# Введение в операционные системы: практический подход в рамках проекта OpenSolaris

Практическое руководство


Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

L'interface d'utilisation graphique OPEN LOOK et Sun a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui, en outre, se conforment aux licences écrites de Sun.

Les produits qui font l'objet de cette publication et les informations qu'il contient sont régis par la legislation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes chimiques ou biologiques ou pour le nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des Etats-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la legislation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement designés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.

# Содержание

◆ ◆ ◆ МОДУЛЬ 1

# 1 Введение

## Цели

Цель этого курса состоит в изучении построения операционных систем с использованием исходного кода операционной системы Solaris™, который можно получить бесплатно в рамках проекта OpenSolaris.

---

**Совет –** Для получения стартового комплекта OpenSolaris Starter Kit, включающего в себя учебные материалы, исходный код и инструменты разработки, зарегистрируйтесь по адресу `https://opensolaris.org/register.jspa`.

---

Для начала рассмотрим, где можно получить доступ к исходному коду, сообществам, дискуссиям, проектам и браузеру исходного кода для проекта OpenSolaris. Затем кратко опишем, как функции ОС Solaris изменили технологии операционных систем, и продемонстрируем две наиболее революционные технологии на практических примерах:

- Настройка зон
  - Администрирование зон
  - Сетевые соединения зон
  - Идентификация зон, видимость ЦП и пакеты
  - Создание, установка и начальная загрузка новой зоны
- Виртуализация веб-серверов при помощи зон
  - Создание двух неглобальных зон

- Настройка файловых систем с ZFS
  - Создание зеркальных пулов устройств хранения данных ZFS
  - Создание файловой системы и каталогов `/home`
  - Настройка RAID-Z

После этого опишем процесс разработки OpenSolaris и компоненты среды, а затем обратимся к дополнительным ресурсам для установки. Наконец, рассмотрим следующие практические примеры, предназначенные для демонстрации типичных проблем операционных систем при помощи OpenSolaris:

- Отладка процессов
  - Включение простых датчиков DTrace
  - Вывод списка отслеживаемых датчиков
  - Программирование в D
  - Включение датчиков DTrace режима пользователя
- Отладка приложений
  - Трассировка приложений при помощи DTrace
  - Использование DTrace для профилирования и отладки программы C++
- Управление памятью
  - Использование DTrace и MDB для исследования виртуальной памяти
- Наблюдение за процессами
  - Трассировка процесса, выполняющегося в зоне, при помощи DTrace

# Благодарности

Ниже перечислены лидеры сообщества Documentation Community, которые проанализировали вторую версию настоящего документа и представили свои предложения и замечания по ней:

- Ben Rockwood

- Rainer Heilke
- Eric Lowe

Следующие инженеры Sun предоставили ценную новую информацию:

- Narayana Janga
- Shivani Khosa

Также отдельное спасибо Steven Cogorno, David Comay, Teresa Giacomini, Stephen Hahn, Patrick Finch и Sue Weber за их работу над первоначальной версией.

Для участия в подготовке следующих версий этого документа см.:

`http://www.opensolaris.org/os/community/documentation/reviews`

◆ ◆ ◆ МОДУЛЬ 2

# Что такое проект OpenSolaris?

## Цели

Проект OpenSolaris начался 14 июня 2005 года и имел своей целью публичную разработку с использованием в качестве отправной точки кода операционной системы Solaris™. Этот проект представляет собой базис для совместной разработки, где участники – как из компании Sun, так и независимые – могут сотрудничать в целях разработки и усовершенствования технологии операционной системы. Исходный код OpenSolaris находит самые разнообразные сферы применения, например, является основой для будущих версий продукта ОС Solaris, других операционных систем, сторонних продуктов и дистрибутивов, представляющих интерес для сообщества. Проект OpenSolaris в настоящее время спонсируется Sun Microsystems, Inc.

За первый год зарегистрировались более 16 000 участников. Сообщество инженеров непрерывно растет и меняется согласно требованиям разработчиков, системных администраторов и конечных пользователей операционной системы Solaris.

Обучение в рамках проекта OpenSolaris имеет следующие преимущества по сравнению с учебными операционными системами:

- Доступ к коду революционных технологий операционной системы Solaris 10
- Доступ к коду для создания коммерческой ОС с возможностью ее использования в различных средах и масштабирования до более крупных систем

- Превосходные инструменты для наблюдения и отладки
- Поддержка аппаратных платформ, в том числе архитектур SPARC, x86 и AMD x64
- Ведущая позиция в 64-разрядных вычислениях
- Неограниченное право пользования за 0,00 у.е.
- Доступная, увлекательная, инновационная, полная, единая, надежная база кода
- Доступность под одобренной OSI Общей лицензией разработки и распространения (CommonDevelopment and Distribution License, CDDL) означает свободу использования, модификации и производных действий.

# Веб-ресурсы по OpenSolaris

Загрузить исходный код OpenSolaris, просмотреть условия лицензии и инструкции по доступу для компоновки исходного кода и установки предварительно собранных архивов можно по адресу: `http://www.opensolaris.org/os/downloads`.

Значки в верхней правой части веб-страниц OpenSolaris являются ссылками на ресурсы дискуссий, сообществ, проектов, загрузки и браузера исходного кода.

Кроме того, на веб-сайте OpenSolaris предусмотрены функции поиска по всему содержимому сайта и агрегированным блогам.

## Обсуждение

Дискуссии позволяют связаться с экспертами, работающими над новыми технологиями открытого кода. Здесь также можно просмотреть архив прежних обсуждений, в котором можно найти ответы на возникающие вопросы. Полный список форумов, на которые можно подписаться, см. на `http://www.opensolaris.org/os/discussions`.

# Сообщества

Сообщества позволяют поддерживать связь с другими участниками проекта OpenSolaris с похожими интересами. Сообщества формируются вокруг групп по интересам, технологий, поддержки, инструментальных средств и групп пользователей, например:

| | |
|---|---|
| Наука и исследования | http://www.opensolaris.org/os/community/edu |
| DTrace | http://www.opensolaris.org/os/community/dtrace |
| ZFS | http://www.opensolaris.org/os/community/zfs |
| Зоны | http://www.opensolaris.org/os/community/zones |
| Документация | http://www.opensolaris.org/os/community/documentation |
| Драйверы устройств | http://www.opensolaris.org/os/community/device_drivers |
| Инструментальные средства | http://www.opensolaris.org/os/community/device_drivers |
| Сообщества пользователей | http://www.opensolaris.org/os/community/os_user_groups |
| Безопасность | http://www.opensolaris.org/os/community/security |
| Производительность | http://www.opensolaris.org/os/community/performance |
| Системные администраторы | http://www.opensolaris.org/os/community/sysadmin |

Только некоторые из 40 сообществ активно работают над OpenSolaris. Полный список см. по адресу http://www.opensolaris.org/os/communities.

# Проекты

Проекты, расположенные на сайте http://www.opensolaris.org/, представляют собой совместные начинания, в результате которых на свет появляются изменения кода, документы, графика, продукты совместного авторства и пр. Проекты имеют свои репозитарии кода и объединяют различных участников; кроме того, они могут организовываться независимо или в рамках какого-либо сообщества.

Новые проекты инициируются участниками по запросу в дискуссиях. Проектам, которые были объявлены и поддержаны еще хотя бы одним заинтересованным участником, выделяется пространство на странице проектов для начала работы. Текущий список новых проектов см. по адресу http://www.opensolaris.org/os/projects.

# OpenGrok

OpenGrok™ – быстрый и удобный инструмент поиска исходного кода и перекрестных ссылок, используемый в OpenSolaris. Опробуйте его по адресу http://cvs.opensolaris.org/source!

OpenGrok стал первым проектом, размещенным на opensolaris.org. Информацию об этом продолжающемся проекте по разработке см. по адресу http://www.opensolaris.org/os/project/opengrok.

Совершите интерактивную экскурсию по исходному коду и ознакомьтесь с аккуратно написанным, тщательно откомментированным кодом, который можно читать как книгу.

Если планируется работать над каким-либо проектом OpenSolaris, можно загрузить полную кодовую базу. Если требуется лишь выяснить, как работают те или иные функции ОС Solaris, удобной альтернативой может служить браузер исходного кода. OpenGrok распознает различные форматы файлов программ и методы управления версиями, такие как SCCS, RCS и CVS, что позволяет лучше понять открытый код.

◆ ◆ ◆ МОДУЛЬ 3

# Функциональные возможности ОС Solaris

## Цели

В этом разделе описываются основные функции ОС Solaris и фундаментальные изменения, которые эти функции внесли в технологии операционных систем.

## Обзор

Теперь, после рассмотрения компонентов, процессов и руководящих принципов разработки OpenSolaris, можно кратко обсудить следующие функциональные возможности операционной системы:

- Технология безопасности: минимальный приоритет
- Прогнозируемое самовосстановление
- Средство управления службами (SMF)
- Зоны
- Маркированные зоны (BrandZ)
- Секстибайтная файловая система (ZFS)
- Средство динамической трассировки (DTrace)
- Модульный отладчик (MDB)

# Технология безопасности: минимальный приоритет

В UNIX® исторически использовалась бескомпромиссная модель полномочий, налагающая следующие ограничения:

- отсутствует способ ограничения полномочий пользователя `root`;
- отсутствует способ выполнения привилегированных операций некорневыми пользователями;
- приложения, требующие незначительного количества привилегированных операций, приходится запускать от имени `root`;
- очень немногие пользователи обладают полномочиями root, в числе таких пользователей практически отсутствуют студенты.

В ОС Solaris реализована *детализация* полномочий. Детализация полномочий позволяет пользователям запускать приложения с минимально необходимыми полномочиями. *Наименьшие* полномочия позволяют студентам получить права, необходимые для выполнения курсовой работы, участия в исследованиях или поддержки части инфраструктуры университетского городка или отдела.

# Прогнозируемое самовосстановление

В ОС Solaris 10 прогнозируемое самовосстановление реализовано двумя способами. В этом разделе описывается новая архитектура управления устранением отказов и средство управления службами, которые вместе составляют технологию самовосстановления.

## Архитектура управления устранением отказов (FMA)

ОС Solaris содержит новую архитектуру, FMA, которая служит для построения надежных обработчиков ошибок, телеметрии ошибок, ПО автоматизированной диагностики, реакционных агентов и последовательной модели обработки системных ошибок для стека управления. Многие компоненты Solaris уже участвуют в FMA, включая обработку ошибок процессора и памяти для UltraSPARC III и IV, UltraSPARC PCI HBA, и проч. Поддержка Opteron планируется для сборки 34. В разработке находится ряд проектов, в том числе полная поддержка ошибок процессора, памяти и ввода/вывода на Opteron, преобразование ряда ключевых драйверов устройств и интеграция с различными стеками управления.

В управление устранением отказов интегрируется каждая подсистема; обработка ошибок становится более надежной, что позволяет системе продолжать работу, несмотря на имеющуюся ошибку. Для управления автоматизированной диагностикой и реагированием используются события телеметрии. Инструментальные средства и архитектура управления устранением отказов позволяют разрабатывать самовосстанавливающиеся программные и аппаратные средства как для микроскопических, так и для макроскопических системных ресурсов и предоставляют унифицированные данные администраторам и программам управления системой.

Для получения информации об участии в сообществе по управлению устранением отказов или для загрузки MIB управления устранением отказов, находящейся в настоящее время в разработке, см.
`http://opensolaris.org/os/community/fm`.

## Средство управления службами (SMF)

SMF создает в проекте OpenSolaris поддерживаемую унифицированную модель для управления огромным количеством служб, таких как доставка электронной почты,

запросы `ftp` и удаленное выполнение команд. Структура `smf(5)` замещает (совместимым способом) существующий механизм запуска `init.d(4)` и включает в себя расширенный `inetd(1M)`, в результате чего служба превращается в объект операционной системы первого класса. SMF дает разработчикам следующие возможности:

- автоматизированный перезапуск служб с учетом их взаимозависимости при возникновении ошибок администрирования, ошибок в программе или неисправимых аппаратных ошибок;
- унифицированный API для управления службами, конфигурирования и наблюдения;
- доступ к управлению ресурсами на основе служб;
- упрощенная отладка процесса начальной загрузки.

Ознакомиться со схемой служб SMF и их взаимосвязью на платформе x86 с установленной ОС Solaris Nevada build 24 можно по адресу `http://opensolaris.org/os/community/smf/scfdot`.

Кроме усовершенствований в управлении на уровне служб, в проекте OpenSolaris предусмотрены функции уровня приложений и функциональность для создания разделенных и защищенных сред выполнения. Мощные средства управления ресурсами зон – это решение для тех уникальных задач, которые возникают при разработке приложений и их тестировании в среде совместного пользования.

# Зоны

Зона – абстракция виртуальной операционной системы, обеспечивающая защищенную среду выполнения приложений. Защита приложений друг от друга позволяет добиться программной изоляции отказов. В целях упрощения задачи управления множеством приложений и их сред они сосуществуют в одном экземпляре операционной системы и обычно обслуживаются как одна сущность.

Каждая зона имеет свои собственные характеристики, например, имя зоны, IP-адреса, имя хоста, службы имен, корневые и некорневые пользователи. По умолчанию ОС работает в глобальной зоне. Администратор может виртуализировать среду выполнения путем определения одной или нескольких неглобальных зон. Для ограничения ущерба в случае нарушения безопасности можно запустить соответствующие сетевые службы. Так как зоны реализованы программно, они не подвержены ограничениям по гранулярности, накладываемыми аппаратными средствами. Вместо этого зоны обеспечивают гранулярность на уровне более низком, чем уровень процессора.

Зоны можно объединить со средствами управления ресурсами, имеющимися в OpenSolaris, с целью получения более полных, изолированных сред. В такой схеме зона обеспечивает безопасность, пространство имен и изоляцию отказов, а средства управления ресурсами предотвращают использование процессами из одной зоны слишком большой доли системных ресурсов или гарантируют определенный уровень обслуживания для этих процессов. Зоны и управление ресурсами вместе часто называются контейнерами.

Ответы на большое количество часто возникающих вопросов о зонах и ссылки на самую последнюю документацию по администрированию см. на `http://opensolaris.org/os/community/zones/faq`.

Зоны обеспечивают защищенные среды для приложений Solaris.Благодаря BrandZ в проекте OpenSolaris доступны раздельные и защищенные среды выполнения.

# Маркированные зоны (BrandZ)

BrandZ представляет собой платформу расширения инфраструктуры зон, позволяющую создавать маркированные зоны, т. е. зоны, содержащие не собственные рабочие среды. Маркированная зона может быть просто средой, в которой стандартные утилиты Solaris заменены их эквивалентами GNU,

или, в более сложном случае, содержать полноценное пространство пользователя Linux.

Маркировка `lx` позволяет запускать в Solaris неизмененные двоичные приложения Linux внутри зон, в которых реализуется полное пространство пользователя Linux. Маркировка `lx` позволяет пользовательскому программному обеспечению Linux работать на компьютере с ядром OpenSolaris и включает в себя инструменты, необходимые для установки дистрибутива CentOS или Red Hat Enterprise Linux внутри зоны в рамках системы Solaris. Маркировка `lx` может применяться на компьютерах x86/x64 с 32-разрядным или 64-разрядным ядром. Независимо от используемого ядра, возможно выполнение только 32-разрядных приложений Linux. Эта функция в настоящее время доступна только для архитектур x86 и AMD x64. Однако перенесение на SPARC может оказаться интересным проектом сообщества, поскольку BrandZ `lx` все еще находится в активной разработке.

Для изучения требований и инструкций по установке см. `http://opensolaris.org/os/community/brandz/install`.

Проект OpenSolaris позволяет найти решения для тех уникальных задач, которые возникают при разработке операционных систем и тестировании работы приложений с использованием таких функциональных возможностей, как зоны. Более того, код ZFS в проекте OpenSolaris упрощает сегментирование файловых систем для разработки ядра.

# Секстибайтная файловая система (ZFS)

Файловые системы ZFS не зависят от ограничений, накладываемых отдельными устройствами, поэтому подобно каталогам, их можно создавать просто и быстро. Они автоматически наращиваются внутри пространства, выделенного для пула хранения данных.

ZFS представляет модель хранения данных по принципу пула с отказом от концепции томов и устраняет типичные проблемы разделов, подготовки, ненадлежащего использования пропускной способности и неорганизованного хранения информации.

