uint64_t
  - SystemTap
    - $имя — аргументы вызова и имя — локальные переменные tapset
    - $$vars, $$locals, $$params — sprintf варианты
- Для return-проб:
  - DTrace: arg0 — точка возврата, args[1] — возвращаемое значение
  - SystemTap: $return — возвращаемое значение

21 SL 4.2 D [5]

При привязке к пробе-алиасу или пробе-вызову функции важное значение имеют аргументы, передаваемые в эту функцию. Например рассмотрим функцию Solaris:

```
void spa_sync(spa_t *spa, uint64_t txg);
```

Ее первый аргумент — структура, связанная с пулом ZFS, а второй — номер текущей записываемой транзакции. Таким образом, используя аргументы проб можно получить имя пула и вывести его:

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

В данном примере использовалась типизированная форма аргументов args[*N*]. DTrace не всегда предоставляет информацию об их типах (например, такое ограничение вводится для проб, помеченных как нестабильные), и в таком случае предоставляются только аргументы arg*N*, имеющие тип uintptr_t, для которых нужно осуществлять явное приведение типов:

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

SystemTap использует отладочную информацию DWARF, которая содержит как имена и типы аргументов, так и локальных переменных. Они располагаются в отдельном пространстве имен: чтобы обратится к аргументу функции нужно указать символ $ и ее имя.

Однако, их значения могут быть не всегда доступны. Например, рассмотрим функцию vfs_read:

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

   /*Тело функции*/

   return ret;
}
```

К сожалению, GCC оптимизировал переменную ret, и она приняла неопределенное значение:

```
# 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
```

Таким образом, хотя возвращаемое значение мы получили $return, фактически, фактически значение переменной ret не известно SystemTap. Чтобы избегать подобных ситуаций необходимо использовать @defined — это выражение, проверяет, доступно ли в контексте пробы то или иное значение:

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

Полезным может оказаться и макрос @choose_defined. Более подробная информация:
http://sourceware.org/systemtap/wiki/TipContextVariables

Для удобства трассировки SystemTap предоставляет специальные строковые переменные, в которые уже записаны названия переменных и аргументов а также их значения:

- $$parms определяет аргументы функции
- $$locals определяет локальные аргументы функции
- $$vars — сумма parms и locals
- $$return — только возвращаемое значение

Пример информации, помещаемой $$vars, показан выше.

Кроме этого, SystemTap позволяет конвертировать аргументы и локальные переменные в строки, включая доступ к полям структур, если в конце имени переменной добавить $:

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

Контекст пробы

Контекст пробы

- Содержит текущее состояние системы на момент выполнения пробы:
  - Значения регистров
  - Контекст: текущий исполняемый процесс, поток/LWP, текущий процессор
  - Текущая исполняемая проба
- В DTrace выражены в виде встроенных переменных, например probefunc, в SystemTap — функций: execname()

22

2


[3]

Значения регистров пользовательского процесса в DTrace можно получить через встроенную переменную-массив uregs. В SystemTap — через Embedded-C и функцию ядра task_pt_regs. Также в Embedded-C функциях SystemTap доступна переменная CONTEXT, в которой в частности видны значения регистров (в том числе и ядра), см. реализацию функций uaddr() и print_regs().

Назначение	*DTrace*	*SystemTap*
Текущий исполняемый поток	curthread	task_current()
Номер текущего потока	tid	tid()
Номер текущего процесса	pid	pid()
Номер родителя текущего процесса	ppid	ppid()
Номер пользователя/группы, от которых запущен текущий процесс	uid/gid	uid()/gid(), euid(), egid()
Имя текущего процесса	execname curpsinfo->ps_fname	execname()
Аргументы коммандной строки	curpsinfo->ps_psargs	cmdline_*()
Номер процессора	cpu	cpu()
Имя пробы, функции, модуля и т. п.	probeprov, probemod, probefunc, probename	pp(), pn(), ppfunc(), probefunc(), probemod()

Более подробная информация:
https://wikis.oracle.com/display/DTrace/Variables#Variables-BuiltinVariables и
http://sourceware.org/systemtap/tapsets/context_stp.html

Предикаты

Предикаты

SystemTap

```
probe syscall.write {
   if(pid() != target())
       next;

   printf("Written %d
bytes", $count);
}
```

DTrace

```
syscall::write:entry
/pid == $target/
{
    printf("Written %d
bytes", args[3]);
}
```

Предикаты позволяют отметать ненужные пробы в начале их выполнения

23 SL 6.7 D [4]

Предикаты ставятся в начале пробы и позволяют отметать ненужные пробы настолько рано настолько это возможно, что позволяет экономить процессорное время.

В SystemTap предикаты строятся на базе классического оператора if, пропуск пробы осуществляется оператором next; в DTrace для этого предусмотрена специальная синтаксическая конструкция /predicate/. Т. к. языки C-подобны, возможно группировать условия с помощью операторов || и &&, а также инвертировать с помощью !.

Замечание: SystemTap также может трассировать своего потребителя (демона stapio). Для того, получить его pid, можно использовать функцию stp_pid()

Специальная макро-подстановка $target в DTrace или функция target() в SystemTap возвращает PID процесса, к которому привязана утилита через опции -p/-c в случае DTrace или -c/-x в случае SystemTap.

Переменные

Переменные

- Скалярные типы (int, uint32_t)
- Указатели
- Строки (string)
- Ассоциативные массивы
- Агрегации (статистики)
- Пользовательские C-типы (структуры, перечисления, объединения, массивы)
  - В DTrace поддерживаются по-умолчанию
  - В SystemTap — только через Embedded C

25

5.2

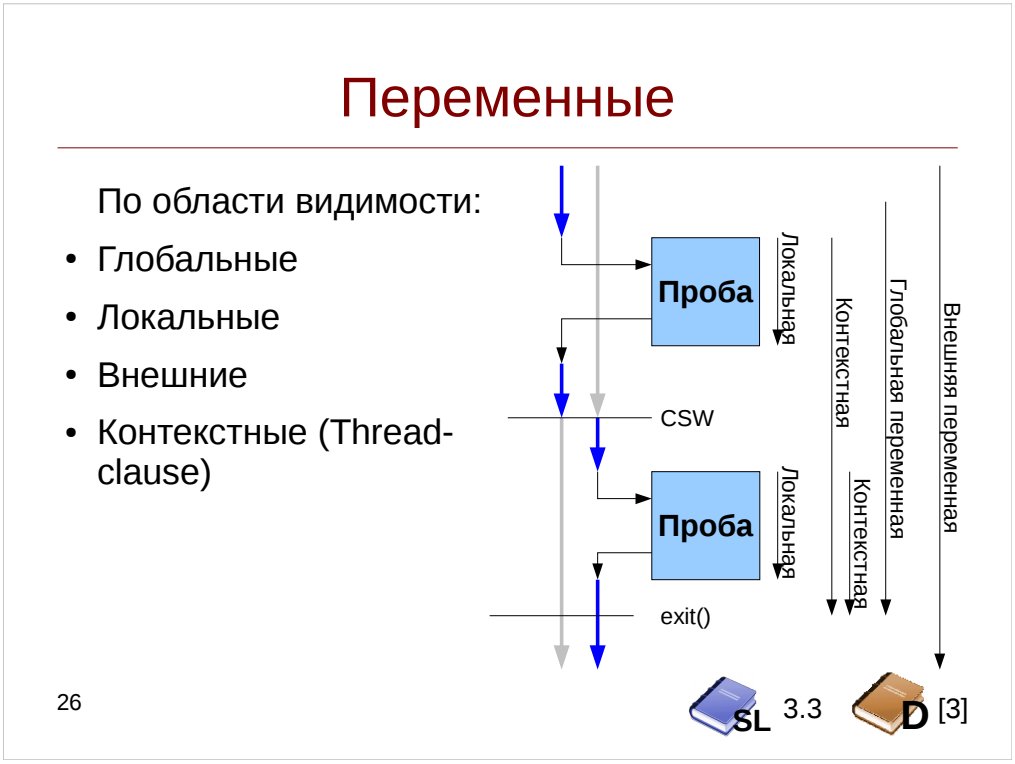

[3]

Язык D поддерживает большинство встроенных для ANSI C типов кроме разве что типов с плавающей точкой, включая пользовательские типы, объявляемые через enum или struct. Поддерживается также и ключевое слово typedef. Объявление типа возможно, но не обязательно во всех случаях — это зависит от области видимости переменной. В случае ошибки типизации, будет сгенерировано сообщение *operands have incompatible types*. Преобразование типов также производится в стиле C:

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

В SystemTap есть только два типа переменных: string (строки). и long (целые и указатели). Объявление типа переменных не производится, хотя имена типов потребуются при декларации функций, а тип определяется в момент первого присваивания — самого раннего в исходном коде. В случае если типы присваивания не совпадут (например строке присваивается интегральный литерал), будет сгенерировано сообщение *type mismatch*.

Парсер SystemTap не поддерживает пользовательские типы, поэтому требуется использовать Embedded C и соответственно Guru mode. Обращение к полям структур, переданных в качестве аргументов, однако возможно, равно как и преобразование через операцию @cast.



Также, как и в настоящих языках программирования, в SystemTap и DTrace переменные обладают областью видимости, и, как следствие, сроком жизни.

Самые «долгоживущие» - это *внешние переменные*, то есть экспортируемые ядром или приложением, к примеру это параметры модулей. В DTrace внешние переменные составляют отдельное пространство имен и обращение к нем ведется через косую кавычку:

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

В SystemTap получить их значение все же можно используя возможности Embedded C:

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

Кроме этого, начиная с SystemTap 2.7 можно использовать @var-выражение: @var("jiffies")

Далее следуют *глобальные переменные* — как следует из названия время их жизни — выполнение скрипта. Глобальные переменные необходимо декларировать в начале скрипта — в SystemTap с ключевым словом global:

```
global somevar;
```

В DTrace с явным указанием типа этой переменной (хотя строго говоря в случае DTrace явно объявлять глобальную переменную не обязательно):

```
uint32_t globalvar;
```

Самой короткой областью жизни обладают *локальные переменные* — их область видимости ограничивается одной пробой (в случае пробы-алиаса, все локальные переменные, присвоенные в tapset будут видны в ней). В SystemTap локальные переменные объявлять явно не нужно:

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

В DTrace локальные переменные используются всегда с префиксом this->. При необходимости их можно объявить в начале скрипта:

```
this uint32_t localvar;
```

Замечание: DTrace не проверяет области видимости для локальных переменных, и при выделении места под них не инициализурует их нулем. Например рассмотрим скрипт:

```
int global;
this int local;

syscall::read:entry {
     this->local++;
     global++; }

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

Запустим его на исполнение, на фоне запустив dd:

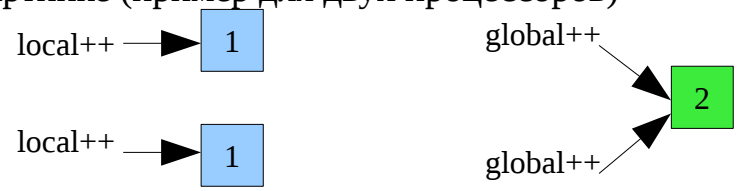
```
# dd if=/dev/zero of=/dev/null &
# dtrace -qs clauselocal.d
```

Как видим, значения global и local совпадают:

```
local: 26765 global: 26765
```

Запустим теперь несколько процессов dd на мультипроцессорной системе (теоретически эффект может проявиться и на однопроцессорной при преемптивной смене процесса, но дождаться этого тяжелее).

Теперь мы будем видеть, что значения local могут не совпадать. Связано это с тем, что переменные local постоянно выделяются, а global — едина для всех, как показано на картинке (пример для двух процессоров)



Наконец, последний класс переменных — thread-local, то есть *контекстные*, область видимости которых — поток исполнения, а время жизни — от первого ее обращения до уничтожения потока, к которому они привязаны (или завершения работы скрипта). Главное отличие их от всех остальных переменных — при переключении контекста — меняется и набор привязанных к нему переменных, что крайне удобно при сборе попроцессной статистики.

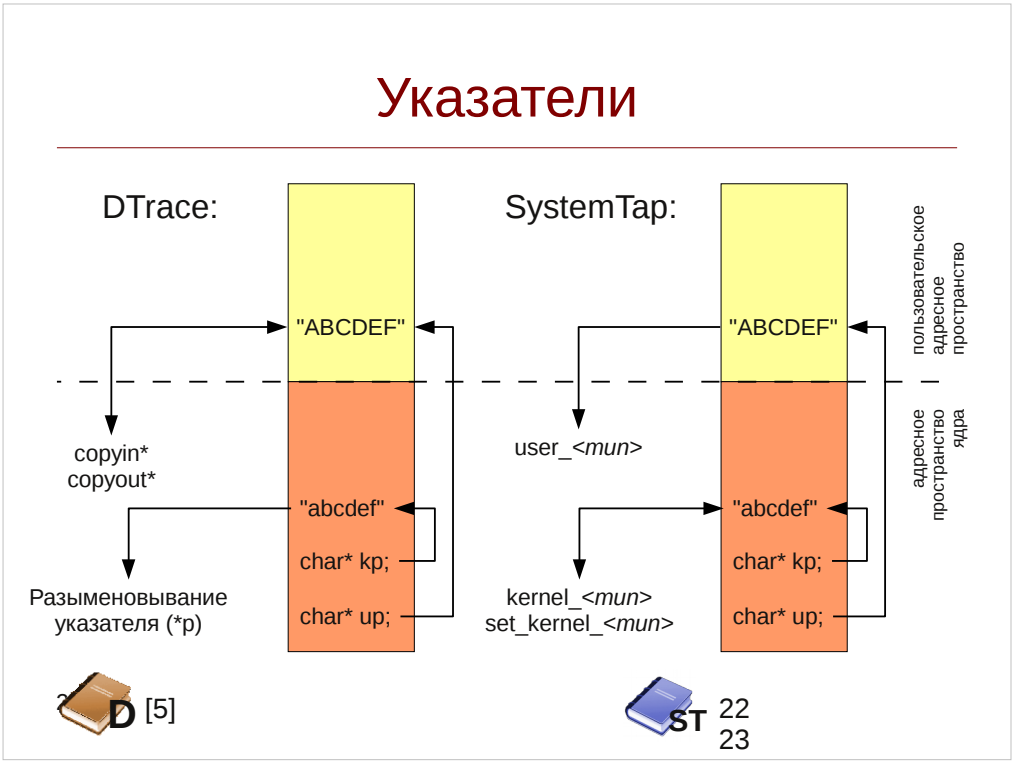
По сути контекстная переменная — это ассоциативный массив, в котором ключом является номер потока:

```
global time;
syscall.read {
     do_trace[tid()] = 1;
}
syscall.read.return {
     delete do_trace[tid()];
}
```

В DTrace для этого есть синтаксический сахар в виде префикса self:

```
syscall::read:entry {
     self->do_trace = 1;
}
syscall::read:return {
     self->do_trace = 0;
}
```

В этом примере используется специальный флаг — do_trace, который позволяет используя предикаты трассировать только те потоки, которые в данный момент исполняют системный вызов read(2) и игнорировать остальные.



Язык SystemTap в явном виде не поддерживает работу с указателями — все указатели там воспринимаются как переменные типа long (разумеется, явная работа с указателями поддерживается в Embedded C). DTrace имеет внутри себя поддержку C-подобных массивов, указателей и даже динамического выделения памяти.

При обращении к переменным по их указателю основная проблема — риск обратиться по неверному адресу (и как следствие вызвать панику ядра), поэтому DTrace и SystemTap вынуждены:

- Для пользовательского адресного пространства — проверять на принадлежность адрес соответствующему сегменту, сравнивая адрес с базовым адресом ядра, а также добавлять дополнительные инструкции в обработчики страничных сбоев (page fault).
- Проверять на принадлежность адреса запретным зонам (например области OpenFirmware в DTrace для SPARC)

Средства динамической трассировки всегда работают в режиме ядра, однако иногда требуется доступ к данным в пользовательском пространстве, например при трассировке пространства пользователя. Также это происходит при системных вызовах, например таких как open — строковое значение параметра pathname при вызове находится в пространстве пользователя.

В DTrace доступ к переменным пространства ядра осуществляется простым разыменованием указателя, например функция fop_open принимает указатель на указатель на vnode, таким образом, чтобы получить указатель на vnode_t, нужно сначала разыменовать аргумент args[0]:

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

Замечание: изменять переменные ядра в DTrace невозможно!

Для чтения из пользовательского пространства служат функции copyin, copyinstr, copyinto, например скопируем первую запись из массива fds, передаваемого системному вызову poll(2):

```
# 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);  }'
```

В SystemTap работа с внешними переменными ведется с помощью функций (определены в tapset conversions.stp и converstions-guru.stp):

- kernel_<тип> - пытается считать данные по указателю из пространства памяти ядра, например после записи позиция в файле будет отличаться от той, что была изначально, для чего вызов vfs_write принимает указатель на поле или переменную, хранящее эту позицию:

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

- set_kernel_<тип> устанавливает значение переменной в пространстве ядра (требуется Guru-mode)
- user_<тип> читает значение переменной по указателю из пространства пользователя

В Embedded C для безопасного доступа к данным ядра необходимо использовать функцию kread().

При попытке доступа к некорректному адресу, будут сгенерированы сообщения об ошибке. Для DTrace оно выглядит следующим образом:

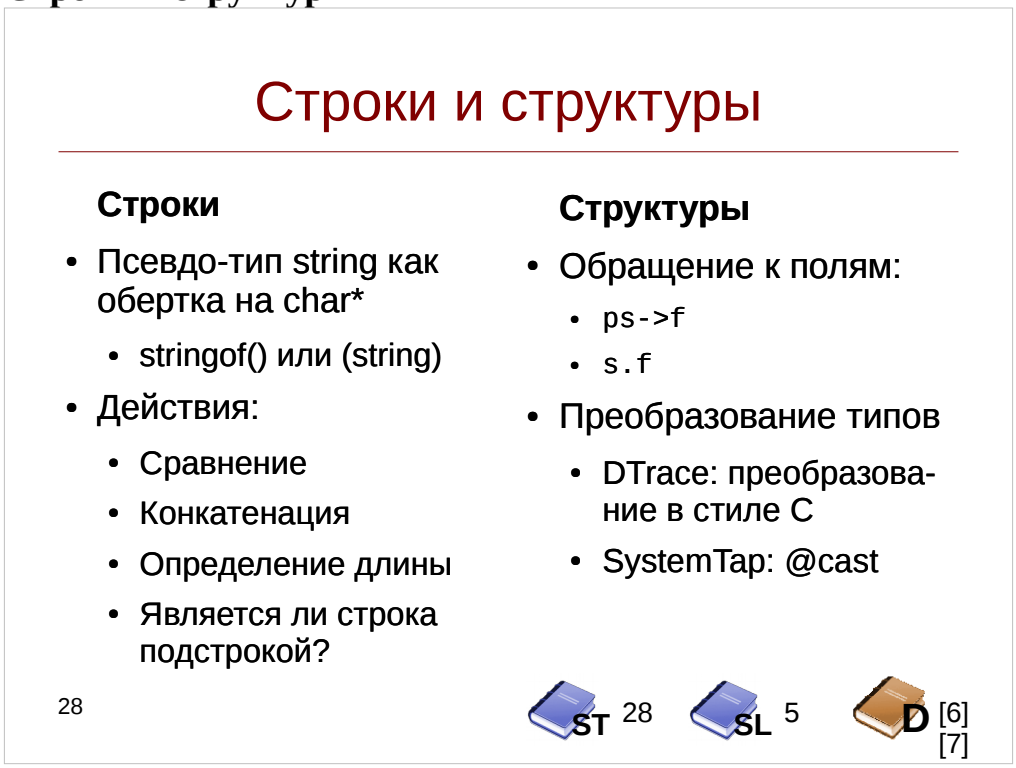
```
dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address
(0x4) in action #1 at DIF offset 16
```

В SystemTap будет выведено следующее сообщение:

```
ERROR: kernel string copy fault at 0x0000000000000001 near identifier
'kernel_string' at /usr/share/systemtap/tapset/conversions.stp:18:10
```

Для того, чтобы выявить эти ошибки, DTrace и SystemTap специальным образом обрабатывают страничные сбои. Иногда корректные адреса также могут вызывать ошибки, если данные например еще не были загружены в память.

Строки и структуры



Строки в DTrace и SystemTap представляют из себя обертку над типом char* (т. е. нуль-терминирумые строки), однако если в SystemTap — это простой синоним, то в DTrace тип string имеет ряд ограничений (например, нельзя получить доступ к символу строки по его индексу).

Преобразование к типу string в DTrace выполняется посредством преобразования в C-стиле, то есть: (string) arg0 или оператора stringof() – любой скалярный тип может быть преобразован с помощью них в строку. Получить строку из пользовательского адресного пространства можно с помощью подпрограммы copyinstr().

В SystemTap получить строки можно с помощью семейства функций kernel_string*() или user_string*(), а вот преобразование выполняется с помощью sprint() и sprintf().

Строки поддерживают следующие операции:

- Сравнение выполняется с помощью операторов ==, >=, <=, !=, >, < которые семантически повторяют функцию strcmp.
- Конкатенация выполняется с помощью функции strjoin (DTrace) или оператора конкатенации **.** (SystemTap).
- Длину строки можно выяснить с помощью функции strlen
- Выяснить, содержится ли строка в искомой подстроке можно с помощью функции isinstr (SystemTap) или strstr (DTrace).

Т. к. в основе программирования ядер Linux и Solaris лежит ООП-парадигма, то значительное количество данных представляется в виде структур C, например:

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

```
        struct dentry *dentry;
};
```

Обращение к полям структуры производится с использованием C-подобного синтаксиса, однако если в DTrace в зависимости от типа нужно использовать операторы . и ->, в SystemTap поддерживается только оператор-стрелка.

Информация о типах хранится в специальных секциях — CTF в Solaris/BSD и DWARF в Linux и для структур сохраняет типы и имена полей. Для преобразования типов (например, если в функцию передается нетипизованный указатель void*) в DTrace используется C-подобный синтаксис:

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

В SystemTap преобразование производится аналогичным образом, но с использованием оператора @cast:

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

В качестве третьего параметра оператора @cast можно использовать путь до заголовочного файла, содержащего описание структуры.

Контрольные вопросы

1. Каково назначение опций коммандной строки -c и -x в DTrace и SystemTap?

_____
_____
_____
_____
_____

2. Как объявляются глобальные переменные в DTrace и SystemTap?

_____
_____
_____
_____
_____

3. В каких переменных располагаются возвращаемые значения в return-пробах?

_____
_____
_____
_____
_____

4. Как узнать внутри пробы ее имя и соответствующий ей модуль?

_____
_____
_____
_____
_____

Упражнение 1

Напишите скрипты opentrace.d и opentrace.stp, которые бы трассировали системные вызовы open(). На каждый вызов в одну строчку должна выводится следующая информация:

- Контекст вызова: имя исполняемого файла, номер процесса, номера пользователя и группы от которой исполняется процесс.
- Путь до открываемого файла
- Маска флагов, включая флаги O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT — в виде строки
- Возвращаемое значение вызова

Например:

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

Значения битовых флагов системного вызова open() приведены в следующей таблице:

Флаг	*Solaris*	*Linux (x86)*
O_RDONLY	биты 0-1 не установлены	
O_WRONLY	1	1
O_RDWR	2	2
O_APPEND	8	1024
O_CREAT	256	64

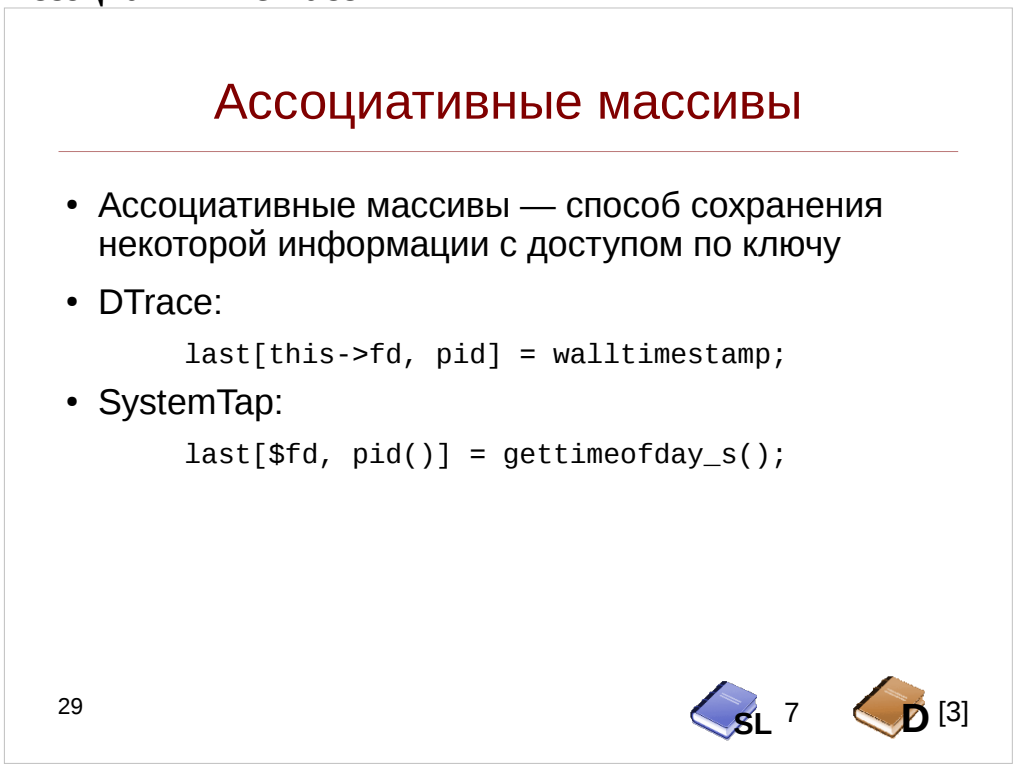
Протестируйте корректность функционирования скрипта, экспериментируя с утилитой коммандной строки tee и перенаправлением в коммандном интерпретаторе:

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

Замечание. В Solaris 11 системный вызов open() заменен более универсальным системным вызовом openat().

Опционально: дополните скрипты так, чтобы выводились только те файлы, в имени которых содержится /etc.

Ассоциативные массивы



Ассоциативные массивы — это последовательности значений с доступом по одному или нескольким ключам. В качестве ключей выступают любые типы данных, однако внутренняя реализация массивов использует их хеши.

Ассоциативные массивы удобно использовать для сохранения истории событий или состояний, например последнее действие, выполняемое над открытым процессом файлом.

Создать новый элемент можно присвоив ему некоторое значение:

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

В DTrace также потребуется предварительно объявить массив и типы его ключей:

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

Чтобы удалить запись из ассоциативного массива, в DTrace нужно присвоить ей значение 0, а в SystemTap использовать инструкцию delete. Если значение, соответствующее ключу, никогда не присваивалось или было удалено, вы получите значение 0 как значение по-умолчанию.

В DTrace доступ к элементу возможен только по его ключу, в SystemTap возможен также обход элементов с помощью оператора foreach:

```
foreach([pid+, fd] in last_fop limit 100) {
```

```
        printf("%d\t%d\t%s\n", pid, fd, last_fop[pid, fd]);
    }
```

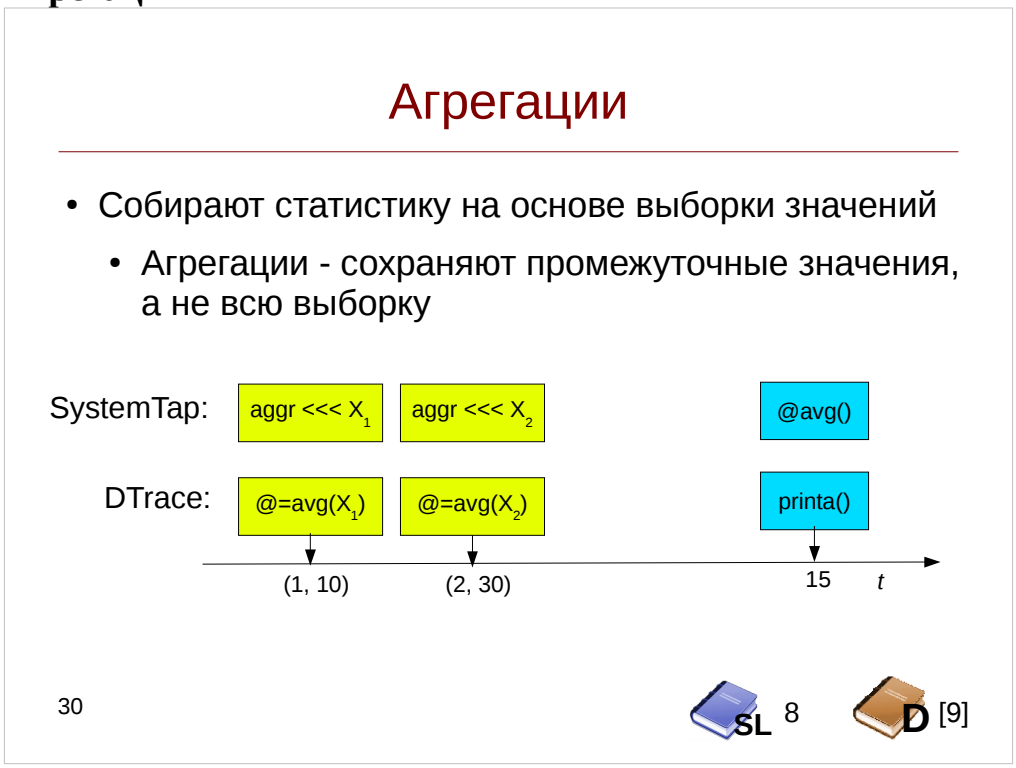
В квадратных скобках указывается список переменных-ключей, знаки + или - после нее означает сортировку в возрастающем или убывающем порядке соответственно (поддерживается только сортировка по одному ключу).

Размер ассоциативного массива в SystemTap ограничен константой времени компиляции MAXMAPENTRIES или же для каждого массива индивидуально:

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

Замечание: В SystemTap 2.1 было добавлено улучшение для поддержки многопоточности в ассоциативных массивах — теперь для каждого возможного процессора выделяется до MAXMAPENTRIES, при этом вся память выделяется единомоментно. При этом для хранения строк используется статически выделяемый буфер. В результате для ассоциативного массива со строковым ключом потребуется как минимум NR_CPUS * MAXMAPENTRIES * MAP_STRING_LENGTH. На CentOS 7.0 x86_64 с настройками по-умолчанию это дает 128 мегабайт!

Агрегации



Наиболее полезный инструмент при оценке производительности представляет агрегация (в SystemTap они носят названия статистик, statistics) — для агрегации для каждого ключа сохраняется при добавлении нового значения обновляется промежуточное состояние, а на основании этого состояния вычисляется агрегирующая функция, например среднее арифметическое или строится гистограмма. Например, для среднего арифметического — это количество сохраненных значений и их сумма.

В DTrace агрегации располагаются в отдельном пространстве имен — все их имена начинаются с @. Самое короткое имя агрегации — @ является алиасом для @_, что удобно для использования в маленьких скриптах. К тому же по завершении скрипта информация, собранная в этой агрегации, автоматически выведется в консоль.

В SystemTap агрегации – это переменные специального типа, для которых осуществляется не операция присвоения (=), а добавления с помощью оператора <<<. Для доступа по ключу необходимо работать с ассоциативным массивом агрегаций.

Функции:

- count — количество значений
- sum — сумма значений
- min/max/avg — минимальное, максимальное и среднее
- stddev (отсутствует в SystemTap) — стандартное отклонение
- lquantize (hist_linear в SystemTap) — линейная гистограмма
- quantize (hist_log в SystemTap) — логарифмическая гистограмма

	DTrace	*SystemTap*
Добавление	`@aggr[keys] = func(value);`	`aggr[keys] <<< value;`

Вывод	`printa(@aggr);`
	`printa("строка формата",`
	`@aggr);`
		`foreach([keys] in aggr) {`
		`print(keys,`
		`@func(aggr[keys]));`
		`}`

Вывод нескольких агрегаций	`printa("%10d %20d", @aggr1, @aggr2);`	-
Очистка	`clear(@aggr);`	`delete aggr;`
Нормализация	`normalize(@aggr, value);` `denormalize(@aggr);`	… `@func(aggr) / value` …

Выбор значений	`trunc(@aggr, num);`	`foreach([keys] in aggr`
		`limit num) {`
		`print(keys,`
		`@func(aggr[keys]));`
		`}`

⚠ *Замечание*: в примерах на выбор значений и вывод для SystemTap подразумевается, что агрегация является также и ассоциативным массивом.

⚠ *Замечание*: В DTrace при выводе нескольких агрегаций можно выполнять сортировку по одному из значений с помощью опций aggsortkey, aggsortpos, aggsortkeypos и aggsortrev.

⚠ *Замечание*: функция clear() в DTrace очищает агрегацию, но сохраняет ключи. Чтобы очистить и ключи можно использовать вызов trunc без параметра num.

Агрегации позволяют создавать stat-подобные утилиты. Так, реализуем скрипт, подсчитывающий количество системных вызовов write и записанных килобайт программами.

Листинг 2. Скрипт wstat.d

```
#pragma D option aggsortkey
#pragma D option aggsortkeypos=0

syscall::write:entry
{
        @wbytes[pid, execname, arg0] = sum(arg2);
        @wops[pid, execname, arg0] = count();
}

tick-1s
{
        normalize(@wbytes, 1024);
```

```
        printf("%5s %12s %3s %7s %7s\n",
                 "PID", "EXECNAME", "FD", "OPS", "KBYTES");
        printa("%5u %12s %3u %7@d %7@dK\n", @wops, @wbytes);
        clear(@wbytes);
}
```

Обратите внимание, что в строке форматирования printa сначала следуют ключи ассоциативного массива, а за ними — агрегации в том же порядке, в котором они следуют в качестве параметров вызова. Сортировка будет выполняться по номеру процесса PID а не по количеству операций или записанных байт. Опция aggsortkeypos здесь не обязательна, так как по-умолчанию сортировка выполняется по нулевому ключу.

В SystemTap код будет выгдядеть похожим образом, однако подобие printa придется реализовать самостоятельно. Зато в SystemTap для этих целей придется держать лишь одну переменную для агрегации:

Листинг 3. Скрипт wstat.stp

```
global wstat;

probe syscall.write {
    wstat[pid(), execname(), fd] <<< count;
}

probe timer.s(1) {
    printf("%5s %12s %3s %7s %7s\n",
        "PID", "EXECNAME", "FD", "OPS", "KBYTES");

    foreach([pid+, execname, fd] in wstat) {
        printf("%5d %12s %3d %7u %7u\n",
                    pid, execname, fd, @count(wstat),
                    @sum(wstat) / 1024);
    }

    delete wstat;
}
```

Вывод для DTrace и SystemTap будет выглядеть схожим образом:

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

Время

Время

DTrace:

- timestamp
- vtimestamp
- walltimestamp
- `printf("%Y", walltimestamp);`

SystemTap:

- jiffies()
- get_cycles()
- gettimeofday_*();
- local_clock_*()
- cpu_clock_*(cpu)
- `printf("%Y", gettimeofday_s());` или ctime()

31

ST 3 D [3]

Человек привык жить, используя календарь и 24-часовое представление времени. Сейчас для этого применяется стандарт всемирного координированного времени UTC. Для большинства процессов внутри ядра такая скрупулезность не нужна, поэтому внутри него существует множество разных источников времени, и узнать время можно с помощью разных функций DTrace и SystemTap.

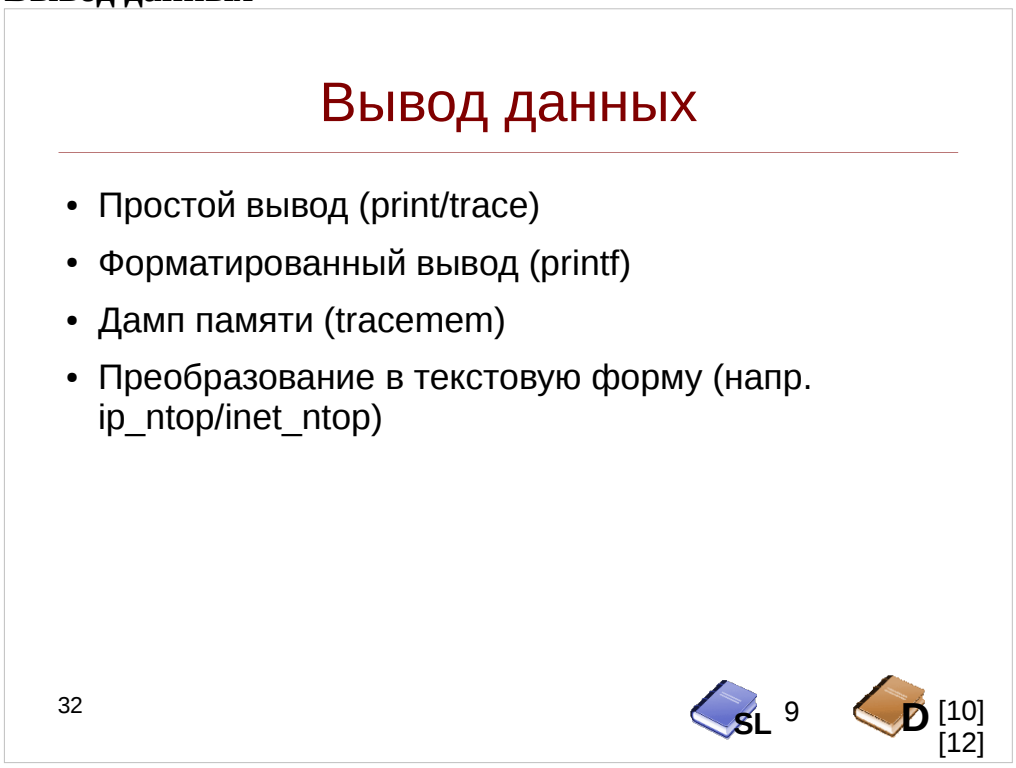
Назначение	*DTrace*	*SystemTap*
Системный таймер — отвечает за обработку периодических событий в ядре, например переключение процессов. Заметим, что интервал системного таймера разработчики ядер используют как самостоятельную единицу времени и называют ее тиком, lbolt или jiffy.	`lbolt и `lbolt64	jiffies()
Счетчик тактов процессора — специальный регистр процессора, подсчитывающий количество отработанных процессором тактов, например TSC в x86 или %tick в SPARC. Не обязан быть монотонным.		get_cycles()
Монотонное время с момента загрузки системы. Может использовать таймеры высокого разрешения процессора (HPET), однако не гарантирует синхронность на всех ядрах процессора.	timestamp	local_clock_X() и cpu_clock_X(N)

Назначение	*DTrace*	*SystemTap*
Виртуальное монотонное время потока. То же, что и предыдущий счетчик, но не считает периоды, когда поток снят с процессора.	vtimestamp	
Реальное время, прошедшее с начала эпохи Unix (т. е. 1 января 1970 года 00:00 по UTC).	walltimestamp	gettimeofday_X()

Здесь: X — суффикс единицы времени (s — секунды, ms — миллисекунды, us — микросекунды и ns — наносекунды), N — номер процессора. Счетчики DTrace всегда имеют наносекундную точность.

В общем случае, монотонные счетчики лучше использовать для измерений интервалов времени, а реальное время для логирования событий с понятной временной меткой. Для вывода временной метки можно использовать специальный формат %Y или функцию ctime() в SystemTap.

Вывод данных



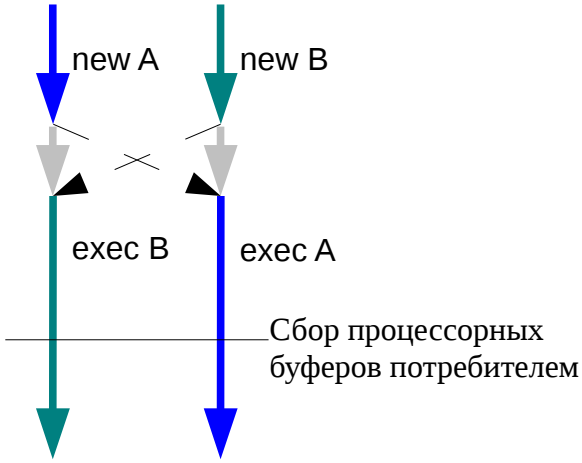
Как уже говорилось, DTrace и SystemTap выводят данные во временный буфер, которые потом программа-потребитель перенаправляет в терминал или файл. SystemTap кроме того поддерживает несколько транспортов через relayfs, позволяющие выводить данные немедленно с использованием функций log и warn.

Функции вывода можно разделить условно на несколько типов:
Функции вывода можно разделить условно на несколько типов:

- При простом выводе один или несколько объектов выводятся безо всякого форматирования. В DTrace это выполняется с помощью функции trace, принимающий один аргумент. В SystemTap — print/println, printd/printdln, выводящие один или несколько аргументов с переносом на следующую строку и/или разделителем.
- Форматированный вывод осуществляется посредством функции printf. Строка форматирования строится по правилам, аналогичным принятым в языке C, хотя и с некоторыми расширениями. Подробности можно найти в документации по SystemTap и DTrace
- Вывести дамп памяти можно с помощью функции tracemem (поддерживается только в DTrace). В SystemTap реализуемо средствами самого языка (Цикл for + kernel_int + printf).
- Также, некоторые структуры данных имеют специальные функции для получения их состояния в текстовом виде, например для форматирования 4-байтного IP-адреса предусмотрены функции inet_ntop (DTrace) и ip_ntop (SystemTap)

Для того, чтобы исключить конфликты за доступ к буферу вывода в многопроцессорных системах, а также ускорить процесс и исключить излишние переключения контекста между процессом-потребителем и ядром, SystemTap и

DTrace выделяют отдельные буферы для каждого процессора, которые затем по таймеру обходятся, данные из них собираются и передаются процессу-потребителю.



Такой подход несколько запутывает трассировку комплексных событий. Представим себе, что на процессоре 0 порождается (new) запрос A, а на процессоре 1 — запрос B. После чего запрос A планируется на исполнение потоком на процессоре 1 и наоборот для запроса B (показано на рисунке). Таким образом вывод будет выглядеть как:

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

Это несколько запутывает его интерпретацию, особенно когда самих запросов и событий десятки (например при трассировке конвеера ZIO в файловой системе ZFS).

Решить эту проблему можно выводя вместе с каждым сообщением pid процесса или другой идентификатор запроса и затем группировать записи при пост-обработке.

Спекуляции

Спекуляции

- Создать новый буфер: speculation()
- Выбрать буфер для трассировки: speculate()
- Вывести данные из буфера: commit()
- Отбросить буфер: discard()

33

31


[13]

Мы уже говорили о предикатах, как о способе отбросить ненужные пробы, и сократить вывод скрипта. Но что если трассировка запросов происходит во множестве проб, а решение о выводе принимается в последней из них? Решить такую проблему позволяют спекуляции.

Для каждого из запросов создается независимый буфер посредством функции speculation(), которая возвращает номер буфера. Его имеет смысл поместить в ассоциативный массив, в качестве ключа выбрав идентификатор запроса — например, указатель на соответствующую структуру. При завершении обработки запроса можно или отбросить буфер или переместить вывод в основной буфер вывода. Максимальное количество буферов в DTrace регулируется переменной nspec.

Вывод в буфер осуществляется с помощью функции speculate(). В SystemTap она имеет параметр output, на место которого подставляется строка, записываемая в спекуляцию (ее можно получить с помощью sprintf). В DTrace в отличие этого параметра нет, вместо этого она «переключает» все следующие за ней функции вывода на вывод в буфер спекуляции.

Пример использования спекуляций вы найдете в разделе «Блочный ввод-вывод» на с. 142.

Трансляторы и tapset'ы

Трансляторы и tapset'ы

- DTrace — трансляторы

```
struct stat_info {
    long long st_size;
};
translator struct stat_info < uintptr_t s > {
    st_size = * ((long long*)
        copyin(s + offsetof(struct stat64_32, st_size),
        sizeof (long long)));
};
```

- SystemTap — tapset'ы

```
probe lstat = kernel.function("sys_newlstat").return,
              kernel.function("sys32_lstat64").return {
    filename = user_string($filename);
    size = user_uint64(&$statbuf->st_size);
}
```

34

3.2


[40]

Мы уже обсуждали проблему обеспечения стабильности проб. Кроме того, что могут меняться структуры данных с течением времени, несколько вариантов одной структуры могут сосуществовать в одном ядре, например 32-х битные и 64-х битные варианты структуры stat. Рассмотрим, как можно унифицировать доступ к полям такой структуры.

В DTrace для этого предназначен механизм трансляторов, как показано в скрипте stat.d. Сначала реализуется сам транслятор, который описывает правила преобразования исходной структуры данных stat64_32, соответствующей 64-битной структуре stat для программ с 32-х битным ABI, в целевую, известного формата — предварительно определенной структуре stat_info. Затем, в коде самой пробы вызывается оператор xlate, собственно осуществляющий преобразование.

Листинг 4. Скрипт stat.d

```
struct stat_info {
    long long st_size;
};

translator struct stat_info < uintptr_t s > {
    st_size = * ((long long*) copyin(s + offsetof(struct stat64_32, st_size),
            sizeof (long long)));
};

syscall::fstatat64:entry
/arg1 != 0/
{
        self->filename = copyinstr(arg1);
        self->statptr = arg2;
```

```
}

syscall::fstatat64:return
/arg1 == 0 && self->statptr != 0/
{
        printf("STAT %s size: %d\n", self->filename,
                xlate<struct stat_info*>(self->statptr)->st_size);
}
```

Стандартные трансляторы DTrace вы найдете в директории /usr/lib/dtrace/

В SystemTap механизма трансляторов нет, зато есть возможность создать prolog- (или epilog-) алиас, выполняющий необходимые преобразования. Такие пробы-алиасы группируются в скрипты-библиотеки (tapset'ы) и помещаются в директорию /usr/share/systemtap/tapset. Большинство рассматриваемых в следующих главах проб реализованы в виде проб-алиасов.

Так, в Linux для системного вызова stat() существует несколько вариаций для разного формата структуры stat и разной битности вызывающей программы. С помощью следующего tapset'а мы нивелируем эти различия так что переменная size будет содержать актуальный размер файла для всех случаев. Заметим, однако что данный пример несколько надуман, и в реальной жизни куда удобнее привязаться к вызову функции vfs_lstat, имеющей одинаковых формат аргумента stat для всех ABI данного вызова.

Листинг 5. Скрипт /usr/share/systemtap/tapset/lstat.stp

```
probe lstat = kernel.function("sys_lstat64").return ? ,
              kernel.function("sys32_lstat64").return ? {
    filename = user_string($filename);
    size = user_uint64(& @cast($statbuf, "struct stat64")->st_size);
}

probe lstat = kernel.function("sys_newlstat").return ? {
    filename = user_string($filename);
%( arch == "x86_64"
%?  size = user_uint64(& @cast($statbuf, "struct stat")->st_size);
%:  size = user_uint32(& @cast($statbuf, "struct stat")->st_size);
%)
}
```

Чтобы создать epilog-алиас (он выполняется после кода пробы), нужно использовать оператор += вместо оператора = . Суффикс "?" используется, чтобы избежать ошибок, если какие-либо функции отсутствуют в ядре.

Замечание: приведенный в примере tapset реализован только для Intel-архитектуры. Для платформ PowerPC, ARM и S/390 требуется вставить

дополнительные проверки.

После этого созданные алисы можно использовать так:

Листинг 6. Скрипт lstat.stp

```
probe lstat {
    printf("%s %s %d\n", pp(), filename, size);
}
```

Также, в скриптах динамической трассировки иногда необходимо определять некоторые константы, соответствующие константам ядра. В DTrace для этого предназначена конструкция enum, аналогичная конструкции из языка C, а также определение глобальных переменных через модификатор inline:

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

В SystemTap также можно определять константы в tapset'ax:

```
global TASK_RUNNING = 0;
```

Также и в SystemTap (только в Embedded C-коде) и DTrace (при использовании опции -C) доступна директива препроцессора #define.

Контрольные вопросы

1. Что такое контекстные (thread-clause) переменные? Каким образом можно реализовать их в SystemTap?

   _____
   _____
   _____
   _____
   _____

2. Каким образом обратиться к данным по указателю в DTrace и SystemTap?

   _____
   _____
   _____
   _____
   _____

3. С помощью какого типа переменной можно организовать подсчет количества системных вызовов по процессу?

   _____
   _____
   _____
   _____
   _____

4. В чем разница между переменными timestamp, vtimestamp, walltimestamp в DTrace?

   _____
   _____
   _____
   _____
   _____

Упражнение 2

Переработайте скрипты из упражнения 1 так, чтобы они для процессов, выполняющихся в системе подсчитывали:

- Количество попыток открытия файла
- Количество попыток создания файла
- Суммарное количество успешных попыток

С интервалом, задаваемым параметром коммандной строки должны выводиться:

- Текущие дата и время в человеко-понятном формате
- Таблица, содержащая указанные показатели, а также имя исполняемого файла и номер процесса

После чего показатели должны сбрасываться.

Для демонстрации скриптов предлагается воспользоваться модулем `file_opener`. Этот модуль заполняет директорию (задается в параметре `root_dir`) файлами необходимого количества (регулируется параметром `created_files`), а затем генерирует число от 1 до `max_files` и пытается открыть или создать файл (в зависимости от параметра запроса create.

Запустите свои скрипты и несколько экземпляров нагрузчика (директория и количество предварительно создаваемых файлов варьируется):

```
# 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
```

Дайте объяснения различия в показателях между процессами.