Для всех файловых систем постоянно доступна объединенная полоса пропускания для операций ввода/вывода.

Каждый пул хранения состоит из одного или нескольких виртуальных устройств, которые описывают структуру физического уровня и ее свойства с точки зрения возможных отказов. Для ознакомления с демонстрацией администрирования зеркальных пулов с ZFS см. *100 Mirrored Filesystems in 5 Minutes* по адресу `http://opensolaris.org/os/community/zfs/demos/basics`.

Кроме размещения по принципу пула, ZFS обеспечивает резервирование данных RAID-Z по принципу избыточности. RAID-Z представляет собой виртуальное устройство, которое хранит данные и информацию о четности на нескольких дисках, подобно RAID-5.

В RAID-Z ZFS использует полосы RAID переменной ширины и обеспечивает запись на всю полосу. Это возможно исключительно благодаря интеграции в ZFS управления файловыми системами и устройствами, при которой метаданные файловой системы содержат достаточно информации о базовой модели репликации данных для обработки полос RAID переменной ширины. RAID-Z – первое в мире чисто программное решение проблемы дыры при записи RAID-5.

# Динамическая трассировка (DTrace)

DTrace обеспечивает мощную инфраструктуру, позволяющую администраторам, разработчикам и сервисному персоналу

давать краткие ответы на любые вопросы о поведении операционной системы и пользовательских программ. DTrace предоставляет следующие возможности:

- Это достигается за счет тысяч динамически активируемых и управляемых "датчиков".
- Предикаты и действия динамически связываются с датчиками.
- Становится возможным динамическое управление буферами трассировки и служебной информацией датчиков.
- Данные трассировки для изучения можно взять с работающей системы или из дампа полного отказа системы.
- К DTrace подключаются новые провайдеры данных трассировки.
- Внедряются потребители данных трассировки, обеспечивающие отображение данных.
- Добавляются инструменты конфигурирования датчиков DTrace.

Страницы сообщества DTrace:
`http://opensolaris.org/os/community/dtrace`.

Кроме DTrace, в рамках проекта OpenSolaris предоставляются средства отладки для разработки низкоуровневого типа, например, драйверов устройств.

# Модульный отладчик (MDB)

MDB – это отладчик, предназначенный для упрощения анализа проблем, диагностика и устранение которых требуют низкоуровневых средств отладки, исследования файлов ядра и знания ассемблера. Как правило, разработчики ядра и устройств используют `mdb` для определения места и причины некорректной работы кода.

Доступ к MDB осуществляется посредством двух команд, имеющих общие функции: `mdb` и `kmdb`. Команда `mdb` используется в интерактивном режиме или в скриптах для отладки действующих пользовательских процессов, файлов ядра пользовательских процессов, аварийных дампов ядра, действующей операционной системы, объектных и прочих файлов. Команда `kmdb` используется для отладки действующего ядра системы и драйверов устройств, когда также требуется контролировать и приостанавливать выполнение ядра.

Существует активное сообщество по MDB, где можно задать вопросы экспертам или ознакомиться с предыдущими дискуссиями и часто задаваемыми вопросами. См. `http://opensolaris.org/os/community/mdb`.

◆ ◆ ◆ МОДУЛЬ 4

# Настройка зон

## Цели

Цель этого модуля состоит в ознакомлении с более сложными понятиями, связанными с зонами, а также в демонстрации настройки, установки и начальной загрузки новой зоны. Также будет продемонстрирована виртуализация веб-сервера с использованием двух неглобальных зон.

## Краткое описание зон

Зону можно представить в виде контейнера, в котором одно или несколько приложений запускается изолированно от всех остальных приложений на компьютере. Большую часть программного обеспечения, работающего под управлением OpenSolaris, можно запускать в зоне без дополнительных изменений. Так как зоны не приводят к изменению интерфейса прикладного программирования (API) OpenSolaris или двоичного интерфейса приложений (ABI), для запуска приложения в зоне не требуется перекомпиляция приложения.

Небольшое количество приложений, которые обычно выполняются от имени `root` или с особыми полномочиями, могут перестать работать внутри зоны, если они полагаются на возможность обращения к какому-либо глобальному ресурсу или его изменения. В качестве примера можно привести возможность изменения системного времени. В случае приложений из этой

немногочисленной категории для корректного выполнения в зоне может потребоваться изменение или, в некоторых случаях, выполнение в глобальной зоне.

Ниже приведен ряд основных принципов:

- Приложения, осуществляющие доступ к сети и файлам и не выполняющие других операций ввода/вывода, должны работать корректно.
- Приложения, требующие прямого доступа к определенным устройствам (например, разделам диска), как правило, будут работать при условии правильной настройки зоны. Однако в некоторых случаях при этом может возрасти угроза безопасности.
- Для корректной работы приложений, требующих прямого доступа к этим устройствам, может потребоваться модификация этих приложений. Например, это может быть `/dev/kmem` или сетевое устройство. Вместо этого приложениям рекомендуется использовать одну из многих IP-служб.

BrandZ расширяет инфраструктуру зон в пространстве пользователя следующими способами:

- Во время настройки зоны ей назначается специальный атрибут, т.н. "бренд".
- Каждый бренд предусматривает собственную программу установки, которая позволяет устанавливать в маркированной зоне произвольный набор программного обеспечения.
- Каждый бренд может обеспечивать скрипты, выполняющиеся до и после загрузки, которые позволяют выполнять любую настройку или конфигурирование во время загрузки.
- Для определения и вывода типа бренда зоны можно воспользоваться утилитами `zonecfg` и `zoneadm`.

BrandZ обеспечивает ряд точек вставки в ядре:

- Эти точки находятся по пути `syscall`, пути загрузки процессов, пути создания потока и т.д.
- Эти точки вставки относятся только к процессам в маркированной зоне.
- В каждой из этих точек для бренда может быть выбрано дополнение или замещение стандартного поведения ОС Solaris.
- Для фундаментально различающихся брендов могут потребоваться новые точки вставки.

# Администрирование зон

Администрирование зон реализуется следующими командами:

- `zonecfg` – создание зон, настройка зон (добавление ресурсов и свойств). Конфигурация сохраняется в закрытом файле XML в каталоге `/etc/zones`.
- `zoneadm` – выполняет административные шаги для зон, такие как list, install, (re)boot и halt.
- `zlogin` – разрешает пользователю регистрироваться в зоне для выполнения задач обслуживания.
- `zonename` – отображает текущее имя зоны.

С зонами используются следующие глобальные свойства:

- `zonepath` – путь в глобальной зоне к корневому каталогу, под которым будет установлена зона.
- `autoboot` – необходимость загрузки при начальной загрузке глобальной зоны.
- `pool` – пулы ресурсов, к которым должны быть привязаны зоны.

  Ресурсы могут относиться к любому из следующих типов:

  - `fs` – файловая система;
  - `Inherit-pkg-dir` – каталог, связанные пакеты которого наследуются от глобальной зоны;

- `net` – сетевое устройство;
- `device` – устройства.

# Сетевые соединения зон

Для системы используется единственный стек TCP/IP, поэтому зоны экранированы от подробных конфигурационных сообщений устройств, маршрутизации и т. д. Каждой зоне можно назначить адреса IPv4/IPv6 и собственное портовое пространство. Приложения могут привязываться к INADDR_ANY и получать трафик только данной зоны. Трафик других зон при этом распознаваться не будет.

Исходящие пакеты зоны имеют адрес источника, принадлежащий этой зоне. Передача пакетов из зоны может осуществляться только через интерфейс, на котором определен адрес этой зоны. Зона может использовать маршрутизатор по умолчанию, только если он непосредственно достижим из этой зоны. Маршрутизатор по умолчанию должен находиться в той же самой IP-подсети, что и зона.

Зоны не могут изменить свою конфигурацию сети или таблицу маршрутизации и не могут видеть конфигурацию сети других зон. Каталог `/dev/ip` в зоне отсутствует. Вместо этого агенты SNMP должны открывать `/dev/arp`. Несколько зон могут совместно использовать широковещательный адрес, а также вступать в одну широковещательную группу.

На зоны накладываются следующие ограничения в отношении сетевых соединений:

- нельзя размещать физический интерфейс внутри зоны;
- IPFilter не работает между зонами;
- DHCP для IP-адресов зон не поддерживается;
- отсутствует динамическая маршрутизация.

# Идентификация зон, видимость ЦП и пакеты

Каждая зона управляет своим именем узла, часовым поясом и службами имен, такими как LDAP и NIS. Эти параметры можно настроить командой `sysidtool`. Раздельные файлы `/etc/passwd` означают, что зоне можно делегировать полномочия `root`. Идентификаторы пользователей могут соответствовать разным именам в разных доменах.

По умолчанию все ЦП видны всем зонам. При использовании пулов ресурсов автоматически активируется режим ограниченной видимости.

В зонах могут добавляться собственные пакеты. К этим пакетам можно применять патчи. Системные патчи применяются в глобальной зоне. После этого в неглобальных зонах автоматически выполняется начальная загрузка с опцией -s для применения патча. Необходимо сохранять непротиворечивость пакета SUNW_PKG_ALLZONES между глобальной зоной и всеми неглобальными зонами. Параметром SUNW_PKG_HOLLOW определяется наличие имени пакета в неглобальных зонах (NGZ) в целях выявления зависимостей, однако его содержимое не устанавливается.

# Устройства зон

В каждой зоне имеются собственные устройства. Зонам видно подмножество *защищенных* псевдоустройств в их каталогах `/dev`. Приложения ссылаются на логический путь устройства, содержащийся в `/dev`. Каталог `/dev`, в отличие от каталога `/devices`, существует и в неглобальных зонах. Такие устройства, как random, console и null, являются защищенными, в то время как другие, например, `/dev/ip`, не защищены.

Зоны могут изменять полномочия своих устройств, но не могут выдавать `mknod(2)`. Файлы физических устройств, подобные дискам с доступом на физическом уровне, можно помещать в

зону с соблюдением определенных предосторожностей. Устройства могут совместно использоваться разными зонами, однако в таких случаях следует тщательно продумать способы обеспечения безопасности.

Например, может потребоваться назначить некоторые устройства определенным зонам. Разрешение доступа к блочным устройствам непривилегированным пользователям может привести к тому, что с этих устройств можно будет вызвать панику системы, сброс шины или другие нежелательные действия. Помещение физического устройства в несколько зон может привести к созданию скрытого канала между зонами. Приложения глобальной зоны, использующие такое устройство, испытывают риск компрометации данных или нарушения целостности данных неглобальной зоной.

# Введение в администрирование зон

Этот практический пример представляет собой введение в создание зон.

## Аннотация

В этом упражнении приведены подробные примеры по созданию, установке и начальной загрузке зоны.

---

**Примечание –** Эта процедура не относится к маркированной зоне lx.

---

# Создание, установка и начальная загрузка зоны

1 **Для конфигурирования новой зоны можно использовать следующий пример:**

```
# zonecfg -z Apache
Apache: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:Apache> create
zonecfg:Apache> set zonepath=/export/home/Apache
zonecfg:Apache> add net
zonecfg:Apache:net> set address=192.168.0.50
zonecfg:Apache:net> set physical=bge0
zonecfg:Apache:net> end
zonecfg:Apache> verify
zonecfg:Apache> commit
zonecfg:Apache> exit
```

2 **Для установки и начальной загрузки новой зоны можно использовать следующий пример:**

```
# zoneadm -z Apache install
Preparing to install zone <Apache>.
Creating list of files to copy from the global zone.
Copying <6029> files to the zone.
Initializing zone product registry.
Determining zone package initialization order.
Preparing to initialize <1038> packages on the zone.
Initialized <1038> packages on zone.
Zone <Apache> is initialized.
Installation of these packages generated warnings: ....
The file </export/home/Apache/root/var/sadm/system/logs/install_log>
contains a log of the zone installation.
```

Создаются необходимые каталоги. Зона готова к начальной загрузке.

**3 Просмотр каталогов:**

```
# ls /export/home/Apache/root
bin       etc       home      mnt       platform  sbin
tmp       var     dev       export    lib       opt
 proc      system    usr
```

Пакеты не устанавливаются повторно.

```
# /etc/mount
/export/home/Apache/root/lib on /lib read only
/export/home/Apache/root/platform on /platform read only
/export/home/Apache/root/sbin on /sbin read only
/export/home/Apache/root/usr on /usr read only
/export/home/Apache/root/proc on proc
read/write/setuid/nodevices/zone=Apache
```

**4 Начальная загрузка зоны.**

```
# ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL>
 mtu 8232 index 1 inet 127.0.0.1 netmask ff000000
bge0: flags=1004803<UP,BROADCAST,MULTICAST,DHCP,IPv4> mtu 1500 index 2
        inet 192.168.0.4 netmask ffffff00 broadcast 192.168.0.255
        ether 0:c0:9f:61:88:c9
# zoneadm -z Apache boot
#  ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL>
 mtu 8232 index 1  inet 127.0.0.1 netmask ff000000
lo0:1: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL>
 mtu 8232 index 1 zone Apache  inet 127.0.0.1
bge0: flags=1004803 inet 192.168.0.4 netmask ffffff00 broadcast
 192.168.0.255  ether 0:c0:9f:61:88:c9
bge0:1: flags=1000803mtu 1500 index 2 zone Apache inet
192.168.0.50 netmask ffffff00 broadcast 192.168.0.255
```

**5 Конфигурирование зоны и вход в систему:**

```
# zlogin -C Apache
[Connected to zone 'Apache' pts/5]
# ifconfig -a
lo0:2: flags=2001000849  mtu 8232 index 1 inet 127.0.0.1
```

```
netmask ff000000
bge0:2: flags=1000803 inet 192.168.0.50 netmask ffffff00
 broadcast 192.168.0.255
# ping -s 192.168.0.50
64 bytes from 192.168.0.50: icmp_seq=0. time=0.146 ms
# exit
[Connection to zone 'Apache' pts/5 closed]
```

# Виртуализация веб-серверов при помощи зон

В этом примере демонстрируется добавление поддержки двух разных наборов групп пользователей веб-сервера на одном физическом хосте.

## Аннотация

Одновременный доступ к обоим веб-серверам настраивается так, что каждый веб-сервер и система оказываются защищенными в случае компрометации одного из них.

## Создание двух неглобальных зон

**1 Создание неглобальной зоны Apache1:**

```
# zonecfg -z Apache1 info
zonepath: /export/home/Apache1
autoboot: false
pool:
inherit-pkg-dir:  dir: /lib
inherit-pkg-dir: dir: /platform
inherit-pkg-dir: dir: /sbin
inherit-pkg-dir: dir: /usr
net: address: 192.168.0.100/24
        physical: bge0
```

**2 Создание неглобальной зоны Apache2:**

```
# zonecfg -z Apache2 info
zonepath: /export/home/Apache2
autoboot: false
pool:
inherit-pkg-dir:  dir: /lib
inherit-pkg-dir: dir: /platform
inherit-pkg-dir: dir: /sbin
inherit-pkg-dir: dir: /usr
net: address: 192.168.0.200/24
        physical: bge0
```

**3 Регистрация в Apache1 и установка приложения:**

```
# zlogin Apache1
# zonename
Apache1
# ls /Apachedir
apache_1.3.9    apache_1.3.9-i86pc-sun-solaris2.270.tar
#cd /Apachedir/apache_1.3.9  ; ./install-bindist.sh /local
You now have successfully installed the Apache 1.3.9 HTTP server.
```

**4 Регистрация в Apache2 и установка приложения:**

```
# zlogin Apache2
# zonename
Apache2
# ls /Apachedir
httpd-2.0.50     httpd-2.0.50-i386-pc-solaris2.8.tar
# cd /Apachedir/httpd-2.0.50;  ./install-bindist.sh /local
You now have successfully installed the Apache 2.0.50 HTTP server.
```

**5 Запуск приложения Apache1:**

```
# zonename
Apache1
# hostname
Apache1zone
# /local/bin/apachectl start
/local/bin/apachectl start: httpd started
```

6 **Запуск приложения Apache2:**

```
# zonename
Apache2
# hostname
Apache2zone
# /local/bin/apachectl start
/local/bin/apachectl start: httpd started
```

7 **Отредактируйте в глобальной зоне файл** `/etc/hosts`**:**

```
#  cat /etc/hosts
#
# Internet host table
#
127.0.0.1       localhost
192.168.0.1     loghost
192.168.0.100   Apache1zone
192.168.0.200   Apache2zone
```

8 **Откройте веб-браузер и перейдите к следующему URL:**

`http://apache1zone/manual/index.html`

Веб-сервер Apache1 запущен и работает.