Модуль 3. Основы динамической трассировки

Применение трассировки

Заявленные нами цели трассировки ядра включают в себя сбор статистики и анализ производительности, динамическую отладку ядра и аудит системы. Покажем, как инструменты динамической трассировки помогают решать эти задачи. Представим себе ситуацию, в которой пользователи открывают файлы, тогда в первую очередь нам нужно собрать следующие данные:

- Для *анализа производительности*, если например пользователи жалуются на слишком медленное открытие файла, нам потребуется и измерить время, затрачиваемое пользователем на системные вызовы open() и read(), и если это время существенно, необходимо двигаться дальше по стеку виртуальной файловой системы, чтобы локализовать проблему. Например, можно измерить время поиска файла в директории (уровень драйвера файловой системы) или время доступа диска (уровень блочного ввода-вывода).
- Для *динамической отладки* системы нам потребуется выяснить код ошибки, сохраняемый в пользовательской переменной errno. Эти значения возвращаются системным вызовом и используемыми им вызовами. Техника их получения будет показана в разделе «Динамический анализ кода» на с. 66.
- Для *аудита системы* нам нужно собрать пути до файлов и uid пользователей, их открывающих, чтобы отследить попытки доступа к системным файлам.

Все эти три задачи могут быть решены подобными скриптами:

```
# 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("<= uid: %d pid: %d ret: %d %lld\n",
                uid, pid, arg1, (long long) timestamp);
}'

# 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("<= uid: %d pid: %d %d %d\n",
                        uid(), pid(), $return, local_clock_ns());
        }'
```

Выводит он следующие данные:

```
=> uid: 60004 pid: 1456 open: /etc/shadow 16208212467213
<= uid: 60004 pid: 1456 ret: -1 16208212482430
```

Во-первых мы таким образом измерили время, затраченное на системный вызов open():

16208212482430 — 16208212467213 = 15217 = 15.2 мкс

Во вторых, мы видим, что пользователь получил ошибку (об этом свидетельствует код возврата -1), и можем продолжить поиск причины, трассируя функции, вызываемые из системного вызова open(). Наконец, с точки зрения задачи аудита, мы видим, что пользователь с uid = 6004 попытался получить доступ к файлу /etc/shadow, что обычным пользователям запрещено делать.

В этом модуле мы рассмотрим некоторые аспекты применения анализа трасс, получаемых в результате работы систем динамической трассировки. В дальнейшем, в модулях 4 и 5, в которых пойдет речь о конкретных провайдерах, используемых для трассировки операционных систем и приложений, все примеры будут даваться именно в ключе трассировки событий, а необходимые средства обработки трасс предлагается делать самостоятельно.

Динамический анализ кода

Динамический анализ кода (Wikipedia) — анализ программного обеспечения, выполняемый при помощи выполнения программ на реальном или виртуальном процессоре (анализ, выполняемый без запуска программ называется статический анализ кода). Простейшие инструменты динамического анализа кода: печать стека и дерева вызовов — помогают разобраться в незнакомом модуле операционной системы.

В ходе исполнения кода предпочтительней всего использовать регистры общего назначения, как наиболее быструю по времени доступа память. Однако при совершении вызова внешней функции, она также будет их использовать, и перед вызовом необходимо сохранять состояние регистров, включая счетчик инструкций, чтобы по окончании выполнения внешней функции она перешла на верную инструкцию. Сохраненное состояние регистров, локальные переменные и адреса возврата сохраняются в стеке — специально отведенной области памяти. Если вычленить из стека указатели возврата и аргументы функций, то можно получить немало полезной информации. Например, рассмотрим следующий стек вызов, полученный из дампа паники ядра Solaris:

```
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)
...
```

Если посмотреть на него, можно видеть, что проблема возникла в подсистеме ZAP при монтировании файловой системы. Второй аргумент функции zfs_domount — это название проблемного датасета. Переведя этот датасет в readonly-режим мы смогли запустить систему.

В DTrace функции вывода стека могут использоваться как ключи ассоциативных массивов, так и как самостоятельные функции (в этом случае, они выведут результат в буфер): stack() - текущий стек ядра, ustack() - пользовательского процесса, jstack() - Java-приложения:

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

Опциональный параметр этих функций — предельная глубина стека.

В SystemTap стек можно вывести с помощью семейства функций:

- backtrace(), ubacktrace() возвращает список в виде списка адресов в 16-ричном формате.
- print_stack() и print_ustack() соответственно печатают его и преобразуют адреса к именам символов там где это возможно.
- print_backtrace() и print_ubacktrace() печатает стек сразу
- task_backtrace() выводит стек ядра по указателю на его task_struct

Пример:

```
# 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(); } '
```

Также иногда бывает полезным преобразование известного адреса в соответствующую ему имя функции (т. е. символ, symbol):

	DTrace	*SystemTap*
Символ (польз.)	`usym(%ip) или ufunc(%ip)`	`usymname(%ip)`
Символ (польз.) + смещение	`uaddr(%ip)`	`usymdata(%ip)`
Имя модуля (польз.)	`umod(%ip)`	`umodname(%ip)`
Символ (ядро)	`sym(%ip) или func(%ip)`	`symname(%ip)`
Символ (ядро) + смещение	-	`symdata(%ip)`
Имя модуля (ядро)	`mod(%ip)`	`modname(%ip)`
Имя модуля, символ + смещение (ядро)	`printf("%a", %ip)`	

Например:

```
# 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)); }'

# 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); }'
```

В данном примере при вызове функции открытия файла на уровне VFS скрипт напечатает название модуля и функции соответствующего реальной файловой системе драйвера.

Замечание: в DTrace преобразование имен выполняется потребителем, а не интерпретатором DIF, поэтому они не возвращают строк, которые потом можно было бы использовать в строковых функциях. Кроме того, при трассировке приложений если процесс уже завершился к моменту обработки буфера потребителем, DTrace будет неоткуда взять символы, и функции usym* не смогут обработать адрес.

Посмотрим теперь, как можно отследить путь выполнения программы, построив дерева вызовов функций. Для этого в скрипте создается флаг (например, глобальная переменная с именем traceme), при вызове некоторой функции он устанавливается, а при выходе из нее — сбрасывается. Пробы привязываются ко всем функциям ядра, а в качестве предиката указывается проверка этого флага. При этом при каждом вызове функций увеличивается отступ, чтобы вывод имел древовидную структуру.

Предупреждение. Чтобы привязаться ко всем пробам ядра в SystemTap (исключая модули), следовало бы использовать конструкцию kernel.function("*"). Однако такой подход обычно приводит к панике ядра, так что крайне рекомендуется в SystemTap ограничивать область привязки проб, как это сделано в примере. Конструкция DTrace fbt::: достаточно безопасна и может вызывать лишь небольшое замедение работы системы.

Вывод дерева вызовов функций может быть удобен также и при поиске сбоя в ядре, исходя из кода возвращаемой ошибки. Рассмотрим пример, в котором выполняется команда cat not_exists, в которой естественно возвращается ошибка ENOENT.

В SystemTap для этого можно использовать функции indent/thread_indent. Она содержит внутренний строковой буфер, а в качестве параметра передается значение,

указывающее на сколько увеличить или уменьшить отступ. Ее можно использовать следующим образом:

Листинг 7. Скрипт callgraph.stp

```
#!/usr/bin/stap

global traceme;

probe syscall.open {
    /* После PR15044 (SystemTap 2.3) syscall.open использует
       user_string_quoted, так что используем внешнюю переменную */
    filename = user_string_warn($filename);
    if(pid() != target() || filename != "not_exists")
        next;

    traceme = target();

    printf("=> syscall.open [%s]\n", execname());
}

probe syscall.open.return {
    if(pid() == target()) {
        traceme = 0;
    }
}

probe kernel.function("*@fs/*").call ?,
      kernel.function("*@fs/*").return ? {
    if(!traceme || traceme != pid())
        next;

    if(!is_return()) {
        printf("%s -> %s\n", indent( 1), probefunc());
    }
    else {
        /* У функций, возвращающих void переменная $return не определена,
           используем значение 0 */
        ret = 0;
        if(@defined($return))
                ret = $return;

        printf("%s <- %s [%d]\n", indent(-1), probefunc(), ret);
    }
}
```

Вывод будет следующим:

```
# ./callgraph.stp -c "cat not_exists"
cat: not_exists: No such file or directory
=> syscall.open [cat]
     0 : -> do_sys_open
<cut>
 11020 :  -> do_filp_open
 <cut>
```

```
11982 :   -> do_path_lookup
12277 :    -> path_init
12378 :    <- path_init [0]
12451 :    -> path_walk
12581 :     -> __link_path_walk
<cut>
14284 :     <- __link_path_walk [-2]
14339 :     -> path_put
<cut>
14655 :     <- path_put [0]
14732 :    <- path_walk [-2]
14755 :   <- do_path_lookup [-2]
<cut>
15449 :  <- do_filp_open [4294967294]
<cut>
15851 : <- do_sys_open [-2]
```

Таким образом, мы можем точно указать место сбоя — в данном случае это функция __link_path_walk. Этот пример запускался на CentOS 6, на более свежей CentOS 7 проблемной функцией окажется path_openat — функция __link_path_walk была удалена в ядре версии 2.6.32.

Функция indent также печает временную метку вызова функции: слева от двоеточия указывается время в микросекундах с первого его вызова. Функция thread_indent также выводит сведения о текущем потоке исполнения (и имеет по одному буферу смещения на каждый поток).

В DTrace для вывода проб в виде дерева необходимо использовать параметр flowindent:

Листинг 8. Скрипт callgraph.d

```
#!/usr/sbin/dtrace -s

#pragma D option flowindent

syscall::openat*:entry
/pid == $target && copyinstr(arg1) == "not_exists"/
{
        self->traceme = 1;
}

syscall::openat*:return
/self->traceme/
{
        self->traceme = 0;
}

fbt:::entry
/self->traceme && probefunc != "bcmp"/
```

```
{

}

fbt:::return
/self->traceme && probefunc != "bcmp"/
{
        trace(arg1);
}
```

Вывод скрипта будет иметь следующий вид:

```
# 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  <- open64                                              -3041965
  0  -> openat64
  0  <- openat64                                            -3041965
  0  -> copen
<cut>
  0    -> vn_openat
  0      -> lookupnameat
  0        -> lookupnameatcred
<cut>
  0              -> fop_lookup
  0                -> crgetmapped
  0                <- crgetmapped                 3298657895888
  0                -> zfs_lookup
<cut>
  0                <- zfs_lookup                             2
  0              <- fop_lookup                               2
  0              -> vn_rele
  0              <- vn_rele                     3298588956544
  0            <- lookuppnvp                               2
  0          <- lookuppnatcred                             2
  0        <- lookupnameatcred                             2
  0      <- lookupnameat                                   2
  0    <- vn_openat                                        2
<cut>
  0    -> set_errno
  0    <- set_errno                                        2
  0  <- copen                                              2
dtrace: pid 3890 exited with status 1
```

Обычно, в ходе динамического анализа, бывает полезно следить и за состоянием структур данных, что также позволяет пролить свет на некоторые особенности функционирования ядра или приложения. Мы выяснили, что для поиска файла на файловой системе по его имени Linux использует функции __link_path_walk и path_openat a Solaris — функцию lookuppnvp.

Посмотрим как эти функции обращаются с символическими ссылками: создадим файл, символическую ссылку на нее и откроем ее по символической ссылке:

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

Как видно, Linux вызывает функцию __link_path_walk рекурсивно:

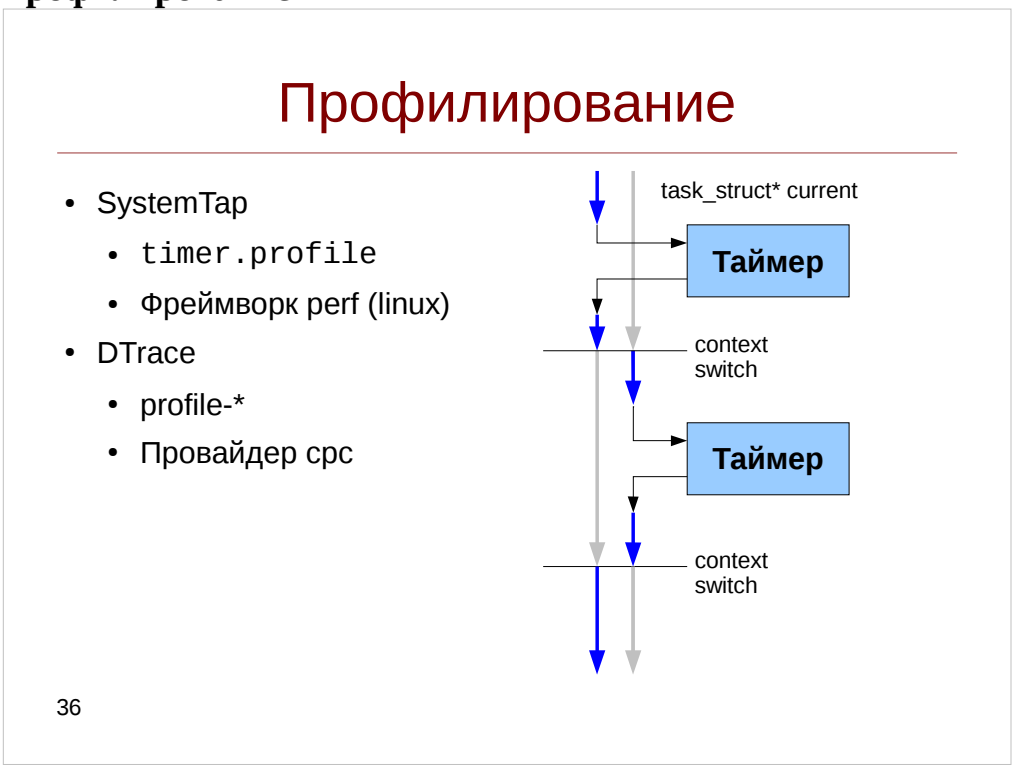
```
# stap -e '
     probe kernel.function(%( kernel_v >= "2.6.32"
                           %? "link_path_walk"
                           %: "__link_path_walk" %) ) {
        println(kernel_string($name));
        print_backtrace(); }' -c 'cat symlink'

symlink
 0xffffffff811ad470 : __link_path_walk+0x0/0x840 [kernel]
 0xffffffff811ae39a : path_walk+0x6a/0xe0 [kernel]
 0xffffffff811ae56b : do_path_lookup+0x5b/0xa0 [kernel]
<cut>
file
 0xffffffff811ad470 : __link_path_walk+0x0/0x840 [kernel]
 0xffffffff811ade31 : do_follow_link+0x181/0x450 [kernel]
 0xffffffff811adc1b : __link_path_walk+0x7ab/0x840 [kernel]
 0xffffffff811ae39a : path_walk+0x6a/0xe0 [kernel]
 0xffffffff811ae56b : do_path_lookup+0x5b/0xa0 [kernel]
<cut>
```

В более свежих ядрах, в которых эта функция была удалена, этого не наблюдается.

В Solaris же аналогичная функция будет вызвана один раз — только для символической ссылки:

```
# dtrace -n '
     lookuppnvp:entry {
          printf("%s\n", stringof(args[0]->pn_path));
          stack(); }' -c 'cat symlink'
  1  19799                  lookuppnvp:entry symlink

              genunix`lookuppnatcred+0x119
              genunix`lookupnameatcred+0x97
              genunix`lookupnameat+0x6b
              genunix`vn_openat+0x147
<cut>
```

Профилирование



Профилирование (Wikipedia) — сбор характеристик работы программы, таких как время выполнения отдельных фрагментов (обычно подпрограмм), число верно предсказанных условных переходов, число кэш промахов и т. д.

Инструмент, используемый для анализа работы, называют профилировщиком. Обычно выполняется в процессе оптимизации программы. В Solaris инструментом профилирования ядра является er_kernel из поставки Solaris Studio, в Linux — OProfile и SysProf, а начиная с версии ядра 2.6.31 — отдельная подсистема ядра — perf.

Обычно под профилированием понимают задачу выявления наиболее часто вызываемой функции или функции, требующих больше всех процессорного времени (на один вызов или суммарно), так как оптимизация таких функций может дать наибольший эффект. В чистом виде такое измерение можно выполнить, установив пробы на входные и возвратные пробы, и засекать временные метки, например так:

```
# dtrace -n '
     fbt:::entry {
          self->start = timestamp;
     }
     fbt:::return
     /self->start/
     {
          @[probefunc] = avg(timestamp - self->start);
     }
     tick-1s {
          printa(@);
          trunc(@); }'
```

Этот подход носит название *профилированием, управляемым событиями*, но он чрезвычайно дорог и значительно снижает производительность системы.

Предупреждение: В SystemTap не рекомендуется использовать конструкцию kernel.function("*"). См. предупреждение на с. 68.

Поэтому обычно для профилирования используется семплирование (*sampling*) — статистический метод, при котором создается выборка значений случайной величины. Так как мы не можем записывать информацию о вызове каждой функции, достаточно в течение длительного времени записывать указатель инструкций с некоторой периодичностью. Предполагая, что наиболее часто вызываемые функции или проводящие на нем длительное время будут чаще попадать в выборку, мы сможем их выявить.

Например, оценим нагрузку процессора по процессам — в обработчике специального профилирующей пробы profile-997hz будем записывать номер процессора и исполняемого приложения, используя считающую агрегацию:

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

Аналогично в SystemTap — с использованеим

```
# stap -e 'global load;
     probe timer.profile {
          load[cpu(), execname()] <<< 1; }
     probe timer.s(20) {
          exit();
     }'
```

В DTrace указатель инструкций (PC) передается в аргументах пробы:

- arg0 — PC в ядре во время срабатывания пробы
- arg1 — PC в пользовательском пространстве

В зависимости от того, где сработала проба, значения arg0 или arg1 могут быть равны 0. Также, для получения значения PC пользовательского процесса можно использовать массив uregs: uregs[REG_PC] или встроенные переменные caller и ucaller.

В SystemTap указатель инструкций можно получить с помощью функции addr(), которая возвращает адрес или в пользовательском пространстве или в ядре в зависимости от того, где сработала проба (может вернуть 0 если текущая проба не поддерживает получение адреса), или uaddr(), который возвращает адрес в пользовательском пространстве.

Замечание: из всех таймерных проб SystemTap указатель инструкций доступен только в специальной таймерной пробе timer.profile. В DTrace для целей

профилирования оптимальнее всего использовать пробу profile-997.

Хотя указатель инструкций и укажет на загрузку процессора, она далеко не всегда отражает полезную работу процессора: значительную часть времени вместо обработки инструкции исполнительными устройствами (например сложение чисел сумматором в АЛУ), процессор будет ожидать данных из памяти или кеша или, в случае ветвления, он может перезагружать конвеер. В таком случае он будет тратить циклы в холостую (в документации Intel такие циклы обычно называются stalled).

Для того, чтобы оценивать эти факторы, современные процессоры имеют счетчики производительности: всякий раз, когда событие, негативно влияющее на эффективность процессора, происходит, он увеличивает значение этого счетчика на 1. В последствии, приложение может прочитать этот счетчик используя инструкцию rdpmc (на x86 системах с процессорами Intel) или сконфигурировать счетчик производительности так, чтобы по достижении определенного значения он бы генерировал прерывание. Это прерывание бы перехватывалось пробой DTrace и SystemTap.

Список доступных событий в Solaris можно посмотреть в выводе утилиты cpustat:

```
# cpustat -h
…
event0:  cpu_clk_unhalted.thread_p inst_retired.any_p
                …
```

Описание событий можно найти в документации на процессор. Для SPARC-процессоров хорошее описание приводится в книге «Solaris Application Programming», однако в книге нет сведений для процессоров T3 и старше. Также, Solaris имеет ряд обобщенных событий, не связанных с конкретной моделью процессора — их имена начинаются с префикса PAPI.

В ядре Linux за эти счетчики отвечает отдельная подсистема: perf. В отличие от Solaris, в perf все события обернуты в обобщенные. Посмотреть из список можно так:

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

Счетчики производительности процессора можно использовать самостоятельно с применением утилит cpustat и cputrack в Solaris или же утилиты perf в Linux.

В DTrace пробы по событиям процессоров используют провайдер cpc. А вот имя имеет ряд дополнительных параметров:

```
Имя_события-Режим[-Маска]-Число
```

Имя события здесь взято из вывода команды cpustat и соответствует документации на процессор (в случае процессоров Intel). *Режим* — это одно из значений kernel, user или all, что позволяет исключать события, возникающие в ядре или

пользовательских приложениях. *Число* — это количество событий после возникновения которых будет вызвана проба. Опциональный параметр *Маска* позволяет фильтровать процессорные устройства (например ядра или контроллеры памяти, которые участвуют в подсчете счетчиков производительности) и выражается шестнадцатиричным числом.

В SystemTap необходимые пробы находятся в tapset'e perf:

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

На самом деле, эти пробы являются алиасами для следующей конструкции:

```
perf.type(x).config(y)[.sample(z)][.process("")][.counter("")]
```

Здесь *type* и *config* — поля структуры ядра perf_event_attr — их значения находятся в файле linux/perf_event.h. *sample* — это количество событий после возникновения которых будет вызвана проба. *process* позволяет фильтровать события по процессам, а *counter* — установить имя счетчика, которое затем можно считать с помощью конструкции @perf.

Исходя из этого, количество промахов мимо кеша последнего уровня (на тестовой системе — это третий уровень) можно оценить следующим образом:

```
# stap -v -e '
     global misses;
     probe perf.hw_cache.ll.read.miss {
          if(!user_mode()) next;
          misses[probefunc()] <<< 1;
     } ' -d <path-to-lib>

# dtrace -n '
     cpc:::PAPI_l3_tcm-user-10000
     /arg1 != 0/ {
          @[usym(arg1)] = count(); }
     END {
          trunc(@, 20);
          printa(@);
     }'
```

Замечание: данные примеры были запущены на процессоре серии Intel Xeon E5-2400.

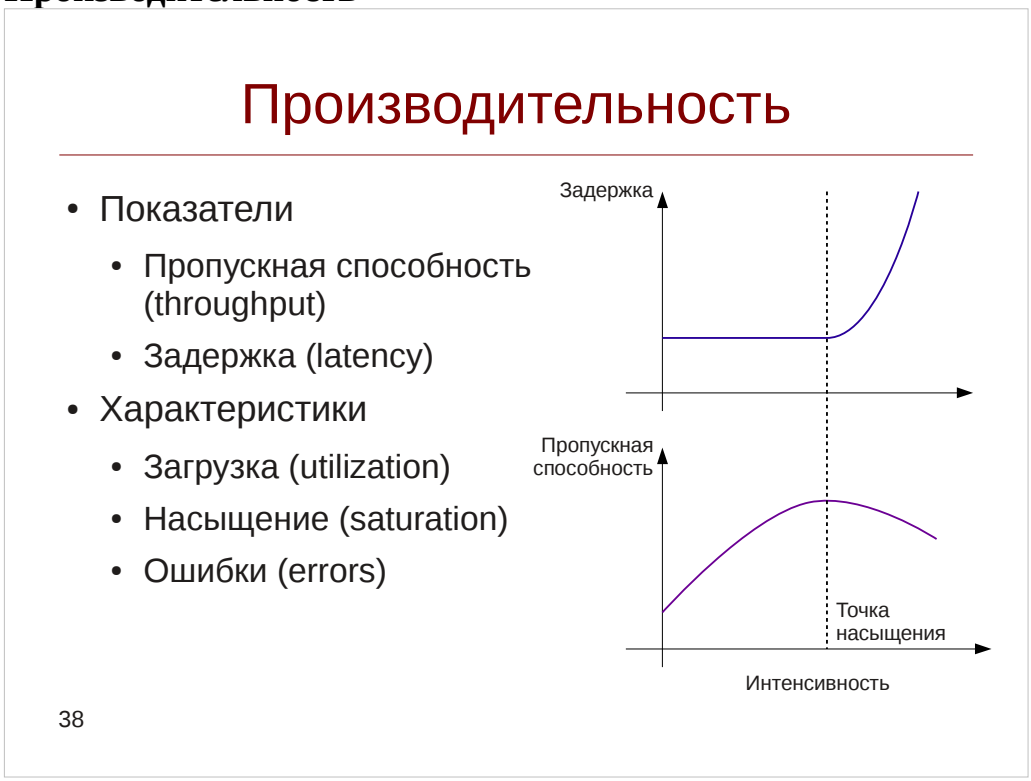
В SystemTap можно также создавать счетчики, индивидуальные для процесса например так:

```
# 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"));
    }'
```

⚠ *Замечание*: на момент написания пособия существовала нерешенная проблема с подобными счетчиками: https://sourceware.org/bugzilla/show_bug.cgi?id=17660 Ее статус на текущий момент неизвестен.

Производительность



Одно из основных направлений по использованию систем динамической трассировки — это конечно же анализ производительности. Для оценки производительности любой системы (не обязательно вычислительной) пользуются обычно двумя параметрами: ее пропускной способностью (throughput) и временем, которое обслуживается каждый запрос, который иногда называется задержкой (latency).

Представим себе газетный киоск, в котором люди покупают разные журналы и газеты. Тогда среднее количество покупателей за час будет являться *интенсивностью поступления заявок (arrival rate)*. Некоторые покупатели будут вставать в очередь, которая будет расти — *длина очереди (queue length)* является показателем *насыщения (saturation)*. Тогда *пропускная способность (throughput)* будет являться количеством покупателей, которые купили газету за час. Однако если покупателей будет слишком много, продавщица газет не будет успевать обслуживать всех — это происходит обычно после преодоления точки насыщения (например очередь в 10 человек), и тогда некоторые покупатели будут покидать киоск — эти ситуации называются *ошибками*. Тогда после преодоления точки насыщения пропускная способность будет наоборот падать из-за неконтролируемо растущих очередей и появления ошибок.

Задержка складывается из *времени обслуживания (service time)*, которое тоже зависит от множества факторов, например: есть ли у покупателя сумма без сдачи, и нужно искать экземпляр газеты под прилавком, и *времени ожидания (waiting time)* в очереди. *Загрузка (utilization)* определяет соотношение между временем,

затраченным продавцом на обслуживание покупателей за некоторый интервал времени, к длине этого интервала. Например, если продавщица в среднем в час 15 минут тратит на покупателей, то загрузка киоска составляет 25%.

Перечисленные определения относятся к теории массового обслуживания (ТМО, Queueing Theory), впервые приложенной к обслуживанию телефонистками абонентов. Она также применима к описанию вычислительных систем: сетевой пакет, операцию блочного ввода-вывода можно рассматривать как *заявку* или *запрос (request)*, а соответствующие драйвер или устройство как *обслуживающий прибор (server)*. В примере с газетным киосками запросами были покупатели, а обслуживающим прибором — продавец.

Для измерения пропускной способности системы можно использовать агрегацию count. По таймеру (например, с 1 секунда), накопленное значение выводится, а агрегация сбрасывается. Например измерим пропускную способность дисковой подсистемы:

```
# stap -e ' global io;
     probe ioblock.end {
          size = 0
          for(vi = 0; vi < $bio->bi_vcnt; ++vi)
               size += $bio->bi_io_vec[vi]->bv_len;
          io[devname] <<< size
     }
     probe timer.s(1) {
          foreach(devname in io) {
               printf("%8s %d\n", devname, @sum(io[devname]));
          }
          delete io;
     }'

# dtrace -n '
     io:::done {
          @[args[1]->dev_statname] = sum(args[0]->b_bcount);
     }
     tick-1s {
          printa(@);
          clear(@);
     } '
```

Чтобы измерить не пропускную способность, а интенсивность поступления запросов в систему, нужно использовать пробы, соответствующие созданию операции ввода-вывода: ioblock.request и io:::start соответственно.

Измерение задержки производится несколько сложнее. Для этого придется ввести ассоциативный массив, ключами к которому будут являтся уникальные значение, например указатель на запрос к подсистеме дискового ввода вывода. Необходимо, чтобы ключ был уникален и не изменялся при исполнении запроса. Если ключом выступает номер потока, то можно использовать Thread-Local переменные DTrace.

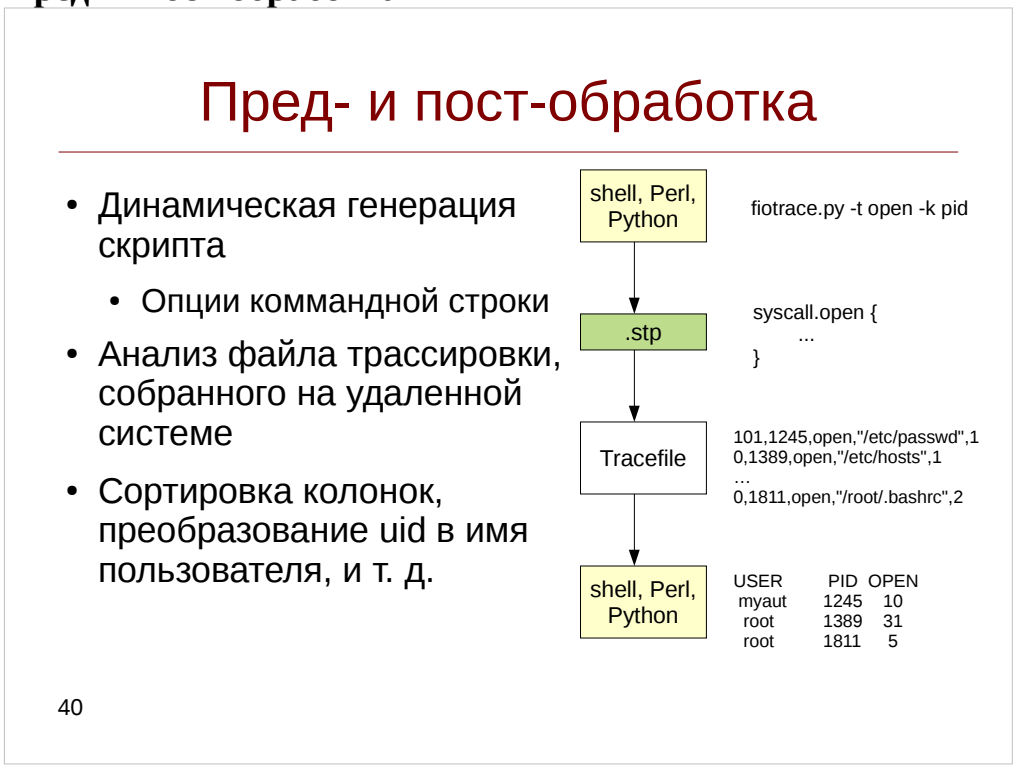
```
# stap -e ' global start, times;
   probe ioblock.request {
       start[$bio] = gettimeofday_us();
   }
   probe ioblock.end {
       if(start[$bio] != 0)
            times[devname] <<< gettimeofday_us() - start[$bio];
       delete start[$bio];
   }
   probe timer.s(1) {
       printf("%12s %8s %s\n", "DEVICE",
               "ASVC_T", ctime(gettimeofday_s()));
       foreach([devname] in times) {
           printf("%12s %8d\n", devname,
                    @avg(times[devname]));
       }
       delete times;
   }'
```

и

```
# 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);
    } '
```

Загрузку можно измерять используя профилирование: таймер с высоким разрешением определяет, занято ли в данный момент устройство или нет, и подсчитывает моменты, когда оно было занято. Длина очереди может быть измерена как разница между пришедшими на исполнение запроса и обработанными запросами, либо же используя ее внутреннее представление в ядре.

Пред- и пост-обработка



Несмотря на гибкость, предоставляемую языками динFoгамической трассировки D и SystemTap, их возможностей хватает не всегда. Например даже если они поддерживают аргументы коммандной строки для передачи параметра (например номера процесса PID или названия дискового устройства, для которого выводится информация), возможности в зависимости от опций коммандной строки использовать разные предикаты уже нет. В таких случаях можно прибегнуть к помощи языков общего назначения: Perl или Python, а также shell-скриптам. Они генерируют DTrace или SystemTap-скрипт на лету, а затем запускают его. Такой подход весьма распространен в наборе скриптов DTraceToolkit.

Дополним наш скрипт, трассирующий системный вызов open() фильтрами по номеру процесса (PID) или номеру пользователя (UID), а также сделаем его универсальным — позволим ему запускаться и в DTrace и в SystemTap.

Листинг 9. Скрипт opentrace.py

```
#!/usr/bin/env python

import sys, os, subprocess, platform
from optparse import OptionParser

# ----------------------------
# opentrace.py - Trace open syscalls via SystemTap or DTrace
# supports filtering per UID or PID

optparser = OptionParser()

optparser.add_option('-S', '--stap', action='store_true',
                     dest='systemtap', help='Run SystemTap')
```

```
optparser.add_option('-D', '--dtrace', action='store_true',
                     dest='dtrace', help='Run DTrace')
optparser.add_option('-p', '--pid', action='store', type='int',
                     dest='pid', default='-1', metavar='PID',
                     help='Trace process with specified PID')
optparser.add_option('-u', '--uid', action='store', type='int',
                     dest='uid', default='-1', metavar='UID',
                     help='Filter traced processes by UID')
optparser.add_option('-c', '--command', action='store', type='string',
                     dest='command', metavar='CMD',
                     help='Run specified command CMD and trace it')

(opts, args) = optparser.parse_args()

if opts.pid >= 0 and opts.command is not None:
    optparser.error('-p and -c are mutually exclusive')
if (opts.pid >= 0 or opts.command is not None) and opts.uid >= 0:
    optparser.error('-p or -c are mutually exclusive with -u')
if opts.systemtap and opts.dtrace:
    optparser.error('-S and -D are mutually exclusive')

if not opts.systemtap and not opts.dtrace:
    # Try to guess based on operating system
    opts.systemtap = sys.platform == 'linux2'
    opts.dtrace = sys.platform == 'sunos5'
if not opts.systemtap and not opts.dtrace:
      optparser.error('DTrace or SystemTap are non-standard for your platform,
please specify -S or -D option')

def run_tracer(entry, ret, cond_proc, cond_user, cond_default,
               env_bin_var, env_bin_path,
               opt_pid, opt_command, args, fmt_probe):
    cmdargs = [os.getenv(env_bin_var, env_bin_path)]
    if opts.pid >= 0:
        cmdargs.extend([opt_pid, str(opts.pid)])
        entry['cond'] = ret['cond'] = cond_proc
    elif opts.command is not None:
        cmdargs.extend([opt_command, opts.command])
        entry['cond'] = ret['cond'] = cond_proc
    elif opts.uid >= 0:
        entry['cond'] = ret['cond'] = cond_user % opts.uid
    else:
        entry['cond'] = ret['cond'] = cond_default
    cmdargs.extend(args)

    proc = subprocess.Popen(cmdargs, stdin=subprocess.PIPE)
    proc.stdin.write(fmt_probe % entry)
    proc.stdin.write(fmt_probe % ret)

    proc.stdin.close()
    proc.wait()

if opts.systemtap:
    entry = {'name': 'syscall.open',
             'dump': '''printf("=> uid: %d pid: %d open: %s %d\\n",
                        uid(), pid(), filename, gettimeofday_ns());'''}
    ret =   {'name': 'syscall.open.return',
             'dump': '''printf("<= uid: %d pid: %d ret: %d %d\\n",
                        uid(), pid(), $return, gettimeofday_ns());'''}

    run_tracer(entry, ret, cond_proc = 'pid() != target()',
```

```
                cond_user = 'uid() != %d', cond_default = '0',
                env_bin_var = 'STAP_PATH',
                env_bin_path = '/usr/bin/stap',
                opt_pid = '-x', opt_command = '-c',
                args = ['-'],
                fmt_probe = ''' probe %(name)s {
                     if(%(cond)s) next;

                     %(dump)s
                 }
                 ''' )
elif opts.dtrace:
    # In Solaris >= 11 open is replaced with openat
    is_sol11 = int(platform.release().split('.')[-1]) >= 11
    sc_name = 'openat*' if is_sol11 else 'open*'
    fn_arg = 'arg1' if is_sol11 else 'arg0'

    entry = {'name': 'syscall::%s:entry' % sc_name,
        'dump': '''printf("=> uid: %%d pid: %%d open: %%s %%lld\\n",
        uid, pid, copyinstr(%s), (long long) timestamp); ''' % fn_arg}
    ret =   {'name': 'syscall::%s:return' % sc_name,
        'dump': '''printf("<= uid: %d pid: %d ret: %d %lld\\n",
                uid, pid, arg1, (long long) timestamp);'''}

    run_tracer(entry, ret, cond_proc = 'pid == $target',
                cond_user = 'uid == %d', cond_default = '1',
                env_bin_var = 'DTRACE_PATH',
                env_bin_path = '/usr/sbin/dtrace',
                opt_pid = '-p', opt_command = '-c',
                args = ['-q', '-s', '/dev/fd/0'],
                fmt_probe = ''' %(name)s
                 /%(cond)s/
                 {
                     %(dump)s
                 }
                 ''' )
```

В зависимости от переданных опций коммандной строки он генерирует условие предиката. Вызвать его можно например так:

```
# python opentrace.py -D -u 100
```

Пост-обработка заключается наоборот в анализе уже собранного файла трссировки. Это позволяет отложить обработку трассировки, если например неизвестны измеряемые показатели и их детализация — в этом случае в процессе трассировки собираются все возможные данные в файл трассировки, а затем для его анализа запускается скрипт на языке общего назначения: Perl или Python. Кроме того, пост-обработка позволяет применять более комплексные методы анализа (прибегая, например к пакетам статистического анализа наподобие R) и обходить некоторые ограничения, например переупорядочивать записи в файле трассировки по времени или запросам, обходя попроцессорную обработку буферов потребителем.

Создадим скрипт, осуществляющий склейку проб входа и выхода из системного вызова open(). Для этого введем словарь (аналог ассоциативных массивов в Python) states где будем сохранять временную метку входа и имя

открытого файла. Также мы будем преобразовывать UID пользователя в его логин. В пробе выхода из системного вызова вычислим разность, чтобы измерить время обслуживание и выведем его в человеко-понятном виде:

Листинг 10. Скрипт openproc.py

```
#!/usr/bin/env python

import re
import sys

# -----------------------------
# openproc.py - Collect data from opentrace.py and merge :entry and :return
probes

# Open trace file or use stdin
try:
    inf = file(sys.argv[1], 'r')
except OSError as ose:
    print ose
    print '''openproc.py [filename]'''
    sys.exit(1)
except IndexError:
    inf = sys.stdin

# Convert time to human time
def human_time(ns):
    ns = float(ns)
    for unit in ['ns', 'us', 'ms']:
        if abs(ns) < 1000.0:
            break
        ns /= 1000.0
    else:
        unit = 's'
    return "%.2f %s" % (ns, unit)

# Parse /etc/passwd and create UID-to-USERNAME map
uid_name = lambda user: (int(user[2]), user[0])
users = dict([uid_name(user.split(':'))
              for user in file('/etc/passwd')])

# Per-PID state - tuples (start time, file name)
state = {}

# Regular expressions for parsing tracer output
re_entry = re.compile("=> uid: (\d+) pid: (\d+) open: (.*?) (\d+)")
re_return = re.compile("<= uid: (\d+) pid: (\d+) ret: (-?\d+) (\d+)")

for line in inf:
    if line.startswith('=>'):
        # :entry probe, extract start time and filename
        m = re_entry.match(line)
        _, pid, fname, tm = m.groups()

        state[int(pid)] = (int(tm), fname)
    elif line.startswith('<='):
        # :return probe, get return value, timestamp and print information
        m = re_return.match(line)
```

```
    uid, pid, ret, tm = map(int, m.groups())

    if pid not in state:
        continue

    status = 'FD %d' % ret if ret >= 0 else 'ERROR %d' % ret

    print 'OPEN %s %d %s => %s [%s]' % (users.get(uid, str(uid)),
                                        pid, state[pid][1], status,
                                        human_time(tm - state[pid][0]))
    del state[pid]
```

Перенаправив вывод скрипта opentrace.py на вход openproc.py можно получить подобный вывод:

```
# 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]
…
```

Замечание: В действительности возможности этого скрипта можно реализовать средствами DTrace или SystemTap. Так, с помощью деструктивного действия system() можно вызвать оболочку, чтобы получить имя пользователя, а человеко-понятный вывод временных интервалов можно сделать с помощью функции в SystemTap или предикатов в DTrace.

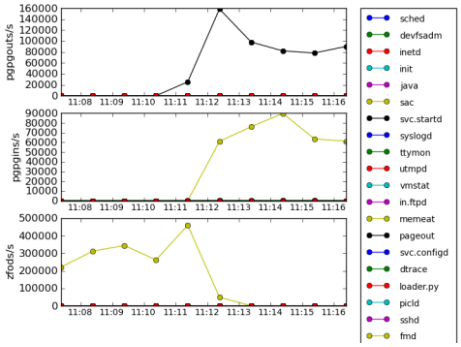
Однако следует помнить также, что это вызовет дополнительные накладные расходы по производительности, и дополнительные риски безопасности: в частности действие system() порождает новые системные вызовы open(), что вводит систему в бесконечный цикл.

Пример сочетания пред- и пост-обработки вы можете найти например в скрипте tmptop, описанном здесь: http://bit.ly/1cWnQCJ

Визуализация

Визуализация

- Временные диаграммы
  - Линейные
  - Горизонтальные гистограммы
- Гистограммы распределений
- Тепловые графики



41

Чтение файлов трассировки — утомительный процесс, поэтому распространенный сценарий пост-обработки — это визуализация. Визуализацию можно выполнять например так:

- Используя пакет GNU Plot, как показано здесь: https://sourceware.org/systemtap/wiki/WSUtilGraphWithGnuplot
  При этом в ходе трассировки генерируются его команды.
- Используя плагин DTrace Chime для NetBeans: http://wiki.netbeans.org/NetBeans_DTrace_GUI_Plugin_1_0
- Используя среду SystemTap GUI: http://stapgui.sourceforge.net/
- Самостоятельно обрабатывая трассировку используя язык общего назначения и соответствующие библиотеки, например язык Python и matplotlib

Какие же типы диаграмм использовать в процессе визуализации? Самый простой — это диаграмма, осью абсцисс которого является время, что позволяет отслеживать поведение системы во времени. В приведенном примере специальный бенчмарк memeat потреблял всю оперативную память в системе, что можно видеть по росту событий zfod на нижнем графике в его левой части. Это приводит к тому, что Solaris вынужден обращаться к дисковому свопу, что видно в правой части верхних графиков: процесс svc.startd вытесняется, а memeat подгружает свои данные из дискового свопа.

Запустим теперь такой эксперимент: выполним трассировку диспетчера процессов:

```
# dtrace -n '
     sched:::on-cpu {
          printf("%d %d %s\n", timestamp,
          cpu, (curthread ==
```

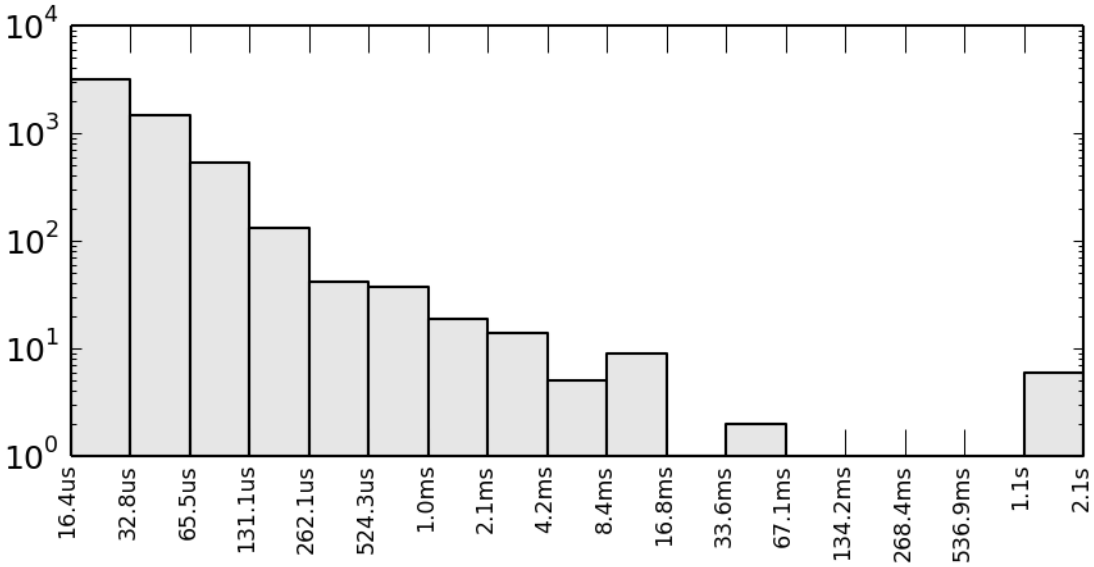
```
                curthread->t_cpu->cpu_idle_thread)
                    ? "idle"
                    : execname ); }'
```

Иногда в одной из оболочек мы будем исполнять следующее:

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

Посмотрим, как визуализация нам поможет идентифицировать процессорно-интенсивную задачу.

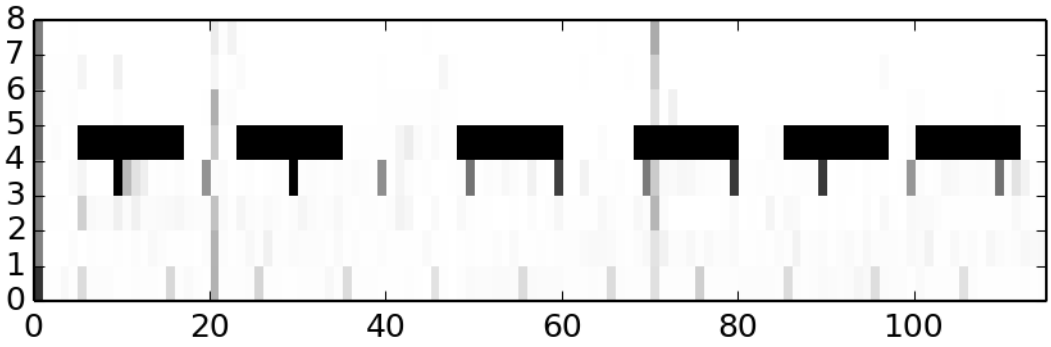
Очень часто при анализе производительности прибегают к средним значениям, что позволяет сделать хорошую мину при плохой игре. Представим себе завод, на котором работает 100 человек, а средняя зарплата составляет 30 тысяч рублей. Эта цифра может означать как то что на заводе уборщицы получают по 15 тысяч, рабочие - 30, а директор 100 тысяч, так и то, что зарплата директора составляет 2 миллиона, а остальных работников — 7- тысяч рублей. В исследовании производительности точно также: среднее время обслуживания запроса в 10 мс вовсе не означает отсутствия «тяжелых» запросов:



Обе оси на этом графике — логарифмические, по оси x показан двоичный логарифм от времени, которое задача занимала на процессоре. Хотя среднее значение составлет около 2 мс, как видим, некоторые некоторые интервалы превышают по длительности 1 секунду, что свидетельствует о наличии в системе процессорно-интенсивной задачи.

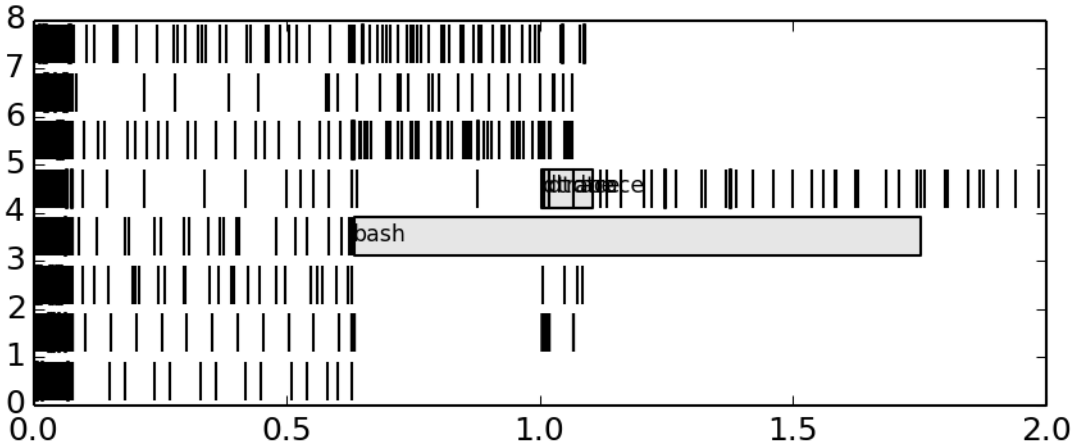
Замечание: агрегации quantize и lquantize позволяют построить распределения непосредственно в текстовом терминале — см. раздел «Агрегации» на с. 51

Когда осей не хватает, чтобы избежать построения трехмерной диаграммы, можно воспользоваться т. н. heatmap — тепловым графиком. В таком графике значение величины показано интенсивностью цвета прямоугольника:



На приведенном графике видно, что иногда на 4м процессоре появляются интенсивные задачи.

Когда данные имеют дискретный характер: в каждый момент времени система находится в одном из конечного числа состояний, то более предпочтительной является горизонтальная гистограмма. Примеры таких состояний: программа ожидает приход данных по сети или выполняет пользовательский запрос; процессор выполняет один из потоков или ожидает — он и показан на рисунке.



Итак мы выявили «нарушителя» - это процесс bash. Можно было бы также построить диаграмму распределения так, чтобы для каждому прямоугольнику соответствовал определенный процесс, тогда мы бы сразу идентифицировали процессорно-интенсивную задачу.

Замечание: графики в данном разделе были построены с использованием библиотеки matplotlib для языка Python.

Модуль 4. Динамическая трассировка операционных систем

Управление процессами

Все функционирующее на компьютере программное обеспечение, иногда включая собственно операционную систему, организовано в виде набора последовательных процессов, или, для краткости, просто процессов. Процессом является выполняемая программа, включая текущие значения счетчика команд, регистров и переменных (*Эндрю Таненбаум «Современные Операционные Системы»*).