9 **Откройте веб-браузер и перейдите к следующему URL:**

10 `http://apache2zone/manual/`

Веб-сервер Apache2 запущен и работает.

# Обсуждение

Для конечного пользователя каждая из зон выглядит как отдельная система. Каждый веб-сервер имеет собственную службу имен:

- `/etc/nsswitch.conf`
- `/etc/resolv.conf`

Злонамеренная атака на один из веб-серверов не распространяется на другие зоны. Конфликты портов более не являются проблемой!

◆ ◆ ◆ МОДУЛЬ 5

# Настройка файловых систем с ZFS

## Цели

Цель этого урока состоит в кратком описании ZFS и демонстрации способа создания простого пула ZFS с зеркальной файловой системой.

## Дополнительные ресурсы

*ZFS Administration Guide* и справочные страницы (man):
`http://opensolaris.org/os/community/zfs/docs`

## Создание пулов с использованием смонтированных файловых систем

Каждый пул хранения состоит из одного или нескольких виртуальных устройств, которые описывают структуру физического уровня и ее свойства с точки зрения возможных отказов.

Базовым компоновочным блоком для такого пула является физическое устройство, предназначенное для хранения данных. Это может быть любое блочное устройство объемом, по крайней мере, 128 МБ. Обычно это жесткий диск, который виден системе в каталоге `/dev/dsk`. Устройство хранения может быть целым

диском (`c0t0d0`) или отдельным разделом (`c0t0d0s7`). Рекомендуется использовать весь диск, так как в этом случае его не потребуется специально форматировать. ZFS форматирует диск с использованием метки EFI, содержащей один большой раздел (slice).

Этот раздел мы начнем с изучения процедуры создания зеркального пула хранения. Затем будет описана настройка RAID-Z.

В традиционных структурах хранения, в которых применяются разделы или тома, область хранения фрагментируется по отдельным дискам. В системе ZFS область хранения организуется по принципу пула, что позволяет устранить проблемы управления, связанные с томами, и реализовать совместное использование всего доступного пространства. Ценность последнего заключается в возможности восстановления поврежденных данных.

# Создание зеркальных пулов хранения

Целью этого примера является создание и просмотр зеркалированного пула хранения с помощью команды `zpool`.

## Аннотация

ZFS – это не так сложно, так что приступим, не теряя времени! Создадим первый пул:

## Создание зеркальных пулов хранения

1. **Откройте окно терминала.**

2. **Создайте однодисковый пул хранения с названием** `tank`**:**

   ```
   # zpool create tank c1t2d0
   ```

   Теперь существует однодисковый пул хранения под названием `tank`, с единственной файловой системой, смонтированной как `/tank`.

3. **Проверьте, что пул действительно был создан:**

```
# zpool list
NAME   SIZE    USED   AVAIL    CAP  HEALTH     ALTROOT
tank                   80.0G   22.3G   47.7G    28%  ONLINE     -
```

4. **Создайте зеркальное отражение** `tank`**:**

   ```
   # zpool create tank mirror c1t2d0 c2t2d0
   ```

   Пул зеркально отражен на `c2t2d0`.

# Создание файловой системы и каталогов /home

В этом примере демонстрируется создание файловой системы с несколькими каталогами `/home`.

## Аннотация

В этом примере мы воспользуемся командой `zfs` для создания файловой системы и установки ее точки монтирования.

## Создание файловой системы и каталогов /home

1 **Откройте окно терминала.**

2 **Создайте файловую систему** `/var/mail`**:**

```
# zfs create tank/mail
```

3 **Задайте точку монтирования для файловой системы** `/var/mail`**:**

```
# zfs set mountpoint=/var/mail tank/mail
```

4 **Создайте каталог** `home`**:**

```
# zfs create tank/home
```

5 **Затем задайте точку монтирования для каталога** `/home`**:**

```
# zfs set mountpoint=/export/home tank/home
```

6 **Наконец, создайте каталоги** `home` **для всех разработчиков:**

```
# zfs create tank/home/developer1
      # zfs create tank/home/developer2
      # zfs create tank/home/developer3
      # zfs create tank/home/developer4
```

Свойство `mountpoint` наследуется как префикс имени пути. Это означает, что `tank/home/developer1` автоматически монтируется в `/export/home/developer1`, потому что каталог `tank/home` монтируется в `/export/home`.

# Настройка RAID-Z

В этом примере представлено краткое введение в конфигурирование RAID-Z.

# Аннотация

Настройка RAID-Z вместо зеркальных пулов может потребоваться для обеспечения большей избыточности.

# Настройка RAID-Z

1 **Откройте окно терминала.**

2 **Создайте пул с одним устройством RAID-Z, состоящим из 5 дисковых разделов:**

```
# zpool create tank raidz c0t0d0s0 c0t0d1s0 c0t0d2s0 c0t0d3s0 c0t0d4s0
```

В вышеприведенном примере диск должен быть предварительно отформатирован и иметь нулевой раздел требуемого размера. Диски можно указать по их полному пути. Запись `/dev/dsk/c0t0d4s0` идентична просто `c0t0d4s0`.

Следует отметить, что использовать в конфигурации RAID-Z дисковые разделы необязательно. Вышеописанная команда – всего лишь пример использования дисковых разделов в пуле хранения.

◆ ◆ ◆ МОДУЛЬ 6

# Планирование среды OpenSolaris

## Цели

В этом разделе представлено описание системных требований, информации поддержки и документации для установки и конфигурирования в рамках проекта OpenSolaris.

## Дополнительные ресурсы

- *Solaris 10 Installation Guide: Basic Installations*. Sun Microsystems, Inc., 2005.
- *Sun Studio 11: C User's Guide*. Sun Microsystems, Inc, 2005. Щелкните по *Sun Studio 11 Collection* для просмотра книг Sun Studio о `dbx` и `dmake`, Performance Analyzer, а также других ресурсов, касающихся разработки программного обеспечения.
- *Ресурсы по ОС Solaris на портативных компьютерах*:

  `http://www.sun.com/bigadmin/features/articles/laptop_resources.html`
- Сообщество OpenSolaris Laptop Community: `http://opensolaris.org/os/community/laptop`
- Стартовый комплект OpenSolaris Starter Kit: `http://opensolaris.org/os/project/starterkit`

**Совет –** Для получения стартового комплекта OpenSolaris Starter Kit, включающего в себя учебные материалы, исходный код и инструменты разработки, зарегистрируйтесь по адресу `https://opensolaris.org/register.jspa`.

# Конфигурирование среды разработки

Практический опыт работы с кодом операционной системы и прямой доступ к модулям ядра просто незаменимы. Решение уникальных задач, связанных с разработкой ядра и доступом к полномочиям `root` системы, можно упростить путем обращения к инструментальным средствам, форумам и документации в рамках проекта OpenSolaris.

При планировании среды разработки следует учитывать следующие особенности OpenSolaris:

**ТАБЛИЦА 6–1** Поддержка пользовательских конфигурируемых компонентов

| Поддержка конфигурируемых компонентов | в рамках проекта OpenSolaris |
|---|---|
| Аппаратные средства | OpenSolaris поддерживает системы, в которых используются процессорные архитектуры SPARC® и x86: UltraSPARC®, SPARC64, AMD64, Pentium и Xeon EM64T. Список поддерживаемых систем см. в матрице *Solaris OS Hardware Compatibility List* по адресу `http://www.sun.com/bigadmin/hcl`. |
| Исходные файлы | Подробные инструкции по сборке из исходных файлов см. по адресу `http://opensolaris.org/os/downloads`. |

**ТАБЛИЦА 6–1** Поддержка пользовательских конфигурируемых компонентов *(Продолжение)*

| Поддержка конфигурируемых компонентов | в рамках проекта OpenSolaris |
|---|---|
| Установочные образы | Предварительно собранные дистрибутивы OpenSolaris ограничиваются Solaris Express: Community Release [DVD-версия], Build 32 или выше<br><br>В случае ядра OpenSolaris с операционной средой GNU можно попробовать `http://www.gnusolaris.org/gswiki/Download-form`. |
| Архивы BFU | Файл `on-bfu-DATE.PLATFORM.tar.bz2` предназначен для установки из предварительно собранных архивов. |
| Инструменты сборки | Файл `SUNWonbld-DATE.PLATFORM.tar.bz2` служит для компоновки из исходного кода. |
| Компиляторы и инструменты | Компиляторы и инструменты Sun Studio 11 свободно распространяются среди разработчиков OpenSolaris. См. инструкции по загрузке и установке последних версий по адресу `http://www.opensolaris.org/os/community/tools/sun_studio_tools/`. Сообщество gcc можно посетить по адресу `http://www.opensolaris.org/os/community/tools/gcc`. |
| Требования к памяти/диску | ■ Требования к памяти: 256 МБ минимум, 1 ГБ рекомендуется<br>■ Требования к дисковому пространству: 350 МБ |

**ТАБЛИЦА 6–1** Поддержка пользовательских конфигурируемых компонентов *(Продолжение)*

| Поддержка конфигурируемых компонентов | в рамках проекта OpenSolaris |
|---|---|
| Виртуальные среды ОС | Зоны и маркированные зоны в OpenSolaris предоставляют защищенные и виртуализированные среды операционной системы внутри одной системы Solaris, за счет которых обеспечивается изоляция одного или более процессов от остальной системы.<br><br>OpenSolaris поддерживает Xen – механизм мониторинга виртуальной машины с открытым кодом, разработанный командой Xen в компьютерной лаборатории университета Кембриджа. Подробную информацию и ссылки на проект Xen см. по адресу `http://www.opensolaris.org/os/community/xen/`.<br><br>OpenSolaris также может выступать в роли гостевой системы VMWare™; см. недавнюю статью `http://www.opensolaris.org/os/project/content`, описывающую начало работы. |

Для получения дополнительной информации о том, как зоны и маркированные зоны позволяют разрабатывать приложения Solaris и Linux в режиме ядра и в режиме пользователя, предотвращая влияние на работу других разработчиков в отдельных зонах, см. раздел 2.

# Сетевые соединения

Проект OpenSolaris готов к будущим испытаниям, связанным с работой в сети, радикально повышая производительность сети и не требуя при этом изменения существующих приложений.

- При использовании усовершенствованного стека TCP/IP производительность приложений увеличивается примерно на 50 процентов.
- Поддерживаются многие из новейших сетевых технологий, например 10 Gigabit Ethernet, беспроводные сетевые соединения и аппаратная разгрузка (offloading).
- Такие сетевые функции, как высокая доступность, потоковая передача данных и Voice over IP (VoIP), реализуются посредством расширенной поддержки маршрутизации и протоколов.
- Поддерживаются текущие спецификации IPv6.

Более подробно ознакомиться с текущими разработками по сетевым соединениям в рамках проекта OpenSolaris можно здесь: `http://www.opensolaris.org/os/community/networking`. Участие в проекте OpenSolaris может повысить общую производительность в сети с применением последних технологий. Если лабораторная среда основана на OpenSolaris, она становится самоподдерживающейся, поскольку работа всегда происходит в самой новой и самой совершенной среде, причем обновлять ее можно самостоятельно.

◆ ◆ ◆ МОДУЛЬ 7

# Политика OpenSolaris

## Цели

В этом разделе представлено общее описание этапов процесса разработки и стиля кодирования, принятого в проекте OpenSolaris.

## Дополнительные ресурсы

*OpenSolaris Development Process*; `http://www.opensolaris.org/os/community/onnv/os_dev_process/`

*C Style and Coding Standards for SunOS*; `http://www.opensolaris.org/os/community/documentation/getting_started_docs/`

## Процесс разработки и стиль кодирования

Процесс разработки в рамках проекта OpenSolaris включает в себя следующие этапы:

1. Идея

   Сначала у кого-нибудь возникает идея усовершенствования или недовольство каким-либо дефектом. Поиск существующей ошибки либо подача заявления о новой ошибке или запроса на усовершенствование (RFE) осуществляется на веб-странице `http://bugs.opensolaris.org/`. Затем следует оповестить

других разработчиков в соответствующем электронном списке рассылки. Оповещение дает следующие преимущества:

- инициируется дискуссия по изменению или расширению;
- определяется сложность предложенного изменения (изменений);
- оценивается интерес сообщества;
- определяются потенциальные участники группы.

2. Проект

   На этапе проектирования определяется, нужна ли вообще формальная оценка проекта. Если формальная оценка проекта нужна, выполняются следующие три шага:

   - Определяются рецензенты проекта и архитектуры.
   - Пишется проектная документация.
   - Пишется план тестирования.
   - Проводятся оценки проекта и получаются соответствующие одобрения.

3. Реализация

   Этап реализации состоит из следующих шагов:

   - Создается фактический код в соответствии с политикой и стандартами.

     Документ *C Style and Coding Standards for SunOS* можно загрузить здесь: `http://www.opensolaris.org/os/community/documentation/getting_started_docs/`.

   - Пишутся наборы тестов.
   - Проводятся разнообразные модульные и предынтеграционные тесты.
   - Если необходимо, то пишется или обновляется пользовательская документация.
   - По ходу подготовки к интеграции определяются рецензенты кода.

4. Интеграция

Интеграция происходит после завершения всех оценочных рассмотрений и выдачи разрешения на интеграцию.

Фаза интеграции необходима для подтверждения выполнения всех необходимых действий, что означает выполнение оценки кода, документации и полноты реализации.

В документе формального процесса для OpenSolaris предыдущие этапы описываются более подробно, с блок-схемами, иллюстрирующими этапы разработки. В этом документе также раскрываются подробности следующих принципов проектирования и базовых ценностей, относящихся к разработке исходного кода для проекта OpenSolaris:

- **Надежность** – OpenSolaris должен работать корректно, обеспечивая правильность результатов и исключая потерю или повреждение данных.
- **Доступность** – службы должны быть разработаны с учетом возможности перезапуска в случае отказа приложения, и сам OpenSolaris должен быть способен восстановиться после нефатальных аппаратных отказов.
- **Удобство обслуживания** – должна существовать возможность диагностировать как фатальные, так и временные проблемы; везде, где это возможно, диагностика должна автоматизироваться.
- **Безопасность** – безопасность OpenSolaris должна быть включена в операционную систему при проектировании, обеспечивая механизмы контроля изменений, внесенных в систему, и разработчиков, вносящих эти изменения.
- **Производительность** – производительность OpenSolaris не должна уступать другим операционным системам, работающим в идентичных средах.
- **Управляемость** – должна позволять управление отдельными компонентами – программными или аппаратными – ясным и непротиворечивым образом.

- **Совместимость** – новые подсистемы и интерфейсы должны быть расширяемыми и иметь возможности управления версиями, позволяя расширения в будущем без потери совместимости.
- **Удобство сопровождения** – архитектура OpenSolaris должна быть построена с учетом объединения общих подпрограмм в библиотеки или модули ядра, которые могут использоваться неограниченным числом пользователей.
- **платформо-независимость** – OpenSolaris должен оставаться нейтральным в отношении платформы, а низкоуровневые абстракции необходимо разрабатывать с учетом нескольких платформ, включая будущие платформы.

Для получения дополнительной информации о процессе, применяемом при совместной разработке кода OpenSolaris см. `http://www.opensolaris.org/os/community/onnv/os_dev_process/`.

Подобно многим проектам, OpenSolaris предписывает стиль кодирования передаваемого в проект кода, независимо от источника. Этот стиль подробно описывается в `http://opensolaris.org/os/community/onnv/`.

Для проверки ряда элементов стиля кодирования в составе дистрибутива OpenSolaris поставляются два инструмента. Эти инструменты – `cstyle(1)` для проверки соответствия кода C большей части принципов стиля, и `hdrchk(1)` для проверки стиля заголовков C и C++.

◆ ◆ ◆ МОДУЛЬ 8

# 8 Принципы программирования

## Цели

В этом разделе описываются фундаментальные принципы среды программирования OpenSolaris, а именно:

- Поточное программирование
- Краткое описание ядра
- Планирование ЦП
- Отладка процессов

## Дополнительные ресурсы

- *Solaris Internals (2nd Edition)*, Prentice Hall PTR (May 12, 2006) by Jim Mauro and Richard McDougall
- *Solaris Systems Programming*, Prentice Hall PTR (August 19, 2004), by Rich Teer
- *Multithreaded Programming Guide*. Sun Microsystems, Inc., 2005.
- *STREAMS Programming Guide*. Sun Microsystems, Inc., 2005.
- *Solaris 64-bit Developer's Guide*. Sun Microsystems, Inc., 2005.

## Управление процессами и системой

Основная единица рабочей нагрузки – *процесс*. Идентификаторы процессов (PID) пронумерованы в системе последовательно. По умолчанию каждый пользователь приписывается системным

администратором к *проекту*, который является административным идентификатором во всей сети. Каждая успешная регистрация в проекте создает новую *задачу*; задачи представляют собой механизм группирования процессов. Задача содержит процесс регистрации, а также последующие дочерние процессы.

Средство объединения ресурсов в пулы обобщает ресурсы, которые можно привязать к процессу, в общую абстракцию, называемую термином *пул*. Наборы процессоров и другие сущности конфигурируются, группируются и маркируются таким образом, что компоненты операционной мощности оказываются связанными с подмножеством общих ресурсов системы. Если функциональность пулов деактивирована, все процессы принадлежат к одному пулу, `pool_default`, и управление наборами процессоров осуществляется через системный вызов `pset()`. В противном случае наборы процессоров должны управляться при помощи механизма организации пулов. Можно создавать новые пулы, связывая их с наборами процессоров. Процессы можно привязывать к пулам, имеющим непустые наборы ресурсов.

Выполнив поиск OpenGrok по `pool.c`, можно найти обширные комментарии к коду, описывающие взаимосвязь между задачами, пулами, проектами и процессами следующим образом:

Операция привязки задач и проектов к пулам носит элементарный характер. Это означает, что либо все процессы в данной задаче или проекте привязываются к новому пулу, либо (в случае ошибки) они все остаются привязанными к старому пулу. Процессы в данной задаче или проекте могут быть привязанными к разным пулам, только если они были индивидуально привязаны как одиночные процессы. Потоки или LWP одного процесса не имеют привязки к пулу, и привязываются к тем же наборам ресурсов, что связаны с пулом ресурсов данного процесса.

Процессы могут также выполняться внутри зоны. Зоны создаются системными администраторами, часто в целях обеспечения безопасности, для изоляции группы пользователей или процессов друг от друга.

# Поточное программирование

Рассмотрев процессы в контексте задач, проектов, пулов ресурсов, зон и маркированных зон, обсудим теперь процессы в контексте потоков. В традиционном UNIX уже присутствует принцип потоков. Каждый процесс содержит один поток, так что программирование с использованием нескольких процессов одновременно является программированием с использованием нескольких потоков. Однако процесс также является адресным пространством, и при создании процесса создается новое адресное пространство.

Связь между потоками одного процесса проста, поскольку потоки совместно используют все, включая общее адресное пространство и открытые дескрипторы файлов. Таким образом, данные, созданные одним потоком, сразу же становятся доступны всем остальным потокам.

Для потоков POSIX используются библиотеки `libpthread`, а для потоков OpenSolaris – `libthread`. Многопоточность обеспечивает гибкость путем разделения ресурсов уровня ядра и уровня пользователя. В OpenSolaris поддержка многопоточности для обоих наборов интерфейсов предоставляется стандартной библиотекой C.

Для добавления нового потока управления к текущему процессу используется функция `pthread_create(3C)`.

```
int        pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
    void*(*start_routine)(void *), void *arg);
```

Функция `pthread_create ()` вызывается с параметрами (`attr`), содержащими необходимое поведение. `start_routine` – функция, с которой новый поток начинает выполнение. После

возврата `start_routine` потока завершается со статусом выхода, установленным в значение, возвращенное `start_routine`. `pthread_create()` возвращает нуль, если вызов завершается успешно. Любое другое возвращаемое значение указывает на ошибку. Перейдите к `/on/usr/src/lib/libc/spec/threads.spec` в OpenGrok для просмотра полного списка функций и деклараций `pthread`.

Синхронизация потоков позволяет управлять работой программы и получать доступ к совместно используемым данным для одновременно выполняющихся потоков. Четыре объекта синхронизации – блокировка взаимных исключений (mutex), блокировка чтения/записи, переменные условия и семафоры.

- **Блокировка взаимных исключений** (mutual exclusion, mutex) позволяет только одному потоку одновременно исполнять определенный участок кода или получать доступ к определенным данным.
- **Блокировка чтения-записи** разрешает одновременное чтение и эксклюзивную запись в защищенный совместно используемый ресурс. Для изменения ресурса поток должен сначала получить исключительную блокировку записи. Исключительная блокировка записи не разрешается, пока не будут отпущены все блокировки чтения.
- **Переменные условия** блокируют потоки, пока не выполнится определенное условие.
- **Семафоры-счетчики** обычно координируют доступ к ресурсам. Счетчик – это предел количества потоков, которые могут иметь доступ к семафору. Когда счетчик достигает определенного значения, поток, пытающийся получить доступ к ресурсу, блокируется.

## Синхронизация

Объекты синхронизации – переменные в памяти, доступ к которым осуществляется точно так же, как к данным. Потоки различных процессов могут взаимодействовать друг с другом через объекты синхронизации, которые помещаются в

совместно используемой памяти, контролируемой потоками. Потоки могут взаимодействовать друг с другом, несмотря на то, что потоки различных процессов, как правило, невидимы друг другу. Объекты синхронизации могут также помещаться в файлы. Объекты синхронизации могут иметь сроки службы, превышающие срок существования создающего их процесса.

Для поиска `libthread` в дереве исходного кода можно использовать OpenGrok, при этом второй по релевантности результат находится в `mutex.c`, вместе со следующим фрагментом комментария к коду:

Способ реализации всех интерфейсов потоков между ld.so.1 и libthread. В среде без потоков все интерфейсы потока векторизуются в холостые команды. При вызове через _ld_concurrency() из libthread эти векторы переназначаются действительным интерфейсам потоков. Поддерживаются две модели:

TI_VERSION == 1 В этой модели libthread обеспечивает rw_rwlock/rw_unlock, через который векторизуются все вызовы rt_mutex_lock/rt_mutex_unlock. В lib/libthread эти интерфейсы предоставляли _sigon/_sigoff, в отличие от lwp/libthread, где сигнал блокируется через bind_guard/bind_clear.

TI_VERSION == 2 В этой модели используются только интерфейсы libthreads bind_guard/bind_clear и thr_self. Обе libthreads блокируют сигналы в интерфейсах bind_guard/bind_clear. Блокирование на более низком уровне производится из внутренне связанных интерфейсов _lwp_. При этом устраняются рекурсивные проблемы, которые возникают при получении блокирующих интерфейсов из libthread. Взаимное исключение вместо блокировки чтения/записи также позволяет использовать условные переменные для управления параллелизмом потоков (разрешает доступ к объектам только после завершения их .init).

Результаты глобального поиска по POSIX в OpenGrok показывают файл `POSIX.pod`, включающий модуль, описанный в следующем фрагменте комментария к коду:

Модуль POSIX позволяет осуществлять доступ ко всем (или почти всем) стандартным идентификаторам POSIX 1003.1. Для многих из этих идентификаторов определены интерфейсы, похожие на Perl. Объекты, которые являются C<#defines> C, как EINTR или O_NDELAY, автоматически экспортируются в используемое пространство имен. Функции экспортируются только при явном запросе. Наиболее вероятно, что пользователи предпочтут использовать полностью определенные имена функций.

Теперь, после изучения азов определения объектов синхронизации в многопоточном программировании, можно перейти к рассмотрению того, как этими объектами можно управлять при помощи классов планирования.

# Планирование ЦП

Процессы выполняются в классе планирования с отдельной политикой планирования, применяемой в отношении каждого класса, следующим образом:

- **Realtime** (RT) – класс планирования наивысшего приоритета обеспечивает политику для процессов, требующих быстрого ответа и абсолютного контроля пользователя или приложения над приоритетами планирования. Планирование RT может применяться ко всему процессу или к одному или нескольким легковесным процессам (LWP) внутри процесса. Для использования класса реального времени необходимы полномочия `proc_priocntl`. Подробности см. на справочной странице `privileges(5)`.
- **System** (SYS) – класс планирования со средним приоритетом, который не может применяться к пользовательскому процессу.
- **Timeshare** (TS) – класс планирования с наименьшим приоритетом – TS, являющийся также классом по умолчанию. Политика TS приводит к равномерному распределению вычислительных ресурсов между процессами с разными характеристиками использования

ЦП. Другие части ядра могут монополизировать процессор на непродолжительное время, что не приводит к заметному для пользователя замедлению реакции системы.

- **Inter-Active** (`IA`) – интерактивная (`IA`) политика равномерно распределяет вычислительные ресурсы между процессами с разными характеристиками использования ЦП, обеспечивая при этом быстрый отклик при взаимодействии с пользователем.
- **Fair Share** (`FSS`) – политика `FSS` справедливо распределяет вычислительный ресурс среди проектов независимо от количества процессов, которыми они владеют, путем указания *долей* для управления доступом процесса к ресурсам ЦП. В системе сохраняется информация о времени, в течение которого ранее использовался тот или иной ресурс. В зависимости от интенсивности использования ресурса по сравнению с другими процессами права на повторное занятие этого ресурса увеличиваются или уменьшаются соответственно.
- **Fixed-Priority** (`FX`) – политика `FX` обеспечивает планирование по прерыванию с фиксированным приоритетом для процессов, требующих отказа от динамического регулирования системой приоритетов планирования и контроля пользователя или приложения над приоритетами планирования. Этот класс – удобная отправная точка для воздействия на политики выделения ресурсов ЦП.

Класс планирования поддерживается для каждого легковесного процесса (LWP). Потоки имеют класс планирования и приоритет LWP, на которых они основываются. Каждый LWP в процессе может иметь уникальный класс планирования и приоритет, которые видимы ядру. Приоритеты потоков регулируют конкуренцию за объекты синхронизации.

Оба класса планирования – `RT` и `TS` – вызывают `priocntl(2)` для задания уровня приоритета процессов или LWP внутри процесса. После поиска базы кода `priocntl` при помощи OpenGrok можно просмотреть переменные, использующиеся в классах планирования `RT` и `TS` в файле `rtsched.c`:

```
27 #pragma ident    "@(#)rtsched.c    1.10    05/06/08 SMI"
     28
     29 #include "lint.h"
     30 #include "thr_uberdata.h"
     31 #include <sched.h>
     32 #include <sys/priocntl.h>
     33 #include <sys/rtpriocntl.h>
     34 #include <sys/tspriocntl.h>
     35 #include <sys/rt.h>
     36 #include <sys/ts.h>
     37
     38 /*
     39  * The following variables are used for caching information
     40  * for priocntl TS and RT scheduling classs.
     41  */
     42 struct pcclass ts_class, rt_class;
     43
     44 static rtdpent_t *rt_dptbl;    /* RT class parameter table */
     45 static int rt_rrmin;
     46 static int rt_rrmax;
     47 static int rt_fifomin;
     48 static int rt_fifomax;
     49 static int rt_othermin;
     50 static int rt_othermax;
...
```

Команда man priocntl в окне терминала показывает подробности всех классов планирования и описывает их атрибуты и способы использования. Например:

```
% man priocntl
Reformatting page.  Please Wait... done

User Commands                                          priocntl(1)

NAME
     priocntl - display or set scheduling parameters of specified
     process(es)

SYNOPSIS
```

```
     priocntl -l

     priocntl -d [-i idtype] [idlist]

     priocntl  -s  [-c class]  [  class-specific   options]   [-
     i idtype] [idlist]

     priocntl -e [-c class] [  class-specific   options]  command
     [argument(s)]

DESCRIPTION
     The priocntl command displays or sets scheduling  parameters
     of the specified process(es). It can also be used to display
     the current configuration information for the system's  pro-
     cess  scheduler or execute a command with specified schedul-
     ing parameters.

     Processes  fall  into  distinct  classes  with  a  separate
     scheduling policy applied to each class. The process classes
     currently supported are the  real-time  class,  time-sharing
     class,  interactive  class,  fair-share class, and the fixed
     priority class. The characteristics of these classes and the
     class-specific  options  they  accept are described below in
     the USAGE section under the headings Real-Time Class,  Time-
     Sharing  Class,  Inter-Active  Class,  Fair-Share Class, and
     --More--(4%)
```

# Краткое описание ядра

Теперь, получив общую информацию о процессах, потоках и планировании, рассмотрим ядро и отличия его модулей от пользовательских программ. Ядро Solaris выполняет следующие функции:

- управление системными ресурсами, включая файловые системы, процессы и физические устройства;
- предоставление приложениям системных служб, например, управления вводом/выводом, виртуальной памяти и планирования;

- координация взаимодействия всех пользовательских процессов и системных ресурсов;
- назначение приоритетов, обслуживание запросов ресурсов, а также аппаратных прерываний и исключений;
- планирование и переключение потоков, подкачка страниц памяти и свопинг процессов.

В следующем разделе рассматривается ряд важных различий между модулями ядра и пользовательскими программами.

## Различия в исполнении модулей ядра и пользовательских программ

Следующие особенности модулей ядра подчеркивают существенные различия в исполнении модулей ядра и пользовательских программ:

- **Модули ядра имеют отдельное адресное пространство.** Модуль выполняется в *пространстве ядра*. Приложение выполняется в *пространстве пользователя*. Системное программное обеспечение защищено от пользовательских программ. Пространство ядра и пространство пользователя имеют собственные адресные пространства памяти.
- **Модули ядра имеют более высокий приоритет исполнения.** Код, выполняющийся в пространстве ядра, имеет больший приоритет по сравнению с кодом, выполняющимся в пространстве пользователя.
- **Модули ядра не выполняются последовательно.** Пользовательская программа обычно выполняется в последовательном режиме, т.е. каждая задача выполняется от начала до конца. Модуль ядра не выполняется последовательно. Модуль ядра регистрируется для обслуживания будущих запросов.
- **Для модулей ядра возможно прерывание.** Несколько процессов могут одновременно запрашивать модуль ядра. Например, обработчик прерываний может запросить модуль ядра в то же самое время, когда модуль ядра обслуживает системный вызов. В симметричной

многопроцессорной (SMP) системе модуль ядра может выполняться одновременно на нескольких ЦП.

- **Модули ядра должны быть выгружаемыми.** Нельзя предполагать, что код модуля ядра абсолютно защищен, только на основании того, что код драйвера не блокируется. Драйвер следует проектировать с учетом возможности выгрузки модуля.
- **Модули ядра могут совместно использовать данные.** Разные потоки прикладной программы не обязательно совместно используют данные. Напротив, структуры данных и программы, составляющие драйвер, совместно используются всеми потоками, для обработки которых применяется данный драйвер. Драйвер должен быть способен решать конфликты из-за конкуренции, возникающие вследствие множественных запросов. Структуры данных драйвера следует тщательно проектировать с учетом разделения разных потоков исполнения.

## Структурные различия между модулями ядра и пользовательскими программами

Приведенные ниже особенности модулей ядра подчеркивают важные различия между структурой модулей ядра и структурой пользовательские программ:

- **Модули ядра не определяют главную программу.** Модули ядра, включая драйверы устройств, не имеют функции `main()`. Напротив, модуль ядра – это набор подпрограмм и данных.
- **Модули ядра компонуются только с ядром.** Модули ядра не компонуются с теми же библиотеками, что пользовательские программы. Единственные функции, которые может вызывать модуль ядра, – это функции, экспортируемые ядром.
- **Модули ядра используют другие файлы заголовков.** Набор файлов заголовков, требуемый модулями ядра, отличается от набора, необходимого для пользовательских программ.

Список требуемых файлов заголовков приводится на справочной странице (man) для каждой функции. Модули ядра могут включать файлы заголовка, которые используются совместно с пользовательскими программами, если пользовательские и ядерные интерфейсы в таких общих файлах заголовков определены условно с помощью макроса `_KERNEL`.