В Linux каждому процессу ставится в соответствие структура task_struct, определенная в файле include/linux/sched.h. Для извлечения данных из структуры task_struct в SystemTap предусмотрен ряд вспомогательных функций:

- task_current() — возвращает указатель на task_struct для текущего процесса, выполняемого на процессоре
- task_state(task) — возвращает флаг состояния для указанного процесса
- task_parent(task) — возвращает указатель на родительский процесс
- task_execname(task) — возвращает строку, содержащую имя исполняемого файла процесса
- task_pid(task), task_tid(task) — возвращает pid соответствующего процесса. Хотя семантически tid является номером потока, Linux не поддерживает потоки в режиме ядра и поэтому task_tid также будет возвращать pid соответствующей task_struct.
- task_cpu(task) — номер процессора, на котором запланировано выполнение задачи

Также представляют интерес следующие поля структуры task_struct:

- .comm (char[]) — содержит имя команды, соответствующей задаче.

Фактически, представляет из себя часть строки, передаваемой в виде аргумента execve после последнего слеша.

- .mm (mm_struct*) — указатель на информацию о виртуальной памяти процесса
  - .mm.exe_file (file*) — указатель на исполнимый файл
  - .mm.arg_start и .mm.arg_end — указатели на расположение аргументов в пользовательском пространстве процесса. Можно получить используя например, user_string: см. реализацию cmdline_args в context.stp
- .fs (fs_struct*) — содержит объекты path, соответствуюшей рабочей директории процесса в поле pwd и корневой директории в поле root
- .start_time и .real_start_time (типа timespec, начиная с ядра 3.17 — целочисленное u64) — время запуска задачи относительно времени загрузки. .start_time использует монотонное время, а .real_start_time — реальное, подробнее эти различия описаны в man-странице clock_gettime(3)
- .files (files_struct*) — указатель на структуру, содержащую таблицу файловых дескрипторов
- .utime и .stime (cputime_t) — время выполнения процесса в режиме ядра (системы) и пользовательском пространстве. Подробнее — см. tapset task_time.stp

---

Листинг 11. Скрипт dumptask.stp

```
/**
 * taskdump.stp
 *
 * Один раз в секунду печатает информацию по текущему процессу
 * Включает функции извлечения из task_struct
 */

/**
 * В старых ядрах структуры dentry и vfsmnt существовали раздельно.
 * В новых ядрах они были объединены в единую структуру path.
 *
 * SystemTap не кеширует полный путь до директории, поэтому
 * нам бы пришлось воспользоваться функцией task_dentry_path(), чтобы
 * собрать его например так:
 *      dentry = @cast(file, "file")->f_path->dentry;
 *      vfsmnt = @cast(file, "file")->f_path->mnt;
 *      return task_dentry_path(task, dentry, vfsmnt);
 *
 * Однако в SystemTap присутствует баг 16991, закрытый в версии 2.6
 * Поэтому мы ограничимся выводом имени файла внутри директории
 */
function file_path:string(task:long, file:long) {
    if(@defined(@cast(file, "file")->f_vfsmnt))
        return d_name(@cast(file, "file")->f_dentry);
    return d_name(@cast(file, "file")->f_path->dentry);
}
function task_root_path:string(task:long, fs_ptr:long) {
    if(@defined(@cast(fs_ptr, "fs_struct")->rootmnt))
        return d_name(@cast(fs_ptr, "fs_struct")->root);
    return d_name(@cast(fs_ptr, "fs_struct")->root->dentry);
}
```

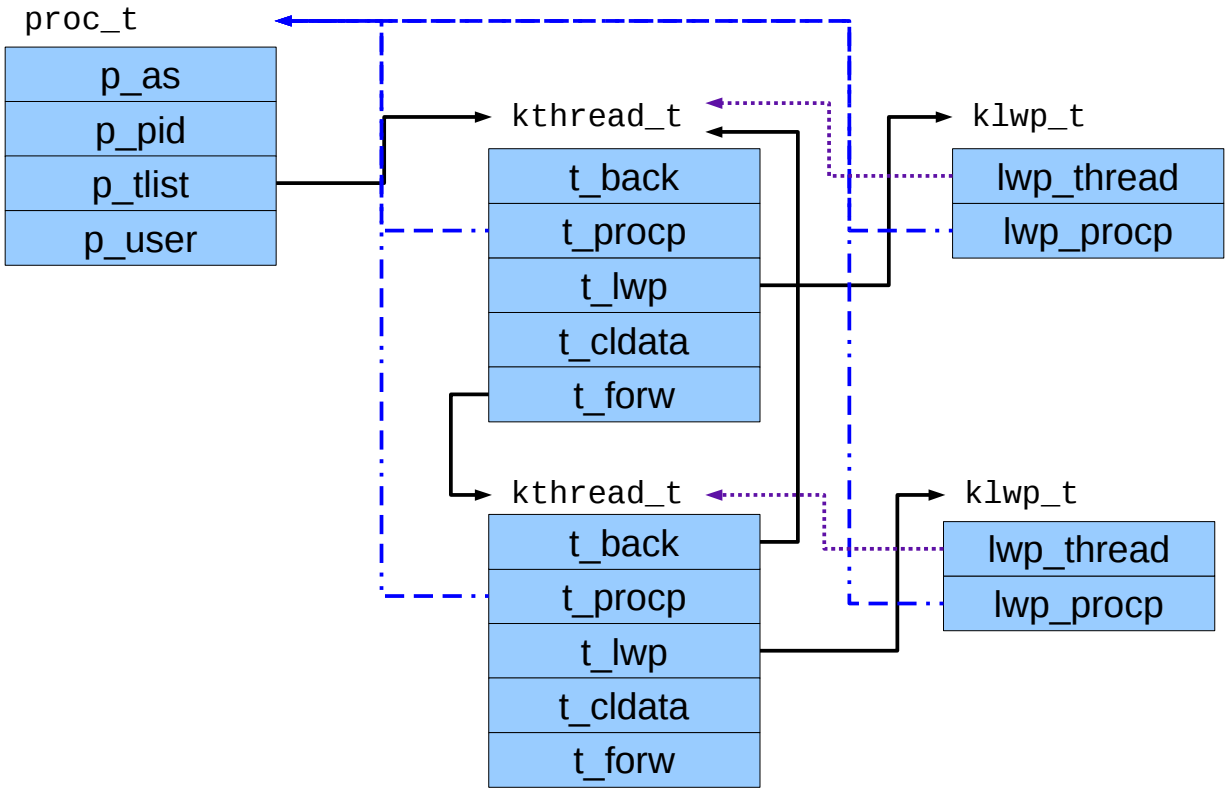
```
function task_pwd_path:string(task:long, fs_ptr:long) {
    if(@defined(@cast(fs_ptr, "fs_struct")->pwdmnt))
        return d_name(@cast(fs_ptr, "fs_struct")->pwd);
    return d_name(@cast(fs_ptr, "fs_struct")->pwd->dentry);
}

/**
 * Печатает имя исполняемого файла по mm->exe_file */
function task_exefile(task:long, mm_ptr:long) {
     if(mm_ptr) {
           printf("\texe: %s\n",
               file_path(task, @cast(mm_ptr, "mm_struct")->exe_file));
     }
}
/**
 * Печатает корневую и текущую директории процесса
 */
function task_paths(task:long, fs_ptr:long) {
     if(fs_ptr) {
           printf("\troot: %s\n", task_root_path(task, fs_ptr));
           printf("\tcwd: %s\n", task_pwd_path(task, fs_ptr));
     }
}

/**
 * Печатает аргументы. Аргументы отображены в виртуальное адресное
 * пространство процесса и представляют собой область памяти (mm->arg_start;
 * mm_arg_end), содержащую последовательно стоящие строки, например:
 *
 * +-----+----+-------------+----+
 * | cat | \0 | /etc/passwd | \0 |
 * +-----+----+-------------+----+
 * ^                           ^
 * arg_start                   arg_end
 *
 * ЗАМЕЧАНИЕ: функции user_string* читают из текущего пространства процесса.
 * Для получения аргументов из стороннего процесса используйте Embedded C
 * и функцию ядра proc_pid_cmdline
 */
function task_args(mm_ptr:long) {
     if(mm_ptr) {
           arg_start = @cast(mm_ptr, "mm_struct")->arg_start;
           arg_end = @cast(mm_ptr, "mm_struct")->arg_end;
           if (arg_start != 0 && arg_end != 0)
           {
                 len = arg_end - arg_start;
                 nr = 0;

                 /*Выбираем первый аргумент*/
                 arg = user_string2(arg_start, "");
                 while (len > 0)
                 {
                       printf("\targ%d: %s\n", nr, arg);
                       arg_len = strlen(arg);
                       arg_start += arg_len + 1;
                       len -= arg_len + 1;
                       nr++;

                       arg = user_string2(arg_start, "");
                 }
           }
```

```
	}
}

/**
 * Возвращает файловый дескриптор из таблицы по номеру fd
 * См. также: реализацию pfiles.stp
 */
function task_fd_filp:long(files:long, fd:long) {
	return @cast(files, "files_struct")->fdt->fd[fd];
}

function task_fds(task:long) {
	task_files = @cast(task, "task_struct", "kernel<linux/sched.h>")->files;

	if(task_files) {
		max_fds = task_max_file_handles(task);

		for (fd = 0; fd < max_fds; fd++) {
			filp = task_fd_filp(task_files, fd);
			if(filp) {
				printf("\tfile%d: %s\n", fd, file_path(task, filp));
			}
		}
	}
}

/**
 * Печатает разницу время запуска системы
 * start time - монотонное (monotonic)
 * real start time - с момента запуска системы (bootbased)
 *
 * Замечание: эта функция работает с timespec, однако начиная с 3.17 они
 * были заменены на u64
 */
function task_start_time_x(task:long) {
	if(@defined(@cast(task, "task_struct", "kernel<linux/sched.h>")
			->start_time)) {
		start_time_sec = @cast(task, "task_struct", "kernel<linux/sched.h>")
			->start_time->tv_sec;
		real_time_sec = @cast(task, "task_struct", "kernel<linux/sched.h>")
			->real_time->tv_sec;
	  printf("\tstart time: %ds\t real start time: %ds\n", start_time_sec,
real_time_sec);
	}
	else {
		real_time_sec = @cast(task, "task_struct", "kernel<linux/sched.h>")
			->real_start_time->tv_sec;
	  printf("\treal start time: %ds\n", real_time_sec);
	}

}

/**
 * Печатает статистику выполнения задачи на процессоре */
function task_time_stats(task:long) {
	user = @cast(task, "task_struct", "kernel<linux/sched.h>")->utime;
	kernel = @cast(task, "task_struct", "kernel<linux/sched.h>")->stime;
	printf("\tuser: %s\t kernel: %s\n", cputime_to_string(user),
cputime_to_string(kernel));
}
```

```
function dump_task(task:long) {
      task_mm = @cast(task, "task_struct", "kernel<linux/sched.h>")->mm;
      task_fs = @cast(task, "task_struct", "kernel<linux/sched.h>")->fs;

      printf("Task %p is %d@%d %s\n", task, task_pid(task), task_cpu(task),
task_execname(task));

      task_exefile(task, task_mm);
      task_paths(task, task_fs);
      task_args(task_mm);
      task_fds(task);
      task_start_time_x(task);
      task_time_stats(task);
}

probe timer.s(1) {
    dump_task(task_current());
}
```

Замечание: В данном случае вызовы типа task_pid(task) полностью аналогичны вызовам контекстным функций pid() и т. п.

Solaris в отличие от Linux поддерживает потоки на уровне ядра (т. н. легковесные процессы, Light-Weight Process или LWP), из-за чего каждому процессу соответствуют три структуры:



Получить информацию о процессе можно с помощью встроенной перемнной curthread типа kthread_t. DTrace также предоставляет удобные трансляторы curpsinfo и curlwpsinfo, дающие информацию по процессу и LWP соответственно.

Чтобы обратиться к родительскому процессу можно использовать поле p_parent структуры proc_t, например: curthread->t_procp->p_parent, основная же полезная информация хранится в полях структуры user (поле p_user).

Листинг 12. Скрипт dumptask.d

```
#!/usr/sbin/dtrace -qCs

/**
 * dumptask.d
 *
 * Один раз в секунду печатает информацию по текущему процессу
 * Включает макросы извлечения из kthread_t и сопутствующих структур
 * Использует стандартные трансляторы psinfo_t* и lwpsinfo_t*s
 *
 * Оттестировано на Solaris 11.2
 */

int argnum;
void* argvec;
string pargs[int];

int fdnum;
uf_entry_t* fdlist;

#define PSINFO(thread) xlate<psinfo_t *>(thread->t_procp)
#define LWPSINFO(thread) xlate<lwpsinfo_t *>(thread)

#define PUSER(thread) thread->t_procp->p_user

/**
 * Вытаскиваем указатель из массива указателей в зависимости
 * от модели процесса (32- или 64-бита)
 */
#define DATAMODEL_ILP32 0x00100000
#define GETPTR(proc, array, idx)                                    \
    ((uintptr_t) ((proc->p_model == DATAMODEL_ILP32)                \
    ?  ((uint32_t*) array)[idx] : ((uint64_t*) array)[idx]))
#define GETPTRSIZE(proc)                                            \
    ((proc->p_model == DATAMODEL_ILP32)? 4 : 8)

#define FILE(list, num)     list[num].uf_file
#define CLOCK_TO_MS(clk)    (clk) * (`nsec_per_tick / 1000000)

/* Вытаскиваем путь до vnode */
#define VPATH(vn)                                   \
    ((vn) == NULL || (vn)->v_path == NULL)          \
        ? "unknown" : stringof((vn)->v_path)

/* Корневая директория процесса - для зон может не совпадать с / */
#define DUMP_TASK_ROOT(thread)                      \
    printf("\troot: %s\n",                          \
        PUSER(thread).u_rdir == NULL                \
        ? "/"                                       \
        : VPATH(PUSER(thread).u_rdir));

/* Текущая рабочая директория процесса */
#define DUMP_TASK_CWD(thread)                       \
```

```
    printf("\tcwd: %s\n",                          \
        VPATH(PUSER(thread).u_cdir));

/* Исполнимый файл процесса */
#define DUMP_TASK_EXEFILE(thread)                 \
    printf("\texe: %s\n",                         \
        VPATH(thread->t_procp->p_exec));

/* Копируем до 9 аргументов процесса. Само количество получаем
   через транслятор psinfo_t, сами же аргументы и указатели (необходимой
   в процессе длины) размещаются в стеке процесса. Копируем указатели
   в argvec и сами аргументы в массив pargs.
   См. также функцию ядра exec_args() */
#define COPYARG(t, n)                                      \
    pargs[n] = (n < argnum && argvec != 0)                 \
        ? copyinstr(GETPTR(t->t_procp, argvec, n)) : "???"
#define DUMP_TASK_ARGS_START(thread)                       \
    printf("\tpsargs: %s\n", PSINFO(thread)->pr_psargs);   \
    argnum = PSINFO(thread)->pr_argc;                      \
    argvec = (PSINFO(thread)->pr_argv != 0) ?              \
              copyin(PSINFO(thread)->pr_argv,              \
                     argnum * GETPTRSIZE(thread->t_procp)) : 0;\
    COPYARG(thread, 0); COPYARG(thread, 1); COPYARG(thread, 2); \
    COPYARG(thread, 3); COPYARG(thread, 4); COPYARG(thread, 5); \
    COPYARG(thread, 6); COPYARG(thread, 7); COPYARG(thread, 8);

/* Время запуска процесса */
#define DUMP_TASK_START_TIME(thread)                       \
    printf("\tstart time: %ums\n",                         \
        (unsigned long) thread->t_procp->p_mstart / 1000000);

/* Процессорное время, затраченное процессом. Вообще говоря,
   оно измеряется через LWP microstate, а поля p_utime и p_stime
   устанавливаются при выходе из процесса  */
#define DUMP_TASK_TIME_STATS(thread)                       \
    printf("\tuser: %ldms\t kernel: %ldms\n",              \
        CLOCK_TO_MS(thread->t_procp->p_utime),             \
        CLOCK_TO_MS(thread->t_procp->p_stime));

#define DUMP_TASK_FDS_START(thread)                        \
    fdlist = PUSER(thread).u_finfo.fi_list;                \
    fdcnt = 0;                                             \
    fdnum = PUSER(thread).u_finfo.fi_nfiles;

#define DUMP_TASK(thread)                                  \
    printf("Task %p is %d/%d@%d %s\n", thread,             \
            PSINFO(thread)->pr_pid,                        \
            LWPSINFO(thread)->pr_lwpid,                    \
            LWPSINFO(thread)->pr_onpro,                    \
            PUSER(thread).u_comm);                         \
    DUMP_TASK_EXEFILE(thread)                              \
    DUMP_TASK_ROOT(thread)                                 \
    DUMP_TASK_CWD(thread)                                  \
    DUMP_TASK_ARGS_START(thread)                           \
    DUMP_TASK_FDS_START(thread)                            \
    DUMP_TASK_START_TIME(thread)                           \
    DUMP_TASK_TIME_STATS(thread)

#define _DUMP_ARG_PROBE(probe, argi)                       \
probe /argi < argnum/ {                                    \
    printf("\targ%d: %s\n", argi, pargs[argi]); }
```

```
#define DUMP_ARG_PROBE(probe)                                   \
    _DUMP_ARG_PROBE(probe, 0)   _DUMP_ARG_PROBE(probe, 1)    \
    _DUMP_ARG_PROBE(probe, 2)   _DUMP_ARG_PROBE(probe, 3)    \
    _DUMP_ARG_PROBE(probe, 4)   _DUMP_ARG_PROBE(probe, 5)    \
    _DUMP_ARG_PROBE(probe, 6)   _DUMP_ARG_PROBE(probe, 7)    \
    _DUMP_ARG_PROBE(probe, 8)

#define _DUMP_FILE_PROBE(probe, fd)                             \
probe /fd < fdnum && FILE(fdlist, fd)/ {                        \
    printf("\tfile%d: %s\n", fd,                                \
                VPATH(FILE(fdlist, fd)->f_vnode)); }
#define DUMP_FILE_PROBE(probe)                                  \
    _DUMP_FILE_PROBE(probe, 0)  _DUMP_FILE_PROBE(probe, 1)   \
    _DUMP_FILE_PROBE(probe, 2)  _DUMP_FILE_PROBE(probe, 3)   \
    _DUMP_FILE_PROBE(probe, 4)  _DUMP_FILE_PROBE(probe, 5)   \
    _DUMP_FILE_PROBE(probe, 6)  _DUMP_FILE_PROBE(probe, 7)

BEGIN {
    proc = 0;
    argnum = 0;
    fdnum = 0;
}

tick-1s {
    DUMP_TASK(curthread);
}

DUMP_ARG_PROBE(tick-1s)
DUMP_FILE_PROBE(tick-1s)
```

Определить, является ли процесс потомком заданного можно с помощью функции progenyof (DTrace). В SystemTap такой функции не предусмотрено, однако можно использовать комбинацию task_parent/task_current/task_pid:

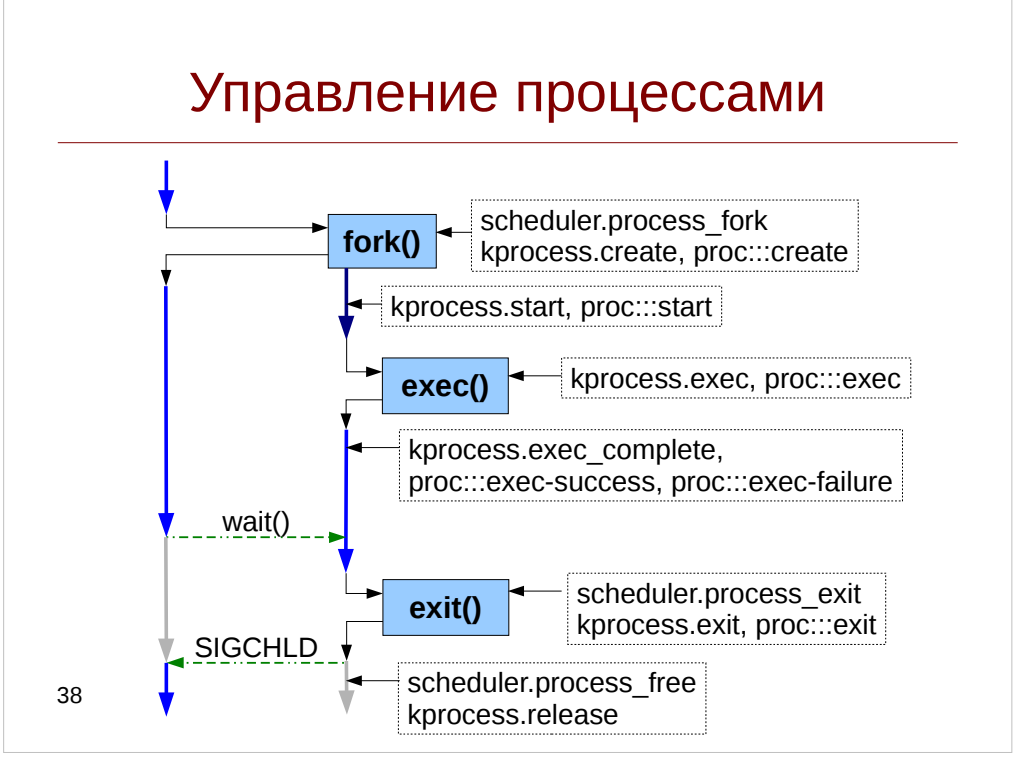
```
/*Является ли текущая задача дочерней для задачи pid?*/
function progenyof(pid:long) {
     parent = task_parent(task_current());
     task = pid2task(pid);

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

          parent = task_parent(parent);
     }
}

probe syscall.open {
     if(progenyof(target()))
               printdln(" ", pid(), execname(), filename);
}
```

Привяжем выполнение скрипта к интерпретатору (pid 2953)

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

Откроем новый сеанс интерпретатора. Так как cat является потомком интепретатора с pid 2953 (хотя и не непосредственным), мы видим его pid в выводе SystemTap.

```
root@lktest:~# bash
root@lktest:~# 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@lktest:~# cat /etc/passwd
<cut>
```



Поговорим теперь о времени жизни процесса. Начинается он с порождения (spawning) нового процесса. В отличие от операционной системы Windows, в Unix-системах оно происходит в два этапа:

- Процесс-родитель вызывает системный вызов fork(). При этом ядро создает точную копию родительского процесса (включая адресное пространство и ресурсы, такие как открытые файлы), и присваивает ему новый PID. При этом получаются два процесса с общим адресным пространством, но различным контекстом.

Замечание: Строго говоря, в Linux существует три системных вызова: fork(), vfork() и clone() - все они являются обертками над функцией do_fork() и отличаются передаваемыми ей аргументами.

- В контексте порожденного процесса вызывается системный вызов execve(). Он замещает адресное пространство, загружая бинарный образ приложения.

Когда процесс завершается, то он вызывает системный вызов exit(), также процесс может принудительно завершиться из-за аппаратного сбоя или сбоя в ядре (machine fault) или одного из сигналов завершения.

Также нередка ситуация, когда родительский процесс блокируется, ожидая завершения дочернего (например так поступает оболочка bash когда процесс вызван в foreground-режиме). Для этого предусмотрены системные вызовы wait* (на самом деле они должны вызываться всегда, иначе процесс останется в системе и станет зомби). Также ядро уведомляет родительский процесс с помощью сигнала SIGCHLD.

Отслеживать поведение процессов в SystemTap можно с помощью tapset'ов kprocess и частично scheduler, в DTrace данный функционал обеспечен провайдером proc. Также для отслеживания системных вызовов можно использовать пробы системных вызовов.

Перечислим ключевые пробы:

Действие	*DTrace*	*SystemTap*
Создание процесса	proc:::create	kprocess.create scheduler.process_fork
Выполнение execve()	proc:::exec — запуск execve, proc:::exec-success - в случае успешного запуска, а proc:::exec-failure — в случае сбоя (args[0] содержит код ошибки errno)	kprocess.exec - запуск execve, kprocess.exec_complete — возвращение из exec. success содержит true, если выполнено успешно, а errno — код ошибки
Запуск процесса	proc:::start — вызывается в контексте нового процесса	kprocess.start — вызывается в контексте нового процесса, scheduler.wakeup_new — процесс впервые ставится на исполнение планировщиком
Завершение процесса	proc:::exit — по exit() proc:::fault — вынужденное завершение	kprocess.exit, scheduler.process_exit
Освобождение процесса	-	kprocess.release, scheduler.process_free
Управление потоками LWP	proc:::lwp-create proc:::lwp-start proc:::lwp-exit	LWP-потоки не поддерживаются в Linux

Примеры скриптов, демонстрирующих поведение процесса представлены в листингах 38.1 и 38.2:

Листинг 13. Скрипт proc.stp

```
#!/usr/bin/stap

probe scheduler.process*, scheduler.wakeup_new, syscall.fork,
```

```
    syscall.exec*, syscall.exit, syscall.wait*, kprocess.* {
      printf("%6d[%8s]/%6d[%8s] %s\n",
            pid(), execname(), ppid(), pid2execname(ppid()), pn());
}

probe scheduler.process_fork {
      printf("\tPID: %d -> %d\n", parent_pid, child_pid);
}

probe kprocess.exec {
    printf("\tfilename: %s\n", filename);
}

probe kprocess.exit {
    printf("\treturn code: %d\n", code);
}
```

Вызовем например из оболочки команду uname, тогда вывод будет выглядеть следующим образом:

```
  2578[    bash]/  2576[    sshd] syscall.fork
  2578[    bash]/  2576[    sshd] kprocess.create
  2578[    bash]/  2576[    sshd] scheduler.process_fork
        PID: 2578 -> 3342
  2578[    bash]/  2576[    sshd] scheduler.wakeup_new
  3342[    bash]/  2578[    bash] kprocess.start
  2578[    bash]/  2576[    sshd] syscall.wait4
  2578[    bash]/  2576[    sshd] scheduler.process_wait
        filename: /bin/uname
  3342[    bash]/  2578[    bash] kprocess.exec
  3342[    bash]/  2578[    bash] syscall.execve
  3342[   uname]/  2578[    bash] kprocess.exec_complete
        return code: 0
  3342[   uname]/  2578[    bash] kprocess.exit
  3342[   uname]/  2578[    bash] syscall.exit
  3342[   uname]/  2578[    bash] scheduler.process_exit
  2578[    bash]/  2576[    sshd] kprocess.release
```

Листинг 14. Скрипт proc.d

```
#!/usr/sbin/dtrace -qCs

#define PARENT_EXECNAME(thread) \
    (thread->t_procp->p_parent != NULL) \
        ? stringof(thread->t_procp->p_parent->p_user.u_comm)    \
        : "???"

proc:::, syscall::fork*:entry, syscall::exec*:entry,
    syscall::wait*:entry  {
    printf("%6d[%8s]/%6d[%8s] %s::%s:%s\n",
        pid, execname, ppid, PARENT_EXECNAME(curthread),
           probeprov, probefunc, probename);
}

proc:::create {
    printf("\tPID: %d -> %d\n", curpsinfo->pr_pid, args[0]->pr_pid);
```

```
}


proc:::exec {
    printf("\tfilename: %s\n", args[0]);
}

proc:::exit {
    printf("\treturn code: %d\n", args[0]);
}
```

Вывод выглядит следующим образом:

```
 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_initial: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
```

Упражнение 3

Задание 1

Измените скриты dumptask.d и dumptask.stp так, чтобы они выводили информацию не по таймеру а при после загрузки бинарного кода и при выходе из процесса.

Напишите небольшую программу:

Листинг 15. Исходный код lab3.c

```
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char* argv[]) {
    while(--argc > 0) {
        memset(argv[argc], 'X', strlen(argv[argc]));
    }

    open("/etc/passwd", O_RDONLY);
    return 0;
}
```

И скомпилируйте ее:

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

Запустите измененные скрипты и запустите полученный бинарный файл в разных условиях.

- Запустите lab3 с опциональным аргументом:
  ```
  # ./lab3 arg1
  ```
- Создайте символическую ссылку на lab3 и запустите lab3 через нее:
  ```
  # ln -s lab3 lab3-1
  # ./lab3-1
  ```
- Создайте chroot-окружение и запустите lab3 внутри него:
  ```
  # mkdir -p /tmp/chroot/bin /tmp/chroot/lib64 /tmp/chroot/lib
  # mount --bind /lib64 /tmp/chroot/lib64          (в Linux)
  # mount -F lofs /lib /tmp/chroot/lib             (в Solaris)
  # cp lab3 /tmp/chroot/bin
  # chroot /tmp/chroot/ /bin/lab3
  ```

Какие показания изменяются в этих случаях?

Задание 2

Так ли страшен fork(), как его малюют — на этот вопрос мы попробуем ответить во второй части этого упражнения. В этом упражнении для каждого запускаемого в системе процесса потребуется потребуется замерять следующие показатели:

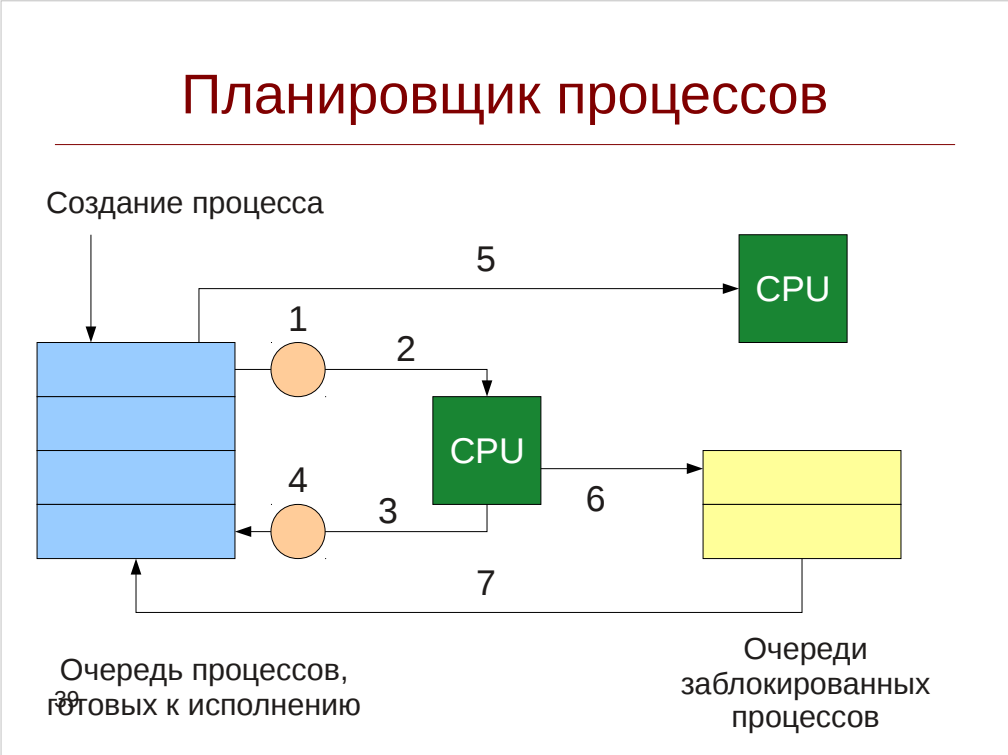
- время, затрачиваемое на выполнение системных вызовов fork() и execve();
- время, затрачиваемое на инициализацию дочернего процесса: закрытие файлов и сброс обработчиков сигналов выполняемое после fork() и перед execve();
- полезное время процесса в ходе которого выполняется код его бинарных файлов

Заметим, что из полезного времени процесса следовало бы также вычесть время, затрачиваемое на загрузку и инициализацию динамических библиотек, однако так как эта тема пока не рассматривалась в курсе, мы опустим ее.

Сохраните измеренные показатели в ассоциативный массив, ключами в котором являются имя процесса и его аргументы, а раз в секунду выводите измеренные показатели. Время замеряйте в микросекундах.

Для демонстрации работы скриптов используйте нагрузку proc_starter. Соответствующий модуль стартует оболочку sh (путь до нее задается в параметре shell) и передает ей команды из параметра запроса command. По желанию, вы можете изменить исследуемый набор команд и вероятности их возникновения.

Планировщик процессов



Планировщик процессов (scheduler) — ключевой компонент современных многозадачных (multitasking) операционных систем, так как именно он распределяет процессорное время между несколькими задачами, что и создает иллюзию параллельного выполнения программ.

Основное действие при планировании — это *переключение контекста*. При переключении контекста планировщик снимает с исполнения одну задачу и ставит другую. В Solaris функция, вызывающая переключение контекста называется swtch(), в Linux — schedule().

Переключение контекста можно разбить на следующие этапы:

1. Активная задача снимается с процессора. В DTrace эта проба называется sched:::off-cpu, в SystemTap — scheduler.cpu_off
   При этом снятие задачи может быть вызвано следующими причинами:
   - Задача блокируется на объекте синхронизации внутри ядра. **(6)** Разблокирование позже инициируется другим потоком (например при освобождении мьютекса) **(7)**
   - Задача явным образом отдает управление через вызов yield **(3)**
   - Задача вытесняется, в случае если она израсходовала время, выделенное ей планировщиком (т. н. timeslice) **(3)** Для измерения обычно служит системный таймер. Соответствующие срабатывания можно определять исходя из пробы в Solaris sched:::tick, в Linux — scheduler.tick Вытеснение исполняющейся задачи называется преемптивностью. Также, некоторые системные вызовы могут также приводить к переключению по исчерпанию времени (в Linux).

- Задача порождает новый процесс или поток. В этом случае планировщик может запустить на исполнение дочерний процесс или новый поток. **(3)**

2. Планировщик выбирает из очереди следующую задачу на исполнение. **(1)**
3. Ядро переключает контекст — при этом, в частности обновляется указатель current. В Linux это выполняется функцией context_switch(). Им соответствует высокоуровневая проба scheduler.ctxswitch, в Solaris — низкоуровневая функция resume.
4. Запускается новая задача. Соответственно, отследить это событие можно по пробам sched:::on-cpu и scheduler.cpu_on **(2)**

В многопроцессорных системах создается по одной очереди на процессор, так что возможна ситуация когда процесс перемещается с одного процессора на другой. Это называется миграцией процесса **(5)** и может быть вызвано балансировкой нагрузки. В Linux ей соответствует проба scheduler.migrate. В Solaris — сравнивая номер процессора в пробах cpu_off и cpu_on:

```
# dtrace -n '
     sched:::off-cpu {
          self->cpu = cpu; }
     sched:::on-cpu
     /self->cpu != cpu/
     {
          /* Миграция */ } '
```

Рассмотрим подробнее случай блокировки задачи. Например, это происходит при вводе данных с терминала — пока пользователь не ввел данных задача не должна занимать процессор, так как ей нечего обрабатывать. При этом процесс помещается в специальную очередь ожидания.

В DTrace определить заблокированную задачу можно по пробе sched:::sleep, а разблокирование — по событию sched:::wakeup:

```
# dtrace -n '
     sched:::sleep {
          printf("%s[%d] sleeping", execname, pid);
     }
     sched:::wakeup {
          printf("%s[%d] wakes up %s[%d]", execname, pid,
               args[1]->pr_fname, args[1]->pr_pid); }' | grep cat
```

В SystemTap к сожалению, явным образом блокирование на объектах синхронизации не описано. Однако как правило для этого используется функция add_wait_queue(), а для определения момента "просыпания" потока можно использовать пробу scheduler.wakeup.

```
# stap -e '
     probe kernel.function("add_wait_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
```

Для проверки примеров используйте следующий скипт-однострочник:

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

Строго говоря, правильный способ отслеживания блокирования процесса — трассировка функции schedule() и проверка состояния задачи — она должна быть TASK_INTERRUPTIBLE и TASK_UNINTERRUPTIBLE.

Помещение задачи в очередь на исполнение определяется в Solaris с помощью пробы sched:::enqueue, а в Linux — по вызову enqueue_task(), аналогично — при снятии задачи sched:::dequeue и dequeue_task().

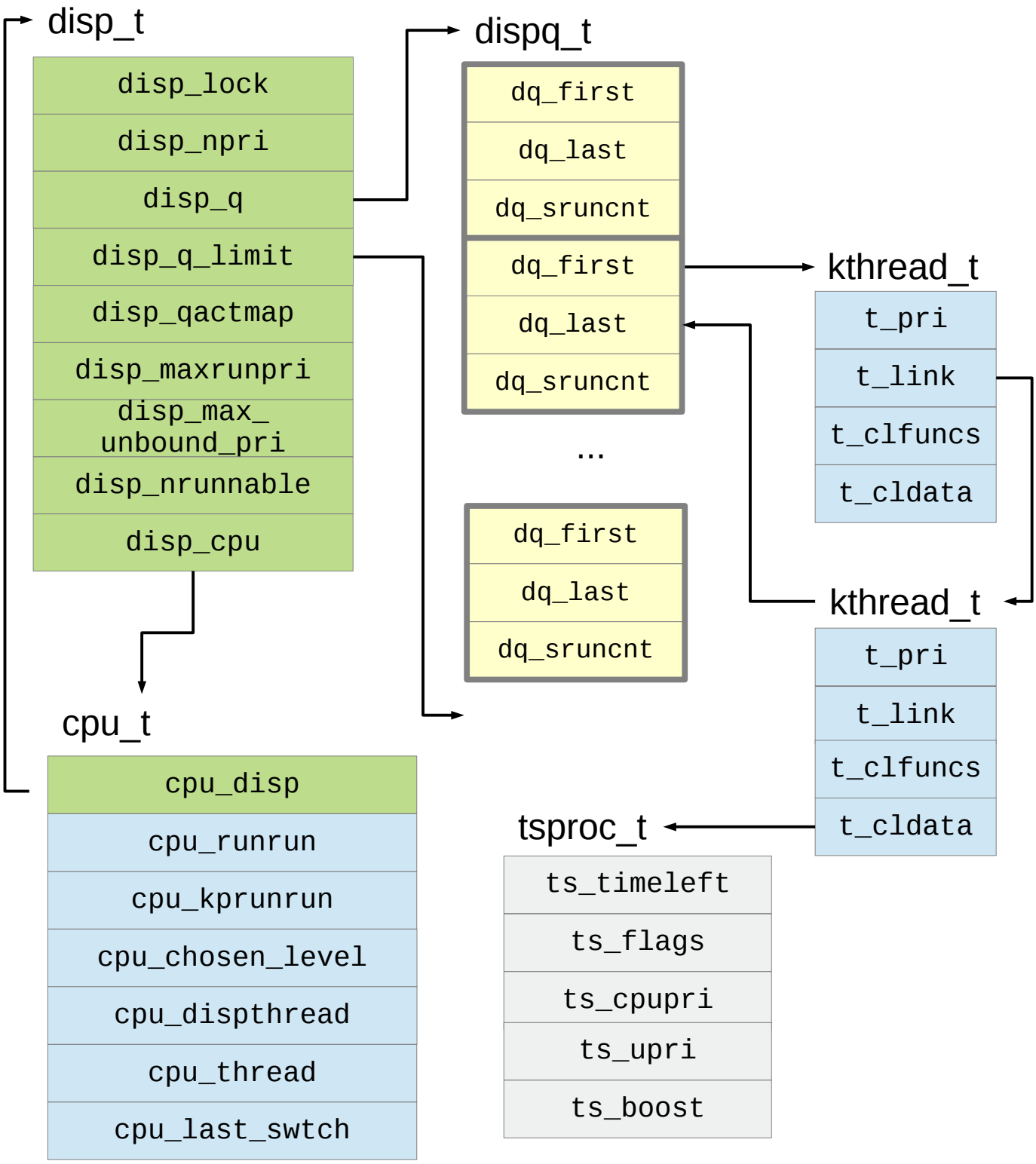
Поговорим теперь о том, как планировщик выбирает задачу на исполнение, и как решает, когда задача истратила свой временной отрезок. Классический Unix-планировщик состоит из множества очередей, каждая из которых соответствует значению приоритета процесса. Планировщик выбирает наиболее приоритетную очередь, а из нее — первый процесс в очереди. Такая дисциплина носит название циклического планирования. В справедливом планировании, наоборот: процессорное время делится поровну между всеми сущностями планирования: например пользовательскими сессиями.

В Solaris реализовано циклическое планирование с некоторыми добавлениями справедливого (класс FSS). Каждый поток kthread_t имеет приоритет t_pri от 0 до 160 и определенный класс планирования. Методы класса планирования содержатся в структуре classfuncs, на которую указывает поле t_clfuncs, а параметры потока, специфичные для класса — в поле t_cldata.

Таблица. Классы планировщика Solaris

Класс	*Приоритеты*	*Комментарий*
-	160-169	Потоки прерываний (без планировщика)
RT	100-159	RealTime — процессы реального времени
SYS	60-99	SYStem — системные процессы. По сути не имеют дисциплин планирования
SYSDC	0-99	SYStem Duty Cycles — системные процессы, интенсивно потребляющие процессорные ресурсы (например, ZFS). В отличие от класса SYS, имеют планировщик.
TS/IA	0-59	TimeSharing / InterActive — пользовательские процессы. Имеют динамические приоритеты: если поток выработал целиком свой временной отрезок, его приоритет уменьшается, чтобы процессор получил другой поток. Отличие класса InterActive в том, что процесс, соответствующий активному (находящемуся в фокусе) пользовательскому окну — к его приоритету добавляется

		небольшой бонус iaboost.
FX	0-59	FiXed — фиксированный приоритет
FSS	0-59	Fair Share Scheduler — позволяет пропорционально распределить процессорное время между проектами



Очереди планирования в Solaris относятся к структуре, описывающей процессор cpu_t, включающей в себя поля cpu_dispthread — поток, выбранный планировщиком, cpu_thread — поток, в данный момент исполняемый на процессоре, cpu_last_swtch — время в тиках (lbolts) последнего переключения контекста. Структура disp_t включает в себя структуру планировщика disp_t, и массив голов очередей dispq_t. Каждая голова очереди содержит указатели на первый и последний элемент и количество потоков в очереди dq_sruncnt, а сама очередь организована как односвязный список через указатели t_link.

Замечание: в новых обновлениях версиях Solaris 11 переменная cpu_last_swtch теперь использует немасштабируемое время высокого разрешения hrtime_t.

В структуре disp_t хранится количество очередей disp_npri (равное обычно 160), указатели на первую disp_q и следующую за последней disp_q_limit очередь, битовая карта очередей, в которых есть активные процессы disp_qactmap. Максимальный приоритет disp_maxrunpri и максимальный приоритет процесса, не привязанного к процессору disp_max_unbound_pri, а также суммарное количество активных процессов во всех очередях disp_nrunnable.

Рассмотрим подробнее планировщик TS, назначаемый Solaris по умолчанию. Основной его параметр — ts_timeleft, обозначающий оставшийся квант времени для процесса (определяется полем ts_quantum таблицы планировщика). В новых версиях Solaris ему на смену (для потоков для которых не установлен флаг TSKPRI) пришло поле ts_timer, хранящее временной интервал в тиках. Для высокоприоритетных процессов он равен 20 мс, а для низкоприоритетных — 200 мс (что компенсирует низкие шансы попасть на процессор). Приоритет в планировщике TS динамический: если процесс израсходует весь квант, то он получит приоритет ts_tqexp, а если проснется после ожидания, то ts_slpret. Посмотреть таблицу планировщика можно с помощью команды dispadmin:

```
# dispadmin -c TS -g
```

Трассировщик диспетчера Solaris представлен в следующем листинге:

Листинг 16. Скрипт tstrace.d

```
#!/usr/sbin/dtrace -qCs
/**
    tstrace.d - трассирует планировщик потоков в Solaris
    Использование: tstrace.d <cpu#>
    Замечание: на Solaris 10 определите макрос SOLARIS10

    Протестировано на Solaris 10 SPARC и на Solaris 11.1 (SPARC и x86)
*/

string classnames[struct thread_ops*];
int disp_spec;
int disp_commit;
```

```
/* Преобразует время, содержащееся в t_disp_time/cpu_last_swtch в наносекунды
    - В Solaris 10 используется время в тиках (lbolt)
    - В Solaris 11 используется немасштабированное (unscaled) hrtime_t
   Макрос HRT_CONVERT преобразует немасштабированное время в наносекунды,
   HRT_DELTA вычисляет разность с текущим системным временем */
#ifdef SOLARIS10
#   define HRT_DELTA(ns)      (`nsec_per_tick * (`lbolt64 - (ns)))
#else
#   define __HRT_CONVERT(ts)                                              \
        (((ts >> (32 - NSEC_SHIFT)) * NSEC_SCALE)        +                \
         ((((ts << NSEC_SHIFT) & 0xFFFFFFFF) * NSEC_SCALE) >> 32))
#   if defined(__i386) || defined(__amd64)
#       define NSEC_SHIFT    5
#       define NSEC_SCALE    `hrt->nsec_scale
#       define HRT_CONVERT(ts) \
           ((`tsc_gethrtime_enable) ? __HRT_CONVERT(ts) : ts )
#   elif defined(__sparc)
#       define NANOSEC       1000000000
#       define NSEC_SHIFT    4
#       define NSEC_SCALE                              \
     ((uint64_t)((NANOSEC << (32 - NSEC_SHIFT)) / `sys_tick_freq) & ~1)
#       define HRT_CONVERT       __HRT_CONVERT
#   endif
#   define HRT_DELTA(ns)       ((timestamp > HRT_CONVERT(ns)) ? timestamp -
HRT_CONVERT(ns) : 0)
#endif

#define TSINFO(t)           ((tsproc_t*) (t->t_cldata))
#define KTHREAD(t)          ((kthread_t*) (t))
#define KTHREADCPU(t)       ((kthread_t*) (t))->t_cpu->cpu_id
#define PSINFO(thread)      xlate<psinfo_t *>(thread->t_procp)

#define TSKPRI      0x01
#define TSBACKQ     0x02
#define TSIA        0x04
/* Флаги TSIA* не учитываем  */
#define TSRESTORE   0x20

/* TSFLAGSSTR выводит строковое представление флагов */
#define TSFLAGS(t)              (TSINFO(t)->ts_flags)
#define TSFLAGSSTR(t)                                                  \
        strjoin(                                                       \
            strjoin(                                                   \
                (TSFLAGS(t) & TSKPRI)    ?   "TSKPRI|" : "",       \
                (TSFLAGS(t) & TSBACKQ)   ?   "TSBACKQ|" : ""),     \
            strjoin(                                                   \
                (TSFLAGS(t) & TSIA)      ?   "TSIA|" : "",         \
                (TSFLAGS(t) & TSRESTORE) ?   "TSRESTORE" : ""))

/* Возвращает, относится ли поток t к классам TS или IA */
#define ISTSTHREAD(t)                                                   \
    ((t->t_clfuncs == &`ts_classfuncs.thread) ||                        \
     (t->t_clfuncs == &`ia_classfuncs.thread))

#define TCLNAME(t_clfuncs)      classnames[t_clfuncs]

#define DUMP_KTHREAD_INFO(hdr, thread)                                  \
    printf("\t%s t: %p %s[%d]/%d %s pri: %d\n", hdr, thread,            \
       PSINFO(thread)->pr_fname, PSINFO(thread)->pr_pid, thread->t_tid,  \
       TCLNAME(thread->t_clfuncs), thread->t_pri)
```

```
#define DUMP_DISP_INFO(disp)                                            \
    printf("\tDISP: nrun: %d npri: %d max: %d(%d)\n",                   \
            (disp)->disp_nrunnable, (disp)->disp_npri,                  \
            (disp)->disp_maxrunpri, (disp)->disp_max_unbound_pri) \

#define DUMP_CPU_INFO(cpup)                                                 \
    this->delta = HRT_DELTA((cpup)->cpu_last_swtch);                        \
    printf("\tCPU : last switch: T-%dus rr: %d kprr: %d\n",                   \
            this->delta / 1000, (cpup)->cpu_runrun, (cpup)->cpu_kprunrun); \
    DUMP_KTHREAD_INFO("\tcurrent", (cpup)->cpu_thread);                     \
    DUMP_KTHREAD_INFO("\tdisp   ", (cpup)->cpu_dispthread);                 \
    DUMP_DISP_INFO(cpup->cpu_disp)

#define TS_QUANTUM_LEFT(tspp)                   \
    ((tspp)->ts_flags & TSKPRI)                 \
        ?   (tspp)->ts_timeleft                 \
        :   (tspp)->ts_timer - (tspp)->ts_lwp->lwp_ac.ac_clock

#define DUMP_TSPROC_INFO(thread)                                        \
    printf("\tTS: timeleft: %d flags: %s cpupri: %d upri: %d boost: %d => %d\n",
\
            TS_QUANTUM_LEFT(TSINFO(thread)), TSFLAGSSTR(thread),    \
            TSINFO(thread)->ts_cpupri, TSINFO(thread)->ts_upri,     \
            TSINFO(thread)->ts_boost, TSINFO(thread)->ts_scpri )

BEGIN {
    printf("Tracing CPU%d...\n", $1);

    classnames[&`ts_classfuncs.thread]  =  "TS";
    classnames[&`ia_classfuncs.thread]  =  "IA";
    classnames[&`sys_classfuncs.thread] =  "SYS";
    classnames[&`fx_classfuncs.thread]  =  "FX";
    /* classnames[&`rt_classfuncs.thread] = "RT"; */
    classnames[&`sysdc_classfuncs.thread] = "SDC";
}

/* Вспомогательные функции:
    cpu_surrender - вызывается, когда поток покидает процессор
    setbackdq  - вызывается, когда поток помещается в хвост очереди
    setfrontdq  - вызывается, когда поток помещается в голову очереди */
fbt::cpu_surrender:entry
/cpu == $1/ {
    DUMP_KTHREAD_INFO("cpu_surrender", KTHREAD(arg0));
}

fbt::set*dq:entry
/KTHREADCPU(arg0) == $1 && ISTSTHREAD(KTHREAD(arg0))/ {
    printf("=> %s \n", probefunc);
    DUMP_KTHREAD_INFO(probefunc, KTHREAD(arg0));
    DUMP_TSPROC_INFO(KTHREAD(arg0));
}

/* Основная функция планировщика disp() */
fbt::disp:entry
/cpu == $1/ {
    disp_spec = speculation();
    disp_commit = 0;

    speculate(disp_spec);
    printf("=> disp \n");
    DUMP_CPU_INFO(`cpu[$1]);
```

```
}

fbt::disp:entry
/cpu == $1/ {
    speculate(disp_spec);   DUMP_KTHREAD_INFO("curthread: ", curthread);
}

fbt::disp:entry
/cpu == $1 && ISTSTHREAD(curthread)/ {
    speculate(disp_spec);   DUMP_TSPROC_INFO(curthread);
    disp_commit = 1;
}

fbt::disp:return
/cpu == $1/ {
    speculate(disp_spec);   DUMP_KTHREAD_INFO("disp", KTHREAD(arg1));
}

fbt::disp:return
/cpu == $1 && ISTSTHREAD(KTHREAD(arg1))/ {
    speculate(disp_spec);   DUMP_TSPROC_INFO(KTHREAD(arg1));
    disp_commit = 1;
}

fbt::disp:return
/cpu == $1 && disp_commit/ {
    commit(disp_spec);
}

fbt::disp:return
/cpu == $1 && !disp_commit/ {
    discard(disp_spec);
}

/* Функция системного тика clock_tick */
sched:::tick
/cpu == $1 && ISTSTHREAD(KTHREAD(arg0))/ {
    printf("=> clock_tick \n");
    DUMP_CPU_INFO(`cpu[$1]);
    DUMP_KTHREAD_INFO("clock_tick", KTHREAD(arg0));
    DUMP_TSPROC_INFO(KTHREAD(arg0));
}

sched:::wakeup
/KTHREADCPU(arg0) == $1 && ISTSTHREAD(KTHREAD(arg0))/ {
    printf("=> %s [wakeup] \n", probefunc);
    DUMP_CPU_INFO(`cpu[$1]);
    DUMP_KTHREAD_INFO("wakeup", KTHREAD(arg0));
    DUMP_TSPROC_INFO(KTHREAD(arg0));
}
```

Посмотрим, как динамически ведет себя планировщик TS в Solaris 11.2. Для этого запустим эксперимент нагрузчика TSLoad под названием «duality», в котором к одному и тому же аппаратному потоку будут привязаны два потока — «manager», редко выполняющий короткие задания (LWPID=7) и «worker», занимающий процессор постоянно (LWPID=5).