- **Модули ядра не должны включать в себя глобальные переменные.** Отказ от глобальных переменных в модулях ядра еще более важен, чем в пользовательских программах. По возможности символы следует определять как статические (`static`). Если избежать использования глобальных символов невозможно, им следует давать префикс, являющийся уникальным в пределах ядра. Использование этого префикса для частных символов внутри модуля также является правильным подходом.
- **Модули ядра могут быть адаптированы под аппаратные средства.** Модули ядра могут выделять регистры процесса на определенные роли. Код ядра можно оптимизировать для конкретного процессора. Также можно настроить библиотеки, что поддерживается в OpenSolaris для ряда недавних платформ x86/x64 и UltraSPARC. Так, несмотря на то, что ядро может выделить определенные регистры на определенные роли, как для ядра, так и для прикладных программ/библиотек можно писать адаптированный код.
- **Модули ядра можно загружать и выгружать по запросу.** Набор подпрограмм и данных, составляющих драйвер устройства, может компилироваться в одиночный загружаемый модуль объектного кода. Этот загружаемый модуль можно затем статически или динамически скомпоновать с ядром и разорвать компоновку с ядром. В ядро можно добавить функциональные возможности при работающей системе. Новые версии драйвера можно тестировать без перезагрузки системы.

# Отладка процессов

Отладка процессов на всех уровнях стека разработки – ключевая задача при написании модулей ядра.

Глобальный поиск `libthread` в OpenGrok показывает следующие комментарии к коду в файле `mdb_tdb.c`, описывающем связь между многопоточной отладкой и работой `mdb`:

Для правильной отладки многопоточных программ, цель proc должна быть способной запросить и изменить такую информацию, как набор регистров потока, с использованием либо собственных служб LWP, предоставляемых libproc (если процесс не скомпонован с libthread), либо с помощью служб, предоставляемых libthread_db (если процесс скомпонован с libthread). Кроме того, процесс может начинаться как однопотоковый процесс, а затем выполнить dlopen() для libthread. Поэтому необходимо быть готовым к переключению режимов на лету. Возможны также две реализации libthread (одна в /usr/lib и одна в /usr/lib/lwp), поэтому mdb нельзя напрямую скомпоновать с libthread_db; вместо этого надо выполнить dlopen для соответствующей libthread_db на лету, в зависимости от разновидности libthread.so, открытой контролируемым процессом. Наконец, mdb разработан с учетом возможности одновременной активности нескольких целей, так что *обе* libthread_db могут быть открыты одновременно. Это может произойти, если просматриваются два многопоточных процесса в аварийном дампе, причем один из них использует /usr/lib/libthread.so, а другой – /usr/lib/lwp/libthread.so. Для выполнения этих требований в данном файле реализуется "кэш" libthread_db. Цель proc вызывает mdb_tdb_load() с именем пути libthread_db, которую необходимо загрузить, и если она еще не открыта, в отношении ее выполняется dlopen(), проводится поиск символов, на которые необходимо сослаться, и заполняется вектор операции, возвращаемый вызывающей стороне. После загрузки объекта выгружать его может потребоваться только в случае явного сброса всего кэша. Этот механизм также имеет ценное свойство,

заключающееся в отсутствии необходимости загрузки libthread_db, пока она не потребуется, так что отладчик запускается быстрее.

Для доступа к LWP многопоточной программы можно воспользоваться следующими командами `mdb`:

- `$l` – печать идентификатора LWP представительного потока, если цель – пользовательский процесс.
- `$L` – печать идентификаторов LWP каждого LWP в цели, если цель – пользовательский процесс.
- *PID* `::attach` – присоединение к проекту с использованием `pid` (идентификатора процесса).
- `::release` – освобождение ранее присоединенного процесса или файла ядра. Этот процесс можно впоследствии продолжить командой `prun(1)` или возобновить при помощи MDB или другого отладчика.
- *адрес* `::context` – переключение контекста на указанный процесс. Применение этих команд для задания условных точек останова часто бывает оправдано.
- `[ addr ] ::bp [+/-dDestT] [-c cmd] [-n count] sym ...` – задание точки останова в указанных местах.
- `addr ::delete [id | all]` – удаление спецификаторов событий с данным идентификационным номером.

Датчики DTrace создаются подобно запросам MDB. Практические упражнения будут продолжены с DTrace, а MDB будет добавлен по мере перехода к более сложным процессам отладки.

◆ ◆ ◆ МОДУЛЬ 9

# 9 Введение в DTrace

## Цели

В рамках этого практического примера будет представлено введение в DTrace с использованием скрипта "датчика" для системного вызова при помощи DTrace.

## Дополнительные ресурсы

- *Solaris Dynamic Tracing Guide*. Sun Microsystems, Inc., 2005.
- *DTrace User Guide*, Sun Microsystems, Inc., 2006

## Включение простых датчиков DTrace

Следующий пример раскрывает основные цели и возможности датчиков DTrace.

### Аннотация

Разъяснение принципов работы DTrace можно начать с построения простейших запросов при помощи датчика под названием BEGIN, который срабатывает всякий раз при запуске нового запроса трассировки. Для включения датчика по его строковому имени можно использовать опцию `-n` утилиты `dtrace(1M)`.

# Включение простого датчика DTrace

1 **Откройте окно терминала.**

2 **Включение датчика:**

```
# dtrace -n BEGIN
```

После краткой паузы команда `dtrace` сообщит об активизации одного датчика, и будет показана строка вывода, указывающая на срабатывание датчика BEGIN. После вывода этих данных `dtrace` остается в приостановленном состоянии и ожидает срабатывания других датчиков. Так как другие датчики не были включены, а BEGIN срабатывает лишь однажды, нажмите CTRL-C в оболочке для выхода из `dtrace` и возврата в приглашение оболочки:

3 **Вернитесь в приглашение оболочки путем нажатия комбинации клавиш CTRL-C:**

```
# dtrace -n BEGIN
dtrace: description 'BEGIN' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                      :BEGIN
^C
#
```

Выходные данные показывают, что датчик под названием BEGIN сработал один раз. Также выводятся его имя и целочисленный идентификатор, 1. Следует отметить, что по умолчанию выводится целочисленное имя ЦП, на котором сработал этот датчик. В этом примере в столбце CPU указывается, что в момент срабатывания датчика команда `dtrace` выполнялась на ЦП 0.

Запросы DTrace можно конструировать с использованием произвольного количества датчиков и действий. Создадим простой запрос с двумя датчиками, добавив датчик END к

предыдущей демонстрационной команде. Датчик END срабатывает один раз по завершении трассировки.

**4 Добавление датчика END:**

```
# dtrace -n BEGIN -n END
dtrace: description 'BEGIN' matched 1 probe
dtrace: description 'END' matched 1 probe
CPU     ID                    FUNCTION:NAME   0      1    :BEGIN
^C
0      2                     :END
#
```

Датчик END срабатывает один раз по завершении трассировки. Как можно видеть, нажатие CTRL-C для выхода из DTrace приводит к срабатыванию датчика END. DTrace сообщает о срабатывании этого датчика перед выходом.

# Вывод списка отслеживаемых датчиков

В этом примере проводится более подробное исследование датчиков и демонстрируется вывод списка датчиков в системе.

## Аннотация

В предшествующих примерах было объяснено использование простых датчиков с названиями `BEGIN` и `END`. Но откуда появились эти датчики? Датчики DTrace происходят из набора модулей ядра, называемых *провайдерами*, в каждом из которых могут быть созданы датчики. Например, провайдер `syscall` обеспечивает датчики в каждом системном вызове, а провайдер `fbt` – в каждой функции в ядре.

При использовании DTrace каждому провайдеру предоставляется возможность опубликовать датчики, которые он может поставить платформе DTrace. После этого можно включить и связать действия по трассировке с любым из

опубликованных датчиков.

# Вывод списка трассируемых датчиков

1. **Откройте окно терминала.**

2. **Введите следующую команду:**

   ```
   # dtrace
   ```

   Параметры команды `dtrace` подаются на вывод.

3. **Введите команду** `dtrace` **с параметром** `-l`**:**

```
# dtrace -l | more
  ID   PROVIDER            MODULE          FUNCTION NAME
   1     dtrace                                     BEGIN
   2     dtrace                                     END
   3     dtrace                                     ERROR
   4   lockstat           genunix       mutex_enter adaptive-acquire
   5   lockstat           genunix       mutex_enter adaptive-block
   6   lockstat           genunix       mutex_enter adaptive-spin
   7   lockstat           genunix       mutex_exit  adaptive-release
--More--
```

Список датчиков, доступных в системе, выводится со следующими пятью элементами данных:

- ID – внутренний идентификатор датчика.
- Provider – имя провайдера. Провайдеры используются для классификации датчиков. Это значение также указывает используемый метод инструментальных средств.
- Module – имя модуля Unix или прикладной библиотеки датчика.
- Function – имя функции, в которой содержится датчик.
- Name – имя датчика.

**4 Предыдущую команду можно передать по каналу (pipe) в `wc` для определения общего числа датчиков в системе:**

```
# dtrace -l | wc -l
30122
```

В выходных данных указывается количество датчиков, известных системе на данный момент. Это количество изменяется в зависимости от типа системы.

**5 Для фильтрации списка можно добавить одну из следующих опций:**

- `-P` по провайдеру
- `-m` по модулю
- `-f` по функции
- `-n` по имени

Рассмотрим следующие примеры:

```
# dtrace -l -P lockstat
ID   PROVIDER            MODULE          FUNCTION NAME
 4   lockstat           genunix       mutex_enter adaptive-acquire
 5   lockstat           genunix       mutex_enter adaptive-block
 6   lockstat           genunix       mutex_enter adaptive-spin
 7   lockstat           genunix       mutex_exit  adaptive-release
```

Выводятся только датчики, доступные в провайдере `lockstat`.

```
# dtrace -l -m ufs
  ID  PROVIDER            MODULE          FUNCTION NAME
  15   sysinfo               ufs     ufs_idle_free ufsinopage
  16   sysinfo               ufs ufs_iget_internal ufsiget
 356       fbt               ufs            allocg entry
```

Выводятся только датчики из модуля UFS.

```
# dtrace -l -f open
 ID   PROVIDER            MODULE          FUNCTION NAME
  4   syscall                                 open entry
  5   syscall                                 open return
```

```
116   fbt                genunix              open entry
117   fbt                genunix              open return
```

Выводятся только датчики с именем функции `open`.

```
# dtrace -l -n start
  ID   PROVIDER            MODULE          FUNCTION NAME
 506       proc              unix   lwp_rtt_initial start
2766         io           genunix    default_physio start
2768         io           genunix           aphysio start
5909         io               nfs          nfs4_bio start
```

Приведенная выше команда выводит список всех датчиков, имеющих имя датчика `start`.

# Программирование в D

Теперь, после объяснения основных терминов, включения и вывода списка датчиков, можно написать DTrace-версию всеобщей первой программы, "Hello, World".

## Аннотация

Как показано в этом примере, эксперименты DTrace можно конструировать не только в командной строке, их можно также писать в текстовых файлах с помощью языка программирования D.

## Написание программы DTrace

1 **Откройте окно терминала.**

2 **Создайте в текстовом редакторе новый файл с названием `hello.d`.**

3 **Введите первую программу на D:**

```
BEGIN {     trace(″hello, world″);     exit(0); }
```

4 **Сохраните файл** `hello.d`**.**

5 **Выполните программу с использованием опции** `dtrace -s`**:**

```
# dtrace -s hello.d
dtrace: script 'hello.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0        1                      :BEGIN   hello, world
#
```

Как можно видеть, `dtrace` выводит те же самые данные, что и прежде, плюс текст "`hello, world`". В отличие от предыдущего примера, не требовалось ждать и нажимать CTRL-C. Эти изменения – результат *действий*, указанных для датчика `BEGIN` в `hello.d`. Для пояснения рассмотрим структуру программы на D более подробно.

# Обсуждение

Каждая программа на D состоит из серии *выражений*, причем каждое выражение описывает один или несколько включаемых датчиков и дополнительный набор действий, выполняемых при срабатывании датчика. Эти действия приводятся в виде серии операторов, заключенных в фигурные скобки `{ }` после имени датчика. Каждый оператор заканчивается точкой с запятой (`;`).

Первый оператор задействует функцию `trace()` для указания того, что DTrace должен записать указанный аргумент – строку "`hello, world`" – при срабатывании датчика `BEGIN`, а затем напечатать его. Второй оператор использует функцию `exit()` для указания того, что DTrace должен прекратить трассировку и выйти из команды `dtrace`.

DTrace предоставляет ряд удобных функций, подобных `trace()` и `exit()`, для вызова в программах D. Для вызова функции

необходимо указать ее имя, а затем список аргументов в скобках. Полный набор функций D описан в руководстве *Solaris Dynamic Tracing Guide*.

К настоящему времени, если читатель знаком с языком программирования C, из имени и примеров, вероятно, стало очевидно, что язык программирования D DTRACE очень похож на C и `awk(1)`. В самом деле, D происходит из подмножества C, объединенного со специальным набором функций и переменных для упрощения трассировки.

При наличии опыта написания программ на C не составит большого труда немедленно применить имеющиеся знания в целях построения программ трассировки на D. Даже без подобного опыта изучение языка D чрезвычайно просто. Однако сначала отступим от правил языка и подробнее изучим принципы действия DTrace, а потом вернемся к составлению более интересных программ на D.

◆ ◆ ◆ МОДУЛЬ 10

# Отладка приложений при помощи DTrace

## Цели

В этом разделе рассматривается использование DTrace для контроля событий приложений.

## Дополнительные ресурсы

*Application Packaging Developer's Guide*. Sun Microsystems, Inc., 2005.

## Включение датчиков режима пользователя

DTrace позволяет динамически добавлять датчики в функции уровня пользователя. Код пользователя не требует перекомпиляции, особых флагов или даже перезапуска. Датчики DTrace можно включить простым вызовом провайдера.

Описание датчика имеет следующий синтаксис:

```
pid:mod:function:name
```

- `pid`: format pid *идентификатор_процесса* (например, `pid5234`)
- `mod`: имя библиотеки или a.out (исполняемого файла)

- `function`: имя функции
- `name`: `entry` для входа функции `return` для выхода функции

# Трассировка приложений при помощи DTrace

В этом упражнении будет описано использование DTrace на приложениях пользователя.

## Аннотация

В этом примере для трассировки связанного приложения в описание датчика включается идентификатор процесса. Сложность шагов увеличивается по мере приближения к концу упражнения, возрастает также объем выводимой информации о поведении приложения.

## Отладка `gcalctool` при помощи DTrace

**1 Запустите калькулятор из меню Application или Program.**

**2 Отыщите идентификатор только что запущенного процесса:**

```
# pgrep gcalctool
8198
```

Этот номер – идентификатор процесса `calc`, далее называемый `procid`.

**3 Следуя нижеприведенным шагам, создайте D-скрипт, считающий число вызовов любой функции в `gcalctool`.**

**a. Создайте в текстовом редакторе новый файл с названием `proc_func.d`.**

**b. Используйте в качестве описания датчика `pid$1:::entry`.**

`$1` – первый аргумент, передаваемый в скрипт, предикат следует оставить пустым.

**c. В разделе действий добавьте агрегацию для подсчета числа вызова функций с использованием оператора агрегации `@[probefunc]=count()`.**

```
pid$1:::entry
{
                @[probefunc]=count();
}
```

**d. Запустите написанный скрипт.**

```
# dtrace -qs proc_func.d идентификатор процесса
```

Замените *идентификатор процесса* идентификатором процесса `gcalctool`.

**e. Выполните вычисление на калькуляторе.**

**f. Нажмите CTRL+C в окне, где запущен D-скрипт.**

---

**Примечание –** Скрипт DTrace собирает данные и ожидает прекращения сбора нажатием CTRL+C. Если не требуется печатать созданную агрегацию, DTrace напечатает ее самостоятельно.

---

**4 Теперь измените скрипт для подсчета только функций из библиотеки `libc`.**

**a. Скопируйте `proc_func.d` в `proc_libc.d`.**

**b. Измените описание датчика в файле `proc_libc.d` на следующее:**

```
pid$1:libc::entry
```

**c. Новый скрипт должен выглядеть следующим образом:**

```
pid$1:libc::entry
 {                          @[probefunc]=count();
 }
```

**5 Теперь запустите скрипт.**

```
# dtrace -qs  proc_libc.d  идентификатор процесса
```

Замените *идентификатор процесса* идентификатором процесса `gcalctool`.

**a. Выполните вычисление на калькуляторе.**

**b. Нажмите CTRL+C в окне, где запущен D-скрипт для просмотра результатов.**

**6 Наконец, измените скрипт для нахождения количества времени, затраченного в каждой функции.**

**a. Создайте файл и назовите его `func_time.d`.**

В `func_time.d` используются два описания датчиков.