Для проснувшегося потока «manager» мы получим следующую ситуацию

(часть вывода сокращена):

```
=> cv_unsleep [wakeup]
        CPU : last switch: T-50073us rr: 0 kprr: 0
        wakeup t: ffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
=> setbackdq
        setbackdq t: ffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
=> setfrontdq
        setfrontdq t: ffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
=> disp
        CPU : last switch: T-50140us rr: 1 kprr: 0
                current t: ffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
                disp    t: ffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
        DISP: nrun: 2 npri: 170 max: 59(65535)
        curthread:  t: ffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
        TS: timeleft: 19 flags:  cpupri: 0 upri: 0 boost: 0 => 0
        disp t: ffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
        TS: timeleft: 3 flags:  cpupri: 59 upri: 0 boost: 0 => 0
=> disp
        CPU : last switch: T-1804us rr: 0 kprr: 0
                current t: ffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
                disp    t: ffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
        DISP: nrun: 1 npri: 170 max: 0(65535)
        curthread:  t: ffffc100054f13e0 tsexperiment[1422]/7 TS pri: 59
        disp t: ffffc10005e90ba0 tsexperiment[1422]/5 TS pri: 0
```

Обратите внимание, что спустя некоторое время приоритет потока «worker» упадет до 0, но несмотря на это диспетчер помещает вновь проснувшийся поток «manager» в хвост очереди, а исполняемый на процессоре «worker» в голову. В функции disp() будет выбран поток «manager» как более высокоприоритетный, но спустя 1,8 мс он покинет процессор, и вновь будет выбран «worker».

Теперь, в эксперименте «concurrency» запустим два «worker»-процесса:

```
=> disp
        CPU : last switch: T-39971us rr: 1 kprr: 0
                current t: ffffc10009711800 tsexperiment[1391]/6 TS pri: 40
                disp    t: ffffc10009711800 tsexperiment[1391]/6 TS pri: 40
        DISP: nrun: 2 npri: 170 max: 40(65535)
        curthread:  t: ffffc10009711800 tsexperiment[1391]/6 TS pri: 40
        TS: timeleft: 4 flags:  cpupri: 40 upri: 0 boost: 0 => 0
        disp t: ffffc10005c07420 tsexperiment[1391]/5 TS pri: 40
        TS: timeleft: 4 flags:  cpupri: 40 upri: 0 boost: 0 => 0
=> clock_tick
        clock_tick t: ffffc10005c07420 tsexperiment[1391]/5 TS pri: 40
        TS: timeleft: 3 flags:  cpupri: 40 upri: 0 boost: 0 => 0
=> clock_tick
        clock_tick t: ffffc10005c07420 tsexperiment[1391]/5 TS pri: 40
        TS: timeleft: 2 flags:  cpupri: 40 upri: 0 boost: 0 => 0
=> clock_tick
        clock_tick t: ffffc10005c07420 tsexperiment[1391]/5 TS pri: 40
        TS: timeleft: 1 flags:  cpupri: 40 upri: 0 boost: 0 => 0
        cpu_surrender t: ffffc10005c07420 tsexperiment[1391]/5 TS pri: 30
=> clock_tick
        clock_tick t: ffffc10005c07420 tsexperiment[1391]/5 TS pri: 30
        TS: timeleft: 0 flags: TSBACKQ| cpupri: 30 upri: 0 boost: 0 => 0
=> setbackdq
        setbackdq t: ffffc10005c07420 tsexperiment[1391]/5 TS pri: 30
        TS: timeleft: 8 flags:  cpupri: 30 upri: 0 boost: 0 => 0
=> disp
```

```
        curthread:  t: ffffc10005c07420 tsexperiment[1391]/5 TS pri: 30
        TS: timeleft: 8 flags:  cpupri: 30 upri: 0 boost: 0 => 0
        disp t: fffffffc80389c00 sched[0]/0 SYS pri: 60
=> disp
        curthread:  t: fffffffc80389c00 sched[0]/0 SYS pri: 60
        disp t: ffffc10009711800 tsexperiment[1391]/6 TS pri: 40
        TS: timeleft: 4 flags:  cpupri: 40 upri: 0 boost: 0 => 0
```

Обратите внимение на поле timeleft — оно соответствует разности между полями ts_timer и ts_lwp->lwp_ac.ac_clock. При каждом срабатывании системного таймера (clock_tick) последнее увеличивается на 1, и соответственно уменьшается число timeleft. Так как поток «worker» выбрал весь квант времени, то по его окончании, приоритет потока уменьшается до 30, а он сам помещается в хвост соответствующей очереди. Также между потоками вклинился системный поток («sched»).

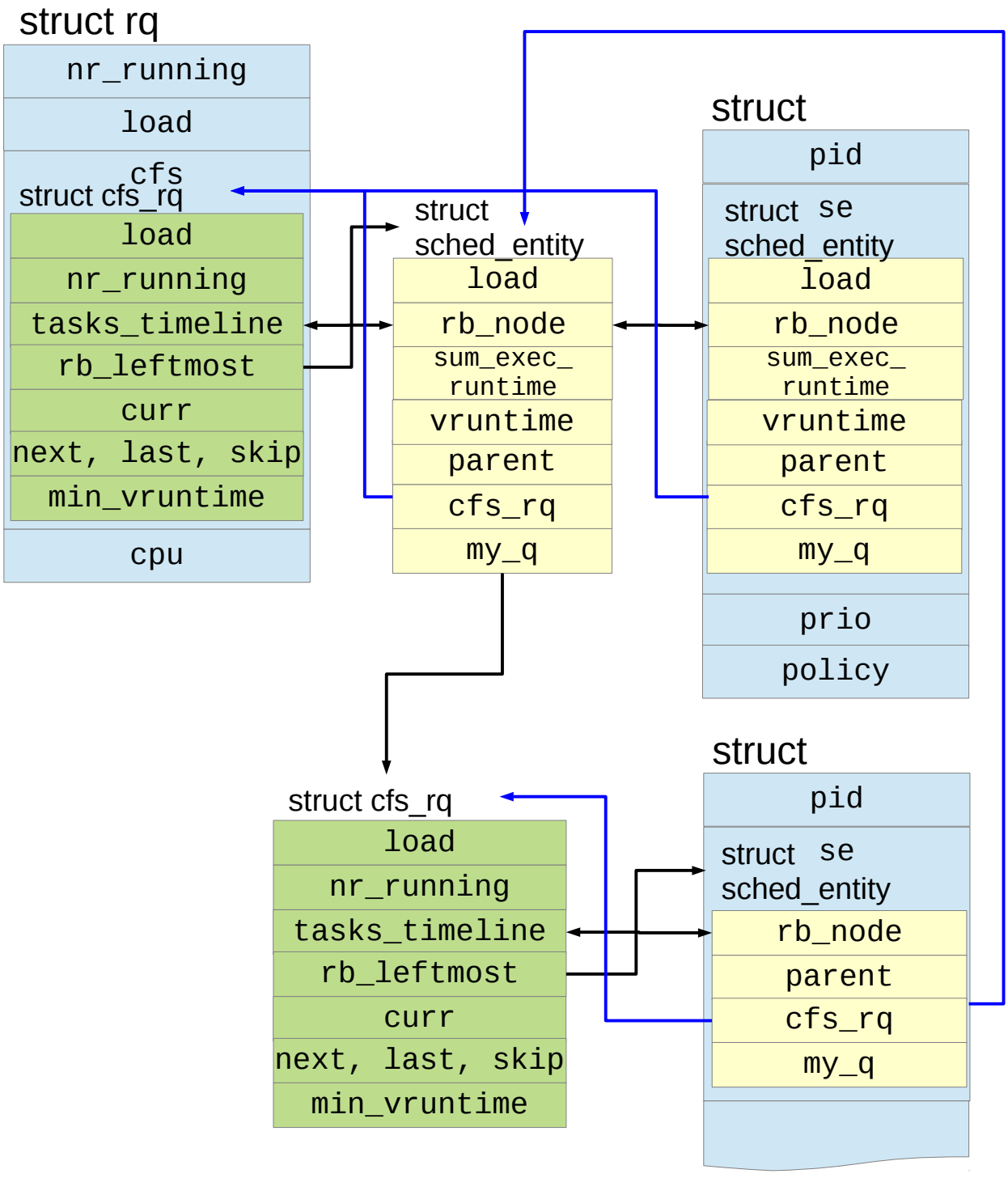
В Linux также использовалось циклическое планирование, реализуемое планировщиком O(1), однако в ядре 2.6.32 он был заменен планировщиком Completely Fair Scheduler (CFS). Механизмы циклического планирования продолжают использоваться классом планировщика RT (Run-Time). Кроме них, есть ряд нестандартных планировщиков, таких как BFS. Дисциплина планирования определяется полем policy (политика) структуры task_struct.

Таблица. Классы и политики планирования Linux.

# n/n	*Класс*	*Политика*	*Комментарий*
1	stop	-	Процесс остановки процессора. Всегда имеет наивысший приоритет.
2	rt	SCHED_RR	Реализует циклическое планирование по дисциплине Round-Robin или FIFO
		SCHED_FIFO	
3	fair (CFS)	SCHED_NORMAL (SCHED_OTHER)	Политика по-умолчанию. Используется большинством пользовательских процессов
		SCHED_BATCH	Пакетные задания. В отличие от SCHED_NORMAL, не требует интерактивности — проснувшееся пакетное задание не может прервать текущий исполняемый процесс.
4	idle	SCHED_IDLE	Фоновый процесс. Выбирается, только если остальные классы не имеют готовых процессов

Рассмотрим планировщик CFS подробнее. Как следует из его названия, он реализует механизмы справедливого планирования, выделяющий вычислительные ресурсы пропорционально сущностям планирования (им соответствует структура sched_entity), а сами эти сущности являются процессами или очередями таких

сущностей — структура cfs_rq. Очереди сущностей могут быть сформированы автоматически через механизм autogrouping или через подсистему CGroup.



Сущности планирования содержат поле load, определяющее ее весовой коэффицент (он содержится в подполе weight), а в поле load очереди cfs_rq содержится сумма всех весовых коэффициентов. Выделяемый сущности временной отрезок определяется как соотношение между ними:

$$slice = \frac{\text{se->load.weight}}{\text{cfs\_rq->load.weight}} \cdot period$$

Тогда, если есть одна задача с весом 1024 (он соответствует процессу со значением nice 0) и другая с весом 655 (nice = 2), то первая получит ~61% кванта, а вторая — 39%. Заметим, однако, что если планировщик использует системный тик, а не таймеры высокого разрешения для переключения контекста, то эти значения будут смещаться.

Вторая характеристика сущности планирования — это поле vruntime, содержащее взвешенное время исполнения процесса на процессоре. Оно вычисляется при помещении процесса в очередь исполняемых при его активации:

$$vruntime = \text{cfs\_rq->min\_vruntime} - k * latency$$

или после снятия процесса с процессора:

$$vruntime += \frac{\text{NICE\_0\_LOAD}}{\text{se->load.weight}} \cdot \text{delta\_exec} \quad .$$

Чем меньше значение vruntime, тем более высокий приоритет имеет процесс.

Структура rq — это очередь исполняемых процессов, соответствующая процессору с номером cpu. Она содержит суммарный вес всех процессов load, их количество nr_running и две структуры, описывающие очереди — поля cfs и rt. Структура cfs_rq, соответствующая очереди планировщика CFS, хранит все сущности планирования в красно-черном дереве tasks_timeline, отсортированные по времени vruntime (указатель на самый левый элемент rb_leftmost, то есть на самый высокоприоритетную сущность планирования). Кроме этого, есть ряд специальных указателей: curr указывает на текущую сущность планирования, next — недавно проснувшуюся, last — недавно вытесненную проснувшейся, skip — вызвавшую системный вызов sched_yield().

Реализуем также трассировку планировщика CFS:

Листинг 17. Скрипт cfstrace.stp

```
#!/usr/bin/env stap
/**
 * cfstrace.stp
 *
 * Трассирует переключение процессов планировщиком CFS Linux
 * Использование:
 *      cfstrace.stp CPU
 *
 * Оттестировано на 3.10 (CentOS 7)
 */

/* Глобальные параметры */
global be_verbose = 1;
global print_cfs_tree = 1;

global pnetime;
global cpetrace;

global trace_cpu;

global policy_names;
```

```
global cpu_cfs_rq;
global rq_curr;
global rq_clock_task;

/* get_task_se возвращает task_struct для se */
function get_task_se:long(se:long) {
    offset = &@cast(0, "struct task_struct")->se;
    return se - offset;
}

/* get_task_se возвращает sched_entity для task */
function get_se_task:long(task:long) {
    offset = &@cast(0, "struct task_struct")->se;
    return task +  offset;
}

/* get_rb_se возвращает sched_entity для rb */
function get_rb_se:long(rb:long) {
    offset = &@cast(0, "struct sched_entity")->run_node;
    return rb - offset;
}

/* возвращает строку, соответствующую задаче t */
function sprint_task(t:long) {
    policy = @cast(t, "struct task_struct")->policy;
    return sprintf("t: %p %s/%d %s", t, task_execname(t), task_tid(t),
                   policy_names[policy]);
}

function sprint_cfs_rq_path(cfs_rq:long) {
    tg = @cast(cfs_rq, "struct cfs_rq")->tg;
    if(!tg)
        return "???";

%( CONFIG_SCHED_AUTOGROUP == "y"
%?
    if(@cast(tg, "struct task_group")->autogroup) {
        return sprintf("/autogroup-%d",
                       @cast(tg, "struct task_group")->autogroup->id);
    }
%)

    cgroup = @cast(tg, "struct task_group")->css->cgroup;

    try {
        return reverse_path_walk(@cast(cgroup, "struct cgroup")->dentry);
    }
    catch {
        return "/???";
    }
}

function sprint_vruntime(se:long, min_vruntime:long) {
    vruntime = @cast(se, "struct sched_entity")->vruntime;
    if(min_vruntime)
        return sprintf("MIN+%d", vruntime - min_vruntime);
    else
        return sprintf("%d", vruntime);
}
```

```
/* Выводит информацию о сущности планирования se */
function print_se(s:string, se:long, verbose:long, min_vruntime:long) {
    printf("\tse:%8s ", s);

    my_q = @cast(se, "struct sched_entity")->my_q;

    if(my_q == 0) {
        println(sprint_task(get_task_se(se)));
    } else {
        printf("se: %p my_q: %p %s", se, my_q, sprint_cfs_rq_path(my_q));
    }

    if(verbose) {
        printf("\t\tload.weight: %d exec_start: RQ+%d vruntime: %s
sum_exec_runtime: %d\n",
               @cast(se, "struct sched_entity")->load->weight,
               rq_clock_task - @cast(se, "struct sched_entity")->exec_start,
               sprint_vruntime(se, min_vruntime),
               @cast(se, "struct sched_entity")->sum_exec_runtime);
    }
}

/* Выводит информацию об очереди cfs_rq */
function print_cfs_rq(cfs_rq:long, verbose:long) {
    firstrb = @cast(cfs_rq, "struct cfs_rq")->rb_leftmost;
    skip = @cast(cfs_rq, "struct cfs_rq")->skip;
    last = @cast(cfs_rq, "struct cfs_rq")->last;
    nextse = @cast(cfs_rq, "struct cfs_rq")->next;
    min_vruntime = @cast(cfs_rq, "struct cfs_rq")->min_vruntime;

    printf("\tCFS_RQ: %s\n", sprint_cfs_rq_path(cfs_rq));

    if(verbose) {
        printf("\t\tnr_running: %d load.weight: %d min_vruntime: %d\n",
            @cast(cfs_rq, "struct cfs_rq")->nr_running,
            @cast(cfs_rq, "struct cfs_rq")->load->weight,
            @cast(cfs_rq, "struct cfs_rq")->min_vruntime);
        if(@defined(@cast(cfs_rq, "struct cfs_rq")->runnable_load_avg)) {
            printf("\t\trunnable_load_avg: %d blocked_load_avg: %d \n",
                   @cast(cfs_rq, "struct cfs_rq")->runnable_load_avg,
                   @cast(cfs_rq, "struct cfs_rq")->blocked_load_avg);
        }
        else {
            printf("\t\tload_avg: %d\n",
                   @cast(cfs_rq, "struct cfs_rq")->load_avg);
        }
    }

    if(firstrb) {
        firstse = get_rb_se(firstrb);
        print_se("first", firstse, verbose, min_vruntime);
    }

    if(skip) print_se("skip", skip, 0, min_vruntime);
    if(last) print_se("last", last, 0, min_vruntime);
    if(nextse) print_se("next", nextse, 0, min_vruntime);

    if(print_cfs_tree)
        dump_cfs_rq(cpu_cfs_rq, be_verbose, min_vruntime);
}
```

```
function dump_cfs_rq_rb(indstr:string, rb:long, verbose:long, min_vruntime:long)
{
    left  = @cast(rb, "struct rb_node")->rb_left;
    right = @cast(rb, "struct rb_node")->rb_right;

    print_se(sprintf("%s:", indstr), get_rb_se(rb), verbose, min_vruntime);

    if(left) dump_cfs_rq_rb(sprintf("%s-l", indstr), left, verbose,
min_vruntime);
    if(right) dump_cfs_rq_rb(sprintf("%s-r", indstr), right, verbose,
min_vruntime);
}

/* Выводит дерево tasks_timeline очереди cfs_rq */
function dump_cfs_rq(cfs_rq:long, verbose:long, min_vruntime:long) {
    root = @cast(cfs_rq, "struct cfs_rq")->tasks_timeline->rb_node;

    if(root)
        dump_cfs_rq_rb("rb", root, verbose, min_vruntime);
}

probe begin {
    pnetime = local_clock_ns();
    cpu_cfs_rq = 0;    rq_curr = 0;

    trace_cpu = $1;

    printf("Tracing CPU%d...\n", trace_cpu);

    policy_names[0] = "SCHED_NORMAL";   policy_names[1] = "SCHED_FIFO";
    policy_names[2] = "SCHED_RR";       policy_names[3] = "SCHED_BATCH";
    policy_names[4] = "SCHED_ISO";      policy_names[5] = "SCHED_IDLE";
}

/* pick_next_task_fair пытается найти следующую подходящую задачу на
   исполнение. pick_next_entity - функция-помощник для работы с группами */
probe kernel.function("pick_next_task_fair") {
    if(cpu() != trace_cpu) next;

    pnetime2 = local_clock_ns();
    printf("=> pick_next_task_fair D=%d J=%d\n", pnetime2 - pnetime, $jiffies);

    pnetime = pnetime2;
    cpu_cfs_rq = &$rq->cfs;
}

probe kernel.function("pick_next_task_fair").return {
    if(cpu() != trace_cpu || cpu_cfs_rq == 0) next;

    printf("<= pick_next_task_fair\n");

    if($return != 0) {
        se = &$return->se;
        print_se("sched", se, 0,
                 @cast(cpu_cfs_rq, "struct cfs_rq")->min_vruntime);
    }

    println("");
    cpu_cfs_rq = 0;
}
```

```
probe kernel.function("pick_next_entity") {
    if(cpu() != trace_cpu || cpu_cfs_rq == 0) next;

    printf("\tpick_next_entity\n");
    print_cfs_rq(cpu_cfs_rq, be_verbose);
}

/* task_tick_fair - функция системного тика*/
probe kernel.function("task_tick_fair") {
    if(cpu() != trace_cpu) next;

    printf("=> task_tick_fair J=%d queued: %d curr: %s\n", $jiffies, $queued,
           sprint_task($curr));

    rq_curr = get_se_task($curr);
    cpu_cfs_rq = &$rq->cfs;
}

probe kernel.function("sched_slice").return {
    if(cpu() != trace_cpu) next;

    printf("\tsched_slice: %d\n", $return);
}

probe kernel.function("task_tick_fair").return {
    if(cpu() != trace_cpu) next;

    print_se("curr", rq_curr, be_verbose,
             @cast(cpu_cfs_rq, "struct cfs_rq")->min_vruntime);
    printf("\t\tdelta_exec: %d\n",
           @cast(rq_curr, "struct sched_entity")->sum_exec_runtime -
           @cast(rq_curr, "struct sched_entity")->prev_sum_exec_runtime);

    firstrb = @cast(cpu_cfs_rq, "struct cfs_rq")->rb_leftmost;
    if(firstrb) {
        firstse = get_rb_se(firstrb);
        printf("\t\tdelta: %d\n",
            @cast(rq_curr, "struct sched_entity")->vruntime -
            @cast(firstse, "struct sched_entity")->vruntime);
    }

    cpu_cfs_rq = 0;
    rq_curr = 0;

    println("<= task_tick_fair\n");
}

/* check_preempt_wakeup - проверяет, может ли просыпающаяся задача
    вытеснить текущую */
probe kernel.function("check_preempt_wakeup") {
    if(cpu() != trace_cpu) next;
    cpetrace = 1;

    cpu_cfs_rq = &$rq->cfs;
    min_vruntime = @cast(cpu_cfs_rq, "struct cfs_rq")->min_vruntime;

    t_curr = task_current();
    t_se = $p;

    println("=> check_preempt_wakeup:");
    print_se("curr", get_se_task(t_curr), be_verbose, min_vruntime);
```

```
    print_se("se", get_se_task(t_se), be_verbose, min_vruntime);
}

probe kernel.function("check_preempt_wakeup").return {
    if(cpu() != trace_cpu) next;

    cpetrace = 0;

    print_cfs_rq(cpu_cfs_rq, be_verbose);
    println("<= check_preempt_wakeup\n");
}

/* resched_task - перепланировать задачу. В новых версиях
   ядер заменена на resched_curr. */
probe kernel.function("resched_task") {
    if(cpu() != trace_cpu || cpetrace == 0) next;

    if(@defined($p))
       task_string = sprint_task($p)
    else
       task_string = "???";

    printf("\tresched_task %s\n", task_string);
}

probe kernel.function("update_rq_clock").return {
    rq_clock_task = $rq->clock_task;
}
```

Проведем те же эксперименты, что и для Solaris. В эксперименте «duality» поток «manager» (TID=6063) вытесняет задачу не сразу, однако будучи единственным процессом в дереве (процесс «worker» находится на процессоре и потому в дереве не сохраняется), он сразу становится крайним левым. При следующем системном тике он и выбирается на планирование. Обратите внимание, что для процесса «worker» (TID=6061) в этот момент vruntime уже превышает min_vruntime очереди: `MIN+260001`.

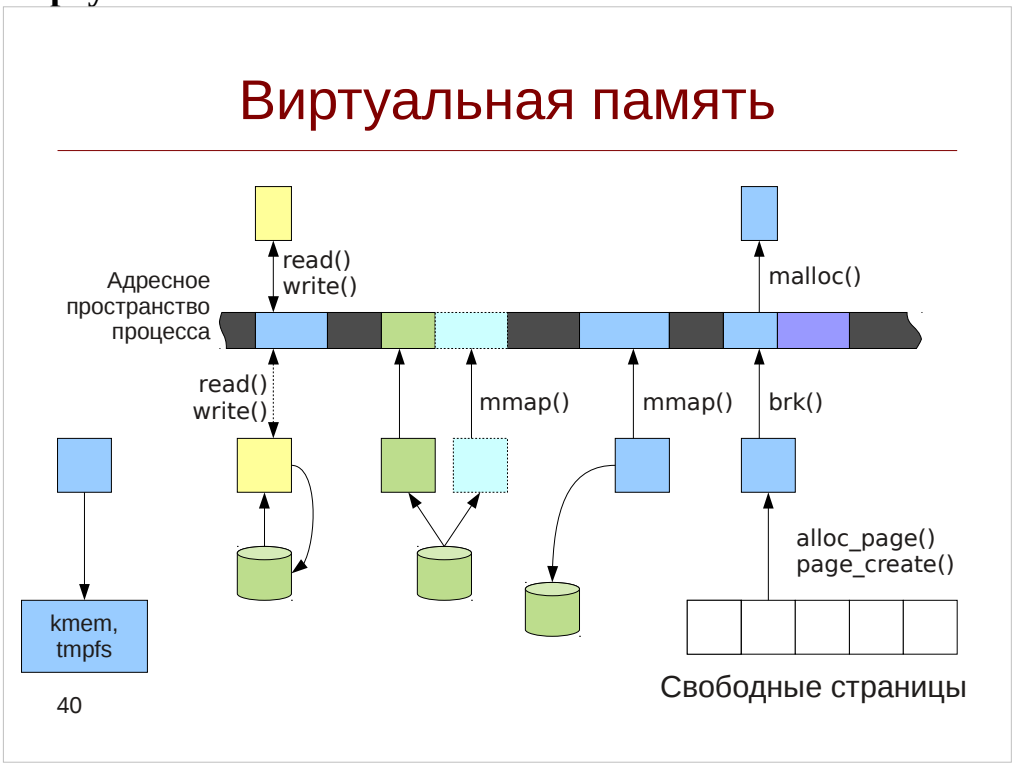
```
=> check_preempt_wakeup:
    se:    curr tsexperiment/6061 SCHED_NORMAL
        load.weight: 1024 exec_start: RQ+-978205 vruntime: MIN+0
    se:      se tsexperiment/6063 SCHED_NORMAL
        load.weight: 1024 exec_start: RQ+41023325 vruntime: MIN+-6000000
    CFS_RQ: /
        nr_running: 2 load.weight: 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
```

В эксперименте «concurrency», потоку выделяется квант времени в 6 мс, так что при каждом тике увеличивается vruntime текущего процесса пока он не нагонит min_vruntime:

```
=> pick_next_task_fair D=7015045 J=4302974601
<= pick_next_task_fair
    se:   sched t: 0xffff880015ba0000 tsexperiment/6304 SCHED_NORMAL

=> task_tick_fair J=4302974602 queued: 0 curr: t: 0xffff880015ba0000
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: 0xffff880015ba0000
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
```

Виртуальная память

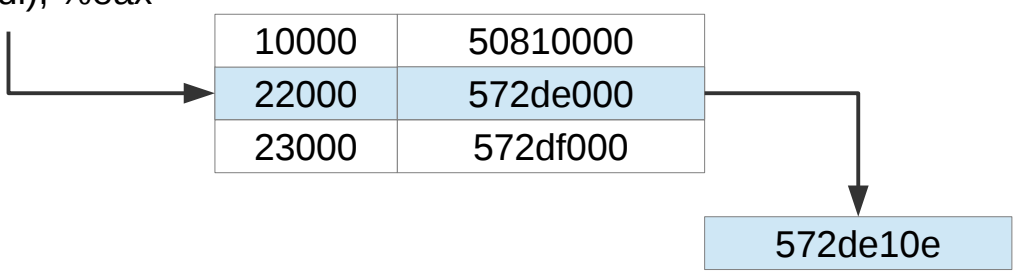


Исторически ЭВМ работали в условиях недостатка физической памяти. С появлением многозадачного режима, все еще более усложнилось, так как теперь в памяти необходимо было располагать одновременно несколько программ. Кроме того, требовалось гарантировать, что одновременно запущенные программы не смогут получить доступ к данным друг друга и повредить их. Все это привело к появлению механизма виртуальной памяти, в частности страничной организации.

При такой организации вся физическая память разделяется на небольшие блоки — страницы, размером 4кб для архитектуры x86 (чтобы избежать фрагментации, существует также поддержка т. н. больших страниц (huge page). При этом процессор при исполнении команды, например записи в память, обращается не к физической памяти а берет адрес, и ищет запись в каталоге страниц, соответствующую этому адресу, и только после этого вычисляет физический адрес в оперативной памяти, как показано на рисунке:

%edi = 2210e

mov (%edi), %eax



Рассмотрение особенностей реализации каталогов страниц мы оставим за рамками нашего курса.

С точки зрения ядра виртуальная память представляется двумя наборами структур данных и подсистем:

- Представление адресного пространства процесса, разделенного на сегменты. Каждый такой сегмент имеет базовый адрес, размер и права доступа.
- Аллокатор страниц и структура, описывающая страницу памяти на уровне ядра (и соответственно, запись в каталоге страниц). Страницы составляют сегменты памяти.

Рассмотрим, какие сегменты будут созданы при запуске в ОС Solaris процесса cat. Для этого воспользуемся утилитой pmap:

```
bash# pmap 12853
12853:  cat
00010000       8K r-x--  /usr/bin/cat
00022000       8K rwx--  /usr/bin/cat
00024000      40K rwx--    [ heap ]
FF200000    1216K r-x--  /lib/libc.so.1
FF330000      40K rwx--  /lib/libc.so.1
FF33A000       8K rwx--  /lib/libc.so.1
FF390000      24K rwx--    [ anon ]
FF3A0000       8K r-x--  /platform/sun4u-us3/lib/libc_psr.so.
FF3B0000     208K r-x--  /lib/ld.so.1
FF3F0000       8K rwx--    [ anon ]
FF3F4000       8K rwx--  /lib/ld.so.1
FF3F6000       8K rwx--  /lib/ld.so.1
FFBFE000       8K rw---    [ stack ]
 total      1592K
```

На Linux можно также кроме нее посмотреть виртуальный файл /proc/PID/maps

В первой колонке указан виртуальный адрес в адресном пространстве процесса, затем размер сегмента, права доступа к нему и назначение. Приложение может содержать следующие сегменты:

- Текстовые сегменты (иначе говоря, содержащие программный код) и сегменты данных самого исполнимого файла и разделяемых библиотек. При этом сегменты кода имеют установленные биты r и x (чтение и исполнение), а данных еще и бит w (запись). Строго говоря, такие сегменты создаются загрузчиком `ld.so` с помощью системных вызовов `mmap()`.
- Стек (обозначается как `[ stack ]`) . Понятие стека было рассмотрено в разделе «Динамический анализ кода» на с. 66. Стек расширяется автоматически ядром операционной системы при достижении его головы. В случае многопоточных программ каждый из потоков имеет свой собственный стек.
- Куча (обозначается как `[ heap ]`) - динамически распределяемая область памяти. Для выделения памяти в программах на C используется функция `malloc()`, а для расширения области — системный вызов `brk()`.

- Анонимные области памяти (обозначаются как `[ anon ]`) и отображенные в память файлы (memory-mapped files). Эти области памяти создаются явно с помощью системного вызова `mmap()`, а уничтожаются с помощью `munmap()`.
- Сегменты общей памяти System V.

В SystemTap трассировка системных вызовов и выполняется с помощью tapset'a vm:

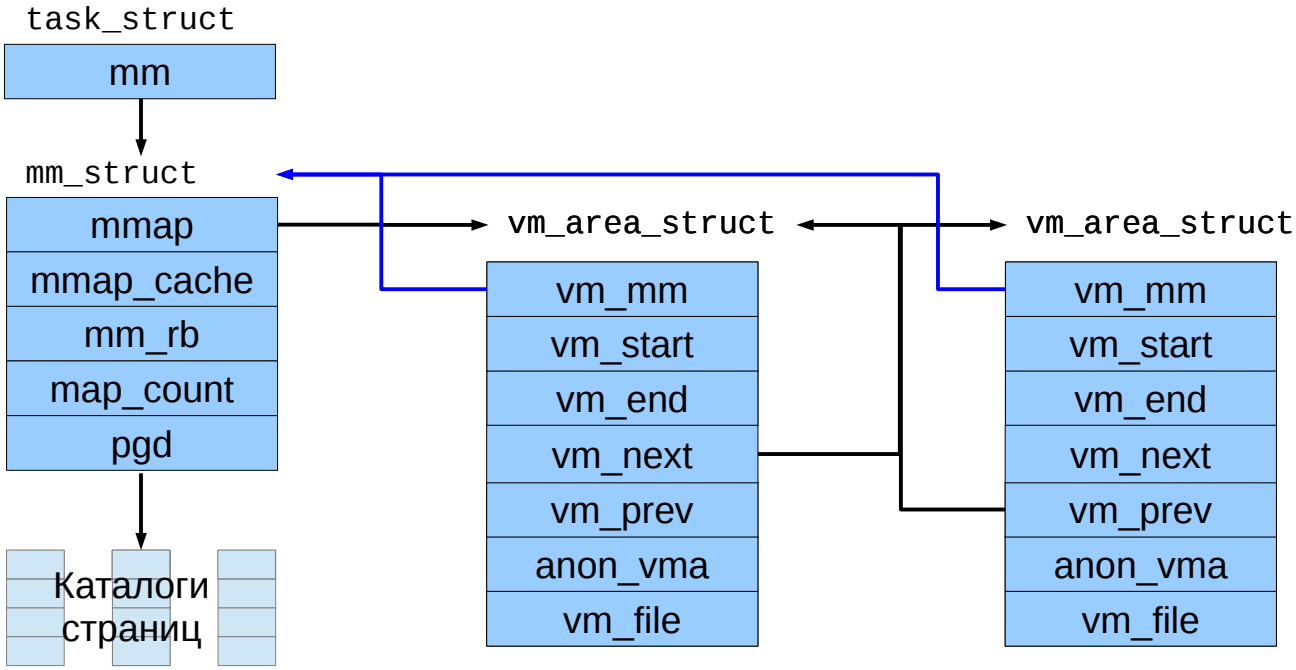
```
# stap -d $(which python) --ldd -e '
     probe vm.brk, vm.mmap, vm.munmap {
          printf("%8s %s/%d %p %d\n",
               name, execname(), pid(), address, length);
          print_ubacktrace();
     }' -c 'python -c "range(10000)"'
```

Для DTrace и Solaris можно воспользоваться привязками к операциям над адресным пространством as_map и as_unmap:

```
# 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)"
```

При выполнении обоих скриптов, мы увидим, что и в Linux и в Solaris функция builtin_range, соответствующая функции range() языка Python приводит к выделению памяти на куче.

Со стороны процесса его адресное пространство представляется структурами as в Solaris и mm_struct в Linux:



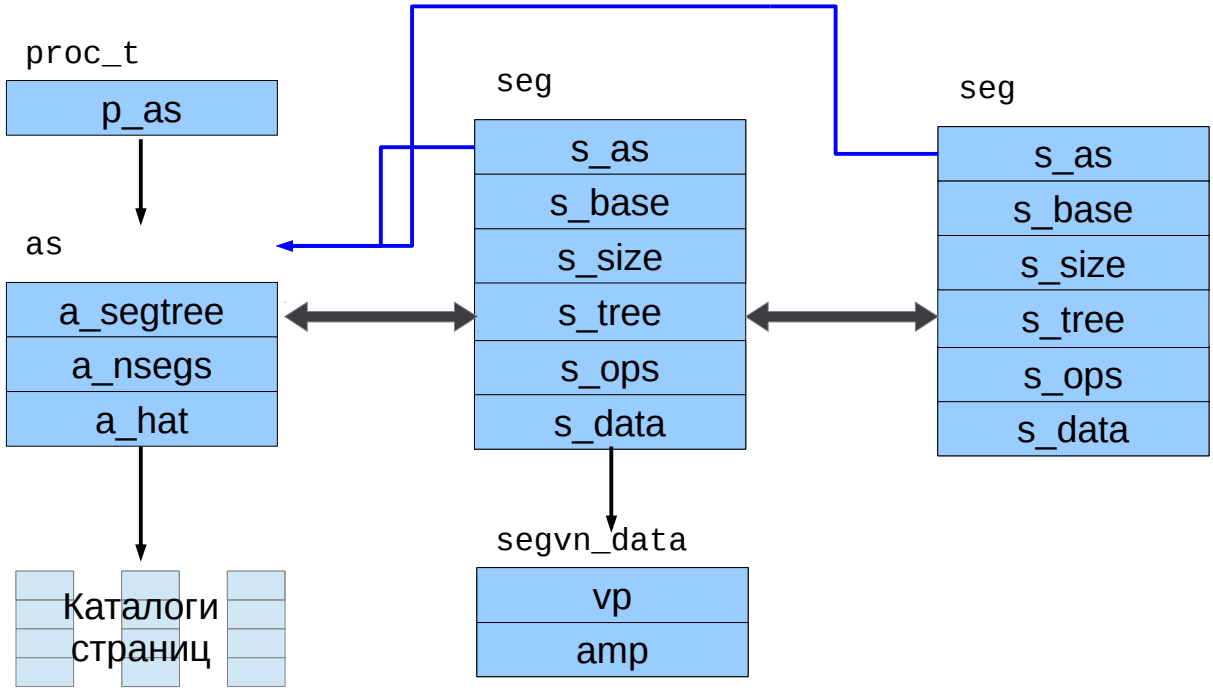
В Linux каждый сегмент представляется структурой vm_area_struct, для которой определены: виртуальный адрес начала vm_start и конца vm_end сегмента, а

также поле для анонимного отображения anon_vma или для файлов — поле vm_file. Внутри mm_struct сегменты организованы как в древовидную структуру (через поле mm_rb), так и в циклический список, головой которого является mm->mmap, а полями, указывающими на элементы внутри списка — vm_next и vm_prev.

Кроме того, в структуре mm хранятся счетчики потребляемой памяти процессом, которые доступны через tapset memory и функции proc_mem_*, например:

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

В Solaris организация сегментов похожа, однако не существует связанного

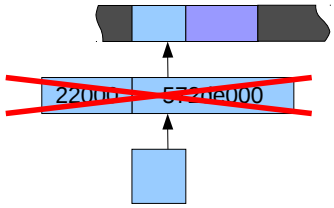


Кроме того, все назначение сегментов определяется полями s_ops, содержащим таблицу операций сегмента и s_data содержащий приватные данные. Например для сегментов, основанных на vnode структура таких приватных данных — segvn_data, содержащая поля vp и amp, описывающие отображенный файл и таблицу анонимных отображений соответственно.

Кроме процессов, память используется внутри самого ядра, например для аллокатора ядра kmem, файловой системы tmpfs. Работа с файловыми системами выполняется через страничный кеш — данные, читаемые или записываемые на диск предварительно кешируются с целью уменьшить количество дисковых операций. В случае нехватки памяти в системе, ядро может переместить неиспользуемые страницы на дисковый раздел подкачки (swap). Аналогичные механизмы используются и для записи страниц файловой системы из страничного кеша.

Страничный сбой (pagefault)

- Minor
  - Не создана запись в каталоге
  - Страница не была выделена
- Major
  - Необходимо чтение с диска
- Invalid
  - Доступ к отсутствующему адресу
  - Нарушение прав доступа к памяти (protection)



41

В случае, если MMU (модуль управления памятью) процессора обнаруживает, что исполняемая инструкция пытается обратиться к области памяти, для которой не была создана запись в каталоге страниц, он генерирует специальное исключение, называемое страничный сбой (pagefault), позволяя операционной системе обработать такую ситуацию. Операционная система в свою очередь выясняет причину такого поведения, и в зависимости от этого реагирует:

- *Незначительным (minor)* сбоем считается ситуация, когда запись в каталоге не была создана. Например при выделении памяти приложением, Solaris и Linux не выделяют страницы, дожидаясь первого обращения к ней — это приводит к незначительным страничным сбоям.
- *Значительным (major)* считается сбой, при котором требуется чтение с диска. Это может быть вызвано обращением к отображенному в память файлу или случаем, когда память приложения была вытеснена на дисковое устройство подкачки.
- *Некорректным (invalid)* считается сбой, когда приложение пытается получить доступ используя адрес, не соответствующий ни одному из сегментов его адресного пространства, или с нарушением прав (например, запись в сегмент кода, для которого запись запрещена). В этом случае приложение получает сигнал SIGSEGV. Также такой страничный сбой может использоваться чтобы отследить запись при ситуации copy-on-write.

Заметим, что страничные сбои не обязательно могут происходить и в адресном пространстве ядра. Более того, они используются самими средствами инструментирования, в случае если запущенный скрипт вызвал страничный сбой (например при попытке обратиться к некорректному адресу внутри ядра).

В Linux профилирование страничных сбоев можно производить по пробе

vm.pagefault.return и установленных там битов в переменной fault_type. Также в Linux есть точки трассировки mm_anon_* и mm_filemap_*, однако они специфичны для RedHat-подобных дистрибутиов. В Solaris для этого предназначен провайдер vminfo:

Страничный сбой	*DTrace*	*SystemTap*
Любой	vminfo:::as_fault	perf.sw.page_faults
Minor		perf.sw.page_faults_min
Major	vminfo:::maj_fault За ним следует vminfo:::pgin	perf.sw.page_faults_maj
Invalid	vminfo:::cow_fault — сбой для copy-on-write страниц vminfo:::prot_fault — нарушение прав или доступ к некорректной области памяти	См. замечание 1

Замечание 1: в Linux отсутствуют пробы для профилирования некорректных страничных сбоев. Такие ситуации обрабатываются внутри платформо-специфичной функции do_page_fault. На платформе x86 реализован набор функций bad_area*, которые можно использовать для отслеживания таких ситуаций:

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

Замечание 2: Пробы perf по умолчанию срабатывают только после достижения определенного количества событий. Изменить это поведение можно указав perf-пробу в сыром виде, например так:

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

Значения для параметров type и config можно посмотреть в файле /usr/share/systemtap/perf.stp

Обработка страничного сбоя в Solaris выполняется функцией as_fault:

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

Для того чтобы определить, к какой области памяти относится проблемный адрес, мы будем перехватывать вызовы функции as_segat из as_fault, которая возвращает соответствующий сегмент адресного пространства. Для сегментов, соотносящихся с драйвером segvn мы также сможем узнать название отображенного в память файла. Если же страничный сбой был связан с обращением по некорректному адресу, то as_segat вернет NULL.

Листинг 18. Скрипт pagefault.d

```
#!/usr/sbin/dtrace -qCs

/**
 * pagefault.d
 *
 * Трассирует страничные сбои, обрабатываемые функцией as_fault
 *
 * Оттестировано на Solaris 11
 */

string fault_type[4];
string seg_rw_type[6];
string prot[8];

#define DUMP_AS_FAULT()                                 \
    printf("as_fault pid: %d as: %p\n", pid, self->as); \
    printf("\taddr: %p size: %d flags: %s|%s \n",       \
        self->addr, self->size,                         \
        fault_type[self->pf_type],                      \
        seg_rw_type[self->pf_rw]                        \
    )

#define PROT(p) prot[(p) & 0x7],                        \
                  ((p) & 0x8) ? "u" : "-"

#define VNODE_NAME(vp)  (vp)                            \
                        ? ((vp)->v_path)                \
                            ? stringof((vp)->v_path)    \
                            : "???"                     \
                        : "[ anon ]"

#define DUMP_SEG_VN(seg, seg_vn)                        \
    printf("\t[%p:%p] %s%s\n\tvn: %s\n\tamp: %p:%d \n", \
        (seg)->s_base, (seg)->s_base + (seg)->s_size,   \
        PROT((seg_vn)->prot), VNODE_NAME((seg_vn)->vp), \
        (seg_vn)->amp, (seg_vn)->anon_index             \
    )

#define IS_SEG_VN(s)  (((struct seg*) s)->s_ops == &`segvn_ops)

BEGIN {
    /* See vm/seg_enum.h */
    fault_type[0] = "F_INVAL";          fault_type[1] = "F_PROT";
    fault_type[2] = "F_SOFTLOCK";       fault_type[3] = "F_SOFTUNLOCK";

    seg_rw_type[0] = "S_OTHER";         seg_rw_type[1] = "S_READ";
    seg_rw_type[2] = "S_WRITE";         seg_rw_type[3] = "S_EXEC";
    seg_rw_type[4] = "S_CREATE";        seg_rw_type[5] = "S_READ_NOCOW";

    prot[0] = "---";                    prot[1] = "r--";
    prot[2] = "-w-";                    prot[3] = "rw-";
    prot[4] = "--x";                    prot[5] = "r-x";
    prot[6] = "-wx";                    prot[7] = "rwx";
}

fbt::as_fault:entry {
    self->in_fault = 1;
```

```
    self->as      = args[1];
    self->addr    = args[2];
    self->size    = args[3];

    self->pf_type = args[4];
    self->pf_rw   = args[5];
}

fbt::as_fault:return
{
    self->in_fault = 0;
}

fbt::as_segat:return
/self->in_fault && arg1 == 0/
{
    DUMP_AS_FAULT();
}

fbt::as_segat:return
/self->in_fault && arg1 != 0 && IS_SEG_VN(arg1)/
{
    this->seg = (struct seg*) arg1;
    this->seg_vn = (segvn_data_t*) this->seg->s_data;

    DUMP_AS_FAULT();
    DUMP_SEG_VN(this->seg, this->seg_vn);
}
```

Этот скрипт трассирует страничные сбои и формирует записи следующего вида:

```
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
```

Как видим, страничный сбой произошел при попытке записи в сегмент данных (об этом говорят права доступа rwx), а его тип — нарушение прав доступа (флаг F_PROT), что указывает на copy-on-write страничный сбой. Поле amp относится к структурам анонимной памяти.

Если запустить скрипт параллельно с выделением и инициализацией большого объема памяти, например как это было в примере с конструкцией Python `range(4000000)`, то можно увидеть большое количество страничных сбоев F_INVAL с последовательно идущими адресами:

```
as_fault pid: 987 as: ffffc10008fc6110
        addr: 81f8000 size: 1 flags: F_INVAL|S_WRITE
        [8062000:a782000] rw-u
as_fault pid: 987 as: ffffc10008fc6110
        addr: 81f9000 size: 1 flags: F_INVAL|S_WRITE
        [8062000:a782000] rw-u
as_fault pid: 987 as: ffffc10008fc6110
        addr: 81fa000 size: 1 flags: F_INVAL|S_WRITE
```

```
            [8062000:a782000] rw-u
    …
```

Это связано с тем, что при выделении памяти, она резервируется, но объекты страниц памяти не создаются. Однако при первом обращении на запись, происходит страничный сбой, ядро проверяет адресное пространство, и если страница не была создана ранее, создает ее.