**b. Запишите первый датчик:**

```
pid$1:::entry
```

**c. Запишите второй датчик:**

```
pid$1:::return
```

**d. В разделе действий первого датчика сохраните `timestamp` в переменной `ts`.**

Временная метка (timestamp) – встроенный элемент DTrace, подсчитывающий число наносекунд, прошедших с момента в прошлом.

**e. В разделе действий второго датчика подсчитываются прошедшие наносекунды с использованием следующей агрегации:**

```
@[probefunc]=sum(timestamp - ts)
```

**f. Новый `func_time.d` скрипт должен соответствовать следующему:**

```
pid$1:::entry
 {                          ts = timestamp;
 }
              pid$1:::return /ts/
 {                        @[probefunc]=sum(timestamp - ts);
 }
```

**7 Запустите новый скрипт `func_time.d`:**

```
# dtrace -qs  func_time.d идентификатор процесса
```

Замените *идентификатор процесса* идентификатором процесса `gcalctool`.

**a. Выполните вычисление на калькуляторе.**

**b. Нажмите CTRL+C в окне, где запущен D-скрипт для просмотра результатов.**

```
^C
gdk_xid__equal                                            2468
_XSetLastRequestRead                                     2998
_XDeq                                                     3092

...
```

В левом столбце показывается имя функции, а в правом – количество времени, проведенного в этой функции. Время исчисляется в наносекундах.

◆ ◆ ◆ МОДУЛЬ 11

# Отладка приложений C++ при помощи DTrace

## Цели

Примеры в этом модуле демонстрируют использование DTrace для диагностики ошибок приложений C++. Эти примеры также используются для сравнения DTrace с другими инструментами отладки приложений, в том числе Sun Studio 10 и `mdb`.

## Использование DTrace для профилирования и отладки программы C++

Для демонстрации часто встречающейся в приложениях C++ ошибки – утечки памяти – создана типовая программа `CCtest`. Во многих случаях утечка памяти происходит в случаях, когда объект создается, но не уничтожается; в программе, рассматриваемом в этом разделе, имеет место как раз такой случай.

При отладке программы C++ можно заметить, что компилятор преобразует некоторые имена C++ в искаженные, трудно различимые строки символов и цифр. Такая корректировка имен технически необходима для поддержки перегрузки функций C++ для обеспечения действительных внешних имен у тех функций C++, которые включают специальные символы, и различения экземпляров одного имени, объявленного в разных пространствах имен и классах.

Например, использование `nm` для извлечения таблицы символов из типовой программы под названием CCtest приводит к следующим результатам:

```
# /usr/ccs/bin/nm CCtest
...
[61] | 134549248| 53|FUNC |GLOB |0 |9  |__1cJTestClass2T5B6M_v_
[85] | 134549301| 47|FUNC |GLOB |0 |9  |__1cJTestClass2T6M_v_
[76] | 134549136| 37|FUNC |GLOB |0 |9  |__1cJTestClass2t5B6M_v_
[62] | 134549173| 71|FUNC |GLOB |0 |9  |__1cJTestClass2t5B6Mpc_v_
[64] | 134549136| 37|FUNC |GLOB |0 |9  |__1cJTestClass2t6M_v_
[89] | 134549173| 71|FUNC |GLOB |0 |9  |__1cJTestClass2t6Mpc_v_
[80] | 134616000| 16|OBJT |GLOB |0 |18 |__1cJTestClassG__vtbl_
[91] | 134549348| 16|FUNC |GLOB |0 |9  |__1cJTestClassJClassName6kM_pc_
...
```

---

**Примечание –** Исходный код и файл makefile для CCtest приведены в конце этого раздела.

---

Из этих выходных данных можно сделать вывод о том, что эти скорректированные символы связаны с классом `TestClass`, но невозможно определить, связаны ли эти символы с конструкторами, деструкторами или функциями класса.

Компилятор Sun Studio содержит следующие три утилиты, которые можно использовать для преобразования этих скорректированных символов в их эквиваленты C++: `nm -C`, `dem` и `c++filt`.

---

**Примечание –** Здесь используется программное обеспечение Sun Studio 10, однако примеры тестировались как в Sun Studio версии 9, так и 10.

---

Если приложение C++ скомпилировано с помощью `gcc/g++`, возможен дополнительный вариант устранения искажений в приложении – в дополнение к утилите `c++filt`, распознающей скорректированные имена как Sun Studio, так и GNU, утилита с открытым кодом `gc++filt` из каталога `/usr/sfw/bin` может использоваться для исправления символов в приложении `g++`.

Примеры: Символы Sun Studio без `c++filt`:

```
# nm CCtest | grep TestClass
[65] | 134549280| 37|FUNC |GLOB |0 |9 |__1cJTestClass2t6M_v_
[56] | 134549352| 54|FUNC |GLOB |0 |9 |__1cJTestClass2t6Mi_v_
[92] | 134549317| 35|FUNC |GLOB |0 |9 |__1cJTestClass2t6Mpc_v_
...
```

Символы Sun Studio с `c++filt`:

```
# nm CCtest | grep TestClass | c++filt
[65] | 134549280| 37|FUNC |GLOB |0 |9 |TestClass::TestClass()
[56] | 134549352| 54|FUNC |GLOB |0 |9 |TestClass::TestClass(int)
[92] | 134549317| 35|FUNC |GLOB |0 |9 |TestClass::TestClass(char*)
...
```

Символы g++ без `gc++filt`:

```
[86]  | 134550070| 41|FUNC |GLOB |0 |12 |_ZN9TestClassC1EPc
[110] | 134550180| 68|FUNC |GLOB |0 |12 |_ZN9TestClassC1Ei
[114] | 134549984| 43|FUNC |GLOB |0 |12 |_ZN9TestClassC1Ev
    ...
```

Символы g++ с `gc++filt`:

```
# nm gCCtest | grep TestClass | gc++filt
[86]  | 134550070| 41|FUNC |GLOB |0 |12 |TestClass::TestClass(char*)
[110] | 134550180| 68|FUNC |GLOB |0 |12 |TestClass::TestClass(int)
[114] | 134549984| 43|FUNC |GLOB |0 |12 |TestClass::TestClass()
    ...
```

И, наконец, отображение символов с `nm -C`:

```
[64] | 134549344| 71|FUNC |GLOB |0 |9 |TestClass::TestClass()
                                        [__1cJTestClass2t6M_v_]
[87] | 134549424| 70|FUNC |GLOB |0 |9 |TestClass::TestClass(const char*)
                                        [__1cJTestClass2t6Mpkc_v_]
[57] | 134549504| 95|FUNC |GLOB |0 |9 |TestClass::TestClass(int)
                                        [__1cJTestClass2t6Mi_v_]
```

Используем эту информацию для создания скрипта DTrace, выполняющего агрегирование вызовов объектов, связанных с тестовой программой. Провайдер DTrace `pid` можно использовать для включения датчиков, связанных со скорректированными символами C++.

Для тестирования теории конструкторов/деструкторов начнем с подсчета следующего:

- Количество созданных объектов – вызовы `new()`.
- Количество уничтоженных объектов – вызовы `delete()`.

Следующий скрипт используется для извлечения из программы CCtest символов, соответствующих функциям `new()` и `delete()`:

```
# dem 'nm CCtest | awk -F\| '{ print $NF; }'' | egrep "new|delete"
__1c2k6Fpv_v_ == void operator delete(void*)
__1c2n6FI_pv_ == void*operator new(unsigned)
```

Соответствующий скрипт DTrace используется для включения датчиков на `new()` и `delete()` (сохраняется как CCagg.d):

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

pid$1::__1c2n6FI_pv_:
{
            @n[probefunc] = count();
}
            pid$1::__1c2k6Fpv_v_:
{
            @d[probefunc] = count();
}

END
{
    printa(@n);
    printa(@d);
}
```

Запустите программу CCtest в одном окне, затем запустите только что созданный скрипт в другом окне:

```
# dtrace -s ./CCagg.d 'pgrep CCtest' | c++filt
```

Выходные данные DTrace передаются по каналу (pipe) через `c++filt` для исправления скорректированных символов C++, со следующим предостережением.

**Внимание –** Скрипт DTrace нельзя завершить при помощи ^C, как обычно, поскольку `c++filt` будет завершен вместе с DTrace без вывода выходных данных. Для отображения вывода этой команды перейдите к другому окну в системе и введите:

```
# pkill dtrace
```

Эта же последовательность шагов используется в остальных упражнениях:

Окно 1:

```
# ./CCtest
```

Окно 2:

```
# dtrace -s имя_скрипта | c++filt
```

Окно 3:

```
# pkill dtrace
```

Выходные данные скрипта агрегирования в окне 2 должны выглядеть следующим образом:

```
void*operator new(unsigned)                              12
void operator delete(void*)                               8
```

Таким образом, предположение о том, что создается больше объектов, чем удаляется, может быть правильным.

Проверим адреса памяти объектов и попытаемся сопоставить экземпляры `new()` и `delete()`. Аргументы DTrace используются для отображения адресов, связанных с объектами. Так как

указатель на объект содержится в возвращаемом значении `new()`, значение указателя, использованное в `arg0`, должно встречаться и в вызове `delete()`. Немного изменив первоначальный скрипт, получим следующий вариант, который назовем `CCaddr.d`:

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

#pragma D option quiet
/*
__1c2k6Fpv_v_ == void operator delete(void*)
__1c2n6FI_pv_ == void*operator new(unsigned)
*/

/* return from new() */
pid$1::__1c2n6FI_pv_:return
{
            printf("%s: %x\n", probefunc, arg1);
}

/* call to delete() */            pid$1::__1c2k6Fpv_v_:entry
{
            printf("%s: %x\n", probefunc, arg0);
}
```

Запустите этот скрипт:

```
# dtrace -s ./CCaddr.d ‘pgrep CCtest‘ | c++filt
```

Подождите в течение некоторого времени, а затем введите в окне 3 следующее:

```
# pkill dtrace
```

Выходные данные выглядят как повторяющаяся структура из трех вызовов `new()` и двух вызовов `delete()`:

```
void*operator new(unsigned): 809e480
void*operator new(unsigned): 8068a70
void*operator new(unsigned): 809e4a0
```

```
void operator delete(void*): 8068a70
void operator delete(void*): 809e4a0
```

При исследовании повторяющихся выходных данных обнаруживается закономерность. По-видимому, первый вызов `new()` повторяющейся последовательности не имеет соответствующего вызова `delete()`. Итак, на данном этапе источник утечки памяти идентифицирован!

Продолжим работу с DTrace и выясним, что еще можно узнать из этой информации. Все еще неизвестно, какой тип класса связан с объектом, создаваемым по адресу `809e480`. Подсказку может дать включение вызова `ustack()` на входе `new()`. Ниже приведен измененный вариант предыдущего скрипта, названный на этот раз `CCstack.d`:

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

#pragma D option quiet

/*
__1c2k6Fpv_v_ == void operator delete(void*)
__1c2n6FI_pv_ == void*operator new(unsigned)
*/

pid$1::__1c2n6FI_pv_:entry
{
            ustack();
}
            pid$1::__1c2n6FI_pv_:return
{
            printf("%s: %x\n", probefunc, arg1);
}
            pid$1::__1c2k6Fpv_v_:entry
{
            printf("%s: %x\n", probefunc, arg0);
}
```

Выполните `CCstack.d` в окне 2, затем введите `pkill dtrace` в окне 3. Появится следующий результат:

```
# dtrace -s ./CCstack.d 'pgrep CCtest' | c++filt


              libCrun.so.1'void*operator new(unsigned)
              CCtest'main+0x19
              CCtest'0x8050cda
void*operator new(unsigned): 80a2bd0

              libCrun.so.1'void*operator new(unsigned)
              CCtest'main+0x57
              CCtest'0x8050cda
void*operator new(unsigned): 8068a70

              libCrun.so.1'void*operator new(unsigned)
              CCtest'main+0x9a
              CCtest'0x8050cda
void*operator new(unsigned): 80a2bf0
void operator delete(void*): 8068a70
void operator delete(void*): 80a2bf0
```

Данные `ustack()` указывают, что `new()` вызывается из `main+0x19`, `main+0x57` и `main+0x9a` – и что интерес представляет объект, связанный с первым вызовом `new()` в `main+0x19`.

Для определения типа конструктора, вызываемого в `main+0x19`, используем `mdb`:

```
# gcore 'pgrep CCtest'
gcore: core.1478 dumped
# mdb core.1478
Loading modules: [ libc.so.1 ld.so.1 ]
> main::dis
main:           pushl  %ebp
main+1:         movl   %esp,%ebp
main+3:         subl   $0x38,%esp
main+6:         movl   %esp,-0x2c(%ebp)
main+9:         movl   %ebx,-0x30(%ebp)
main+0xc:       movl   %esi,-0x34(%ebp)
main+0xf:       movl   %edi,-0x38(%ebp)
main+0x12:      pushl  $0x8
```

```
main+0x14:      call    -0x2e4   <PLT=libCrun.so.1‘__1c2n6FI_pv_>
main+0x19:      addl    $0x4,%esp
main+0x1c:      movl    %eax,-0x10(%ebp)
main+0x1f:      movl    -0x10(%ebp),%eax
main+0x22:      pushl   %eax
main+0x23:      call    +0x1d5   <__1cJTestClass2t5B6M_v_>
    ...
```

Конструктор вызывается после вызова new, по смещению `main+0x23`. Итак, обнаружен вызов конструктора `__1cJTestClass2t5B6M_v_`, который не уничтожается. Используем `dem` для исправления этого скорректированного символа и получим следующее:

```
# dem __1cJTestClass2t5B6M_v_
__1cJTestClass2t5B6M_v_ == TestClass::TestClass #Nvariant 1()
```

Таким образом, причиной утечки памяти является вызов new `TestClass()` в `main+0x19`. Исследование исходного файла `CCtest.cc` показывает:

```
...
t = new TestClass();
cout << t->ClassName();

t = new TestClass((const char *)"Hello.");
cout << t->ClassName();

tt = new TestClass((const char *)"Goodbye.");
cout << tt->ClassName();

delete(t);
delete(tt);
...
```

Ясно, что первое использование переменной `t = new TestClass();` перезаписывается при ее использовании во второй раз: `t = new TestClass((const char *)"Hello.");`. Утечка памяти обнаружена, и можно внести исправления.

Провайдер DTrace `pid` позволяет включить датчик на любой инструкции, связанной с исследуемым процессом. Этот пример служит для моделирования подхода DTrace к интерактивной отладке процессов. В примере используются следующие функции DTrace: агрегирование, отображение аргументов и возвращаемых значений функции и просмотр стека вызовов пользователя. Команды `dem` и `c++filt` в программном обеспечении Sun Studio и `gc++filt` в `gcc` были использованы для извлечения датчиков функций из таблицы символов программы и отображения выходных данных DTrace в совместимом с исходным файлом формате. Исходные файлы, созданные для этого примера:

**ПРИМЕР 11–1** TestClass.h

```
class TestClass
{
    public:
        TestClass();
        TestClass(const char *name);
        TestClass(int i);
        virtual ~TestClass();
        virtual char *ClassName() const;
    private:
        char *str;
};
```

TestClass.cc:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include "TestClass.h"

TestClass::TestClass() {
    str=strdup("empty.");
}

TestClass::TestClass(const char *name) {
```

**ПРИМЕР 11–1** TestClass.h *(Продолжение)*

```
    str=strdup(name);
}

TestClass::TestClass(int i) {
    str=(char *)malloc(128);
    sprintf(str, "Integer = %d", i);
}

TestClass::~TestClass() {
    if ( str )
        free(str);
}

char *TestClass::ClassName() const {
    return str;
}
```

**ПРИМЕР 11–2** CCtest.cc

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "TestClass.h"

int main(int argc, char **argv)
{
    TestClass *t;
    TestClass *tt;

    while (1) {
        t = new TestClass();
        cout << t->ClassName();

        t = new TestClass((const char *)"Hello.");
        cout << t->ClassName();
```

**ПРИМЕР 11–2** CCtest.cc *(Продолжение)*

```
        tt = new TestClass((const char *)"Goodbye.");
        cout << tt->ClassName();

        delete(t);
        delete(tt);
        sleep(1);
    }
}
```

**ПРИМЕР 11–3** Makefile

```
OBJS=CCtest.o TestClass.o
PROGS=CCtest

CC=CC

all: $(PROGS)
    echo "Done."