В Linux аналогичную роль играет функция handle_mm_fault, для которой реализована обертка в виде пробы vm.pagefault:

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

Листинг 19. Скрипт pagefault.stp

```
#!/usr/bin/stap

/**
 * pagefault.stp
 *
 * Трассирует страничные сбои, обрабатываемые функцией handle_mm_fault
 *
 * Оттестировано на Linux 3.10
 */

global fault_flags;
global vma_flags;

probe begin {
    /* См. linux/mm.h */
    fault_flags[0] = "WRITE";         fault_flags[1] = "NONLINEAR";
    fault_flags[2] = "MKWRITE";       fault_flags[3] = "ALLOW_RETRY";
    fault_flags[4] = "RETRY_NOWAIT"; fault_flags[5] = "KILLABLE";

    vma_flags[0] = "VM_GROWSDOWN";      vma_flags[2] = "VM_PFNMAP";
    vma_flags[3] = "VM_DENYWRITE";      vma_flags[5] = "VM_LOCKED";
    vma_flags[6] = "VM_IO";             vma_flags[7] = "VM_SEQ_READ";
    vma_flags[8] = "VM_RAND_READ";      vma_flags[9] = "VM_DONTCOPY";
    vma_flags[10] = "VM_DONTEXPAND";    vma_flags[12] = "VM_ACCOUNT";
    vma_flags[13] = "VM_NORESERVE";     vma_flags[14] = "VM_HUGETLB";
    vma_flags[15] = "VM_NONLINEAR";     vma_flags[16] = "VM_ARCH_1";
    vma_flags[18] = "VM_DONTDUMP";      vma_flags[20] = "VM_MIXEDMAP";
    vma_flags[21] = "VM_HUGEPAGE";      vma_flags[22] = "VM_NOHUGEPAGE";
}

function prot_str:string(prot: long) {
    return sprintf("%s%s%s%s",
            (prot & 0x1) ? "r" : "-",
            (prot & 0x2) ? "w" : "-",
            (prot & 0x4) ? "x" : "-",
            (prot & 0x8) ? "s" : "-");
}

function vma_flags_str:string(flags: long) {
    prot = flags & 0xf;
```

```
    mprot = (flags >> 4) & 0xf;
    flags = flags >> 8;

    for(i = 0; i < 23; ++i) {
        if(flags & 1) {
            str = sprintf("%s|%s", str, vma_flags[i]);
        }

        flags >>= 1;
    }

    return sprintf("prot: %s may: %s flags: %s",
                   prot_str(prot), prot_str(mprot),
                   substr(str, 1, strlen(str) - 1));
}

function fault_flags_str:string(flags: long) {
    for(i = 0; i < 6; ++i) {
        if(flags & 1) {
            str = sprintf("%s|%s", str, fault_flags[i]);
        }

        flags >>= 1;
    }

    /* Вырезаем первый знак '|' */
    return substr(str, 1, strlen(str) - 1);
}

function vm_fault_str(fault_type: long) {
    if(vm_fault_contains(fault_type, VM_FAULT_OOM))
        return "OOM";
    else if(vm_fault_contains(fault_type, VM_FAULT_SIGBUS))
        return "SIGBUS";
    else if(vm_fault_contains(fault_type, VM_FAULT_MINOR))
        return "MINOR";
    else if(vm_fault_contains(fault_type, VM_FAULT_MAJOR))
        return "MAJOR";
    else if(vm_fault_contains(fault_type, VM_FAULT_NOPAGE))
        return "NOPAGE";
    else if(vm_fault_contains(fault_type, VM_FAULT_LOCKED))
        return "LOCKED";
    else if(vm_fault_contains(fault_type, VM_FAULT_ERROR))
        return "ERROR";

    return "???";
}

probe vm.pagefault {
    printf("vm.pagefault pid: %d mm: %p\n", pid(), $mm);
    printf("\taddr: %p flags: %s\n", $address, fault_flags_str($flags));
    printf("\tVMA [%p:%p]\n", $vma->vm_start, $vma->vm_end);
    printf("\t%s\n", vma_flags_str($vma->vm_flags));
    printf("\tamp: %p\n", $vma->anon_vma)

    if($vma->vm_file != 0)
        printf("\tfile: %s\n", d_name($vma->vm_file->f_path->dentry))
}

probe vm.pagefault.return {
    printf("\t => pid: %d pf: %s\n", pid(), vm_fault_str(fault_type));
```

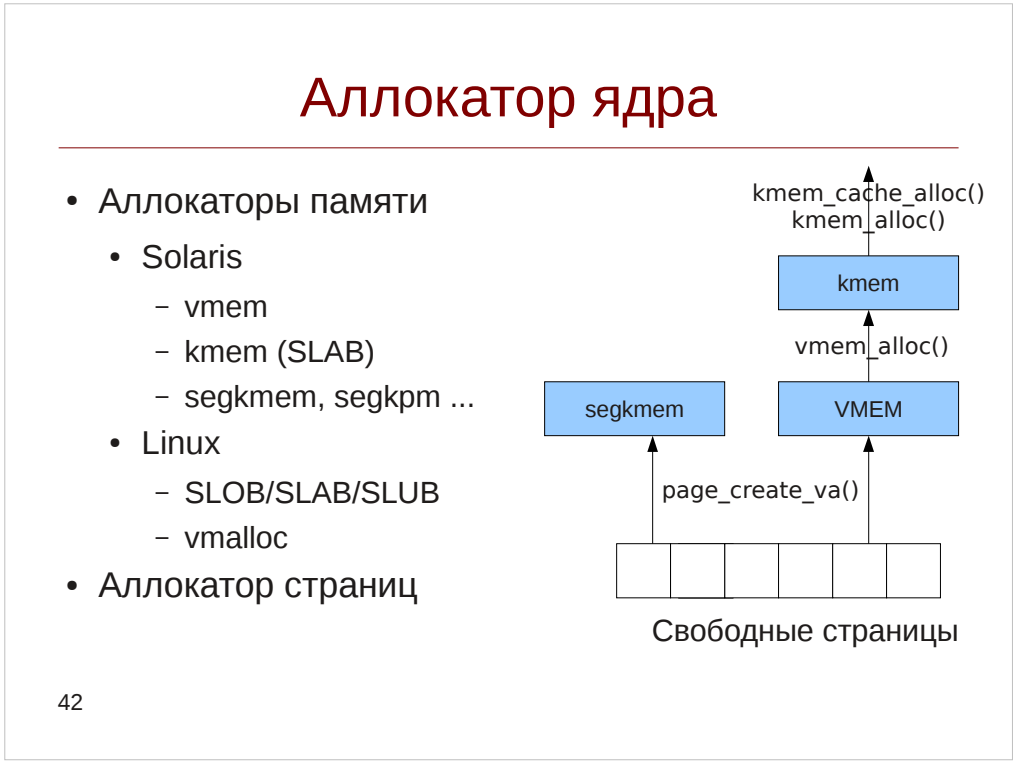
```
}
```

Вывод данного скрипта аналогичен:

```
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
```

Замечание: флаги vma_flags сильно изменчивы от версии к версии. В приведенном скрипте использовались флаги для ядра, поставляемого с CentOS 7.0. Сверяйтесь с исходным кодом ядра в файле include/linux/mm.h.

Аллокатор ядра



Внутри самого ядра распределением памяти занимается специальная подсистема, которая носит название *аллокатор ядра*. Память потребляется как конечными приложениями, так и внутри самого ядра для различных буферов и управляющих структур.

Нижний уровень аллокатора — аллокатор страниц, выделяющий физическую память из пула свободных страниц. На Solaris выделение страниц осуществляется через функцию page_create_va(), а непосредственную трассировку можно выполнять с помощью статичных проб page-get и page-get-page:

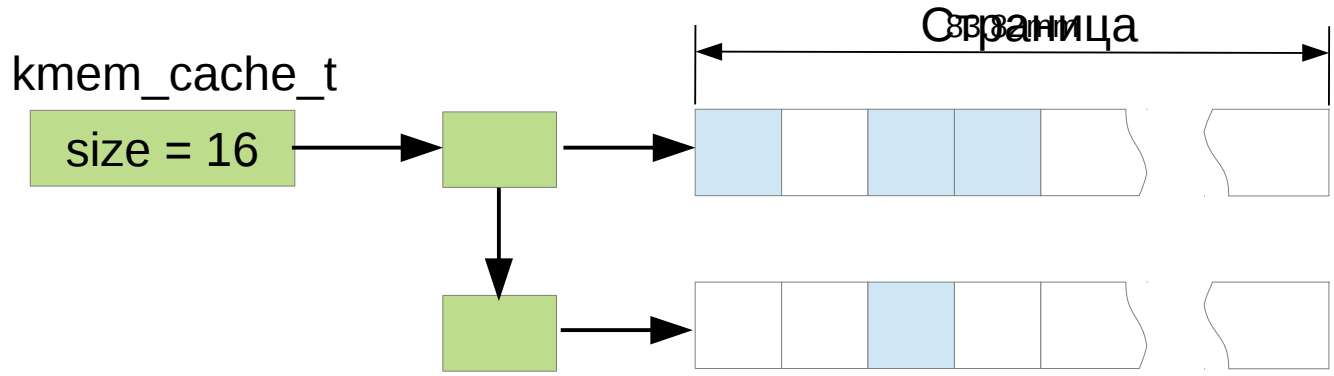
```
# dtrace -qn '
     page-get* {
          printf("PAGE lgrp: %p mnode: %d bin: %x flags: %x\n",
               arg0, arg1, arg2, arg3);
     }'
```

Замечание: вся информация о Solaris дается для традиционного аллокатора. В Solaris 11.1 появилась поддержка аллокатора VM2, однако его закрытость затрудняет его описание в рамках данного курса.

В Linux интерфейс выделение страниц производится через набор функций alloc_pages* (и сопуствующую __get_free_pages). Трассировку выделения страниц можно выполнять установив пробу в точку трассировки mm_page_alloc:

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

Естественно, что для большинства объектов в ядре гранулярность в одну страницу слишком большая. Так как большинство управляющих структур имеют стабильный размер, а также выделяется большое количество объектов одного типа, использование кучи предназначенной для общего случая — выделение разнородных кусков памяти — не выгодно. Вместо этого используется специальный аллокатор, называемый слябовым. В нем вводится специальная структура (кеш), которая организовывает страницы памяти, разделяя их на блоки одинакового размера, как показано на рисунке:



Функции для выделения областей памяти произвольного размера также используют слябовый распределитель, выбирая кеш с ближайшим размером блока, например кеши size-32 в Linux и kmem_magazine_32 в Solaris. Посмотреть текущее состояние кешей можно из файла /proc/slabinfo в Linux и команды ::kmastat утилиты mdb в Solaris.

Объект	*DTrace*	*SystemTap*

Произвольный блок	*Выделение:*
	`fbt::kmem_alloc:entry`
	`fbt::kmem_zalloc:entry`
	arg0 — размер блока
	arg1 — флаги
	
	Освобождение:
	`fbt::kmem_free:entry`
	arg0 — указатель на блок
	arg1 — размер блока
		Выделение:
		`vm.kmalloc`
		`vm.kmalloc_node`
		caller_function — вызывающий
		bytes_req — требуемый размер блока
		bytes_alloc — выделенный размер блока
		gfp_flags и gfp_flags_str — флаги
		ptr — указатель на выделеный блок
		
		Освобождение:
		`vm.kfree`
		caller_function — вызывающий
		ptr — указатель на блок

Объект	*DTrace*	*SystemTap*

Блок из кеша	*Выделение:*
	`fbt::kmem_cache_alloc:entry`
	arg0 — кеш
	arg1 — флаги
		Выделение:
		`vm.kmem_cache_alloc`
		`vm.kmem_cache_alloc_node`
		см. vm.kmalloc

Объект	*DTrace*	*SystemTap*
Блок из кеша (продолжение)	*Освобождение:* `fbt::kmem_cache_free:entry` arg0 — кеш arg1 — указатель на блок	*Освобождение:* `vm.kmem_cache_free` см. vm.kfree

Существуют также интерфейсы для выделения блоков данных, превышающих размер страницы — в Linux за это отвечает подсистема vmalloc, в Solaris — подсистема vmem и ряд сегментных драйверов, таких как segkmem, segkpm и другие.

Упражнение 4

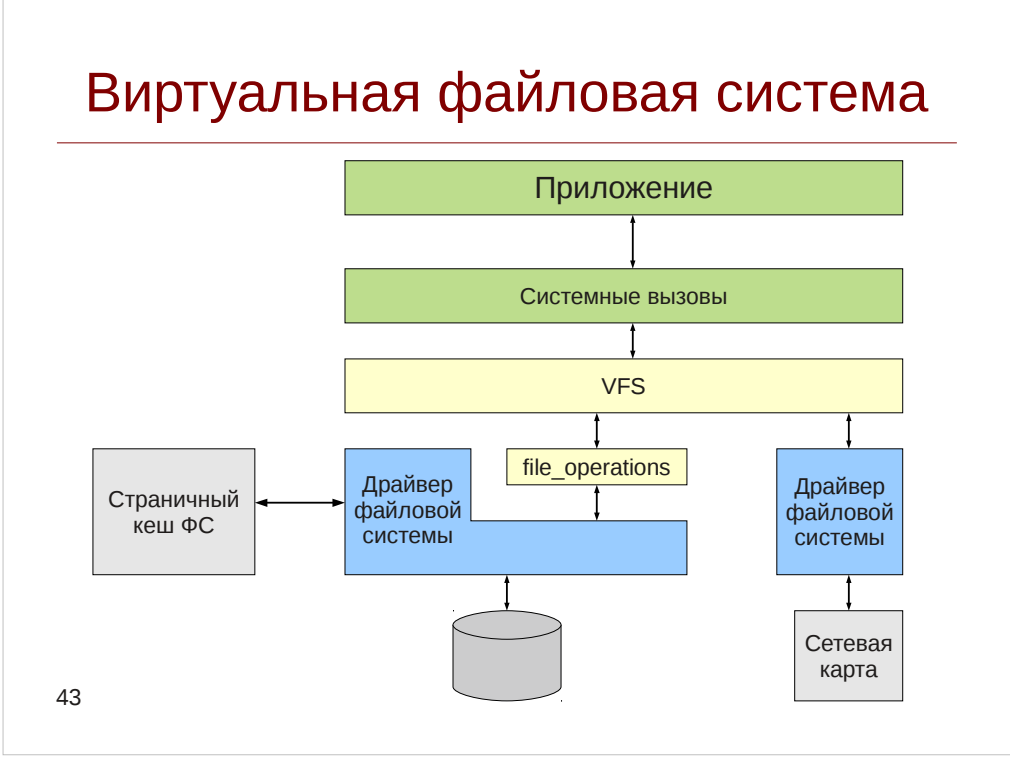
Упражнение 4.1

Переработайте скрипты pagefault.d и pagefault.stp из раздела "Страничный сбой" так, чтобы они выводили статистику по количеству страничных сбоев, группируя по имени файла, к которому они относятся. После этого запустите нагрузку proc_starter из упражнения 3. Выводить и обнулять статистику необходимо раз в секунду.

Упражнение 4.2

Реализуйте скрипты kmemstat.stp и kmemstat.d, которые собирают статистику по количеству выделений и освобождений памяти в слябовом кеше ядра. Запустите эксперимент file_opener из упражнений 1 и 2, и проанализируйте, на какие кеши приходится наибольшая нагрузка.

Виртуальная файловая система

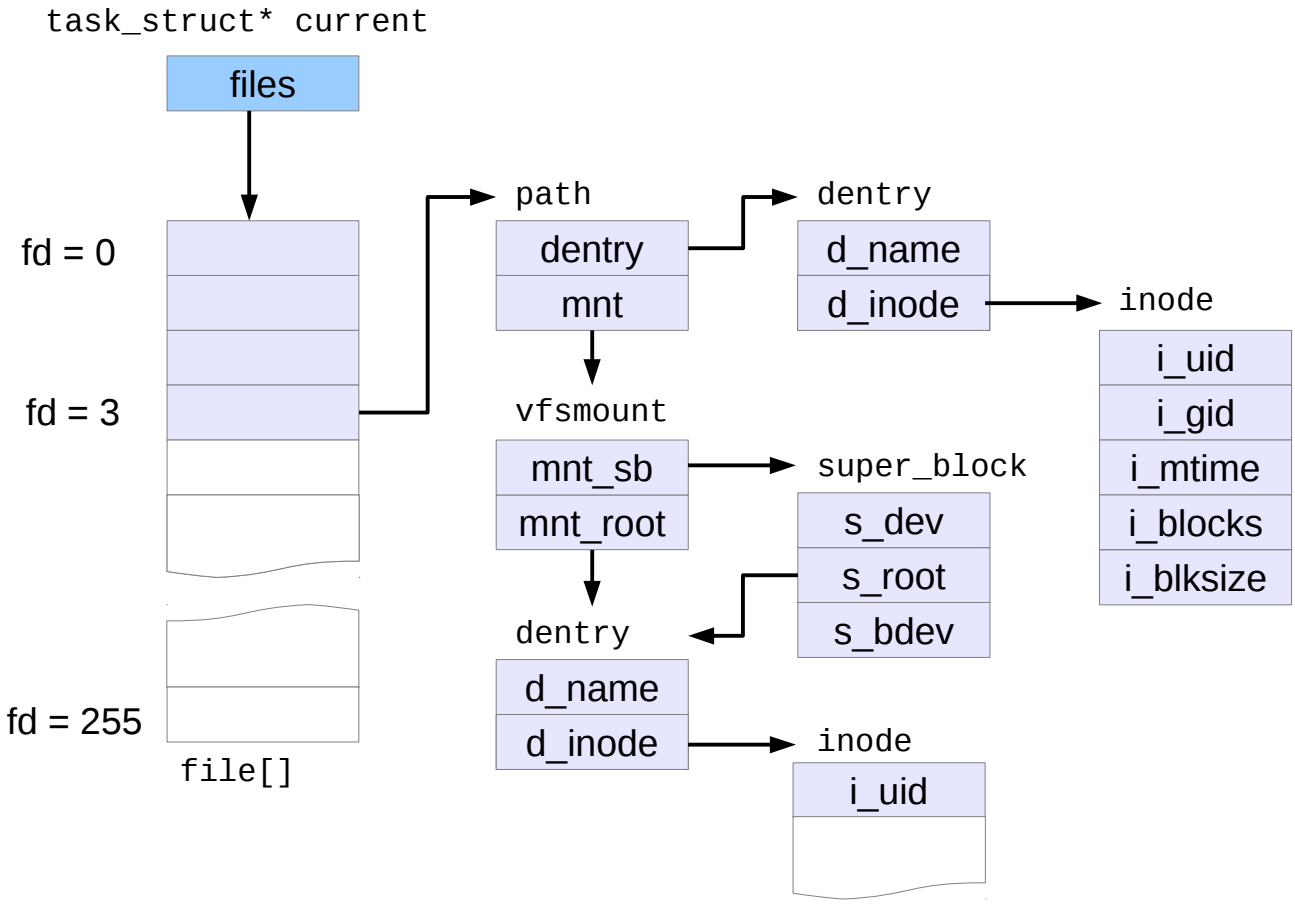


С распространением Unix-систем стала доминирующей организация данных в виде набора файлов, иерархически объединяемые в файловые системы, в конечном счете вырожденная в принцип «Все есть файл» («Everything is a file»). Однако хотя концептуально сами файловые системы похожи, их реализации сильно отличаются друг от друга. С этой проблемой столкнулись разработчики SunOS (предшественника Solaris), когда попытались увязать дисковую файловую систему UFS и сетевую NFS в одной системе. Они предусмотрели слой промежуточных интерфейсов, названный виртуальной файловой системой (VFS), который также используется и в Linux.

Для доступа к интерфейсам VFS приложение использует набор стандартных системных вызовов, таких как open() и close(), read() и write(), lseek(), и др. Приложение идентифицирует файл по специальному номеру, называемому файловым дескриптором, который по сути является индексом в таблице открытых файлов. Внутри слоя VFS используются управляющие структуры для файловой системы и ее узла: директории, файла и т. п, как указано в таблице:

	Solaris	*Linux*
Открытый файл	uf_entry_t — запись в поле p_user file	file
Файловая система	vfs_t	vfsmount super_block
Узел файловой системы	vnode_t	dentry inode

Организация управляющих структур в операционной системе Linux представлена на рисунке:



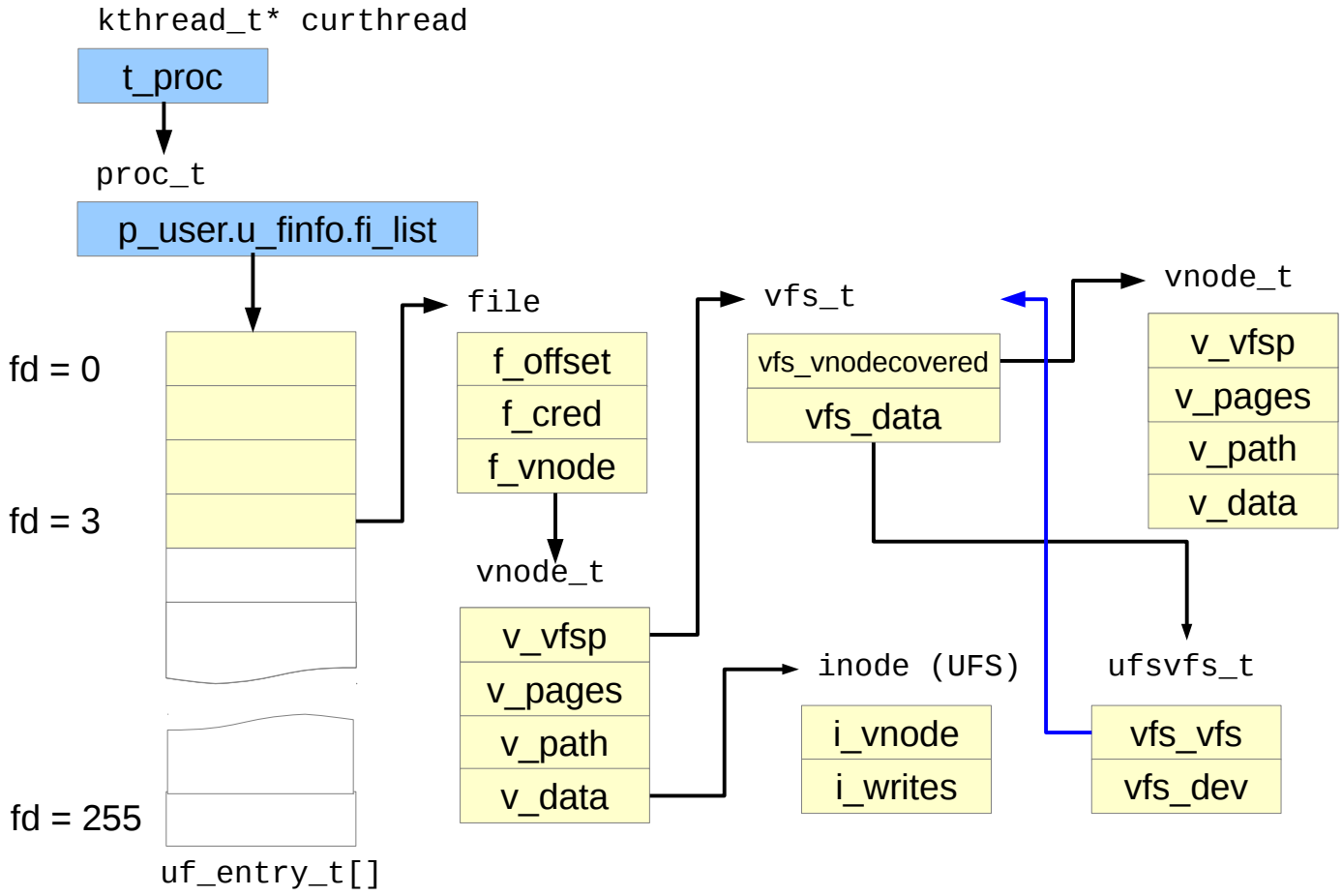
На таблицу открытых файлов указывает поле files структуры task_struct. Мы говорили о ней когда говорили об организации управления процессами в Linux. У открытого файла есть поле f_path, которое указывает на структуру path — путь до файла, включающую указатель на структура dentry — часть имени файла и vfsmount — описывающий всю файловую систему. Каждая файловая система описывается структурой super_block которая содержит указатель на корневую dentry и различные сведения о файловой системе — например указатель на блочное устройство, на котором она размещается s_bdev.

Таблица открытых файлов доступна в поле files структуры задачи task_struct. Например, для системного вызова read(), у которого первый аргумент — fd — номер открытого файлового дескриптора, и обратившись по индексу к массиву открытых файлов можно получить информацию об этом файле например так:

```
# stap -e '
     probe syscall.read {
          f = @cast(task_current(), "task_struct")->files->
               fdt->fd[fd];
          if(!f)
               next;
          d = @cast(f, "struct file")->f_path->dentry;
          printf("READ %d '%s'\n", fd, d_name(d));
     } '  -c 'cat /etc/passwd > /dev/null'
```

Замечание: массив fdt защищается специальным механизмом синхронизации read-copy-update, используемом в ядре Linux, поэтому формально мы должны захватить/освободить соответствующую блокировку, как это делают разработчики скрипта pfiles.stp.

Организация управляющих структур в операционной системе Solaris представлена на рисунке:



Как и в Linux, в Solaris существуют структуры данных, представляющие отдельный файл — это vnode_t на уровне VFS и специфичное для файловой системы представление, например для UFS это структура типа inode, а для целой файловой системы — структура vfs_t и специфичная для драйвера структура.

Для обращения к свойствам файла в таблице открытых файлов в DTrace предусмотрен специальный транслятор — массив fds, предоставляющий записи fileinfo_t:

```
# dtrace -n '
     syscall::read:entry {
          trace(fds[arg0].fi_name);
     }' -c 'cat /etc/passwd > /dev/null'
```

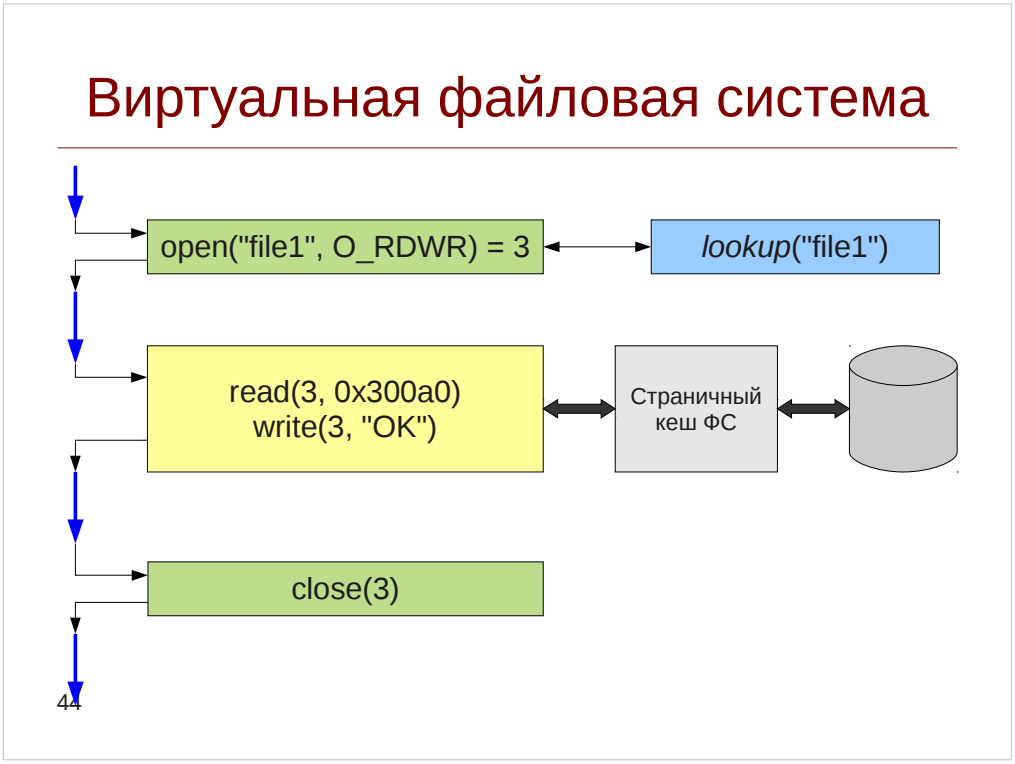
Однако его возможности ограничены и для доступа к соотвествующей vnode, нам потребуется обращаться к структурам данных, рассмотренным выше:

```
# dtrace -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;
     trace(stringof(this->vn->v_path));
}' -c 'cat /etc/passwd > /dev/null'
```

Для трассировки действий пользователя и приложений на файловой системе можно привязываться к системным вызовам. На уровне VFS это можно осуществить следующим образом:

- В Solaris/DTrace используются промежуточные функции fop_*, которые осуществляют предобработку данных и вызывают драйвер соответствующей файловой системы через таблицу операций vnodeops:
  ```
  # dtrace -n '
       fop_mkdir:entry {
            trace(stringof(args[1]));
       }' -c 'mkdir /tmp/test1'
  ```
- В Solaris/DTrace реализован провайдер fsinfo — набор статических проб, реализованных внутри функций fop_*. Как и массив fds предоставляет структуру fileinfo_t в args[0]
  ```
  # dtrace -n '
       fsinfo:::mkdir {
            trace(args[0]->fi_pathname); trace(args[0]->fi_mount);
       }' -c 'mkdir /tmp/test2'
  ```
  При запуске этого скрипта fi_pathname будет содержать строку "<unknown>", так как на момент срабатывания пробы поле v_path еще не заполнено.
- В Linux/SystemTap можно использовать некоторые из функций vfs_*, для мониторинга действий на файловой системе. Кроме этого можно использовать систему fsnotify, который позволяет отслеживать изменения в файловой системе, (если она соответственно сконфигурирована в системе):
  ```
  # stap -e '
       probe kernel.function("fsnotify") {
            if(!($mask == 0x40000100))
                 next;
            println(kernel_string2($file_name, "???"));
       } ' -c 'mkdir /tmp/test3'
  ```
  Здесь значение 0x40000100 — это маска флагов FS_CREATE | FS_ISDIR.



Чтобы открыть файл, приложение инициирует системный вызов open(), при этом операционная система должна по указанному имени файла отобразить его имя на соответствующую ему структуру vnode_t или inode — это делает условная операция lookup на нашем рисунке. На Solaris этак функция реализована через функцию lookuppnvp, однако вся ответственность по кешированию результатов этой операции лежит на драйвере файловой системы, к которой делается вызов fop_lookup:

```
# dtrace -n '
     lookuppnvp:entry /execname == "cat"/ {
          trace(stringof(args[0]->pn_path));
     }
     fop_lookup:entry /execname == "cat"/ {
          trace(stringof(arg1));
     } ' -c 'cat /var/log/messages > /dev/null'
```

В Linux за поиск соответствий имен и соответствующих им структур inode отвечает подсистема Directory Entry Cache, а поиск выполняется функцией d_lookup и ее низкоуровневыми реализациями:

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

Самые важные операции в стеке VFS — это чтение и запись файлов, так как они наиболее часто порождают низкоуровневые операции ввода-вывода и соответственно интересны с точки зрения анализа производительности. В Solaris и Linux все дисковые операции происходят через страничный кеш, аналогично операциям mmap, с той лишь разницей, что вместо того, чтобы напрямую

отображать прочитанную с диска страницу в адресное пространство процесса, данные копируются из нее в буфер, предоставленный пользователем в операции read() и из него в страницу памяти в операции write(). Исключение составляет лишь файловая система ZFS, использующая собственный механизм кеширования (ARC-кеш), который в свою очередь работает поверх аллокатора ядра.

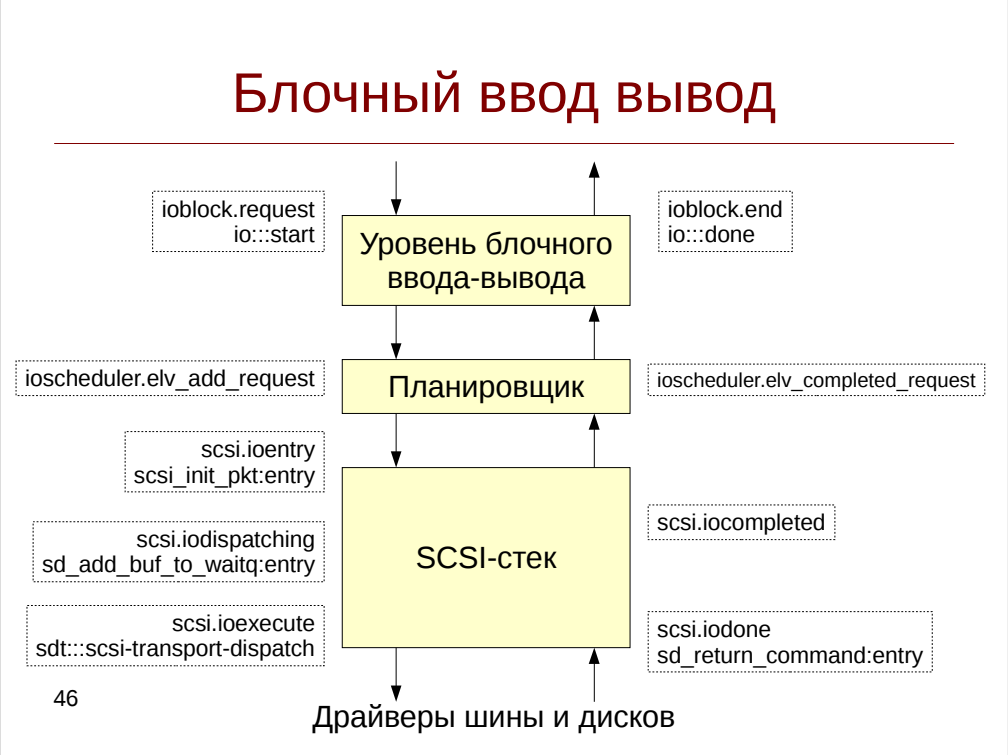
Последовательность действий для операций чтения выглядит следующим образом:

Действие	*Solaris*	*Linux*
Приложение инициирует вызов read()	read()	sys_read()
Вызов обрабатывается стеком VFS	fop_read()	vfs_read()
Вызов передается драйверу файловой системы	v_ops->vop_read	file->f_op->read() или do_sync_read()
Для прямого ввода-вывода (directIO) вызывается соответств. функция Выход.	Например ufs_directio_read()	a_ops->direct_IO
Страница, соответствующая файлу ищется в страничном кеше Если найдена — выход.	vpm_data_copy() или segmap_getmap_flt()	file_get_page()
Если страница не найдена, она запрашивается у файловой системы	v_ops ->vop_getpage()	a_ops->readpage()
Создается запрос к подсистеме дискового ввода-вывода	bdev_strategy()	submit_bio()
Выход.		

Замечание: разумеется, такая таблица крайне упрощена и не включает например особенности работы с журналируемыми файловыми системами.

Как видим, каждый из драйверов файловых систем реализуют свою таблицу операций для работы с файлами, которую использует высокоуровневый код VFS: в Solaris это таблица vnodeops_t, включающая в себя как операции чтения и записи (vop_read()/vop_write()) так и работы со страницами (vop_getpage()/vop_putpage()). В Linux она распадается на две таблицы — одна из них address_space_operations (в таблице указана как a_ops по названию поля в структуре file), предназначенная для работы на уровне страниц, а вторая — для непосредственной реализации операций VFS — file_operations и inode_operations. К тому же Linux предоставляет ряд универсальных операций, их можно отслеживать в tapset'e generic.

Блочный ввод-вывод



После того, как запрос обрабатывается VFS и если он требует совершения дискового ввода-вывода, создается запрос к подсистеме блочного ввода-вывода: операционная система или запрашивает страницу памяти с диска при чтении или записывает ее часть при записи на диск. Блочными такие устройства называются из-за того, что оперировать с ними можно блоками фиксированной длины — как правило это значения, кратные 512 байтам — размеру сектора диска. В отличие от них, символьные устройства (терминалы, клавиатуры) передают данные посимвольно, сетевые же устройства имеют произвольную длину пакета данных.

Трассировка блочного ввода-вывода выполняется в Solaris с помощью провайдера io, например для запросов на чтение:

```
# 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"
```

В SystemTap тем же целям служит tapset ioblock:

```
# stap -e '
     probe ioblock.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"
```

Запросу в подсистеме ввода-вывода соответствуют структуры buf в Solaris (в

пробах провайдера io используется транслятор bufinfo_t) и bio в Linux:

Поле	*bufinfo_t или struct buf*	*struct bio*
Флаги операции	b_flags	bi_flags
Чтение или запись	флаги B_WRITE, B_READ в b_flags	bi_rw, см. также bio_rw_num() и bio_rw_str()
Количество байт	b_bcount	bi_size
Номер блока	b_blkno, b_lblkno	bi_sector
Метод завершения операции	b_iodone	bi_end_io
Дескриптор устройства	b_edev, b_dip	bi_bdev
Указатель на данные	b_addr или b_pages (только struct buf) для B_PAGEIO	см. замечание 1
Указатель на дескриптор файла	b_file (только struct buf)	см. замечание 2

⚠ *Замечание 1*: Структура bio использует таблицу bi_io_vec, каждый элемент которой содержит указатель на страницу bv_page, длину bv_len и смещение внутри страницы bv_offset. Количество структур в таком векторе определяется полем bi_vcnt, текущий индекс полем bi_idx.

⚠ *Замечание 2*. Каждый bio может содержать множество связанных с ними файлов (например после склеивания нескольких операций планировщиком ввода-вывода). Можно получить доступ к inode через поле mapping страницы памяти bv_page, используемой в bi_io_vec, как это сделано в фукции __bio_ino (см. /usr/share/systemtap/tapset/ioblock.stp)

⚠ *Замечание 3*: Кроме указанных полей дескриптора устройства и файла, в пробах провайдера io предоставляются трансляторы fileinfo_t в args[1] и devinfo_t в args[2].

После того, как запрос создан, он передается планировщику, который переупорядочивает запросы так, чтобы доступ к блокам не требовал значительных перемещений головок дисков. В Linux реализовано несколько планировщиков, включая Deadline и CFQ — эта подсистема играет важную роль во всей системе блочного ввода-вывода, а его трассировка может выполняться посредством tapset ioscheduler. В Solaris отдельного планировщика нет: ZFS использует собственные механизмы для этого, а единственный глобальный механизм планирования — лифтовой планировщик, реализованный в функции sd_add_buf_to_waitq().

Уровень блочного ввода-вывода передает запрос драйверу дискового устройства. Чтобы унифицировать этот уровень, разработчики используют за основу протокол SCSI (более того, в ядре Linux команды к устройству сначала транслируются в формат команды SCSI, а затем в ATA). Общение с устройством происходит посредством SCSI-команд, обернутых в соответствующие пакеты

данных:

Действие	*Solaris*	*Linux*
Создается новый экземпляр SCSI-пакета	scsi_init_pkt()	scsi.ioentry
SCSI-пакет помещается в очередь запросов (диспетчеризуется)	sd_add_buf_to_waitq()	scsi.iodispatching
SCSI-пакет передается низкоуровневому драйверу	sdt::scsi-transport-dispatch scsi_transport()	scsi.ioexecute (опционально)
Выполнение команды завершено	sd_return_command()	scsi.iocompleted scsi.iodone


Замечание 1. Проба scsi.ioexecute не обязана срабатывать на все SCSI-команды: обычно в процессе диспетчеризации драйвер шины/контроллера помещает запрос в свою внутреннюю очередь и обрабатывает его независимо от SCSI-стека.


Замечание 2. В данном случае для Solaris приведены функции драйвера дисков sd. Драйвер для оптических дисков ssd собирается из тех же самых исходных кодов, поэтому достаточно лишь заменить sd на ssd в названиях функций.

Листинг 20. Скрипт sdtrace.d

```
#!/usr/sbin/dtrace -qCs

#pragma D option nspec=512

int specs[uint64_t];
uint64_t timestamps[uint64_t];

#define BUF_SPEC_INDEX(bp)                                          \
    ((uint64_t) bp) ^ ((struct buf*) bp)->_b_blkno._f
#define BUF_SPECULATE(bp)                                           \
    speculate(specs[BUF_SPEC_INDEX(bp)])

#define PROBE_PRINT(probe, bp)                                      \
    printf("%-24s %p cpu%d %+llu\n", probe, bp, cpu,                \
           (unsigned long long) (timestamp -                        \
                                 timestamps[BUF_SPEC_INDEX(bp)]))

#define PROC_PRINT()                                                \
    printf("\tPROC: %d/%d %s\n", pid, tid, execname);

#define BUF_PRINT_INFO(buf)                                         \
    printf("\tBUF flags: %s %x count: %d blkno: %d comp: ",        \
           (buf->b_flags & B_WRITE)? "W" : "R",  buf->b_flags,     \
           buf->b_bcount, buf->b_blkno);                            \
           sym((uintptr_t) buf->b_iodone); printf("\n")

#define DEV_PRINT_INFO(dev)                                         \
    printf("\tDEV %d,%d %s\n", dev->dev_major, dev->dev_minor,     \
           dev->dev_name);

#define FILE_PRINT_INFO(file)                                       \
```

```
    printf("\tFILE %s+%d\n", file->fi_pathname, file->fi_offset);

#define PTR_TO_SCSIPKT(pkt) ((struct scsi_pkt*) pkt)
#define SCSIPKT_TO_BP(pkt)  ((struct buf*) PTR_TO_SCSIPKT(pkt)->pkt_private)

#define SCSIPKT_PRINT_INFO(pkt)                                     \
    printf("\tSCSI PKT flags: %x state: %x comp: ",                 \
            pkt->pkt_flags, pkt->pkt_state);                        \
            sym((uintptr_t) pkt->pkt_comp); printf("\n")

io:::start {
    specs[BUF_SPEC_INDEX(arg0)] = speculation();
    timestamps[BUF_SPEC_INDEX(arg0)] = timestamp;
}

io:::start {
    BUF_SPECULATE(arg0);

    printf("---------------------\n");
    PROBE_PRINT("io-start", arg0);
    PROC_PRINT();
    BUF_PRINT_INFO(args[0]);
    DEV_PRINT_INFO(args[1]);
    FILE_PRINT_INFO(args[2]);
}

*sd_initpkt_for_buf:entry {
    self->bp = arg0;
}

*sd_initpkt_for_buf:return
/arg1 != 0/ {
    BUF_SPECULATE(self->bp);
    PROBE_PRINT("ALLOCATION FAILED", self->bp);
}

*sd_initpkt_for_buf:return
/arg1 != 0/ {
    commit(specs[BUF_SPEC_INDEX(self->bp)]);
}

*sdstrategy:entry {
    BUF_SPECULATE(arg0);
    PROBE_PRINT(probefunc, arg0);
}

*sd_add_buf_to_waitq:entry {
    BUF_SPECULATE(arg1);
    PROBE_PRINT(probefunc, arg1);
}

scsi-transport-dispatch {
    BUF_SPECULATE(arg0);
    PROBE_PRINT(probename, arg0);
}

scsi_transport:entry {
    this->bpp = (uint64_t) SCSIPKT_TO_BP(arg0);

    BUF_SPECULATE(this->bpp);
    PROBE_PRINT(probefunc, this->bpp);
```

```
    SCSIPKT_PRINT_INFO(PTR_TO_SCSIPKT(arg0));
}

*sdintr:entry {
    self->bpp = (uint64_t) SCSIPKT_TO_BP(arg0);

    BUF_SPECULATE(self->bpp);
    PROBE_PRINT(probefunc, self->bpp);
    SCSIPKT_PRINT_INFO(PTR_TO_SCSIPKT(arg0));
}

io:::done {
    BUF_SPECULATE(arg0);
    PROBE_PRINT("io-done", arg0);
    BUF_PRINT_INFO(args[0]);
}

io:::done {
    commit(specs[BUF_SPEC_INDEX(arg0)]);
    specs[BUF_SPEC_INDEX(arg0)] = 0;
}
```

Данный скрипт сохраняет информацию о всех фазах операции блочного ввода-вывода в спекуляцию, и по завершению операции коммитит ее. Заметим, что так как спекуляции в DTrace имеют по буферу на каждый процессор, вывод может некорректно отображаться, если прерывание sdintr случилось на ином процессоре, чем был создан запрос.

Для каждого из запросов мы увидим записи следующего вида:

```
---------------------
io-start                ffffc100040c4300 cpu0 2261
        PROC: 1215/1 dd
        BUF flags: R 200061 count: 512 blkno: 0 comp:   0x0
        DEV 208,192 sd
        FILE <none>+-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
sdintr                  ffffc100040c4300 cpu0 565121
        SCSI PKT flags: 14000 state: 1f comp:   sd`sdintr
io-done                 ffffc100040c4300 cpu0 597642
        BUF flags: R 2200061 count: 512 blkno: 0 comp:   0x0
```

Каждая строчка, соответствующая фазе запроса, содержит имя этой фазы, адрес структуры buf, номер процессора и время в наносекундах с момента создания запроса. В нашем случае наибольший «провал» наблюдается между фазами scsi_transport и sdintr (~0,5 мс), что естественно, так как в это время происходит физический ввод-вывод.

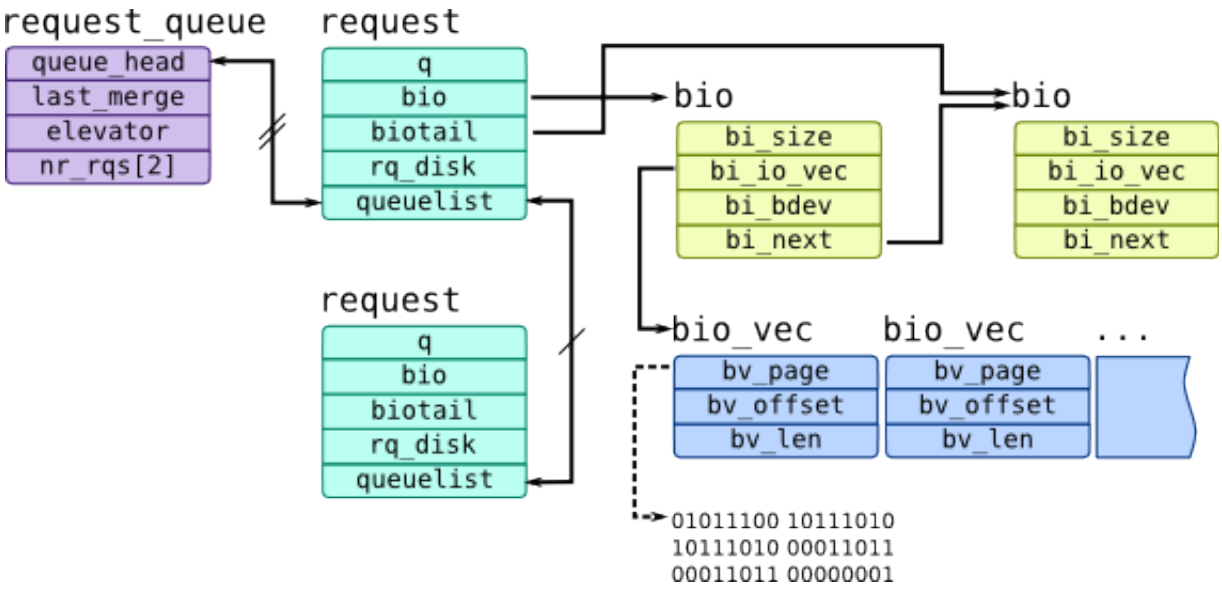
Также SCSI-стек в Solaris использует механизм подпрограмм завершения (completion routines) – вызывающая функция может сохранить указатель на эту функцию в поле pkt_comp SCSI-пакета и после завершения его обработки она будет

вызвана, как функция sdintr в данном примере. Аналогичную роль играет b_iodone в структуре buf.

SCSI-стек в Linux устроен несколько сложнее чем в Solaris, так как в нем присутствует полноценный планировщик ввода-вывода, тогда как в Solaris присутствует лишь disksort-алгоритм, а вся работа по планированию операций ввода-вывода лежит на верхних уровнях стека, в частности подсистеме VDEV файловой системы ZFS.

Для того, чтобы поддерживать инфраструктуру планировщиков, запросы в виде структуры bio оборачиваются в структуру request – см. функцию ядра blk_queue_bio(), которая или склеивает bio с последним запросом request или создает новый, используя функцию get_request(). Запросы могут склеиваться и самим планировщиком, однако мы оставим такую ситуацию за рамками нашего скрипта-трассировщика.

Таким образом, реальная организация запросов в подсистеме блочного ввода-вывода Linux выглядит следующим образом:



Листинг 21. Скрипт scsitrace.stp

```
#!/usr/bin/stap

global rqs, bio2rq, specs, times;

function probe_print:string(bio:long) {
    return sprintf("%-24s %p cpu%d %u\n", pn(), bio, cpu(),
                   gettimeofday_ns() - times[bio2rq[bio]]);
}

function rq_probe_print(rq:long, bio:long) {
    if(bio == 0)
```

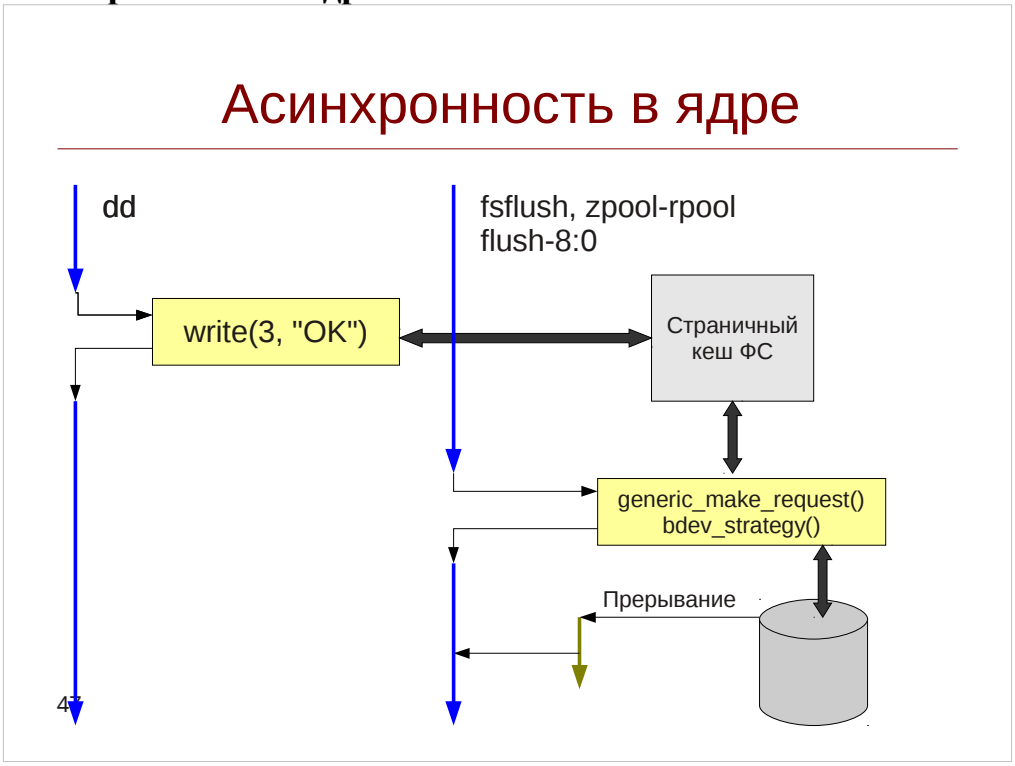
```
        bio = @cast(rq, "struct request")->bio;
    return sprintf("%-24s %p %p cpu%d %u\n", pn(), bio, rq, cpu(),
                   gettimeofday_ns() - times[bio]);
}

function proc_print:string() {
    return sprintf("\tPROC: %d/%d %s\n", pid(), tid(), execname());
}

function handle_bio2rq(bio:long, rq:long) {
    if(specs[rq] == 0) {
        specs[rq] = speculation();
    }

    rqs[rq] += 1;
    bio2rq[bio] = rq;

    speculate(specs[rq],
        rq_probe_print(rq, bio)
       .proc_print()
       .sprintf("\tBUF flags: %s %x count: %d blkno: %d comp: %s\n",
           bio_rw_str(@cast(bio, "bio")->bi_rw), @cast(bio, "bio")->bi_flags,
                @cast(bio, "bio")->bi_size, @cast(bio, "bio")->bi_sector,
                symname(@cast(bio, "bio")->bi_end_io))
       .sprintf("\tDEV %d,%d\tINO %d\n",
                MAJOR(@cast(bio, "bio")->bi_bdev->bd_dev),
                MINOR(@cast(bio, "bio")->bi_bdev->bd_dev), __bio_ino(bio)));
}

probe ioblock.request {
    times[$bio] = gettimeofday_ns();
}

probe kernel.function("bio_attempt_front_merge").return,
      kernel.function("bio_attempt_back_merge").return {
    if($return) {
        /* BIO was merged with request */
        rq = $req;
        bio = $bio;

        if(bio == 0) next;

        handle_bio2rq(bio, rq);
    }
}

probe kernel.function("get_request").return {
    rq = $return;
    bio = $bio;

    if(bio == 0) next;

    /* BIO were created a new request */
    handle_bio2rq(bio, rq);
}

probe ioscheduler.elv_add_request, ioscheduler.elv_completed_request {
    if(rq == 0 || specs[rq] == 0) next;
    speculate(specs[rq],
              rq_probe_print(rq, 0)
             .sprintf("\tDEV %d,%d\n", disk_major, disk_minor));
```

```
}

probe scsi.ioentry, scsi.iodone, scsi.iocompleted, scsi.iodispatching {
    if(req_addr == 0 || specs[req_addr] == 0) next;
    speculate(specs[req_addr],
              rq_probe_print(req_addr, 0));
}

probe scsi.iodispatching {
    if(req_addr == 0 || specs[req_addr] == 0) next;
    speculate(specs[req_addr],
              rq_probe_print(req_addr, 0)
             .sprintf("\tSCSI DEV %d:%d:%d:%d %s\n",
                      host_no, channel, lun, dev_id, device_state_str)
             .sprintf("\tSCSI PKT flags: %x comp: %s\n",
                      @cast(req_addr, "struct request")->cmd_flags,
                      symname($cmd->scsi_done)));
}

probe ioblock.end {
    bio = $bio;
    rq = bio2rq[bio];

    delete bio2rq[bio];
    delete times[bio];

    rqs[rq] -= 1;
    if(rqs[rq] == 0) {
        speculate(specs[rq], probe_print(bio));
        speculate(specs[rq], "----------\n");
        commit(specs[rq]);

        delete specs[rq];
    }
}
```

Вывод скрипта показан ниже:

```
----------
kernel.function("get_request@block/blk-core.c:1074").return 0xffff880039ff1500
0xffff88001d8fea00 cpu0 4490
        PROC: 16668/16674 tsexperiment
        BUF flags: R f000000000000001 count: 4096 blkno: 779728 comp:
end_bio_bh_io_sync
        DEV 8,0 INO 0
ioscheduler.elv_add_request 0xffff880039ff1500 0xffff88001d8fea00 cpu0 15830
        DEV 8,0
scsi.ioentry            0xffff880039ff1500 0xffff88001d8fea00 cpu0 19847
scsi.iodispatching      0xffff880039ff1500 0xffff88001d8fea00 cpu0 25744
        SCSI DEV 2:0:0:0 RUNNING
        SCSI PKT flags: 122c8000 comp: 0x0
scsi.iodispatching      0xffff880039ff1500 0xffff88001d8fea00 cpu0 29882
scsi.iodone             0xffff880039ff1500 0xffff88001d8fea00 cpu1 4368018
scsi.iocompleted        0xffff880039ff1500 0xffff88001d8fea00 cpu0 4458073
ioblock.end             0xffff880039ff1500 cpu0 1431980041275998676
```

Асинхронность в ядре



Вернемся к нашим скриптам sdtrace.d и scsitrace.stp. В строчке, описывающей имя и pid процесса, совершающего операцию ввода-вывода мы увидим интересную информацию:

```
PROC: 5/15 zpool-rpool
```

или

```
PROC: 643/643 flush-8:0
```

Этот процесс является неким внутрисистемным процессом, и к процессу, непосредственно инициировавшим ввод-вывод (например, обратившись к файловой системе через системный вызов write, как в нашем примере) не имеет отношения.

Это вызвано механизмами асинхронности, повсеместно применяющемся в ядре: вместо того, чтобы непосредственно выполнить пользовательский запрос, ядро помещает его в специальную очередь, а обработкой заданий в этой очереди занимается отдельный процесс: например, вызов write() для файлов, при открытии которых не использовался флаг O_SYNC, всего лишь обновляет данные в страничном кеше и помечает страницу как «грязную». Один из процессов механизма writeback в Linux или процесс fsflush в Solaris периодически просыпаются и обходят грязные страницы, записывая данные из них на диск. Кроме того, дисковый контроллер генерирует прерывания, чтобы уведомить эти процессы о завершении дискового ввода-вывода. Таким образом, при обработке запроса мы имеем как минимум три контекста, что мешает использовать Thread-Clause переменные в DTrace, и вообще оценивать полное время обработки запроса.

Чтобы избежать этого, нужно ловить запросы на этапе их создания и помещения в очередь пользовательским процессом, и сохранить необходимые данные (например, pid процесса) в ассоциативном массиве, а в качестве ключа можно использовать адрес управляющей структуры, например адрес структуры

page страничного кеша:

```
# 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());
     } '
…
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
```

Как видно из вывода этого небольшого скрипта, реальный ввод-вывод выполняется потоком ядра #665. Точно также для файловой системы ZFS в Solaris можно воспользоваться указателем на dbuf:

```
# 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);
    } '
```

О некоторых механизмах обеспечения асинхронности в ядрах мы поговорим позднее.

Упражнение 5

Задание 1

Напишите скрипты deblock.stp и deblock.d, которые бы демонстрировали эффекты невыровненного ввода-вывода (деблокирования) при синхронной записи. Для этого собирайте с помощью агрегаций общий объем переданных данных на уровне виртуальной файловой системы (VFS) и блочного ввода-вывода (BIO) и выводите результат по таймеру. Данные должны группироваться по имени дискового устройства и/или точке монтирования файловой системы.

Создайте файловую систему на тестовом LUN'e, например ext4 в CentOS 7:

```
# mkdir /tiger
# mkfs.ext4 /dev/sda
# mount /dev/sda /tiger
```

и ZFS в Solaris:

```
# zpool create tiger c3t0d0
```

Замечание 1. В данном примере тестовое дисковое устройство имеет имя /dev/sda в Linux и c3t0d0 в Solaris. Замените их на имена в вашем лабораторном окружении. Точка монтирования /tiger используется в экспериментах.

Замечание 2. Вы можете использовать и другие файловые системы, однако комментарии в конце книги будут приведены для ext4 и ZFS.

Запустите эксперимент deblock/ для демонстрации эффекта деблокирования. В этом примере создается файл /tiger/DEBLOCK размером в 1 Мб и осуществляется случайная синхронная запись с размером блока от 512 до 16384 байт (дискретно равномерно распределено). Файловая система вынуждена «округлять» размер блока, передаваемого нагрузчиком, до кратного размеру блока самой файловой системы, что и составляет накладные расходы. Это также может приводить к дополнительным операциям чтения.

Чтобы нивелировать эффекты деблокирования, установите фиксированный размер блока, например поменяв параметры вариатора так:

```
# /opt/tsload/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
```

Перезапустите скрипт и посмотрите, как изменились характеристики.

Задание 2

Во втором задании нам предстоит познакомиться с предвыборкой или упреждающим чтением (readahead или prefetch, соответственно). Эта техника значительно улучшает производительность при последовательном чтении, так как при чтении блока она также пытается прочитать последующие блоки и поместить в страничный кеш.

Напишите скрипты readahead.stp и readahead.d, чтобы выяснить какой слой ответственен за предвыборку – для этого соберите количество операций, выполняемых на уровнях VFS, BIO и SCSI. Требования к группировке и выводу данных такие же как и в задании 1.

Так как после создания тестового файла, он уже окажется в страничном кеше, нам придется сбрасывать его перед каждым экспериментом. Чтобы сделать это отмонтируйте и смонтируйте снова файловую систему:

```
# umount /tiger/ ; mount /dev/sda /tiger/
```

В ZFS для этого придется экспортировать и импортировать пул:

```
# zpool export tiger ; zpool import tiger
```

Можно также воспользоваться опцией drop_caches в Linux.

Так как модуль SimpleIO начинает эксперимент сразу после создания файла, то сбросить кеш не получится. Чтобы избежать этого, мы создадим файл предварительно, например с помощью dd:

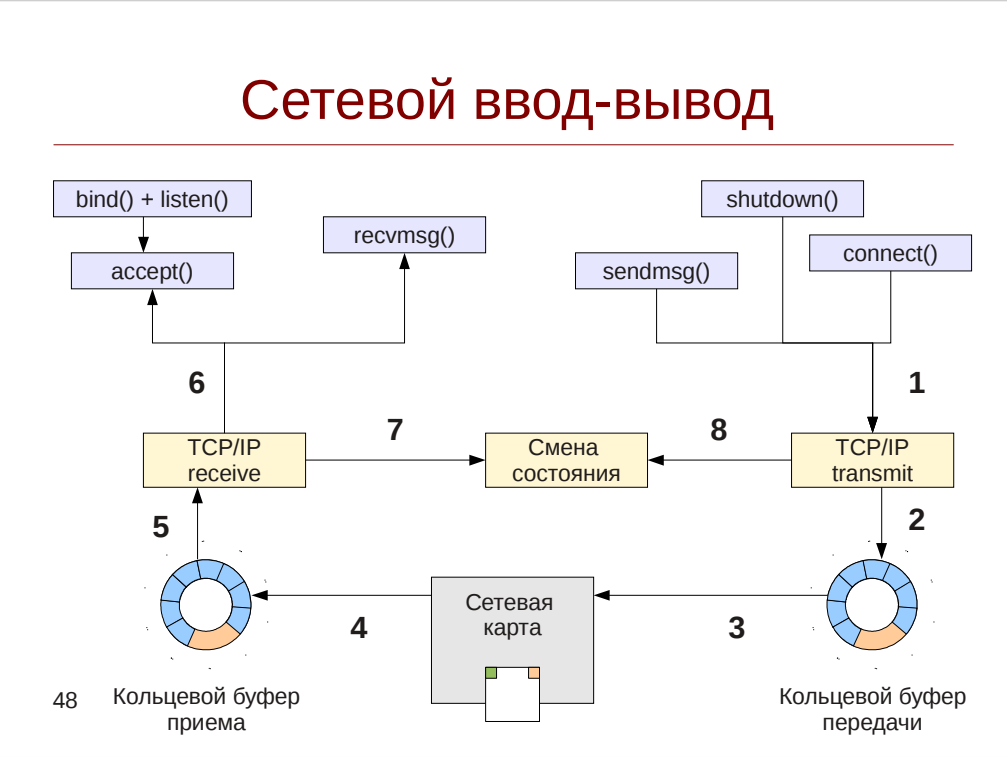
```
# dd if=/dev/zero of=/tiger/READAHEAD count=40960
```

И укажем опцию overwrite в эксперименте.

Чтобы разрушить эффекты упреждающего чтения, замените последовательное чтение на случайное:

```
# /opt/tsload/bin/tsexperiment -e readahead/experiment.json run \
    -s workloads:fileio:params:offset:randgen:class=lcg
```

Сетевой ввод-вывод



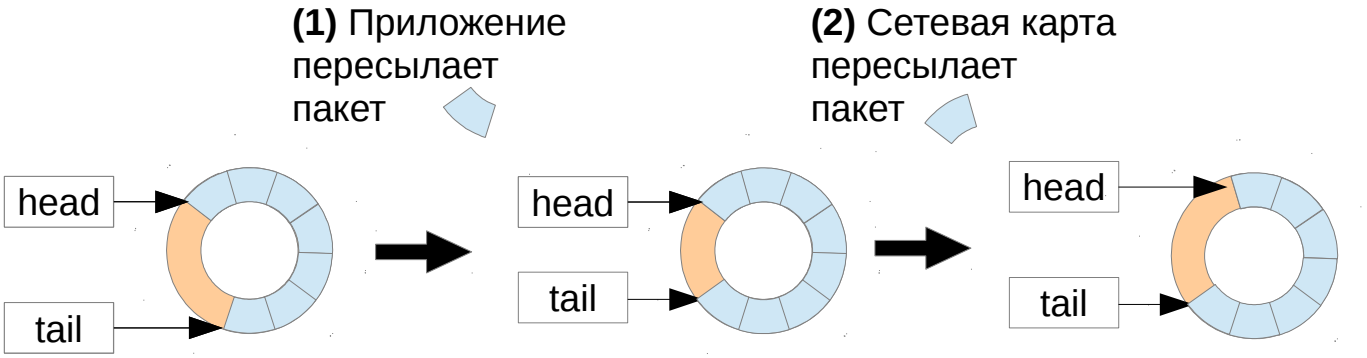
Последняя крупная подсистема ядра, которую мы рассмотрим — сетевой стек. Называется он так из-за иерархичности протоколов, предусмотренной моделью OSI: в процессе пересылки пакета в сеть, он оборачивается в заголовки и концевики, соответствующие протоколу более низкого уровня (этот процесс носит название инкапсуляции), а при приеме, заголовки и концевики анализируются и отбрасываются, так что приложение на узле-приемнике (хосте) получает исходный пакет. Такими протоколами являются TCP или UDP, а также IP, лежащий ниже них — и реализованы они в виде драйверов ядра. Заметим, что в отличие от дискового ввода-вывода, где запись может быть отложена — в сетевом вводе-выводе, который должен обеспечивать наименьшую задержку при пересылке данных, это неприемлимо — и как правило вызов функций пересылки и приемки пакетов происходит в едином контексте.

В сетевом вводе-выводе Unix можно выделить три главных архитектурных слоя:

- Уровень сокетов BSD, предоставляющий интерфейс для работы с сетевыми соединениями в виде набора системных вызовов.
- Промежуточные драйвера протоколов, включая пакетные фильтры: ip, tcp, udp, и так далее.
- На нижнем уровне — слой доступа к среде передачи MAC и архитектура драйверов сетевых карт, например в Solaris она носит название GLD.

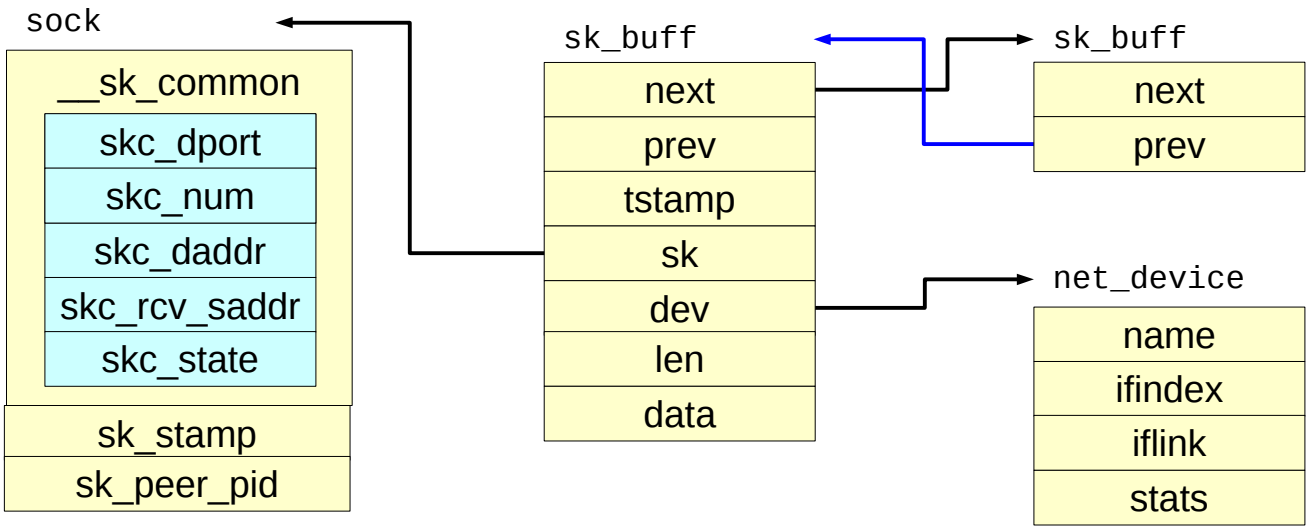
Так как объемы передаваемых данных огромны для передачи пакета сетевой карте используется DMA (Direct Memory Access) — техника, при которой устройство самостоятельно забирает данные из памяти без участия центрального

процессора. Для обмена с сетевой картой используются *кольцевые буферы* (ring



(1) Когда драйвер хочет передать пакет в среду, он записывает его в область памяти, отведенную для концевого буфера, в его хвост (tail) и обновляет указатель хвоста. **(2)** Когда сетевая карта начинает передачу данных, она извлекает пакет из головы (head) и обновляет соответствующий указатель.

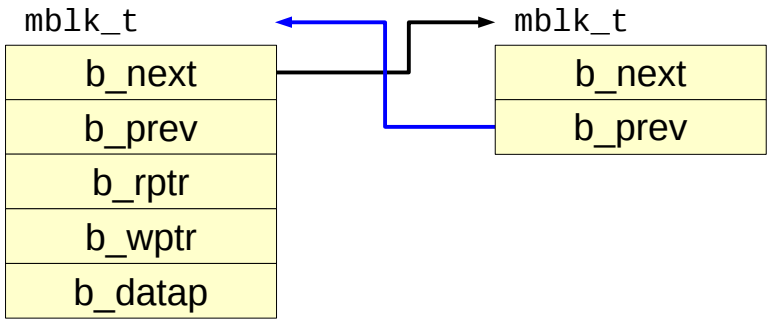
Структуры данных, представляющие пакет в кольцевом буфере передаются дальше по стеку и носят универсальный характер. В Linux ей является структура sk_buff:



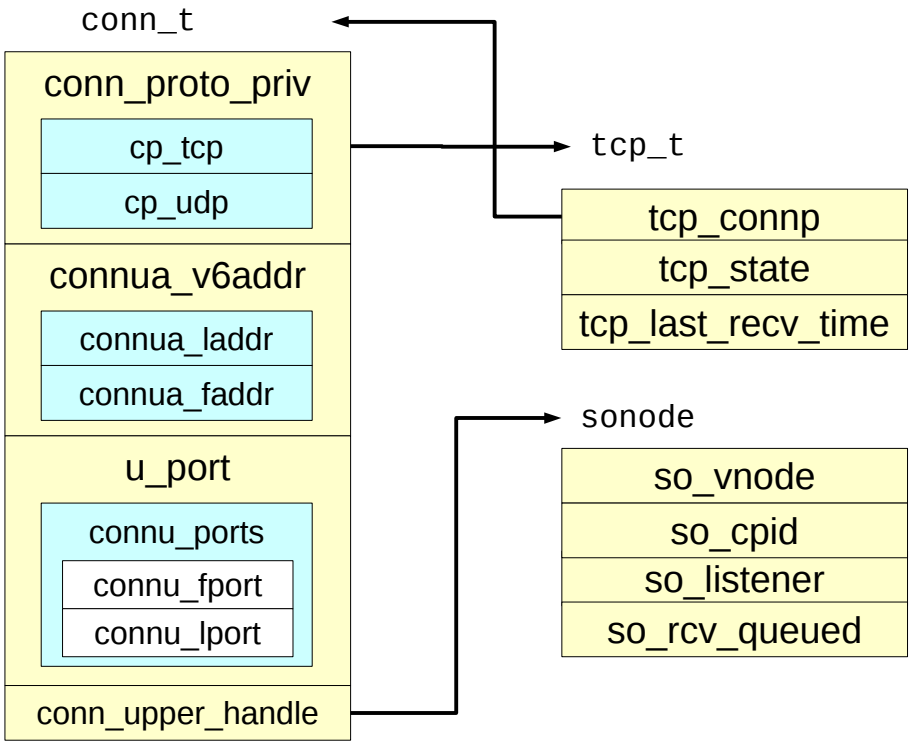
Эта структура содержит ряд указателей на данные, в том числе на заголовки различных слоев сетевого стека, длину len и временную метку создания блока tstamp, а также указатели next и prev для организации кольцевого буфера. Структура sock представляет из себя сетевой сокет: адрес и порт локального соткета записан в поля skc_rcv_saddr и skc_dnum, а удаленного — skc_daddr и skc_dport. Структура dev представляет из себя дескриптор устройства-сетевой карты. В более ранних версиях ядра порты и адреса принадлежат структуре inet_sock.

Отметим, что из-за различного порядка байт в процессорах (Little и Big-Endian) для работы с полями, содержащими порт и адрес необходимо применять функции преобразования сетевого порядка байт: ntohs, htohl и ntohll. Обратные им функции: htons, htonl и htonll соответственно.

В Solaris для сетевого стека исторически применяется Unix-подсистема STREAMS, обеспечивающая передачу сообщений между различными драйверами, а сообщение в ней представляется структурой mblk_t:



Длина сообщения определяется разницей между указателями b_wptr и b_rptr. Как видно, она не содержит никаких указателей на устройство или соответствующей ему сокет. Вместо этого функции сетевого стека передают их в виде первого аргумента (arg0): MAC-слой — mac_impl_t, IP-слой — ill_t, а TCP/UDP слой — conn_t:



В Solaris сокеты образуют виртуальную файловую систему sockfs, так что каждому подключению соответствует структура sonode. Поля connu_laddr и connu_lport соответствуют локальному адресу и порту, а connu_faddr и connu_fport — удаленному. Заметим, что в Solaris 10 эти названия несколько отличаются.

Для Linux 3.9 скрипт, трассирующий прием сообщений будет выглядеть следующим образом:

```
# 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);
        }' -v
```

В Linux 2.6.32, как уже и говорилось, потребуется преобразовать указатель к 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);
        }' -v
```

В Solaris 11 такой скрипт будет выглядеть следующим образом:

```
# dtrace -C -n '
    tcp_input_data:entry {
        this->conn = (conn_t*) arg0;
        this->mp = (mblk_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.
                              connua_laddr._S6_un._S6_u32[3])),
                ntohs(this->conn->u_port.connu_ports.connu_lport),

                this->mp->b_wptr - this->mp->b_rptr);
    }'
```

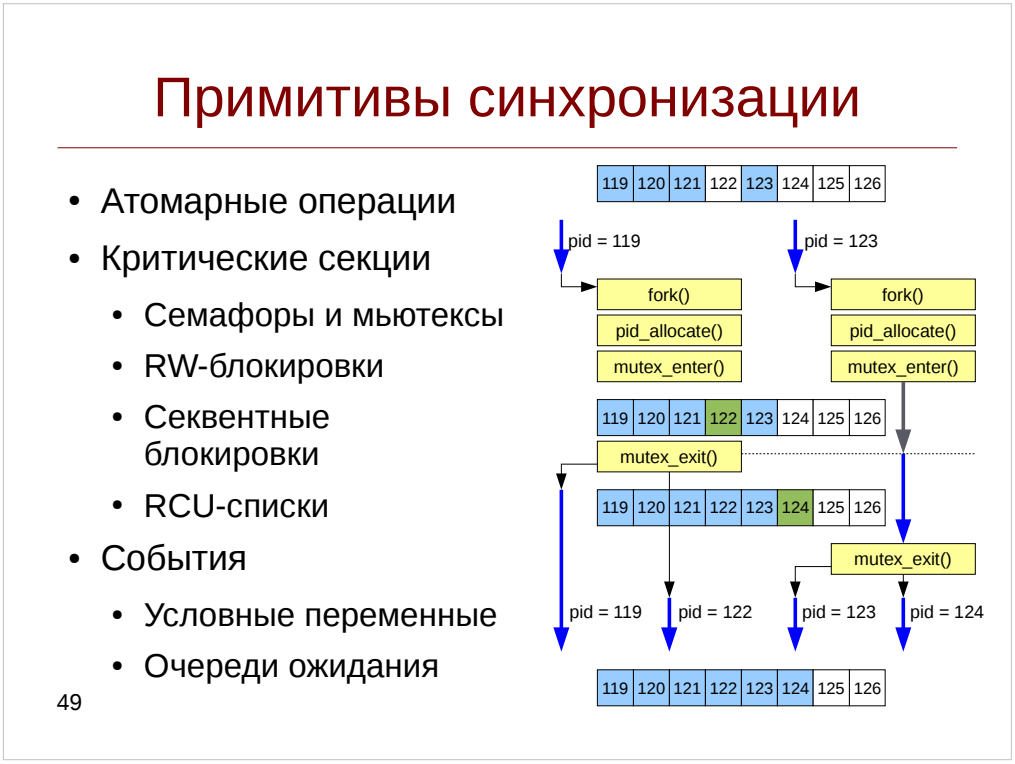
В Solaris 11 появился ряд сетевых провайдеров: tcp, udp, ip. Рассмотрим пробы предоставляемые ими а также их аналоги из Linux и SystemTap:

Действие	*DTrace*	*SystemTap*
TCP		
Подключение к удаленному узлу	tcp:::connect-request tcp:::connect-established tcp:::connect-refused	kernel.function("tcp_v4_connect")
Принятие подключения	tcp:::accept-established tcp:::accept-refused	kernel.function("tcp_v4_hnd_req")
Отключение	fbt:::tcp_disconnect	tcp.disconnect
Изменение состояния	tcp::state-change	
Передача	tcp:::send	tcp.sendmsg
Прием	tcp:::receive	tcp.receive tcp.recvmsg
IP		
Передача	ip:::send	kernel.function("ip_output")
Прием	ip:::receive	kernel.function("ip_rcv")
Сетевое устройство		
Передача	mac_tx, или функция драйвера NIC, например fbt::e1000g_receive:entry	netdev.transmit netdev.hard_transmit
Прием	mac_rx_common, или функция драйвера NIC, например fbt::e1000g_receive:entry	netdev.rx

Трассировку уровня сокетов удобно выполнять с помощью трассировки системных вызовов, с ними связанных. Кроме того, в SystemTap есть специальный tapset socket. В DTrace аналогичного провайдера нет, поэтому необходимо использовать привязку к функциям ядра через провайдер fbt. Сокеты в Solaris образуют отдельную файловую систему, с чем и связано наличие обратной ссылки so_vnode.

Кроме того, в Linux и Solaris есть ряд сетевых статистик, предоставляемых протоколом SNMP (посмотреть их значения можно вызвав команду netstat -s). Привязки к увеличениям счетчиков статистик реализованы провайдером mib в Solaris и tapset'ами tcpmib, ipmib и linuxmib в Linux.

Примитивы синхронизации



Современные ядра Linux и Solaris поддерживают многопоточность и параллелизм исполнения как служебных процедур, так и пользовательских процессов. Однако всегда есть общие ресурсы за которые потоки исполнения конкурируют между собой, что может вызывать неоднозначное поведение системы, порчу данных, а чаще всего системные паники.

Рассмотрим пример: пусть у нас есть процессы с pid 119 и pid 123, исполняющиеся на различных процессорах (сторого говоря, конкуренция за ресурсы может возникать и для одного процессора, если планировщик переключит задачу в «неудачное» время). И оба эти процесса практически одновременно вызвали системный вызов fork(). Если не предусмотреть никакого способа синхронизации между ними, то с большой долей вероятности они захватят первый свободный pid 122, что приведет к появлению в системе двух различных процессов с одинаковым pid. Для того, чтобы избежать этой ситуации, они используют мьютекс (как это делает Solaris) или комбинацию спин-блокировки и атомарной операции (как это делает Linux). Вызов mutex_enter в Solaris гарантирует, что продолжит исполняться только один процесс а второй или уснет пока процесс 119 не закончит свою работу или будет выполнять busy loop: цикл, проверяющий состояние процессора. Как только процесс с pid 119 освободит мьютекс, к тому времени зарезервировав pid 122 для дочернего процесса, процесс 123 разблокируется, и ему уже достанется следующий свободный номер — 124.

Самым простым типом примитивов синхронизации являются атомарные операции: они позволяют ограничить доступ к переменной на время выполнения операций над ней, например заблокировав системную шину, как это делает префикс lock на архитектуре x86. Атомарные операции широко используются в ядре Linux, однако не распространены в Solaris.

Если необходимо выполнить более одной ассемблерной инструкции, вводятся *критические секции* — участки кода, допускающие выполнение только одного потока. В частности в функции pid_allocate() ядра Solaris в нашем примере такой критической секцией является поиск свободного pid. Они реализуются разного рода блокировками, наиболее распространенными из которых являются *мьютексы* (mutex, от англ. MUTual EXclusion — взаимное исключение), допускающие исполнение в критической секции только одного потока. Мьютексы могут вызывать или выполнение потоком цикла активного ожидания (busy loop, уже обсуждавшийся ранее) или поместить поток в очередь ожидания (sleep queue) и таким образом передать процессор другому потоку.

Замечание: В англоязычной литературе блокировка называется *lock* (замок), тогда как помещение процессора в очередь ожидания — *block*. На русский язык оба этих слова переводятся как блокировка, что увы вызывает путаницу.

В Solaris наиболее распространены адаптивные мьютексы: они проводят некоторое время в цикле активного ожидания, и если за это время не удалось захватить мьютекс, уступают процессор. В Linux же наоборот: спин-блокировки (приводящие к циклу активного ожидания) и мьютексы, уступающие процессору — не связанные между собой примитивы. Семафоры — это мьютексы, допускающие исполнение фиксированного количества потоков в критической секции, а RW-блокировки (Read-Write-Lock), делящие потоки на читателей, не вносящие измения в общий ресурс, и писателей. RW-блокировки допускают одновременное исполнения сколь угодно большого количества читателей, но только одного писателя. Кроме того, в Linux реализованы секвентные блокировки (seqlock) и RCU-списки (Read-Copy-Update), рассмотрение которых мы оставим за рамками курса.

Linux в отличие от Solaris не содержит проб для трассировки блокировок, а использование привязок к функциям может привести к панике или зависанию системы. Хотя мы и приведем имена этих функций, используйте их на свой страх и риск. Также такие привязки доступны только в Guru-mode.

Список блокировок и соответствующих им проб ядра приведен в следующей таблице:

Действие	*DTrace*	*SystemTap*
Адаптивные блокировки		
Захват	lockstat:::adaptive-acquire lockstat:::adaptive-block lockstat:::adaptive-spin	kernel.function("mutex_lock*") kernel.function("debug_mutex_add_waiter")

Действие	*DTrace*	*SystemTap*
Адаптивные блокировки (продолжение)		
Освобождение	lockstat:::adaptive-release	kernel.function("mutex_unlock*") kernel.function("debug_mutex_unlock") kernel.function("debug_mutex_wake_waiter")
Спин-блокировки		
Захват	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")
Освобождение	lockstat:::spin-release	kernel.function("spin_unlock*") kernel.function("debug_spin_unlock")
RW-блокировки		
Захват	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")
Освобождение	lockstat:::rw-release	kernel.function("_raw_read_unlock") kernel.function("_raw_write_unlock") kernel.function("do_raw_read_unlock") kernel.function("debug_write_unlock")
R → W	lockstat:::rw-upgrade	
W → R	lockstat:::rw-downgrade	

⚠ *Замечание*: Названия проб в SystemTap, написанные мелким шрифтом, требуют включения специальных конфигурационных опций ядра, таких как CONFIG_DEBUG_MUTEXES и доступны только в отладочных ядрах.

⚠ *Замечание*: Пробы *-spin и *-block в DTrace срабатывают в тот же момент, что и acquire, однако предоставляют информацию о времени, проведенном на блокировке.

В Solaris для трассировки блокировок предусмотрен провайдер lockstat, а также одноименная утилита.

Еще один подвид примитивов синхронизации связан с необходимостью уведомлять потоки друг друга о происходящих событиях: например процессы командной оболочки, ожидающие ввода-вывода пользователя должны попадать в очередь соответствующего tty-устройства, а при вводе символов в терминал снова становится активными. В ядре Linux для реализации этого примитива синхронизации используются очереди ожидания (wait queue), имеющие однако вспомогательный интерфейс переменной завершения (completion variable)

Листинг 22. Скрипт wqtrace.stp

```
probe kernel.function("prepare_to_wait*"),
      kernel.function("add_wait_queue*") {
    if(pid() == stp_pid()) next;

    state = -1;
    if(@defined($state))
        state = $state;

    printf("[%d]%s %s:%s\n\twq head: %p wq: %p\n",
           pid(), execname(), symname(caller_addr()),
           probefunc(), $q, $wait);
    printf("\ttsk: %p state: %x func: %s\n",
           $wait->private, state, symname($wait->func));
}

probe kernel.function("wait_for_completion*") {
    if(pid() == stp_pid()) next;

    timeout = 0;
    if(@defined($timeout))
        timeout = $timeout;

    printf("[%d]%s %s:%s\n\tcompletion: %pwq head: %p timeout: %d\n",
            pid(), execname(), symname(caller_addr()),
            probefunc(), $x, &$x->wait, timeout);
}

probe kernel.function("wait_for_completion*").return {
    if(pid() == stp_pid()) next;

    printf("[%d]%s %s:%s\n\tcompletion: %p\n",
            pid(), execname(), symname(caller_addr()), probefunc(), $x);
}

probe kernel.function("finish_wait"),
      kernel.function("remove_wait_queue") {
    if(pid() == stp_pid()) next;

    printf("[%d]%s %s:%s\n\twq head: %p wq: %p\n",
           pid(), execname(), symname(caller_addr()),
           probefunc(), $q, $wait);
}

probe kernel.function("complete"),
      kernel.function("complete_all") {
    if(pid() == stp_pid()) next;

    printf("[%d]%s %s:%s\n\tcompletion: %p wq head: %p\n",
            pid(), execname(), symname(caller_addr()),
            probefunc(), $x, &$x->wait);
}

probe kernel.function("__wake_up"),
      kernel.function("__wake_up_locked*"),
      kernel.function("__wake_up_sync*") {
    if(pid() == stp_pid()) next;

    nr = -1
```

```
    if(@defined($nr_exclusive))
        nr = $nr_exclusive;
    if(@defined($nr))
        nr = $nr;

    printf("[%d]%s %s:%s\n\twq head: %p state: %p nr: %d\n",
            pid(), execname(), symname(caller_addr()),
            probefunc(), $q, $mode, nr);
}
```

Запустим следующую команду в одном из терминалов:

```
# bash -c 'for I in a b c d e f g h;
       do echo $I;
           sleep 0.1; done' | cat > /dev/null
```

Если мы запустим команду cat в терминале в SSH-сессии, то увидим подобный вывод:

```
[11704]bash pipe_write:__wake_up_sync_key
        wq head: 0xffff8800371f6628 state: 0x1 nr: 1
[11704]bash do_wait:add_wait_queue
        wq head: 0xffff880039e11220 wq: 0xffff88001f89ff20
        tsk: 0xffff88003971a220 state: ffffffffffffffff func:
child_wait_callback
[11705]cat pipe_wait:finish_wait
        wq head: 0xffff8800371f6628 wq: 0xffff88001ee47d40
[11705]cat pipe_read:__wake_up_sync_key
        wq head: 0xffff8800371f6628 state: 0x1 nr: 1
[11705]cat pipe_wait:prepare_to_wait
        wq head: 0xffff8800371f6628 wq: 0xffff88001ee47d40
        tsk: 0xffff8800397196c0 state: 1 func: autoremove_wake_function
```

Как видим, процесс bash вызвал функцию pipe_write(), чтобы записать данные в пайп, что в свою очередь разбудило процессы, ожидающие на очереди с адресом головы 0xffff8800371f6628. После этого проснулся процесс cat (о чем говорит вызов finish_wait), предупредил процессы-писатели через вызов (__wake_up_sync_key) и после вывода буквы в /dev/null, снова стал ожидать данных в пайпе.

Заметим, что если бы не перенаправили cat на /dev/null, он бы разбудил процесс висящий на tty, а если бы это был sshd, то мы также увидели и ожидание на сетевых сокетах, что однако затруднило бы восприятие примера.

В ядре Solaris реализованы условные переменные, работу с которыми можно оттрассировать с помощью следующего скрипта:

Листинг 23. Скрипт cvtrace.d

```
cv_wait*:entry  {
    self->timeout = 0;
}

cv_timedwait_hires:entry,
cv_timedwait_sig_hires:entry {
    self->timeout = (arg2 / arg3) * arg3;
```

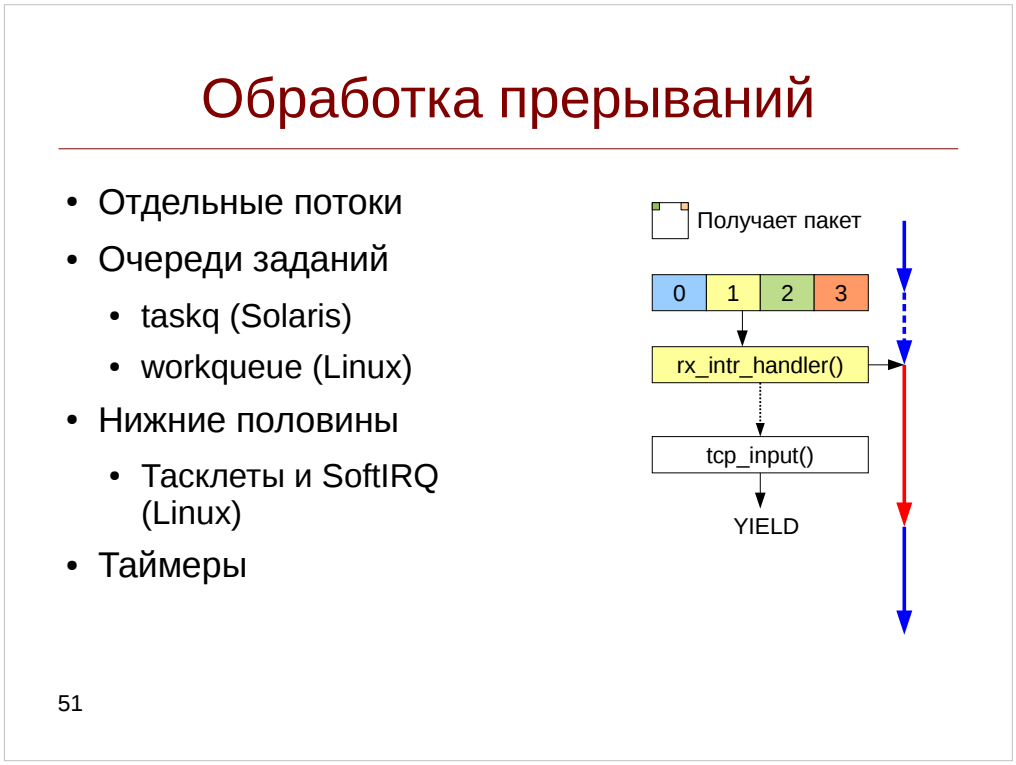
```
}

cv_wait:entry,
cv_wait_sig:entry,
cv_timedwait_hires:entry,
cv_timedwait_sig_hires:entry  {
    printf("[%d] %s %s cv: %p mutex: %p timeout: %d\n",
            pid, execname, probefunc, arg0, arg1, self->timeout);
    stack(4);
}

cv_signal:entry,
cv_broadcast:entry {
    printf("[%d] %s %s cv: %p\n",
            pid, execname, probefunc, arg0);
    stack(4);
}
```

Вывод для примера с утилитой cat выглядит так:

```
[15087] cat cv_wait_sig_swap_core cv: ffffc1000c9dde9c mutex: ffffc1000c9dde40
timeout: 0

              genunix`cv_wait_sig_swap+0x18
              fifofs`fifo_read+0xc7
              genunix`fop_read+0xaa
              genunix`read+0x30c
[15086] bash cv_broadcast cv: ffffc1000c9dde9c

              fifofs`fifo_wakereader+0x2f
              fifofs`fifo_write+0x316
              genunix`fop_write+0xa7
              genunix`write+0x309
```

Кроме этого ядро частично обеспечивает поддержку примитивов синхронизации в пространстве пользователя. В Solaris LWP-мьютексы используются только в ситуации инверсии приоритетов (priority inversion), во всех остальных случаях используются системные вызовы lwp_park()/lwp_unpark(). В Linux для этого предусмотрен специальный системный вызов: futex() (fast userspace mutex).

Обработка прерываний



До сих пор мы в том или ином виде рассматривали воздействия приложения на ядро, выполняемые посредством системных вызовов. Однако любое приложение не мыслимо без взаимодействия с пользователем, нажимающем на клавиатуру или опосредованно пересылающим пакет по сети. Как же приложение узнает о действиях пользователя? Самый простой способ реализовать это: постоянный опрос, например проверка клавиатурного регистра или кольцевого буфера:

```
while(1) {
     if(*keyboard > 0) {
          handle_key(*keyboard);
     }
}
```

Однако при пользователь нажимает клавиши не чаще нескольких раз в секунду, за это время процессор мог бы проделать миллиарды операций, вместо однообразного считывания одного и того же регистра. Для этого используется специальная схема — контроллер прерываний. В определенный внешними условиями момент времени (например, получение пакета по сети) он заставляет процессор сменить указатель инструкций, так что он вместо текущего процесса начинает исполнять код специальной *функции-обработчика прерывания* (она обычно называется ISR — Interrupt Service Routine, тогда как само прерывание — IRQ — Interrupt ReQuest).

Эта функция например вызывает функцию handle_key(), а когда так завершает

ся, например поместив событие в очередь устройства, управление возвращается процессу, и он продолжает исполнение с последней пропущенной инструкции. Механизмы, схожие с прерыванием также используются для уведомления процессором операционной системы о внутренней ошибке — в таком случае оно называется исключением. Примеры такой ошибки — исполнение процессором неверной инструкции, деление на ноль или уже упоминавшийся страничный сбой.

Прерывание может быть обработано двумя способами:

- В контексте потока, в котором это прерывание произошло. Такой способ обычно используется для высокоприоритетных прерываний, таких как Non-Maskable Interrupt
- С выбором на исполнение Interrupt Thread — отдельного потока, соответствующего прерыванию. В Linux каждый такой поток создается драйвером устройства, сохраняется в поле handler структуры irqaction, и активируется, если обработчик прерывания возвращает код IRQ_WAKE_THREAD. В Solaris это наиболее распространенный способ обработки прерывания, а каждому процессору соответствует свой поток-обработчик прерывания, сохраняемый в поле cpu_intr_thread структуры cpu_t. В таком случае диспетчеризация потоков происходит на общих основаниях (хотя в Solaris они имеют наивысший приоритет).

Для динамического инструментирования прерываний в Linux и Solaris предусмотрены специальные пробы и алиасы:

```
# 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)); }  '

# 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);  } '
```

Хотя прерывания позволяют более эффективно использовать процессорные ресурсы, они также таят проблемы для производительности системы: всякий раз, когда исполняется обработчик прерываний, исполнявшийся ранее процесс простаивает, а пользователь при этом теряет время. По этой причине разработчики ядра стараются максимально уменьшить объем кода обработчика прерывания, а все остальные действия «отложить на потом» — в ядре Linux эти действия образуют так называемую «нижнюю половину» обработчика прерывания (bottom half).

Для организации нижних половин в ядрах предусмотрен ряд примитивов:

- Отложенные прерывания softirq и тасклеты в Linux. Передают обработку прерываний процессам ksoftirqd-n. Активации отложенного прерывания соответствует точка трассировки ядра kernel.trace("softirq_raise"), а точкам входа и возврата — пробы softirq.entry и softirq.exit соответственно. Тасклеты используют softirq в качестве среды исполнения (имеют приоритет TASKLET_SOFTIRQ), и несут меньшие накладные расходы по сравнению с отложенными прерываниями.
- Таймеры в Linux и подсистема callout, а также вызовы timeout()/untimeout() в Solaris. Передают управление процедуре нижний половины по истечении указанного интервала времени; могут использоваться не только для обработки прерываний.
- Очереди заданий, включающие потоки-обработчики заданий и собственно очередь, в которую они помещаются обработчиком прерывания, а затем извлекаются одним из потоков-обработчиков. Такой подход позволяет накопить задания в очереди прежде чем передавать управление потокам-обработчикам, что позволяет повысить гранулярность планирования. Хотя многие драйвера самостоятельно реализуют концепцию очередей заданий, в Linux и Solaris есть встроенные механизмы для ее реализации: workqueue и taskq, соответственно.

Их трассировка может выполняться следующим образом:

```
# stap --all-modules -e '
     probe workqueue.insert, workqueue.execute {
          printf("%-16s %s %s(%p)\n",
               pn(), task_execname(wq_thread),
               symname(work_func), work);
     }  '

# 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); }'
```

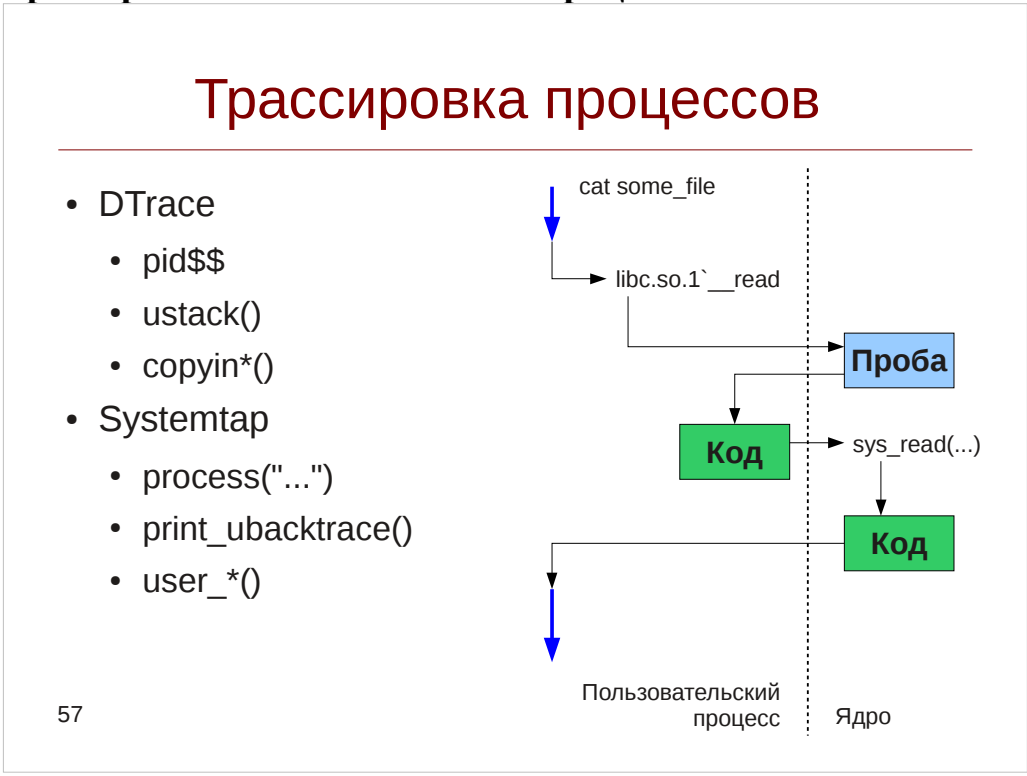
Замечание: в Solaris для динамических очередей заданий должны применяться пробы taskq-d-exec-start и taskq-d-exec-end.

Замечание: В ядре 2.6.36 механизм workqueue был переработан, из-за чего точки трассировки были переименованы, а tapset workqueue стал неработоспособен. К тому же до SystemTap 2.5 новые точки трассировки не доступны. Оттрассировать workqueue можно например так:

```
# 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);  } '
```

Модуль 5. Динамическая трассировка пользовательских приложений

Трассировка пользовательских процессов



Хотя ядро и является сложной по своей архитектуре программной сущностью, в реальных системах оно является лишь вершиной айзберга, так как реальная система не мыслима без пользовательского приложения, будь то веб-сервер, база данных или утилита cat, как во многих примерах, которые вы встречали ранее. Как и в случае ядра, для приложения можно установить пробу на вход и выход из любой функции, однако механизм трассировки приложений имеет ряд отличий: так как код проб располагается в ядре, для ее срабатывания требуется переключиться в режим ядра, как это происходит при прерываниях или системных вызовах.

Отличаются также и способы доступа к памяти приложения: для этого нужно предварительно скопировать ее участок в пространство ядра. В DTrace это выполняется функциями copyin(), самостоятельно выделяющая область памяти, copyinto(), копирующая в область памяти, предварительно выделенную процедурой alloca() и copyinstr(), копирующая строку, заканчивающуюся нулем. В SystemTap для этого предназначено семейство функций user_<тип>.

Трассировка пользовательских процессов в DTrace осуществляется провайдером pid, причем после его имени следует PID процесса, который будет трассироваться:

```
pid1449:libc:__read:entry
```

Полю модуль соответствует имя разделяемой библиотеки или a.out для программы. Вместо PID процесса можно указать специальный аргумент $$target, передав PID через опции -p или запустив команду посредством опции -c:

```
# dtrace -n '
     pid$$target:a.out:main:entry {
          ustack();
     }' -c cat
```

В SystemTap трассировка выполняется посредством проб вида process("*").function("*"), где в первых кавычках указывается имя бинарного файла (разделяемой библиотеки или исполняемого файла):