clean:
    rm $(OBJS) $(PROGS)

CCtest: $(OBJS)
    $(CC) -o CCtest $(OBJS)

.cc.o:
    $(CC) $(CFLAGS) -c $<
```

◆ ◆ ◆ МОДУЛЬ 12

# 12 Управление памятью при помощи DTrace и MDB

## Цели

Этот раздел основывается на материале, изученном в связи с DTrace, и описывает наблюдение за процессами путем исследования ошибок, вызванных отсутствием страницы. Затем для поиска проблемы в коде будет включена низкоуровневая отладка с использованием MDB.

## Дополнительные ресурсы

*Solaris Modular Debugger Guide*. Sun Microsystems, Inc., 2005.

## Управление программной памятью

В управлении памятью OpenSolaris для непосредственного управления виртуальной памятью процессов и самого ядра используются программные конструкции, называемые сегментами. Большая часть структур данных, задействованных в программной составляющей управления памятью, определяются в заголовке `/usr/include/vm/*.h`. В этом разделе будет исследован код и структуры данных, используемые для обработки отсутствия страниц.

# Использование DTrace и MDB для исследования виртуальной памяти

В этом практическом примере рассматривается обработка ошибки, вызванной отсутствием страницы, при помощи DTrace и MDB.

## Аннотация

Начнем со скрипта DTrace, отслеживающего события отсутствия одной страницы для конкретного процесса. Скрипт печатает пользовательский виртуальный адрес, вызвавший отказ, а затем трассирует каждую функцию, вызванную с момента отказа, пока не произойдет возврат обработчика отсутствия страницы. Для определения исходного кода, который необходимо исследовать подробнее, ниже приведены выходные данные скрипта.

---

**Примечание –** В этом разделе мы добавили пояснительный текст к обширным выходным данным кода. Для нахождения связанного текста в выходных данных ищите символ `<----`.

---

## Отладка отсутствия страницы для одиночного процесса при помощи DTrace

1 **Откройте окно терминала.**

2 **Создайте файл** `pagefault.d` **со следующим скриптом:**

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

#pragma D option flowindent
```

```
pagefault:entry
/execname == $$1/
{
    printf("fault occurred on address = %p\n", args[0]);
    self->in = 1;
}

pagefault:return
/self->in == 1/
{
    self->in = 0;
    exit(0);
}

entry
/self->in == 1/
{
}

return
/self->in == 1/
{
}
```

**3 Запустите скрипт для Mozilla.**

---

**Примечание –** В качестве имени исполняемого файла необходимо указать `mozilla-bin`, поскольку `mozilla` не является точным именем. Поскольку включен режим подтверждений (assertion), будут показаны различные вызовы, например, `mutex_owner()`, который используется только с `ASSERT()`. Подтверждения включаются только для ядер отладки.

---

```
# ./pagefault.d mozilla-bin
dtrace: script './pagefault.d' matched 42626 probes
CPU FUNCTION
  0  -> pagefault              fault occurred on address = fb985ea2
```

```
0   | pagefault:entry <-- i86pc/vm/vm_machdep.c or sun4/vm/vm_dep.c
0    -> as_fault   <-- generic address space fault common/vm/vm_as.c
0      -> as_segat
0        -> avl_find  <-- segments are in AVL tree
0          -> as_segcompar          <-- search segments for segment
0          <- as_segcompar          <-- containing fault address
0          -> as_segcompar          <-- common/vm/vm_as.c
0          <- as_segcompar
0          -> as_segcompar
0          <- as_segcompar
0          -> as_segcompar
0          <- as_segcompar
0          -> as_segcompar
0          <- as_segcompar
0          -> as_segcompar
0          <- as_segcompar
0          -> as_segcompar
0          <- as_segcompar
0          -> as_segcompar
0          <- as_segcompar
0        <- avl_find
0      <- as_segat
0      -> segvn_fault<-- segment containing fault is found, (not SEGV)
                       <-- common/vm/seg_vn.c
0        -> hat_probe <-- look for page table entry for page
                      <-- i86pc/vm/hat_i86.c or sfmmu/vm/hat_sfmmu.c
0          -> htable_getpage   <-- page tables are hashed on x86
0            -> htable_getpte  <-- i86pc/vm/htable.c
0              -> htable_lookup
0              <- htable_lookup
0              -> htable_va2entry
0              <- htable_va2entry
0              -> x86pte_get   <-- return a page table entry
0                -> x86pte_access_pagetable
0                  -> hat_kpm_pfn2va
0                  <- hat_kpm_pfn2va
0                <- x86pte_access_pagetable
0                -> x86pte_release_pagetable
0                <- x86pte_release_pagetable
0              <- x86pte_get
```

```
0            <- htable_getpte
0          <- htable_getpage
0          -> htable_release
0          <- htable_release
0        <- hat_probe
0        -> fop_getpage <-- file operation to retrieve page(s)
0          -> ufs_getpage<--file in ufs fs(common/fs/ufs/ufs_vnops.c)
0            -> bmap_has_holes  <-- check for sparse file
0            <- bmap_has_holes
0            -> page_lookup     <-- check for page already in memory
0              -> page_lookup_create  <-- common/vm/vm_page.c
0              <- page_lookup_create  <-- create page if needed
0            <- page_lookup
0            -> ufs_getpage_miss <-- page wasn't in memory
0              -> bmap_read  <-- get block number of page from inode
0                -> bread_common
0                  -> getblk_common
0                  <- getblk_common
0                <- bread_common
0              <- bmap_read
0              -> pvn_read_kluster <-- read pages (common/vm/vm_pvn.c)
0               -> page_create_va  <-- create some pages
0               <- page_create_va
0               -> segvn_kluster
0               <- segvn_kluster
0              <- pvn_read_kluster
0             -> pageio_setup <-- setup page(s) for io common/os/bio.c
0             <- pageio_setup
0             -> lufs_read_strategy <-- logged ufs read
0              -> bdev_strategy <-- read device common/os/driver.c
0               -> cmdkstrategy <-- common disk driver (cmdk(7D))
                                <-- common/io/dktp/disk/cmdk.c
0                -> dadk_strategy  <-- direct attached disk (dad(7D))
                       <-- for ide disks(common/io/dktp/dcdev/dadk.c)
                -- driver sets up dma and starts page in
0               <- dadk_strategy
0              <- cmdkstrategy
0             <- bdev_strategy
0             -> biowait  <-- wait for pagein complete common/os/bio.c
0              -> sema_p  <-- wakeup sema_v from completion interrupt
```

```
0               -> swtch  <-- let someone else run(common/disp/disp.c)
0                -> disp  <-- dispatch to next thread to run
0                <- disp
0                -> resume <-- actual switching occurs here
                           <-- intel/ia32/ml/swtch.s
0                 -> savectx <-- save old context
0                 <- savectx
                  <-- someone else is running here...
0                 -> restorectx  <-- restore context (we're awakened)
0                 <- restorectx
0                <- resume
0               <- swtch
0              <- sema_p
0             <- biowait
0             -> pageio_done <-- undo pageio_setup
0             <- pageio_done
0             -> pvn_plist_init
0             <- pvn_plist_init
0            <- ufs_getpage_miss <-- page is in memory
0           <- ufs_getpage
0         <- fop_getpage
0       -> segvn_faultpage <-- call hat to load pte(s) for page(s)
0        -> hat_memload
0         -> page_pptonum  <-- get page frame number
0         <- page_pptonum
0         -> hati_mkpte    <-- build page table entry
0         <- hati_mkpte
0         -> hati_pte_map  <-- locate entry in page table
0          -> x86_hm_enter
0          <- x86_hm_enter
0          -> hment_prepare
0          <- hment_prepare
0          -> x86pte_set   <-- fill in pte into page table
0            -> x86pte_access_pagetable
0              -> hat_kpm_pfn2va
0              <- hat_kpm_pfn2va
0            <- x86pte_access_pagetable
0            -> x86pte_release_pagetable
0            <- x86pte_release_pagetable
0          <- x86pte_set
```

```
  0          -> hment_assign
  0          <- hment_assign
  0          -> x86_hm_exit
  0          <- x86_hm_exit
  0        <- hati_pte_map
  0       <- hat_memload
  0      <- segvn_faultpage
  0     <- segvn_fault
  0    <- as_fault
  0  <- pagefault

#
```

Следует помнить, что вышеприведенные выходные данные приводятся в сокращенном виде. На высоком уровне при обнаружении отсутствия страницы произошло следующее:

- Для обработки отсутствия страниц вызывается процедура `pagefault()`.
- Процедура `pagefault()` вызывает `as_fault()` для обработки отсутствия страниц в данном адресном пространстве.
- `as_fault()` обходит дерево AVL структур `seg` в поисках сегмента, содержащего адрес, вызвавший ошибку. Если сегмент не обнаруживается, процессу передается сигнал SIGSEGV (нарушение сегментации).
- Если сегмент обнаруживается, вызывается обработчик отказов конкретного сегмента. Для большинства сегментов это `segvn_fault()`.
- `segvn_fault()` выполняет поиск отсутствующей страницы, уже находящейся в памяти. Если страница уже существует (но была освобождена), она "восстанавливается" из списка свободных страниц. Если страница не существует, ее необходимо подкачать. В данном случае страница пока не существует в памяти, поэтому вызывается `ufs_getpage()`.
- `ufs_getpage()` находит номер(а) блока страницы (страниц) в файловой системе путем вызова `bmap_read()`.
- После этого вызывается программа strategy драйвера устройства. Обзор действий, выполняемых программой strategy, см. в `strategy(9E)`.

- Пока читается страница, поток, вызвавший отсутствие страницы, блокируется (т.е. отключается) через вызов `swtch()`. В это время выполняются другие потоки.
- После завершения ввода/вывода подкачки страниц обработчик прерываний дискового драйвера снова активирует заблокированный поток `mozilla-bin`.
- Дисковый драйвер выполняет возврат обратно в `segvn_fault()` через код файловой системы.
- Затем `segvn_fault()` вызывает `segvn_faultpage()`.
- `segvn_faultpage()` вызывает уровень трансляции аппаратных адресов (HAT) для загрузки `()` таблицы страниц (PTE) для данной страницы.
- На данный момент виртуальный адрес, вызвавший отсутствие страницы, уже должен быть сопоставлен с действительной физической страницей. После возврата `pagefault()` инструкция, вызвавшая отсутствие страницы, повторяется и должна на этот раз завершиться успешно.

**4** `mdb` **можно использовать для исследования структур данных ядра и обнаружения страницы физической памяти, вызывавшей ошибку, следующим образом:**

**a. Откройте окно терминала.**

**b. Найдите количество сегментов, используемых mozilla, с помощью** `pmap`**:**

```
# pmap -x ‘pgrep mozilla-bin‘ | wc
     368    2730   23105
#
```

Выходные данные показывают, что есть приблизительно 368 сегментов.

**Примечание –** Поиск сегмента, содержащего адрес отсутствующей страницы, показывает правильный сегмент после 8 сегментов. См. вызовы `as_segcompar` в выходных данных DTrace, приведенных выше. Использование дерева AVL способствует сокращению поиска!

**c. Используйте `mdb` для нахождения сегмента, содержащего адрес отказа.**

**Примечание –** Если требуется пойти дальше, можно использовать следующее: `::log /tmp/logfile` в `mdb`, а затем `!vi /tmp/logfile` для поиска. Также можно просто выполнить `mdb` внутри буфера редактора.

```
# mdb -k
Loading modules: [ unix krtld genunix specfs dtrace
ufs ip sctp usba random fctl s1394
 nca lofs crypto nfs audiosup sppp cpc fcip ptm ipc ]
> ::ps !grep mozilla-bin  <-- find the mozilla-bin process
R   933   919   887   885   100 0x42014000 ffffffff81d6a040 mozilla-bin

> ffffffff81d6a040::print proc_t p_as | ::walk seg | ::print struct seg
   <-- Lots of output has been omitted... -->
{
    s_base = 0xfb800000  <-- the seg we want, fault addr (fb985ea2)
    s_size = 0x561000    <-- greater/equal to base and < base+size
    s_szc = 0
    s_flags = 0
    s_as = 0xffffffff828b61d0
    s_tree = {
        avl_child = [ 0xffffffff82fa7920, 0xffffffff82fa7c80 ]
        avl_pcb = 0xffffffff82fa796d
    }
    s_ops = segvn_ops
    s_data = 0xffffffff82d85070
}
    <-- and lots more output omitted -->
```

```
> ffffffff82d85070::print segvn_data_t  <-- from s_data
{
    lock = {
        _opaque = [ 0 ]
    }
    segp_slock = {
        _opaque = [ 0 ]
    }
    pageprot = 0x1
    prot = 0xd
    maxprot = 0xf
    type = 0x2
    offset = 0
    vp = 0xffffffff82f9e480   <-- points to a vnode_t
    anon_index = 0
    amp = 0   <-- we'll look at anonymous space later
    vpage = 0xffffffff82552000
    cred = 0xffffffff81f95018
    swresv = 0
    advice = 0
    pageadvice = 0x1
    flags = 0x490
    softlockcnt = 0
    policy_info = {
        mem_policy = 0x1
        mem_reserved = 0
    }
}

> ffffffff82f9e480::print vnode_t v_path
v_path = 0xffffffff82f71090
"/usr/sfw/lib/mozilla/components/libgklayout.so"

> fb985ea2-fb800000=K  <-- offset within segment
                185ea2 <-- rounding down gives 185000 (4kpage size)

> ffffffff82f9e480::walk page !wc  <-- walk list of pages on vnode_t
    1236    1236   21012  <-- 1236 pages,(not all are necessarily valid)
```

```
> ffffffff82f9e480::walk page | ::print page_t<-- walk pg list on vnode
    <-- lots of pages omitted in output -->
{
    p_offset = 0x185000  <-- here is matching page
    p_vnode = 0xffffffff82f9e480
    p_selock = 0
    p_selockpad = 0
    p_hash = 0xffffffffae21c00
    p_vpnext = 0xffffffffaca9760
    p_vpprev = 0xffffffffb3467f8
    p_next = 0xffffffffad8f800
    p_prev = 0xffffffffad8f800
    p_lckcnt = 0
    p_cowcnt = 0
    p_cv = {
        _opaque = 0
    }
    p_io_cv = {
        _opaque = 0
    }
    p_iolock_state = 0
    p_szc = 0
    p_fsdata = 0
    p_state = 0
    p_nrm = 0x2
    p_embed = 0x1
    p_index = 0
    p_toxic = 0
    p_mapping = 0xffffffff82d265f0
    p_pagenum = 0xbd62  <-- the page frame number of page
    p_share = 0
    p_sharepad = 0
    p_msresv_1 = 0
    p_mlentry = 0x185
    p_msresv_2 = 0
}

    <-- and lots more output omitted -->

> bd62*1000=K  <-- multiple page frame number time page size (hex)
```

```
                bd62000  <-- here is physical address of page

> bd62000+ea2,10/K  <-- dump 16 64-bit hex values at physical address
0xbd62ea2:      2ccec81ec8b55   e8575653f0e48300 32c3815b00000000
                5d89d46589003ea7 840ff6850c758be0 e445c7000007df
                1216e8000000    dbe850e4458d5650 7d830cc483ffeeea
                791840f00e4     c085e8458904468b 500c498b088b2474
                8b17eb04c483d1ff e8458de05d8bd465 c483ffeeeac8e850
                458b0000074ce904

> bd62000+ea2,10/ai  <-- data looks like code, let's try dumping as code
0xbd62ea2:
0xbd62ea2:      pushq  %rbp
0xbd62ea3:      movl   %esp,%ebp
0xbd62ea5:      subl   $0x2cc,%esp
0xbd62eab:      andl   $0xfffffff0,%esp
0xbd62eae:      pushq  %rbx
0xbd62eaf:      pushq  %rsi
0xbd62eb0:      pushq  %rdi
0xbd62eb1:      call   +0x5     <0xbd62eb6>
0xbd62eb6:      popq   %rbx
0xbd62eb7:      addl   $0x3ea732,%ebx
0xbd62ebd:      movl   %esp,-0x2c(%rbp)
0xbd62ec0:      movl   %ebx,-0x20(%rbp)
0xbd62ec3:      movl   0xc(%rbp),%esi
0xbd62ec6:      testl  %esi,%esi
0xbd62ec8:      je     +0x7e5   <0xbd636ad>
0xbd62ece:      movl   $0x0,-0x1c(%rbp)

> ffffffff81d6a040::context <--change context from kernel to mozilla-bin
debugger context set to proc ffffffff81d6a040, the address of process

> fb985ea2,10/ai <-- and dump from faulting virtual address
0xfb985ea2:
0xfb985ea2:     pushq  %rbp  <-- looks like a match
0xfb985ea3:     movl   %esp,%ebp
0xfb985ea5:     subl   $0x2cc,%esp
0xfb985eab:     andl   $0xfffffff0,%esp
0xfb985eae:     pushq  %rbx
0xfb985eaf:     pushq  %rsi
```

```
0xfb985eb0:     pushq  %rdi
0xfb985eb1:     call   +0x5     <0xfb985eb6>
0xfb985eb6:     popq   %rbx
0xfb985eb7:     addl   $0x3ea732,%ebx
0xfb985ebd:     movl   %esp,-0x2c(%rbp)
0xfb985ec0:     movl   %ebx,-0x20(%rbp)
0xfb985ec3:     movl   0xc(%rbp),%esi
0xfb985ec6:     testl  %esi,%esi
0xfb985ec8:     je     +0x7e5   <0xfb9866ad>
0xfb985ece:     movl   $0x0,-0x1c(%rbp)

> 0::context
debugger context set to kernel

> ffffffff81d6a040::print proc_t p_as <-- get as for mozilla-bin
p_as = 0xffffffff828b61d0

> fb985ea2::vtop -a ffffffff828b61d0  <-- check our work
virtual fb985ea2 mapped to physical bd62ea2  <--physical address matches
```

После нахождения сегмента напечатаем структуру `segvn_data`. В этом сегменте `vnode_t` отображает данные сегмента. В `vnode_t` содержится список страниц, которые "принадлежат" `vnode_t`. Найдем страницу, соответствующую смещению, внутри сегмента. После обнаружения `page_t` мы получаем номер страничного блока. После этого номер страничного блока преобразуется в физический адрес, и исследуются некоторые из данных по этому адресу. Оказывается, что эти данные – код. После этого проверяем физический адрес при помощи команды `vtop` (virtual-to-physical) `mdb`.