```
# stap -e '
     probe process("/lib64/libc.so.6").function("*readdir*") {
          print_ubacktrace();
}' -c ls -d /usr/bin/ls
```

В отличие от DTrace, эта проба будет срабатывать в контексте любого процесса, исполняющего функцию readdir().

Замечание: если бинарный файл можно найти используя переменную пути $PATH, абсолютный путь до него в выражении process можно опустить:

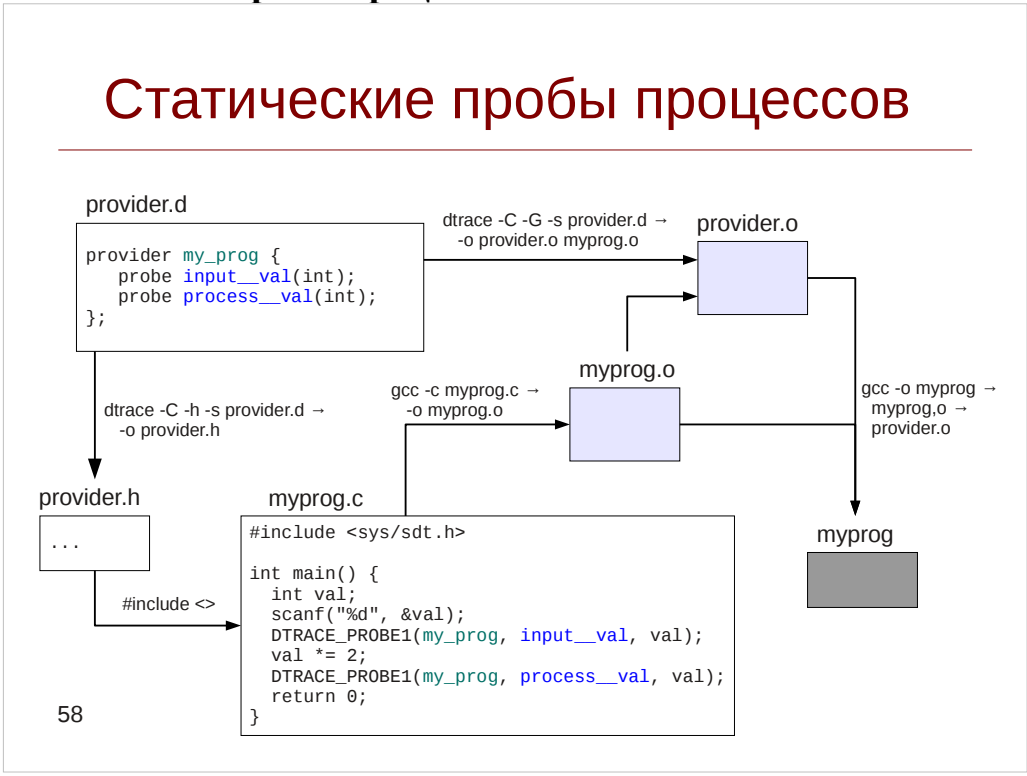
```
process("libc.so.6").function("*readdir*")
```

Для трассировки пользовательских процессов SystemTap в старых ядрах требует расширения ядра UTrace: оно доступно только при включенной опции CONFIG_UTRACE в параметрах ядра, что обычно делается в debug-сборках ядра. Начиная с ядра 3.5 подсистема utrace уступила свое место подсистеме uprobes, входящей в стандартную поставку ядра. Также отладочные символы вырезаются при сборке стандартных пакетов, так что потребуется установить debuginfo-пакеты (на Debian-подобных дистрибутивах они имеют суффикс -dbg).

Доступ к аргументам осуществляется также как и в ядровых пробах — через arg0-argN в DTrace или $имя_аргумента в SystemTap.

Отметим, что в DTrace привязка одновременно к нескольким PID невозможна — что затрудняет трассировку порожденных через truss -f. Кроме того, если разделяемая библиотека подгружается динамически через dlopen(), он также не сможет найти пробы и провайдеры в нем. Это ограничение можно обойти, используя destructive-действия DTrace: при загрузке требуемой библиотеки или fork() процесс останавливается через stop(). Посредством вызова system() запускается необходимый скрипт.

Статические пробы процессов



Как и в случае с ядром, DTrace и SystemTap поддерживают встраивание статических проб в любую точку программного кода. Чтобы сделать это, необходимо написать манифест провайдера .d, и затем пересобрать программу. Этот подход носит название User Statically Defined Probes (USDT).

Инструкции доступны для DTrace и SystemTap: https://wikis.oracle.com/display/DTrace/Statically+Defined+Tracing+for+User+Applications и https://sourceware.org/systemtap/wiki/AddingUserSpaceProbingToApps

Порядок действий для DTrace и SystemTap выглядит одинаково, для этого последний даже включает бинарный файл dtrace.

1. Создается манифест провайдера provider.d с описанием проб и их аргументов.
2. Из провайдера манифеста компилируется заголовочный файл:
   `# dtrace -C -h -s provider.d -o provider.h`
3. Пробы вносятся в инструментируемый файл в виде
   `DTRACE_PROBEn(<имя провайдера>, <имя пробы>, arg1, arg2 … argn);`
   или для этого используются макросы, сгенерированные утилитой dtrace:
   `MY_PROG_INPUT_VAL(arg1)`
4. Этот файл включается в программный код посредством директивы #include. Также нужно включить файл <sys/sdt.h>. Сама программа компилируется:
   `# gcc -c myprog.c -o myprog.o`
5. Создается ELF-файл провайдера. В качестве аргументов указываются объектные файлы, содержащие пробы и созданные на 3-м шаге:
   `# dtrace -C -G -s provider.d -o provider.o myprog.o`
6. Программа линкуется:
   `# gcc -o myprog myprog.o provider.o`

   SystemTap полностью совместим с системой USDT, и для этого содержит

утилиту DTrace и файл sys/sdt.h, который определяет как макросы STAP_PROBEn, так и обратно совместимые с DTrace DTRACE_PROBEn.

Также существуют специальные enabled-макросы, позволяющей определить, включена ли проба, или нет. Это позволяет избежать вычисления аргументов проб:

```
if(MY_PROG_INPUT_VAL_ENABLED()) {
     int arg1 = abs(val);
     MY_PROG_INPUT_VAL(arg1);
}
```

Использование USDT во многом похоже на провайдер pid$$ и конструкцию process:

```
# stap -e '
     probe process("./myprog").mark("input__val") {
          println($arg1);
     }'
# dtrace -n '
     input-val {
          printf("%d", arg0);
     }'
```

Полное имя пробы на DTrace:

```
my_prog10678:myprog:main:input-val
```

Полное имя пробы на SystemTap:

```
process("./myprog").provider("my_prog").mark("input__val")
```

Обратите внимание, что в DTrace два последовательных подчеркивания '__' в имени пробы заменяются на дефис '-'. Кроме того, как и для провайдера pid$$, пробы на DTrace становятся доступными уже после того, как программа запущена.

Проверить наличие USDT-проб в бинарном файле можно так: в Solaris — по наличии ELF-секции .SUNW_dof:

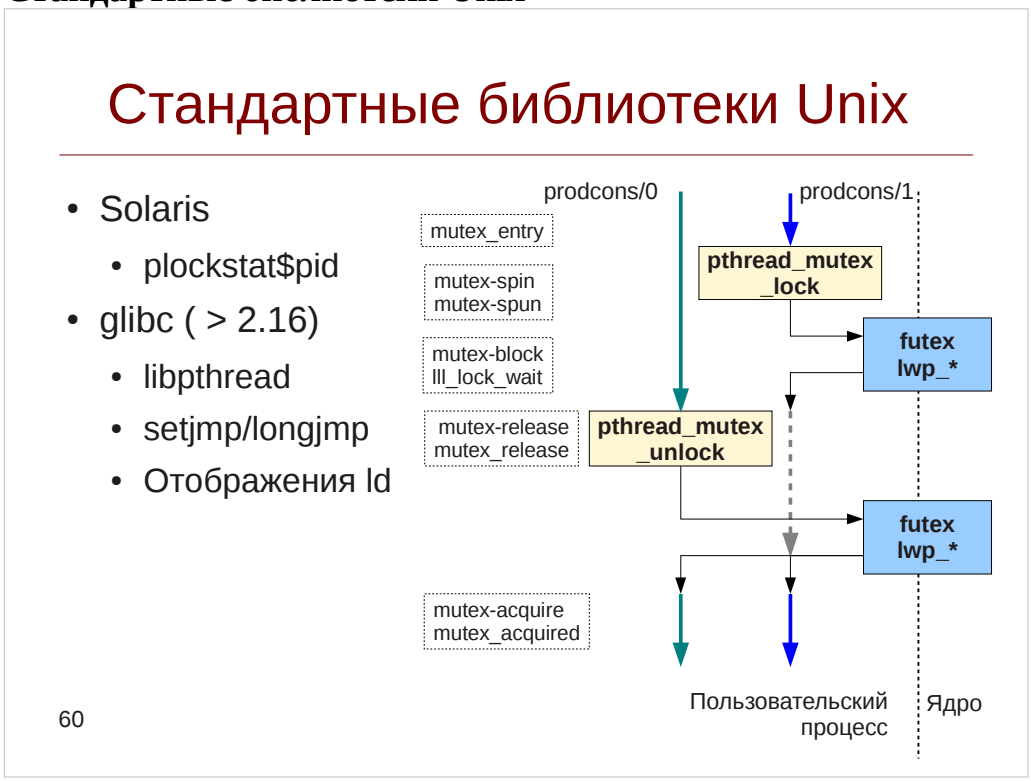
```
# elfdump ./myprog | ggrep -A 4 SUNW_dof
Section Header[19]:  sh_name: .SUNW_dof
    sh_addr:      0x8051708       sh_flags:   [ SHF_ALLOC ]
    sh_size:      0x7a9           sh_type:    [ SHT_SUNW_dof ]
    sh_offset:    0x1708          sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x8
```

на Linux — по наличию заметок в ELF-файле с типом stapsdt:

```
# readelf -n ./myprog | grep stapsdt
  stapsdt              0x00000033      Unknown note type: (0x00000003)
  stapsdt              0x00000035      Unknown note type: (0x00000003)
```

USDT-пробы могут быть порождены и динамически — для DTrace доступна соответствующая библиотека: https://github.com/chrisa/libusdt

Стандартные библиотеки Unix



USDT-пробы реализованы в стандартных библиотеках Unix, к которым можно отнести стандартную библиотеку C libc и libm, библиотеку POSIX Threads libpthread, библиотеку динамического компоновщика ld и некоторые другие. В Solaris пробы существуют лишь для мьютексов и блокировок чтения-записи пространства пользователя. В реализации библиотеки libc для Linux glibc (версии 2.16 или новее) такие пробы реализованы для более широкого спектра интерфейсов libpthread, в частности для создания и операции join для потоков, а также условных переменных pthread_cond_t. Кроме того, в Linux есть привязки к setjmp/longjmp и некоторые привязки для динамического компоновщика ld.

Рассмотрим пробы, связанные с блокировками на примере мьютексов в Solaris libc и glibc. В Solaris и DTrace эти пробы предоставляются провайдером plockstat, и запись выглядит следуюшим образом:

```
plockstat<pid>:::<имя пробы>
```

В SystemTap необходимо обращаться к ним через стандартную нотацию USDT-проб:

```
probe process("libpthread.so.0").mark("<имя пробы>")
```

В отличие от пространства ядра, пользовательские процессы вынуждены явно обращаться к ядру для того, чтобы разблокировать процесс, ожидающий освобождения блокировки (и соответственно заблокировать себя, если блокировка занята). В Linux для этих целей служит системный вызов futex (обращения к нему представлены низкоуровнеми пробами low-level lock, чье название начинается с lll), а в Solaris — группа системных вызовов lwp_*.

Приведем список проб, связанных с мьютексами в DTrace и SystemTap. В arg0

($arg1) передается указатель на мьютекс, а за информацией о других параметрах и пробах можно обратиться к документации.

Действие	*DTrace*	*SystemTap*
Создание	-	mutex_init
Уничтожение	-	mutex_destroy
Попытка захвата	-	mutex_entry
Активное ожидание	mutex-spin mutex-spun	-
Блокировка	mutex-block	lll_lock_wait
Захват	mutex-blocked mutex-acquire mutex-error	mutex_acquired
Освобождение	mutex-release	mutex_release lll_futex_wake

Напишем трассировщик событий на SystemTap:

Листинг 24. Скрипт pthread.stp

```
#!/usr/bin/env stap

@define libpthread %( "/lib64/libpthread.so.0" %)
@define libc %( "/lib64/libc.so.6" %)

probe process(@libpthread).mark("pthread_create") {
    if(pid() != target()) next;

    thread_id = user_long($arg1);
    thread_caller = usymname($arg3);
    thread_arg = $arg4;

    printf("[%d] pthread_create %x %s(%x)\n", tid(),
            thread_id, thread_caller, thread_arg);
}

probe process(@libpthread).mark("pthread_start") {
    if(pid() != target()) next;

    thread_id = $arg1;
    printf("[%d] pthread_start %x\n", tid(), thread_id);
}

probe process(@libpthread).mark("pthread_join") {
    if(pid() != target()) next;

    thread_id = $arg1;
    printf("[%d] pthread_join %x\n", tid(), thread_id);
}

probe process(@libpthread).mark("pthread_join_ret") {
```

```
    if(pid() != target()) next;

    thread_id = $arg1;
    printf("[%d] pthread_join %x return -> %d/%d \n", tid(),
            thread_id, $arg2, $arg3);
}

probe process(@libpthread).mark("mutex_*"),
      process(@libpthread).mark("cond_*"),
      process(@libpthread).mark("rdlock_*"),
      process(@libpthread).mark("wrlock_*"),
      process(@libpthread).mark("rwlock_*") {
    if(pid() != target()) next;

    printf("[%d] %s %p\n", tid(), pn(), $arg1);
    print_ustack(ucallers(5));
}

probe process(@libpthread).mark("lll_*"),
      process(@libc).mark("lll_*") {
    if(pid() != target()) next;

    printf("[%d] %s\n", tid(), pn());
    print_ustack(ucallers(5));
}
```

Если запустить его одновременно с нагрузчиком TSLoad, можно получить подобный вывод:

```
[8972] process("/lib64/libpthread.so.0").mark("mutex_entry") 0xe1a218
  0x7fbcf890fa27 : tpd_wqueue_put+0x26/0x6a [/opt/tsload/lib/libtsload.so]
[8972] process("/lib64/libpthread.so.0").mark("mutex_acquired") 0xe1a218
 0x7fbcf890fa27 : tpd_wqueue_put+0x26/0x6a [/opt/tsload/lib/libtsload.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/tsload/lib/libtsload.so]
[8971] process("/lib64/libpthread.so.0").mark("mutex_entry") 0xe1a628
 0x7fbcf9148fed : cv_wait+0x2d/0x2f [/opt/tsload/lib/libtscommon.so]
 0x7fbcf890f93f : tpd_wqueue_pick+0x44/0xbc [/opt/tsload/lib/libtsload.so]
[8971] process("/lib64/libpthread.so.0").mark("mutex_acquired") 0xe1a628
```

Можно видеть, что чтобы поместить в очередь запрос (в функции tpd_wqueue_put), управляющий поток (PID=8972) захватывает мьютекс, соответствующий очереди. После этого он рассылает всем потокам-обработчикам сигнал через cond_broadcast, из-за чего один из потоков-обработчиков (PID=8971) просыпается, также захватывает мьютекс в функции cv_wait.

Аналогичный трассировщик на DTrace будет выглядеть следующим образом:

Листинг 25. Скрипт pthread.d

```
#!/usr/bin/dtrace
#pragma D option bufsize=8m
#pragma D option switchrate=100hz
```

```
pid$target::pthread_create:entry {
    self->thread_id_ptr = (uintptr_t) arg0;
    self->thread_func = arg2;
    self->thread_arg = arg3;
}
pid$target::pthread_create:return {
    this->thread_id = * (uint_t*) copyin(self->thread_id_ptr, sizeof(uint_t));
    printf("[%d] pthread_create %x ", tid, this->thread_id);
    usym(self->thread_func);
    printf("(%x)\n", self->thread_arg);
}
pid$target::pthread_join:entry {
    self->thread_id = (uint_t) arg0;
    printf("[%d] pthread_join %x\n", tid, self->thread_id);
}
pid$target::pthread_join:return {
    printf("[%d] pthread_join:return %x -> %d\n", tid, self->thread_id, arg1);
}

plockstat$target:::,
pid$target::pthread_cond_*wait*:entry,
pid$target::pthread_cond_*wait*:return,
pid$target::pthread_cond_signal:entry,
pid$target::pthread_cond_broadcast:entry {
    printf("[%d] %s:%s ", tid, probefunc, probename);
    usym(arg0);
    printf("[%p]\n", arg0);
    ustack(6);
}
```

В Solaris и DTrace мы получим аналогичные результаты:

```
[7] mutex_lock_impl:mutex-acquire   0x46d4a0                    [46d4a0]
              libtsload.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]
```

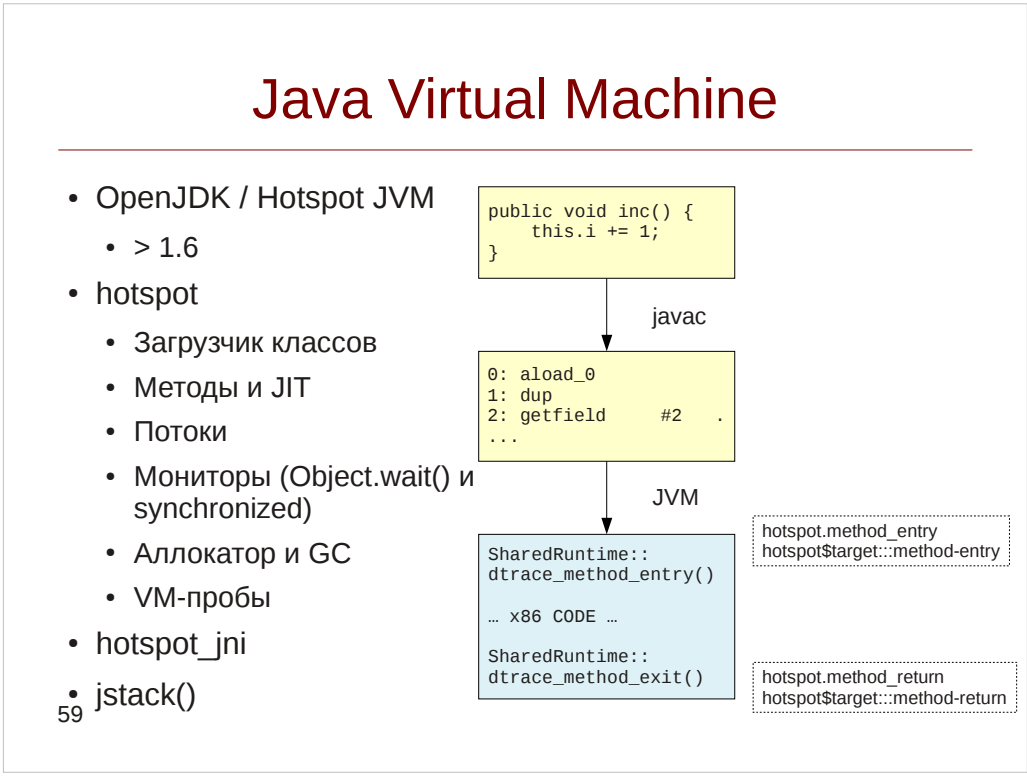
Упражнение 6

Реализуйте скрипты mtxtime.d и mtxtime.stp, которые подсчитывали бы задержку между попыткой блокировки мьютекса и собственно его захватом. Группируйте времена по пользовательскому стеку, а вывод осуществляйте с помощью логарифмических гистограмм.

Для демонстрации работоспособности написанных скриптов, используйте эксперимент pthread – как и в приведенных ранее примерах объектом тестирования будет сам нагрузчик TSLoad. Попытайтесь выявить мьютексы, которые демонстрируют задержки более 1 мс.

Замечание: Чтобы избежать проблем с выводом пользовательских стеков после завершения процесса в DTrace, для вывода агрегации, можно привязаться к функции файла tsexperiment experiment_unconfigure(), вызываемой при деконфигурации эксперимента.

Java Virtual Machine



DTrace и SystemTap предназначены для инструментирования нативных языков. Если же язык является интерпретируемым или имеет промежуточную среду исполнения — виртуальную машину, то для трассировки конструкций языка будь то методы или выбрасываемые исключения потребуется или поддержка DTrace и SystemTap виртуальной машиной (например, через USDT-пробы) или дополнительные действия со стороны среды динамической трассировки. Например в стековых интерпретаторах, вытаскивать аргументы функции arg0-argN пришлось из специальной области памяти, отведенной интерпретатором, таким образом пришлось бы реализовать соответствующий tapset.

DTrace и SystemTap поддерживаются в виртуальной машине Java OpenJDK и пересекающейся с ней по кодовой базе официальной реализации реализации от Oracle Hotspot JVM (Oracle Java) начиная с версии 1.6 и включает в себя два провайдера: hotspot и hotspot_jni. Последний предназначен для трассировки Java Native Interface, и мы оставим его рассмотрение за рамками курса, а вот на первом остановимся подробнее.

По-умолчанию провайдер hotspot предоставляет лишь пробы редких событий: загрузка классов, запуск потоков, глобальные события VM: ее запуск, остановка и работа сборщика мусора, работа JIT-компилятора. Это сделано с целью уменьшить влияние DTrace/SystemTap на производительность JVM. Дополнительные опции доступны только после передачи опций коммандной строки Java:

- -XX:+DTraceMethodProbes — разрешает трассировку вызовов и возвратов из методов
- -XX:+DTraceAllocProbes — разрешает трассировку выделения объектов на

куче
- -XX:+DTraceMonitorProbes — разрешает трассировку мониторов
- -XX:+ExtendedDTraceProbes — включает все эти опции

Эти опции могут быть также динамически установлены утилитой jinfo

Провайдер реализован в разделяемой библиотеке libjvm.so, которая загружается через dlopen(), и из-за указанных ограничений провайдера pid$$, в DTrace нельзя использовать синтаксис hotspot$$target:

```
# dtrace -c 'java Test' -n 'hotspot$target:::'
dtrace: invalid probe specifier hotspot$target:::: probe
description hotspot3662::: does not match any probes
```

Чтобы запустить скрипт потребуется использовать вспомогательный скрипт dtrace_helper.d: он останавливает исполнение Java в момент загрузки libjvm.so посредством деструктивного вызова stop(), а продолжается оно уже после компиляции и запуска дочернего скрипта. Кроме того, начиная с JDK7 в Solaris при линковке используется флаг -xlazyload из-за которого JVM не будет регистрировать hotspot-пробы автоматически. Поэтому, даже если процесс java уже запущен, его пробы не будут отображаться в выводе команды dtrace -l. Однако пробы доступны если явно обратиться к ним через PID процесса:

```
# dtrace -p 3682 -n 'hotspot$target:::'
dtrace: description 'hotspot$target:::' matched 66 probes
```

Такеже можно воспользоваться опцией -Z команды dtrace.

В SystemTap таких ограничений нет, и существует также tapset hotspot, содержащий алиасы проб из JVM:

```
# stap -e 'probe hotspot.* { println(pn()); }' -c 'java Test'
```

Напишем программу Greeter, которая будет используя четыре потока выводить строку «Hello, DTrace!». Ее реализация была взята из: http://www.solarisinternals.com/wiki/index.php/DTrace_Topics_Java с той разницей, что метод Greeting.greet() содержит ключевое слово synchronized, и как следствие использует монитор.

Листинг 26. Класс Greeting.java

```
public class Greeting {
        public synchronized void greet() {
                System.out.println("Hello, DTrace!");
        }
}
```

Листинг 27. Класс GreetingThread.java

```
class GreetingThread extends Thread {
        Greeting greeting;
```

```
        GreetingThread(Greeting greeting) {
            this.greeting = greeting;
            super.setDaemon(true);
        }

        public void run() {
            while(true) {
                        greeting.greet();
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                        }
            }
        }
}
```

Листинг 28. Класс Greeter.java

```
public class Greeter {
        public static void main(String[] args) {
                Greeting greeting = new Greeting();
                GreetingThread threads[] = new GreetingThread[4];

                for(int i = 0; i < 4; ++i) {
                        threads[i] = new GreetingThread(greeting);
                        threads[i].start();
                }

                for(int i = 0; i < 4; ++i) {
                        try {
                                threads[i].join();
                        }
                        catch(InterruptedException ie) {
                        }
                }
        }
}
```

Реализуем трассировщик на SystemTap:

Листинг 29. Скрипт hotspot.stp

```
#!/usr/bin/stap

probe hotspot.class_loaded
{
        printf("%12s [???] %s\n", name, class);
}

probe hotspot.method_entry, hotspot.method_return
{
        printf("%12s [%3d] %s.%s\n", name, thread_id, class, method);
```

```
}

probe hotspot.thread_start, hotspot.thread_stop
{
        printf("%12s [%3d] %s\n", name, id, thread_name);
}

probe hotspot.monitor_contended_enter, hotspot.monitor_contended_exit
{
        printf("%12s [%3d] %s\n", name, thread_id, class);
}
```

Аналогичный скрипт на DTrace будет выглядеть следующим образом и должен вызываться через dtrace_helper.d или использовать опцию -Z:

Листинг 30. Скрипт hotspot.d

```
#!/usr/sbin/dtrace -qs

#pragma D option switchrate=10hz

hotspot$target:::class-loaded
{
        printf("%12s [???] %s\n", probename, stringof(copyin(arg0, arg1)));
}

hotspot$target:::method-entry,
hotspot$target:::method-return
{
        printf("%12s [%3d] %s.%s\n", probename, arg0,
                 stringof(copyin(arg1, arg2)),
                 stringof(copyin(arg3, arg4)));
}

hotspot$target:::thread-start,
hotspot$target:::thread-stop
{
        printf("%12s [%3d] %s\n", probename, arg3,
                 stringof(copyin(arg0, arg1)));
}

hotspot$target:::monitor-contended-enter,
hotspot$target:::monitor-contended-exit
{
        printf("%12s [%3d] %s\n", probename, arg0,
                 stringof(copyin(arg2, arg3)));
}
```

Обратите внимание, что для чтения строки из JVM используется два аргумента и конструкция copyin(). Это связано с тем, что в отличие от C-строк, строки в JVM не являются нуль-терминированными и помимо самого указателя на строку в следующем аргументе передается ее длина.

Вывод скриптов будет выглядеть следующим образом:

```
class-loaded [???] Test
...
class-loaded [???] Greeting<init>
...
class-loaded [???] GreetingThread
...
thread-start [ 14] Thread-1
method-entry [  9] GreetingThread.run
method-entry [  9] Greetingt.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
```

Приведем список проб провайдера hotspot$target и tapset hotspot:

Действие	*DTrace*	*SystemTap*
Запуск и остановка JVM		
Запуск	vm-init-begin vm-init-end	hotspot.vm_init_begin hotspot.vm_init_end
Завершение	vm-shutdown	hotspot.vm_shutdown
Потоки		
Запуск	thread-start arg0:arg1 — имя потока arg2 — номер потока Java arg3 — системный номер потока arg4 — является ли поток демоном?	hotspot.thread_start thread_name — имя потока id — номер потока Java native_id — системный номер потока is_daemon — является ли поток демоном?
Завершение	thread-stop См. thread-start	hotspot.thread_stop См. hotspot.thread_start
Методы		
Вызов	method-entry arg0 — номер потока Java arg1:arg2 — имя класса arg3:arg4 — имя метода arg5:arg6 — сигнатура метода	hotspot.method_entry thread_id — номер потока Java class — имя класса method — имя метода sig — сигнатура метода
Возврат	method-return См. method-entry	hotspot.method_return См. hotspot.method_entry

Действие	*DTrace*	*SystemTap*
Загрузчик классов		
Загрузка	class-loaded arg0:arg1 — имя класса arg2 — номер загрузчика классов arg3 — является ли класс разделяемым (из разделяемого архива)?	hotspot.class_loaded class — имя класса classloader_id — номер загрузчика классов is_shared — является ли класс разделяемым (из разделяемого архива)?
Выгрузка		
Попытка входа	monitor-contended-enter arg0 — номер потока Java arg1 — уникальный номер монитора arg2:arg3 — имя класса	hotspot.monitor_contended_enter thread_id — номер потока Java id — уникальный номер монитора class — имя класса
Вход	monitor-contended-entered См. monitor-contended-enter	hotspot.monitor_contended_entered См. monitor_contended_enter
Выход	monitor-contended-exit См. monitor-contended-enter	hotspot.monitor_contended_exit См. monitor_contended_enter
Мониторы (wait)		
.wait() - вход	monitor-wait См. monitor-contended-enter + в arg4 — таймаут	hotspot.monitor_wait См. monitor_contended_enter + в переменной timeout — таймаут
.wait() - выход	monitor-waited См. monitor-contended-enter	hotspot.monitor_waited См. monitor_contended_enter
.notify()	monitor-notify См. monitor-contended-enter	hotspot.monitor_notify См. monitor_contended_enter
.notifyAll()	monitor-notifyAll См. monitor-contended-enter	hotspot.monitor_notifyAll См. monitor_contended_enter
Аллокатор и сборщик мусора		
Запуск сбора	gc-begin arg0 — является ли полным	hotspot.gc_begin is_full — является ли полным
Конец сбора	gc-end	hotspot.gc_end
Запуск сбора в пуле памяти	mem-pool-gc-begin arg0:arg1 — имя менеджера arg2:arg3 — имя пула памяти arg4 — начальный размер пула arg5 — используемая память arg6 — количество закоммиченных страниц arg7 — максимальный размер	hotspot.mem_pool_gc_begin manager— имя менеджера pool — имя пула памяти initial — начальный размер used — используемая память committed — количество закоммиченных страниц max — максимальный размер
Конец сбора в пуле памяти	mem-pool-gc-end См. mem-pool-gc-begin	hotspot.mem_pool_gc_end См. hotspot.mem_pool_gc_begin

Действие	*DTrace*	*SystemTap*
Аллокатор и сборщик мусора (продолжение)		
Выделение объекта	object-alloc arg0 — номер потока Java arg1:arg2 — имя класса arg3 — размер	hotspot.object_alloc thread_id — номер потока Java class — имя класса size — размер
JIT-Компилятор		
Начало компиляции	method-compile-begin arg0:arg1 — имя компилятора arg2:arg3 — имя класса arg4:arg5 — имя метода arg6:arg7 — сигнатура метода	hotspot.method_compile_begin compiler — имя компилятора class — имя класса method — имя метода sig — сигнатура метода
Окончание компиляции	method-compile-end См. method-compile-begin + B arg8 — результат компиляции	hotspot.method_compile_end См. hotspot.method_compile_begin + B $arg9 — результат компиляции
Загрузка скомпилированного метода	compiled-method-load arg0:arg1 — имя класса arg2:arg3 — имя метода arg4:arg5 — сигнатура метода arg6 — указатель на код arg7 — размер скомилированного кода	hotspot.compiled_method_load class — имя класса method — имя метода sig — сигнатура метода code — указатель на код size — размер скомилированного кода
Выгрузка скомпилированного метода	compiled-method-unload arg0:arg1 — имя класса arg2:arg3 — имя метода arg4:arg5 — сигнатура метода	hotspot.compiled_method_unload class — имя класса method — имя метода sig — сигнатура метода

Замечание: В SystemTap во всех пробах предоставляются также аргументы name, содержащий имя пробы, и probestr, содержащий отформатированные аргументы.

Замечание: Кроме перечисленных также есть ряд недокументированных проб, таких как class-initialization-* и ряд проб потоков: thread-sleep-*, thread-yield и др.

Также существует возможность вывести стек вызовов Java-методов:

```
# dtrace -n '
     syscall::write:entry
     / execname == "java" /
     { jstack(); }'
```

Замечание: На момент написания пособия существовал баг JDK-7187999, из-за которого в Solaris 11 jstack() не работает. Чтобы обойти его, можно воспользоваться следующей командой:

```
# dtrace -P foo<Java-PID>
```

Она вызовет ошибку, но заставит DTrace извлечь функции-помошники из процесса Java.

В SystemTap требуется выполнить дополнительные действия при инициализации JVM, поэтому jstack доступен только если запускать процесс через аргумент -c:

```
# stap -e '
     probe syscall.write {
     if(pid() == target())
          print_jstack();
     } ' -c 'java Test'
```

Отметим, что пробы методов не содержат передаваемых аргументов, таким образом инструментирование самих Java-приложений затруднено. Однако в DTrace есть поддержка JSDT — создание USDT-проб непостредственно из Java приложения. Она не поддерживается только в DTrace и только на операционных системах BSD и Solaris.

JSDT реализуется в пакете com.sun.tracing и sun.tracing. Каждый провайдер расширяет интерфейс com.sun.tracing.Provider, а каждая проба — это метод, определенный в таком провайдере. Модифицируем наш пример с классом Greeting, добавив поддержку JSDT:

Листинг 31. Класс Greeting.java

```
public class Greeting {
        GreetingProvider provider;

        public Greeting(GreetingProvider provider) {
                this.provider = provider;
        }

        public void greet(int greetingId) {
                provider.greetingStart(greetingId);
                System.out.println("Hello DTrace!");
                provider.greetingEnd(greetingId);
        }
}
```

Листинг 32. Класс GreetingProvider.java

```
import com.sun.tracing.Provider;

public interface GreetingProvider extends Provider {
        public void greetingStart(int greetingId);
        public void greetingEnd(int greetingId);
}
```

Листинг 33. Класс JSDT.java

```
import com.sun.tracing.*;

public class JSDT {
   static public void main(String[] args) {
       ProviderFactory providerFactory =
           new sun.tracing.dtrace.DTraceProviderFactory();
       GreetingProvider greetingProvider = (GreetingProvider)
            providerFactory.createProvider(GreetingProvider.class);

       Greeting greeter = new Greeting(greetingProvider);

       for(int id = 0; id < 100; ++id) {
            greeter.greet(id);

            try { Thread.sleep(500); }
            catch(InterruptedException ie) {}
       }

       greetingProvider.dispose();
   }
}
```

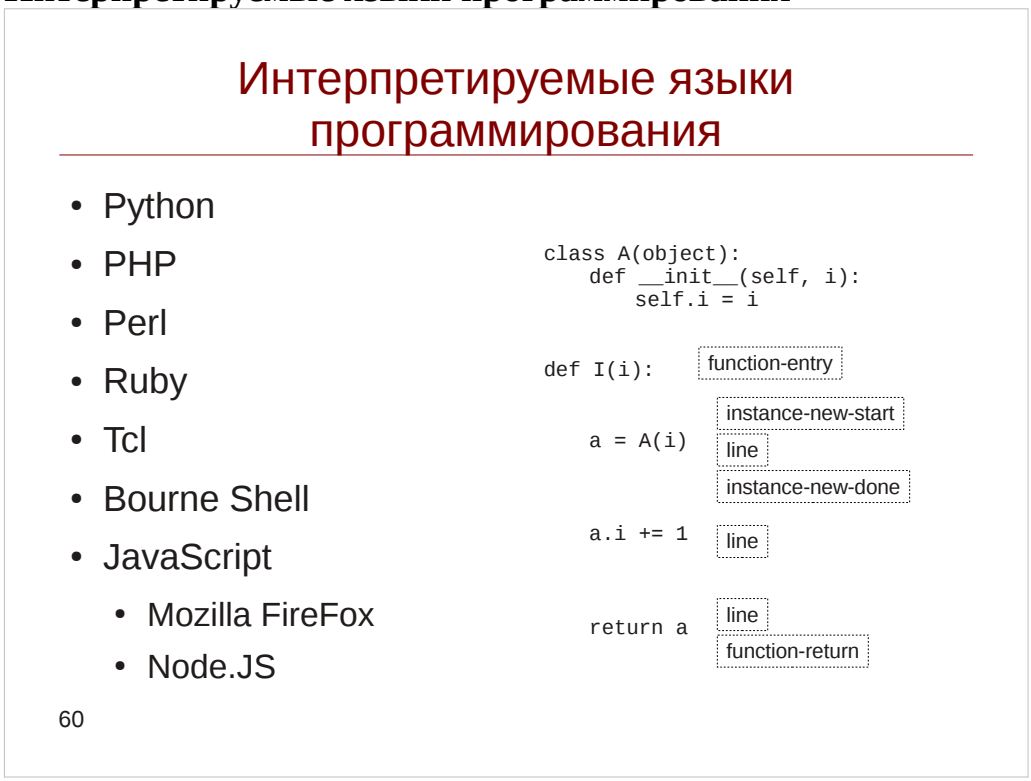
Так как классы из пакета sun.tracing являются «закрытыми», необходимо указать специальную опцию компилятору Java:

```
# javac -XDignore.symbol.file JSDT.java
```

Запустим на исполнение класс JSDT и посмотрим, стали ли доступны пробы DTrace:

```
root@sol11:~/java1/hs1# dtrace -l | grep GreetingProvider
69255 GreetingProvider3976      java_tracing                      unspecified
greetingStart
69256 GreetingProvider3976      java_tracing                      unspecified
greetingEnd
root@sol11:~/java1/hs1# dtrace -n 'greetingStart { trace(arg0); }'
dtrace: description 'greetingStart ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  69255        unspecified:greetingStart               61
```

Интерпретируемые языки программирования



Как и Java, многие интерпретируемые языки программирования поддерживают USDT, хотя и очень ограниченно.

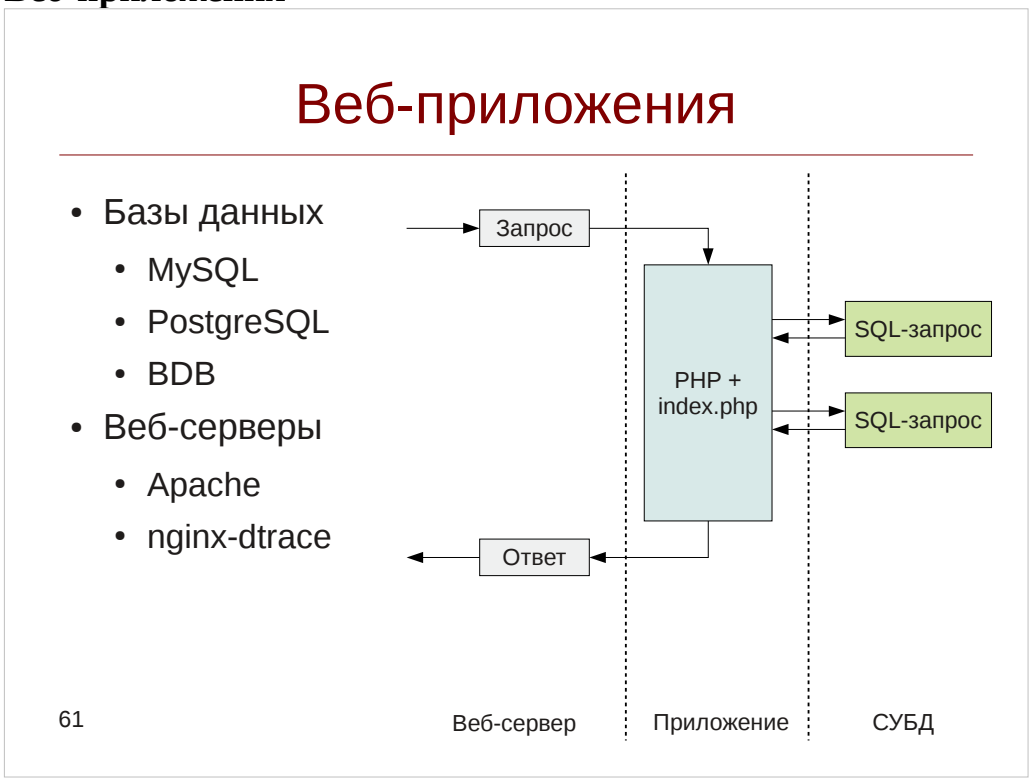
Язык	*Поддержка в исходных кодах*	*Поддержка в бинарных пакетах*	*Примечания*
Python	Нет	Да*	http://bugs.python.org/issue13405
PHP	Да	Нет	http://www.php.net/manual/ru/features.dtrace.dtrace.php
Perl	Да	Да*	http://perldoc.perl.org/perldtrace.html
Ruby	Да	Да*	
Tcl	Да	Да**	http://www2.tcl.tk/19923
Bourne Shell	Нет	Нет	https://blogs.oracle.com/brendan/entry/dtrace_bourne_shell_sh_provider1
Клиентский JavaScript	Да	Нет	Mozilla Firefox + TraceMonkey https://blogs.oracle.com/brendan/entry/dtrace_meets_javascript
Серверный JavaScript	Да	N/A	Node.JS + libV8

* В Fedora и Solaris
** Только в Fedora

Предоставляемые пробы можно разделить на несколько категорий:

- *Пробы входа и выхода из функции* — наиболее распространены:
  - В PHP, Python, Firefox/JavaScript — function-entry/function-return
  - В Perl — sub-entry/sub-return
  - В Ruby — method-entry/method-return
  - В Tcl — proc-entry/proc-return
- *Пробы исполнения внутри функции:* line в Python, соответствующая строчке исходного кода и execute-entry/execute-return, соответствующие инструкциям виртуальной машины Zend
- *Пробы загрузки и компиляции файла:* такие как compile-file-entry/compile-file-return в PHP и loading-file/loaded-file в Perl
- *Пробы ошибок и исключений:* raise в Ruby и exception-thrown/exception-caught/error в PHP
- *Пробы создания объекто*в: obj-create/obj-free в Tcl, instance-new-*/instance-delete-* в Python, object-create-start/object-create-done/object-free
- *Пробы сброщика мусора:* gc-start/gc-done в Python, gc-*-begin/gc-*-end в Ruby 2.0 или gc-begin/gc-end в Ruby 1.8

Веб-приложения



Думаю, вы обратили внимание, что среди интерпретируемых языков программирования встречаются такие языки, как PHP, Perl, Python и Ruby, крайне популярные в среде веб-разработки. Кроме того, статические пробы USDT реализованы в веб-серверах Apache HTTP Server и nginx (в отдельном дереве исходных кодов), а также в реляционных СУБД MySQL и PostgreSQL и нереляционной Berkeley DB.

Замечание: хотя разработчики Apache HTTP Server и заявляют поддержку DTrace проб, система сборки не была модифицирована должным образом, и поэтому указав опцию `--enable-dtrace` вы увидите следующее сообщение:

```
DTrace Support in the build system is not complete. Patches Welcome!
```

Во время подготовки курса я предоставил патч разработчикам: https://issues.apache.org/bugzilla/show_bug.cgi?id=55793, однако текущий статус бага неясен.

Приведем списки полезных проб для веб-сервера Apache, базы данных MySQL и интерпретатора PHP. В таблице указаны только имена меток (проб). Для Apache необходимо указывать провайдер ap* в DTrace:

```
ap*:::<mark-name>
```

В SystemTap process("httpd"), предварительно указав путь до бинарника httpd в переменной PATH:

```
process("httpd").mark("<mark__name>")
```

Аналогично, для MySQL — mysql* и process("mysqld"), а для SAPI-модуля PHP:

php* и process("libphp5.so").

Для Apache список проб выглядит следующим образом:

Действие	*DTrace*	*SystemTap*
Перенаправление запроса	internal-redirect arg0 — старый URI arg1 — новый URI	internal__redirect $arg1 — старый URI $arg2 — новый URI
Получение запроса	read-request-entry arg0 — структура request_rec arg1 — структура conn_rec	read__request__entry $arg1 — структура request_rec $arg2 — структура conn_rec
	read-request-success arg0 — структура request_rec arg1 — метод (GET/POST/...) arg2 — URI запроса arg3 — имя сервера arg4 — статус HTTP	read__request__success $arg1 — структура request_rec $arg2 — метод (GET/POST/...) $arg3 — URI запроса $arg4 — имя сервера $arg5 — статус HTTP
	read-request-failure arg0 — структура request_rec	read__request__failure $arg1 — структура request_rec
Обработка запроса	process-request-entry arg0 — структура request_rec arg1 — URI запроса	process__request__entry $arg1 — структура request_rec $arg2 — URI запроса
	process-request-return arg0 — структура request_rec arg1 — URI запроса arg2 — статус HTTP	process__request__return $arg1 — структура request_rec $arg2 — URI запроса $arg3 — статус HTTP

Замечание: во время запуска пробы read-request-entry структура request_rec еще не заполнена.

Замечание: значительное количество проб предоставляются системой Apache Hooks. Однако, к сожалению, они не имеют полезных аргументов.

Для интерпретатора PHP список проб будет выглядеть следующим образом:

Действие	*DTrace*	*SystemTap*
Обработка запроса		
Начала	request-startup arg0 — файл arg1 — URI запроса arg2 — метод запроса	request__startup $arg1 — файл $arg2 — URI запроса $arg3 — метод запроса
Завершение	request-shutdown См. request-startup	request__shutdown См. request__startup

Действие	*DTrace*	*SystemTap*
JIT-компиляция		
Компиляция	compile-file-entry arg0 — имя исходного файла arg1 — имя скомпилированного файла	compile__file__entry $arg1 — имя исходного файла $arg2 — имя скомпилированного файла
Завершение	compile-file-return См. compile-file-entry	compile__file__return См. compile__file__entry
Функции		
Вызов	function-entry arg0 — имя функции arg1 — имя файла arg2 — номер строки arg3 — имя класса arg4 — оператор области видимости ::	function__entry $arg1 — имя функции $arg2 — имя файла $arg3 — номер строки $arg4 — имя класса $arg5 — оператор области видимости ::
Возврат	function-return См. function-entry	function__return См. function__entry
Исполнение		
Начало	execute-entry arg0 — имя файла arg1 — номер строки	execute__entry $arg1 — имя файла $arg2 — номер строки
Окончание	execute-return См. execute-entry	execute__return См. execute__entry
Ошибки и исключения		
Ошибка	error arg0 — сообщение об ошибке arg1 — имя файла arg2 — номер строки	error $arg1 — сообщение об ошибке $arg2 — имя файла $arg3 — номер строки
Брошенное исключение	exception-thrown arg0 — имя класса исключения	exception__thrown arg0 — имя класса исключения
Пойманное исключение	exception-caught См. exception-thrown	exception__caught См. exception__thrown

Для базы данных MySQL полный список проб можно найти здесь: http://dev.mysql.com/doc/refman/5.7/en/dba-dtrace-mysqld-ref.html

Приведем несколько значимых для приложений проб MySQL:

Действие	*DTrace*	*SystemTap*
Подключение		
Установка соединения	connection-start arg0 — номер подключения arg1 — имя пользователя arg2 — имя хоста	connection__start $arg1 — номер подключения $arg2 — имя пользователя $arg3 — имя хоста
Завершение	connection-done arg0 — статус подключения arg1 — номер подключения	connection__done $arg1 — статус подключения $arg2 — номер подключения
Разбор запроса		
Начало	query-parse-start arg0 — текст запроса	query__parse__start $arg1 — текст запроса
Окончание	query-parse-done arg0 — статус	query__parse__done $arg1 — статус
Исполнение запроса		
Начало	query-exec-start arg0 — текст запроса arg1 — номер подключения arg2 — имя базы данных arg3 — имя пользователя arg4 — имя хоста arg5 — источник запроса (курсор, процедура, и др.)	query__exec__start $arg1 — текст запроса $arg2 — номер подключения $arg3 — имя базы данных $arg4 — имя пользователя $arg5 — имя хоста $arg6 — источник запроса (курсор, процедура, и др.)
Окончание	query-exec-done arg0 — статус	query__exec__done $arg1 — статус

Трассировщик стека веб-приложений на языке SystemTap будет выглядеть следующим образом:

Листинг 34. Скрипт web.stp

```
@define httpd %( "/usr/local/apache2/bin/httpd" %)
@define libphp5 %( "/usr/local/apache2/modules/libphp5.so" %)
@define mysqld %( "/usr/local/mysql/bin/mysqld" %)

global parsequery;
global execquery;

function basename:string(s:string) {
    len = strlen(s)
    i = len

    while(i > 0) {
        /* 47 is code for '/' */
        if(stringat(s, i - 1) == 47)
            return substr(s, i, len - i);
```

```
        --i;
    }

    return s;
}

probe process(@httpd).mark("internal__redirect")  {
    printf("[httpd] redirect\t'%s' -> '%s'\n",
           user_string($arg1), user_string($arg2));
}

probe process(@httpd).mark("read__request__entry") {
    printf("[httpd] read-request\n");
}

probe process(@httpd).mark("read__request__success") {
    servername = ($arg4) ? user_string($arg4) : "???";

    printf("[httpd] read-request\t%s %s %s  [status: %d]\n",
           user_string($arg2), servername, user_string($arg3), $arg5);
}

probe process(@httpd).mark("process__request__entry") {
    printf("[httpd] process-request\t'%s'\n", user_string($arg2));
}

probe process(@httpd).mark("process__request__return") {
    printf("[httpd] process-request\t'%s' access-status: %d\n",
           user_string($arg2), $arg3);
}

probe process(@libphp5).mark("request__startup"),
      process(@libphp5).mark("request__shutdown") {
    printf("[ PHP ] %s\n\t%s '%s' file: %s \n", pn(), user_string($arg3),
           user_string($arg2), user_string($arg1));
}

probe process(@libphp5).mark("function__entry"),
      process(@libphp5).mark("function__return") {
    printf("[ PHP ] %s\n\t%s%s%s file: %s:%d \n", pn(),
          user_string($arg4), user_string($arg5), user_string($arg1),
          basename(user_string($arg2)), $arg3);
}

probe process(@mysqld).mark("query__parse__start") {
   parsequery[tid()] = user_string_n($arg1, 1024);
}

probe process(@mysqld).mark("query__parse__done") {
   printf("[MySQL] query-parse\t'%s' status: %d\n", parsequery[tid()], $arg1);
}

probe process(@mysqld).mark("query__exec__start") {
   execquery[tid()] = user_string_n($arg1, 1024);
}

probe process(@mysqld).mark("query__exec__done") {
   printf("[MySQL] query-exec\t'%s' status: %d\n", execquery[tid()], $arg1);
}
```

Вывод скрипта для главной страницы Drupal 7 будет выглядеть так (вывод сокращен):