**d. Дополнительно: просмотрите таблицы страниц процесса для выяснения способа преобразования виртуального адреса в физический.**

◆ ◆ ◆ МОДУЛЬ 13

# Отладка драйверов при помощи DTrace

## Цели

Этот раздел посвящен использованию DTrace для отладки разрабатываемых драйверов и включает в себя демонстрацию на конкретном примере.

## Портирование драйвера `smbfs` из Linux в ОС Solaris

В этом примере подчеркивается использование возможностей DTrace для разработки драйверов устройств.

Исторически для отладки драйвера устройства требовался вызов таких функций, как `cmn_err()`, для записи диагностической информации в файл `/var/adm/messages`. Эта достаточно неудобная процедура для обнаружения программных ошибок кодирования требует угадывания, перекомпиляции и перезагрузки системы. Разработчики, владеющие ассемблером, могут использовать `adb` и создавать собственные модули в C для `mdb` для диагностики программных ошибок. Однако традиционные подходы к разработке и отладке ядра отнимают весьма много времени.

DTrace позволяет упростить решение некоторых из задач диагностики. Вместо просеивания данных в файле `/var/adm/messages` или страницах вывода `truss`, DTrace позволяет получить информацию только по тем событиям, которые требуется

исследовать. Преимущества DTrace нагляднее всего можно продемонстрировать на нескольких простых примерах.

Сначала создадим шаблон драйвера smbfs, основанный на драйвере nfs от Sun. После успешной компиляции драйвера выясним, может ли этот драйвер успешно загружаться и выгружаться. Вначале скопируем драйвер-прототип в /usr/kernel/fs и попытаемся загрузить его вручную (modload):

```
# modload /usr/kernel/fs/smbfs
    can't load module: Out of memory or no room in system tables
```

Файл /var/adm/messages содержит следующее:

```
genunix: [ID 104096 kern.warning] WARNING: system call missing
from bind file
```

Поиск сообщения об отсутствия системного вызова показывает, что оно находится в функции mod_getsysent() в файле modconf.c, в неудачном вызове mod_getsysnum. Вместо ручного поиска в потоке mod_getsysnum() последовательно по каждому исходному файлу можно использовать простой скрипт DTrace, позволяющий активизировать все события входа и возврата в провайдере fbt (Function Boundary Tracing – трассировка границ функции) после входа в mod_getsynum().

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

    #pragma D option flowindent

    fbt::mod_getsysnum:entry
    /execname == "modload"/
    {
        self->follow = 1;
    }

    fbt::mod_getsysnum:return
    {
        self->follow = 0;
        trace(arg1);
```

```
}

fbt:::entry
/self->follow/
{
}

fbt:::return
/self->follow/
{
    trace(arg1);
}
```

**Примечание –** `trace(arg1)` отображает возвращаемое функцией значение.

Выполнив этот скрипт и запустив команду `modload` в другом окне, получим следующие выходные данные:

```
# ./mod_getsysnum.d
   dtrace: script './mod_getsysnum.d' matched 35750 probes


   CPU FUNCTION
     0  -> mod_getsysnum
     0    -> find_mbind
     0     -> nm_hash
     0     <- nm_hash                                          41
     0     -> strcmp
     0     <- strcmp                                   4294967295
     0     -> strcmp
     0     <- strcmp                                           7
     0    <- find_mbind                                        0
     0  <- mod_getsysnum                                4294967295
```

Таким образом, виновником является либо `find_mbind()`, возвращающая '0', либо `nm_hash()`, возвращающая '41'. Быстрый просмотр `find_mbind()` показывает, что возвращаемое значение 0 указывает на состояние ошибки. Просмотр исходного файла

от find_mbind() в файле /usr/src/uts/common/os/modsubr.c показывает, что нужно искать строку char в хэш-таблице. Воспользуемся DTrace для отображения содержимого строки поиска хэш-таблицы.

Для просмотра содержимого строки поиска добавим к предыдущему скрипту mod_getsysnum.d трассировку strcmp():

```
fbt::strcmp:entry
    {
        printf("name:%s, hash:%s", stringof(arg0),
            stringof(arg1));
    }
```

Ниже приведены результаты следующей попытки загрузки драйвера:

```
# ./mod_getsysnum.d
   dtrace: script './mod_getsysnum.d' matched 35751 probes
   CPU FUNCTION
     0  -> mod_getsysnum
     0    -> find_mbind
     0     -> nm_hash
     0     <- nm_hash                                          41
     0     -> strcmp
     0      | strcmp:entry            name:smbfs,
                      hash:timer_getoverrun
     0     <- strcmp                                  4294967295
     0     -> strcmp
     0      | strcmp:entry            name:smbfs,
                      hash:lwp_sema_post
     0     <- strcmp                                           7
     0    <- find_mbind                                        0
     0  <- mod_getsysnum                                4294967295
```

Таким образом, выполняется поиск smbfs в хэш-таблице, а он в ней отсутствует. Каким образом smbfs попадает в эту хэш-таблицу? Вернемся к find_mbind() и выясним, как переменная хэш-таблицы sb_hashtab передается в функцию nm_hash(), вызывающую ошибку.

Быстрый поиск по исходному коду показывает, что sb_hashtab инициализируется путем вызова read_binding_file(), который принимает в качестве аргументов файл config, хэш-таблицу и указатель на функцию. Еще несколько щелчков в браузере исходного кода раскрывают содержимое файла config, который определен как /etc/name_to_sysnum в файле /usr/src/uts/common/os/modctl.c. Похоже на то, что для драйвера не была включена конфигурационная запись. Добавим следующий код к /etc/name_to_sysnum файлу и выполним перезагрузку.

```
'smbfs            177'
    (read_binding_file() is read once at boot time.)
```

После перезагрузки драйвер загружается успешно.

```
# modload /usr/kernel/fs/smbfs
```

Убедитесь в том, что драйвер загружен, при помощи команды modinfo:

```
# modinfo | grep smbfs
   160 feb21a58  351ac 177      1  smbfs (SMBFS syscall,client,comm)
   160 feb21a58  351ac  24      1  smbfs (network filesystem)
   160 feb21a58  351ac  25      1  smbfs (network filesystem version 2)
   160 feb21a58  351ac  26      1  smbfs (network filesystem version 3)
```

**Примечание –** Следует помнить, что этот драйвер основан на шаблоне nfs, что и объясняет подобные выходные данные.

Удостоверимся, что модуль также можно выгрузить:

```
# modunload -i 160
    can't unload the module: Device busy
```

Наиболее вероятно, что подобное сообщение возникает в связи с возвращаемым значением EBUSY errno. Однако теперь, так как драйвер smbfs – загруженный модуль, возможен доступ ко всем функциям smbfs:

```
# dtrace -l fbt:smbfs:: | wc -l
   1002
```

Это замечательно! Без какого-либо особенного кодирования мы получили 1002 события входа и возврата, содержащихся в драйвере. Эти 1002 метки-манипулятора позволяют отлаживать код без специальной версии драйвера с "инструментованным кодом"! Проконтролируем все вызовы smbfs при вызове modunload с помощью этого простого скрипта DTrace:

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

   #pragma D option flowindent

   fbt:smbfs::entry
   {
   }

   fbt:smbfs::return
   {
       trace(arg1);
   }
```

Похоже на то, что modunload **не** обращается к коду smbfs. Используем DTrace для исследования modunload при помощи этого скрипта:

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

    #pragma D option flowindent

    fbt::modunload:entry
    {
         self->follow = 1;
        trace(execname);
        trace(arg0);
    }

    fbt::modunload:return
    {
```

```
        self->follow = 0;
       trace(arg1);
}

fbt:::entry
/self->follow/
{
}

fbt:::return
/self->follow/
{
     trace(arg1);
}
```

Here's the output of this script:

```
# ./modunload.d
dtrace: script './modunload.d' matched 36695 probes
CPU FUNCTION
  0  -> modunload                modunload             160
  0   | modunload:entry
  0    -> mod_hold_by_id
  0     -> mod_circdep
  0     <- mod_circdep                                    0
  0     -> mod_hold_by_modctl
  0     <- mod_hold_by_modctl                             0
  0    <- mod_hold_by_id                           3602566648
  0    -> moduninstall
  0    <- moduninstall                                  16
  0    -> mod_release_mod
  0     -> mod_release
  0     <- mod_release                             3602566648
  0    <- mod_release_mod                          3602566648
  0  <- modunload                                         16
```

Следует отметить, что возвращаемое значение EBUSY "16" берется из `moduninstall`. Обратимся к исходному коду для

moduninstall. moduninstall возвращает EBUSY в нескольких случаях, так что рассмотрим следующие возможности:

1. if (mp->mod_prim || mp->mod_ref || mp->mod_nenabled != 0) return (EBUSY);
2. if ( detach_driver(mp->mod_modname) != 0 ) return (EBUSY);
3. if ( kobj_lookup(mp->mod_mp, "_fini") == NULL )
4. smbfs_fini()

Невозможно непосредственно оценить все эти вероятности, однако к ним можно применить метод исключения. Воспользуемся следующим скриптом для отображения содержимого различных структур и возвращаемых значений из moduninstall.

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

    #pragma D option flowindent

    fbt::moduninstall:entry
    {
        self->follow = 1;
        printf("mod_prim:%d\n",
            ((struct modctl *)arg0)->mod_prim);
        printf("mod_ref:%d\n",
            ((struct modctl *)arg0)->mod_ref);
        printf("mod_nenabled:%d\n",
            ((struct modctl *)arg0)->mod_nenabled);
        printf("mod_loadflags:%d\n",
            ((struct modctl *)arg0)->mod_loadflags);
    }

    fbt::moduninstall:return
    {
        self->follow = 0;
        trace(arg1);
    }
```

```
    fbt::kobj_lookup:entry
    /self->follow/
    {
    }

    fbt::kobj_lookup:return
    /self->follow/
    {
        trace(arg1);
    }

    fbt::detach_driver:entry
    /self->follow/
    {
    }

    fbt::detach_driver:return
    /self->follow/
    {
         trace(arg1);
    }
```

This script produces the following output:

```
    # ./moduninstall.d
    dtrace: script './moduninstall.d' matched 6 probes
    CPU FUNCTION
      0  -> moduninstall
        mod_prim:0
        mod_ref:0
        mod_nenabled:0
        mod_loadflags:1
      0    -> detach_driver
      0    <- detach_driver                                        0
      0    -> kobj_lookup
      0    <- kobj_lookup                                 4273103456
      0  <- moduninstall                                         16
```

Сравнение этих выходных данных с кодом показывает, что ошибка не связана со значениями структуры mp или возвращаемыми значениями detach_driver() из kobj_lookup(). Таким образом, по методу исключения, это должен быть статус, возвращаемый через вызов status = (*func)();, который вызывает процедуру smbfs _fini(). Вот что содержит процедура smbfs_fini():

```
int _fini(void)
    {
        /* don't allow module to be unloaded */
        return (EBUSY);
    }
```

Изменение возвращаемого значения на "0" и перекомпиляция кода позволяет получить драйвер, который теперь можно загружать и выгружать, т.е. поставленная задача выполнена. В этих примерах использовались исключительно функции провайдера fbt. Следует отметить, что fbt – лишь один из многих провайдеров DTrace.

◆ ◆ ◆ МОДУЛЬ 14

# Наблюдение процессов в зонах с помощью DTrace

## Цели

В этом разделе более подробно поясняются способы наблюдения за процессами в зоне при помощи DTrace.

## Дополнительные ресурсы

- *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones, Sun Microsystems, Inc., 2005*
- *Solaris Containers-Resource Management and Solaris Zones Developer Guide, Sun Microsystems, Inc., 2005*

## Глобальные и неглобальные зоны

После изучения общих вопросов, связанных с отладкой приложений, рассмотрим отладку приложений, работающих в зонах.

В каждой системе OpenSolaris есть глобальная зона `global`. Зона `global` выполняет две функции. Зона `global` является одновременно зоной по умолчанию для системы и зоной, используемой для общесистемного административного управления.

Существует два типа моделей корневой файловой системы неглобальной зоны: "разреженная" (sparse root) и "целостная"

(whole root). Разреженная модель позволяет оптимизировать совместное использование объектов. Целостная модель обеспечивает максимальные возможности по настройке файловых систем.

Для неглобальной зоны устанавливается класс планирования системы. Класс планирования зоны также можно задать с помощью динамических пулов ресурсов. Если зона связана с пулом, свойство `pool.scheduler` которого представляет действительный класс планирования, то процессы, выполняющиеся в этой зоне, работают в этом классе планирования по умолчанию.

Пул ресурса может совместно использоваться несколькими зонами; с другой стороны, к пулу может быть привязана только одна область в целях обеспечения гарантированного обслуживания. По умолчанию всем зонам, включая глобальную зону, присвоена одна (1) доля планировщика долевого распределения. Процент ресурсов ЦП, отведенных зоне, представляет собой отношение долей зоны к общему числу долей для всех зон, привязанных к определенному пулу ресурсов.

Для конфигурирования зоны путем установки различных параметров виртуальной платформы зоны и ее прикладной среды глобальным администратором используется команда `zonecfg`. Затем глобальный администратор устанавливает зону при помощи команды администрирования зон `zoneadm`, которая служит для установки программного обеспечения на уровне пакетов в иерархии файловой системы зоны. Глобальный администратор может зарегистрироваться в установленной зоне с помощью команды `zlogin`. При первой регистрации выполняется внутреннее конфигурирование зоны. После этого используется команда `zoneadm` для начальной загрузки зоны.

# Трассировка процесса, выполняющегося в зоне, при помощи DTrace

Главное внимание в этом примере уделяется наблюдению процессов, выполняющихся в зоне. Для наблюдения процессов в других зонах из зоны `global` можно использовать такие инструменты, как `prstat(1M)`, `ps(1)` и `truss(1)`.

## Аннотация

DTrace может использоваться из зоны `global`; поддерживается переменная `zonename` и поле `pr_zoneid` в `psinfo_t` для использования с провайдером `proc`.

## Трассировка процесса в зоне, при помощи DTrace

1 **Откройте окно терминала.**

2 **Зарегистрируйтесь в зоне** `global`**:**

```
% zlogin
password:
#
```

3 **Подсчитайте число операций ввода/вывода в каждой зоне:**

```
# dtrace -n io:::start{@[zonename] = count()}
```