```
[httpd] read-request
[httpd] read-request    GET ??? /drupal/modules/contextual/images/gear-
select.png  [status: 200]
[httpd] process-request '/drupal/modules/contextual/images/gear-select.png'
[httpd] process-request '/drupal/modules/contextual/images/gear-select.png'
access-status: 304
[httpd] read-request
[httpd] read-request    GET ??? /drupal/  [status: 200]
[httpd] process-request '/drupal/'
[ PHP ] request-startup GET '/drupal/index.php' file:
/usr/local/apache2/htdocs/drupal/index.php
[ PHP ] function-entry  main file: index.php:19
[ PHP ] function-return main file: index.php:19
…
[ 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.sid = 'yIR5hLWScBNAfwOby2R3FiDfDokiU456ZE-rBDsPfu0''
status: 0
[MySQL] query-exec      'SELECT u.*, s.* FROM users u INNER JOIN sessions s ON
u.uid = s.uid WHERE s.sid = 'yIR5hLWScBNAfwOby2R3FiDfDokiU456ZE-rBDsPfu0''
status: 0
...
[ PHP ] request-shutdown        GET '/drupal/index.php' file:
/usr/local/apache2/htdocs/drupal/index.php
[httpd] process-request '/drupal/index.php' access-status: 200
```

Замечание: после запуска скрипта web.stp требуется перезапустить веб-сервер Apache.

Аналогичный скрипт но на DTrace выглядит так:

Листинг 35. Скрипт web.d

```
#pragma D option strsize=2048
#pragma D option bufsize=128M
#pragma D option switchrate=20hz

ap*:::internal-redirect  {
    printf("[httpd] redirect\t'%s' -> '%s'\n", copyinstr(arg0),
copyinstr(arg1));
}

ap*:::read-request-entry {
    printf("[httpd] read-request\n");
}

ap*:::read-request-success {
    this->servername = (arg3) ? copyinstr(arg3) : "???";

    printf("[httpd] read-request\t%s %s %s  [status: %d]\n",
        copyinstr(arg1), this->servername, copyinstr(arg2), arg4);
}
```

```
ap*:::process-request-entry {
    printf("[httpd] process-request\t'%s'\n", copyinstr(arg1));
}

ap*:::process-request-return {
    printf("[httpd] process-request\t'%s' access-status: %d\n",
        copyinstr(arg1), arg2);
}

php*:::request-startup,
php*:::request-shutdown {
    printf("[ PHP ] %s\t%s '%s' file: %s \n", probename,
        copyinstr(arg2), copyinstr(arg1), copyinstr(arg0));
}

php*:::function-entry,
php*:::function-return {
    printf("[ PHP ] %s\t%s%s%s file: %s:%d \n", probename,
        copyinstr(arg3), copyinstr(arg4), copyinstr(arg0),
        basename(copyinstr(arg1)), arg2);
}

mysql*:::query-parse-start {
   self->parsequery = copyinstr(arg0, 1024);
}

mysql*:::query-parse-done {
   printf("[MySQL] query-parse\t'%s' status: %d\n", self->parsequery, arg0);
}

mysql*:::query-exec-start {
   self->execquery = copyinstr(arg0, 1024);
}

mysql*:::query-exec-done {
   printf("[MySQL] query-exec\t'%s' status: %d\n", self->execquery, arg0);
}
```

Замечание: из-за большого количества выводимой информации (в основном — это функции PHP) в скрипте были увеличены размеры буферов.

Упражнение 7

Напишите скрипты topphp.d и topphp.stp, измеряющие среднее время выполнения каждой функции PHP и количество ее вызовов. Группируйте функции по URI вызова и полному имени функции, включая имя класса, если таковое имеется.

Для демонстрации используйте нагрузку drupal, а запускайте нагрузчик на внешнем узле (например для тестирования в Linux в качестве нагрузчика можно использовать хост с Solaris и наоборот), переопределяя при этом параметр host нагрузки http:

```
# /opt/tsload/bin/tsexperiment -e drupal/ run  \
     -s workloads:drupal:params:server=192.168.13.102
```

Приложение 1. Подсказки и решения к упражнениям

Упражнение 1

Данное упражнение призвано закрепить некоторые особенности языков динамической трассировки, которые изучались в модуле 1. Итак, выберем пробы для трассировки: так как мы будем трассировать системные вызовы, то они будут относиться к провайдеру и тапсету syscall. Найдем соответствующие им и системному вызову open() пробы в SystemTap:

```
     # 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
```

И информацию о пробах в DTrace:

```
     # dtrace -l -f openat\* -v
   ID   PROVIDER            MODULE                  FUNCTION NAME
14167    syscall                                    openat entry
 ...
```

Чтобы извлечь возвращаемое значение, нужно в return-пробах взять переменную $return в SystemTap или первый аргумент arg1 в DTrace. Нам также потребуются значения флагов — в вызове openat() в Solaris они передаются вторым аргументом arg2, в системном вызове open() в Linux — для этого можно использовать переменную-аргумент $flags или соответствующую ей локальную переменную (она создается tapset'ом syscall) flags.

Аналогично, путь до открываемого файла содержится в первом аргументе arg0 системного вызова openat() в Solaris и переменными $filename и filename в Linux и SystemTap. Нюанс заключается в том, что путь представляет из себя строчку, которая содержится в пользовательском адресном пространстве, поэтому чтобы вывести ее, нужно сначала ее скопировать с помощью функции DTrace copyinstr() и одной из функций user_string* SystemTap(). Замечу, что локальная переменная filename создается с помощью вызова user_string_quoted(), ее мы и будем использовать в скриптах.

Выводимые нами переменные доступны в разных пробах: флаги и путь до файла — во входной пробе (в SystemTap они также доступны и в возвратной пробе, но не обязаны иметь корректное значение, как уже упоминалось в курсе), а возвращаемое значение — в возвратной пробе. Из-за этого надо как-то сохранить значения аргументов, для чего нам подойдут контекстные переменные.

Наконец, определенная сложность возникает при создании строки соответствующей маске флагов, но она легко устраняется с помощью операций if-else в SystemTap, или тернарного оператора ?: в DTrace, и операций конкатенации строк.

Подытоживая вышесказанное, у меня получились следующие вариации скриптов:

Листинг 36. Скрипт opentrace.d

```
/* Подсказка: определены в /usr/lib/dtrace/io.d
inline int O_WRONLY = 1;
inline int O_RDWR = 2;
inline int O_APPEND = 8;
inline int O_CREAT = 256;
*/
inline int O_CLOEXEC = 8388608;

this string flag_str;

syscall::openat*:entry {
    self->path = copyinstr(arg1);
    self->flags = arg2;
}

syscall::openat*:return
{
    this->flags_str = strjoin(
        self->flags & O_WRONLY
        ? "O_WRONLY"
        : self->flags & O_RDWR
            ? "O_RDWR"
            : "O_RDONLY",
        strjoin(
            self->flags & O_APPEND  ? "|O_APPEND" : "",
            self->flags & O_CREAT   ? "|O_CREAT" : ""));

    printf("%s[%d(%d:%d)] open(\"%s\", %s) = %d\n",
                execname, pid, uid, gid,
                self->path, this->flags_str, arg1);
}
```

Для SystemTap я воспользовался операцией sprintf вместо конкатенации строк:

Листинг 37. Скрипт opentrace.stp

```
global O_WRONLY = 1;
global O_RDWR = 2;
global O_APPEND = 1024;
global O_CREAT = 64;

global t_path, t_flags;

probe syscall.open {
    t_path[tid()] = filename;
    t_flags[tid()] = flags;
}

probe syscall.open.return {
    flags = t_flags[tid()];

    if(flags & O_RDWR) {
        flags_str = "O_RDWR";
```

```
    } else if(flags & O_WRONLY) {
        flags_str = "O_WRONLY";
    } else {
        flags_str = "O_RDONLY";
    }
    if(flags & O_APPEND) {
        flags_str = sprintf("%s|%s", flags_str, "O_APPEND");
    }
    if(flags & O_CREAT) {
        flags_str = sprintf("%s|%s", flags_str, "O_CREAT");
    }

    printf("%s[%d(%d:%d)] open(%s, %s) = %d\n",
                execname(), pid(), uid(), gid(),
                t_path[tid()], flags_str, $return);
}
```

Остается только дополнить эти скрипты предикатами так, чтобы выводились только пути, содержащие /etc. Для этого придется воспользоваться функцией strstr() в DTrace или isinstr() из SystemTap.

Упражнение 2

В данном упражнении для каждого показателя нужно будет применить агрегацию count для системных вызовов open() или openat(). Для очистки агрегации следует воспользоваться действием trunc (DTrace) или операцией delete (SystemTap). В качестве пример следует использовать скрипты wstat.d и wstat.stp, представленные в разделе Агрегации на с. 51.

Чтобы вывести текущее системное время в DTrace, в качестве переменной нужно использовать walltimestamp и указать спецификатор форматирования %Y в printf, тогда как в SystemTap можно воспользоваться парой функций ctime и gettimeofday_s. Чтобы выводить данные с периодичностью, нужно воспользоваться таймерной пробой timer.s(X) — в SystemTap — или tick-Xs — в DTrace. Так как период задается аргументом коммандной строки, на место X надо подставить $1.

Итак, скрипт для DTrace будет выглядеть следующим образом:

Листинг 38. Скрипт openaggr.d

```
syscall::openat*:entry
/(arg2 & O_CREAT) == O_CREAT/ {
    @create[execname, pid] = count();
}

syscall::openat*:entry
/(arg2 & O_CREAT) == 0/ {
    @open[execname, pid] = count();
}

syscall::openat*:return
/ arg1 > 0 / {
```

```
    @success[execname, pid] = count();
}

tick-$1s {
    printf("%Y\n", walltimestamp);
    printf("%12s %6s %6s %6s %s\n",
              "EXECNAME", "PID", "CREATE", "OPEN", "SUCCESS");
    printa("%12s %6d %@6d %@6d %@d\n", @create, @open, @success);
    trunc(@create); trunc(@open); trunc(@success);
}
```

Для SystemTap получился такой скрипт:

Листинг 39. Скрипт openaggr.stp

```
global open, creat, success
global O_CREAT = 64;

probe syscall.open {
    if(flags & O_CREAT)
        creat[execname(), pid()] <<< 1;
    else
        open[execname(), pid()] <<< 1;
}

probe syscall.open.return {
    if($return >= 0)
        success[execname(), pid()] <<< 1;
}

probe timer.s($1) {
    println(ctime(gettimeofday_s()));

    printf("%12s %6s %6s %6s %s\n",
           "EXECNAME", "PID", "CREATE", "OPEN", "SUCCESS");
    foreach([en, pid+] in open) {
        printf("%12s %6d %6d %6d %d\n",
               en, pid, @count(creat[en, pid]), @count(open[en, pid]),
               @count(success[en, pid]));
    }

    delete open; delete creat; delete success;
}
```

Упражнение 3

Задание 1

В задании 1 нам потребуется на примере посмотреть какие поля структур task_struct и proc_t отвечают за разные аспекты функционирования процесса. Мы не будем приводить полный код обновленных скриптов трассировки. Отметим лишь, что в скрипте dumptask.d необходимо заменить пробу tick-1s на пару проб proc:::exec-* и proc:::exit, а в SystemTap — аналогично пробы kprocess.exec_complete и kprocess.exit используются вместо пробы timer.s(1). Выбор проб exec_complete и exec-* обусловлен тем, что до выполнения системного вызова execve() область

памяти, отвечающая за аргументы коммандной строки не инициализирована, файлы родительского процесса не закрыты, и т. д.

В результате выполнения задания у вас должны получиться следующие результаты:

- При запуске с дополнительным аргументом, так как он затирается внутри тела функции main() (так, например, поступают программы, получающие пароли в виде аргументов коммандной строки), при запуске этот аргумент отображается, а при выходе на его месте находятся символы 'X'.
- При запуске через символическую ссылку lab3-1, узел VFS, ссылающийся на образ бинарного файла: поле p_exec структуры proc_t в Solaris и поле exe_file структуры mm_struct будет указывать на регулярный файл lab3, в то же время execname будет различаться в DTrace и SystemTap.
- При запуске в chroot окружении корневая директория поменяется с / на /tmp/chroot.

Задание 2

В первую очередь в данном задании нам потребуется создать несколько ассоциативных массивов, содержащих в виде ключа pid процесса (использовать thread-clause переменные мы не можем, так как системный вызов exit() может быть вызван из любого потока), а в поле для данных — временную метку вызова. Подробности использования агрегаций мы опустим, так как они в достаточной степени были изучены в упражнении 2.

В качестве проб мы используем пробы из слайда к теме «Управление процессами». Отметим однако, что в пробах syscall.fork / syscall::fork:entry номер дочернего процесса еще не известен, поэтому нам нужно будет сохранить временную метку в промежуточный массив, а в соответствующих возвратных пробах — если возвращаемое значение больше 1 (что говорит об успешности вызова fork()) - сохранять уже в нужной агрегации используя возвращаемое значение как номер PID.

Чтобы сохранить аргументы нам бы пришлось использовать кошмарную функцию task_args() из скрипта dumptask.stp. Однако эти функциональные возможности были добавлены в SystemTap 2.5 и в пробе kprocess.exec получить список аргументов можно в переменной argstr, чем мы и воспользуемся. Для измерения времени мы задействуем local_clock_us(), а расхождением времени между процессорами пренебрежем. Наконец, чтобы сократить потребление памяти и учитывая, что количество порождаемых процессов в эксперименте будет невелико, ограничим размер ассоциативных массивов:

Листинг 40. Скрипт forktime.stp

```
global tm_fork_start_par[128], tm_fork_start[128], tm_fork_end[128],
       tm_exec_start[128], tm_exec_end[128], p_argstr[128];
```

```
global fork[128], postfork[128], exec[128], proc[128];

probe syscall.fork {
    tm_fork_start_par[tid()] = local_clock_us();
}
probe syscall.fork.return {
    if($return > 1) {
        tm_fork_start[$return] = tm_fork_start_par[tid()];
        delete tm_fork_start_par[tid()];
    }
}
probe kprocess.start {
    tm_fork_end[pid()] = local_clock_us();
}
probe kprocess.exec {
    p_argstr[pid()] = argstr;
    tm_exec_start[pid()] = local_clock_us();
}
probe kprocess.exec_complete {
    tm_exec_end[pid()] = local_clock_us();
}
probe kprocess.exit {
    argstr = p_argstr[pid()];

    fork[execname(), argstr] <<< tm_fork_end[pid()] - tm_fork_start[pid()];
    postfork[execname(), argstr] <<< tm_exec_start[pid()] - tm_fork_end[pid()];
    exec[execname(), argstr] <<< tm_exec_end[pid()] - tm_exec_start[pid()];
    proc[execname(), argstr] <<< local_clock_us() - tm_exec_end[pid()];

    delete tm_fork_start[pid()];    delete tm_fork_end[pid()];
    delete tm_exec_start[pid()];    delete tm_exec_end[pid()];
    delete p_argstr[pid()];
}

probe timer.s(1) {
    printf("%48s %8s %8s %8s %8s\n",
            "COMMAND", "FORK", "POSTFORK", "EXEC", "PROC");
    foreach([execname, args] in proc) {
        printf("%10s %36s  %6dus %6dus %6dus %6dus\n", execname, args,
                @avg(fork[execname, args]), @avg(postfork[execname, args]),
                @avg(exec[execname, args]), @avg(proc[execname, args]));
    }
    delete fork; delete postfork; delete exec; delete proc;
}
```

Тот же самый скрипт, но на DTrace. Обратите внимание, что мы использовали поле pr_psargs из транслятора psinfo_t, содержащее первый 80 символов строки аргументов.

Листинг 41. Скрипт forktime.stp

```
uint64_t tm_fork_start[int];
uint64_t tm_fork_end[int];
uint64_t tm_exec_start[int];
uint64_t tm_exec_end[int];

syscall::*fork*:entry {
    self->tm_fork_start = timestamp;
```

```
}
syscall::*fork*:return
/arg1 > 0/
{
    tm_fork_start[arg1] = self->tm_fork_start;
}

proc:::start {
    tm_fork_end[pid] = timestamp;
}
proc:::exec {
    tm_exec_start[pid] = timestamp;
}
proc:::exec-* {
    tm_exec_end[pid] = timestamp;
}

proc:::exit
/ tm_fork_start[pid] > 0 && tm_fork_end[pid] > 0 &&
  tm_exec_start[pid] > 0 && tm_exec_end[pid] > 0 /
{
    @fork[curpsinfo->pr_fname, curpsinfo->pr_psargs] =
            avg(tm_fork_end[pid] - tm_fork_start[pid]);
    @postfork[curpsinfo->pr_fname, curpsinfo->pr_psargs] =
            avg(tm_exec_start[pid] - tm_fork_end[pid]);
    @exec[curpsinfo->pr_fname, curpsinfo->pr_psargs] =
            avg(tm_exec_end[pid] - tm_exec_start[pid]);
    @proc[curpsinfo->pr_fname, curpsinfo->pr_psargs] =
            avg(timestamp - tm_exec_end[pid]);

    tm_fork_start[pid] = 0; tm_fork_end[pid] = 0;
    tm_exec_start[pid] = 0; tm_exec_end[pid] = 0;
}

tick-1s {
    normalize(@fork, 1000);     normalize(@postfork, 1000);
    normalize(@exec, 1000);     normalize(@proc, 1000);

    printf("%32s %8s %8s %8s %8s\n",
                "COMMAND", "FORK", "POSTFORK", "EXEC", "PROC");
    printa("%10s %22s %@6dus %@6dus %@6dus %@6dus\n",
                @fork, @exec, @postfork, @proc);

    clear(@fork); clear(@postfork); clear(@exec); clear(@proc);
}
```

Упражнение 4

Задание 1

Если вы когда-нибудь запускали трассировку системных вызовов нового процесса с помощью strace в Linux или truss в Solaris, вы наверное видели, что первым выполняется динамический компоновщик ld.so, который отображает в память разделяемые библиотеки (в Linux также есть отдельный системный вызов uselib для этого):

```
# 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\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)
```

Хотя соответствующие страницы памяти уже загружены в страничный кеш, записи страниц операционная система должна создать заново, так как адреса отображения могут отличаться, могут требоваться другие права доступа (конечно, это маловероятно в контексте разделяемых библиотек, однако системный вызов mmap() никак об этом не знает). Так как она это делает лениво, то есть до первого страничного сбоя, мы и будем наблюдать большое количество минорных сбоев при запуске новых процессов.

Для написания скрипта на SystemTap для Linux нам потребуется создать агрегацию с использованием функции @count, а в качестве ключа использовать объект VFS $vma->vm_file->f_path->dentry, доступный из пробы vm.pagefault:

Листинг 42. Скрипт pfstat.stp

```
#!/usr/bin/stap

global pfs;

probe vm.pagefault {
    vm_file = "???";
    if($vma->vm_file != 0)
        vm_file = d_name($vma->vm_file->f_path->dentry);

    pfs[vm_file] <<< 1;
}

probe timer.s(1) {
    printf("%8s %s\n", "FAULTS", "VMFILE");
    foreach([vm_file] in pfs) {
        printf("%8u %s\n", @count(pfs[vm_file]), vm_file);
    }

    delete pfs;
}
```

В Solaris, так же как и в оригинальном скрипте нам придется перехватывать вызовы as_segat(), чтобы определять сегмент, к которому относится страничный

сбой, и проверять принадлежность этого сегмента к драйверу segvn.

Листинг 43. Скрипт pfstat.d

```
#!/usr/sbin/dtrace -qCs

#define VNODE_NAME(vp)                                  \
    (vp) ? ((vp)->v_path)                               \
         ? stringof((vp)->v_path) : "???" : "[ anon ]"

#define IS_SEG_VN(s)  (((struct seg*) s)->s_ops == &`segvn_ops)

fbt::as_fault:entry {
    self->in_fault = 1;
}
fbt::as_fault:return {
    self->in_fault = 0;
}

fbt::as_segat:return
/self->in_fault && arg1 == 0/ {
    @faults["???"] = count();
}

fbt::as_segat:return
/self->in_fault && arg1 != 0 && IS_SEG_VN(arg1)/ {
      this->seg = (struct seg*) arg1;
      this->seg_vn = (segvn_data_t*) this->seg->s_data;

      @faults[VNODE_NAME(this->seg_vn->vp)] = count();
}

tick-1s {
      printf("%8s %s\n", "FAULTS", "VNODE");
      printa("%@8u %s\n", @faults);
      trunc(@faults);
}
```

Если запустить эти скрипты, то можно будет увидеть, что наибольшее количество сбоев будет приходиться на библиотеку libc.

Задание 2

Как видно из задания, нам потребуется получить названия кешей аллокатора, однако в документации к пробам данный параметр не описан, поэтому чтобы выполнить его, придется немного углубиться в чтение исходных кодов ядер Solaris и Linux. Так как мы ищем имя (строковой параметр), то нам придется найти описание структуры кеша аллокатора, а внутри него - массив символов char[] или указатель char*.

Обратимся к прототипу функции kmem_cache_alloc, на которую ссылается таблица из раздела Аллокатор ядра на с. 132 — первый аргумент (arg0) и есть нужная нам структура. Как видно она называется kmem_cache_t:

```
void *
kmem_cache_alloc(kmem_cache_t *cp, int kmflag)
```
(из usr/src/uts/common/os/kmem.c)

Этот тип является алиасом для типа struct kmem_cache, определенного в файле usr/src/uts/common/sys/kmem_impl.h. Внимательно просмотрев определение типа, можно найти поле cache_name, которое нам и нужно:

```
struct kmem_cache {
    [...]
    char        cache_name[KMEM_CACHE_NAMELEN + 1];
```

Тогда реализация скрипта на DTrace будет выглядеть так:

Листинг 44. Скрипт kmemstat.d

```
#!/usr/sbin/dtrace -qCs

#define CACHE_NAME(arg)     ((kmem_cache_t*) arg)->cache_name

fbt::kmem_cache_alloc:entry {
    @allocs[CACHE_NAME(arg0)] = count();
}

fbt::kmem_cache_free:entry {
    @frees[CACHE_NAME(arg0)] = count();
}

tick-1s {
    printf("%8s %8s %s\n", "ALLOCS", "FREES", "SLAB");
    printa("%@8u %@8u %s\n", @allocs, @frees);

    trunc(@allocs); trunc(@frees);
}
```

Путь к соответствующему полю в Linux несколько более тернист, так как соответствующие пробы, как следует из документации не предоставляют доступ к объекту кеша. Более того, в Linux три реализации аллокатора ядра, однако мы сразу исключим SLOB, так как она предназначена для встраиваемых систем.

Проба vm.kmem_cache_alloc должна быть определена в tapset'e — найдем соответствующую пробу-алиас в директории /usr/share/systemtap/tapset/linux/. Она ссылается или на точку трассировки kmem_cache_alloc или на функцию ядра с тем же именем. Чтобы найти точку трассировки, нужно добавить к ее имени префикс "trace" и осуществить поиск по вызовам функций — он приведет нас к двум похожим реализациям функций в аллокаторах SLAB и SLUB:

```
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;
}
```
(из mm/slab.c)

В SLUB-аллокаторе первый аргумент называется не cachep, а s, поэтому, чтобы написать скрипт, поддерживающий обе реализации, нам придется прибегнуть к конструкции @choose_defined. Как видно, в Linux для описания кеша используется структура с тем же именем. Имя кеша однако сохраняется в поле name:

```
     struct kmem_cache {
          [...]
          const char *name;
```
(из include/linux/slab_def.h)

Итоговый скрипт будет выглядеть следующим образом:

Листинг 45. Скрипт kmemstat.stp

```
#!/usr/bin/stap

global allocs, frees;

probe kernel.function("kmem_cache_alloc"),
      kernel.function("kmem_cache_alloc_node") {
    cache = @choose_defined($s, $cachep);
    name = kernel_string(@cast(cache, "struct kmem_cache")->name);

    allocs[name] <<< 1;
}

probe kernel.function("kmem_cache_free")  {
    cache = @choose_defined($s, $cachep);
    name = kernel_string(@cast(cache, "struct kmem_cache")->name);

    frees[name] <<< 1;
}

probe timer.s(1) {
    printf("%8s %8s %s\n", "ALLOCS", "FREES", "SLAB");
    foreach([cache] in allocs) {
        printf("%8u %8u %s\n", @count(allocs[cache]),
                @count(frees[cache]), cache);
    }

    delete allocs;
    delete frees;
}
```

После запуска эксперимента, можно будет увидеть что значительное количество аллокаций приходится на кеш dentry.

Упражнение 5

Сначала рассмотрим скрипт deblock.stp. Заметим, что скрипт readahead.stp полностью аналогичен, с той лишь разницей, что к нему добавляется трассировка SCSI-операций, а сам скрипт должен трассировать операции чтения, а не записи.

Для группировки данных, мы будем использовать указатель на структуру block_device. Как мы знаем, структура данных, отвечающая за файловую систему называется super_block, и она содержит указатель s_bdev, имеющий тип struct block_device*. Из этой структуры можно извлечь минорный и мажорный номер устройства с помощью функций MINOR и MAJOR, и поля bd_dev. Мы однако, воспользуемся недокументированной функцией bdevname().

Для трассировки операций на виртуальной файловой системе, воспользуемся функциями vfs_read() и vfs_write(). Они принимают указатель на файл (struct file*) в качестве первого аргумента, а также размер переданных данных в переменной count.

Уровень BIO можно трассировать с помощью tapset'a ioblock, что мы и сделаем используя пробу ioblock.request. Она имеет переменные bdev – указатель на block_device, size – размер операции BIO и rw — флаг чтения/записи. Флаг, соответственно можно сравнивать с константами BIO_READ и BIO_WRITE.

Резюмируя, получаем следующий скрипт:

Листинг 46. Скрипт deblock.stp

```
global vfstp, biotp;

probe kernel.function("vfs_write") {
    file = $file;
    if(!file) next;

    sb = @cast(file, "file")->f_path->mnt->mnt_sb;
    if(!sb) next;

    bdev = @cast(sb, "super_block")->s_bdev;
    if(bdev)
        vfstp[bdev] <<< $count;
}

probe ioblock.request {
    if(bio_rw_num(rw) != BIO_WRITE)
        next;

    biotp[bdev] <<< size;
}

probe timer.s(1) {
    printf("%12s %8s BDEV KB/s\n", "BDEV", "VFS KB/s");
    foreach([bdev] in vfstp) {
        printf("%12s %8d %d\n", bdevname(bdev),
                @sum(vfstp[bdev]) / 1024,
```

```
                @sum(biotp[bdev]) / 1024);
    }
    delete vfstp; delete biotp;
}
```

Заменим функцию vfs_write на vfs_read, флаг BIO_WRITE на BIO_READ, а аггрегации sum на агрегации count (можно также избавиться от переменных, содержащих размер операции, такие как $count и size).

Для трассировки SCSI-операций воспользуемся пробой scsi.ioentry. Выяснить, какая SCSI-команда использовалась, можно разобрав CDB-буфер, однако мы опустим эту деталь и будем трассировать все SCSI-операции. Получить имя устройства, однако не так просто – запрос типа request содержит указатели на структуры gendisk и hd_struct, однако найти соответствующий block_device затруднительно (наоборот, block_device ссылается на эти структуры). Поэтому мы воспользуемся первой операцией блочного ввода-вывода в списке bio … biotail и извлечем имя устройства аналогично тому, как это сделано в BIO-пробе.

Ниже приведен полученный скрипт:

Листинг 47. Скрипт readahead.stp

```
global vfsops, bioops, scsiops;

probe kernel.function("vfs_read") {
    file = $file;
    if(!file) next;

    sb = @cast(file, "file")->f_path->mnt->mnt_sb;
    if(!sb) next;

    bdev = @cast(sb, "super_block")->s_bdev;
    if(bdev)
        vfsops[bdev] <<< 1;
}

probe ioblock.request {
    if(bio_rw_num(rw) != BIO_READ)
        next;

    bioops[bdev] <<< 1;
}

probe scsi.ioentry {
    bio = @cast(req_addr, "struct request")->bio;
    if(!bio) next;

    bdev = @cast(bio, "bio")->bi_bdev;
    if(bdev)
        scsiops[bdev] <<< 1;
}

probe timer.s(1) {
    printf("%12s %8s %8s SCSI OP/s\n", "BDEV", "VFS OP/s", "BDEV OP/s");
```

```
    foreach([bdev] in vfsops) {
        printf("%12s %8d %8d %d\n", bdevname(bdev), @count(vfsops[bdev]),
               @count(bioops[bdev]), @count(scsiops[bdev]));
    }
    delete vfsops; delete bioops; delete scsiops;
}
```

В Solaris получить имя устройства можно с помощью транслятора devinfo_t, примененного к структуре buf. Пробы провайдера io делают это автоматически и передают соответствующую структуру в виде args[1]. Кроме этого, она содержит минорный и мажорный номера устройства. С получением имени устройства на уровне виртуальной файловой системы, однако возникают затрудения – структура vfs_t, описывающая файловую систему, содержит только номер устройства vfs_dev.

Для файловой системы ZFS определить соответствующие ей устройства в принципе затруднительно, так как в ней множеству файловых систем соответствует множество дисковых устройств, объединенных в пул. Поэтому мы воспользуется полем vfs_vnodecovered структуры vfs_t, содержащим vnode точки монтирования и извлечем путь до точки монтирования.

Трассировать операции файловой системы мы будем с помощью функций fop_read() и fop_write() которые первым аргументом принимают указатель на vnode_t файла, а вторым аргументом – структуру uio, содержающую в поле uio_resid объем передаваемых данных.

Полученный скрипт deblock.d приведен ниже:

Листинг 48. Скрипт deblock.d

```
#!/usr/sbin/dtrace -qCs

#define VFSMNTPT(vfs)   ((vfs)->vfs_vnodecovered            \
             ? stringof((vfs)->vfs_vnodecovered->v_path)    \
             : "???")
#define NBITSMINOR      32
#define MAXMIN          0xFFFFFFFF

fbt::fop_write:entry
/args[1]->uio_resid != 0/ {
    this->dev = args[0]->v_vfsp->vfs_dev;
    @vfs[getmajor(this->dev),
         getminor(this->dev),
         VFSMNTPT(args[0]->v_vfsp)] = sum(args[1]->uio_resid);
}

io:::start
/args[0]->b_bcount != 0 && args[0]->b_flags & B_WRITE/ {
    @bio[args[1]->dev_major,
         args[1]->dev_minor,
         args[1]->dev_statname] = sum(args[0]->b_bcount);
}

tick-1s {
    normalize(@vfs, 1024);  normalize(@bio, 1024);
```

```
    printf("%9s %16s %8s BDEV KB/s\n", "DEV_T", "NAME", "VFS KB/s");
    printa("%3d,%-5d %16s %8@u %@u\n", @vfs, @bio);

    trunc(@vfs); trunc(@bio);
}
```

Трассировать SCSI-стек в скрипте readahead.d мы будем с помощью пробы scsi-transport-dispatch, которая как и io:::start получает первым аргументом struct buf, который однако не обработан транслятором. Другие изменения относительно deblock.d аналогичны тем, что были сделаны в SystemTap:

Листинг 49. Скрипт readahead.d

```
#!/usr/sbin/dtrace -qCs

#define VFSMNTPT(vfs)   ((vfs)->vfs_vnodecovered            \
             ? stringof((vfs)->vfs_vnodecovered->v_path)    \
             : "???")
#define HASDI(bp)       (((struct buf*) bp)->b_dip != 0)
#define DEVINFO(bp)     xlate<devinfo_t*>((struct buf*) bp)

fbt::fop_read:entry
/args[1]->uio_resid != 0/ {
    this->dev = args[0]->v_vfsp->vfs_dev;
    @vfs[getmajor(this->dev),
         getminor(this->dev),
         VFSMNTPT(args[0]->v_vfsp)] = count();
}

io:::start
/args[0]->b_bcount != 0 && args[0]->b_flags & B_READ/ {
    @bio[args[1]->dev_major,
         args[1]->dev_minor,
         args[1]->dev_statname] = count();
}

scsi-transport-dispatch
/arg0 != 0 && HASDI(arg0)/ {
    @scsi[DEVINFO(arg0)->dev_major,
          DEVINFO(arg0)->dev_minor,
          DEVINFO(arg0)->dev_statname] = count();
}

tick-1s {
      printf("%9s %16s %8s %8s SCSI OP/s\n", "DEV_T", "NAME", "VFS OP/s", "BDEV
OP/s");
    printa("%3d,%-5d %16s %8@u %@8u %@u\n", @vfs, @bio, @scsi);

    trunc(@vfs); trunc(@bio); trunc(@scsi);
}
```

Упражнение 6

Данное упражнение незначительно отличается от любого другого примера с подсчетом задержки и сохранением ее в агрегацию.

Обратите внимание, что провайдер plockstat не предоставляет пробы для попытки захвата мьютекса. Как мы знаем, в таких случаях можно воспользоваться трассировкой по границам функций – провайдером pid$$. Нужная нам функция будет называться mutex_lock_impl – ее мы можем выявить, используя ustack() в трассировочном скрипте pthread.d.

Для вывода сохранения агрегации воспользуемся функцией quantize(), а вывода – printa(). Итоговый скрипт будет выглядеть следующим образом:

Листинг 50. Скрипт mtxtime.d

```
pid$target::mutex_lock_impl:entry
{
      self->mtxtime = timestamp;
}

plockstat$target:::mutex-acquire
/ self->mtxtime != 0 /
{
      @[ustack()] = quantize(timestamp - self->mtxtime);
      self->mtxtime = 0;
}

pid$target::experiment_unconfigure:entry
{
      printa(@);
}
```

При запуске скрипта потребуется привязать его к процессу tsexperiment через опции -c/-p.

В SystemTap нам потребуется использовать две статические пробы: mutex_entry и mutex_acquired. Однако, при создании скрипта нужно будет аккуратно использовать функции работы со стеками пользовательского процесса:

- При вызове скрипта, использовать опции -d с путем до бинарного файла или опцию --ldd.
- Так как из-за довольно частого обращения к мьютексам в нагрузчике TSLoad, вызовы ubacktrace() создают большую нагрузку, компилировать скрипт нужно с опцией -DSTP_NO_OVERLOAD. Кроме этого, можно собирать только часть стека с помощью функции ucallers(), а разрешение символов отложить до вывода агрегации.

Наконец, можно отфильтровать редко используемые мьютексы, используя функцию @count, а вывести агрегацию мы сможем с помощью функции @hist_log.

Итоговый скрипт будет иметь следующий вид:

Листинг 51. Скрипт mtxtime.stp

```
global mtxtime[128], mtxlockt;

@define libpthread %( "/lib64/libpthread.so.0" %)
@define tsexperiment %( "/opt/tsload/bin/tsexperiment" %)

probe process(@libpthread).mark("mutex_entry") {
      if(pid() != target()) next;

      mtxtime[tid()] = local_clock_ns();
}

probe process(@libpthread).mark("mutex_acquired") {
      if(pid() != target()) next;

      tm = mtxtime[tid()];
      if(tm == 0) next;

      mtxlockt[ucallers(6)] <<< local_clock_ns() - tm;
      delete mtxtime[tid()];
}

probe process(@tsexperiment).function("experiment_unconfigure") {
      foreach([ub] in mtxlockt) {
            if(@count(mtxlockt[ub]) < 100)
                  continue;

            println("-=-=-=-=-=-=-=-=-=-=-=-=-");
            print_usyms(ub);
            print(@hist_log(mtxlockt[ub]));
      }
}
```

Упражнение 7

При конфигурации тестового окружения мы использовали PHP в виде Apache-модуля (за это отвечает опция --with-apxs2 скрипта configure), поэтому весь PHP-код будет выполняться в рамках одного с Apache процесса и потока и мы можем смело использовать Thread-Local переменные. В случае PHP-FPM, это было бы затруднительно.

Таким образом, задействуем пробу process-request-entry для получения URI запроса, и пару проб function-entry/function-return, чтобы измерить время выполнения PHP-функции, а имя функции извлечем в возвратной пробе и соберем с помощью операции конкатенации. Для сбора статистики по вызовам, как всегда используем агрегации. Вместо пробы Apache process-request-entry можно использовать и пробу PHP request-startup.

Тогда скрипт на DTrace будет выглядеть подобно следующему:

Листинг 52. Скрипт topphp.d

```
pid$target::mutex_lock_impl:entry
{
      self->mtxtime = timestamp;
}

plockstat$target:::mutex-acquire
/ self->mtxtime != 0 /
{
      @[ustack()] = quantize(timestamp - self->mtxtime);
      self->mtxtime = 0;
}

pid$target::experiment_unconfigure:entry
{
      printa(@);
}
```

Скрипт на SystemTap будет выглядеть аналогичным образом:

Листинг 53. Скрипт topphp.stp

```
@define httpd %( "/usr/local/apache2/bin/httpd" %)
@define libphp5 %( "/usr/local/apache2/modules/libphp5.so" %)

global rquri, starttime, functime;

probe process(@httpd).mark("process__request__entry") {
    rquri[tid()] = user_string($arg2);
}

probe process(@libphp5).mark("function__entry") {
      starttime[tid()] = gettimeofday_ns();
}

probe process(@libphp5).mark("function__return") {
      if(starttime[tid()] == 0) next;

      func = user_string($arg4) . user_string($arg5) . user_string($arg1);
      functime[func, rquri[tid()]] <<< (gettimeofday_ns() - starttime[tid()]);
}

probe end {
      foreach([func, uri] in functime) {
            printf("%40s %32s %8d %d\n", func, uri,
                  @count(functime[func, uri]), @avg(functime[func, uri]));
      }
}
```

Приложение 2. Подготовка систем для выполнения лабораторных работ

Изначально для проведения лабораторных работ использовались виртуальные машины на основе Oracle VirtualBox с установленными операционными системами Solaris 11.0 и CentOS 6.4. Однако эти версии устарели, а сам по себе VirtualBox — не самое удачное решение, так как может порождать большое аномалии производительности, что мешает проводить эксперименты по анализу производительности.

Актуальная версия курса подготовлена для Xen 4.4 в режиме HVM и операционных систем Solaris 11.2 и CentOS 7.0. Я предполагаю, что вы уже установили эти операционные системы и выполнили базовую настройку систем: установили IP-адрес, пароль пользователя root.

Настройка CentOS 7.0

1. Как уже говорилось, в RHEL-подобных дистрибутивах требуется установка специальных debuginfo-пакетов. В CentOS они расположены в отдельном репозитории, который нужно разрешить:

```
# sed 's/^enabled=0/enabled=1/g'                    \
     /etc/yum.repos.d/CentOS-Debuginfo.repo
```

Замечание: В CentOS 7.0 поставляется некорректный GPG-ключ от debuginfo-репозитория, см. https://bugs.centos.org/view.php?id=7516, поэтому обновите пакет centos-release:
`# yum install centos-release`

2. Установите SystemTap
   `# yum install systemtap systemtap-sdt-devel systemtap-client`

3. Выполните скрипт stap-prep. Этот скрипт установит необходимые пакеты для сборки модулей ядра и debuginfo-пакеты для него:
   `# stap-prep`

Замечание: Пакет kernel-debuginfo можно установить и вручную, но надо помнить об одной особенности: если не указать пакетному менеджеру YUM, точную версию пакета, будет установлена самая свежая, которая не будет совпадать с уже установленным ядром.

4. Установите утилиту debuginfo-install:
   `# yum install yum-utils`

5. Установите debuginfo-пакеты:
   `# debuginfo-install cat python`

6. Измените точку монтирования /tmp на использование файловой системы tmpfs. Для этого добавьте в /etc/fstab следующую строчку:
   ```
   tmpfs        /tmp        tmpfs      defaults           0 0
   ```
   После чего очистите содержимое /tmp и выполните команду mount -a.
7. Сборка нагрузчика TSLoad и сопутствующих модулей.
   - Установите 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
     ```
   - Установите вспомогательные библиотеки
     ```
     # yum install libuuid-devel libcurl-devel
     ```
   - Соберите нагрузчик:
     ```
     # scons --prefix=/opt/tsload install
     ```
   - Соберите вспомогательные модули:
     ```
     # scons --with-tsload=/opt/tsload/share/tsload/devel install
     ```
8. Установите OpenJDK7:
   ```
   # yum install java-1.7.0-openjdk-devel.x86_64
   ```

Настройка Solaris 11.2

1. Сборка нагрузчика TSLoad и сопутствующих модулей.
   - Установите 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
     ```
   - Установите вспомогательные пакеты
     ```
     # pkg install pkg:/developer/gcc-45
     # pkg install pkg:/developer/build/onbld
     ```
   - Соберите нагрузчик:
     ```
     # scons --prefix=/opt/tsload install
     ```
   - Соберите вспомогательные модули:
     ```
     # scons --with-tsload=/opt/tsload/share/tsload/devel install
     ```
2. Установите JDK7:
   ```
   # pkg install --accept pkg:/developer/java/jdk-7
   ```

Настройка iSCSI

В разделе «Блочный ввод-вывод» и Упражнении 5 нам потребуется устройство, при чтении или записи которого операционная система использует стек SCSI-драйверов. Среда Xen поддерживает симуляцию SCSI-устройств, но эмулирует при этом устаревший контроллер LSI 53c895a, который не поддерживается в Solaris. Поэтому для эмуляции SCSI-устройств мы создадим

iSCSI-таргет в Dom0. Данная инструкция касается использования iSCSI Enterprise Target в Debian 7, в текущих ядрах для этого используется LIO.

1. Установите пакеты IET:
   ```
   # aptitude install iscsitarget iscsitarget-dkms
   ```
2. Создайте логические диски для виртуальных машин, например логические тома LVM – в нашем случае это будут /dev/mapper/vgmain-sol11--base--lab и /dev/mapper/vgmain-centos7--base--lab.
3. Создайте таргеты в конфигурационном файле /etc/iet/ietd.conf:
   ```
   Target iqn.2154-04.tdc.r520:storage.lab5-sol11-base
       Lun 0 Path=/dev/mapper/vgmain-sol11--base--lab,Type=blockio

   Target iqn.2154-04.tdc.r520:storage.lab5-centos7-base
       Lun 0 Path=/dev/mapper/vgmain-centos7--base--lab,Type=blockio
   ```
   Заметим, что имена таргетов должны соответствовать DNS-домену хоста, их предоставляющему.
4. Чтобы Solaris не имел доступ к LUN'у предоставляемому CentOS, и наоборот, сконфигурируйте файл /etc/iet/initiators.allow. Удалите или закомментируйте строчку «ALL ALL» и добавьте строчки с именами таргетов и IP-адресами виртуальных машин:
   ```
   iqn.2154-04.tdc.r520:storage.lab5-sol11-base    192.168.50.179
   iqn.2154-04.tdc.r520:storage.lab5-centos7-base  192.168.50.171
   ```
5. Перезапустите демон IET:
   ```
   # /etc/init.d/iscsitarget restart
   ```
6. Сконфигурируйте инициатор на Solaris. Здесь 192.168.50.116 — IP-адрес домена 0.
   ```
   # iscsiadm add discovery-address 192.168.50.116
   # iscsiadm modify discovery -t enable
   # svcadm restart svc:/network/iscsi/initiator:default
   ```
7. Аналогично, сконфигурируйте инициатор на CentOS:
   ```
   # 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
   ```

Установка и настройка Web-стека

Для последнего пункта нашего курса нам потребуется установить следующий набор приложений: веб-сервер Apache HTTPD 2.4, реляционную СУБД MySQL Community Edition 5.6 и интерпретатор языка PHP 5.6, поверх которых мы развернем систему управления контентом Drupal 7.

Данные инструкции подходят и для CentOS 7 и для Solaris 11 кроме некоторых команд, для которых будет сделана соответствующая пометка. Нам потребуется собрать программы из исходных кодов, чтобы включить в них поддержку трассировки с помощью USDT.

I. Сначала скачайте дистрибутивы упомянутых программ.

1. Скачайте их с помощью утилиты 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
```

2. Также нам потребуется патч для системы сборки Apache HTTPD:

```
# wget -O dtrace.patch \
  https://bz.apache.org/bugzilla/attachment.cgi?id=31665
```

3. (*Только CentOS7*) Удалите имеющиеся в стандартной поставке программы и установите некоторые зависимости:

```
# yum erase php mysql mysql-server httpd
# yum install libxml2-devel bzip2
```

4. (*Только Solaris*) Во всех вызовах make нам нужно будет использовать GNU Make. Для этого можно создать алиас:

```
# alias make=gmake
```

5. Распакуйте скачанные архивы

```
# 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
```

II. Соберите и установите MySQL из исходных кодов:

1. Перейдите в директорию с исходными кодами:

```
# cd mysql-5.6.25
```

2. (*Только CentOS7*) Установите зависимости:

```
# yum install cmake bison ncurses-devel gcc-c++
```

3. (*Только Solaris*) Установите зависимости:

```
# pkg install pkg:/developer/build/cmake
# pkg install pkg:/developer/parser/bison
```

4. Соберите и установите MySQL (будет установлен в /usr/local/mysql)

```
# cmake --enable-dtrace .
# make -j4
# make install
```

III. Соберите и установите Apache HTTPD из исходных кодов:

1. Перейдите в директорию с исходными кодами:

```
# cd ../httpd-2.4.9
```

2. (*Только CentOS7*) Установите зависимости:

```
# yum install pcre-devel autoconf flex patch
```

3. (*Только Solaris*) Установите зависимости:

```
# pkg install pkg:/developer/build/autoconf
# pkg install pkg:/developer/macro/gnu-m4
# pkg install pkg:/developer/lexer/flex
```

4. Наложите патч на систему сборки и пересоздайте configure-скрипт:

```
# patch -p0 < ../dtrace.patch
# autoconf
```

5. Создайте файл для сборки MPM:

```
# (cd server/mpm/event/ &&
   echo "$(pwd)/event.o $(pwd)/fdqueue.o" > libevent.objects)
```

6. Исправьте файл server/Makefile.in:

```
# sed -i 's/apache_probes.h/"apache_.*probes.h"/' server/Makefile.in
```

7. Соберите и установите MySQL (будет установлен в /usr/local/apache2):

```
# ./configure --with-included-apr --enable-dtrace
# make -j4
# make install
```

IV. Соберите и установите PHP из исходных кодов:

1. Перейдите в директорию с исходными кодами:

```
# cd php-5.6.10
```

2. (*Только CentOS7*) Установите зависимости:

```
# yum install libjpeg-turbo-devel libpng12-devel
```

3. (*Только Solaris*) Установите зависимости:

```
# pkg install pkg:/system/library/gcc-45-runtime
```

4. Соберите и установите 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 -j4
# make install
```

V. Настройте базу данных MySQL:

1. Перейдите в соответствующую директорию:

```
# cd /usr/local/mysql
```

2. (*Только CentOS7*) Создайте пользователя mysql:

```
# groupadd -g 666 mysql
# useradd -u 666 -g mysql -d /usr/local/mysql/home/ -m mysql
```

3. Создайте файлы данных:

```
# chown -R mysql:mysql data/
# ./scripts/mysql_install_db
```

4. Создайте конфигурационный файл /etc/my.cnf:

```
# cat > /etc/my.cnf << EOF
[mysqld]
datadir=/usr/local/mysql/data
socket=/tmp/mysql.sock
user=mysql
# Disabling symbolic-links is recommended to prevent assorted security
risks
symbolic-links=0
bind-address=0.0.0.0

[mysqld_safe]
log-error=/var/log/mysqld.log
pid-file=/var/run/mysqld/mysqld.pid
EOF
```

5. Создайте файл логов и директорию для PID-файла:

```
# touch /var/log/mysqld.log &&
  chown mysql:mysql /var/log/mysqld.log
# mkdir /var/run/mysqld/ &&
  chown mysql:mysql /var/run/mysqld/
```

6. Заполните системные базы данных:

```
# chown -R mysql:mysql data/
# ./scripts/mysql_install_db --ldata=/usr/local/mysql/data
```

7. Запустите демон mysqld:

```
# ./support-files/mysql.server start
```

8. Задайте пароль администратора MySQL (changeme)

```
# /usr/local/mysql/bin/mysqladmin -u root password changeme
```

9. Создайте пользователя drupal

```
# /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;
```

VI. Настройте Apache и PHP

1. (*Только CentOS7*) отключите брандмауэр:

```
# systemctl disable firewalld
# systemctl stop firewalld
```

2. Перейдите в директорию Apache HTTPD:

```
# cd /usr/local/apache2
```

3. Сделайте копию конфигурационного файла и добавьте в него поддержку PHP 5 (в Solaris надо использовать gsed вместо sed):

```
# 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 <<EOF
# Use for PHP 5.x:
LoadModule php5_module        modules/libphp5.so
AddHandler php5-script  .php
EOF
```

4. Запустите Apache HTTPD:

```
# ./bin/httpd
```

VII. Установите Drupal 7:

1. Перейдите в директорию документов веб-сервера:

```
# cd /usr/local/apache2/htdocs
```

2. Скачайте и распакуйте 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 .
```

3. Создайте базу данных «drupal»:

```
# /usr/local/mysql/bin/mysql --user=drupal -p -h localhost
Enter password: password
mysql> CREATE DATABASE drupal;
```

4. Перейдите по адресу http://АДРЕС СЕРВЕРА/drupal/install.php в веб браузере и следуйте указаниям инсталлятора Drupal. При настройке базы данных используйте следующие данные:

- Database name: drupal
- Database username: drupal
- Database password: password

VIII. Установите модуль Devel и сгенерируйте тестовые данные

1. Скачайте и установите его:

```
# 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
```

2. Перейдите на главную страницу Drupal, в верхнем меню выберите Modules, и в выведенном списке внизу, установите галочки напротив пунктов "Devel" и "Devel generate", а затем нажмите на "Save Configuration".
3. После того, как модули будут включены, выберите "Configure" напротив модуля "Devel generate", в предложенном меню выберите"Generate content", выберите галочку "Article" и нажмите "Generate". На главной странице должны появиться 50 тестовых страниц.

Замечание: Для запуска после перезагрузки используйте следующие две команды:

```
# /usr/local/mysql/support-files/mysql.server start
# /usr/local/apache2/bin/httpd
```

Данная инструкция предназначена только для настройки лабораторного окружения. Конфигурация может быть недостаточно безопасной, поэтому не используйте ее для настройки продуктивных систем!